SZKOŁA JESIENNA PTI

WSPÓŁCZESNE KIERUNKI ROZWOJU INFORMATYKI

Organizowana przez POLSKIE TOWARZYSTWO INFORMATYCZNE
przy współudziale ZAKŁADÓW ELEKTRONICZNYCH „ELWRO"

MRĄGOWO 4—8 XI 1985 r.

SZKOŁA JESIENNA PTI

WSPÓŁCZESNE KIERUNKI ROZWOJU INFORMATYKI

Organizowana przez POLSKIE TOWARZYSTWO INFORMATYCZNE
przy współudziale ZAKŁADÓW ELEKTRONICZNYCH „ELWRO"

MRĄGOWO 4—8 XI 1985 r.

PROGRAM SZKOŁY

1. Prof. Dines Bjørner /Technical University of Denmark/
   Software engineering aspects of VDM

2. Doc. Jacek Błażewicz /Politechnika Poznańska/
   Złożoność obliczeniowa problemów analizy i projektowania systemów komputerowych

3. Doc. Wojciech Cellary /Politechnika Poznańska/
   Rozproszone bazy danych

4. Mgr Jarosław Deminet /Uniwersytet Warszawski/
   Cm* - przykład komputera wieloprocesorowego o strukturze hierarchicznej

5. Dr Jacek Irlik /Uniwersytet Śląski/
   Umowy w zakresie informatyki

6. Dr Michał Jankowski /Uniwersytet Warszawski/
   Grafika komputerowa

7. Doc. Jan Madey /Uniwersytet Warszawski/
   Systemy operacyjne dla mikrokomputerów

8. Doc. Maciej Sysło
   Maszyny i algorytmy współbieżne

SŁOWO WSTĘPNE

Postęp w informatyce uwarunkowany jest nie tylko rozwojem technologii sprzętu i oprogramowania, ale również - i to w coraz większym stopniu - postępem w rozwiązywaniu szeregu problemów podstawowych. Należą do nich między innymi problemy związane z podnoszeniem jakości i niezawodności oprogramowania, z organizowaniem wielkich zbiorów informacji, współdziałaniem procesów i systemów, nowymi formami wymiany informacji na granicy człowiek-komputer itp. Zdaniem wielu specjalistów, wdrożenie rozwiązań tego rodzaju problemów do praktyki przemysłowej jest warunkiem koniecznym dalszego postępu w informatyce. Wyrazem tych opinii jest widoczny od kilku lat wzrost zainteresowania problemami podstawowymi w środowisku praktyków, i to zarówno tych, którzy informatykę tworzą, jak i tych, którzy ją stosują. Wiele firm organizuje kursy dla swoich pracowników poświęcone tematom uznawanym niegdyś za akademickie.

Organizacja spotkań naukowych poświęconych informatyce jest jednym z ważnych zadań statutowych Polskiego Towarzystwa Informatycznego. Kierując się tą myślą, Zarząd Główny PTI pierwszej kadencji podjął na jednym z pierwszych posiedzeń jakie odbyły się po Zjeździe Założycielskim w maju 1981 r. inicjatywę zmierzającą do zorganizowania Szkoły PTI poświęconej współczesnym kierunkom rozwoju informatyki, Niestety, z powodów od Towarzystwa niezależnych do realizacji tego zamierzenia doszło dopiero w październiku 1985.

Niniejsza Jesiennna Szkoła PTI jest drugą z kolei tego typu imprezą naukową organizowaną przez nasze Towarzystwo. Podobnie jak impreza poprzednia poświęcona jest ona przeglądowi wybranych kierunków związanych z rozwojem informatyki i jej zastosowań na świecie. Wszystkie wykłady przygotowane zostały specjalnie na zamówienie organizatorów i pomyślane są tak, aby były zrozumiałymi dla każdego słuchacza z wyższym wykształceniem informatycznym niezależnie od jego specjalności zawodowej.

Przy ustalaniu tematyki szkoły kierowano się zasadą doboru tematyki najaktualniejszej oraz wiążącej się z rozwojem informatyki i jej zastosowań w krajach informatycznie zaawansowanych. Utrzymano również

przyjętą już poprzednio zasadę, aby każdy wykładowca był uznanym specjalistą w reprezentowanej przez siebie dziedzinie, prowadzącym własne badania w danym zakresie i reprezentującym odpowiednio silne środowisko naukowe.

Inowacją tegorocznej Szkoły jest występ wykładowcy zagranicznego profesora Dinesa Bjornera z Politechniki Duńskiej w Lyngby. Jest on wybitnym teoretykiem i praktykiem - a jednocześnie jednym z twórców - tzw. nowej metody wiedeńskiej /VDM/ specyfikowania i projektowania oprogramowania. Ponieważ wykład prof. D. Bjornera wygłodzony zostanie w języku angielskim w takim też języku publikujemy tekst tego wykładu. W tej sytuacji wykład ten być może nie będzie dostępny dla wszystkich uczestników Szkoły. Z drugiej jednak strony uczestnicy znający język angielski będą mieli okazję zaznajomić się z atrakcyjnym tematem u samego źródła oraz posłuchać jak robi się informatykę, i jak się o niej mówi, poza granicami naszego kraju. Jeżeli eksperyment z wykładowcą obcojęzycznym oceniony zostanie przez słuchaczy pozytywnie, podjęte zostaną starania aby mógł on być kontynuowany. Oczywiście podobnie jak w przypadku poprzedniej szkoły Organizatorzy oczekują na wszelkie uwagi dotyczące tej i następnych szkół jakie uczestnicy zechcą im przekazać.

Warszawa, lipiec 1985 Andrzej Blikle

SOFTWARE ENGINEERING ASPECTS OF *VDM*

the Vienna Development Method

Dines Bjørner

Department of Computer Science
Technical University of Denmark
DK-2800 Lyngby, Denmark

Søren Prehn

Danish Datamatics Centre
Lundtoftevej 1C
DK-2800 Lyngby, Denmark

Ole N.Oest
DDC International
Lundtoftevej 1A
DK-2800 Lyngby, Denmark

ABSTRACT

The *VDM* principles of Software Development are surveyed, their application to diverse projects outlined, and the practical software engineering aspects of managing and mechanically supporting *VDM* developments are overviewed.

The intended audience of this paper is fourfold: computing scientists and educators, software engineers (i.e. programmers) and their managers. Despite this the paper is rather technical, and requires some amount of mathematical maturity from its readers.

The aim of the paper is to acquaint the reader with a formal programming method, to increase the awareness of what formal skills are believed required in software development, and to outline consequences in the areas of research, education, management and tools.

CONTENTS

0. Background 3

1. Overview of the *VDM* 4

1.1 Abstract Software Specification 4
1.2 Concrete Realization 6
1.3 Correctness Criteria and Proof 9
1.4 Summary 10

2. Application of *VDM* to Programming Languages ... 10

2.1 *VDM* Definitions of Programming Languages .. 11
2.1.1 Form of and Approach to Programming Language Definitions 11
2.1.2 Static Semantics 11
2.1.3 Dynamic Sequential Semantics 13
2.1.4 Dynamic Parallel Semantics 17

2.2 Structure of *VDM* Compiler Developments 19
2.2.1 Semantic Analysis 20
2.2.2 Macro Expansion 24
2.2.3 Virtual Machine & Compiling Algorithm 27

3. Research Requirements 31

3.1 Formal Description of Programming Concepts.. 32
3.2 Programming Methodology..... 32

4. *VDM* Support Tools 33

4.1 Document Processors..... 33
4.2 Computer Processors..... 33
4.3 Transformation Processors..... 34
4.4 Correctness Processors..... 34
4.5 Support System Design Requirements..... 35

5. Management Aspects of *VDM* 35

6. Education and Training Aspects of *VDM* 39

7. The Sociology of *VDM* 39

VDM Bibliography 41

General and Background Bibliography 45

Last Page..... 49

0. BACKGROUND

The *VDM* is a formal semantics based method for developing correct programs. *VDM* borrows from a wide range of formal, mathematics based techniques developed during the 1960s and the 1970s in Europe and the U.S. *VDM* in its early form was first devised by a group at the IBM Vienna Laboratory, Austria, around 1973.

The distinguishing, technical features of *VDM* will be outlined in section 1 and briefly illustrated in section 2.

VDM is well-documented in the literature: 5 books and nu numerous published papers and many technical reports. We refer to the Bibliography section of this paper.

VDM has, up till now, been applied in the areas of Programming Languages, Data Bases, Operating Systems, Office Automation Systems, and Special Application Systems. In the following we list "names" of completed or ongoing projects using *VDM* (in section 2 we outline some specific techniques for applying *VDM* to the area of programming languages and their processors; for other areas, the reader is refered to the bibliography):

Formal, principally denotational semantics definitions for the following programming languages: *PL/I, ALGOL 60, Standard Pascal, Pascal/R, Pascal Plus, Modula-2, Edison, CHILL Ada, PROLOG* and *VAL* have been constructed in the *VDM* style. Full language compilers for *Pascal Plus, Edison, CHILL* and *Ada* have been, or are being, developed. All, or main, aspects of the following (kind of) Data Base Systems have been likewise formally defined: the *PL/I* programmers interface to full, concurrent *System/R*; the *SQL* and *DSL-Alpha* predicate calculus and the algebra relational data model query languages; the hierarchical and the network data base models: their data structures and data manipulation languages (DMLs); and aspects of *IMS & (System 2000)* and *CODASYL/DBTG*. Important aspects of Operating Systems have have been experimentally defined, including parallellism models, UNIX-like concepts: pipes, shells, etc. The Toronto University CSG Forms Flow oriented, distributed work station Office Automation System have been given a formal model, with realization-work now commencing. Finally, a generic, structured application system, *LUCAS*, have been designed: formally defined and prototype-implemented.

The above projects have carried out in Europe: Austria, Denmark, Federal Republic of Germany, England and now also Italy. Several of the projects, notably PL/I, CHILL, Ada and Office Automation Systems have been, or are carried out in industry.

It is based on experience gained in these many projects that we can now comment on the following software engineering aspects: how do we manage development projects involving the concurrent interaction and then sequenced action between many *VDM*ers, what are the educational and training skills required (e.g.: can professional, seasoned programmers be re-educated and what does that require); can we, and if so, how do we estimate development resources; to what degree of conviction can we produce (what is believed, or can be proven to be) correct software; and finally: what are the development tools required to support VDM-based developments?

The structure of the paper follows partly from the contents listing: section 1 summarizes main features of *VDM* Software Specification and its Stages of Implementations, while section 2 outlines the application of the *VDM* "doctrines" to programming language definition and compilation. Section 3 puts focus on methodological gaps requiring further research in order for *VDM* to mature into a more widely usable method. Sections 5-6 then touches upon a number of educational, training and management aspects, while section 4 summarizes a wish-list of automated and mechanical aid development support tools. Finally, in section 7, we record some of our experience in propagating *VDM*.

1. OVERVIEW OF THE *VDM*

Presently the *VDM* applies itself only to the software (1) specification, and (2) realization phases of software development. We shall, however, make some remarks on the (0) requirements analysis phase, preceding specification; and on the (3) installation, "maintenance" and usage phases, following realization:

(0) Requirements Analysis and Definitions
(1) Specifications
(2) Realization
(3) Installation, "Maintenance and Use"

The literature, also the one on *VDM* bibliographed here, abounds with motivations for why one emphasizes abstraction in specification, etc.. We shall therefore concentrate on the means used.

1.1 Abstract Software Specification

VDM-based specifications are basically denotational. *VDM* representationally abstracts the Syntactic Domains of objects input to and output from the software to be specified, and the Semantic Domains of the meanings of these objects, and the objects by means of which meanings are composed. *VDM* operationally abstracts the so-called elaboration functions which, when applied to syntactic objects, yields semantic ones. Thus a *VDM*-based specification is a model of the software being architechted. This is in contrast to e.g. Algebraic Specifications. In a model one is already "interpreting" certain objects in specific ways. Stack objects e.g., are modelled in terms of available Domain types, e.g. sequences or recursively defined trees. Operations on stacks, hence, are expressed in terms of "available" primitive operations on such sequences, respectively trees. This is in contrast to algebraic specifications which have no "predefined" (i.e "available") Domain types, and hence no corresponding primitive operations.

Models

A *VDM*-based specification, i.e. a model, consists of basically three components: specification of the Syntactic Domains, the Semantic Domains, and the Elaboration Functions.

Semantic Domains -- an example

In an operating system directory resources are hierarchically "ordered":

```
1.  OS0   :: DIR0   FS
2.  DIR0  =  Rnm ⇉ RES0
3.  RES0  =  Fnm | DIR0
4.  FS    =  Fnm ⇉ FILE
```

where OS_0, DIR_0, FS, Rnm, RES, Fnm, and $FILE$ are mnemonics for operating system (state), directory, file system, resource name, resource, file name and file; the suffix "$_0$" hints at the initial, most abstract model level.

Syntactic Domains -- an example

Commands are defined for retrieving, inserting, ..., resources. Resources are designated by sequences of resource names (hinting at the hierarchical ordering):

```
5.  Cmd   =  Get | Put | ...
6.  Get   :: Rnm+
7.  Put   :: Rnm+  RES0
```

Elaboration Functions -- an example

```
8.  type:  V0-Get: Get → (OS0 ⇝ (DIR0 | FILE))
9.  type:  I0-Put: Put → (OS0 ⇝ OS0)

8.0 V0-Get(mk-Get(rl))(mk-OS0(dir,fs)) ≜
 .1    cases rl:
 .2    (<r>          → (let res0 = dir(r) in
 .3                       is-DIR0(res0) → res0,
 .4                       T             → fs(res0)),
 .5     <r> ^ rl'    → V0-Get(mk-Get(rl'))(mk-OS0(dir(r),fs)))
```

i.e. lists of resource names (.5) designates (here formulated by recursion) sub-hierarchies, finally (.2) yielding either (.3) a "sub-directory" or (.4) a file.

```
9.0 I0-Put(mk-Put(rl,res))(mk-OS0(dir,fs)) ≜
 .1    cases rl:
 .2    (<r>          → mk-OS0(dir ∪ [r ↦ res],fs),
 .3     <r> ^ rl'    → (let put' = mk-Put(rl',res),
 .4                          os'  = mk-OS(dir(r),fs)          in
 .5                     let mk-OS0(dir',) = I0-Put(put')(os') in
 .6                     mk-OS0(dir + [r ↦ dir'],fs))
```

where, as before, (.3) resource name lists designates subhierarchies, here to be (.6) modified, and finally (.2) the resource of the command is entered at the designated subdirectory.

Data Structure Invariants

When defining Semantic Domains we sometimes cannot conveniently avoid defining too much. In our example we could require, e.g. that all files of the file system be named by at least one resource name, and that all file names in the directory name existing file system files:

```
10.0 inv-OS0(mk-OS0(dir,fs)) ≜
  .1   filenames(dir) = domfs
```

```
11.0 filenames(dir) Δ
  .1   {f | fεrngdir ∧ is-Fnm(f)} ∪
  .2   union {filenames(dir') | dir'εrngdir ∧ is-DIR0(dir')}

  .3 type: DIR0 → Fnm-set
```

Input Well-formedness

Similarly for syntactic objects. In our example we might have required that if a *Put* command resource was a directory that it name only file names:

```
12.0 is-wf-Cmd(c) Δ
  .1    cases c:
  .2    (mk-Put(,res)  →  is-DIR0(res) ⊃
  .3                        (∀fεrngres)(is-Fnm(f)),
  .4     mk-Get(...)   →  ...,
  .5     ...)
```

Pre-Conditions

When "applying" a syntactic object meaning to an actual semantic object it is often required that the syntactic object be well-defined in the context of that semantic object ("state"). In our example, resource name lists of *Get* commands must designate a path in the hierarchical directory:

```
13.0 pre-Get(mk-Get(rl))(mk-OS0(dir,)) Δ
  .1    cases rl:
  .2    (<r>          →  (rεdomdir),
  .3     <r> ^ rl'    →  ((rεdomdir)
  .4                      ∧ pre-Get(mk-Get(rl'))(mk-OS0(dir(r),))))
```

1.2 Concrete Realization

The idea of a specification is, amongst others, to serve as a basis for realization.

In *VDM* realization usually takes place in stages, each stage transforming either primarily the previous stages' abstract objects into more concrete objects, or primarily the previous stages' abstract elaboration (and other) functions into more algorithmic ones.

Whichever way we choose, the one entails the other: deciding upon a stage primarily being one of object transformation usually implies a (secondary) operation transformation -- and vice-versa.

(Primary) Object Transformations -- an example

The directories of our example constituted a recursively defined data type. Most programming languages usually do not offer such a facility directly. We therefore choose to realize directory objects as pointer based structures "allocated" in some storage together with a "root" pointer:

```
14. OS1   ::  STG  FS
15. STG   ::  Ptr  (Ptr m→ DIR1)
16. DIR1  =   Rnm m→ RES1
17. RES1  =   Fnm | Ptr
```

Three concerns now arise: (1) that objects of the new, concrete Domains be invariant, (2) that they abstract (or retrieve) to the more abstract objects of the previous stage, and (3) that for each invariant abstract object of the previous stage there be at least one invariant concrete object (of this stage) that abstracts to it. These concerns are expressed in terms of three functions: (1) as before, the invariant predicate, here: $inv\text{-}OS_1$, (2) an abstraction (or retreive) function, here: $retr\text{-}OS_0$, and (3) an adequacy predicate which we do not exemplify here (for a discussion of adequacy, see [Jones 80a]).

Invariance

Usually invariant predicates of subsequent stages have three parts: (1) one which expresses the invariance requirements inherited from the invariant predicate of the previous level, here $inv\text{-}OS_0$; (2) another which expresses invariance of "added" minimum properties of the concrete objects, here: that the pointers combine to form a hierarchy, and not, e.g. some cyclic or collapsed structure), and lastly (3) a third which expresses possible special (maximum, desirable) properties of the concrete objects, e.g. such added for purposes of efficiency of operations.

```
18.0 inv-OS1(mk-OS1(mk-STG(p,m),fs)) ≜
  .1     (wfPtrs(p,m)
  .2     ∧inv-OS0(retr-OS0(mk-OS1(mk-STG(p,m),fs))))
```

The *wfPtrs* function checks aspect (2) above, where line (18.2) checks aspect (1); aspect (3) is not illustrated. The *wfPtrs* predicate analyzes the pointer structure, i.e. whether the *Pointers* are *well-formed*. Well-formedness means that no two directories designate the same pointers, and that all pointers of the $(Ptr \overset{m}{\to} DIR_1)$ *map* are designated either by the "root" pointer or by directories:

```
19.0 wfPtrs(p,m) ≜
  .1     (let ptrs = union {rngdir1 | dir1 ε rngm} \ Fnm in
  .2      (p ε ptrs)
  .3     ∧(domm = ptrs ∪ {p})
  .4     ∧(∀p',p" ε domm)
  .5        ((p' ≠ p") ⊃ (((domm(p')) \ Fnm) ∩
  .6                      ((domm(p")) \ Fnm)) = {}) ))
  .7 type: Ptr (Ptr m→ DIR1) → BOOL
```

Abstraction/Retrieval

```
20.0 retr-OS0(mk-OS1(mk-STG(p,m),fs)) ≜
  .1    mk-OS0(retr-DIR0(p,m),fs)

  .2 type: OS1 ~→ OS0
  .3 pre:  wfPtrs(p,m)

21.0 retr-DIR0(p,m) ≜
  .1    (let dir1 = m(p) in
  .2     [r ↦ res0 | r ε domdir1 ∧
  .3                 (let res1 = dir1(r) in
  .4                  (is-Fnm(res1) → (res0=res1),
  .5                   is-Ptr(res1) → (res0=retr-DIR0(res1,m))))])
  .6 type:  Ptr (Ptr m→ DIR1) ~→ DIR0
  .7 pre:   wfPtrs(p,m)
```

(The "structure" of the map construction expression (21.2.5) "follows" from the two "structures" of the "input" Domain of *m*: $(Ptr \overrightarrow{m} DIR_1)$ and the DIR_0 "output" Domain.)

(Secondary) Operation Transformation -- an example

Having now stated a stage of concretization of the Semantic Domains, we now have to re-state the elaboration functions (8) and (9) above; here we show (only) *V-Get:*

23. *type:* V_1-*Get:* *Get* → $(OS_1 \overrightarrow{\sim} (Ptr \mid FILE))$

That is: we "equate" *Ptr* with DIR_0:

```
23.0 V1-Get(mk-Get(rl))(mk-OS1(mk-STG(p,m),fs)) Δ
  .1   cases rl:
  .2   (<r>       → (let res1 = (m(p))(r) in
  .3                  (is-Ptr(res1) → res1,
  .4                   T            → fs(res1))),
  .5    <r> ^ rl' → V1-Get(mk-Get(rl'))
  .6                      (mk-OS1(mk-STG((m(p))(r),m),fs)))
```

Given OS_0, V_0-*Get* and OS_1, the transcription of V_0-*Get* into V_1-*Get* can be made automatically.

(Primary) Operation and (secondary) object transformations

In preparation for a primary operation transformation we perform a secondary object transformation. The example further illustrates *VDM*s ease of transition from applicative to imperative specifications.

The three OS_1 components are to be kept, now, in globally declared variables (initialized, for some particular state

$$mk\text{-}OS_1(mk\text{-}STG(p,m),fs)):$$

```
24. dcl p  := p  type Ptr;
25. dcl m  := m  type (Ptr m→ DIR1);
26. dcl fs := fs type FS;
```

that is, we introduce into the model a global state Σ:

$$\begin{array}{lll} 27.0 & \Sigma = OS_2 = (p \overrightarrow{m} Ptr) & \cup \\ .1 & \quad\quad\;\; (m \overrightarrow{m} (Ptr \overrightarrow{m} DIR_1)) & \cup \\ .2 & \quad\quad\;\; (fs \overrightarrow{m} FS) & \end{array}$$

with:

28. $inv\text{-}OS_2() \triangleq inv\text{-}OS_1(retr\text{-}OS_1())$

and

29. $retr\text{-}OS_1() \triangleq mk\text{-}OS_1(mk\text{-}STG(\underline{c}p, \underline{c}m), \underline{c}fs)$

being rather straightforward consequences ($\underline{c}$: contents of declared variable). Similarly we could re-express V_1-*Get* imperatively:

```
30.  type: rV2-Get:  Get → (Ptr → (Σ → (Ptr | FILE)))

30.0 rV2-Get(mk-Get(rl))(p) Δ
  .1    cases rl:
  .2    (<r>       → (def res1 : ((cm)(p))(r);
  .3                    (is-Ptr(res1) → return(res1),
  .4                     T            → (cfs)(res1))),
  .5     <r>^ rl' → rV2-Get(mk-Get(rl'))(((cm)(p))(r)))
```

to be initially invoked with $p = cp$. So far so well. Now follows the desired operation transformation. In preparation, we first re-express the above function (30.), changing only the syntactic form of its control structure:

```
31.0 rV3-Get(mk-Get(rl))(p) Δ
  .1    if lenrl = 1
  .2      then (def res1 : ((cm)(p))(hdrl);
  .3              (is-Ptr(res1) → return(res1),
  .4               T            → (cfs)(res1))),
  .5      else rV3-Get(mk-Get(tlrl))(((cm)(p))(hdrl)))
```

This function evidently fits the following recursion schema:

```
32.  F(x,y) Δ if a then b else F(u,v)
```

where:

x ~ *rl* , y ~ *p* , a ~ *lenrl = 1* , b ~ lines .2-.4,
u ~ *tlrl* , v ~ *((cm)(p))(hdrl)*

Such a recursion schema (32.) can always [Burstall 77] be transformed into the iterative schema (i.e. the function body can be replaced by):

```
33.  while ¬a do (xsave := u; y := v; x := cxsave); b
```

Substitution yields:

```
34.0 (rl  := rl;
  .1  p   := p;
  .2  while len crl > 1 do (rlsave := tl crl;
  .3                        p      := ((cm)(cp))(hd crl);
  .4                        rl     := crlsave );
  .5  (def res1 : ((cm)(cp))(hd crl);
  .6   (is-Ptr(res1) → return(res1),
  .7    T            → (cfs)(res1))))
```

as the new body; the reader is invited to perform the last bit of trivial simplification in avoiding the "save" variable.

1.3 Correctness Criteria and Proof

Correctness of an implementation can be formulated in terms of "commuting diagrams": "running" any command, *g*, on any concrete state, os_1, retrievable to some, more abstract "state", os_0, shall yield a result which is retrievable, i.e. "corresponds" to the result of running the same command on the abstract "state":

```
35.1 (∀g∈Get)
  .2   (∀os_0∈OS_0)(inv-OS_0(os_0))
  .3     (∀os_1∈OS_1)(inv-OS_1(os_1))
  .4       (os_0 = retr-OS_0(os_1))
  .5         ⊃ (V_0-Get(g)(os_0) =
  .6             retr-DIRFILE(V_1-Get(g)(os_1),os_1))
```

where we define correspondance as:

```
36.  type: retr-DIRFILE: (Ptr | FILE) → (OS_1 → (DIR_0 | FILE))

36.0 retr-DIRFILE(pf)(mk-OS_1(mk-STG(,m),)) Δ
  .1   (is-FILE(pf) → pf,
  .2    T           → retr-DIR_0(pf,m))
```

With this correspondence, (35) may be proved to hold, by induction on the length of the *rl* component of *g*. The change from the applicative model of equations (14.-23.) to the imperative model of (24.-31.) is rather trivial and its correctness is easily seen. The third step of development shown above: from the recursive formulation of rV_3-*Get* in (31.) to the iterative of (34.) is correct by virtue of the correctness of the transformation rules (32. → 33.). For a more detailed discussion of correctness theorems and proof techniques, see [Jones 80a].

1.4 Summary

We have briefly illustrated various techniques of (*VDM*) specifications and implementations: representational and operational abstraction, applicative versus imperative "programming", recusive versus iterative controls, stepwise transformations of specifications into coded implementations, object- and operation transformations, and finally correctness theorems.

2. APPLICATION OF *VDM* TO PROGRAMMING LANGUAGES

By the above section title we mean: the use of *VDM* in connection with (1) The Formal Definition of the Semantics of Programming Languages, and (2) the Development of Compilers and Run-Time Systems for Programming Languages.

As noted in the Background section, *VDM* has been applied in both connections, and quite extensively.

To apply *VDM* to the definition of programming languages (PLs) is not to be confused with applying it in the development of compilers. Most computing scientists, and especially software engineers, fail to see the distinction. In the former we define something independent of any implementation, in order to establish a "legal" contract between a whole army of PL users and many diverse groups of PL (ie. compiler) implementors. Of course, the previous remarks apply equally well in most other software areas (Data Bases, Operating Systems, etc.).

But to define a language properly and to develop correct compilers for it is vastly more important than to do likewise for e.g. data bases, operating systems and especially application programs -- and

for two reasons. First and foremost: all our programs (for construction of e.g. data base management systems, operating systems, commercial, technical and scientific application programs) must pass through a compiler! Secondly: most ordinary, end-user program-packages are expected to be far more portable than systems to support these! It is therefore no wonder that specification and rigorous development methods were first developed in connection with programming languages and their compilers.

2.1 *VDM* Definitions of Programming Languages

We recall that there now exist complete, formal denotational definitions in the *VDM* style for the following PLs: *PL/I, ALGOL 60*, Standard *Pascal, Pascal/R, Pascal Plus, Modula-2, Edison, CHILL, Ada, PROLOG, VAL, C-code, A-code, P-code* and formal *VDM*-definitions also of *FORTRAN* and *BASIC*. No other method has, to our knowledge, been so thoroughly tested, so widely applied, to such complicated lanlanguages as several of the above, and to languages embodying process notions.

Either of two "extreme" situations may arise: either one is defining formally a PL which somebody else has already, invariably always informally and always incompletely and usually inconsistently, "defined"; or one is in that lucky situation of defining formally a language which one is also architecting oneself! In the case of *VDM* both situations have been thoroughly experienced, and also, in the case of CHILL, a situation where the formal definition work in some measure influenced the language design. The *VDM* formal definition of CHILL is now part of CCITTs internationally binding standard for CHILL -- supplementing a commendably precise informal definition. The frustrations experienced in otherwise trying to "shoot at the moving" or incomplete "targets" of existing or developing informal accounts of e.g. standard *Pascal, Pascal Plus, Modula-2, CHILL* and *Ada* should not be under-estimated or taken lightly.

2.1.1 Form of and Approach to PL Definitions

For languages like *Modula-2, Edison, Pascal Plus, CHILL* and *Ada* definitions in the VDM-style consists of three definition parts:

Static Semantics
Dynamic Sequential Semantics
Dynamic Parallel Semantics

Each of the three parts consists of three definition subparts, defining:

Syntactic Domains
Semantic Domains; Static, resp. Dynamic
Semantic Functions; Static, resp. Dynamic

We now illustrate these notions on a running example.

2.1.2 Static Semantics

-- Syntactic Domains

The Syntactic Domains of a modular language are programs consisting of one (main) and an unordered collection of (sub) modules. All modules contain definition parts. Main modules in addition contain a statement list. Definition parts consist of unordered collections of variable and/or procedure definitions. Definitions are either "local", "imported" or "exported". Variable definitions consist in this ("un-typed") language of just variable identifiers. Local and exported procedure definitions consist of unique procedure identifiers, parameter lists and bodies. Imported procedure definitions consist of just procedure identifiers. (In the sequel, we concentrate on modelling the modularity of the example language; for the moment we therefore leave unspecified statements, procedure parameter lists and procedure bodies.)

```
1.  Prgr        ::  Main      Sub-set
2.  Main        ::  Defs      s-sl:Stmt+
3.  Sub         =   Defs
4.  Defs        ::  Vars      Procs
5.  Vars        ::  s-expvs:Vid-set    s-impvs:Vid-set
                    s-locvs:Vid-set
6.  Procs       ::  s-expps:Procm      s-impps:Pid-set
                    s-locps:Procm
7.  Procm       =   Pid →m Proc
8.  Stmt, Proc  =   ...
```

-- Semantic Functions

In this simple example we omit explicating the static semantics of statements, procedure parameter lists and procedure bodies. We only focus on the static semantics of modularity. As a consequence of this we do not illustrate the notion of Static Semantics Domains. (The principle has been extensively covered in various parers in [Bjørner 78a, Bjørner 82a] and extensively illustrated in [Bjørner 80a, Haff 80a].) Now back to the example: Imported variable and procedure identifiers must be such identifiers exported from exactly one other program module, and, in general, any identifier must be exported by at most a single module. And, within a module, there must be no overlap between the identifiers of the various categories of variables and procedures:

```
9.0 is-wf-Prgr(mk-Prgr(mk-Main(dp,),sms)) ≜
 .1     ((dp¬ εsms)
 .2     ∧(∀dp'εsms∪{dp})
 .3         (is-wf-Defs(dp')(({dp}∪sms)\{dp'})
 .4 type:  Prgr → BOOL


10.0 is-wf-Defs(mk-Defs(vp,pp))(dps) ≜
 .1     is-wf-Names(vp,pp)
 .2    ∧is-wf-Vars(vp)(dps)
 .3    ∧is-wf-Procs(pp)(dps)
 .4 type: Defs ~→ (Defs-set → BOOL)

11.0 is-wf-Names(mk-Vars(xvs,ivs,lvs),
                 mk-Procs(xpds,ips,lpds))      ≜
 .1     is-disjoint(<xvs,ivs,lvs,domxpds,ips,domlpds>)
 .2 type: Vars  Procs → BOOL
```

```
12.0 is-disjoint(lids) Δ
  .1   (∀i,j∈inds lids)(i≠j ⊃ (lids[i] ∩ lids[j] = {}))
  .2 type: (Vid-set | Pid-set)* → BOOL

13.0 is-wf-Vars(mk-Vars(xvs,ivs,lvs))(dps) Δ
  .1    (∀iv∈ivs)(∃! dp∈dps)(iv∈s-expvs(s-Vars(dp)))
  .2  ∧(∀ev∈xvs)(¬∃dp∈dps)(ev∈s-expvs(s-Vars(dp)) ∨
  .3                          ev∈dom s-expps(s-Proc(dp)))
  .4 type: Varpart → (Defpart-set → BOOL)

14.0 is-wf-Proc(mk-Proc(epds,ips,lpds))(dps) Δ
  .1    (∀ip∈ips)(∃! dp∈dps)(ip∈dom s-expps(s-Proc(dp)))
  .2  ∧(∀ep∈domepds)(¬∃dp∈dps)(ep∈s-expvs(s-Vars(dp)) ∨
  .3                          ep∈dom s-expps(s-Proc(dp)))
  .4 type: Proc → (Defs-set → BOOL)
```

One could claim, contrary to what was stated above, that the (10., 13.,14.) formulae do exploit a static semantics domain, namely the *Defs-set* corresponding to the *(dps)* argument. This, however, is a mere coincidence, caused by the stylistic convenience of decomposing *is-wf-Prgr* into several sub-functions.

2.1.3 Dynamic Sequential Semantics

Variable identifiers denote locations, and procedure identifiers denote functions. Denotations are Semantic Domain objects and are recorded in environments. Storages are likewise semantic objects:

-- Semantic Domains -- first approximation

```
15.  ENV  =  (Vid ⇀m LOC) ∪ (Pid ⇀m FCT)
16.  STG  =  LOC ⇀m VAL
17.  FCT  =  VAL* → (Σ → Σ [VAL])
18.  VAL  =  INTG | BOOL | LOC | FCT ...
19.  Σ    =  Stg ⇀m STG
```

Here, we have foreseen, and defined, (19.) a global state for the semantic functions; i.e. we model storage using an imperative formulation. Accordingly, "side effects" of procedures will be modelled as transformations (17: Σ → Σ) on the global state of the model. The choice of an imperative modelling technique is, however, only dictated by convenience, not by necessity.

-- Semantic Functions, Informal Exposé

The statement list of the main module is to be interpreted in an environment, ρmm, which, besides its own local and exported variables and procedures also must record the denotations of imported variables and procedures. To construct their denotation the "total" environment, ρ, of all exported such is initially required. The contributions, ρm and ρsms, to the total ρ come from the exportations of the main, respectively all the sub-modules.

First we "compute" variable locations; then procedure denotations. The reason for this "split" is the following. In computing locations we simultaneously allocate new such, i.e. perform side-effects. In computing procedure denotations we need to know the denotations of all other procedures which can potentially be mutually

recursively invoked. But no new allocations are effected. Both "computations" are recursively defined, but only the latter is genuinely recursive in recursively using the environment it constructs. It turns out [Mosses 81a, Milne 76a] that if we combined the variable location computation into the set of recursive definitions, then the totally undefined environment would be their minimal fix-point solution -- due to the side-effect aspects.

-- Syntactic Domains

Since we need, therefore, to allocate all variables "globally", ie. also local ones, with possibly identically named local variables in distinct modules, we need to make all variable identifiers distinct.

The non-distinctness is, of course, a static feature offered by the modular language; but it has no consequence for the dynamic semantics. So, as the original syntactic domains (1.,5.) are not appropriate, and as the static requirements has been defined in *is-wf-Prgr*, we choose to use the following syntactic domains, where all submodules are uniquely named, and all imported variables are associated with their origin module (name):

```
20.  Prgr'    :: Main (Mnm ⇉ Sub)
21.  Varpart' :: s-expvs:Vid-set
                 s-impvs:(Mnm | MAIN ⇉ Vid-set)
                 s-locvs:Vid-set
```

-- and otherwise as before. As we are now trying to express the dynamic semantics of programs belonging to the original domains in terms of programs of these new domains, we need an association from original to new domains (note: all "new" objects and domains are primed):

```
22.0 rmk-Prgr(mk-Prgr(mm,sms)) ≜
  .1 (let mnms ⊂ Mnm be s:t: card mnms = card sms   in
  .2  let smsm = name-Modules(sms)(mnms)            in
  .3  let sms' = rmk-Modules(sms)(smsm)             in
  .4  let mk-Main(mk-Defs(vp,pp),sl) = mm           in
  .5  let vp'  = rmk-Vars(vp)(smsm)                 in
  .6  mk-Prgr'(mk-Mainmod'(mk-Defs(vp',pp),sl),sms'))
  .7 type:  Prgr ⇉ Prgr'
  .8 pre:   is-wf-Prgr(mk-Prgr(mm,sms))

23.0 name-modules(sms)(mnms) ≜
  .1 (sms={}→[],
  .2   T   → (let sm ε sms,
  .3              mnm ε mnms in
  .4          [mnm↦sm] ∪name-modules(sms\{sm})(mnms\{mnm})))
  .5 type: Submod-set ⇉ (Mnm-set → (Mnm ⇉ Submod))
  .6 pre:  card sms ≤ card mnms

24.0 rmk-Modules(sms)(smsm) ≜
  .1 (sms ={}→[],
  .2   T   → (let sm ε sms                                  in
  .3          let mnm ε dom smsm be s:t: smsm(mnm) = sm     in
  .4          let mk-Defs(vp,pp) = sm                       in
  .5          let sm' = mk-Defs'(rmk-Vars(vp)(smsm),pp)     in
  .6          [mnm↦sm']
  .7          ∪ rmk-Modules(sms\{sm})(smsm)))
```

```
.8 type: Sub-set ≃ ((Mnm ⇀m Sub) → (Mnm ⇀m Sub'))
.9 pre: sms ⊆ rng smsm

25.0 rmk-Vars(mk-Vars(evs,lvs,lvs))(smsm) ≜
.1 (let ivm' = [iv↦mnm | iv∈ivs ∧
.2                       mnm = ((∃mnm'∈dom smsm)
.3                              (iv∈s-expr s(s-Vars(smsm(mnm')))
.4                                 → mnm',
.5                               T → MAIN)]                    in
.6  mk-Vars'(evs,ivm',lvs))

.7 type: Vars → ((Mnm ⇀m Sub) → Vars')
```

As these functions are far from being trivial, and as they do not directly "speak about" interesting properties, some further justification is required:

In many "classical" approaches to formal definition of PLs, such rebuild functions are not found (e.g. [Henhapl 78a]); in all of these classical approaches, the object languages all have a very close connection between "static concepts" and "dynamic concepts". However, more recent language designs have introduced a large number of rather involved static concepts (as well as associated restrictions): Modularity (with complex name binding rules), static type uniqueness (as opposed to considering only "raw" type "forms" and potential storage layout) etc. For such languages, the definition of dynamic semantics would be burdened and shadowed by these static concepts, had it not been for the rebuild functions! As major examples of this technique, refer to [Bjørner 80f] and [Haff 80a].

-- Semantic Domains, Final Approximation

The "total" environment, ρ, has two "parts": one, an "incoming", structured in two levels and recording all variable denotations, and another, an "incoming & resulting", structured in one level and recording all exported denotations only:

```
26.  ρ ∈ TENV = 2ENV ∪ XENV
27.      2ENV = (MAIN ⇀m LENV) ∪ (Mnm ⇀m LENV)
28.      XENV = ENV
29.      LENV = VAL ⇀m LOC
```

-- and otherwise as before.

-- Semantic Functions, Formal Exposé

```
30. type: I_prgr:   Prgr                → (π(Monitor) → (Σ → Σ))
31. type: I_sl:     Stmt                → (ENV → (π(Monitor) → (Σ→Σ)))
32. type: CL_dp:    Defs                → (Σ → (Σ  LENV))
33. type: CL_sms:   (Mnm ⇀m Submod)     → (Σ → (Σ (Mnm ⇀m LENV)))
34. type: C_mm:     Defs                → (TENV → (ENV ENV))
35. type: C_sms:    (Mnm ⇀m Submod)     → (TENV → ENV)
36. type: C_dp:     (MAIN | Mnm)        → (TENV → ENV)
37. type: Procden:  Proc                → (ENV → FCT)
38. type: Alloc:    ()                  → (Σ → (Σ LOC))
39. type: Exp:      Defs                → (Vid | Pid)-set
```

We start here, by stating the type of all needed functions. This is a good way to structure or organize definition work: First the "interesting" domains (Semantic Domains) are settled upon; next the type of the functions needed to create and manipulate them are settled upon, and, finally, the bodies of the functions are filled in. Here we have, from the above discussions on the design of the semantic domains, combined with the structure of the syntactic domains: manipulation of storage (interpretation of program and statement lists (30,31) (the π*(Monitor)* concept "belongs" to parallel semantics; see later)), computation of the location part of definition parts and submodules (32, 33), computation of the final environments of the main module, submodules and definition parts (34, 35,36). Thus, we are ready to "fill in":

```
30.0 I_prgr(mk-Prgr(mk-Main(dp,sl),sms))(monitor) Δ
  .1 (def ml ρ                    : CL_dp(dp),
  .2      sl ρ                    : CL_sms(sms);
  .3  let (ρmm, ρpmm)             = C_mm(dp)ρ,
  .4      ρsms                    = C_sms(sms)ρ,
  .5      ρ                       = [MAIN↦ml ρ] ∪ sl ρ
  .6                                ∪ dρ_mm ∪ ρsms          in
  .7 I_sl(sl)(ρmm)monitor)

32.0 CL_dp(mk-Defs(mk-Vars(xvs,,lvs),)) Δ
  .1 [v ↦ Alloc() | v∈(xvs ∪ lvs)]

33.0 CL_sms(sms)  Δ
  .1 [mnm ↦ CL_dp(sms(mnm)) | mnm ∈ dom sms]

34.0 Cmm(dp)ρ Δ
  .1 (let mk-Defs(mk-Vars(xvs,,),mk-Procs(xpm,,)) = dp in
  .2  let ρmm = C_dp(dp,MAIN)ρ in
  .3 (ρmm,(ρmm| merge{ C_dp(sms(n))ρ| Exp(sms(n))| n∈domsms}))

35.0 C_sms(sms)ρ  Δ
  .1 (sms=[] → []
  .2   T     → (let mnm∈ dom sms in
  .3              Cdp(sms(mnm))ρ ∪ C_sms(sms\{mnm})ρ)

36.0 C_dp(dp,nm)ρ Δ
  .1 (let mk-Defs(mk-Vars(xvs,ivs,),
                  mk-Procs(xpm,ips,lpm))= dp  in
  .2  let ρ' = ρ(nm)
  .3          ∪[ v↦(ρ(nm))(v) | v∈domivn ∧ nm = ivm(v)]
  .4          ∪[ p↦Procden(pm(p))ρ' | p∈dompm ∧ pm=xpmvlpm]
  .5          ∪[ p↦ρ(p) | p∈ips] in
  .6  ρ')

38.0 Alloc() Δ
  .1 (def l∈LOC \dom \dom c Stg;
  .2  stg := c Stg ∪ [l↦undef];
  .3  return l)

39.0 Exp(mk-Defs(mk-Vars(xvs,,),mk-Procs(xpm,,))) Δ
  .1   xvs ∪ dom xpm
```

Above, we have left unspecified the interpretation of statement lists (I_{sl}) and elaboration of the denotation of procedure identi-

fiers (*Procden*), as they are of no consequence to the "modularity" part of the semantics.

Note in 36.2-3 the reference to that part of ρ which was "constructed" in an earlier, non-recursive, and -- by the way -- imperatively expressed phase; while 36.4-5 refers (basically) to the ρ being constructed! The former plays the rôle of "incoming" only; the latter the rôle of both "incoming & resulting", cf. line 36.6.

2.1.4 Dynamic Parallel Semantics

In the above sub-sections various important aspects of definition of both static- and dynamic sequential semantics were illustrated. Here, we illustrate definition of dynamic parallel semantics, to be built "on top of" the previous work: we consider a system consisting of a set of individual programs (as defined above), to be executed concurrently. Synchronization is by means of a set of shared maphores, operated upon by (primitive) *signal* and *wait* statements (the reader may imagine more system components: directories, file-storage, etc., to which access should be synchronized):

-- Syntactic Domains -- Extension

```
40.  System      ::  Semid-set   Prgr-set
     ...
41.  Stmt        =   Signal | Wait | ...
42.  Signal      ::  Semid
43.  Wait        ::  Semid
```

VDM models involving concurrency are expressed as a set of communicating, sequential processes. Although this looks like "cheating", or simply as a re-statement of the object problem, this is most often not so! The concurrency concept used in *VDM* is a rather simple and restricted form of Hoare's *CSP* [Hoare 78a, Folkjär 80a], striving for possessing as properties concurrency, rendezvous, communication and only that; i.e. there are no assumptions abóut resource sharing, scheduling, buffering ing, etc. Thus *VDM* concurrent models have two parts, as have our PL example: A "sequential" part for defining *intra*-process semantics, and a "parallel" part for defining *inter*-process semantics, quite independently. For the example, we will use several processes in our model: One (called the *monitor*) for modelling semaphore semantics, and one for each program in the system. All inter-relation in the system is then modelled in terms of rendez-vous communications between these processes; the objects thus being communicated are the following:

-- Communication Domains

```
44.  Signalreq   ::  Semid
45.  Waitreq     ::  Semid
46.  Continue    ::  ()
```

For simplicity, we have omitted system termination from the model.

-- Processor Definitions

(Model) processes are formulated as started instances of processor "schemas" of which we need two: one for monitor execution and one

for program execution; also we need a "main" elaboration function for the entire system:

```
47.0  I_system (mk-System(sids,prgrs)) Δ
  .1   (def monitor   : start Monitor (sids);
  .2    def pπs       : {start Task(prgr,monitor) | prgrεprgrs};
  .3    ... )
  .4  type: System → ...
```

(as we do not model system termination, we do not know what the result, nor its type, is going to be!)

The Task processor is of course trivial:

```
48.0  processor Task(prgr,monitor) Δ
  .1  (dcl Stg := [] type STG;
  .2   I_prgr(prgr)(monitor))
  .3  type: (Prgr  π(monitor))
```

i.e., for each program in the system, a separate process defines a separate "private" state Σ, modelling the storage of that program.

```
49.0  processor Monitor(sids) Δ
  .1    (dcl sem := [ s↦(0,<>) | sεsids] type Semid ⇉m (N0 π*);
  .2     cycle
  .3     (input mk-Signalreq(s) from π
  .4       =>(def (n,πs) : c sem(s);
  .5          if πs ≠ <>
  .6            then (let π' = hd πs  in
  .7                  sem°(s) := (n,tl πs);
  .8                  output mk-Continue() to π')
  .9            else sem°(s):= (n+1,πs)),
  .10     input mk-Waitreq(s) from π
  .11      => (def (n,πs) : c sem(s);
  .12          if n > 0
  .13            then (sem°(s) := (n-1,πs);
  .14                  output mk-Continue() to π)
  .15            else sem°(s) := (n,πs^<π>))))

  .16  type: (Semid-set)
```

Note (.1,.6,.15) that the scheduling strategy of the system has been explicitly modelled (here, a usual "fair" queue) -- allowing for replacement in favour of any other, more suitable scheme, without affecting the remainder of the model!

Finally, we need to extend the sequential semantics set of formulae with interpretations of the signal and wait statements:

```
50.0  I_signal (mk-Signal(sid))(monitor) Δ
  .1     output mk-Signalreq(sid) to monitor
  .2   type: Signal → π(Monitor) → (Σ → Σ)


51.0  I_wait (mk-Wait(sid))(monitor) Δ
  .1    (output mk-Waitreq(sid) to monitor;
  .2     input mk-Continue() from monitor)
  .3  type: Wait → (π(monitor) → (Σ → Σ))
```

2.2 Structure of *VDM* Compiler Developments

From the static semantics (*SS*) definition of a PL is to be developed that part of the PL compiler front-end which is usually called: Contex Conition Checker, *CCC* (1).

From the dynamic sequential semantics (*DSS*) definition of a PL is to be developed the 'code generator', possibly including an 'optimizer', part of the PL compiler (2). And from the dynamic parallel semantics, *DPS*, definition of a PL one develops the process scheduling and inter-process synchronization run-time system. The latter may be developed for a mono-, a tightly coupled, bus-connected-, or a loosely coupled data communication connected, distributed multi-, processor system (3).

(2) Either one of two situations may arise as concerns code generation: either our compiler is to generate code for a given manufacturer's target machine; or we are at liberty, possibly within constraints, to generate code for a virtual machine which we define. In the latter case the development of the architecture of the virtual machine, *M*, is best pursued hand-in-hand with the development of the PL-to-*M*-Code Compiling algorithm, *CA*. The dynamic sequential semantics functions are at the basis of this development stage, but are usually too abstract. Hence a more concrete, operational semantics, of the so-called macro-substitution type, *MS*, is first developed. It serves as the basis for both the *M* architecture, and to PL-to-*M*Code compiling algorithm developments.

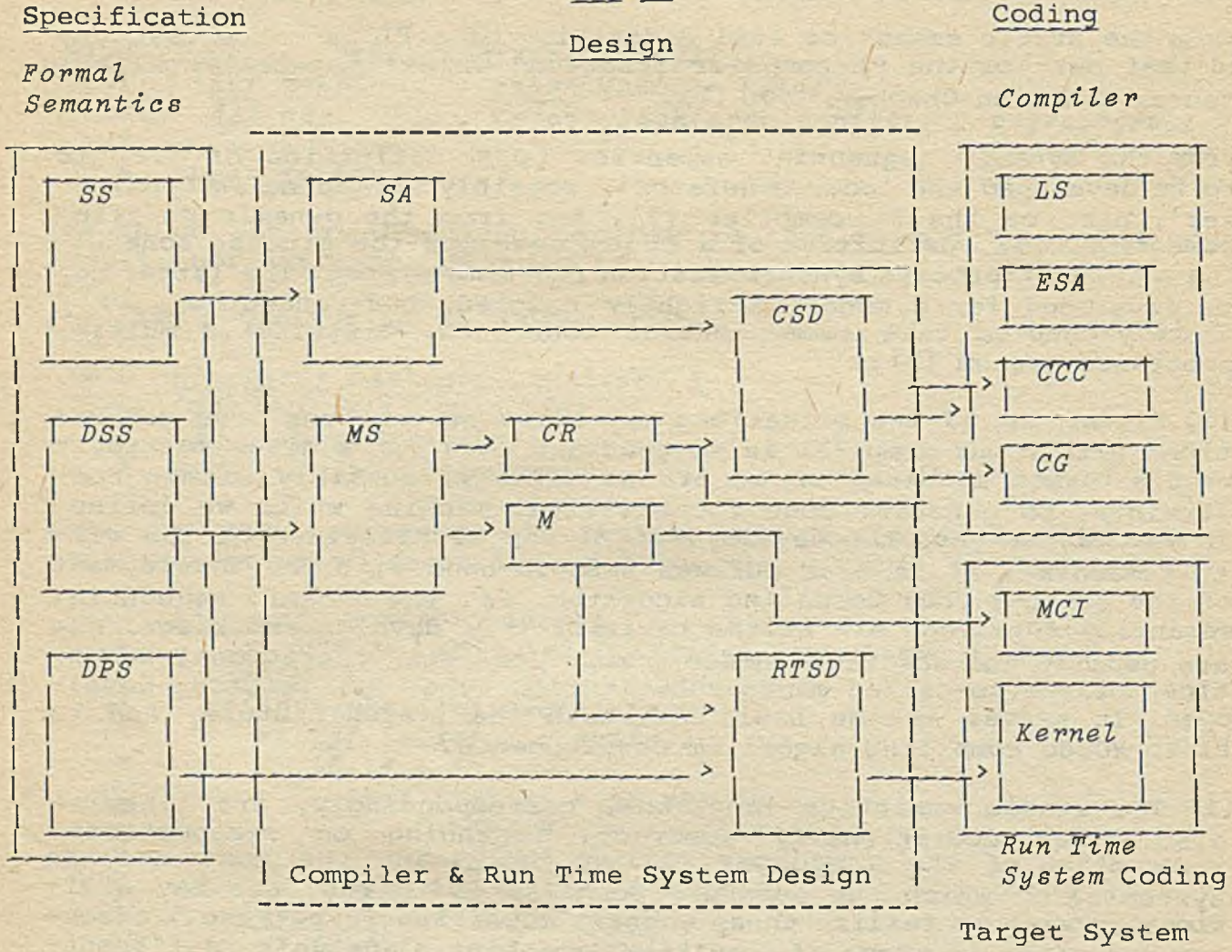
(1) The static semantics functions, correspondingly, are likewise also rather operationally abstract. Depending on circumstances (properties of the language (PL) at hand, and the host machine system(s) on which the eventual compiler is to run) one may additionally wish to realize these context condition (type-,etc.) checking functions in terms of a multi-pass semantic analysis *SA* ("front-end"). As a result, one first develops abstract specifications of what each such pass is to achieve, including the intermediate texts communicated from one pass to the next.

(1,2) Given the multi-pass descriptions of the semantic analysis and of the compiling algorithm, and given the "classical" approaches to lexical scanning *LS*, and (possibly error-correcting) syntax parsing, *(E)SA*, one is then in a position to design the overall structure, *CSD*, of the compiler. Included in this design is usually two additional aspects: the specification and design of the compiler interface to the host operating file and i/o systems, and the specification and design of possible separate compilation facilities.

(2) The specification of a virtual machine serves as the basis for the realization of either software, or firmware micro-programmed-, *M*-code interpreters, *MCI*, or the bitced ced design of a special *M*-code machines, considered as parts of the run-time system.

(1,2,3) in toto we get the following picture of the various compiler and run-time system development activities:

Fig 1.



Thus, the compiler and run-time system development is seen as consisting of three major phases: The Formal Semantics Definition phase, to the left, the staged design, within the dashed box, and the coding. Each of the many activities are carried out by constructing definition (*SS, DSS, DPS, M*) documents, design specification documents (*SA, MS, CA, CSD, RTSD*), and implementation language program modules (*LS, ESA, CCC, CG, MCI, Kernel*). The arrows in the above figure denote a time-wise partial order, and that documents produced by the arrow infixed activities form a pair of staged developments -- and hence are (to be) properly related.

The general *VDM* ideas exposed in section 1 carry over to these dedevelopments. The above figure can be slavically followed, using these techniques. Not before we reach the rightmost "stacked" boxes collectively labelled coding does actual coding start!

We next illustrate some of the above design activities.

2.2.1 Semantic Analysis

The domains and functions to be presented here as the *SA* specification is to be considered as a "conclusion" of two major design considerations. For brevity of exposition they are not separately documented as separate development steps. In our "real" (CHILL & Ada) Compiler Developments they are separately documented.

The concerns are:

- the concrete, linear syntax of our PL, i.e. a primary object refinement of the syntactic domains, imposing a (secondary) operational refinement.

- concrete implementation pragmatics, i.e. the requirement to process program text in left-toright passes; i.e. a (primary) operational refinement

In particular the latter concern introduces much extra "control structure" and necessitates static semantic domains to "accumulate" and "pass on" information (dictionary), while, at the same time, the former concern is absorbed in this machinery! In particular, this last step of "absorbing" is responsible for the rather large distance from our previous static semantics formulae to the following, quite mechanical *SA* specification, an undesirable fact which we will return to in forthcoming papers.

-- Syntactic Domains

We assume a linearized form of the program syntax of formulae (1.-8.) of the previous section:

```
1.   Prgr1     :: Main  Submod*
2.   Prgr2     :: Main  (Mnm Sub)*
3.   Main      :: Defs  Stmt+
4.   Submod    =  Defs
5.   Defs      :: Vars Procs
6.   Vars      :: Vid*  Vid*  Vid*
7.   Procs     :: (Pid Proc)*  Pid*  (Pid Proc)*
8.   Stmt      =  ...
9.   Proc      =  ...
```

-- Semantic Domains

Since we assume e.g. sub-modules to be "sequenced" in any order without affecting neither the static- nor the dynamic semantics, we in general need to construct first, in one pass, a dictionary, which is then used, in a subsequent pass, to check for well-formedness:

```
10.0 DICT = (Vid ~m (Mnm (VAR|CHK) nil))  u
  .1        (Pid ~m (Mnm (PROC|CHK) Proc))
```

Local dictionaries, for the checking of the statement list of the main module, and the procedure bodies of all modules have an identical structure.

-- Semantic Functions

The *is-wf-Prgr*$_1$ function first rebuilds ("remakes") the program *p* into one whose sub-modules are distinctly, but arbitrarily named (11.,12.0-2); then builds a dictionary, δ, which contains names of all exported variables and procedures, (11.2,14.-17. incl). If any such name is exported twice or more, it will be associated with a check mark (16.7). Finally, (11.3) the imported names, (23.8-.10),

local definitions (23.11-.13) and procedure bodies (22.6) are checked, the latter only in the dictionary context of the analyzed module (22.5).

Function Types

```
11. type:  is-wf-Prgr:          Prgr1 → BOOL
12. type:  rmk-Prgr:            Prgr1 → Prgr2
13. type:  Affix:               Submod-set → (Mnm-set →
                                              (MnmSubmod)*)
14. type:  B-Dict:              Prgr2 → DICT
15. type:  BC-Defs-Dict:        Defs → (DICT → (Mnm → DICT))
16. type:  BC-Dict:             (Vid* | (Pid Proc)*) (VAR|PROC)
                                  → (DICT → (Mnm → DICT))
17. type:  BC-subl-Dict:        (Mnm Sub)*→ (DICT → DICT)
18. type:  check-Prgr2:         Prgr2 → (DICT → BOOL)
19. type:  check-Dict:          DICT → BOOL
20. type:  check-Main:          Main → (DICT → BOOL)
21. type:  check-subl:          (Mnm Sub)* → (DICT → BOOL)
22. type:  CBL-Defs-Dict:       Defs → (DICT →
                                          (Mnm → (DICT BOOL)))
23. type:  names:               (Vid | Pid)*  (Pid Proc)*
                                          → (Vid | Pid)-set
24. type:  CBL:                 (Vid | Pid | (Pid Proc))*
                                (IMP | LOC) (VAR | PROC)
                                 → (DICT → (Mnm →
                                          → (BOOL → (Dict BOOL)))
25. type:  check-Stmt-list      Stmt+ → (DICT → BOOL)
26. type:  check-Procs:         (Pid Proc)* (Pid Proc)*
                                → (DICT → BOOL)
```

```
11.0  is-wf-Prgr1(p) Δ mk
  .1    (let p' = rmk-Prgr(p) in
  .2     let δ  = B-Dict(p')  in
  .3     check-Prgr2(p')δ)

12.0  rmk-Prgr(mk-Prgr(mm,sml)) Δ
  .1    (let mnms ⊂ Mnm be s:t: card mnms = len sml in
  .2     mk-Prgr2(mm,Affix(sml)(mnms)))

13.0  Affix(sml)(mnms) Δ
  .1    cases sml:
  .2    (<>          → <>,
  .3     <sm>^sml' → (let n ε mnms in
  .4                  <(n,sm)> ^ Affix(sml')(mnms \ {n})))

14.0  B-Dict(mk-Prgr2(mk-Main(dp,),nsml)) Δ
  .1    (let δ = BC-Defs-Dict(dp)([])(MAIN) in
  .2     BC-subl-Dict(nsml)δ)

15.0  BC-Defs-Dict(mk-Defs(mk-Vars(xvl,,),
                           mk-Procs(xpl,,)))(δ)(n) Δ
  .1    (let δ' = BC-Dict(xvl,VAR)(δ)(n) in
  .2     BC-Dict(xpl,PROC)(δ)(n))
```

```
16.0  BC-Dict(l,vop)(δ)(n) Δ
 .1     cases l:
 .2     (<>       → δ,
 .3      <e>^l' → (let(id,bd) = cases vop:
 .4                               (VAR  → (e,nil),
 .5                                PROC → e)        in
 .6                let δ' = if id ε domδ
 .7                         then δ + [id ↦ (n,CHK,bd)]
 .8                         else δ ∪ [id ↦ (n,vop,bd)] in
 .9                BC-Dict(list',vop)(δ')(n)))

17.0  BC-subl-Dict(nsml)δ Δ
 .1     cases nsml:
 .2     (<>                 → δ
 .3      <(n,dp)>^nsml'→(let δ'= BC-Defs-Dict(dp)(d)(n) in
 .4                      BC-subl-Dict(nsml')(d')))

18.0  check-Prgr2(mk-Prgr2(mm,nsml))(δ) Δ
 .1     check-Dict(δ)
 .2   ∧ check-Main(mm)δ
 .3   ∧ check-subl(nsml)δ

19.0  check-Dict(δ) Δ
        (∃(,chk,) ε rng δ)(chk = CHK)

20.0  check-Main(mk-Main(dp,sl))(δ) Δ
 .1     (let (δ',b) = BC-Defs-dict(dp)(δ)(MAIN) in
 .2      b ∧ check-Stmt-list(sl)(δ'))

21.0  check-subl(nsml)(δ)
 .1     cases nsml:
 .2     (<>                 → true
 .3      <(n,dp)>^nsml' → (let (,b) = BC-Defs-Dict(dp)(δ)(n)  in
 .4                        b ∧ check-submodlist(nsml')(δ)))

22.0  BC-Defs-Dict(mk-Defs(mk-Vars(xvl,ivl,lvl),
                           mk-Procs(xpl,ipl,lpl))(δ)(n) Δ
 .1   (let (δiv,biv) = BC(ivl,IMP,VAR)(δ)(n)(true)    in
 .2    let (δlv,blv) = BC(lvl,LOC,VAR)(δiv)(n)(biv)   in
 .3    let (δip,bip) = BC(ipl,IMP,PROC)(δlv)(n)(biv)  in
 .4    let (δlp,blp) = BC(lpl,LOC,PROC)(δip)(n)(bip)  in
 .5    let δ'        = δlp | names(xvl^ivl^lvl^ipl,xpl^lpl)  in
 .6    let b         = check-Procs(xpl,lpl)δ'                 in
 .7    (δlp,blp ∧ b))

23.0  names(nl,pl) Δ
 .1     elems nl ∪ {p | (p,)} ε elems pl
```

```
24.0  BC(list,iol,vop)(δ)(n)(b) ≜
 .1     cases list:
 .2     (<>          → (δ,b),
 .3      <e>^list' → (let(id,bd) = cases(vop,iol):
 .4                                ((VAR,)      → (e,ml),
 .5                                 (PROC,IMP) → (e,),
 .6                                 (PROC,LOC) → e)          in
 .7                   let b' = cases iol:
 .8                            (IMP → ((id ε domδ
 .9                                    ∧(let (n',t,) = δ(id)  in
 .10                                    (n'≠n) ∧ (t=vop))),
 .11                           LOC → ((id ε domδ) ⊃
 .12                                   (let(n',,) = δ(id)     in
 .13                                   n'≤ n)))               in
 .14                  let bd' = if b' ∧ (iol,vol) = (IMP,PROC)
 .15                            then (let (,,bd'') = δ(id)   in
 .16                                  bd'')
 .17                            else bd                      in
 .18                  let δ'  = δ † [id ↦(n,vop,bd')]        in
 .19                  CBL(list',iol,vop)(δ')(n)(b ∧ b')))
```

2.2.2 Macro-Expansion

The *DSS* of section 2.1.3, ie. formulae (15.-38.) is to be developed into an *MS*. We decide on realizing the combined *(ENV STG)* complex in terms of a complex of so-called activations: one for the main-module, and one for each of the sub-modules. All these activations are allocated "simultaneously", cf. (30.1), respectively (30.2). Each activation is uniquely designated by a pointer. Each activation contains allocations for all exported- & local variables (A32.-33.); "closures" for all procedures, whether exported-, imported-, or local (A36.4-.5) and allocation information for all variables (A36.2-.3).

```
25.   ENVSTG  = Ptr ⇉ ACTV

26.   ACTV    :: s-static:[Ptr] s-dyn:[Ptr]
 .1               s-ra: [Lbl]
 .2               s-env:((Vid ⇉ Ptr) ∪ (Pid ⇉ CLOS))
 .3               s-stg:(Vid ⇉ VAL
                  ...

27.   CLOS    :: Ptr Lbl
```

The idea of the macro-expansion stage is (also) to expand the procedure body text into meta-language "in-line" with the macro-expanded text which, through "call"s refer to those procedures. [Bjørner 77b] exemplifies the idea in detail. Suffice it here to summarize that invocations of procedures in a denotational definition is effected by finding the procedure denotation in the environment, and then applying the function it is to an evaluated argument list. In the macro-expanded, operational semantics version we have "compiled" all the source-language program text into meta-language text; and procedure calls are effected by jumping to an appropriate meta-text point, i.e. a label (in *Lbl*). The denotational procedure denotation embodied the defining environment. Now the operational procedure "closure" contains, besides a label, a pointer to the appropriate activation. Procedure invocations occur in the calling

environment, i.e. lead to an activation "stacked" on top of the calling activation & chained to it by a dynamic pointer (chain). Since procedures may possibly be passed as parameters to other procedures (i.e. to their invocation), or since procedure bodies may contain nested procedure definitions where inner ones may refer to outer ones, we also need to "chain" back to defining environments, i.e. we need, finally, in our activations, a static pointer (chain). All this pointer chaining is nothing new.

For each of the functions (30.-38.) of section 2.1.3 we have to re-define a corresponding set of macro-expansion functions. We now outline our design decisions. Space does not permit us here to motivate or discuss the specific choices as regards the technical structure of the run-time organization. Our point is to illustrate a technique of going from abstract, denotational-, to less abstract, more concrete operational definitions, and of how to relate them in an attempt to convince the reader of the possible correctness of the realization. (In the following we refer to the *DSS* formulae as (A...) and to those of this sub-section as (B...)).

The structure of (A30.) is as follows: (A30.1-.2) performs the allocation of all variables. (B36.1-.2) correspondingly performs the allocation of all activations, and -- within them -- the "allocation" of all relevant variables. (A30.5.6) describe an environment which records the denotations of all variables (A30.5) & procedures (A30.6). (A30.5) is constructed solely in terms of the results of (A30.1-.2). (A30.6) is constructed on the basis of (30.3-.4) -- which (recursively) uses the environment thus being constructed. (B36.3-.4) corresponds to (A30.3.4-.6) as follows: in three "sweeps" across main- & submodules we construct (1) arbitrary, but distinct labels for all procedures (2) insertion into all activations of closures for all procedures, and (3) macro-expansion into meta-text of all procedure bodies -- properly prefixed by their corresponding label. We also exploit the second "sweep" to insert the operational denotation of all imported variables.

We leave it to the reader to further study our solution below. In the right-margin we refer to section 2.1.3 formulae.

In constructing the macro-expansion the following auxiliary macro-expansion time (subscript s) and (meta-text) interpretation time (subscript d, $_s$ for static, $_d$ for dynamic) are used; also we abbreviate *Defs*, *Vars* and *Procs* as *Dp*, *Vp* and *Pp* respectively.

28.	*LblM*	=	*Name* $\overrightarrow{m}$ *PLM*	variables:	*nplm*
29.	*PLM*	=	*Pid* $\overrightarrow{m}$ *Lbl*	variables:	*plm*
30.	*PtrM*	=	*Name* $\overrightarrow{m}$ *Ptr*	variables:	*snpm, npm*
31.	*VarM*	=	*Vid* $\overrightarrow{m}$ *Ptr*	variables:	*xvm, sxvm, vpm*
32.	*Name*	=	*Mnm* | MAIN	variables:	*n*
33.	Σ_s	=	Lbls $\overrightarrow{m}$ *Lbl-set*		
34.	*dcl* Lbls	:=	{ } *type Lbl-set*		
35.	Σ_d	=	(Cpr $\overrightarrow{m}$ [*Ptr*]) $\cup$		*** Dynamic chain
35.1			(Epr $\overrightarrow{m}$ [*Ptr*]) $\cup$		*** Static chain
.2			(Rar $\overrightarrow{m}$ [*Lbl*]] $\cup$		*** Return address
.3			(Stg $\overrightarrow{m}$ *ENVSTG*) $\cup$		
.4			(Stack $\overrightarrow{m}$ *VAL**) $\cup$		...

```
36.   type: Mprgr:  Prgr2 → ((Σs → Σs) → (Σd → Σd))
37.   type: MLdp:   Dp → (Σd →(Σd(VarM Ptr)))
38.   type: MLsmm:  (Mnm ⇉ Dp) → (Σd →(Σd(VarM PtrM)))
39.   type: GLdp:   Dp → (Σs → (Σs PLM))
40.   type: GLsmm:  (Mnm ⇉ Dp)  → (Σs → (Σs LbeM))
41.   type: Ins:    Dp LblM Name → ((Ptr VarM) → (Σd → Σd))
42.   type: Msl:    Stmt+ → ((Σs → Σs) → (Σd → Σd))
43.   type: EP:     Proc  PLM  → (Σd → Σd))

36.0  Mprgr(mk-Prgr2(mk-Mainmod(mm,sl),smm)) Δ
  .1   (defd (xvm,mp)    : MLdp(mm);                          A30.1
  .2    defd (sxvm,snpm): MLsmm(smm),                        A30.2
  .3    letd vpm    = xvm ∪ sxvm,                            A30.5
  .4         npm    = snpm ∪ [MAIN ↦ mp] in                  A30.5
  .5    defs plm    : GLdp(mm);                              A30.3(1)
  .6    defs nplm'  : GLsmm(smm);                            A30.4(1)
  .7    lets nplm   = nplm' ∪ [MAIN ↦ plm] in                A30.3-4(2-3)
  .8    Ins(mm,nplm, MAIN)(mp,vpm);                          A30.3(2)
  .9    ⊧Ins (smm(n),nplm,n)(npm(n),vpm) | n∈dom smm ⊣;      A30.4(2)
  .10   defs lout: Get-Lbl();                                A30.3-4(3)
  .11   Cpr  := mp; Epr := mp;                               A30.7
  .12   Msl(sl);                                            A30.7
  .13   GOTO lout;                                          A30.3-4(3)
  .14   EP(mm,nplm(MAIN));                                  A30.3(3)
  .15   ⊧ EP(smm(n),nplm(n)) | n ∈ dom smm ⊣                A30.4(3)
  .16   Label(lout):)                                       A30.3-4(3)

37.0  MLdp(mk-Dp(mk-Vp(xvs,,lvs),)) Δ                                A32
  .1    (defd p     ∈ Ptr\dom c Stg;
  .2     letd actv = mk-ACTV(nil,nil,nil,[v ↦ p | v∈xvs∪lvs],
  .3                                [v ↦ UNDEF | v∈xvs∪lvs],...) in
  .4     Stg :=c Stg ∪ [p ↦ actv];
  .5     return ([x ↦ p | x∈xvs],p))

38.0  MLsms(smm) Δ                                                   A33
  .1    ((smm = [ ]) → ([],[]),
  .2        T        → (lets n∈ dom smm in
  .3                     defd (xvm,p) : MLdp(smm(n));
  .4                     defd (rxvm,rnpm):MLsms(smm\{n});
  .5                     return (xvm ∪ rxvm,rmpm ∪ [n ↦ p])))

39.0  GLdp(mk-Dp(,mk-Pp(xpm,,lpm))) Δ
  .1    [p↦Get-Lbl() | p∈ dom(xpm ∪ lpm)]

40.0  GLsmm(smm) Δ
        [n ↦ GLdp(smm(n)) | n ∈ dom smm]

41.0  Ins(mk-Dp(mk-Vp(,ivs,),mk-Pp(xpm,ips,lpm)),nplm,n)(mpm,vpm) Δ
  .1    (defd mk-ACTV(sta,dyn,ra,env,stg,...):(cStg)(mpm(n));
  .2     letd env' = env                                          A34.2
  .3                 ∪ [v↦vpm(v) | v∈ ivs]                      A34.3
  .4                 ∪ [p↦mk-CLOS(mpm(n),(nplm(n))(p)           A34.4
  .5                     | p∈ dom (xpm ∪ lpm)]
  .6                 ∪ [p↦mk-CLOS(mpm(n'),(nplm(n'))(p)
  .7                     | p∈ ips ∧ n'∈ dom nplm ∧ p ∈ nplm(n')] in
  .8    Stg  := Stg  ∪[ (mpm(n))↦mk-ACTV(sta,syn,ra,env',stg,...)])
```

```
42.0  M_sl(sl) ≜
        ⦃ M_s(sl[i] | 1≤i≤ len sl ⦄

43.0  EP(mk-Dp(,mk-Pp(xpm,,lpm))(plm) ≜
  .1  ⦃ label(plm(p)): Mproc((xpm∪lpm)(p))(...); GOTO cRar
  .2      | p∈ dom(xpm∪lpm)⦄

44.0  Get-Lbl( )≜
  .1    (def_s l ∈ Lbl\dom c Lbls;
  .1     Lbls:= c Lbls . ⊂l∧;
  .2     return (l))
  .3  type: → (Σ_s → (Σ_s Lbl))
```

2.2.3 Virtual Machine & Compiling Algorithm

The *M*-Code machine architecture now to be developed shall be derived systematically from the macro-expansion specification of the previous section. This is so since the purpose of *M*-code is to be an efficient target for compiled programs. The design of the run-time state, Σ_d, for the execution of the macro-expanded meta-text, hence is the basis for our design of the *M*-Code instruction repertoire.

Our work is, therefore, laid out: to go through each and every line, but only lines, of the macro-expansion and decide which corresponding *M*-Code machine facilities (storage, registers, instructions, etc.) the inspected line should give rise to.

The criterion for the design of *M*-Code concepts is to be constrained by the kind of target machine(s) for which *M*Code interpreters are to be constructed. The addressing structures of these, whether they are stack or registeroriented, etc., influence the *M*-code design.

We shall not exemplify these concerns. We merely list the result of an *M*-code design. We have not, in section 2.1 nor in the previous sub-sections of this section illustrated such PL-constructs as expressions or statements, including call of and return from procedures.

The example of this sub-section would become rather meaningless if we did not now extend the previous examples:

```
45.   Stmt  = Asgn | If | Call ...
46.   Asgn :: Vid Expr
47.   If   :: Expr Stmt* Stmt*
48.   Call :: Pid Expr*
49.   Expr  = Vid | ...
```

-- Denotational Dynamic Semantics

```
50.   type:I_s:  Stmt → (ENV → (Σ → Σ))
51.   type:V_e:  Expr → (ENV → (Σ → (Σ VAL)))


50.0  Is(s)ρ ≜
  .1  cases s:
  .2  (mk-Asgn(v,e)  → (def w : V_e(e)ρ;
  .3                    Stg := c Stg + [ρ(v)↦w]),
```

```
.4   mk-If(e,c,a)   → (def b : Ve(e)ρ;
.5                      if b then Isl(c)ρ else Isl(a)ρ),
.6    mk-Call(p,el) → (def al : <Ve(el[i]ρ | 1≤i≤ len el>;
.7                      let f = ρ(p);
.8                      f(al)))


51.0  Ve(e)(ρ) Δ
  .1  cases
  .2  (is-Vid(e) → (cStg)(ρ(e)),
  .3  ...)
```

-- Macro Expansion

Now, we proceed as before; from the semantic functions we derive the macro-expansion formulae; from (50.,51.) we derive (52.,53) while the functions (54.,57.) may be seen as macro-expansion realisations of the abstract operations on the abstract environment and storage (A16.,..M9.,A26.,.. A29).

```
52.  type: Ms:        Stmt → ((Σd → Σd) → (Σd → Σd))
53.  type: Me:        Expr → (Σd → Σd)
54.  type: R:         Name → (Σd → (ptr | CLOS))
55.  type: RMe:       Expr → (Σd → (Σd → (Σd VAL))
56.  type: Assign:    (Ptr Vid) VAL → (Σd → Σd)
57.  type: Retrieve:  Ptr Vid → (Σd → Σd)

52.0  Ms(s) Δ
  .1  cases s:
  .2  (mk-Asgn(v,e) → (defd ptr : R(v);                          50.3
  .3                   defd w   : RMe(e);                        50.2
  .4                   Asign((ptr,v),w)),                        50.3
  .5   mk-if(e,c,a) → (defs (lcons,lout):(get-Lbl(),Get-Lbl());  50.5
  .6                   defd b : RMe(e);                          50.4
  .7                   ifd b then gotod lcons;                   50.5
  .8                   Msl(a);                                   50.5
  .9                   gotod lout;                               50.5
 .10                   label(lcons): Msl(c);                     50.5
 .11                   label(lout):),                            50.5
 .12   mk-Call(p,el) → (defs lret : Get-Lbl();                   50.6
 .13                   |<Ml(el[i](...) | 1 ≤ i ≤ len el|>;       50.6
 .14                   defd(ep,br): R(p);                        50.7
 .15                   Epr := ep ; Brr:=br;
 .16                   Rar := lret;
 .17                   gotod cBrr;                               50.8
 .18                   label(lret):))

53.0  Me(e) Δ
  .1  cases:
  .2  (is-Vid(e) → (def ptr: R(e);
  .3                 Retrieve (ptr,v)),
  .4   ...)

54.0  R(name) Δ
  .1    (defd env : s-env((cStg)(cCpr));
  .2     return (env(name)))
```

```
55.0  RM_e(e) Δ
  .1    (M_e(e);
  .2     def_d val: hd c Stack;
  .3     Stack := tl c Stack;
  .4     return (val))

56.0  Assign((ptr,v),val) Δ
  .1  (def_d mk-ACTV(sta,dyn,ra,env,stg,...) : (cStg)(ptr);
  .2   let_d stg' = stg + [v↦val] in_d
  .3   Stg := c Stg + [ptr↦mk-ACTV(sta,dyn,ra,env,stg',...)])

57.0  Retrieve(ptr,v) Δ
  .1  (def_d mk-ACTV(sta,dyn,ra,env,stg,...) : (cStg)(ptr);
  .2   def_d val = stg(v) in_d
  .3   Stack := <val> ^ c Stack)
```

-- Virtual Machine Design

The *M*-code machine state will be almost identical to the state enunciated for the macro-expansion, i.e. the run-time state. (Thus we have indeed been very abstract, and not favoured any particular addressing structure of any one target machine.)

Semantic Domains

```
58.0  Σ_M =   (Cpr  ⇗  [Ptr])   ∪
  .1          (Epr  ⇗  [Ptr])   ∪
  .2          (Brr  ⇗  [Lbl])   ∪
  .3          (Rar  ⇗  [Lbl])   ∪
  .4          (Stg  ⇗  ENVSTG)  ∪
  .5          (Stack ⇗ VAL*)    ∪   ...
```

Syntactic Domains

The instruction repertoire is

```
59.   Code   =  Ins*
60.   Ins    =  Lbl | Brch | Cbrch | Iload | Pop | Push |...
61.   Lbl    ::  Lab
62.   Brch   ::  (Lab | Brr | Rar)
63.   Cbrch  ::  Lab
64.   Iload  ::  Lbl  (Brr | Rar)
65.   Pop    ::  Vid
66.   Push   ::  Vid
```

Semantic Functions

```
67. type: I_M:       Code      →  (Σ_M → Σ_M)
68. type: cue-I_IL:  Code N_1  →  (Σ_M → Σ_M)
69. type: I_Ins:     Ins       →  (Σ_M → Σ_M)
70. type: I_pop:     Pop       →  (Σ_M → Σ_M)
71. type: I_push:    Push      →  (Σ_M → Σ_M)

67.0  I_M(code) Δ
  .1    (fixe [l ↦ cue-I_IL(code,i)
  .2           | iεindcode ∧ lεLab ∧ code[i] = mk-Lbl(l)] in
  .3     cue-I_IL(code,1))
```

```
68.0 cue-I_IL(code,i) Δ
  .1   if i > len code
  .2     then I
  .3     else I_Ins(code[i]); cue-I_IL(code,i+1)

69.0 I_Ins(ins) Δ
  .1   cases ins:
  .2   (mk-Lbl( ..)    → I,
  .3    mk-Brch(lr)    → cases:
  .4                      (is-Lab(lr) → exit lr,
  .5                       T          → exit clr),
  .6    mk-Cbrch(l)    → (def b : hd c Stack;
  .7                      Stack := tl c Stack;
  .8                      if b
  .9                        then exit l
  .10                       else I),
  .11   mk-Iload(l,r)  → r := l,
  .12   mk-Pop(...)    → I_pop(ins),
  .13   mk-Push(...)   → I_push(ins),
  .14   ...)

70.0 I_pop(mk-Pop(v)) Δ
  .1   (def mk-ACTV(sta,dyn,rar,env,stg,...): (cStg)(cEpr),
  .2         val: hd cStack;
  .3   Stg := cStg + [(cStg)(cEpr)↦mk-ACTV(sta,dyn,rar,env,
  .4                                       stg+[env(v)↦val],...)];
  .5   Stack := tl cStack)

71.0 I_push(mk-Push(v)) Δ
  .1   (def mk-ACTV(sta,dyn,rar,env,stg,...): (cStg)(cEpr);
  .2   Stack := <stg(env(v))> ^ cStack)
```

-- Compiling Algorithm

Finally, we are in a position to generate code! The compiling algorithm *(CA)* serves to specify, for any language construct, what the instruction sequence for any such construct should be. Here we only briefly illustrate the idea, by showing *CA* parts for the statements introduced above. Also, we hint at the handling of nested procedures by including means to express the static "level" (i.e. defining environment) for any procedure or variable:

Semantic Domains

72. $\delta \in LDICT = (Vid \mid Pid) \overrightarrow{m} N_1$

We do not, however, show the construction of this dictionary.

Semantic Functions

```
73. type: C_sl:     Stmt+  → (LDICT → (N_1 → Code))
74. type: C_s:      Stmt:  → (LDICT → (N_1 → Code))
75. type: C_asgn:   Asgn:  → (LDICT → (N_1 → Code))
76. type: C_if:     If:    → (LDICT → (N_1 → Code))
77. type: C_call:   Call:  → (LDICT → (N_1 → Code))
78. type: C_e:      Expr:  → (LDICT → (N_1 → Code))
79. type: Look-up_v: Vid   → (LDICT → (N_1 → Code))
80. type: Look-up_p: Pid   → (LDICT → (N_1 → Code))
```

```
73.0 C_sl(sl)(δ)(n) Δ
  .1   conc <C_s(sl[i]) | 1 ≤ i ≤ len sl>

74.0 C_s(s)(δ)(n) Δ
  .1   cases s:
  .2   (mk-Asgn(...)  → C_asgn(s)(δ)(n),
  .3    mk-If(...)    → C_if(s)(δ)(n),
  .4    mk-Call(...)  → C_call(s)(δ)(n),
  .5    ...)

75.0 C_asgn(mk-Asgn(v,e))(δ)(n) Δ
  .1   (let_s m = δ(v) in
  .2    C_e(e)(δ)(n) ^
  .3    Look-up_v(v,n-m+1) ^
  .4    <mk-Pop(v)>)

76.0 C_if(mk-If(e,c,a))(δ)(n) Δ
  .1   (def_s (lcons,lout) : (Get-Lab(), Get-Lab());
  .2    C_e(e)(δ)(n) ^
  .3    <mk-Cbrch(lcons)> ^
  .4    C_sl(a)(δ)(n) ^
  .5    <mk-Brch(lout)> ^
  .6    <mk-Lbl(lcons)> ^
  .7    C_sl(c)(δ)(n) ^
  .8    <mk-Lbl(lout)>)

77.0 C_call(mk-Call(p,el))(δ)(n) Δ
  .1   (def_s lret: Get-Lab();
  .2    let_s m = δ(p) in
  .3    conc <C_e(el[i])(δ)(n) | 1 ≤ i ≤ len el> ^
  .4    Look-up_p(p,n-m+1) ^
  .5    <mk-Iload(lret,Rar)> ^
  .6    <mk-Brch(Brr)> ^
  .7    <mk-Lbl(lret)>)
```

3. RESEARCH REQUIREMENTS

Roughly two types of research related to the *VDM* are required: foundational and applied.

Typical subjects of basic (foundational) research are: fixed point theory; proof theories; non-determinacy (incl. parallellism & process) models; etc. These are studies of computer science related matters: un-biased investigations into the kinds of systems that can be expressed, and the kinds of developments that possibly can be pursued -- irrespective of whether one actually wants to construct such systems.

We shall not focus on this area. The field is rich, and many, very exciting research lines, ultimately relevant to *VDM* are being pursued.

Instead we shall briefly dwell on the computing science and software engineering research needed to further support *VDM* like developments. We specifically focus on the Formalization of Programming Concepts and Programming Methodology aspects.

3.1 Formal Description of Programming Concepts

In conceiving and structuring a good software architecture it is important to analyze it into its constituent concepts. For 'ordinary' programming languages these could be: the type concept; the concept of assignable variables, and hence the concept of storage; the concept of sequencing; the scope, binding & extent, i.e. the block concept; the name visibility control, abstract data type, i.e. module concept; the concept of procedures & parameter transmission; the concept of abnormal control flow: gotos, exits, exceptions, etc.; the concept of processes; the concept of dynamic data structures; etc. For data bases the type concept may take on a hierarchical nature: "external", "conceptual" and "internal schemas", to mention just one aspect; while the dynamic data structure aspect usually becomes richer than in most ordinary programming languages. In command languages, say for distributed applications, the concept of processes take the lead rôle; etc.

To each of these concepts there then exists a variety of modelling styles -- each particular one chosen so as to display most effectively the notions expressed. The storage models, e.g. of PL/I and ALGOL 68 are of one variety; those of Pascal and Ada of another; and those of Basic & ALGOL 60 of yet a third variety. Similar for for example 'module' concepts. To become an effective architect, i.e. to extract from potential users answers to fundamental issues, one must master these concepts, and their varietal modelling.

Not enough research, carried e.g. in the style of VDM models, has been done. We need to far more systematically design models of prominent varieties of each class of concepts. [Strachey ***] very succinctly pointed out this direction. [Bjørner 83*] exemplifies varieties for each of the above concepts, with chapter 12 of [Bjørner 82e] focusing on data models for data bases ([Bjørner 82c] adds to these a model of IMS).

3.2 Programming Methodology

Whereas the previous section applied primarily to the abstract definition stage, this section applies to the implementation stages.

We exemplied uses of both object and operation transformations. Darlington & Burstall [Darlington 78a, Darlington ***, Burstall ***], the Munich CIP project [Bauer ***], and [Arsac ***], notably amongst others, have primarily researched operation transformations. We would like, and even prefer, to see more attention being directed towards object transformations. Thus we would ultimately like to see established 'catalogues' of object and operation transformations. [Bjorner 78e, Jones 79a, Jones 80a] and chapter 10 of [Bjorner 82e] illustrate *VDM* based object transformations. Much more work need be done in this area. In the particular application area of compiler development sect. 2 of this paper has also illustrated some transformations. Chapters 8 & 9 of [Bjorner 82e] (by resp. C.B.Jones and D.Bjorner) and [Jones 78c, Bjørner 78a] further illustrate methods for transforming denotational dynamic semantics. Again we can conclude that much more work need be done in this area before *VDM* can really be claimed 'a method'.

4. *VDM* SUPPORT TOOLS

A number of tools are required to support *VDM* based developments. These tools are to support the development and maintenance of formal definitions and design specifications; are to support the stepwise transformation of formal definitions into design specifications, and of these latter into code -in either *BCPL, Ada, Pascal* or *CHILL;* and are to support the correctness verification of evolving design and code.

For the purposes of this exposition these tools may be viewed in four clusters:

Cluster 1: Document Processors

Cluster 2: Compute Processors

Cluster 3: Tranformation Processors

Cluster 4: Correctness Processors

-- that is: this decomposition is presently of only pragmatic nature.

In the following we detail our preliminary understanding of what these tools effect. We outline essentials, using *VDM* jargon, and, at this time, refrain from detailed lists of individual tools by name, function and input/output.

We should like to stress that these tools primariily are of a non-clerical nature and are very much *VDM/META-IV* biased. And we should like to warn that although thus based on very formal grounds that the tools will stretch from rather mechanical, via semi-automatic to fully automatic tools.

4.1 Document Processors

The document processor cluster of tools consists of (1) full *META-IV* oriented syntax directed editors for the bulk and interactive input of arbitrary parts of formal definitions and design specifications; of (2) consistency and completeness checkers; of (3) pretty printers; of (4) formulae index and cross referencing generators; and the like. Consistency and completeness checking includes statically decidable context condition checking, i.e. that used names are defined, (polymorphic) type-checking, etc.

It must be stressed that all of these processors handle not only full *META-IV* and informal extensions thereof, but large scale definitions and specifications. We are here talking of a single, complete definition grossing up to 20.000 lines -- with derived specifications doubling and tripling these numbers! (Of course, these documents are well structured, in the sense of algebraic specifications, and this structuring will eventually enable us to achieve manipulability.)

4.2 Compute Processors

Consistency and completeness checking and formulae indexing and

cross referencing, features of the document processors, are really special examples of computations on definitions and specifications.

In general there is an ever-increaing need for querying definitions (and specifications) as to whether certain properties hold -- short of verifying these through the use of elaborate proof procedures. The cluster of compute processor tools thus include such facilities as data- and controlflow analysis, symbolic- and even "real" interpretive execution of definitions and specifications, etc.

Since we may wish to permit less formal, i.e. less stringent definition parts, the compute processors will have to adapt to that. Thus it is expected that computations on documents is interactive, i.e. can be on-line aided and (re)directed by requestors.

4.3 Transformation Processors

In transforming one stage of development (a definition or a design) into a next stage of development (a design, respectively code) a number of techniques are used. These can be more or less automatically carried out and hence supported by tools.

Two primary transformations techniques are applied in *VDM:* object-, respectively operation-transformations. The latter can be supported in the manner outlined by Burstall and Darlington, Arsac and by the Munich *CIP* project. The former similarly! That is: just as the transformation of e.g. recursively defined applicative functions into iteratively defined imperative procedures are supportable by a catalogue of operation tranformation schemes, so one can establish a catalogue of suitable object transformation schemes. In addition to automatic and semi-automatic transformation schemes, the system must enable, but monitor, user-defined transformations. These are to be checked by means of supporting the definition of user-defined object invariant-, retrieval- (abstraction) and adequacy functions.

The transformation processors thus handle pairs of documents: either (formal definition, design specification), or (high level design, low level design) specification, or (design specification, code).

4.4 Correctness Processors

-- and so do the correctness processor tools: here the problems form a specialized version of computing on definitions and designs, namely correlating these, verifying that a design abstracts to, or "commutes with", the definition from which it was supposedly derived.

For both the compute- and the correctness processors it is important that the meta-language (here: *META-IV*) is formal in the sense of being precisely understood. There are uses of *META-IV* for which an automatic proof system cannot be established (for example such where sets are used, such which involve rather generally expressed predicates, or whenever "pure" *META-IV* is mixed with *CSP*-like extensions).

One, therefore, has to be prepared for this "slack". One could force, in general, a restriction on *META-IV* expressibility to guarantee a proof system. But this would seriously restrict its applicability without in effect gaining very much. As it stands now, full *META-IV* is formal in the sense of every use of every construct being checkable for consistency and completeness, and hence every definition and design being semi-automatically verifiable.

We take Edinburgh *LCF* as an inspiring departure point for our design of correctness processors.

4.5 Support System Design Requirements

A first set of of requirements of the *VDM* support system is that it handle full *META-IV;* that as large subsets of *METAIV* be delineated for automatic computations: interpretation and verification; that remaining, non-atomatable aspects be interactively manipulable by users; and that the system accomodates large to very-large documents: in the order 1030.000 lines per full definition- or design document.

A second set of requirements of the *VDM* support system is that it be thoroughly and extensively architected and formally defined before costly efforts be commenced. That is: we are concerned about defining and designing one set of processor tools which subsequently turns out to be illadaptable to other processors tools; and in general: we are worried about fixing a representation of definition and design documents suitable for one set of processor tools, but ill-chosen for sets of tools of other processors.

Derived requirements appear, therefore, to point to a very general representation of *VDM* definitions and designs. The author of this note presently has the distinct feeling that one perhaps should be inspired by, if not directly taking over, the newest *MENTOR* ideas, but somebody ought check *CDL/2* for alternative and/or complementing ideas [Kahn ***, Koster ***].

5. MANAGEMENT ASPECTS OF VDM

We have illustrated, or hinted at, the following management aspects of VDM-based software development:

(1) As the very first step of any software development, establish the method to be used: not just by its name (VDM, HDM,JSP/JSD, Yordon SA/SD, etc.), but by how it is to be applied. We illustrated this point when outlining the overall strategy for compiler development, cf. sect. 2.2. Similar overall strategies can be established for other kinds of systems and applications software, but see discussion below.

(2) As the next step ascertain the man-power requirements: number, skills & time for each of the steps.

(3) As the final step apply the doctrines of the individual steps of VDM in pursuing the management of each individual step.

Discussion

(1) If no method can be trusted to bring a reasonable team of programmers from requirements via architecture definition, and design to coding, then do not embark on any development. This warning is succinctly expressed, and, not following it, vividly illustrated in [Hoare 81a]. If management cannot itself conceive of a way of applying, e.g. *VDM*, to a particular product development then there is no hope that any meaningful management can ever take place. As a programmer, I would not trust my management if it did not fully appreciate & understand the technical aspects of each subtask to be performed, let alone was potentially capable of carrying out the work themselves.

(2) We have indicated, in fig. 1, sect. 2.2, that the formal definition of e.g. the static semantics undergo a step of development while, or before, being readied for the compiler structure design & subsequent coding. The problem we wish to address here is that one step of development need not be enough. For languages like Edison it might suffice, for languages like Pascal we would advise one step for transforming the static semantics to an n-pass (n=1,2,..) front-end specification which (still) uses abstractions of dictionaries and intermediate texts, while a second step concretizes these latter. For CHILL and Ada several more steps of development (corresponding to the one box labelled *SA* in fig. 1) is strongly advisable. Now to the disturbing conclusion: In compiler developments for languages like Ada, and in general for the development of complex systems, one is not always capable of ascertaining the number of stages of development needed, before, or even right after, a formal architecture definition is to be, resp. has been, constructed.

One may choose, as in the case of Ada, to accept this fact and run the associated risk of having to adjust resource estimates as one goes along. Or one may choose, perhaps more wisely, not to embark on development projects whose individual, detailed stages of object & operation transformation cannot be firmly established beforehand.

(3) The individual step doctrines of VDM are basically these:

(F1) When defining the architecture of a new system, or when trying to formalize some informal description already given, such as e.g. Ada, sketch first the state, i.e. the semantic domains.

(F2) When attempting to fix the semantic domains, including the auxiliary functions on objects of these domains (i.e. in toto: the abstract data types) you may have to unravel the architecture itself in stages. This point is illustrated in chapter 11 of [Bjørner 82a]. The development of CHILL and Ada static semantics formal definitions also underwent several stages of development, in fact, even iterations completely redoing these stages.

(F3) Then sketch the syntactic domains and finalize the semantic domains and the auxiliary functions which from pieces

of syntactic objects construct corresponding pieces of semantic objects. It is at this stage re-do iterations may be required.

(F4) Only when satisfied that the basic problems have been reasonably elegantly understood (i.e. expressed), only then is the time ripe for the usually far easier task of writing the semantic functions.

The author has previously remarked [Bjørner 80b] that doctrines (F1-F4) apparently were not followed and with detrimental results, in other well-publicized formal definition work.

The implications, to management, of doctrines F1-F4 are obvious: the more experimental, the more "advanced" and complex a system one is attempting to develop, the more time must be set aside to carry through the stepwise architecture "experiments".

Whether "standard" or "advanced", management can always expect to receive, sooner or later, confirmation that the semantic domains have been sketched, that the auxiliary functions have first had their types sketched, then their bodies. And after the initial sketches follow the final definitions. Eventually management can tick off that an architecture is now well understood, consistently and completely defined. This marks important progress in a project. One can now turn one's attention from understanding and defining what, to understanding & designing how: from architecture to design.

(D1) When designing an implementation, i.e. when transforming from a more abstract design specification or the architecture definition to a more concrete design specification decide first whether to do a stage of object-, or a stage of operation-transformation, cf. sub-sections 1.2 and 1.3.

(D2) In conjunction with this decide which specific such object- or operation transformation to perform.

(D3) For the case of the decidedly more crucial stages of object transformation proceed by establishing the new invariant predicates, then the abstraction or retrieve functions, and finally, the associated (secondary) operation transformation, cf. sub-section 1.2.

The implications, to management, of doctrines D1-D3 are correspondingly obvious: Once design has begun it can expect a "constant" flow of documents, collections of which represent stages of development -- hence progress in the large -- and individual ones of which represent invariant predicates, retrieval functions, etc.. Each represent a sub-task which is meaningful in either of three senses: either, as for invariant - or retrieve-functions, their oftentimes laborious construction signify that a deeper understanding has been recorded of what is being constructed; or the project has come a step closer to realization; or a new review of what is being implemented has been achieved, a review which views the architecture & the design from yet another angle and now at a closer range! In toto: we can expect that misunderstandings and mistakes made in earlier steps become uncovered in immediately subsequent steps, and hence we can either expect (i.e. "wish for"), or explicitly

prove, correctness of implementation. Management in reality still does not have any running code to point to. But we claim to have something better: something we may better understand, designs whose properties can be evaluated.

Finally, there is the stage of coding. Since it is just a final step of development it is subject to the same doctrines and techniques as the general design stages. Usually commercial software developments start out estimating lines-of-code in the final product, and, based on such hopes, estimates, given various rules-of-thumb, the required man-months to produce these lines. We refuse to speculate before having understood what & how to implement, and even then we have been greatly surprised at realizing certain relationships between sizes of definitions, sizes of design specifications, and final sizes of code. We can also report some startling productivity figures:

Respective Sizes of Documents

CHILL	Definition	Design	Code
Static Semantics(total)	3800	9000	24500
Visibility Analysis	1250	3600	12000
Dynamic Sequential Semantics	1800	4000	11000

Productivity
(measured from before definition to after system integration)

Product	Man Years	Lines-of-Code	LOC/Hour
CHILL	9 1)	55000	4
CHILL	4 2)	55000	9
SLC	3	34000	10
LUCAS	1 3)	8000	4

1) including participation in language design effort and official publication of CHILL formal definition.

2) excluding language definition, i.e. compiler development

3) including research and prototype development

6. EDUCATION AND TRAINING ASPECTS OF *VDM*

The educational aspects of *VDM* are basically these: the current generation of computer science and software engineering candidates from an increasing number of leading universities worldwide will have learnt the theories underlying important ideas of *VDM*, and will have been taught methods similar to, if not identical with *VDM*. The first author of this paper has taught *VDM* courses internationally, at universities, over the last 5 years, and can conclude that it usually takes 2-3 semesters of formal lecturing, exercises and projects for run-of-the-mill, slightly above-average students, already acquainted with, say Pascal programming and computing systems, to become solid practitioners of *VDM*. Each semester is 10-14 weeks of 3 hours of lecturing per week -- with matching 2-3 hour lecture room exercises per week.

(This is not the place to outline such a course, but details are given in [Bjørner 83*].)

The training (or: re-education) aspects of *VDM* is basically this: reasonable mathematically mature, and otherwise welltrained programmers can rather painlessly be re-schooled. Here a most important pre-requisite is one of attitude. We are not speaking of requiring deep mathematical knowledge, nor of dramatic programming-skill upheavals. Such training amounts to an intense 1-2 month full day course centered around a trilogy of formal lectures, exercises and projects. *VDM* courses, like sketched, have been given to professional programmers in industry in Denmark, W.Germany, England, Hungary, China (PRC), and Italy. Approximately 15 such courses have presently been conducted to an average of between 15-25 people, with approximately 5 courses being planned. The following course material has been, or is being used: [Bjorner 78b, Jones 80a, Bjorner 82e, Bjorner 83*] and excerpts from [Bjorner 80a]; except for [Bjorner 83*] they are all published books, easily available.

7. THE SOCIOLOGY OF *VDM*

The physicist Schrödinger said:

> *Old theories are never proven wrong, they just die*
> *-- from lack of interest from new generations*
> *having been brought up with new theories.*

When confronted with a rigorous method, like e.g. *VDM*, managers tend to ask for proofs of *VDM*s superiority over existing ways of "hacking" through a software development. We might one day claim some "spectacular" productivity and quality figures. But presently we are really not that concerned. To us a method like *VDM* makes sense, and most other "methods" do not. (To us *JSD* also makes sense, but *Yordon, SA, SD, SADT, ISAC*, what have you, makes no sense.)

When candidates, well-trained in e.g. *VDM*, leave university to enter a professional life in some software industry some problems may arise. Either they become isolated, because no other colleagues master the formal techniques -- and they effectively have to "drop back" to the informal "systems" for developing software. Or they form a clique, a mass of e.g. *VDM*-capable profesionals critical

enough in size to pursue things their way. In the former case they are not capable of "surviving", i.e. of contributing their skills. In the latter case they, we claim, will eventually outperform "older" groups. In either case: management faces problems too.

VDM BIBLIOGRAPHY

[Bekić 74a] H.Bekić,D.Bjørner,W.Henhapl,C.B.Jones & P.Lucas: *A Formal Definition of a* PL/I *Subset*, Vienna TR25.139, Dec. 1974.

[Bjørner 77a] D.Bjørner: *Programming Languages: Linguistics & Semantics*, in: [Morlet 77a], pp.: 511-536, 1977.

[Bjørner 77b] D.Bjørner: *Programming Languages: Formal Development of Interpreters & Compilers*, in: [Morlet 77a] pp.1-21, 1977.

[Bjørner 78a] D.Bjørner: *The Systematic Development of a Compiling Algorithm*, in: *Le Point la Compilation* (eds. Amirchahy & Neel), IRIA Publ., Paris, pp. 45-88, 1979.

[Bjørner 78b] D.Bjørner, C.B. Jones (eds.): *The Vienna Development Method: The Meta-Language*, LNCS 61, 1978.

[Bjørner 78c] D.Bjørner: *Programming in the Meta-Language: A Tutorial*, in: [Bjørner 78b] pp. 24-217, 1978.

[Bjørner 78d] D.Bjørner: *Software Abstraction Principles: Tutorial Examples of an Operating System Command Language Specification and a* PL/1 *-like On-Condition Language Definition*, in [Bjørner 78b], pp. 337-374, 1978

[Bjørner 78e] D.Bjørner: *The Vienna Development Method: Software Abstraction and Program Synthesis*, 'Math. Studies of Information Processing', LNCS 75, 1979

[Bjørner 80a] D.Bjørner (ed.): *Abstract Software Specifications* LNCS 86, 1980.

[Bjørner 80b] D.Bjørner *Experiments in Block-Structured GOTO-Modelling:Exits vs. Continuations*, in: [Bjørner 80a] pp. 216-247, 1980.

[Bjørner 80c] D.Bjørner *Formalization of Data Base Models*, in: [Bjørner80a], pp 144-215, 1980.

[Bjørner 80d] D.Bjørner: *Formal Description of Programming Concepts: a Software Engineering Viewpoint*, MFCS'80, LNCS 88, pp. 1-21, 1980.

[Bjørner 80e] D.Bjørner: *Application of Formal Models*, in: 'Data Bases', INFOTECH Proceedings, October 1980.

[Bjørner 80f] D.Bjørner & O.Oest (eds.): *Towards a Formal Description of Ada*, LNCS 98, 1980.

[Bjørner 80g] D.Bjørner, O.Oest: *The DDC Ada Compiler Development Project*, in: [Bjørner 80f], pp.1-19, 1980.

[Bjørner 81a] D.Bjørner: *The VDM Principles of Software Specification and Program Design,* in: 'Formalization of Programming Concepts', LNCS 107, pp. 44-74, 1981.

[Bjørner 82a] D.Bjørner (ed.): *Formal Description of Programming Concepts (2)* IFIP TC-2 Work.Conf., Garmisch-Partkirschen, N-H, 1982.

[Bjørner 82c] D.Bjørner: *Formal Semantics of Data Bases,* 8th Int'l. VLDB (Very Large Data Base) Conf., Mexico City, Sept.8-10, 1982.

[Bjørner 82d] D.Bjørner: *The VDM CHILL and Ada Compiler Developments,* 6th Int'l. Conf. on Software Engineering (ICSE), Tokyo, Japan, Sept. 13-16, 1982.

[Bjørner 83*] D.Bjørner: *Software Architectures & Programming Systems Design,* approx. 1000 pages lect. notes, Dept.of Comp.Sci., Techn.Univ.of Denmark, 1983.

[Bundgaard 80a] J.Bundgaard & L.Schultz: *A Denotational (Static) Semantics Method for Defining Ada Context Conditions,* in: [Bjørner 80f], pp 21-212, 1980.

[Dommergaard 80a] O.Dommergaard: *The Design of a Virtual Machine for Ada* in: [Bjørner 80a], pp. 435-605, 1980.

[Dommergaard 80b] O.Dommergaard, S.Bodilsen: *A Formal Definition of P-Code,* Dept.of Comp.Sci., Techn.Univ.of Denmark, 1980.

[Flensholt 81A] J.Flensholt: *Conceptual Graphs: A Denotational Semantics Approach,* M.Sc. Thesis, Dept.Comp.Sci., Copenhagen Univ., Dec. 1981, 97 pages.

[Folkjär 80a] P.Folkjär & D.Bjørner: *A Formal Model of Generalized CSP-like Language,* IFIP'80 (ed. S.H.Lavington), pp. 95-99, 1980.

[Haff 80a] C.C.I.T.T. Period 1980-1984, Working Party XI/3, Geneva 7.Dec.-17.Dec 1981: *CHILL Formal Definiton* Vol.I: Parts I-II-II, Vol.II: Part IV,Dec.1981.

[Hansal 76a] A.Hansal: *A Formal Definition of a Relational Data Base System,* IBM (Peterlee, UK) UKSC0080, June 1976.

[Henhapl 75a] W.Henhapl, H.Izbicki, C.B. Jones & F.Weissenböck: *Some Experiments with using Formal Definitions in Compiler Development,* Vienna LN25.3.107, Dec.1975.

[Henhapl 78a] W.Henhapl & C.B.Jones: *A Formal Definition of ALGOL 60 as described in the 1975 modified Report,* in: [Bjørner 78b], pp: 305-336.

[Izbicki 75a] H.Izbicki: *On a Consistency Proof of a Chapter of the Formal Definition of a* PL/I *Subset,* Vienna TR 25.142, Feb. 1975.

[Jones 71a] C.B.Jones & P.Lucas: *Proving Correctness of Implementation Techniques*, in [Engeler 71a], pp 178-211 1971.

[Jones 72a] C.B.Jones: *Formal Development of Correct Algorithms: An Example Based on Earley's Recognizer*, SIGPLAN 7, 1, Jan.1972.

[Jones 73a] C.B.Jones: *Formal Development of Programs*, IBM (Hursley, UK) TR 12.117, June 1973.

[Jones 75a] C.B.Jones: *Formal Definition in Program Development*, in: 'Programming Methodology', LNCS 23, pp 387-443, 1975.

[Jones 75b] C.B.Jones: *Yet Another Proof of the Block Concept*, Vienna LN25.3.075, presented at IFIP WG2.2 meeting Boston 1970.

[Jones 76a] C.B.Jones: *Formal Definition in Compiler Development*, Vienna TR25.145, Feb. 1976.

[Jones 76b] C.B.Jones: *Program Development Using Data Abstraction*, presented at IFIP WG2.3 meeting, Grenoble, Dec. 1976

[Jones 77a] C.B.Jones: *Program Specifications and Formal Development*, in: [Morlet 77a], pp 537-554, 1977.

[Jones 77b] C.B.Jones: *Structured Design and Coding: Theory versus Practice*, Informatie, Jaargang 19, 6, pp 311-319 June 1977.

[Jones 78a] C.B.Jones: *The Meta-Language: A Reference Manual*, in: [Bjørner 78b], pp 218-277, 1978.

[Jones 78b] C.B.Jones: *Denotational Semantics of GOTO: an Exit Formulation and its Relation to Continuations*, in: [Bjørner 78b], pp 278-304, 1978.

[Jones 78c] C.B.Jones: *The Vienna Development Method: Examples of Compiler Development*, in: Le Point sur la Compilation (ed.Amirchachy & Neel), IRIA Publ., 1979.

[Jones 79a] C.B.Jones: *Constructing a Theory of a Data Structure as an aid to Program Development*, Acta 11, pp 119-137, 1979.

[Jones 80a] C.B.Jones: *Software Development: A Rigorous Approach*, Prentice-Hall International, London 1980.

[Jones 80b] C.B.Jones: *Models of Programming Language Concepts* in: [Bjørner 80b], 1980.

[Jones 81a] C.B.Jones: *Development Methods for Computer Programmes Including a Notion of Interference* D.Phil. Thesis,PRG 25, June 1981.

[Jones 81b] C.B.Jones: *Towards more Formal Specifications* in: 'Software Engineering - Entwurf und Spezifikation' eds.:C.Floyd,H.Kopetz,B.G.Teubner, TH Berlin, 1981

[Jones 81c] C.B.Jones: *Formal Methods in Software Development* in: INFOTECH State of the Art Review, pp. 107-113, 1981.

[Lindenau 81a] J.Lindenau: *Eine Deskriptive Anfragesprache für das Netzwerk-Datenmodell mit formaler Definition der Semantik in Meta-IV,* M.Sc. Thesis (in German), Kiel Univ., Inst.f.Informatik, March 1981, 175 pages.

[Lucas 73a] P.Lucas: *On Program Correctness and the Stepwise Development of Implementations,* in: Proceedings 'Convegno di Onformatica Teorica', Univ. of Pisa, Italy, pp.219-251, March 1973.

[Lucas 78a] P.Lucas: *On the Formalization of Programming Languages: Early History and Main Approaches.,* in: [Bjørner 78a].

[Lucas 80a] P.Lucas: *On the Structure of Application Programs.* in: [Bjørner 80b], 1980.

[Lucas 81a] P.Lucas: *Formal Semantics of Programming Languages VDL* IBM Journal of Devt. & Res., 25, 5, pp 549-561, 1981.

[Løvengreen 80a] H.H.Løvengreen: *Parallelism in Ada,* in: [Bjørner 80f], pp. 309-432, 1980.

[Løvengreen 80b] H.H.Løvengreen & D.Bjørner: *On a Formal Model of the Tasking Concepts in Ada,* Proc. ACM SIGPLAN Ada Symp., Boston 1980.

[Madsen 77a] J.Madsen: *An Experiment in Formal Definition of Operating System Facilities,* IPL 6, 6, pp 187-189, Dec. 1977.

[Madsen 80a] J.Madsen: *Modular Operating System Design,* Ph.D. Thesis, Dept.of Comp.Sci., Techn.Univ.of Denmark, Rept.no. ID805, 194 pages, Aug. 1980.

[Madsen 81ab] J.Madsen: *A Computer System Supporting Data Abstraction,* Pts. 1-2, SIGOPS 15, 1-2, 1981, pp. 45-72, 38-78, 1981.

[Neuhold 80a] E.Neuhold & Ths.Olnhoff: *The Vienna Development Method (VDM) and its Use for the Specification of a Relational Data Base System,* IFIP'80 (ed.S.Lavington), 1980.

[Neuhold 81a] E.Neuhold & Th.Olnhoff: *Building Data Base Management Systems Through Formal Specifications,* in: LNCS 107, pp 169-209, 1981.

[Nilsson 76a] J.F.Nilsson: *Relational Data Base Systems: Formalization and Realization*, Ph.D.Thesis, Dept. of Comp.Sci., Techn. Univ. of Denmark, ID-641, Sept. 1976.

[Nilsson 81a] J.F.Nilsson: *Formalization and Implementation of PROLOG* Dept.of Comp.Sci., Techn.Univ.of Denmark, 1981.

[Olnhoff 81a] Ths.Olnhoff: *Funktionale Semantikbeschreibung von Anfrageoperationen in einem drei-schictigen relationaler Datenbanksystem*, Ph.D.Thesis, Stuttgart/ Hamburg Univ., Inst.f.Informatik, May 1981, 210 pgs.

[Schmidt 81a] U.Schmidt & U.Völler: *Die Formale Entwicklung der Maschin-Unabhängigen Zwischensprache CAT*, in: GI-1 Jahrestagung, Informatik Fachberichte, S-V, pp 57-64, 1981.

[Storbank 80a] J.Storbank Pedersen: *A Formal Semantics Definition of Sequential Ada*, in: [Bjørner 80f], pp 213-308.

[Weissenböck 75a] F.Weissenböck & W.Henhapl: *A Formal Mapping Description*, Vienna TN25.3.105, Feb. 1975.

[Weissenböck 75b] F.Weissenböck: *A Formal Interface Specification*, Vienna TR25.141, Feb. 1975.

GENERAL AND BACKGROUND BIBLIOGRAPHY

[Bekić 70a] H.Bekić: *On the Formal Definition of Programming Languages*, in: ICS'70, GMD, Bonn, Nov. 1970.

[Bekić 71a] H.Bekić: *Towards a Mathematical Theory of Processes*, Vienna TR25.125, Dec.1971.

[Bekić 71b] H.Bekić & K.Walk: *Formalization of Storage Properties*, in: [Engeler 71a], 1971.

[Burstall 77] R.M.Burstall & J.Darlington: *A transformation System for Developing Recursive Programs*, JACM 24,1, Jan. 1977, pp.44-67

[Dahl 72a] O.-J.Dahl, E.W.Dijkstra & C.A.R.Hoare: *Structured Programming*, Academic Press, 1972.

[Dijkstra 69a] E.W.Dijkstra: *Notes on Structured Programming*, in: [Dahl 72a], pp 1-82, 1972.

[Dijkstra 76a] E.W.Dijkstra: *A Discipline of Programming*, Prentice-Hall, 1976.

[Engeler 77a] E.Engeler: *Symposium on Semantics of Algorithmic Languages*, S-V Lecture Notes in Mathematics, Vol. 188, 1971

[Hoare 72b] C.A.R.Hoare: *Proof of Correctness of Data Representations*, Acta 1, pp 271-281, 1972.

[Hoare 72b] C.A.R.Hoare: *A Note of the 'FOR' Statement*, BIT, 12, pp 334-341, 1972.

[Hoare 73b] C.A.R.Hoare: *Recursive Data Structures*, IJCIS 4, 2, pp. 105-132, 1975.

[Hoare 78a] C.A.R. Hoare: *Communicating Sequential Processes* CACM 21, 8, Aug. 1978.

[Hoare 81a] C.A.R.Hoare: *The Emperor's Old Clothes*, CACM 24,2, Feb. 1981

[Landin 64a] P.J.Landin: *The Mechanical Evaluation of Expressions*, Comp.J. 6, 4, pp 308-320, 1964.

[Landin 66a] P.J.Landin: *The Next 700 Programming Languages*, CACM 9, 3, pp 157-166, 1966.

[Landin 72a] P.Landin, R.M.Burstall: *Programs and their Proofs: An Algebraic Approach*, in: MI-4, 1972.

[McCarthy 62a] J.McCarthy: *Towards a Mathematical Science of Computation*, IFIP'62 (ed. C.M.Popplewell), pp 21-28, 1963.

[McCarthy 63a] J.McCarthy: *A Basis for a Mathematical Theory of Computation*, in: Computer Programming and Formal Systems, N-H, 1963.

[McCarthy 66a] J.McCarthy: *A Formal Description of a Subset of ALGOL*, in: [Steel 66a].

[McCarthy 67a] J.McCarthy & J.Painter: *Correctness of a Compiler for Arithmetic Expressions*, in: [Schwartz 67a], pp. 33-41, 1967.

[Milne 76a] R.Milne & C. Strachey: *A Theory of Programming Language Semantics*, Chapman and Hall, London, Halsted Press/John Wiley, New York, 1976.

[Milner 71a] R.Milner: *Program Simulation: An Extended Formal Notion*, Memos 14 & 17, Univ.of Swansea, Dept.Comp. Sci., 1971.

[Milner 71b] R.Milner: *An Algebraic Definition of Simulation between Programs*, Stanford CS-205, 1971.

[Milner 80a] R.Milner: *Calculus of Communication Systems* LNCS 94, 1980.

[Morlet 77a] E.Morlet & D.ribbens: *International Computing Symposium 77*, European ACM, N-H, 1977

[Morris 70a] F.L.Morris: *The next 700 Programming Language descriptions*, unpubl.ms., Univ. of Essex, Comp.Ctr. 1970.

[Morris 73a] F.L.Morris: *Advice on Structuring Compilers and Proving them Correct,* in: POPL, Boston, pp. 144-152, Oct. 1973.

[Mosses 81a] -- Personal Communication

[Naur 66a] P.Naur: *Program Translation Viewed as a General Da ta Processing Problem,* CACM 9, 3, pp 176-179, 1966

[Naur 66b] P.Naur: *Proof of Algorithms by General Snapshots,* BIT 6, pp 310-316, 1966.

[Parnas 72a] D.L.Parnas: *A Technique for Software Module Specification with Examples,* CACM 14, 5, May 1972.

[Reynolds 70a] J.C.Reynolds: *GEDANKEN - A Simple Type-less Language based on the Principle of Completeness and the Reference Concept,* CACM 13,5, pp 308-319,1970.

[Reynolds 72a] J.C.Reynolds: *Definitional Interpreters for Higher -Order Programming Languages,* Proc.25th ACM Nat'l. Conf., pp. 717-740, 1972.

[Reynolds 74b] J.C.Reynolds: *Towards a Theory of Type Structure* in: Programming Symposium, LNCS 19, pp 408-425, 1974.

[Reynolds 78a] J.C.Reynolds: *Syntactic Control of Interference,* POPL, pp 39-46, New York, 1978.

[Reynolds 78b] J.C.Reynolds: *User defined Types and Procedural Data Structures as Complementary Approaches to Data Abstractions,* in 'Programming Methodology',(ed. D.Gries), S-V, pp 309-317, 1978.

[Richards 72a] M.Richards: *INTCODE an Interpretive Machine Code for BCPL.* The Computer Laboratory, Corn Exchange Street, Cambridge, December 1972.

[Richards 74a] M.Richards: *TheBCPL Programming Manual,* The Computer Laboratory, University of Cambridge, UK, 1974.

[Richards 79a] M.Richards & C.Whitby-Strevens: *BCPL - The Language and its Compiler,* Cambridge University Press, 1979

[Scott 70a] D.S.Scott: *Outline of a Mathematical Theory of Computation,* Proc. 4th. Ann.Princeton Conf.on Inf. Sci.& Sys., pp 169, 1970.

[Scott 71a] D.S.Scott & C.Strachey: *Towards a Mathematical Semantics for Computer Languages,* in: MRI-XXI, 1971.

[Scott 72a] D.Scott: *Mathematical Concepts in Programming Lang uage Semantics,* Proc.AFIPS, Spring Joint Computer Conference, 40, pp 225-234, 1972.

[Scott 76a] D.Scott: *Data Types as Lattices,* SIAM Journal on Computer Science, 5, 3, pp 522-587, 1976.

[Scott 77a] D.Scott: *Logic and Programming Languages*, CACM 20, 9, 634-41, 1977.

[Schwartz 67a] J.T.Schwartz(ed.): *Mathematical Aspects of Computer Science*, American Mathematical Society, R.I. 1977

[Sintzoff 80a] M.Sintzoff: *Suggestions for Composing & Specifying Program Design Decisions*, in '4th Int'l Colloq. o "Programming", LNCS 83, April 1980, Paris.

[Steel 66a] T.B.Steel:(ed.): *Formal Language Description Languages* IFIP TC-2 Work.Conf., Baden, N-H 1966.

[Stoy 77a] J.E.Stoy: *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MI Press, 1977.

[Stoy 80a] J.E.Stoy: *Foundations of Mathematical Semantics*, in: [Bjørner 80b] pp. 43-99.

[Strachey 66a] C.Strachey: *Towards a Formal Semantics*, in: [Steel 66a], pp 198-220.

[Strachey 67a] C.Strachey:.. *Fundamental Concepts in Programming Languages*, unpubl.lect.notes, NATO Summer School Copenhagen, 1967.

[Strachey 73a] C.Strachey: *The Varieties of Programming Languages*, PRG-10, 1973.

[Strachey 74a] C.Strachey & C.Wadsworth: *Continuations: A Mathematical Semantics which can deal with Full Jumps* PRG-11, 1974.

[Tennent 73a] R.D.Tennent: *Mathematical Semantics and the Design of Programming Languages*, Ph.D.Thesis, Univ of Toronto, 1973.

[Tennent 73b] R.D.Tennent: *The Mathematical Semantics of SNOBOL 4*, in: POPL, Boston, pp 95-107, Oct. 1973.

[Tennent 76a] R.D.Tennent: *The Denotational Semantics of Programming Languages*, CACM 19, 8, Aug. 1976.

[Tennent 77a] R.D.Tennent: *A Note on Files in Pascal*, BIT, 17, 362-366, 1977.

[Tennent 77b] R.D.Tennent: *Language Design Methods based on Semantic Principles*, Acta 8, pp 97-112, 1977.

[Tennent 77c] R.D.Tennent: *On a new Approach to Representation-independent Data Classes*, Acta 8, pp 315-324, 1977.

[Tennent 78a] R.D.Tennent: *Another Look at Type Compatibility in Pascal*, Software: Practice & Experience, 8, 1, pp 85-97, 1978.

[Tennent 80a] R.D.Tennent: *Principles of Progamming Languages*, Prentice-Hall, Int'l, 1981.

[Tennent 82a] R.D.Tennent: *The Semantics of Interference* ICALP, LNCS, July 1982.

[van Wijngaarden 62a] A.van Wijngaarden: *Generalized ALGOL*, in: 'Symbolic Languages in Data Processing', Proc. ICC Symp., Rome, Gordon & Breach, N.Y., pp.409-419, 1962.

[van Wijngaarden 63b] A.van Wijngaarden: *Recursive Definition of Syntax and Semantics*, in [Steel 66a] ref. 31, pp 13-24.

[Wand 80a] M.Wand: *Continuation-based Program Transformation Strategies*, JACM 27, pp 164-180, 1980.

[Wand 80bc] M.Wand: *Deriving Target Code As a Representation of Continuation Semantics* and *Different Advice on Structuring Compilers and Proving them Correct*, Indiana State Univ., Bloomington, Dept.of Comp.Sci. Techn.Repts. 94-95, June 1980.

[Wand 82a] M.Wand: *Semantics-Directed Machine Architecture*, POPL, Jan. 1982.

ZŁOŻONOŚĆ OBLICZENIOWA PROBLEMÓW ANALIZY I PROJEKTOWANIA SYSTEMÓW KOMPUTEROWYCH

Jacek Błażewicz
Instytut Automatyki
Politechnika Poznańska
ul. Piotrowo 3a
60-965 Poznań, tel. 782-375

1. Wstęp

W zagadnieniach analizy już istniejących bądź przy projektowaniu nowych systemów komputerowych występuje szereg problemów o naturze kombinatorycznej, to jest, ogólnie mówiąc, problemów dotyczących zbiorów skończonych. Problemy kombinatoryczne można między innymi podzielić na decyzyjne i optymalizacyjne. Problemy decyzyjne są, ogólnie mówiąc, sformułowane w postaci pytania, na które odpowiedź brzmi "tak" lub "nie". Natomiast problemy optymalizacyjne, to takie, w których należy ekstremalizować pewną funkcję celu. Istotnym zagadnieniem przy rozpatrywaniu problemów kombinatorycznych, zwłaszcza w aspekcie ich praktycznego zastosowania w interesującej nas problematyce, jest kwestia ich efektywnej rozwiązywalności, przede wszystkim z punktu widzenia czasu obliczeń, gdyż pamięć nie jest zazwyczaj zasobem krytycznym. Zanim przejdziemy do ścisłego omówienia tego zagadnienia, przedstawimy krótko zarys historii jego rozwoju. Otóż, najbardziej zasadniczym pytaniem z interesującego nas zakresu jest pytanie, czy dla rozpatrywanego problemu istnieje w ogóle jakiś rozwią-

zujący go algorytm , to znaczy ogólnie mówiąc, procedura, która np. w przypadku problemu decyzyjnego dla każdych danych wejściowych tego problemu zapewnia odpowiedź "tak" lub "nie". Jeśli taki algorytm istnieje, to można powiedzieć, że nasz problem jest rozstrzygalny, jeśli nie- że jest nierozstrzygalny. Zauważmy, że jest to na razie rozstrzygnięcie teoretyczne w tym sensie, że znaleziony algorytm może w ogóle nie być realizowalny w rozsądnym czasie i przestrzeni za pomocą istniejących systemów komputerowych. Pojawia się zatem naturalny postulat rozróżniania algorytmów lepszych i gorszych z obliczeniowego punktu widzenia. W dalszym ciągu ograniczymy się tylko do rozróżnienia z punktu widzenia czasu obliczeń, pomijając sprawę zajętości pamięci. Pierwsze ściślejsze sformułowanie powyższego postulatu podał Edmonds w 1965r.[12],nazywając "dobrymi" (efektywnymi) algorytmy dające rozwiązanie w czasie ograniczonym od góry przez wielomian zależny od rozmiaru problemu oraz "złymi" (nieefektywnymi) algorytmy pozostałe. Następnie okazało się, że niektóre problemy decyzyjne znane jako "trudne", to znaczy takie dla których mimo wielu usiłowań nie udało się podać "dobrych" algorytmów, sprowadzają się do siebie nawzajem w taki sposób, że znając rozwiązanie jednego z nich można w wielomianowym czasie skonstruować rozwiązanie innego. Doprowadziło to Cooka w 1971r.[11] a następnie Karpa w 1972r.[18]do zdefiniowania podstawowych klas złożonościowych problemów decyzyjnych. Później, w 1978r. dokonano głębszej analizy problemów "trudnych"[14], wyróżniając wśród nich problemy "szczególnie trudne" (silnie NP-zupełne), oraz problemy "łatwiejsze" (NQL-zupełne)[28].

Sprecyzowano również związki zachodzące między problemami optymalizacyjnymi i decyzyjnymi, pozwalające uogólnić powyższe rozważania także na przypadek kombinatorycznych problemów optymalizacyjnych. Dalsza analiza problemu efektywnej rozwiązywalności okazała się zresztą ciekawsza w przypadku problemów optymalizacyjnych, gdyż nawet jeśli nie można danego problemu optymalizacyjnego rozwiązać efektywnie (co zazwyczaj kończy badania w przypadku problemu decyzyjnego), pozostaje zawsze kwestia konstrukcji efektywnego (czasowo) algorytmu aproksymacyjnego, znajdującego rozwiązanie suboptymalne (przybliżone). Jest to szczególnie istotne zwłaszcza w zastosowaniach praktycznych, w których nie wystarczy stwierdzić, że jakiś problem jest obli-

czeniowo trudny, lecz należy zaproponować metodę jego rozwiązania, uwzględniającą zarówno istniejące ograniczenia czasowe jak i żądaną dokładność (odległość od optimum) rozwiązania. Wynika stąd potrzeba rozwoju teorii algorytmów aproksymacyjnych dla problemów kombinatorycznej optymalizacji, a zwłaszcza oceny dokładności konstruowanych przez nie rozwiązań i to zarówno w najgorszym przypadku jak i średnio.

W pracy przedstawiono zarys teorii złożoności obliczeniowej problemów kombinatorycznych, decyzyjnych i optymalizacyjnych, jak również metodykę rozwiązywania "trudnych" problemów optymalizacji kombinatorycznej. W tym sensie praca ta stanowi uzupełnienie istniejących opracowań w języku polskim [2], [24], [30], zajmujących się zasadniczo konstrukcją algorytmów rozwiązujących "łatwe" problemy kombinatoryczne.

Rozważane w pracy pojęcia teoretyczne zilustrowano przykładem analizy wybranego problemu z zakresu konstrukcji systemów operacyjnych. Przy wyborze problemu kierowano się z jednej strony jego ważnością z drugiej zaś strony stanem badań umożliwiającym możliwie pełną ilustrację omawianych pojęć.

2. Zarys teorii złożoności obliczeniowej problemów kombinatorycznych.

Podstawowe definicje dotyczące złożoności obliczeniowej sformułujemy dla klasy problemów decyzyjnych. Można wskazać wiele problemów z różnych dziedzin, takich jak teoria liczb, teoria grafów, logika, teoria automatów, które z natury są problemami decyzyjnymi. Przykładowo rozpatrzmy problem spełnialności wyrażeń boolowskich, w którym dla danego wyrażenia boolowskiego w normalnej postaci koniunkcyjnej (to znaczy w postaci iloczynów sum logicznych zmiennych boolowskich i ich negacji) pytamy o istnienie przydziału zer i jedynek do zmiennych boolowskich, takiego, że wyrażenie to przyjmuje wartość jeden. Problemem decyzyjnym z dziedziny problemów pakowania jest problem trójpodziału, w którym dla danego ograniczenia B i zbioru A składającego się z 3q elementów o znanych całkowitych wagach, pytamy o istnienie podziału tego zbioru na q rozłącznych

trójelementowych podzbiorów takich, że suma wag w każdym z nich jest równa B.

Jednakże oprócz wymienionych problemów decyzyjnych istnieje także duża klasa problemów, które wymagają ekstremalizacji pewnej funkcji celu. Nazywa się je zatem problemami optymalizacyjnymi. Zauważmy, że obie wymienione klasy problemów mogą być badane w sensie ich "natury obliczeniowej" w ten sam sposób. Jeśli bowiem z danym problemem optynalizacyjnym związżemy problem decyzyjny, w którym pytamy o istnienie rozwiązania o wartości danej funkcji celu $\leqslant$ (w przypadku minimalizacji) lub $\geqslant$ (w przypadku maksymalizacji) od pewnej zadanej z góry wartości, to taki problem decyzyjny jest obliczeniowo nie trudniejszy, niż odpowiadający mu pierwotny problem optymalizacyjny, o ile dla danego rozwiązania problemu optymalizacyjnego można relatywnie prosto obliczyć wartość funkcji celu. Zauważmy bowiem, że jeśli można w "prosty" sposób (ściśle: w wielomianowym czasie) rozwiązać problem optymalizacyjny to można także "prosto" rozwiązać związany z nim problem decyzyjny. Z drugiej strony jeśli problem decyzyjny jest obliczeniowo "trudny" (ściśle: NP-zupełny), to "trudny" jest również odpowiadający mu problem optymalizacyjny. Z rozważań tych wynika, że w celu wykazania "łatwości" problemu decyzyjnego wystarczy wykazać "łatwość" odpowiadającego mu problemu optymalizacyjnego, natomiast dla wykazania "trudności" problemu optymalizacyjnego wystarczy wykazać "trudność" związanego z nim problemu decyzyjnego.

Jak wynika z powyższego rozumowania, wykazując obliczeniową "trudność" problemu wystarczy ograniczyć się do sformułowania decyzyjnego. Z drugiej strony sformułowanie to pozwala na łatwiejsze wprowadzenie podstawowych pojęć teorii złożoności obliczeniowej i łatwiejsza analizę złożoności tak sformułowanego problemu. Rozpoczniemy więc od przedstawienia zasadniczych pojęć tej teorii w odniesieniu do problemów decyzyjnych.

Przez <u>problem decyzyjny</u> Π będziemy rozumieli zbiór parametrów (zbiory, grafy, funkcje, liczby itp.), które nie muszą mieć nadanych wartości oraz pytanie, na które odpowiedź brzmi "tak" lub "nie". Ustalając wartości wszystkich parametrów danego problemu Π, otrzymujemy <u>instancję</u> (<u>konkretny problem</u>), który

oznaczymy przez I. Możema zatem także zdefiniować problem decyzyjny Π jako zbiór instancji D_Π oraz jego podzbiór $Y_\Pi \subseteq D_\Pi$, zawierający wszystkie instancje, na które odpowiedź brzmi "tak".

Dane instancji $I \in D_\Pi$ (tzn. konkretne wartości parametrów problemu Π) zapisuje się (koduje) za pomocą skończonego łańcucha x(I) symboli należących do z góry określonego alfabetu Σ zgodnie z ustaloną regułą kodowania. Przez rozmiar instancji, N(I), rozumieć będziemy długość łańcucha x(I), czyli |x(I)|.

Zauważmy, że wymaganie możliwości zakodowania danych instancji za pomocą skończonego łańcucha symboli jest jedynym ograniczeniem klasy problemów decyzyjnych, objętej niżej przedstawionymi rozważaniami o złożoności obliczeniowej. Jest to jednak, rzecz jasna, ograniczenie czysto teoretyczne, zważywszy że chodzi nam o charakterystykę algorytmów i problemów z punktu widzenia zastosowania rzeczywistych komputerów.

Ogólnie, reguły kodowania winny być jednoznaczne, a także zwięzłe, tzn. nie powinny powodować sztucznego wzrostu rozmiaru instancji. Inaczej mówiąc, reguły te powinny spełniać następujące postulaty:

1. łańcuch symboli kodujących dane instancji nie może zawierać nadmiarowych symboli,
2. liczby występujące w danych powinny być zapisane binarnie lub przy dowolnej innej ustalonej podstawie zliczania większej od 1.

Ogólnie mówiąc, możemy zatem przyjąć każdą "rozsądną" regułę kodowania, która nie powoduje wykładniczego wzrostu rozmiaru kodowanej instancji w stosunku do innych reguł kodowania. Innymi słowy, mając dwie reguły kodowania, które powodują powstanie łańcuchów danych dla instancji I o długościach odpowiednio $N_1(I)$ oraz $N_2(I)$, przy czym $N_1(I) \geqslant k^{N2(I)}$ dla pewnej stałej $k > 1$, musimy odrzucić pierwszą regułę kodowania, gdyż powoduje ona "nierozsądny" (wykładniczy) wzrost rozmiaru kodowanej instancji. Z tego właśnie powodu musimy odrzucić tzw. kodowanie jedynkowe, w którym każdej liczbie całkowitej k odpowiada łańcuch złożony z k jedynek.

Zauważmy, że problemowi decyzyjnemu Π i regule kodowania

e odpowiada <u>język</u> $L(\Pi,e)=\{x(I)\in\Sigma^*: I\in D_\Pi \wedge \Sigma$ jest alfabetem używanym przez e $\wedge$ odpowiedź dla I brzmi "tak"$\}$, gdzie Σ^* jest zbiorem wszystkich skończonych łańcuchów symboli należących do Σ. Natomiast rozwiązaniu problemu decyzyjnego Π odpowiada rozpoznanie języka $L(\Pi,e)$. Na tej podstawie wszystkie przedstawione poniżej definicje można sformułować wykorzystując pojęcie języka związanego z danym problemem decyzyjnym. Dla celów tej pracy wystarczą nam jednak definicje na poziomie problemów decyzyjnych, przy czym , jak łatwo udowodnić, wszystkie rozważania nie będą zależały od konkretnej reguły kodowania, jeśli ograniczymy się tylko do "rozsądnych" reguł kodowania.

Zauważmy także, że zazwyczaj w praktyce dogodnie jest wyrazić rozmiar instancji za pomocą jednego parametru, określającego liczbę elementów zbioru charakterystycznego (znaczącego) dla danych rozpatrywanej instancji. Na przykład w przypadku problemów szeregowania mogłaby to być liczba zadań, a dla problemów określania maksymalnego przepływu w sieciach - liczba wierzchołków sieci. Założenie to, powszechnie przyjmowane, w większości przypadków sprowadza się do założenie, iż słowo maszyny cyfrowej jest na tyle długie, by pomieścić każdą z kodowanych binarnie liczb tworzących dane instancji. Wynika to z faktu, że zazwyczaj ustalona liczba parametrów liczbowych charakteryzuje każdy z elementów należących do wspomnianego powyżej zbioru. Jednakże w przyoadku pewnych problemów, na przykład z teorii grafów, przyjęcie liczby wierzchołków n może się okazać zbyt daleko idącym uproszczeniem, gdyż liczba krawędzi w grafie może być równa $n(n-1)/2$, a co na tym idzie rzeczywistemu rozmiarowi instancji bardziej odpowiadałaby w tym przypadku liczba n^2. Tym nie mniej, w praktyce często przyjmuje się to uproszczenie w celu ujednolicenia otrzymanych wyników dotyczących złożoności obliczeniowej. Zauważmy, że nie powoduje to wykładniczego wzrostu rozmiaru instancji.

Przejdźmy teraz do wprowadzenia pojęcia algorytmu i jego funkcji złożoności obliczeniowej. <u>Algorytmami</u> nazywać będziemy dowolne procedury przeznaczone do rozwiązywania problemów. Mówiąc ściślej - możemy uważać , że są one programami napisanymi w pewnym precyzyjnym języku programowania. Powiemy, że algorytm rozwiązuje problem decyzyjny Π , jeśli znajduje rozwiązania

(odpowiedź "tak" lub "nie") dla każdej instancji $I \in D_\pi$.

W ogólności dążymy do znalezienia algorytmu najefektywniejszego, rozwiązującego dany problem. Pojęcie efektywności może zawierać w sobie wykorzystanie wielu zasobów systemu liczącego, wykonującego dany algorytm. Jednakże najczęściej algorytm najefektywniejszy oznacza algorytm najszybszy, gdyż ograniczenia czasowe przesądzają zazwyczaj o praktycznej przydatności algorytmów. Z tego też powodu, jak również ze względu na fakt, iż analiza złożoności czasowej algorytmów jest najciekawsza z teoretycznego punktu widzenia, w dalszych rozważaniach pominiemy inne zasoby systemu liczącego.

Przed zdefiniowaniem funkcji czasowej złożoności obliczeniowej algorytmu wprowadzimy jeszcze jedno pojęcie.

Powiemy, że funkcja $f(k)$ jest rzędu $g(k)$, co zapisujemy $O(g(k))$, jeżeli istnieje taka stała c, że $|f(k) \leq c|g(k)|$ dla wszystkich wartości k.

Przez <u>funkcję złożoności obliczeniowej</u> algorytmu A rozwiązującego problem będziemy rozumieli funkcję złożoności czasowej, przyporządkowującą każdej wartości rozmiaru instancji $I \in D_\pi$ maksymalną liczbę elementarnych kroków (lub jednostek czasu) maszyny cyfrowej potrzebnych do rozwiązania za pomocą algorytmu A instancji o tym rozmiarze, to znaczy funkcję

$$f_A(n) = \max \{ t : t \text{ jest liczbą elementarnych kroków maszyny cyfrowej niezbędną dla rozwiązania instancji } I \in D_\pi \wedge n = N(I) \} .$$

Oczywiście, funkcja ta nie będzie dobrze określona dopóty, dopóki nie sprecyzujemy reguły kodowania i modelu maszyny cyfrowej (listy elementarnych kroków). Jednakże, jak się okazuje, wybór konkretnej "rozsądnej" reguły kodowania i realistycznego modelu maszyny cyfrowej nie ma wpływu na rozróżnienie czynione pomiędzy algorytmami wielomianowymi i wykładniczymi, czyli dwoma zasadniczymi typami algorytmów z punktu widzenia złożoności obliczeniowej. Przez realistyczne modele maszyny cyfrowej rozumieć będziemy takie, które w jednostce czasu mogą wykonać liczbę

kroków ograniczona od góry przez wielomian od rozmiaru rozwiązywanej instancji. Nie trudno zauważyć, że model taki jest łatwo int-erpretowany w sensie właściwości obliczeniowych w odniesieniu do komputerów rzeczywistych. Wśród modeli realistycznych wyróżnia się m.in. jednotaśmową deterministyczną maszynę Turinga (DTM), k taśmową deterministyczną maszynę Turinga (k-DTM) oraz model RAM. Ciekawe jest porównanie prędkości omówionych modeli. Okazuje się, że wszystkie modele są sobie równoważne w tym sensie, że jeśli dana instancja jest rozwiązywana przez jakiś model maszyny cyfrowej w czasie ograniczonym od góry przez wielomian od jej rozmiaru, to jest ona również rozwiązywana w czasie ograniczonym od góry przez wielomian od jej rozmiaru przez każdy inny model, przy założeniu logarytmicznego kryterium kosztów. W celu uzyskania tej równoważności należy jedynie założyć, że czas wykonania elementarnej operacji dla modelu RAM zależy liniowo od długości łańcucha danych kodujących liczby, na których wykonywana jest operacja, zatem od sumy logarytmów tych liczb. Jak z tego wynika, w ramach analizy równoważności algorytmów w sensie wielomianowej złożoności obliczeniowej możemy posługiwać się dowolnym modelem obliczeń spośród przedstawionych powyżej [1].

Zdefiniujemy teraz dwa najważniejsze z punktu widzenia złożoności obliczeniowej typy algorytmów.

Algorytmem wielomianowym nazywamy algorytm, którego funkcja złożoności obliczeniowej (złożoność obliczeniowa) jest $O(p(k))$, gdzie p jest pewnym wielomianem, a k rozmiarem rozwiązywanej instancji. Każdy algorytm , którego funkcja złożoności obliczeniowej nie może być tak ograniczona, nazywa się algorytmem wykładniczym [1)].

Jak wspomnieliśmy, począwszy od pracy Edmondsa algorytmy wielomianowe przyjęło się nazywać efektywnymi z punktu widzenia złożoności obliczeniowej, algorytmy zaś wykładnicze- nieefektywnymi. O tym jak uzasadnione jest to rozróżnienie zwłaszcza dla dużych rozmiarów instancji świadczy poniższa tablica. Pokazuje ona zależność czasu obliczeń (przy założeniu że elementarny krok maszyny cyfrowej trwa 1 μs) od rozmiaru rozwiązywanej

[1)] Funkcja złożoności obliczeniowej tych algorytmów nie musi być funkcją wykładniczą.

instancji dla różnych funkcji złożoności obliczeniowej.

Tablica 1.
Porównanie kilku wielomianowych i wykładniczych funkcji złożoności obliczeniowej

Funkcja złożoności obliczeniowej	Rozmiar n	
	10	60
n	0.00001s.	0.00006s.
n^3	0.001s.	0.216s.
n^5	0.1s.	13.0min.
2^n	0.001s.	366 stuleci
3^n	0.059s.	1.3×10^{13} stuleci

Obecnie możemy przejść do najistotniejszego fragmentu teorii złożoności obliczeniowej, mianowicie do wyodrębnienia klas złożonościowych problemów decyzyjnych.

Klasę P tworzą wszystkie problemy decyzyjne, które w co najwyżej wielomianowym czasie rozwiązuje deterministyczna maszyna Turinga (a zatem dla problemów należących do klasy P istnieją rozwiązujące je algorytmy wielomianowe).

Warto zauważyć, że istnieje liczna klasa problemów, dla których nie są znane algorytmy wielomianowe, lecz dla których można w co najwyżej wielomianowym czasie zweryfikować pozytywną odpowiedź, jeśli zostanie dostarczona pewna dodatkowa informacja. Aby ująć te rozważania bardziej formalnie, będziemy posługiwać się pojęciem niedeterministycznej maszyny Turinga.

Przez niedeterministyczną maszynę Turinga (NDTM) będziemy rozumieć DTM wyposażoną dodatkowo w moduł generujący. Program dla NDTM jest określany w ten sam sposób, co dla DTM. Wykonanie programu przez NDTM dla łańcucha x(I) , stanowiącego dane rozwiązywanej instancji I, różni się jednak od wykonania programu przez DTM, gdyż przebiega w dwóch etapach. W etapie pierwszym moduł generujący generuje w całkowicie dowolny sposób i zapisuje na taśmie w komórkach -1,-2,..., łańcuch S symboli. W etapie drugim NDTM sprawdza, w taki sam sposób, w jaki jest wykonywany

program przez DTM, czy wygenerowany łańcuch S spełnia warunki określone w pytaniu instancji I. Zauważmy, że dla danej instancji I może istnieć wiele łańcuchów. Powiemy, że NDTM rozwiązuje problem decyzyjny Π, gdy dla każdego $I \in D_\Pi$ są spełnione dwa poniższe warunki:

1. jeśli odpowiedź dla I brzmi "tak", to zostanie wygenerowany łańcuch S, który wraz z x(I) spowoduje, że po wykonaniu programu przez NDTM, maszyna ta osiągnie stan końcowy akceptacji,
2. jeśli odpowiedź brzmi "nie", to albo NDTM osiągnie stan końcowy nieakceptacji, albo faza sprawdzania nie zakończy się w skończonym czasie dla każdego wygenerowanego łańcucha S.

Będziemy mówili, że NDTM rozwiązuje problem decyzyjny Π w co najwyżej wielomianowym czasie, jeśli dla każdej instancji $I \in D_\Pi$, dla której odpowiedź brzmi "tak", zostanie wygenerowany taki łańcuch S, że czas wykonania etapu sprawdzania zakończonego odpowiedzią "tak" przez NDTM (dla I oraz S) jest $O(p(N(I))$, dla pewnego wielomianu p.

Możemy teraz zdefiniować klasę NP problemów decyzyjnych jako zawierającą wszystkie problemy decyzyjne, które w co najwyżej wielomianowym czasie rozwiązuje niedeterministyczne maszyna Turinga.[1)]

Z definicji tej wynika, że $P \subseteq NP$.

W celu zdefiniowania najbardziej ineteresującej klasy problemów decyzyjnych, tzw. problemów NP-zupełnych, wprowadza się definicję transformacji wielomianowej.

Transformacją wielomianową problemu Π_2 do problemu Π_1 (co zapisujemy $\Pi_2 \propto \Pi_1$) nazywamy funkcję $f: D_{\Pi_2} \to D_{\Pi_1}$, która spełnia następujące warunki:

1. Dla każdej instancji $I_2 \in D_{\Pi_2}$ odpowiedź brzmi "tak" wtedy i tylko wtedy, gdy dla instancji $f(I_2)$ odpowiedź brzmi również "tak".

---

1) Można wykazać [15], że każdą niedeterministyczną maszynę Turinga rozwiązującą w co najwyżej wielomianowym czasie problem decyzyjny Π można zasymulować na deterministycznej maszynie Turinga rozwiązującej ten problem w czasie ograniczonym od góry przez $2^{p(N(I))}$ dla pewnego wielomianu p, gdzie $I \in D_\Pi$.

2. Czas obliczenia funkcji f przez DTM dla każdej instancji $I_2 \in D_{\pi_2}$ jest ograniczony od góry przez wielomian od $N(I_2)$.

Mówimy, że problem decyzyjny π_1 jest NP-zupełny, jeśli $\pi_1 \in NP$ i dla każdego innego problemu decyzyjnego $\pi_2 \in NP$, $\pi_2 \propto \pi_1$.

Z powyższych definicji wynika, że gdyby istniał algorytm wielomianowy rozwiązujący jakikolwiek problem NP-zupełny, wówczas każdy problem z klasy NP, a więc także każdy problem NP-zupełny, mógłby zostać rozwiązany za pomocą algorytmu wielomianowego. Ponieważ do klasy problemów NP-zupełnych należą klasycznie trudne problemy kombinatoryczne, dla których mimo wielu usiłowań nie udało się podać algorytmu wielomianowego, prawdopodobnie wszystkie problemy NP-zupełne można rozwiązać tylko przy użyciu algorytmów wykładniczych. Oznaczałoby to, że klasa P jest właściwą podklasą klasy NP, a ponadto, że klasy problemów P i NP-zupełnych są rozłączne [1].

Z definicji tych wynika również, że dla wykazania NP-zupełności badanego problemu decyzyjnego wystarczy przetransformować do niego wielomianowo dowolny znany problem NP-zupełny. Oczywiście, najpierw trzeba było podać pierwszy taki problem, wykazując, że transformują się do niego wielomianowo wszystkie problemy z klasy NP. Uczynił to Cook w cytowanej już pracy, w odniesieniu do sformułowanego już problemu SPEŁNIALNOŚCI WYRAŻEŃ BOOLOWSKICH. Aktualnie lista problemów NP-zupełnych obejmuje już kilka tysięcy problemów z różnych dziedzin. Wybór problemu NP-zupełnego, który staramy się przetransformować wielomianowo do badanego problemu, w celu wykazania jego NP-zupełności, chociaż teoretycznie dowolny, ma jednak istotny wpływ na sposób konstrukcji transformacji wielomianowej. Dowody te wymagają zatem dobrej orientacji w zakresie znanych problemów NP-zupełnych, zwłaszcza charakterystycznych dla poszczególnych pokrewnych dziedzin.

W świetle powyższych rozważań można powiedzieć, że do

---

[1] Można wykazać, że jeśli $P \neq NP$, to musi istnieć klasa problemów decyzyjnych $NPI \subset NP$ o złożoności "pośredniej" między klasami problemów NP-zupełnych i P, taka że $NPI \neq P$ i $NPI \neq$ klasa problemów NP-zupełnych.

klasy problemów NP-zupełnych należą problemy najtrudniejsze w klasie NP, które są rozwiązywane przez nieefektywne algorytmy wykładnicze. Okazuje się jednak, że niektóre problemy NP-zupełne można dla spotykanych w praktyce danych rozwiązać przy stosunkowo niewielkim nakładzie czasowym, wykorzystując fakt, że funkcja złożoności obliczeniowej rozwiązujących je algorytmów jest ograniczona od góry przez wielomian zależacy od dwóch zmiennych: rozmiaru instancji N(I) i maksymalnej wartości występujących w tym problemie liczb max(I) (stąd nazwa takich algorytmów - algorytmy pseudowielomianowe). Ponieważ w praktyce max(I) przyjmuje skończone wartości, algorytmy te mają korzystne właściwości z punktu widzenia czasu obliczeń. Nie są to jednak, podkreślmy, algorytmy wielomianowe, gdyż wszystkie liczby są kodowane przy podstawie zliczania większej od 1, zatem długość łańcucha symboli użytego do zakodowania max(I) wynosi log max(I) i złożoność algorytmu wielomianowego byłaby $O(p(N(I),\log\max(I)))$, a nie $O(p(N(I),\max(I)))$, dla danego wielomianu p. Jest także rzeczą oczywistą, że algorytmy pseudowielomianowe można ewentualnie skonstruować jedynie dla liczbowych problemów decyzyjnych Π, tzn. takich, dla których nie istnieje wielomian p, taki że $\max(I) \leq p(N(I))$ dla każdego $I \in D_\Pi$. Do problemów liczbowych należą między innymi problemy: PLECAKOWY, KOMIWOJAŻERA, PODZIAŁU ZBIORU NA TROJELEMENTOWE PODZBIORY (tzw. PROBLEM TROJPODZIAŁU), PROBLEM PLECAKOWY. Nie są natomiast problemami liczbowymi na przykład PROBLEMY : SPEŁNIALNOSCI, MAKSYMALNEGO SKOJARZENIA czy KOLOROWANIA GRAFU [1]. W danych tych problemów nie występują bowiem w ogóle parametry liczbowe, a zatem nierówność $\max(I) \leq p(N(I))$ jest zawsze spełniona. Można także wykazać, że nie dla wszystkich NP-zupełnych liczbowych problemów decyzyjnych istnieje algorytmy pseudowielomianowe.

Powyższe rozważania wskazują na potrzebę głębszej charakteryzacji problemów NP-zupełnych, polegającej na wyodrębnienie tzw. problemów silnie NP-zupełnych [14].

Dla dowolnego problemu decyzyjnego Π i dowolnego wielomianu p określonego dla liczb całkowitych niech Π_p oznacza

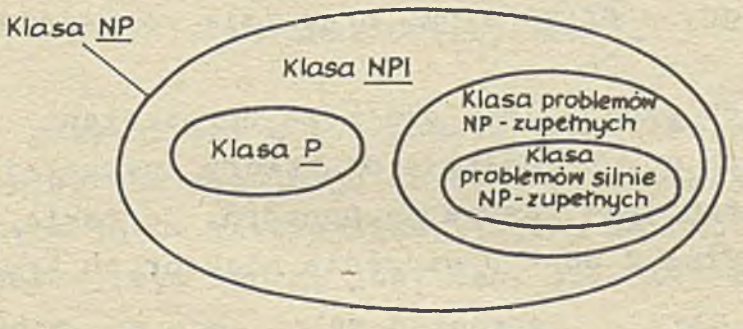
---

[1] Dużymi literami oznaczać będziemy , w przeciwieństwie do problemów optymalizacyjnych, problemy decyzyjne.

podproblem Π otrzymany przez ograniczenie D_Π tylko do tych konkretnych problemów, dla których $max(I) \leq p(N(I))$. Zatem Π_p nie jest problemem liczbowym.

Problem decyzyjny Π jest silnie NP-zupełny, jeśli Π należy do NP i istnieje wielomian p określony dla liczb całkowitych, dla którego Π_p jest NP-zupełny.

Na mocy tej definicji, jeśli problem Π jest NP-zupełny i nie jest problemem liczbowym, to jest silnie NP-zupełny, oraz jeśli problem Π jest silnie NP-zupełny, to istnienie dla niego algorytmu pseudowielomianowego jest równoważne istnieniu algorytmów wielomianowych dla wszystkich problemów NP-zupełnych, czyli równości P=NP , jest więc równie mało prawdopodobne. Zakładając zatem, że P≠NP , możemy obrazowo przedstawić klasę NP problemów decyzyjnych, jak na rys.1. Należy podkreślić, że jest możliwy szczegółowszy podział klasy NP ; subtelniejsze rozróżnienia są aktualnie badane



Rys.1. Struktura klasy NP problemów decyzyjnych.

W pracy [14] wykazano, że problemami liczbowymi silnie NP-zupełnymi są na przykład problemy: KOMIWOJAŻERA i TRÓJPODZIAŁU.

Jak wynika z powyższych rozważań analiza złożoności obliczeniowej daje odpowiedź na pytanie, czy analizowany problem (decyzyjny bądź optymalizacyjny) może być rozwiązany (w sensie znalezienia rozwiązania optymalnego w przypadku problemów optymalizacyjnych) w wielomianowym czasie. Pozytywna odpowiedź na to pytanie wynika z faktu znalezienia wielomianowego algorytmu optymalizacyjnego, którego przydatność zależy od rzędu jego funkcji złożoności obliczeniowej i rodzaju zastosowania. Czasami, gdy

funkcja złożoności obliczeniowej, określająca jak wiadomo zapotrzebowanie algorytmu na czas w najgorszym przypadku, posiada zbyt duży rząd, może się okazać, że średnie zachowanie się algorytmu jest wystarczające. Problem ten przedyskutowano między innymi w [1,2]. Z drugiej strony, jeśli odpowiedź jest negatywna to znaczy gdy decyzyjna wersja badanego problemu jest NP-zupełna, wówczas istnieje kilka dróg dalszej analizy.

Po pierwsze, można osłabić (uprościć) niektóre założenia (ograniczenia) przyjęte w problemie pierwotnym i następnie rozwiązywać powstały w ten sposób problem. Rozwiązanie tego ostatniego jest zazwyczaj dobrym przybliżeniem rozwiązania problemu oryginalnego. Uproszczenie założeń może w przypadku na przykład problemu szeregowania polegać na:

- zezwoleniu na przerywanie wykonywania zadań, gdy oryginalny problem dotyczył zadań niepodzielnych ;
- przyjęciu jednostkowych czasów wykonywania zadań, gdy w oryginalnym problemie rozpatrywano dowolne czasy wykonywania;
- założeniu pewnych typów grafów ograniczeń kolejnościowych, np drzew lub łańcuchów, gdy w oryginalnym problemie rozpatrywano dowolne grafy, itp.

W systemach komputerowych szczególnie pierwsze uproszczenie jest uzasadnione w przypadkach gdy równoległe procesory korzystają ze wspólnej pamięci operacyjnej. Zauważmy ponadto, że takie uproszczenie jest korzystne z punktu widzenia niektórych kryteriów.

Po drugie, rozwiązując trudne problemy optymalizacyjne, bardzo często wykorzystuje się algorytmy aproksymacyjne, które nie zawsze znajdują rozwiązanie optymalne. Oczywistym warunkiem koniecznym stosowania takich algorytmów jest niskiego rzędu funkcja złożoności obliczeniowej. Warunek dostateczny wynika z oceny odległości wartości konstruowanych przez nie rozwiązań od optimum. Ocena ta może dotyczyć zachowania się w najgorszym przypadku lub zachowania się przeciętnego (średniego). W celu większej precyzji podamy kilka definicji. Rozpoczniemy od analizy najgorszego przypadku [15,4].

Jeśli Π jest problemem minimalizacji (maksymalizacji), a I jest jego dowolną instancją , to możemy zdefiniować stosunek

$S_A(I)$ dla algorytmu aproksymacyjnego A jako

$$S_A(I) = \frac{A(I)}{OPT(I)} \qquad \left(S_A(I) = \frac{OPT(I)}{A(I)}\right),$$

gdzie A(I) jest wartością rozwiązania skonstruowanego przez algorytm A dla instancji I, a OPT(I) jest wartością rozwiązania optymalnego dla I. Bezwzględne oszacowanie S_A algorytmu aproksymacyjnego A zastosowanego do rozwiązania problemu Π definiuje się jako:

$$S_A = \inf\{r \geqslant 1;\ S_A(I) \leqslant r \text{ dla każdej instancji } I \in D_\Pi\}.$$

Natomiast asymptotycznym oszacowaniem algorytmu A rozwiązującego problem Π nazywać będziemy wartość S_A^∞ zdefiniowaną następująco

$$S_A^\infty = \inf\{r \geqslant 1 : \text{dla pewnej całkowitej, dodatniej liczby } N,\ S_A(I) \leqslant r \text{ dla każdej instancji } I \in D_\Pi \text{ spełniającej warunek } OPT(I) \geqslant N\}.$$

Uzyskaliśmy w ten sposób jednoznaczną miarę "dobroci" algorytmu aproksymacyjnego zarówno w przypadku problemów minimalizacyjnych jak i maksymalizacyjnych. Im S_A (lub S_A) jest bliższe jedności, tym lepszy jest dany algorytm aproksymacyjny. Zauważmy ponadto, że mogą zajść przypadki, w których powyższe oszacowania nie będą równe. Musi wówczas oczywiście zachodzić $S_A > S_A^\infty$ [1]. Z drugiej strony dla pewnych optymalizacyjnych problemów kombinatorycznych pokazano, że znalezienie algorytmu aproksymacyjnego o określonej dokładności jest mało prawdopodobne (to znaczy zagadnienie to jest tak samo trudne jak znalezienie algorytmu wielomianowego dla dowolnego problemu NP-zupełnego).

Analiza zachowania się algorytmu aproksymacyjnego w najgorszym przypadku powinna być uzupełniona analizą jego zachowania się przeciętnego. Analizy tej można dokonać dwiema metodami.

---

[1] Rozpatruje się kilka możliwości zachowania się w najgorszym przypadku algorytmów aproksymacyjnych, dla których $S_A = 1$. Kwestia ta omówiona została dokładnie w [15,4].

Pierwsza polega na założeniu, że parametry instancji rozpatrywanego problemu Π generowane są z pewnego rozkładu prawdopodobieństwa F i następnie na analitycznym analizowaniu średniego zachowania się algorytmu A. Można tu wyróżnić błąd bezwzględny algorytmu aproksymacyjnego, który definiuje się jako różnicę między wartościami rozwiązania przybliżonego a optymalnego oraz błąd względny definiowany jako stosunek powyższych wartości. Asymptotyczne rezultaty w silniejszym (bezwzględnym) sensie są rzadkie. Z drugiej strony, asymptotyczne wyniki dotyczące błędu względnego można uzyskać w stosunkowo łatwiejszy sposób [4,18,26,29].

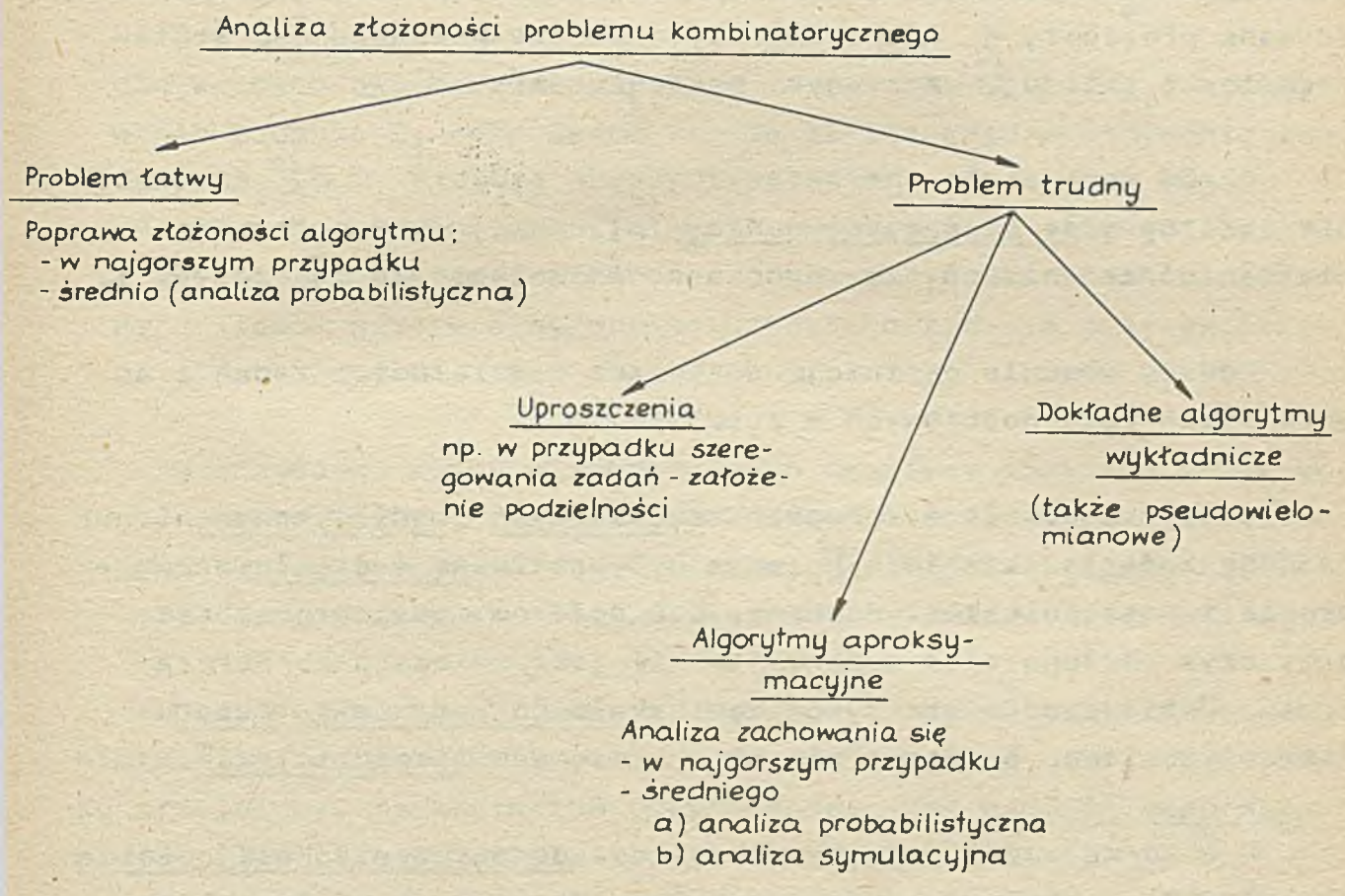
Jest raczej oczywiste, że zachowanie się przeciętne algorytmu może być znacznie lepsze niż zachowanie się w najgorszym przypadku, co usprawiedliwia używanie danego algorytmu aproksymacyjnego. Jednakże trudności związane z wykazaniem tego pierwszego oszacowania, ograniczają jego praktyczne wykorzystanie. Stąd, badania symulacyjne podające również informację o zachowaniu się przeciętnym algorytmów aproksymacyjnych są nadal często stosowane. W tym ostatnim podejściu porównuje się rozwiązania, w sensie wartości kryterium, konstruowane odpowiednio przez algorytm aproksymacyjny i optymalizacyjny, przy czym porównanie to powinno dotyczyć szerokiej, reprezentatywnej próbki instancji. Z powyższego stwierdzenia wynikają pewne zagadnienia praktyczne, które przedyskutowano w [4].

Wreszcie po trzecie, rozwiązując trudne optymalizacyjne problemy kombinatoryczne, można stosować dokładne algorytmy wykładnicze. Jak wspomniano, jeśli analizowany problem (jego decyzyjny odpowiednik) nie jest silnie NP-zupełny, to możliwe jest rozwiązanie go za pomocą algorytmu pseudowielomianowego.

Powyższą dyskusję schematycznie podsumowano na rysunku 2.

W celu ilustracji omówionych w pracy pojęć, w rozdziale następnym przeanalizujemy szczegółowo przykładowy problem szeregowania powstający przy konstrukcji systemu operacyjnego. Przy wyborze problemu kierowaliśmy się z jednej strony jego ważnością, z drugiej zaś aktualnym stanem badań umożliwiającym możliwie

pełną ilustrację omawianych zagadnień [1].



Rys.2. Schemat analizy problemu optymalizacji kombinatorycznej.

3. Analiza przykładowego problemu szeregowania

Jak wspomniano , w rozdziale tym przeanalizujemy z punktu widzenia złożoności obliczeniowej deterministyczny problem szeregowania zadań. Problem ten formułuje się w następujący sposób.

W ogólności rozpatrywać będziemy zbiór n zadań $\mathcal{Z} = \{Z_1, Z_2, \ldots, Z_n\}$ i zbiór m procesorów $\mathcal{P} = \{P_1, P_2, \ldots, P_m\}$. Rozpatry-

---

[1] Czytelnika zainteresowanego analizą innych praktycznych problemów powstających przy analizie i projektowaniu systemów komputerowych odsyłamy do pracy [4].

wać będziemy procesory równoległe, to znaczy takie, które spełniają te same funkcje i pracują równolegle. Wynika z tego, iż każde zadanie może być wykonane przez dowolny procesor. Rozpatrywane procesory są identyczne , to znaczy wykonują wszystkie zadania ze zbioru $\mathcal{Z}$ z równymi prędkościami.

Wśród parametrów charakteryzujących zadanie $Z_j \in \mathcal{Z}$ szczególnie istotny jest czas wykonywania p_j, podający zapotrzebowanie danego zadania na czas procesora (dowolnego ze zbioru $\mathcal{P}$).

Podamy obecnie definicje dotyczące podzielności zadań i ograniczeń kolejnościowych w zbiorze $\mathcal{Z}$.

Będziemy mówili o zadaniach podzielnych , gdy wykonywanie każdego zadania ze zbioru $\mathcal{Z}$ może być przerwane w dowolnym momencie i następnie kontynuowane, być może na innym procesorze, przy czym obsługa tego przerwania nie jest związana ze stratą czasu. Jeśli przerwanie wykonywania każdego zadania ze zbioru jest niemożliwe, będziemy mówili o zadaniach niepodzielnych.

W zbiorze zadań mogą być określone ograniczenia kolejnościowe : $Z_i \prec Z_j$ oznacza, że wykonywanie zadania Z_i musi zostać zakończone przed rozpoczęciem wykonywania zadania Z_j; inaczej mówiąc, zbiór $\mathcal{Z}$ jest częściowo uporządkowany przez relację binarną $\prec$. Jeśli w zbiorze zadań istnieje przynajmniej jedno wyżej określone ograniczenie kolejnościowe, to będziemy mówili o zbiorze zadań zależnych, w przeciwnym razie - o zbiorze zadań niezależnych. Zbiory zadań z określonymi ograniczeniami kolejnościowymi przedstawia się zwykle w postaci grafów skierowanych (digrafów), na ogół w przedstawieniu wierzchołkowym, w którym wierzchołki reprezentują zadania, a łuki skierowane - ograniczenia kolejnościowe.

Podamy teraz definicje związane z pojęciem uszeregowania oraz kryteriami optymalności oceny uszeregowań.

Uszeregowaniem będziemy nazywali takie przyporządkowanie w czasie procesorów ze zbioru $\mathcal{P}$ do zadań ze zbioru $\mathcal{Z}$, dla którego spełnione są następujące warunki:

- w każdej chwili każdy procesor wykonuje co najwyżej jedno zadanie,
- zadanie Z_j jest wykonywane w przedziale czasowym $[r_j, \infty)$.
- wszystkie zadania są wykonane,
- dla każdych dwóch zadań Z_i, Z_j takich, że $Z_i \prec Z_j$, wykonywanie zadania Z_j rozpoczyna się po zakończeniu wykonywania zadania Z_i,
- w przypadku zadań niepodzielnych - wykonywanie żadnego zadania w zbiorze $\mathcal{Z}$ nie jest przerywane, a w przypadku zadań podzielnych liczba przerw w wykonywaniu każdego zadania jest skończona.

W przypadku zadań niepodzielnych będziemy mówili o uszeregowaniu niepodzielnym, a w przypadku zadań podzielnych - o uszeregowaniu podzielnym. W danym uszeregowaniu, dla zadania Z_j, $j=1,2,\ldots,n$ możemy określić moment zakończenia wykonywania C_j.

Uszeregowaniem optymalnym będziemy nazywać uszeregowanie minimalizujące wybrane kryterium optymalności. W naszym przypadku przyjmiemy, że kryterium optymalności jest długość uszeregowania (lub czas wykonania zbioru zadań)

$$C_{max} = \max \{C_j\}.$$

Praktyczną interpretację założeń i wyników jak również motywacje studiowania deterministycznego problemu szeregowania zadań wyjaśniono w książkach [6,5], gdzie rozpatrzono również uogólnienia modelu przedstawionego powyżej.

Jak wspomnieliśmy, przeanalizujemy problem minimalizacji długości uszeregowania dla zadań niezależnych i procesorów równoległych, identycznych. Rozpoczniemy od analizy złożoności obliczeniowej problemu w przypadku zadań niepodzielnych. Wykażemy NP-zupełność odpowiadającego mu problemu decyzyjnego już w przypadku dwóch procesorów.

Twierdzenie 1.

Decyzyjny problem szeregowania zadań niepodzielnych i niezależnych, o dowolnych czasach wykonywania na dwóch identycznych

procesorach, z kryterium długości uszeregowania jest NP-zupełny.

Dowód

Wykażemy NP-zupełność problemu decyzyjnego Π_1, odpowiadającego rozpatrywanemu problemowi szeregowania [1]. W tym celu należy udowodnić, że $\Pi_1 \in \underline{NP}$, oraz $\Pi_2 \propto \Pi_1$, dla pewnego NP-zupełnego problemu Π_2.

Zacznijmy od pierwszej części dowodu. Aby udowodnić przynależność problemu Π_1 do klasy NP należy wykazać, że każda instancja tego problemu zostanie rozwiązana przez niedeterministyczną maszynę Turinga (NDTM) w czasie ograniczonym od góry przez wielomian od rozmiaru tej instancji. Instancja problemu decyzyjnego ma przy tym te same dane, co odpowiednia instancja problemu szeregowania oraz parametr y_1, dla którego pytamy o istnienie uszeregowania o wartości $C_{max} \leq y_1$. Rola modułu generującego NDTM polega na wygenerowaniu łańcucha symboli, określających numer procesora $P_{i(j)}$ i moment rozpoczęcia wykonywania s_j dla zadania Z_j, $j=1,2,\ldots,n$. (Łańcuchów takich może być oczywiście nieskończenie wiele i nie wszystkie muszą odpowiadać uszeregowaniom.) Następnie deterministyczna część NDTM musi dokonać sprawdzenia, czy istnieje łańcuch, dla którego odpowiedź (biorąc pod uwagę rozwiązywaną instancję) brzmi "tak". W rozpatrywanym przypadku należy zatem wykonać następujące punkty.

1. Dla każdych k i l takich, że $1 \leq k < l \leq n$ sprawdź, czy $P_{i(k)} \neq P_{i(l)}$, czy $s_k + p_k \leq s_l$ lub $s_l + p_l \leq s_k$. W ten sposób odrzucane są uszeregowania, w których dwa lub więcej zadań jest wykonywanych na tym samym procesorze w tej samej chwili.
2. Dla każdego k, $1 \leq k \leq n$, sprawdź czy $s_k + p_k \leq y_1$, to znaczy czy wszystkie zadania zostaną zakończone przed upływem chwili y_1.

Można łatwo sprawdzić, że deterministyczna część NDTM może wykonać powyższe punkty w czasie ograniczonym od góry przez wielomian od rozmiaru instancji, a zatem problem $\Pi_1 \in \underline{NP}$.

[1] Korzystając z uwag poczynionych w rozdziale 2, przy obliczaniu złożoności obliczeniowej algorytmów wielomianowych będziemy zakładać, że zapisanie liczby, odjęcie i porównanie dwóch liczb są operacjami elementarnymi maszyny cyfrowej i trwają taki sam kwant czasu (model RAM).

Przejdziemy teraz do drugiej części dowodu. Jako NP-zupełny problem Π_2, który przetransformujemy wielomianowo do problemu Π_1, wybierzemy PROBLEM PODZIAŁU ZBIORU, sformułowany poniżej.

PODZIAŁ ZBIORU

Parametry : Skończony zbiór $C = \{c_1, c_2, \ldots, c_q\}$ oraz waga $s(c_i)$ związana z każdym elementem $c_i \in C$.

Pytanie : Czy istnieje podzbiór $C' \subset C$ taki, że

$$\sum_{c_i \in C'} s(c_i) = \sum_{c_i \in C-C'} s(c_i) \quad ?$$

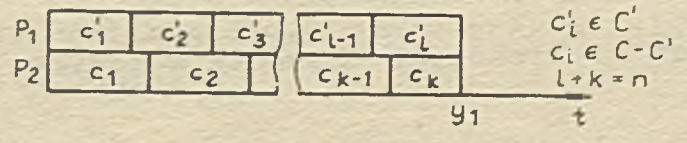
Mając daną instancję $I_2 \in D_{\Pi_2}$, można utworzyć dane odpowiadającej jej instancji $f(I_2) = I_1 \in D_{\Pi_1}$ w następujący sposób:

- zbiór n=q niepodzielnych i niezależnych zadań o czasach wykonywania $p_j = s(c_j)$ $j=1,2,\ldots,n$,
- dwa identyczne procesory,
- wartość funkcji kryterialnej równa $y_1 = \frac{1}{2} \sum_{c_i \in C} s(c_i)$.

Wykażemy teraz, że dla każdej instancji $I_2 \in D_{\Pi_2}$ odpowiedź brzmi "tak" wtedy i tylko wtedy, gdy dla odpowiadającej jej instancji $I_1 \in D_{\Pi_1}$ odpowiedź brzmi również "tak".

Załóżmy najpierw, że odpowiedź dla instancji $I_2 \in D_{\Pi_2}$ brzmi "tak", to znaczy istnieje podzbiór $C' \subset C$ o żądanej właściwości. Wówczas zadania odpowiadające elementom wchodzącym w skład podzbioru C' mogą być wykonywane na procesorze P_1, a pozostałe na procesorze P_2, co zilustrowano na rysunku 3. Ponieważ założenia

$$\sum_{c_i \in C'} s(c_i) = \sum_{c_i \in C-C'} s(c_i), \quad \text{zatem } C_{max} = \frac{1}{2} \sum_{c_i \in C} s(c_i) = y_1.$$



Rys.3.

Załóżmy teraz, że dla instancji $I_1 \in D_{\Pi_1}$ istnieje uszerego-

wanie długości y_1. Ponieważ $\sum_{j=1}^{n} p_j = \sum_{c_i \in C} s(c_i) = 2y_1$, zatem uszeregowanie to musi mieć postać przedstawioną na rysunku 3. Z założenia o niepodzielności zadań wynika, że elementy zbioru C odpowiadające zadaniom wykonywanym na procesorze P_1 tworzą podzbiór C' o żądanej właściwości.

Zauważmy teraz, że czas konstrukcji danych instancji I_1 jest ograniczony od góry przez wielomian zależny od liczby elementów q zbioru C, czyli od rozmiaru instancji $I_2 \in D_{\pi_2}$. Twierdzenie zostało zatem udowodnione.

□

Z powyższego twierdzenia wynika, że nie należy liczyć na skonstruowanie wielomianowego algorytmu optymalizacyjnego o ile nie zostanie udowodniona równość P = NP. Istnienie takiego algorytmu jest zatem mało prawdopodobne.

Zauważmy jednak, że dla wykazania NP-zupełności problemu szeregowania wykorzystaliśmy PROBLEM PODZIAŁU ZBIORU, który nie jest silnie NP-zupełny, zatem wykazaliśmy jedynie zwykłą NP-zupełność rozpatrywanego problemu szeregowania. W istocie problem ten można rozwiązać za pomocą algorytmu pseudowielomianowego. Jego sformułowanie jest następujące [27,22]. Niech

$$x_j(t_1,t_2,\ldots,t_m) = \begin{cases} \text{prawda}, & \text{jeśli zadania } T_1,T_2,\ldots,T_j \text{ mogą być wykonane na procesorach } P_1,P_2,\ldots,P_m, \text{ w taki sposób, że procesor } P_i \text{ jest zajęty w przedziale czasu } [0,t_i],\ i=1,2,\ldots,m \\ \text{fałsz}, & \text{w przeciwnym razie} \end{cases}$$

oraz

$$x_0(t_1,t_2,\ldots,t_m) = \begin{cases} \text{prawda}, & \text{jeśli } t_i=0,\ i=1,2,\ldots,m, \\ \text{fałsz}, & \text{w przeciwnym razie} \end{cases}$$

Następnie definiuje się rekurencyjne równanie w postaci

$$x_j(t_1,t_2,\ldots,t_m) = \bigvee_{i=1}^{m} x_{j-1}(t_1,t_2,\ldots,t_{i-1},t_i-p_j,t_{i+1},\ldots,t_m).$$

Problem polega teraz na obliczeniu wartości $x_j(t_1,t_2,\ldots,t_m)$

dla $j=0,1,\ldots,n$; $t_i=0,1,\ldots,C$; $i=1,2,\ldots,m$, gdzie C oznacza górną granicę długości uszeregowania optymalnego C^*_{max}. Z powyższego wynika, że C^*_{max} wyraża się wzorem

$$C^*_{max} = \min\left\{\max\left\{t_1,t_2,\ldots,t_m\right\} : x_n(t_1,t_2,\ldots,t_m)=\underline{\text{prawda}}\right\}.$$

Powyższa procedura rozwiązuje rozpatrywany przez nas problem szeregowania w czasie $O(nC^m)$, zatem dla ustalonego m jest to algorytm pseudowielomianowy. Wynika stąd, że dla małych wartości m oraz C procedura ta może być wykorzystana w praktyce. W ogólności jej zastosowanie jest jednak ograniczone. (Inne algorytmy wykładnicze dla problemów szeregowania zadań opisano na przykład w [6].)

Inną próbą efektywnego rozwiązania rozpatrywanego problemu może być osłabienie pewnych założeń i konstrukcja algorytmu dla takiego przypadku. Dodatkowo należy wówczas rozpatrzeć przydatność tak skonstruowanego algorytmu dla problemu oryginalnego.

W przypadku naszego problemu szeregowania osłabieniem założeń będzie dopuszczenie przerywania wykonywania zadań. Okazuje się, że w przypadku zadań podzielnych omawiany problem szeregowania można rozwiązać bardzo efektywnie. Zauważmy, że czas wykonania zbiotu zadań podzielnych nie może być mniejszy niż większa z dwóch wartości: najdłuższego z czasów wykonywania i średniego czasu wykonywania, czyli [25]:

$$C^*_{max} = \max\left\{\max_j\left\{p_j\right\}\ ,\ \frac{1}{m}\sum_{j=1}^{n} p_j\right\}.$$

Jeżeli C^*_{max} jest wyznaczone przez $\frac{1}{m}\sum_{j=1}^{n} p_j$, to wszystkie procesory kończą wykonywanie zadań równocześnie. W przeciwnym razie wystąpią przestoje w pracy procesorów. Poniższy algorytm podany przez Mc Naughtona pozwala w łatwy sposób przydzielić zadania do poszczególnych procesorów w celu otrzymania uszeregowania o długości C^*_{max}.

Algorytm 1 [25]

1. Rozpocznij wykonywania dowolnego zadania na dowolnym proceso-

rze w chwili t=0.

2. Wybierz dowolne nie uszeregowane jeszcze zadanie i rozpocznij jego wykonywanie na tym samym procesorze w chwili zakończenia wykonywania poprzedniego zadania. Powtarzaj ten krok do chwili, gdy wszystkie zadania zostaną uszeregowane lub $t=C^*_{max}$.
3. Część zadania pozostająca do wykonania po osiągnięciu $t=C^*_{max}$ przydziel do innego procesora, rozpoczynając jej wykonanie od chwili t=0. Wróć do kroku 2.

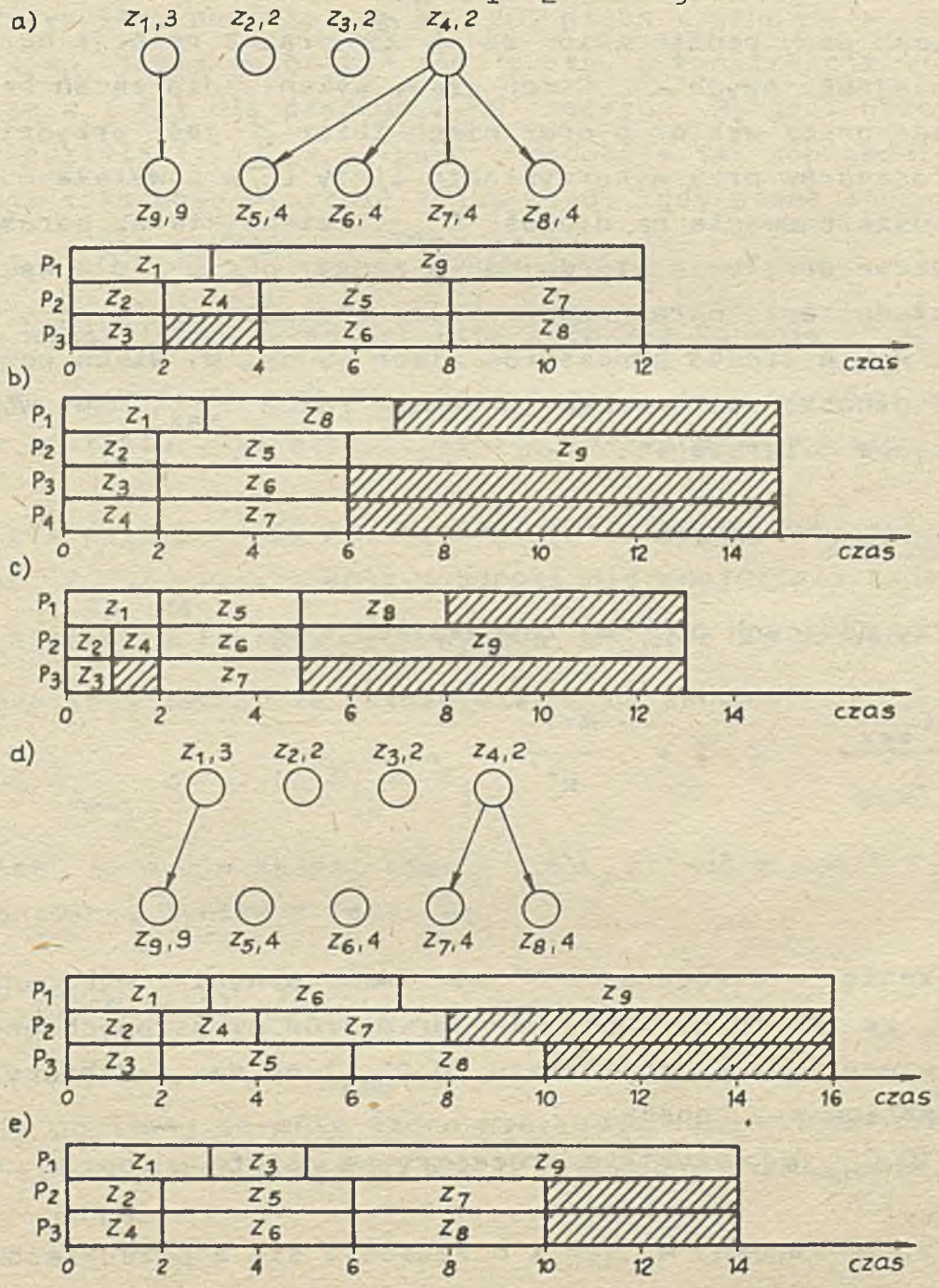
Łatwo zauważyć, że powyższy algorytm zawsze znajduje uszeregowanie, a jego optymalność wynika z faktu, że długość tego uszeregowania jest określona wzorem (1), zatem jest minimalna. Złożoność tego algorytmu jest O(n), gdyż każde zadanie jest w nim rozpatrywane tylko jeden raz, a czas z tym związany jest stały, niezależny od liczby zadań.

Zauważmy, że dopuszczenie możliwości podzielności zadań w istotny sposób ułatwiło rozwiązanie problemu, który z NP-trudnego stał się obliczeniowo łatwy. Pozostaje jeszcze kwestia oceny stopnia przydatności praktycznej tak otrzymanego rozwiązania. Otóż, w systemach komputerowych, wieloprocesorowych, pracujących ze wspólną pamięcią operacyjną, założenie podzielności zadań jest mocno uzasadniona, a rozwiązania uzyskane w ten sposób mają istotne znaczenie praktyczne. Jeśli powyższe podejście jest z jakichś względów niezbyt uzasadnione, pozostaje wówczas próba skonstruowania algorytmu aproksymacyjnego dla oryginalnego problemu i oszacowania jego zachowania się w najgorszym przypadku oraz zachowania się średniego. Najbardziej znanym typem algorytmów aproksymacyjnych dla problemów szeregowania są algorytmy szeregowania listowego, w których zadania ustawione są w pewnym porządku na liście i w momencie, w którym wolny staje się którykolwiek z procesorów, pierwsze na liście nie przydzielone zadanie zostaje wybrane i przydzielone do tego procesora. Dokładność algorytmu listowego zależy od sposobu określania kolejności zadań na liście. Ten typ algorytmów szeregowania może prowadzić do tzw. anomalii szeregowania listowego. Polegają one na tym, że długość uszeregowania może wzrosnąć, gdy:

- wzrośnie liczba procesorów,

- czas wykonywania któregoś z zadań będzie mniejszy niż przyjęty,
- zostaną usunięte niektóre łuki w grafie,
- zostaną zmienione priorytety zadań.

Na rysunku 4 pokazano przykład wpływu powyższych czynników na wzrost długości uszeregowania. Przyjęto uporządkowanie zadań według malejącej priorytetów: $Z_1, Z_2, \ldots, Z_9$.



Rys.4. Przykład anomalii algorytmów szeregowania listowego: (a) graf, uszeregowanie optymalne; (b) zwiększenie m do 4; (c) Zmniejszenie czasów wykonywania: $\bar{p} = [2,1,1,1,3,3,3,3,8]$; (d) opuszczenie niektórych łuków w grafie; (e) zmiana priorytetów zadań: $Z_1, Z_2, Z_4, Z_5, Z_6, Z_3, Z_9, Z_7, Z_8$.

Powyższe anomalie odkrył Graham [16], który oszacował także maksymalne zmiany w długości uszeregowania, które mogą powstać przez zmianę jednego lub wielu parametrów problemu. Przytoczymy ten rezultat, gdyż jego dowód jest jednym z najkrótszych w dziedzinie oszacowań najgorszego zachowania się algorytmów aproksymacyjnych i dobrze ilustruje stosowane w tej dziedzinie metody.

Niech dany będzie zbiór zadań $\mathcal{Z}$ wraz z relacją ograniczeń kolejnościowych $\prec$. Niech czasy wykonywania zadań będą określone przez wektor $\bar{p}$ oraz niech zbiór $\mathcal{Z}$ jest przydzielany do m procesorów przy wykorzystaniu listy L, a powstałe w ten sposób uszeregowanie ma długość C_{max}. Zmieńmy teraz parametry w powyższym problemie szeregowania zadań: $\bar{p}' \leq \bar{p}$ (dla wszystkich składników), ograniczenia kolejnościowe $\prec' \subseteq \prec$, lista $L' \neq L$ a liczba procesorów niech wynosi m'. Niech nowa wartość długości uszeregowania będzie równa C'_{max}. Mamy wówczas następujące twierdzenie.

Twierdzenie 2 [16].

Przy podanych powyżej założeniach mamy

$$\frac{C'_{max}}{C_{max}} \leq 1 + \frac{m-1}{m'} .$$

Dowód

Rozważmy uszeregowanie D' otrzymane w wyniku wykonania zadań Z_i ze zbioru $(\mathcal{Z}, \prec)$, dla parametrów oznaczonych primem. Zdefiniujemy podział przedziału $[0, C'_{max})$ na dwa podzbiory AiB w następujący sposób:
$A = \{ t \in [0, C'_{max}) : \text{wszystkie procesory są zajęte w chwili } t \}$,
$B = [0, C'_{max}) - A$.
Zauważmy, że zarówno A, jak i B składają się z przedziałów rozłącznych jednostronnie otwartych. Niech Z_{j_1} oznacza zadanie zakończone w D' w chwili C'_{max}, tzn. takie, dla którego $C_{j_1} = C'_{max}$.
Istnieją dwie możliwości.

1. Jeśli chwila s_{j_1} rozpoczęcia wykonywania zadania Z_{j_1} jest wewnętrznym punktem zbioru B, to z definicji B wynika, że istnieje procesor P_i, który dla pewnego $\varepsilon > 0$ jest bezczynny w przedziale $[s_{j_1} - \varepsilon, s_{j_1})$. Sytuacja taka może występować jedynie wtedy, gdy dla pewnego zadania Z_{j_2} są spełnione warunki $Z_{j_2} \prec Z_{j_1}$ oraz $C_{j_2} = s_{j_2}$.

2. Przypuśćmy teraz, że s_{j_1} nie jest wewnętrznym punktem zbioru B. Przypuśćmy ponadto, że $s_{j_1} \neq 0$. Niech $x_1 = \sup\{x : x < s_{j_1} \text{ i } x \in B\}$ lub $x_1 = 0$, jeśli zbiór ten jest pusty. Z konstrukcji A i B wynika, że $x_1 \in A$ i dla pewnego $\varepsilon > 0$ procesor P_i jest bezczynny w przedziale $[x_1 - \varepsilon, x_1)$. Lecz, podobnie jak poprzednio, sytuacja taka może wystąpić tylko wtedy, gdy pewne zadanie $Z_{j_2} \prec' Z_{j_1}$ jest wykonywane w tym przedziale.

Wykazaliśmy zatem, że albo istnieje zadanie $Z_{j_2} \prec' Z_{j_1}$ takie, że z faktu $y \in [C_{j_2}, s_{j_1})$ wynika, że $y \in A$ albo dla każdego x, z faktu, że $x < s_{j_1}$, wynika, że albo $x \in A$, albo $x < 0$.

To postępowanie możemyindukcyjnie powtarzać, tworząc ciąg zadań $Z_{j_3}, Z_{j_4}, \ldots$ dopóty, dopóki nie znajdziemy zadania Z_{j_r}, dla którego z faktu $x < s_{j_r}$ wynika, że albo $x \in A$, albo $x < 0$. Pokazaliśmy zatem, że istnieje łańcuch zadań

$$Z_{j_r} \prec' Z_{j_{r-1}} \prec' \ldots \prec' Z_{j_2} \prec' Z_{j_1} \tag{2}$$

takich, że w D'w każdej chwili $t \in B$ któreś z zadań Z_{j_k} jest wykonywane. Wynika z tego, że

$$\sum_{\emptyset' \in D'} p'_{\emptyset'} \leq (m-1) \sum_{k=1}^{r} p'_{j_k} \tag{3}$$

gdzie po lewej stronie wzoru występuje suma czasów trwania wszystkich zadań pustych $\emptyset'$ w D'. Z (2) i z założenia $\prec' \subseteq \prec$ mamy jednak

$$Z_{j_r} \prec Z_{j_{r-1}} \prec \ldots \prec Z_{j_2} \prec Z_{j_1} \tag{4}$$

Zatem

$$C_{max} \geqslant \sum_{k=1}^{r} p_{j_k} \geqslant \sum_{k=1}^{r} p'_{j_k} \qquad (5)$$

Korzystając z (3) i (5)

$$C'_{max} = \frac{1}{m'} \left\{ \sum_{k=1}^{n} p'_k + \sum_{\emptyset' \in D'} p'_{\emptyset'} \right\} \leq \frac{1}{m'} (mC_{max} + (m'-1)C_{max}) \qquad (6)$$

Z powyższego wynika, że

$$\frac{C'_{max}}{C_{max}} \leqslant 1 + \frac{m-1}{m'} \qquad (7)$$

co kończy dowód twierdzenia. []

Wykorzystując powyższe twierdzenie można wyprowadzić wzór na oszacowanie bezwzględne dowolnego algorytmu listowego LS rozwiązującego rozpatrywany przez nas problem szeregowania.

Wniosek 3

Przy podanych powyżej założeniach

$$S_{LS} = 2 - \frac{1}{m} \qquad (8)$$

Dowód

Górna granica w (8) wynika natychmiast z (7) przez przyjęcie $m'=m$. Aby pokazać, że granica ta jest osiągalna, rozważmy następujący przykład: $n=(m-1)m+1$, $\bar{p} = [1,1,\ldots,1,1,m]$, $\prec$ jest pusta, $L = (T_n, T_1, T_2, \ldots, T_{n-1})$ a $L' = (T_1, T_2, \ldots, T_n)$. Odpowiednie uszeregowania pokazano na rys.5.

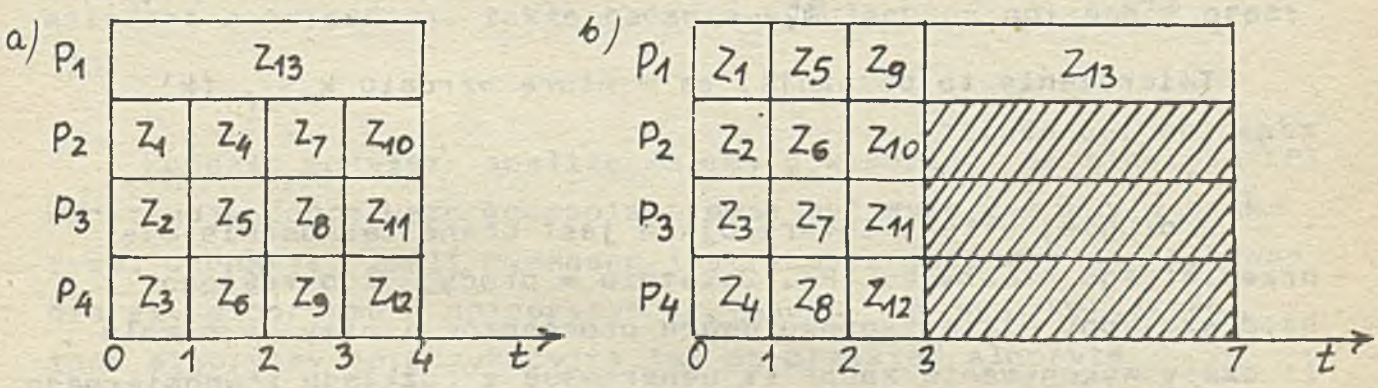
[]

Z powyższych rozważań wynika, że dowolny algorytm szeregowania listowego może konstruować złe uszeregowania, dwukrotnie dłuższe od uszeregowań optymalnych. Można spodziewać się poprawy, gdy dokonamy wstępnego porządkowania listy. Najprostszym

algorytmem jest tak zwany algorytm LPT (longest processing time). Jego działanie jest następujące.

Algorytm 2 (LPT)

1. Ustaw zadania na liście w kolejności malejących czasów wykonywania.
2. W momencie, w którym wolny staje się którykolwiek z procesorów przydziel do niego pierwsze na liście nie przydzielone zadanie. Krok ten powtarzaj do momentu wykonania wszystkich zadań.



Rys.5. Uszeregowania dla przykładu we wniosku 3: a) optymalne, b) przybliżone

Bezwzględne oszacowanie algorytmu LPT jest następujące.

Twierdzenie 4 [17]

Przy podanych wyżej założeniach

$$S_{LPT} = \frac{4}{3} - \frac{1}{3m} \qquad (9)$$

[]

Niestety, ze względu na dość znaczną długość nie jesteśmy w stanie przytoczyć dowodu tego twierdzenia. (Można go między innymi znaleźć w [6].)

Z powyższego twierdzenia wynika, że w najgorszym przypadku uszeregowanie konstruowane przez algorytm LPT może być dłuższe od optymalnego o 33%. Jednakże, wydaje się, że algorytm LPT konstruuje zazwyczaj uszeregowania lepsze niż wskazuje to osza-

cowanie (9), szczególnie w przypadku, gdy liczba zadań staje się duża. W [10] pokazano, że gdy najmniejsza liczba k zadań wykonywanych na jednym procesorze jest duża, to bezwzględnie algorytm LPT poprawia się.

Twierdzenie 6

Przy powyższych założeniach

$$S_{LPT}(k) = 1 + \frac{1}{k} - \frac{1}{mk} \qquad (10)$$

[]

Twierdzenie to pokazuje, że w miarę wzrostu k, $S_{LPT}(k)$ zdąża do jedności.

Z drugiej strony interesująca jest ocena zachowania się przeciętnego algorytmu LPT. Ostatnio w pracy [7] określono błąd względny dla przypadku dwóch procesorów i przy założeniu, że czasy wykonywania zadań są generowane z rozkładu równomiernego z przedziału [0,1].

Twierdzenie 7

Przy powyższych założeniach

$$\frac{n}{4} + \frac{1}{4(n+1)} \leq E(C_{max}^{LPT}) \leq \frac{n}{4} + \frac{e}{2(n+1)} ,$$

gdzie $E(C_{max}^{LPT})$ oznacza wartość średnią długości uszeregowania skonstruowanego przez algorytm LPT

[]

Biorąc pod uwagę, że n/4 jest dolnym ograniczeniem wartości $E(C_{max}^{*})$, otrzymujemy

$$\frac{E(C_{max}^{LPT})}{E(C_{max}^{*})} < 1 + O(1/n^2).$$

Zatem, wraz ze wzrostem n, $E(C_{max}^{LPT})$ zdąża do wartości optymalnej nie wolniej niż $1 + O(1/n^2)$ zdąża do 1. Powyższe oszacowanie

można również uogólnić na przypadek m procesorów, dla których zachodzi [8]:

$$E(C_{max}^{LPT}) \leq \frac{n}{2m} + O(\frac{m}{n}).$$

Ponadto , można wykazać [13], że $C_{max}^{LPT} - C_{max}^{*}$ zmierza do 0 prawie na pewno, gdy $n \to \infty$, rozkład czasów wykonywania ma skończoną wartość średnią, a jego funkcja gęstości f spełnia warunek $f(0) > 0$. Powyższy rezultat osiągnięty skomplikowanymi metodami potwierdzają także badania symulacyjne opisane w pracy [20].

Kończąc powyższą analizę możemy powiedzieć, że algorytm LPT konstruuje dobre uszeregowania i może być wykorzystany w praktyce. Jednakże, jeśli wymagane jest lepsze oszacowanie zachowania się algorytmu w najgorszym przypadku, można wykorzystać inne algorytmy aproksymacyjne jak na przykład algorytm MULTIFIT [9].

Należy podkreślić, że omawiany problem szeregowania jest stosunkowo dobrze i wszechstronnie zbadany i dlatego wybraliśmy go jako problem wzorcowy, dobrze ilustrujący teoretyczne definicje wprowadzone w rozdziale 2. W przypadku innych problemów szeregowania, których sformułowanie jest bardziej skomplikowane nie przeprowadzono dotychczas całościowej analizy. Jednakże niektóre zagadnienia występujące w takiej analizie zostały już dość dokładnie zbadane. Dotyczy to zwłaszcza zagadnienia złożoności obliczeniowej problemu. Przegląd uzyskanych rezultatów zamieszczono w rozdziale 5 pracy [5]. Uwzględniono w nim różne kryteria szeregowania, typy procesorów, założenia dotyczące podzielności zadań a także - ograniczeń kolejnościowych.

Dość dogłębnie przebadano również zachowanie się w najgorszym przypadku istniejących algorytmów aproksymacyjnych dla problemów szeregowania. Dotyczy to zwłaszcza minimalizacji długości uszeregowania (por.[5]). W przypadku innych kryteriów zagadnienie to, poza sporadycznymi przypadkami , nie zostało zbadane. Całkowicie otwarte jest natomiast zagadnienie analizy zachowania się przeciętnego algorytmów aproksymacyjnych. Do-

tychczasowa analiza opierała się bowiem wyłącznie na wynikach badań symulacyjnych.

BIBLIOGRAFIA

1. Aho,A.V., J.E.Hopcroft, J.D.Ullman , The Design and Analysis of Computer Algoeirthms, Addison -Wesley, Reading, Mass.,1974
2. Banachowski,L., A.Kreczmar , Elementy analizy algorytmów, WNT, Warszawa, 1981.
3. Błażewicz,J., Scheduling Problems, in: G.Laporte, S.Martello, M.Minoux, C.Ribeiro, Annals of Discrete Mathematics, North Holland, Amsterdam, 1985.
4. Błażewicz,J., Problemy optymalizacji kombinatorycznej-złożoność obliczeniowa , algorytmy aproksymacyjne, PWN, Warszawa, Łódź, 1985.
5. Błażewicz,J., W.Cellary, R.Słowiński, J.Węglarz, Badania operacyjne dla informatyków, WNT, Warszawa, 1983.
6. Coffman,E.G.,Jr., Teoria szeregowania zadań, WNT,Warszawa 1980.
7. Coffman,E.G.,Jr., G.N.Frederickson, G.S.Lueker, A note on expected makespans for largest-first sequences of independent tasks on two processors, Math.Oper.Res. 9, No.2, 1984, pp.260-266.
8. Coffman,E.G.Jr., G.N.Frederickson, G.S.Lueker, Probabilistic analysis of the LPT processor scheduling heuristic, 1983, unpublished manuscript.
9. Coffman,E.G.,Jr., M.R.Garey, D.S.Johnson, An application of bin-packing to multiprocessor scheduling, SIAM J. on Comput. 7, 1978, pp.1-17.
10. Coffman,E.G.,Jr., R.Sethi , A generalized bound on LPT sequencing, RAIRO-Informatique 10, 1976, pp.17-25.
11. Cook,S.A., The complexity of theorem proving procedures, Proc. 3rd ACM Symposium on Theory of Computing , 1971, pp.151-158.
12. Edmonds,J., Paths, trees and flowers, Canadian Journal of Mathematics 17, 1965, pp.449-467.
13. Frenk,J.B.G., A.H.G.Rinnooy Kan, The asymptotic optimality of the LPT scheduling heuristic, Report, Erasmus University, Rotterdam, 1984.

14. Garey,M.R., D.S.Johnson, "Strong" NP-completeness results: motivation, examples and implications J.ACM 25, No4, 1978, pp.499-508.
15. Garey,M.R., D.S.Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W.H.Freeman, San Francisco, 1979.
16. Graham,R.L., Bounds for certain multiprocessing anomalies, Bell System Tech. J. 45, 1966, pp.1563-1581.
17. Graham,R.L., Bounds on multiprocessing timing anomalies, SIAM J. Appl . Math. 17, 1969, pp.263-269.
18. Karp,R.M., Reducibility among combinatorial problems, in R.E.Miller and J.W.Thatcher (eds.) Complexity of Computer Computation, Plenum Press, New York, 1972, pp.85-104.
19. Karp,R.M., J.K.Lenstra, C.J.H.Mc Diarmid, A.H.G.Rinnooy Kan, Probabilistic analysis of combinatorial algorithms: an annotated bibliography, in: M.O'h Eigearthaigh, J.K.Lenstra, A.H.G.Rinnooy Kan (eds.) Combinatorial Optimization: Annotated Bibliographies, J.Wiley, Chichester, 1984.
20. Kedia,S.K., A job shop scheduling problem with parallel machines, unpublished report, Dept. of Ind.Eng., University of Michigan, Ann Arbor, 1970.
21. Ladner,R.E., On the structure of polynomial time reducibility J.ACM 22, 1975, pp.155-171.
22. Lenstra,J.K., A.H.G.Rinnooy Kan, An introduction to multiprocessor scheduling, Report BW 121, Mathematisch Centrum, Amsterdam 1980.
23. Lenstra,J.K., A.H.G.Rinnooy Kan, P.Brucker, Complexity of machine scheduling problems, Ann. Discrete Math.1, 1977 , pp.343-362.
24. Lipski,W., Kombinatoryka dla programistów, WNT Warszawa, 1982.
25. McNaughton,R., Scheduling with deadlines and loss functions, Management Sci.12, 1959, pp.1-12.
26. Rinnooy Kan,A.H.G., Probabilistic analysis of approximation algorithms , in: G.Laporte,S.Martello, M.Minoux, C.Ribeiro, Annals of Discrete Mathematics, North Holland, Amsterdam 1985
27. Rothkopf,M.H., Scheduling independent tasks on parallel processors, Management Sci. 12., 1966,pp.437-447.
28. Schnorr,C.P., Satisfiability is quasilinear complete in NQL, J.ACM 25, No.1, 1978, pp.136-145.

29. Słomiński,L., Probabilistic analysis of combinatorial algorithms: a bibliography with selected annotations, Computing 28, 1982, pp.257-267.
30. Sysło,M., Algorytmy kombinatoryczne i ich efektywność, Materiały Szkoły Jesiennej PTI, Rydzyna, 1984.

Druga Jesienna Szkoła PTI
Mrągowo, listopad 1985

ROZPROSZONE BAZY DANYCH

doc. Wojciech Cellary
Instytut Automatyki
Politechnika Poznańska
ul. Piotrowo 3A
60-965 Poznań, tel. 782370

1. Wstęp

W ostatnich latach obserwujemy gwałtowny rozwój badań w dziedzinie Systemów Rozproszonych Baz Danych (SRBD). Rozwój ten motywowany jest dwojako. Po pierwsze, jest on konsekwencją postępu jaki dokonał się w dwóch dziedzinach informatyki : bazach danych oraz sieciach komputerowych. Rozwój tych dziedzin stanowi podstawę teoretyczną, techniczną i ekonomiczną rozwoju SRBD. Po drugie, zainteresowanie SRBD wynika z zapotrzebowania na tego rodzaju systemy w gospodarce i administracji.

Można podać szereg argumentów zastosowania SRBD.

Struktura SRBD w naturalny sposób odpowiada strukturze przedsiębiorstw wielozakładowych, posiadających filie lub oddziały, rozproszone geograficznie lub lokalnie.

Budowa SRBD stanowi rozwiązanie problemu połączenia w jedną zintegrowaną bazę danych szeregu baz danych autonomicznych.

SRBD mogą być łatwo rozszerzane przez dołączanie nowych

stanowisk komputerowych.

W SRBD koszty telekomunikacji są mniejsze niż w odpowiadających im systemach scentralizowanych baz danych, dzięki przetwarzaniu lokalnemu dużych ilości informacji, które w przeciwnym przypadku musiałyby być transmitowane do centrum.

W SRBD można uzyskać wysoką równoległość działania, co prowadzi do podwyższenia efektywności.

SRBD cechują się wysoką niezawodnością i dostępnością, co wynika z faktu, że rzadko dochodzi do upadku całego systemu, a w przypadku upadku częściowego system może nadal funkcjonować w mniej lub bardziej ograniczony sposób.

Reasumując powyższe argumenty należy stwierdzić, że SRBD są szczególnie przydatne tam, gdzie samo zastosowanie ma charakter rozproszony. Ponieważ zastosowania takie są częste w praktyce, zatem należy spodziewać się dużej liczby instalacji SRBD. Przewidywanie to uzasadniają również dwa fakty : po pierwsze, handlowej dostępności sieci komputerowych, na "szczycie" których buduje się SRBD oraz po drugie, instalowania obecnie baz danych na stosunkowo małych i tanich komputerach, przy zachowaniu ich funkcjonalnych możliwości. Obecny stan badań w dziedzinie SRBD można uznać za początkowy etap wdrożeń. Znane są już wnioski z eksploatacji eksperymentalnych SRBD kontynuowanych w celach badawczych, takich jak : SDD-1, R* , SIRIUS-DELTA, Distributed--INGRES, DDM, POREL. Na rynku pojawiają się pierwsze systemy komercyjne - np. ENCOMPASS (Tandem) lub Inter System Communication (IBM).

W ramach jednego krótkiego artykułu nie jest oczywiście możliwe przedstawienie choćby tylko głównych wyników dotyczących SRBD. Dlatego celem tego artykułu jest przedstawienie problematyki SRBD, a w szczególności specyfiki tej problematyki w odniesieniu do klasycznych systemów scentralizowanych baz danych, o której zakładamy, że jest znana Uczestnikom Szkoły, co najmniej w ogólnym zakresie.

W rozdziale drugim sprecyzujemy pojęcie SRBD. W rozdziale trzecim natomiast omówimy problematykę zarządzania SRBD, w szczególności problematykę synchronizacji transakcji, optymalizacji planów wykonywania transakcji oraz niezawodności SRBD. Na zakończenie podamy bibliografię wraz z komentarzem.

2. Pojęcie systemu rozproszonej bazy danych

W tym rozdziale zdefiniujemy i skomentujemy pojęcie SRBD. W tym celu rozpoczniemy od definicji Rozproszonej Bazy Danych.

Rozproszoną bazą danych (RBD) nazywamy piątkę $(ST, \mathcal{L}, F, D, \mathcal{L}oc)$, gdzie :

$ST = \{ST_1, \ldots, ST_n\}$ jest zbiorem stanowisk komputerowych przechowywania danych,

$\mathcal{L} = \{x_i^l : i = 1,2,\ldots,N\}$ jest zbiorem danych logicznych, nazywanym logiczną bazą danych

$F = \{x_{ij}^f : i = 1,2,\ldots,N ; j = 1,2,\ldots,k_i\}$ jest zbiorem rozłącznych fragmentów danych logicznych

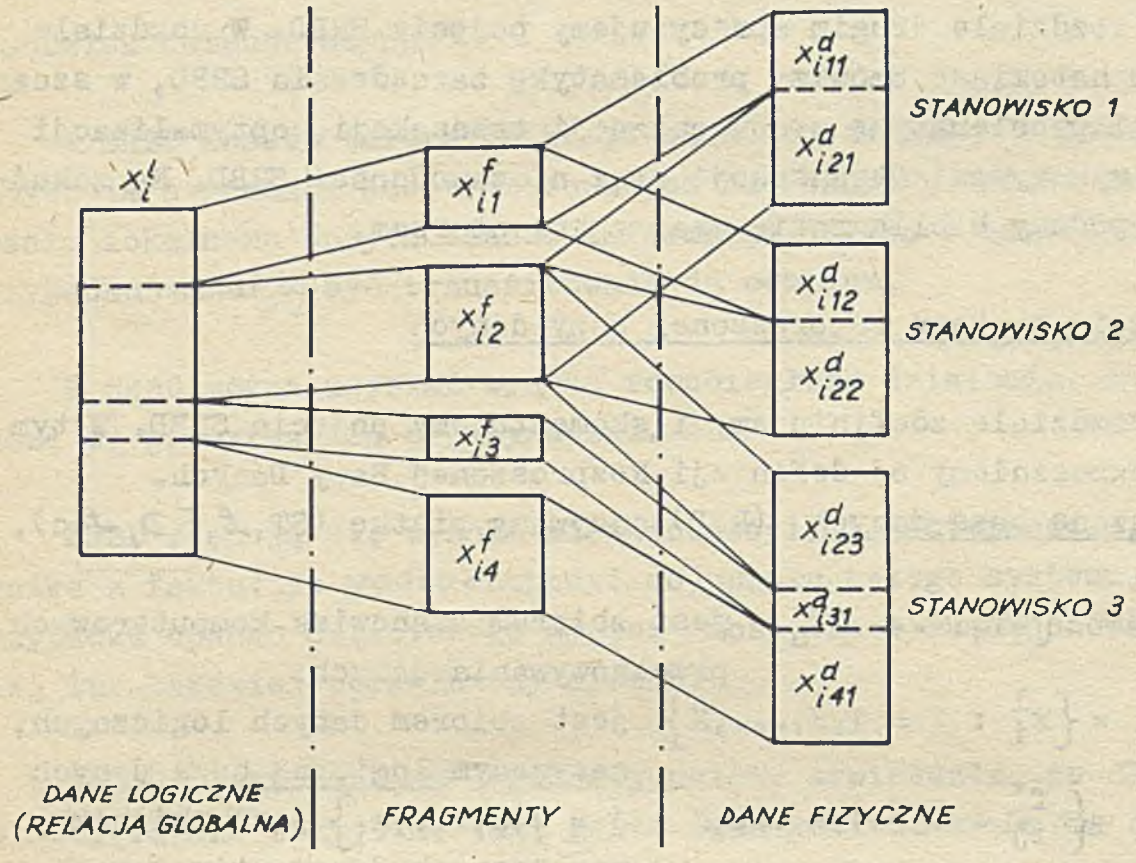
$D = \{x_{ijk}^d : i=1,2,\ldots,N ; j=1,2,\ldots,k_i ; k=1,2,\ldots, M_j\}$ jest zbiorem danych fizycznych, nazywanym fizyczną bazą danych.

$\mathcal{L}oc : D \longrightarrow ST$ jest funkcją, określającą lokalizację danej fizycznej $x \in D$ w zbiorze ST.

Logiczna baza danych $\mathcal{L}$ reprezentuje RBD z punktu widzenia jej użytkownika, który w terminach logicznej bazy danych definiuje swoje zadania zwane transakcjami. W klasycznym modelu RBD, użytkownik nie rozróżnia rozproszonej bazy danych od scentralizowanej bazy danych i pozostaje nieświadomy faktu fizycznego rozproszenia i ewentualnej duplikacji danych, do których żąda dostępu.

W dalszej części artykułu przez daną logiczną rozumieć będziemy relację, nazywaną "relacją globalną".

Każda relacja globalna może być rozbita na rozłączne fragmenty (por. rys.2.1). Wyróżniamy poziomy podział na fragmenty -



Rys.2.1. Podział danych logicznych na fragmenty i dane fizyczne.

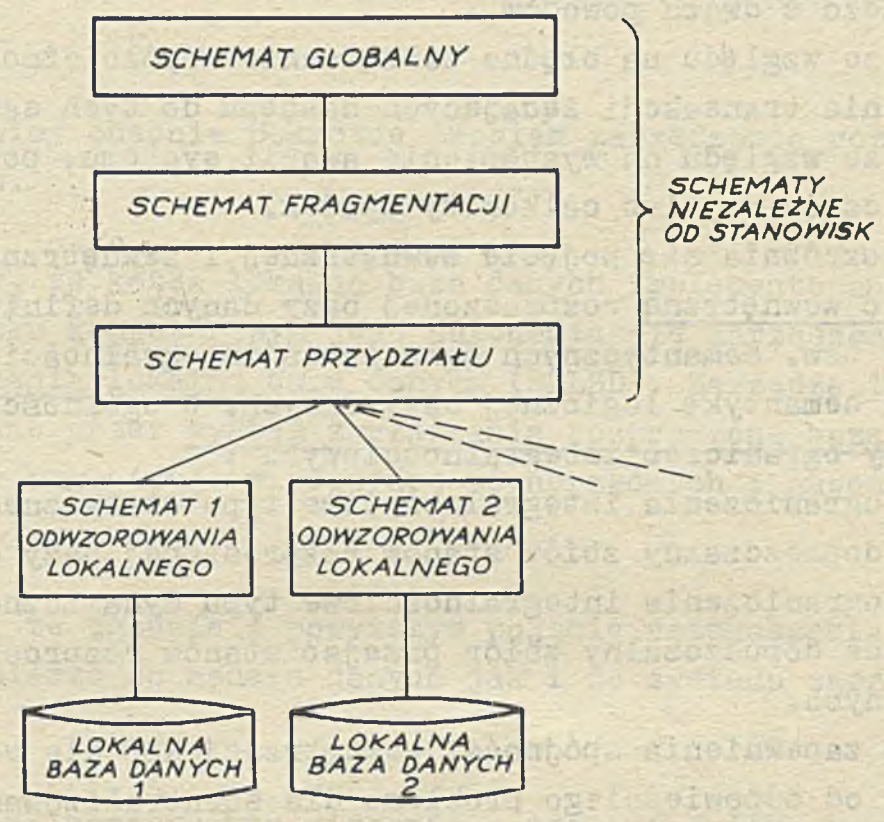
fragment zawiera wówczas podzbiór krotek, pionowy podział na fragmenty - fragment zawiera wówczas podzbiór atrybutów, oraz podział mieszany. Podział relacji globalnej na fragmenty musi być tak dokonany aby możliwa była pełna jej rekonstrukcja z fragmentów.

Z kolei każdemu fragmentowi odpowiada podzbiór danych fizycznych, będących kopiami, przechowywanych na różnych stanowiskach systemu. (por.rys.2.1.) Zbiór wszystkich danych fizycznych nazywamy fizyczną bazą danych. Fizyczna baza danych stanowi implementację logicznej bazy danych na danym sprzęcie komputerowym.

Zbiór wszystkich danych fizycznych zlokalizowanych na pojedynczym stanowisku komputerowym ST_i, nazywamy lokalną bazą danych (LBD_i).

Architektura RBD przedstawiona jest na rys.2.2. Warto podkreślić trzy główne cechy tej architektury :

1° Rozdzielenie fragmentacji danych logicznych od podziału



Rys.2.2. Architektura rozproszonej bazy danych

na dane fizyczne i ich przydziału do stanowisk komputerowych.

2° Pełna kontrola duplikacji danych fizycznych na poziomie schematu fragmentacji.

3° Niezależność od organizacji lokalnych baz danych (możliwość budowy heterogenicznych RBD).

Przejdziemy obecnie do krótkiego naszkicowania problemu spójności RBD oraz wprowadzenia pojęcia transakcji.

Ogólnie rzecz biorąc, przez spójność RBD należy rozumieć maksymalnie wierne odbicie fragmentu rzeczywistości, którego ta baza danych jest abstrakcyjnym odzwierciedleniem. W węższym sensie, przez spójność rozumie się problem zapewnienia poprawności danych i związków między nimi. Zachowanie spójności bazy danych należy wówczas rozumieć jako ochronę danych przed możliwością wystąpienia błędu w wyniku niewłaściwego : uaktualnienia, usunięcia lub wprowa-

dzenia danych. Spójność bazy danych może przy tym zostać naruszona zasadniczo z dwóch powodów :

- ze względu na błędne zarządzanie współbieżnością wykonywania transakcji żądających dostępu do tych samych danych,
- ze względu na wystąpienie awarii systemu, powodującej jego częściowy lub całkowity upadek.

W RBD rozróżnia się pojęcie wewnętrznej i zewnętrznej spójności. Spójność wewnętrzną rozproszonej bazy danych definiujemy w oparciu o zbiór tzw. semantycznych ograniczeń integralnościowych, nałożonych na semantykę logicznej bazy danych. W ogólności wyróżniamy dwa typy ograniczeń integralnościowych :

- ograniczenia integralnościowe typu statycznego, określające dopuszczalny zbiór stanów rozproszonej bazy danych,
- ograniczenia integralnościowe typu dynamicznego, określające dopuszczalny zbiór przejść stanów rozproszonej bazy danych.

Problem zapewnienia spójności wewnętrznej RBD nie różni się w swej istocie od odpowiedniego problemu dla scentralizowanych baz danych. Specyficznym problemem RBD jest problem zapewnienia ich spójności zewnętrznej.

Mówimy, że RBD jest w stanie spójności zewnętrznej, jeżeli przy założeniu, że zbiór aktualnie wykonywanych transakcji jest pusty, dla każdego fragmentu każdej danej logicznej stany wszystkich odpowiadających mu danych fizycznych (kopii) są identyczne.

Jak wiadomo zachowanie spójności wewnętrznej bazy danych, w szczególności rozproszonej bazy danych, zależy od spełnienia w każdej chwili ograniczeń integralnościowych nałożonych na semantykę bazy danych. W rzeczywistości, nie zawsze jest możliwe zdefiniowanie i wyspecyfikowanie wszystkich ograniczeń integralnościowych nałożonych na bazę danych, gdyż nie są one często znane nawet projektantowi bazy danych. Ponadto w praktyce jest bardzo trudne, a w odniesieniu do rozproszonej bazy danych wręcz niemożliwe, zweryfikowanie czy dany użytkownik nie narusza któregoś z ograniczeń integralnościowych nałożonych na bazę danych. Z tego względu wprowadza się koncepcję transaksji jako elementarnej jednostki interakcji użytkownika z RBD. Zasadniczą cechą transakcji jest cecha spójności : transakcja zachowuje spójność rozproszonej bazy danych

w tym sensie, że odwzorowuje stan spójny RBD w stan spójny, przy czym RBD może nie być w stanie spójnym w trakcie wykonywania transakcji.

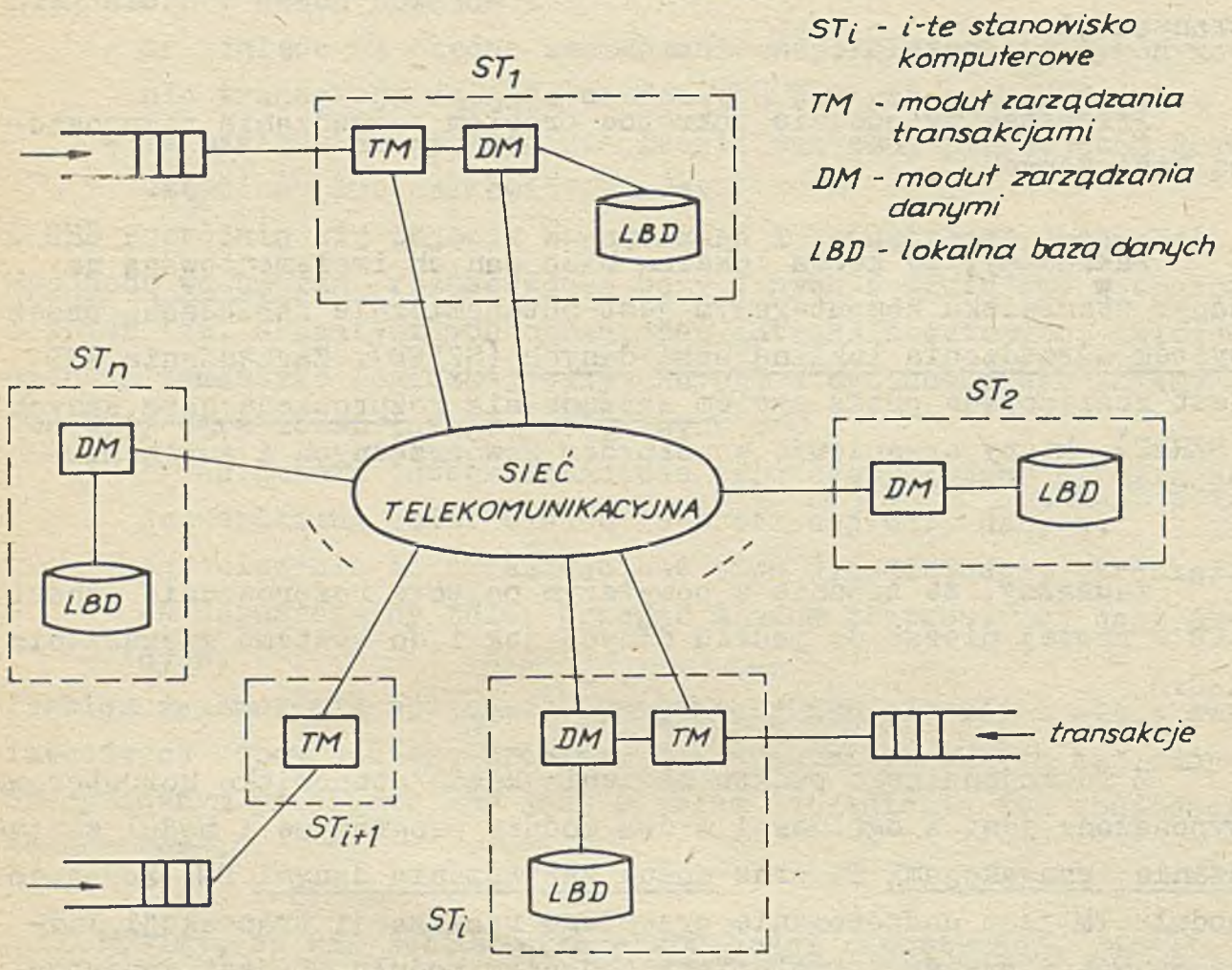
Przedstawimy obecnie pokrótce problem zarządzania rozproszoną bazą danych.

Zakładamy, że każda lokalna baza danych implementowana na danym stanowisku komputerowym jest autonomicznie zarządzana przez system zarządzania lokalną bazą danych (SZLBD). Zarządzanie RBD jest realizowane przez system zarządzania rozproszoną bazą danych (SZRBD), który organizuje współpracę równorzędnych i autonomicznych SZLBD.

Zauważmy, że zgodnie z powyższym pojęcie rozproszenia odnosi się w równej mierze do modelu danych jak i do systemu zarządzania RBD.

Z funkcjonalnego punktu widzenia każde stanowisko komputerowe wyposażone jest w ogólności w dwa moduły programowe : moduł zarządzania transakcjami TM oraz moduł zarządzania danymi DM. Zadaniem modułu TM jest nadzorowanie przebiegu realizacji transakcji inicjowanych w systemie, natomiast zadaniem modułu DM jest zarządzanie dostępem do LBD, zlokalizowanej na tym stanowisku komputerowym. Można przyjąć, że moduły programowe TM i DM wszystkich stanowisk komputerowych tworzą SZRBD, pamiętając, że w istocie w każdym module można wyróżnić dwa poziomy : poziom globalny dotyczący RBD oraz poziom lokalny dotyczący LBD.

W przypadku, gdy stanowisko komputerowe wyposażone jest wyłącznie w moduł TM, mówimy o stanowisku dostępu i zarządzania transakcjami, natomiast w przypadku, gdy stanowisko komputerowe wyposażone jest wyłącznie w moduł DM, mówimy o stanowisku przechowywania i przetwarzania danych (rys.2.3). W praktyce często stanowisko dostępu i zarządzania transakcjami oraz stanowisko przechowywania i przetwarzania danych stanowią fizycznie jeden komputer z pamięcią zewnętrzną i inteligentnymi terminalami. Spotyka się również systemy rozproszone zorganizowane w ten sposób, że występują w nich wyłącznie wydzielone stanowiska dostępu i za-



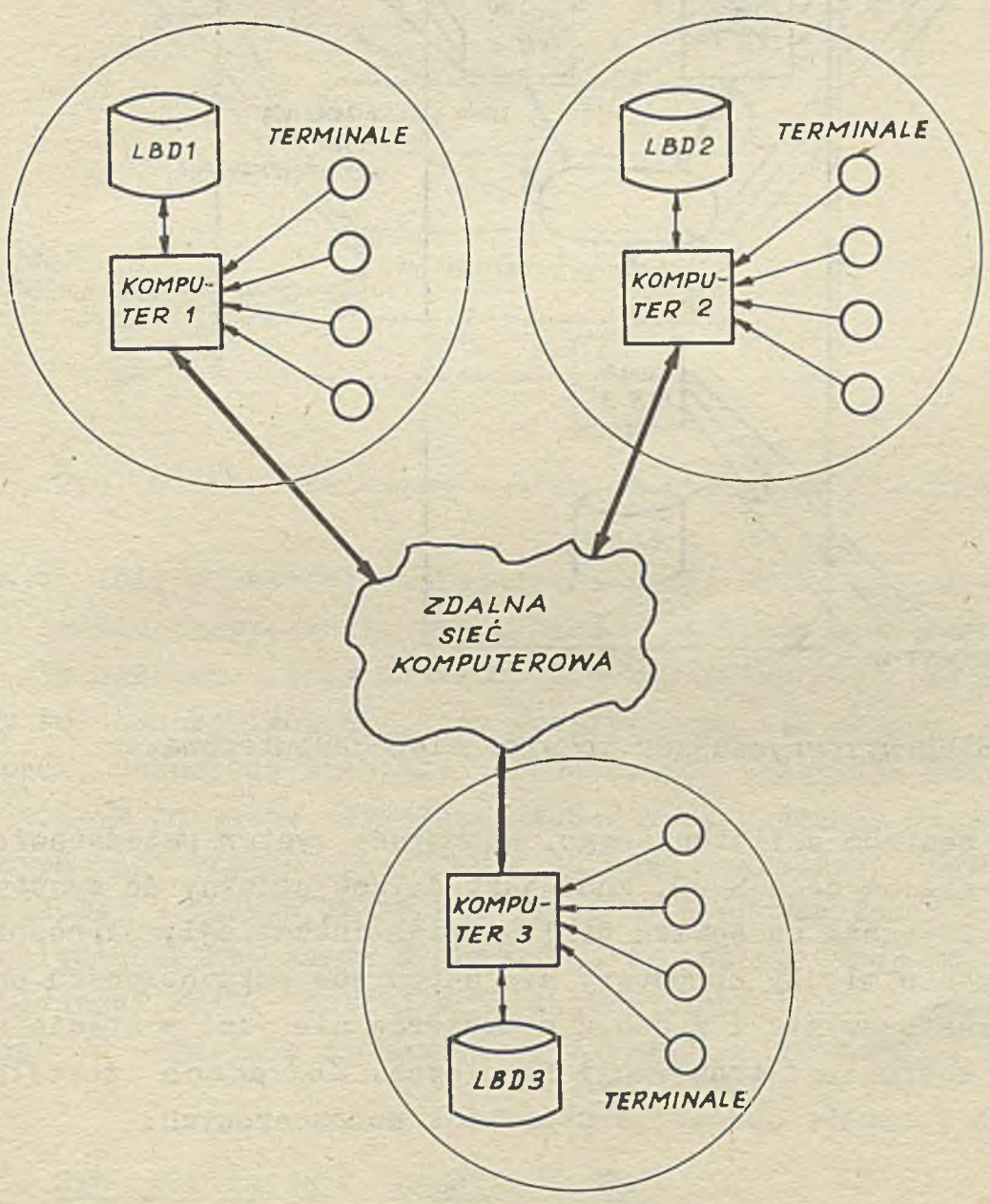
Rys. 2.3. Architektura systemu rozproszonej bazy danych.

zarządzania transakcjami oraz stanowisko przechowywania i przetwarzania danych.

Możemy obecnie zdefiniować System Rozproszonej Bazy Danych (SRBD) jako trójkę (RBD, SZRBD, τ) gdzie RBD jest rozproszoną bazą danych, SZRBD - systemem zarządzania rozproszoną bazą danych, a τ zbiorem transakcji.

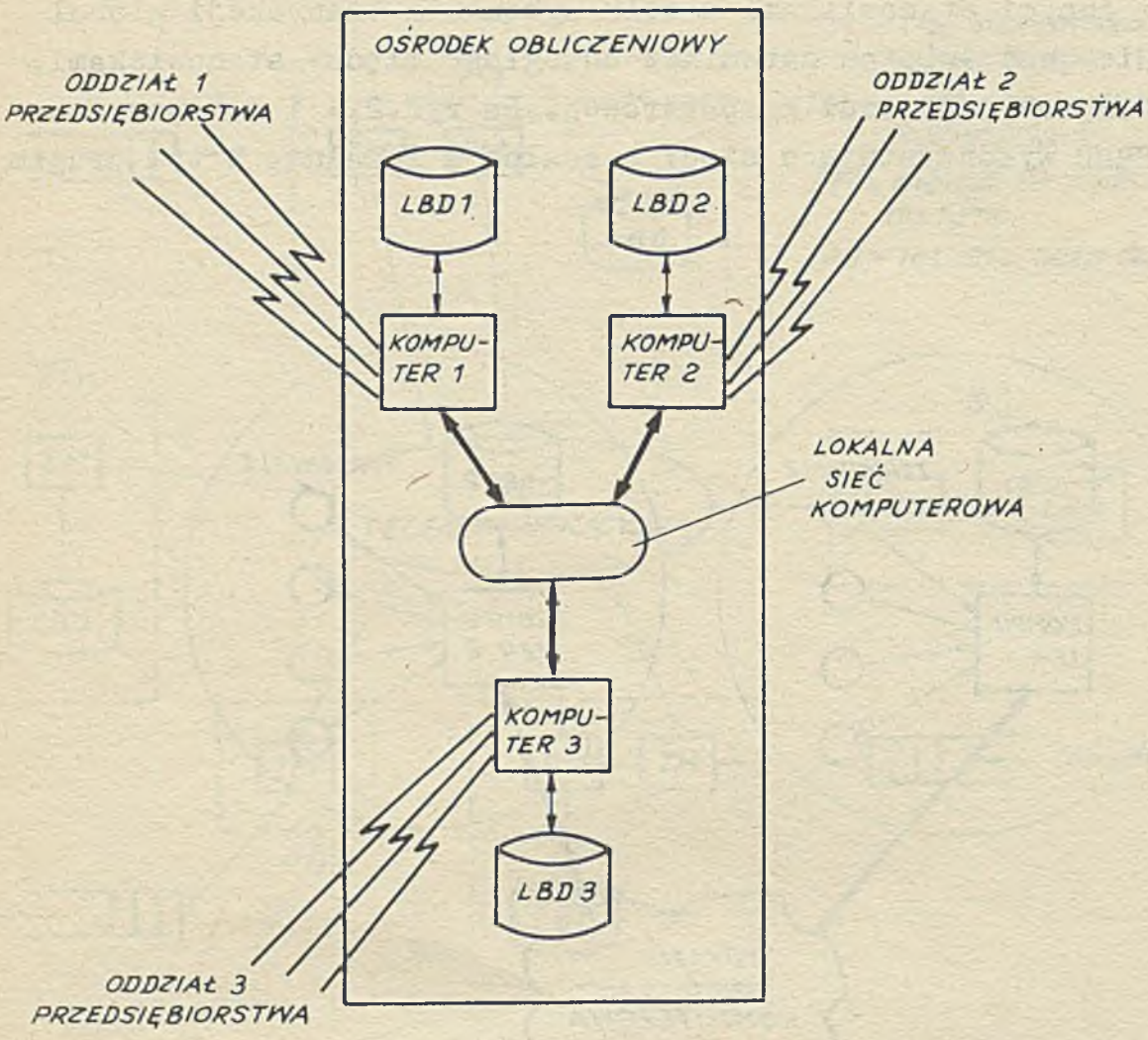
Jak wynika z wprowadzonych uprzednio definicji, główną cechą charakterystyczną SRBD jest autonomia współpracujących ze sobą stanowisk komputerowych wchodzących w jego skład. Każde stanowisko komputerowe, które może być w ogólności implementowane na kilku

komputerach musi być zdolne do samodzielnego wykonania transakcji lokalnych, czyli dotyczących tylko tego stanowiska, oraz do współpracy z innymi stanowiskami w celu wykonania transakcji globalnych. Nie jest istotna natomiast odległość między stanowiskami, ani rodzaj użytej sieci komputerowej. Na rys.2.4 i 2.5 przedstawiono SRBD wykorzystujące sieci : zdalną i lokalną. W tym drugim



Rys. 2.4. SRBD wykorzystujący zdalną sieć komputerową.

przypadku nie ma znaczenia fakt, że cały SRBD został zlokali zowany

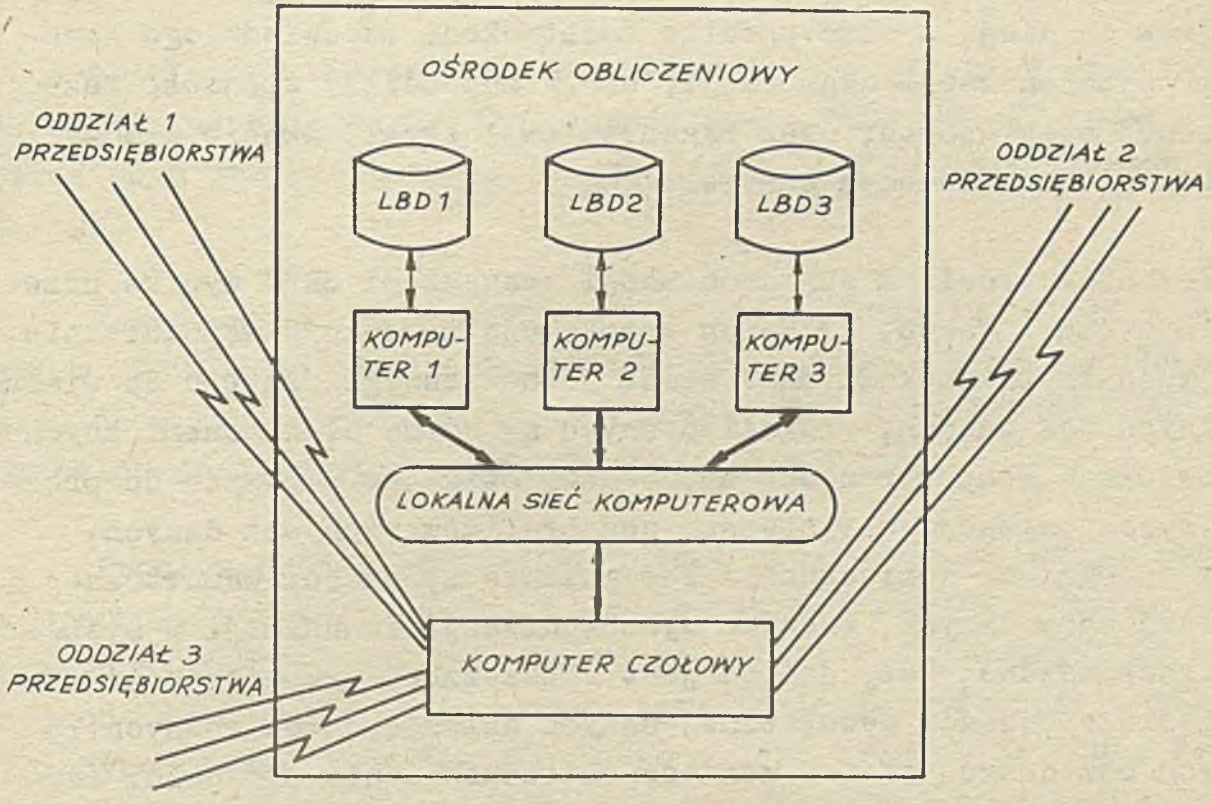


Rys. 2.5 SRBD wykorzystujący lokalną sieć komputerową.

na terenie centrum obliczeniowego. Natomiast system przedstawiony ny rys.2.6 nie ma cech SRBD, choć jest dalece podobny do systemu z rys.2.5. Osiągnął on bowiem taki stopień integracji, że rozproszenie danych pomiędzy komputery nie odpowiada rozproszonemu charakterowi zastosowania i żaden z komputerów nie jest w stanie samodzielnie obsługiwać transakcji lokalnych. Zatracona została zatem w tym systemie autonomia stanowisk komputerowych.

3. Problematyka zarządzania systemami rozproszonych baz danych.

Jak można wywnioskować z rozdziału 2, główne novum problematyki rozproszonych baz danych w odniesieniu do baz danych scentralizo-



Rys. 2.6 Wielokomputerowy system scentralizowanej bazy danych.

wanych leży w zakresie zarządzania nimi. Zauważmy, że dzięki maskowaniu rozproszenia przez SZRBD, problemy występujące na poziomie oprogramowania użytkowego wykorzystującego bazę danych, czy to scentralizowaną , czy rozproszoną, nie różnią się. Również szeroko rozumiane problemy modelu danych i języków nie ulegają zasadniczym zmianom. Natomiast problemy zarządzania rozproszoną bazą danych mają własną specyfikę i są nowe jakościowo. Możemy je sklasyfikować w trzech grupach : problemy synchronizacji transakcji, problemy optymalizacji wykonywania transakcji oraz problemy niezawodności SRBD. Poniżej scharakteryzujemy te problemy, rozpoczynając od problemów synchronizacji transakcji w SRBD.

3.1. Synchronizacja transakcji w SRBD.

Jak wynika z samej istoty SRBD, systemy te stanowią środowisko wieloprocesorowe, wieloprogramowe i wielodostępne. Stąd jednym z najistotniejszych i najtrudniejszych problemów stojących przed projektantem systemu zarządzania rozproszoną bazą danych jest

problem poprawnego zarządzania współbieżnym dostępem wielu użytkowników do niej, a więc problem konstrukcji odpowiedniego algorytmu synchronizacji transakcji, który zapewniłby spójność rozproszonej bazy danych, oraz zagwarantował zrealizowanie każdej transakcji w skończonym czasie.

Złożoność problemu synchronizacji transakcji SRBD wynika przede wszystkim z faktu, że każda transakcja ze zbioru współbieżnie wykonywanych transakcji może żądać jednoczesnego dostępu do wielu lokalnych baz danych, zlokalizowanych na różnych, autonomicznych stanowiskach komputerowych. Wobec powyższego, w stosunku do problemu synchronizacji w systemach scentralizowanych baz danych, który polegał na konieczności zapewnienia spójności wewnętrznej lokalnej bazy danych, problem synchronizacji transakcji w systemach rozproszonej bazy danych jest rozszerzony o konieczność zapewnienia spójności wewnętrznej danych należących do różnych lokalnych baz danych oraz o konieczność zapewnienia spójności zewnętrznej, rozumianej jako identyczność wszystkich kopii fizycznych tej samej danej logicznej.

Dodatkową trudność konstrukcji algorytmów synchronizacji transakcji w SRBD stanowi fakt, że w systemach takich żadne stanowisko komputerowe nie dysponuje pełną informacją o globalnym stanie całego systemu. Stąd konieczność podejmowania decyzji zarządzających na danym stanowisku komputerowym na podstawie niepełnej i nie w pełni aktualnej informacji o działaniu pozostałych stanowisk komputerowych.

W problemie synchronizacji transakcji w systemach rozproszonej bazy danych można wyróżnić dwa aspekty : aspekt spójności rozproszonej bazy danych w kontekście współbieżnej realizacji zbioru transakcji oraz aspekt konfliktów działania systemu.

W celu omówienia aspektu spójności RBD wprowadzimy pojęcie realizacji zbioru transakcji wykonywanych w SRBD. Realizacją zbioru transakcji nazywamy częściowo uporządkowany zbiór wszystkich operacji elementarnych składających się na transakcje, taki, że zachowane są relacje poprzedzania operacji w ramach każdej transakcji oraz że wszystkie operacje wykonywane na tym samym

stanowisku komputerowym są dobrze uporządkowane.

Realizację zbioru transakcji nazywamy realizacją sekwencyjną, jeżeli żadne dwie transakcje nie są wykonywane współbieżnie, w przeciwnym przypadku mówimy o realizacji współbieżnej zbioru transakcji.

Zauważmy, że realizacja jest opisem wykonania zbioru transakcji w SRBD, a ściślej - opisem sposobu dostępu do RBD. Jednocześnie, dana realizacja zbioru transakcji jest wynikiem działania określonego algorytmu synchronizacji transakcji.

Problem poprawności współbieżnej realizacji zbioru transakcji rozpatruje się najczęściej w kontekście tak zwanej informacji czysto syntaktycznej. Oznacza to, że rozumie się poprawność w ten sposób, że dla dowolnego zbioru ograniczeń integralnościowych nałożonych na RBD oraz dla dowolnego zbioru transakcji, dana realizacja odwzorowuje stan spójny RBD w stan spójny oraz każda transakcja widzi stan spójny RBD.

Tak zdefiniowane kryterium poprawności współbieżnej realizacji zbioru transakcji jest równoważne tak zwanemu kryterium uszeregowalności, Mówimy, że dowolna realizacja jest poprawna wtedy i tylko wtedy, gdy jest ona uszeregowalna, tzn. gdy jest ona równoważna dowolnej realizacji sekwencyjnej. Istotne w tej definicji jest pojęcie równoważności. Równoważność tę można interpretować, albo w sensie prezentacji identycznego obrazu stanów bazy danych widzianych przez transakcje należące do τ albo w sensie wygenerowania przez realizację współbieżną zbioru τ identycznego stanu jak w przypadku którejkolwiek z realizacji sekwencyjnych zbioru τ.

Mówić będziemy zatem, że realizacja współbieżna r zbioru transakcji τ jest uszeregowalna jeżeli istnieje realizacja sekwencyjna r' zbioru transakcji τ taka, że dla dowolnych ograniczeń integralnościowych nałożonych na bazę danych oraz dowolnego zbioru transakcji stan RBD widziany przez dowolną transakcję $T \in \tau$ dla realizacji r oraz stan RBD widziany przez tą transakcję dla realizacji r' są identyczne, oraz stany RBD osiągnięte w wyniku realizacji r oraz r' są również identyczne.

Powyższa definicja kryterium uszeregowalności ma charakter ogólny. Warto wspomnieć, że historycznie rzecz biorąc, dawniej pod pojęciem uszeregowalności rozumiano równoważność współbieżnej realizacji r i realizacji sekwencyjnej r' jako równoważność rozwiązania tzw. konfliktów dostępu. Tak rozumianą uszeregowalność nazywamy obecnie D-uszeregowalnością. D-uszeregowalność współbieżnej realizacji r jest warunkiem dostatecznym uszeregowalności tej realizacji. Najczęściej przyjmuje się D-uszeregowalność jako podstawowe kryterium poprawności współbieżnej realizacji zbioru transakcji. Wynika to z dwóch zasadniczych przyczyn. Po pierwsze, problem uszeregowalności jest dla ogólnego modelu transakcji problemem NP-zupełnym, w przeciwieństwie do problemu D-uszeregowalności który należy do klasy problemów P.

Po drugie, ze sformułowania kryterium uszeregowalności nie wynika w jaki sposób można praktycznie zapewnić uszeregowalność dowolnej współbieżnej realizacji zbioru transakcji. Spełnienie kryterium D-uszeregowalności może być natomiast stosunkowo prosto zagwarantowane w praktyce.

Sformułujmy zatem obecnie kryterium D-uszeregowalności. W tym celu niezbędne jest wprowadzenie pojęcia konfliktu dostępu. Mówimy, że dwie transakcje T_i, T_j znajdują się w konflikcie dostępu, jeśli w skład każdej z nich wchodzi operacja elementarna na pewnej danej fizycznej x, przy czym co najmniej jedna z tych operacji jest typu zapis. Zauważmy obecnie, że uszeregowalność zbioru transakcji dotyczy w istocie transakcji znajdujących się w konflikcie. Zauważmy dalej, że wszystkie elementarne operacje znajdujące się w konflikcie muszą być wykonane ściśle sekwencyjnie, a więc innymi słowy, transakcje znajdujące się w konflikcie muszą pozostawać względem siebie w odpowiedniej relacji poprzedzania $T_i \longrightarrow T_j$. Łatwo zauważyć, że relacja $\longrightarrow$ jest relacją częściowo porządkującą zbiór transakcji τ. Możemy obecnie sformułować warunek D-uszeregowalności.

Dowolna realizacja r zbioru transakcji τ jest uszeregowalna, jeżeli relacja poprzedzania $\longrightarrow$ jest acykliczna.

Testowanie poprawności dowolnej realizacji r sprowadza się

obecnie do testowania acykliczności relacji $\longrightarrow$.
Możemy obecnie powiedzieć, że dany algorytm synchronizacji transakcji w SRBD jest poprawny w odniesieniu do aspektu spójności bazy danych, jeśli wszystkie generowane przez niego realizacje zbiorów transakcji są poprawne.

Omówimy teraz aspekt konfliktów działania SRBD.

Kryterium uszeregowalności nie jest wystarczającym kryterium poprawności działania systemów rozproszonej bazy danych, lub innymi słowy, poprawności synchronizacji transakcji w SRBD. Jak wspomnieliśmy we wstępie, problem poprawności działania SRBD rozpatrywać będziemy w ujęciu globalnym, jako :

(i) problem zapewnienia spójności wewnętrznej i zewnętrznej RBD,
(ii) problem zapewnienia zrealizowania każdej transakcji zainicjowanej w systemie.

Przyjęcie określonej metody synchronizacji transakcji w SRBD zapewniającej spójność wewnętrzną i zewnętrzną RBD (warunek (i)) może pociągnąć za sobą możliwość wystąpienia w systemie tzw. konfliktów działania SRBD. Wyróżniamy cztery typy tych konfliktów, a mianowicie : martwy punkt, cykliczne restartowanie, stałe zablokowanie i stałe restartowanie.

Konflikt martwego punktu zachodzi wówczas, gdy dwie lub więcej transakcji oczekuje wzajemnie na uwolnienie danych niezbędnych do zakończenia ich wykonywania.

Konflikt cyklicznego restartowania występuje wówczas, gdy dwie lub więcej transakcji powoduje wzajemne wycofywanie lub wywłaszczanie.

Konflikt stałego zablokowania zachodzi wówczas, gdy na skutek nieskończonego strumienia nowych transakcji inicjowanych w SRBD transakcja lub zbiór transakcji nigdy nie uzyskają dostępu do wszystkich danych, niezbędnych do ich wykonania i tym samym nigdy nie zostaną zrealizowane.

Konflikt stałego restartowania zachodzi wówczas, gdy na skutek nieskończonego strumienia nowych transakcji inicjowanych w systemie w nieznanych a priori momentach, transakcja lub zbiór

transakcji są stale wycofywane lub wywłaszczane, i tym samym nigdy nie zostaną zrealizowane.

Wystąpienie w systemie któregokolwiek z powyższych konfliktów działania prowadzi w konsekwencji do niespełnienia warunku (ii), tj. do niezakończenia wykonywania transakcji lub zbioru transakcji zainicjowanych w systemie. Niezrealizowanie transakcji prowadzi również do naruszenia spójności RBD, rozumianej teraz szeroko jako realcji odpowiedniości pomiędzy "światem zewnętrznym" a jego abstrakcyjnym odzwierciedleniem, jakim jest RBD.

Wystąpienie konfliktów działania SRBD jest konsekwencją przyjętego w danym algorytmie synchronizacji transakcji sposobu postępowania w przypadku wystąpienia konfliktu dostępu.

Istnieją trzy alternatywne rozwiązania w sytuacji, w której transakcja T_i znajduje się w konflikcie dostępu z transakcją T_j, lub innymi słowy, transakcja T_i żąda niekompatybilnego dostępu do danej przydzielonej transakcji T_j.

1° Transakcja T_i jest wstrzymana i czeka na zwolnienie danej przez transakcję T_j.

2° Transakcja T_i zostaje wycofana, uwalnia wszystkie dane, które zostały do niej przydzielone, i następnie restartuje.

3° Transakcja T_i wywłaszcza transakcję T_j z posiadanej danej, co pociąga za sobą konieczność wycofania wywłaszczonej transakcji, czyli uwolnienie wszystkich posiadanych przez nią danych oraz jej restart.

Przyjęcie pierwszego rozwiązania może prowadzić do wystąpienia konfliktu martwego punktu oraz konfliktu stałego zablokowania. Natomiast przyjęcie drugiego lub trzeciego rozwiązania może spowodować konflikt cyklicznego restartu oraz stałego restratowania.

Metody synchronizacji transakcji w SRBD można sklasyfikować w czterech grupach, metody blokowania, metody porządkowania transakcji według etykiet czasowych, metody walidacji oraz metody mieszane. Oczywiście w szczupłych ramach tego artykułu nie jesteśmy w stanie przedstawić tych metod. Odsyłamy Czytelników do bibliografii. Warto jednak wspomnieć, że tak szerokie postawienie problemu synchronizacji rozumianego łącznie jako problem spójności

RBD i konfliktów działania datuje się od niedawna. W starszych metodach ignorowano problem konfliktów działania, lub co najwyżej brano pod uwagę konflikt martwego punktu. W nowych metodach uwzględnia się oczywiście wszystkie konflikty, zwracając ponadto dużą uwagę na efektywność metody, decentralizację zarządzania, oraz powiązanie z innymi problemami, takimi jak na przykład : hierarchizacja ziarnistości bazy danych, czy zabezpieczenia przed upadkiem. Szczególne zainteresowanie budzą ostatnio metody mieszane.

3.2. Optymalizacja planów wykonywania transakcji w SRBD.

Problem optymalizacji planów wykonywania transakcji w SRBD można sformułować następująco. Dla danej transakcji wymagającej wykonania szeregu operacji na danych zlokalizowanych na różnych stanowiskach SRBD, należy określić sekwencję wykonywania tych operacji oraz ich lokalizację tak aby optymalizować wybrane kryterium oceny działania systemu. Najczęściej rozpatrywane kryteria to czas odpowiedzi transakcji oraz sumaryczne obciążenie systemu.
W celu unaocznienia ważności problematyki optymalizacji planów wykonywania transakcji w SRBD rozważmy bazę zawierającą dane o dostawcach i dostarczanych przez nich częściach. Dane rozproszone są w systemie w następujący sposób :

Stanowisko A : relacja DOSTAWCY (NR DOSTAWCY, MIASTO)
relacja POWIAZANIA (NR DOSTAWCY, NR CZĘŚCI)
Stanowisko B : relacja CZĘŚCI (NR CZĘŚCI, KOLOR)

Relacja DOSTAWCY liczy 10 000 krotek, relacja POWIAZANIA 1 000 000 krotek, a relacja CZĘŚCI 100 000 krotek. Krotki wszystkich relacji mają długość 100 bitów. Dodatkowo załóżmy, że liczba czerwonych części równa się 10 oraz że liczba części dostarczanych przez dostawców z Londynu równa się 100 000. Zakładamy, że szybkość transmisji wynosi 10 000 bitów/sek i że opóźnienie dostępu wynosi 1 sekundę.

Rozważmy następującą transakcję : "Podaj numery dostawców z Londynu dostarczających części w kolorze czerwonym". Transakcję tę można zrealizować na sześć sposobów zestawionych w poniższej tablicy.

Plan	Sposób wykonania	Czas transmisji
1	Dla każdej dostawy z Londynu sprawdzenie koloru części	2.3 dnia
2	Przesłanie relacji CZEŚCI i POWIAZANIA na stanowisko B	28 godzin
3	Przesłanie relacji CZEŚCI na stanowisko A	16.7 minut
4	Przesłanie każdej dostawy z Londynu na stanowisko B	16.7 minut
5	Dla każdej czerwonej części sprawdzenie czy pochodzi ona od dostawcy z Londynu	20 sekund
6	Przesłanie części czerwonych na stanowisko A	1 sekunda

Jak widać, czasy realizacji poszczególnych planów wykonywania rozpatrywanej transakcji w SRBD wahają się od 2,3 dnia do 1 sekundy, co potwierdza, że wybór planu w istotny sposób wpływa na efektywność wykonywania transakcji w SRBD.

Dotychczas opracowano cały szereg algorytmów generowania planów wykonywania transakcji w SRBD. W algorytmach tych szczególną wagę poświęca się sposobom wykonywania operacji relacyjnej połączenia. Wynika to z faktu, że koszt wykonywania tej operacji w zdecydowanym stopniu determinuje koszt wykonywania całej transakcji. Operacja połączenia wymaga zarówno transmisji dużych woluminów danych, jeśli łączone transakcje znajdują się na różnych stanowiskach, jak i czasochłonnego przetwarzania. W większości algorytmów realizuje się operację połączenia wykonując najpierw

określoną sekwencję operacji półpołączenia.

W większości metod optymalizacji planów wykonywania transakcji w SRBD opracowanych dotychczas dąży się wyłącznie do minimalizacji wolumínów danych transmitowanych pomiędzy stanowiskami SRBD, a zatem do minimalizacji czasów transmisji, z pominięciem czasów przetwarzania transakcji na stanowiskach komputerowych. Podejście takie jest uzasadnione w systemach rozproszonych baz danych starszego typu, wykorzystujących sieci komputerowe o małej szybkości transmisji (np.2 400 bd) i duże komputery w roli stanowisk SRBD. Wówczas bowiem mamy rzeczywiście do czynienia z silną dominacją czasów transmisji nad czasami przetwarzania. Jednakże postęp technologiczny ostatnich kilku lat odwrócił te proporcje. Obserwujemy bowiem ostatnio gwałtowny wzrost szybkości transmisji w sieciach komputerowych. Wspomnijmy tu o sieciach z łączami światłowodowymi o szybkości transmisji rzędu gigabd. Przyrost ten jest znacznie szybszy niż przyrost mocy obliczeniowej komputerów.

Z drugiej strony, obserwuje się silne tendencje do stosowania małych komputerów - mini i mikrokomputerów w roli stanowisk SRBD. Oznacza to relatywne zmniejszenie mocy obliczeniowej stanowisk w stosunku do możliwości transmisyjnych sieci, a więc innymi słowy wzrost czasów przetwarzania w stosunku do czasów transmisji. Z tego względu, najnowsze badania dotyczące optymalizacji planów wykonywania transakcji w SRBD idą w kierunku uwzględnienia zarówno czasów przetwarzania, jak i czasów transmisji, w szczególności w ramach tych pierwszych uwzględnia się : czasy przygotowania transmisji, wykonywania operacji relacyjnych, dekompozycji transakcji, obsługi synchronizacji transakcji, obsługi zabezpieczeń przed upadkiem systemu, itd.

3.3.Niezawodność SRBD.

Jak wiadomo, ogólnie przez niezawodność systemu rozumiemy miarę zgodności jego rzeczywistego działania z pewną autorytatywną specyfikacją tego działania. Rozumiejąc SRBD jako system użytkowy

(np. bankowy) wykorzystujący rozproszoną bazę danych, taka autorytatywna specyfikacja jego działania na najwyższym poziomie jest w ogólności zależna od zastosowania, gdyż obejmuje wszystkie nałożone ograniczenia integralnościowe. Celowe jest rozbicie problemu niezawodności SRBD na dwa podproblemy : uzależniony od zastosowania i niezależny od zastosowania. Rozbicie to możliwe jest dzięki wprowadzeniu koncepcji transakcji. W rozdziale 2 zaznaczyliśmy, że transakcja zachowuje spójność rozproszonej bazy danych. W rozdziale 3.1. rozszerzyliśmy tą własność na zbiór transakcji wymagając, aby jego realizacja współbieżna spełniała kryterium uszeregowalności. Podamy obecnie dwie następne cechy transakcji, które są istotne z punktu widzenia niezawodności :
Atomowość, którą rozumiemy w ten sposób, iż wykonane muszą być albo wszystkie operacje składające się na transakcję, albo żadna z nich nie może być wykonana. (W szczególności oznacza to, że w przypadku braku możliwości dokończenia wykonywania transakcji np. ze względu na upadek stanowiska SRBD, skutki wykonanych do tej pory operacji transakcji muszą być usunięte).
Trwałość, która oznacza, że wyniki poprawnie zakończonej transakcji są na stałe wprowadzone do RBD i nie zostaną utracone nawet na skutek upadku systemu.
Niezawodność w zakresie niezależnym od zastosowania rozumiemy obecnie jako wymaganie zachowania przez transakcje cech : atomowości i trwałości. Oczywiście oznacza to konieczność włączenia do SRBD odpowiednich procedur zapewniających te cechy. W odniesieniu do zapewnienia atomowości transakcji, badania koncentrują się na poszukiwaniu tak zwanych protokołów akceptacji transakcji (ang. commitment protocols), których zadaniem jest zagwarantowanie, że uaktualnienie rozproszonej bazy danych nastąpi na wszystkich stanowiskach SRBD, których dotyczy transakcja lub na żadnym z nich. Oczywiście główna trudność polega na tym, aby gwarancja ta rozciągała się również na przypadki różnorodnych uszkodzeń SRBD (np. upadek stanowiska, strata komunikatu, itp). Ważnym aspektem badań nad protokołami akceptacji transakcji jest dążenie do uzyskania maksymalnej dostępności systemu, to znaczy możliwości jego działania nawet w przypadku częściowego upadku.

Natomiast w zakresie zapewnienia trwałości transakcji, badania koncentrują się na technikach odtwarzania RBD (ang. distributed database recovery). Chodzi tu o techniki, które pozwalają

na odtworzenie RBD po uszkodzeniu powodującym częściową utratę jej zawartości. Równorzędną rolę w tych badaniach odgrywa poszukiwanie właściwych algorytmów odtwarzania oraz struktur danych umożliwiających odtworzenie.

Drugi podproblem niezawodności SRBD - niezawodność w zakresie uzależnionym od zastosowania - rozumiemy przede wszystkim jako wymaganie zachowania cechy spójności transakcji, czyli zapewnienia respektowania przez transakcje ograniczeń integralnościowych. Rozwiązanie tego problemu odbywa się na drodze specyfikacji, weryfikacji i testowania programów transakcji oraz sprawdzania w czasie rzeczywistym czy ograniczenie integralnościowe nie są naruszane.

BIBLIOGRAFIA

Ponieważ nie jesteśmy w stanie podać reprezentatywnej bibliografii dla poruszanej problematyki, która liczy setki pozycji, podamy jedynie pewne najnowsze prace, w których można znaleźć szczegółowe odwołania.

Ogólne informacje na temat SRBD, w szczególności na temat ich architektury oraz zakresu problematyki SZRBD :
S. Ceri, G. Pelagatti, Distributed Databases. Principles and Systems, Mc Graw-Hill Book Company, New York, 1984

Problematyka spójności RBD :
C.J. Date, An Introduction to Database Systems Volume II, Addison-Wesley Pub.Co., 1982, 1983

Klasyczna praca poświęcona problematyce synchronizacji transakcji w SRBD to :
P.A. Bernstein, N. Goodman, Concurrency Control in Distributed Database Systems, ACM Computing Surveys, Vol.13, No.2, 1981, pp. 185-221.

Najnowsze wyniki dotyczące synchronizacji znaleźć można w :
W. Cellary, T. Morzy, Problemy i metody synchronizacji w systemach rozproszonych baz danych, (przedłożone do druku Archiwum Automaty-

ki i Telemechaniki).

Problematyka konfliktów działania SRBD i metod zapobiegania nim przedstawiona jest w :
W. Cellary, T. Morzy, Locking with Prevention of Cyclic and Infinite Restarting in Distributed Database Systems, 11 th Int. Conf. on VLDB, Stockholm 1985, pp. 115-126.

Problematyka optymalizacji generowania planów wykonywania transakcji w systemach baz danych, w tym również rozproszonych opracowana jest w :
W. Kim, D.S. Reiner, D.S. Batory (Editors) Query Processing in Database Systems, Topics in Information Systems, Springer-Verlag Pub. Co. 1984.

Najnowsze wyniki w tym zakresie odnoszące się do SRBD wraz z badaniami efektywności można znaleźć w :
Z. Królikowski, Optymalizacja planów wykonywania transakcji w systemach rozproszonych baz danych,
Rozprawa Doktorska, Wydział Elektroniki, Politechnika Gdańska, 1985.

Dwie podstawowe prace poświęcone problematyce niezawodności w SRBD :
W.H. Kohler, A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems, ACM Computing Surveys, Vol.13, No.2, 1981, pp.149-184.
Z. Haerder, A. Reuter, Principles of Transaction-Oriented Database Recovery, ACM Computing Surveys Vol.15, No.4, December 1983, pp. 287-317.

Jesienna Szkoła PTI
listopad 1985

Cm* - PRZYKŁAD KOMPUTERA WIELOPROCESOROWEGO O STRUKTURZE HIERARCHICZNEJ

mgr Jarosław Deminet
Instytut Informatyki
Uniwersytetu Warszawskiego
PKiN skr. poczt. 1210
00-901 Warszawa, tel. 200211/4021

1. Wstęp

Cm* jest wielomikroprocesorowym komputerem, zbudowanym w latach 1975 - 1980 na Uniwersytecie Carnegie-Mellon (CMU) w Pittsburghu i używanym do chwili obecnej. Przez te 10 lat stanowił podstawę wielu eksperymentów, zarówno sprzętowych jak i programowych. W referacie spróbuję krótko przedstawić najciekawsze z nich.

2. Historia

Czynne zainteresowanie systemami wieloprocesorowymi w CMU zaczęło się na początku lat siedemdziesiątych wraz z pojawieniem się minikomputerów PDP 11/20. Powstały wówczas komputer C.mmp miał 16 procesorów i 16 bloków pamięci połączonych ze sobą poprzez pełną krzyżownicę (ang. crossbar).

C.mmp był bardzo intensywnie wykorzystywany. Początkowo liczono wręcz na to, że stanie się podstawowym narzędziem pracy.

Bardzo dużo trudu włożono w oprogramowanie narzędziowe (co najmniej kilkanaście osób przez przeszło 5 lat). Stworzono kompletny system operacyjny Hydra, skrośny kompilator języka Bliss-11, skrośny asembler, specjalny program konsolidujący, program obsługi łącza sieci Arpanet itp. Niestety jednak w momencie, gdy cały sprzęt i oprogramowanie były już gotowe, stały się moralnie przestarzałe. Podstawową wadą była mała przestrzeń adresowa (64 kB). Wprawdzie całkowita pamięć programów mogła być większa, ale wymagało to skomplikowanego programowania. W międzyczasie pojawiły się nowe duże komputery i C.mmp pozostał jedynie narzędziem eksperymentalnym. W 1980 został rozebrany.

W wyniku badań nad C.mmp sformułowano kilka wniosków, które zostały wykorzystane przy następnych projektach badawczych. Oto najbardziej interesujące:

- Można zbudować komputer wieloprocesorowy przy wykorzystaniu gotowych układów minikomputerowych, lecz należy liczyć się z koniecznością znacznych modyfikacji (150 dodatkowych układów scalonych do 400-układowego procesora). Takie modyfikacje są tym trudniejsze, im jest większa skala integracji użytych układów (jest się trudniej dostać "do środka").

- W praktyce nie da się wykorzystać zachwalanej często cechy systemów wieloprocesorowych, jaką jest rzekomo możliwość dzielenia kodu programu między kilka procesów. W rzeczywistości jeden blok pamięci był zdolny obsłużyć co najwyżej 4 programy równocześnie. Ta nauka jest chyba ważna do dziś. Wprawdzie dzisiejsze pamięci są szybsze, ale i procesory korzystają z pamięci w mniejszych odstępach czasu.

- Po odrzuceniu dzielenia kodu okazuje się, że zdecydowana większość odwołań do pamięci dotyczy kodu, zmiennych lokalnych i stosu, a więc obszarów niejako prywatnych dla zadania. Jedynie 3 - 10% odwołań dotyczy pamięci dzielonej między kilka programów.

- W systemie wieloprocesorowym niezbędne okazuje się zaszycie niektórych operacji w mikroprogramach procesorów. W początkowej wersji C.mmp procesory nie były mikroprogramowane. Gdy tylko po-

jawiły się mikroprogramowane PDP 11/40, zostały one zmodyfikowane tak, aby umożliwić dopisywanie fragmentów mikroprogramu i zdefiniowano np. nowe instrukcje przesłań grupowych oraz instrukcje manipulujące pewnymi specjalnymi, chronionymi obiektami. Przyspieszyło to pracę całego systemu.

Gdy prace nad C.mmp dobiegały końca, zaczęto prowadzić przygotowania do nowego projektu, który szedłby dalej w podobnym kierunku. Założono, że architektura nowego komputera powinna pozwolić na połączenie nawet kilku tysięcy procesorów i gigabajtów pamięci operacyjnej, przy czym powinno być możliwe stosunkowo łatwe dodawanie nowych procesorów i pamięci.

W tym samym czasie (1975) firma DEC opracowała pierwsze swoje mikrokomputery LSI-11 (jest na nich wzorowana Mera 60). Są to mikrokomputery 16-bitowe, na pojedynczej płytce drukowanej, z możliwością zaadresowania do 56 kB pamięci, bez żadnych mechanizmów sprzętowej ochrony zasobów. Komunikacja z pamięcią i urządzeniami odbywa się poprzez asynchroniczną magistralę Q-bus. Na płytce procesora znajdują się 4 układy scalone tworzące wspólnie kompletny procesor oraz układy o małej i średniej skali integracji odpowiedzialne przede wszystkim za współpracę z magistralą. Sam procesor ma 8-bitową budowę wewnętrzną, co powoduje dość wolną pracę (przy cyklu zegara ok 400 ns typowa instrukcja trwa od 3 do 10 μs). Pamięć użyta w Cm* była oparta o układy dynamiczne o pojemności 4 kb, później wymienione na układy 16 kb.

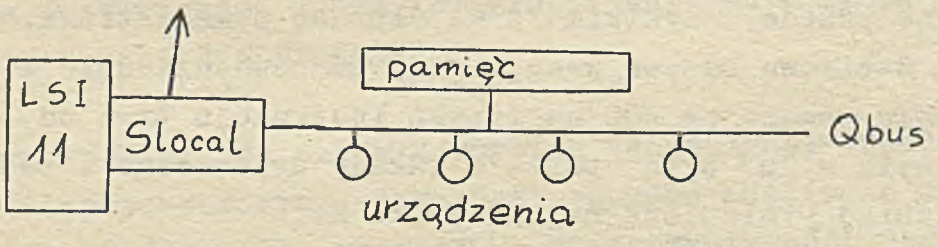
Przy opracowywaniu szczegółów architektury Cm kierowano się łatwością zaimplementowania systemu operacyjnego zbliżonego w swej koncepcji do Hydry. Nad takim systemem, o nazwie StarOS, pracowano pod kierunkiem Anity Jones. Po pewnym czasie grupa podzieliła się. Pięć osób pozostało w oryginalnym zespole, natomiast trzy przystąpiły do prac nad mniejszym i prostszym systemem, nazwanym Medusa. Założeniem twórców Medusy było maksymalne wykorzystanie możliwości architektury Cm*. Odmienność obu systemów rzuca interesujące światło na problematykę specyfikowania produktów sprzętowych i programowych.

3. Struktura

Przy opracowywaniu architektury nowego komputera trzeba było przede wszystkim rozwiązać problem struktury sieci połączeń między procesorami a pamięcią. Pełna krzyżownica oczywiście zupełnie się nie nadawała. Liczba połączeń jest w niej bowiem rzędu n^2, gdzie n jest liczbą składników. Już w C.mmp dla 16 procesorów złożoność krzyżownicy była taka, jak całej reszty konfiguracji (5 tys. układów scalonych).

Każda wersja struktury stanowi pewien kompromis między różnymi parametrami: liczbą węzłów, czasem połączenia, stopniem zrównoleglenia, prostotą ustalenia trasy połączenia itp. W efekcie dokładnej analizy zdecydowano się wybrać strukturę hierarchiczną. W ten sposób powstała koncepcja Cm*. Cm jest skrótem od ang. computer module, natomiast gwiazdka oznacza iterację (teoretycznie - nieograniczoną).

Podstawowym elementem struktury jest moduł (Cm), składający się z procesora, pamięci i ewentualnych urządzeń dołączonych do magistrali, tak jak w zwykłym mikrokomputerze (rysunek poniżej).



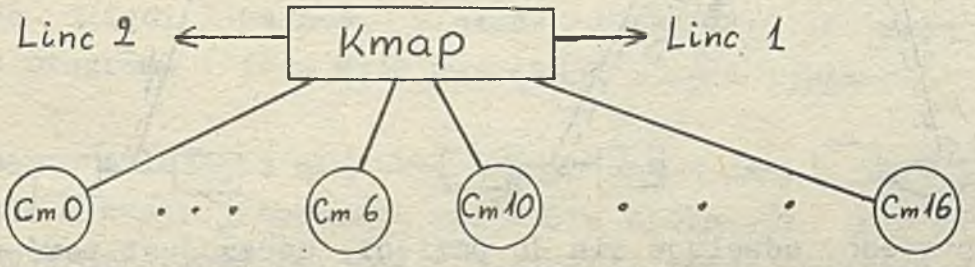
Pomiędzy procesor a magistralę włączono dodatkowy element - Slocal (ang. local switch). Slocal pełni dwie różne funkcje. Po pierwsze, wspomaga procesor, nadrabiając niektóre jego braki (np. pozwala na rozróżnienie trybu pracy programów systemowych i użytkowych, przy czym te drugie nie mają prawa wykonywać niektórych potencjalnie niebezpiecznych instrukcji). Należy podkreślić, że przy wysokim stopniu zintegrowania jakakolwiek modyfikacja procesora była bardzo trudna (potrzebne było do tego 80 układów scalonych).

Slocal rozszerza także możliwości adresowe LSI-11. Przestrzeń adresowa jest podzielona na 16 stron wirtualnych po 4 kB każda. Każda strona może być odwzorowana na dowolny blok pamięci o takim samym rozmiarze leżący w tym samym module. Slocal zawiera rejestry sterujące tym odwzorowaniem. Podobnie jak w innych kom-

puterach, istnieją osobne zestawy rejestrów dla programów pracujących w trybie systemowym i w trybie użytkowym.

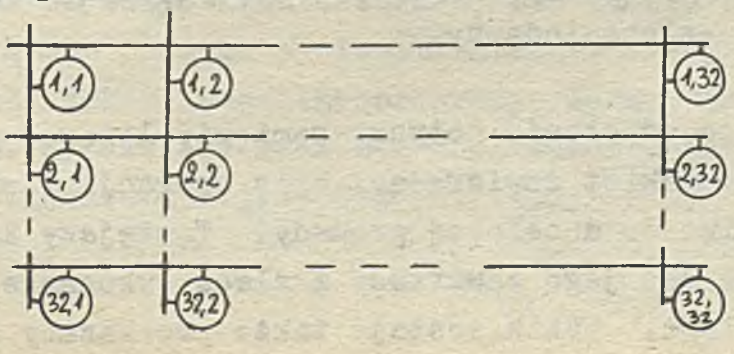
Po drugie, Slocal umożliwia komunikację modułu z resztą maszyny. W szczególności odwołania do niektórych stron (niektórych adresów) są kierowane na wyższe piętro hierarchii.

Następnym poziomem w hierarchii jest gromada (ang. cluster). Gromada obejmuje do 14 modułów połączonych z wyspecjalizowanym procesorem komunikacyjnym, zwanym Kmap (ang. mapping Kontroller, rysunek poniżej). Kmap pośredniczy w wymianie informacji między modułami. W ograniczonym zakresie Kmap może też nadzorować pracę procesorów LSI-11, np. zgłaszając przerwania.

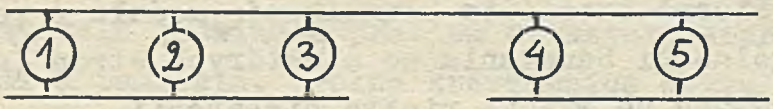


Czas cyklu procesora Kmap wynosi 160 ns. Kmap jest mikroprogramowany poziomo, a mikrorozkaz ma 80 bitów szerokości i jest zapisany w pamięci RAM, dostępnej z zewnątrz.

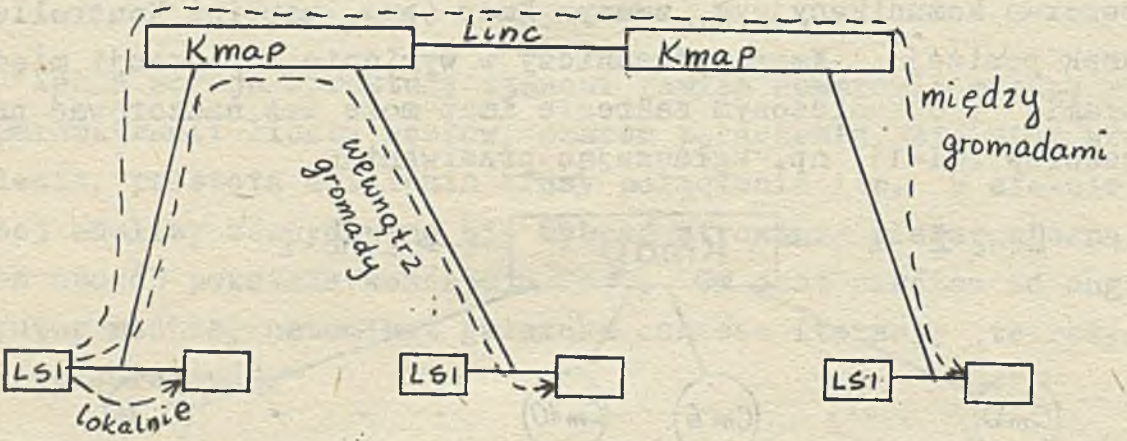
Kmap ma dwa porty, do których można dołączyć szybkie magistrale prowadzące do innych gromad (zwane Linc). Poprzez magistrale są przesyłane kilkusłowowe pakiety danych. Szybkość transmisji wynosi ok. 2 Mb/s. Koncepcja Cm* nie nakłada żadnych ograniczeń na liczbę różnych magistrali Linc oraz na ich konfigurację. Ogranicza się natomiast do 64 liczbę gromad podłączonych do jednej magistrali. Przykładowym rozwiązaniem jest macierz 32 × 32 gromady, połączone 32 magistralami poziomymi i 32 pionowymi (rysunek poniżej). W takiej konfiguracji każda transmisja między gromadami wymaga co najwyżej jednego "pośrednika". Konfiguracja miałaby



ok. 13 tys. procesorów i do 4 GB pamięci operacyjnej. Rzeczywistość jest bardziej prozaiczna. Uruchomiona konfiguracja Cm* składa się z 5 gromad, 50 procesorów i 3 MB pamięci (rysunek poniżej).



Rysunek poniżej przedstawia sposób odwoływania się przez procesor do pamięci.



Gdy procesor odwołuje się do pamięci, adres jest porównywany przez Slocal z rejestrami sterującymi. Jeśli ich zawartość wskazuje, że odwołanie nastąpiło do strony, umieszczonej w lokalnej pamięci modułu, to Slocal wyznacza adres fizyczny i przekazuje go bezpośrednio na magistralę. Dalej operacja przebiega tak samo, jak w przypadku zwykłego mikrokomputera.

Jeśli odwołanie nastąpiło do strony, oznakowanej uprzednio jako nie-lokalna, to Slocal przekazuje wszystkie informacje (adres i ew. dane) do procesora Kmap. Teraz ten z kolei decyduje o dalszym losie operacji, na podstawie swojego mikroprogramu i posiadanych danych. Na ogół Kmap określa, w którym module leży adresowana pamięć i przekazuje doń odpowiednie żądanie. Slocal docelowego modułu przekazuje adres i ew. dane na magistralę. Wynik operacji (potwierdzenie zapisu lub przeczytane dane) wraca tą samą drogą, przez Kmap, do procesora-zleceniodawcy.

Jeśli Kmap uznał, że odwołanie dotyczy pamięci, leżącej w innej gromadzie, to tworzy pakiet zawierający opis operacji i przesyła ten pakiet przez Linc do docelowej gromady. Tamtejszy Kmap odbiera pakiet, interpretuje jego zawartość i zleca wykonanie operacji ustalonemu modułowi. Wynik zostaje także przekazany po-

przez Linc.

Cechą charakterystyczną tego machanizmu jest jego całkowita przezroczystość dla programu, z którego pochodzi odwołanie do pamięci. Nie ma żadnej różnicy jakościowej między odwołaniem do pamięci lokalnej, wewnątrz gromady i między gromadami. Różni się natomiast czas, w jakim żądana operacja zostanie zakończona. Odwołanie do pamięci lokalnej trwa ok. 3 μs; do pamięci wewnątrz gromady - minimum 9 μs; do pamięci w innej gromadzie - powyżej 27 μs. Dwa ostatnie parametry dotyczą warunków optymalnych; gdy liczba nielokalnych odwołań rośnie, to następuje "zatkanie" procesora Kmap i operacje mogą trwać nawet 200 μs. W praktyce z architektury Cm* wynika, że możliwie duża część odwołań powinna się odnosić do pamięci lokalnej. W szczególności tam powinien znajdować się kod programu i jego stos zawierający dane lokalne.

Kmap z założenia miał być bardzo elastyczny i uniwersalny; możliwość mikroprogramowania zapewniła spełnienie tego założenia. Zmiana mikroprogramu pozwala zrealizować różne strategie adresowania. W szczególności Kmap może rozpoznawać odwołania do niektórych adresów jako żądania wykonania specjalnych funkcji. W najprostszych przypadkach mogą to być żądania zmiany zawartości wewnętrznych rejestrów opisujących przekształcanie adresów. W bardziej skomplikowanych - żądania przesłania bloku kilkuset słów pod wskazany adres, być może w odległym module. Wreszcie Kmap może sam obsługiwać struktury danych, np. kolejki i stosy: zapisanie słowa pod wskazany adres powoduje dodanie go do stosu, a przeczytanie spod tego adresu - pobranie wierzchołka stosu.

Istnieją co najmniej trzy w pełni sprawdzone mikroprogramy. Jeden bardzo prosty, stosowany to testowania sprzętu i do prostych eksperymentów programowych, pozwalał na adresowanie dowolnych słów pamięci we wszystkich modułach, bez żadnej ochrony. Dwa pozostałe są związane z dwoma systemami operacyjnymi, opisanymi poniżej. Istniały także inne mikroprogramy, mające charakter eksperymentalny, np. symulujące sieć bez możliwości dzielenia pamięci lub związane z konkretnym językiem (np. Algol 68).

Cm* został obudowany aparaturą wspomagającą uruchamianie sprzętu i oprogramowania. Przede wszystkim dodatkowe mikrokomputery LSI-11 są podłączone do procesorów Kmap i pozwalają na zapisanie ich mikroprogramu, oglądanie stanu rejestrów wewnętrznych, zapisywanie do nich nowych wartości, ustawianie punktów przerwań i pracę krokową.

Użytkownik komunikuje się z modułami poprzez 10 linii szeregowych (po 2 na gromadę). Wszystkie te linie są zarządzane przez dedykowany komputer PDP 11/20, zwany Cm* Host. Host pozwala dodatkowo zatrzymać i uruchomić procesory w poszczególnych modułach, a także załadować do nich programy.

Oprogramowanie i mikrooprogramowanie dla Cm* było przygotowywane skrośnie, na komputerze DEC-10 a następnie transmitowane - początkowo przez linie szeregowe, a potem przez dwa szybkie łącza pamięciowe.

W miarę upływu czasu Cm był rozbudowywany o nowe urządzenia, mające często charakter eksperymentalny (np. inteligentny, mikroprogramowany sterownik dyskowy; bardzo szybki monitor ekranowy; łącze sieci lokalnej Ethernet). Ich opis wykracza poza zakres referatu.

4. System StarOS

StarOS jest systemem operacyjnym opartym na dojściach ang. capability . Dojście można interpretować jako wskaźnik do obiektu, określający dodatkowo typ obiektu i zbiór operacji, którymi dysponuje posiadacz dojścia. W szczególności posiadacz może ale nie musi mieć prawo powielania dojścia oraz usuwania go. Dojścia są przechowywane w pamięci, ale dostęp do nich jest możliwy tylko za pośrednictwem operacji implementowanych przez Kmap. Każdy obiekt może składać się z części "danych" dostępnych bezpośrednio i z części "dojść".

Struktura zbioru obiektów systemu StarOS odzwierciedla strukturę komputera. Każda gromada jest traktowana jako osobne pańs-

two, mające własny program tworzący obiekty i zarządzający posiadaną pamięcią. Każdy tworzony obiekt jest opisywany przez pozycję słownika. Położenie obiektu w słowniku jest stałe przez cały czas istnienia obiektu, nawet jeśli obiekt został w międzyczasie przesunięty, choćby i do innego modułu (pozostając jednak wewnątrz gromady). Dojścia wskazują na obiekt podając numer gromady oraz numer pozycji w słowniku.

Podstawowymi obiektami są strony pamięci. Tylko do nich program może mieć szybki dostęp za pośrednictwem zwykłych instrukcji procesora. Jeśli strona pamięci znajduje się w module, wykonującym program, to Kmap przy pierwszym odwołaniu zapisuje odpowiednią wartość do rejestru wewnątrz przełącznika Slocal tak, aby następne odwołania były wykonywane całkowicie lokalnie. W odwołaniach do stron położonych w innych modułach, oraz do wszelkich innych obiektów, niezależnie od ich położenia, musi pośredniczyć Kmap. Jednak i tu stosuje się różne formy optymalizacji, np. Kmap zapisuje w swojej szybkiej pamięci roboczej pomocniczą informację o fizycznym położeniu obiektu.

Komunikacja między programami odbywa się za pośrednictwem tzw. skrzynek pocztowych (ang. mailbox). Poszczególne programy mogą mieć dojścia do różnych skrzynek, z prawem wysyłania lub odbierania komunikatów. Każdy komunikat może zawierać zarówno zwykłe zmienne, dostępne bezpośrednio dla procesora, jak i dojścia.

Początkowo w systemie istnieje kilka programów usługowych, np. program tworzący obiekty, program ładujący i system plików. Każdy program sam jest obiektem - część pamięciowa zawiera parametry, natomiast część dojściowa - dojścia do stron pamięci zawierających kod i dane programu oraz do skrzynek pocztowych używanych do komunikacji.

Aby użytkownik mógł skorzystać z systemu, StarOS tworzy dla niego program - interpreter komend. Po zidentyfikowaniu użytkownika, na podstawie uprzednio zapisanych informacji, program ten tworzy inicjalny zbiór dojść posiadanych przez użytkownika, definiujący zbiór obiektów dostępnych dlań. Niektóre obiekty zawsze są dostępne dla wszystkich - np. skrzynki pocztowe programów usłu-

gowych.

Jeśli użytkownik zażądał utworzenia nowego programu, to interpreter komend przekazuje to żądanie poprzez skrzynkę do programu ładującego. Żądanie musi określać, do jakich obiektów nowy program ma mieć dostęp, a także musi zawierać dojście do skrzynki, poprzez którą program ładujący ma potwierdzić wykonanie zlecenia.

Program ładujący po przyjęciu żądania rozpoczyna od zajęcia odpowiedniego obszaru pamięci na sam program oraz na jego skrzynki i strony pamięci. Aby to osiągnąć, program wysyła żądanie do programu obsługi obiektów i powrotną pocztą otrzymuje dojścia do nowych obiektów. Po utworzeniu i wypełnieniu inicjalną zawartością stron nowego programu, program ładujący tworzy obiekt - program. Odbywa się to poprzez tzw. wzmocnienie dojścia. System zostaje poinformowany o tym, że obiekt, który do tej pory był zwykłą stroną, od tej pory ma typ "program". Posiadanie dojścia do takiego obiektu nie uprawnia do swobodnego modyfikowania go - można tylko zlecać dokonanie na nim operacji (np. zmiany parametrów) programowi, który utworzył obiekt (w przypadku programów - programowi ładującemu). StarOS pozwala na dynamiczne definiowanie nowych typów i programów ich obsługi.

Po wpisaniu parametrów i dojść do dostępnych obiektów do obiektu-programu, dojście do niego zostaje odesłane powrotną pocztą użytkownikowi. Ten może zlecić wykonanie programu, przesyłając dojście programowi szeregującemu, który z kolei według ustalonej strategii przydziela poszczególnym programom czas poszczególnych procesorów i wpisuje odpowiednie parametry w obszarze dzielonym z jądrem systemu. To ostatnie dokonuje faktycznego uruchamiania i wywłaszczania programów.

Mechanizm dojść szczególnie dobrze pasuje do architektury wieloprocesorowej. Bardzo łatwo można zorganizować duży zbiór programów, dzielących w dowolny sposób swe zasoby, położone w różnych modułach, a nawet gromadach. Dzięki temu, że program odwołuje się do systemu za pośrednictwem skrzynek, istnieje możliwość dzielenia pracy programów systemowych (np. może istnieć kilka programów obsługi plików, każdy dostępny dla innych programów użytko-

wych).

Od początku zakładano, że użytkownik będzie tworzył zespoły programów (ang. task force) współpracujących nad wykonaniem określonych zadań. StarOS sam powinien umieć tak rozmieścić różne obiekty (programy, skrzynki, strony pamięci) aby zoptymalizować wydajność systemu. W praktyce jednak nie udało się tego osiągnąć.

Aby zobrazować problemy związane z architekturą systemu hierarchicznego, poświęcimy kilka zdań odśmiecaniu. Obiekt, do którego nie istnieją żadne "żywe" dojścia, może zostać usunięty. Znajdowanie martwych obiektów jest zadaniem odśmiecacza. Trzeba się jednak liczyć z tym, że równolegle z odśmiecaczem będą pracowały inne programy, które mogą kopiować posiadane dojścia lub tworzyć nowe obiekty. Aby uniknąć zmylenia odśmiecacza przyjęto zasadę, że w czasie jego pracy Kmap specjalnie oznacza "na żółto" nowe obiekty oraz nowe dojścia. Wskazywane obiekty nie zostaną na pewno usunięte w tym cyklu odśmiecania. To jednak nie wystarcza: kopiowaniem dojść zajmuje się Kmap, a jego jurysdykcja rozciąga się tylko na jedną gromadę. Są więc kłopoty z kolorowaniem obiektów położonych w innych gromadach. Przyjęto zatem dodatkowo malowanie "na czerwono" obiektów, do których istnieją dojścia spoza gromady. W czasie normalnego odśmiecania czerwone obiekty są ignorowane. Dopiero gdy ich liczba zbyt wzrośnie, będzie włączany super-odśmiecacz, działający równocześnie we wszystkich gromadach.

5. System Medusa

Medusa jest systemem prostszym i mniej elastycznym niż StarOS. Przede wszystkim z góry jest określony zbiór możliwych typów obiektów i każdy z nich jest obsługiwany przez program stanowiący część systemu operacyjnego. Każdy obiekt jest opisywany przez deskryptor, podobny w swej koncepcji do dojścia. Deskryptory mogą jednak być zapisane tylko w specjalnych obiektach typu "lista deskryptorów".

Podstawowym obiektem dynamicznym jest zespół (ang. task force)

obejmujący od kilku do kilkudziesięciu procesów. Struktura zespołu jest sztywna - proces nie może przenieść się do innego zespołu, nie może też istnieć poza zespołem. Z każdym zespołem jest związana dzielona lista deskryptorów, opisująca obiekty dostępne dla wszystkich procesów zespołu. Każdy proces ma swoją własną prywatną listę deskryptorów, dla obiektów do których tylko on ma dostęp.

Podstawowym obiektem statycznym jest strona pamięci. Może dla niej istnieć tylko jeden deskryptor, a więc może ona być dzielona przez cały zespół, albo prywatna dla jednego procesu. Dla innych obiektów liczba istniejących deskryptorów może być większa, ale nigdy nie może przekroczyć liczby ustalonej w czasie tworzenia obiektu. Medusa notuje przy każdym obiekcie adresy wszystkich jego deskryptorów. Z jednej strony ułatwia to odśmiecanie (można łatwo określić, kiedy zostaje usunięty ostatni deskryptor). Z drugiej, znacznie komplikuje koordynację pracy procesorów Kmap w systemie obejmującym kilka gromad.

Medusa traktuje całą konfigurację jednolicie. Każdy program systemowy obsługuje wszystkie procesy, być może położone w kilku gromadach. Do komunikacji procesów z systemem oraz procesów między sobą służą potoki, podobne do potoków w Unixie. Każdy proces systemowy ma kilka potoków wejściowych, odpowiadających różnym funkcjom. Deskryptory tych potoków są zapisane w systemowych listach przydzielonych każdemu modułowi. Zlecenie wykonania funkcji jest interpretowane przez Kmap jako wysłanie komunikatu poprzez odpowiedni potok.

Autorzy zwrócili szczególną uwagę na podział systemu na zespoły i na zdefiniowanie funkcji wewnątrz zespołów. Chodziło o to, aby pogodzić koncepcję podziału na warstwy (funkcje należące do warstwy wyższej korzystają z funkcji należących do warstwy niższej) z podziałem tematycznym (np. funkcje związane z obsługą plików powinny móc łatwo korzystać ze wspólnych danych). Poza tym należało zapewnić, że niezależnie od obciążenia system nie zablokuje się. Ostatecznie każdy zespół systemowy został podzielony na warstwy, każda związana z jednym potokiem danych. Z kolei cały zbiór warstw wszystkich zespołów tworzy acykliczny graf wywołań. Poza uniknięciem blokady ułatwia to także obsługę sytuacji awaryjnych.

Formalne usankcjonowanie istnienia zespołów spowodowało konieczność innego spojrzenia na szeregowanie procesów. Zazwyczaj struktura zespołu jest taka, że wstrzymanie jednego procesu może spowodować spowolnienie lub wręcz zablokowanie pozostałych procesów. Zagadnienie przydziału procesorów dla procesów zespołu staje się podobne do zagadnienia przydziału i wymiany stron w pamięci wirtualnej, z groźbą migotania włącznie. Aby tego uniknąć, wprowadzono dwie zasady. Po pierwsze, system stara się szeregować całe zespoły, równocześnie przydzielając procesory wszystkim procesom zespołu (można to porównać do koncepcji pola roboczego). Po drugie, nawet gdy proces zawiesza się w oczekiwaniu na jakieś zdarzenie zewnętrzne, system przez pewien czas nie przydziela procesora innemu procesowi, aby umożliwić szybkie wznowienie procesu np. po otrzymaniu komunikatu od innego członka zespołu. Dostrajanie parametrów takiego systemu wydaje się jednym z najbardziej fascynujących zagadnień związanych z systemami wieloprocesorowymi.

Twórcy Medusy poświęcili wiele uwagi problemowi reagowania na błędy oprogramowania i awarie sprzętu. Wprowadzono m.in. pojęcie "kumpli" (ang. buddy). Każdy proces może zażądać, aby w razie powstania sytuacji wyjątkowej zawiadamiać jego kumpla, którym może być dowolny inny proces wewnątrz zespołu. Zakłada się bowiem, że proces - ofiara może nie być w stanie sam poprawić swego stanu (np. awaria mogła spowodować uszkodzenie kodu). Proces - kumpel w czasie pomagania procesowi - ofierze ma pełny dostęp do wszystkich obiektów tego ostatniego.

Medusa była tworzona bardzo efektywnie, z zastosowaniem wszelkich reguł "sztuki dobrego programowania". Właściwy system operacyjny (nie licząc mikroprogramu i oprogramowania skrośnego) ma ok. 20 tys. linii w Blissie. Wszystkie te programy zostały najpierw w całości zaprojektowane i napisane, a dopiero potem rozpoczęto ich uruchamianie. Z jednej strony spowodowało to konieczność sporych modyfikacji dalszych programów w miarę uruchamiania poprzednich, ale z drugiej pozwoliło skrócić okres uruchamiania do ok. 2 miesięcy. Uruchamianie rozpoczęto od programów testujących i śledzących. Przez cały czas bezwzględnie przestrzegano zasady zgodności programów z dokumentacją i niewprowadzania ad hoc ulepszeń i uzupełnień zmieniających specyfikację programów. Przest-

rzeganie tych zasad pozwoliło np. autorowi referatu w ciągu dwóch tygodni przejąć odpowiedzialność za kontynuowanie uruchamiania, modyfikowania i pielęgnowania systemu.

6. Problematyka językowa

Zdecydowana większość oprogramowania dla Cm* (w tym oba systemy operacyjne) została napisana w języku Bliss-11. Nie zapewniał on wprawdzie żadnych specjalnych mechanizmów programowania równoległości, ale dzięki temu był bardzo elastyczny, a poza tym istniał jego doskonały kompilator skrośny. Dla użytkowników Medusy został przeniesiony język C.

W początkowej fazie prac na Cm* został przeniesiony z C.mmp kompilator Algolu 68. Kompilator ten wykrywał możliwe zrównoleglenia w programie (np. niezależne obliczanie podwyrażeń) i generował odpowiedni kod. Okazało się jednak, że w ten sposób można zająć najwyżej kilka procesorów; wewnętrzne zrównoleglenie wewnątrz programu sekwencyjnego jest niewielkie.

Najbardziej udanym językiem stworzonym specjalnie dla Cm* był AMPL, łączący w sobie synchronizację przepływem danych ze strukturalnością Moduli i pewnymi koncepcjami CSP. Moduły mogą się ze sobą komunikować tylko poprzez komunikaty, wysyłane do różnych skrzynek odbiorczych. Komunikaty mogą oznaczać żądanie wykonania pewnych operacji i zwrócenia wyniku. Eksperymenty dowiodły, że dla niektórych problemów programy w AMPL, korzystające z systemu Medusa, działają niewiele wolniej niż standardowe zespoły korzystające z dzielonej pamięci.

7. Eksperymenty

W oparciu o Cm* przeprowadzono sporo eksperymentów programowych. Na próbnej 10-modułowej wersji prowadzono doświadczenia z algorytmami, badanymi uprzednio na C.mmp: sortowaniem, iteracyjnym rozwiązywaniem układów równań różniczkowych, programowaniem całko-

witoliczbowym i rozpoznawaniem mowy. Powstały pierwsze, kolejkowe modele zachowania się skomplikowanej struktury Cm*. Badano czas obsługi typowego żądania przy znanej częstotliwości odwołań do dzielonej pamięci. Później, w miarę stabilizowania się systemów operacyjnych, liczba użytkowników rosła. Niestety jednak nie udało się zwabić na Cm* żadnych użytkowników zainteresowanych dużymi projektami, dla których Cm* byłby narzędziem do uruchamiania programów np. z zakresu sztucznej inteligencji albo grafiki. Wszyscy użytkownicy Cm* byli zainteresowani wieloprocesorowością samą w sobie i z natury rzeczy starali się uruchamiać proste, przykładowe programy.

Zachowanie się algorytmów uruchamianych na Cm* było określone przez dwie główne cechy architektury: asynchronicznością pracy poszczególnych procesorów i niejednorodnością pamięci. Widać to najlepiej na przykładzie algorytmu rozwiązywania układu równań różniczkowych, polegającego na iteracyjnym przetwarzaniu kwadratowej macierzy. Macierz ta była zapisana w pamięci, a każdy proces odpowiadał za iterację jednego prostokątnego fragmentu. Zdecydowanie najlepiej działał algorytm całkowicie asynchroniczny, w którym iteracje przebiegały niezależnie jedna od drugiej. Dla wielu procesów pracujących w różnych gromadach liczby iteracji dla poszczególnych fragmentów mogły się różnić nawet o rząd wielkości (w czasie, gdy proces umieszczony w tym samym module, co macierz, wykonał 20 tys. iteracji, proces w innej gromadzie wykonał ich tylko ok. 2 tys.). Okazało się, że wprawdzie algorytm gwarantuje zawsze zbieżność obliczeń, ale szybkość zbieżności zależy istotnie od tego, które fragmenty macierzy są obsługiwane przez najbliższe (a więc i najszybsze) procesy. Bardzo istotne okazało się także rozdzielenie macierzy między kilka modułów, co pozwoliło uniknąć wąskiego gardła powstającego przy dostępie kilkudziesięciu procesów do jednej pamięci. Po starannym dostrojeniu parametrów udało się doprowadzić do prawie liniowego przyspieszenia. Później opracowano jeszcze lepsze wersje algorytmu, korzystające z podziału macierzy na sześciokąty i minimalizujące obszar dzielonej pamięci.

Algorytm szybkiego sortowania okazał się natomiast zupełnie

nieodporny na problemy związane z dostępem do dzielonej pamięci. Już przy kilkunastu procesach rozdzielonych między dwie gromady wydajność systemu przestawała rosnąć. W pierwszej chwili może natomiast zaskakiwać fakt, że punkt równowagi między sortowaniem szybkim i bąbelkowym (tzn. punkt, poniżej którego należy przejść na to drugie) okazał się być w tym samym miejscu, co i dla sortowania sekwencyjnego (n = 10).

Podjęto próbę skonstruowania równoległego oprogramowania do symulowania zdarzeń, wzorowanego na Simuli 67. Okazało się to trudne i niezbyt eleganckie, ale możliwe. Przy tej okazji przeprowadzono eksperymenty dotyczące optymalnego rozmieszczania procesów w modułach, przy założeniu że kod procesu musi znajdować się w tym samym module, co wykonujący go procesor. Ręczna optymalizacja pozwoliła zmniejszyć czas pracy o ok. 30%, co wskazuje na wagę problemu.

Poważniejszą pracą była symulacja systemu energetycznego, składającego się z węzłów o znanej charakterystyce (napięcie, obciążalność) i linii przesyłowych. Symulacja polega na iteracyjnym rozwiązywaniu układu równań liniowych z uwzględnieniem zachowania się węzłów. Okazało się, że standardowo używany algorytm zawiera w sobie sporo równoległości i łatwo dał się przenieść na Cm*. W efekcie przyspieszenie algorytmu było prawie liniowe, a szybkość wersji 6-procesorowej bliska szybkości programu pracującego na komputerze DEC-20.

Uruchomiono także kilka algorytmów związanych z modelowaniem procesów chemicznych, m.in. analizę zachowania się cząsteczek wody przy zastosowaniu metody Monte-Carlo. Dla 45 procesorów osiągnięto 30-krotne przyspieszenie. Należy jednak zaznaczyć, że LSI-11 ma bardzo powolną arytmetykę zmienno-pozycyjną, zajmującą większość czasu procesora, a więc odwołania do danych następują w programach numerycznych z niewielką częstotliwością.

Podobne ograniczenie odnosi się do interpretacji eksperymentu z szybką transformacją Fouriera (FFT). Podzielono w nim 1024-elementowy wektor na 32 równe odcinki i każdy proces dwukrotnie wyko-

nywał obliczenia dla swojego odcinka, w międzyczasie wymieniając zawartość swojego odcinka z innymi. Udało się osiągnąć przeszło 25-krotne przyspieszenie w stosunku do wersji jednoprocesorowej, lecz użycie szybszego procesora (np. 11/23 z procesorem zmiennopozycyjnym) zapewne pogorszyłoby ten wynik.

8. Refleksje

Realizacja Cm* nie dała pełnej odpowiedzi na pytanie, czy opłaca się korzystać z gotowego sprzętu przy projektowaniu systemów wielomikroprocesorowych. W trakcie przygotowywania oprogramowania systemowego programiści bez przerwy mocowali się np. z ograniczeniami wynikającymi z niedostępności wnętrza LSI 11 dla procesorów Kmap, które ze względu na swą wydajność i położenie pełniły rolę nadzorczą.

Cm* jest z pewnością systemem niezrównoważonym. W przeciwieństwie do procesorów Kmap procesory LSI 11 z perspektywy wydają się zdecydowanie zbyt słabe, aby z nich budować tak złożony komputer. Dostępne już wkrótce później procesory LSI 11/23 (a zwłaszcza jednokostkowe 11/73) na pewno uprościłyby konstrukcję modułu, oferując mechanizmy ochrony zasobów, zarządzanie 22-bitową przestrzenią adresową itp.

Kmap był zaprojektowany wyłącznie w oparciu o układy małej i średniej skali integracji. W dwa - trzy lata później projektanci użyliby zapewne procesorów segmentowych i matryc programowalnych, co pozwoliłoby zmniejszyć rozmiary, pobór mocy i koszt (oryginalny Kmap kosztował 7 tys. $; koszt jednego modułu wynosił 6 tys.$: 3 tys. za LSI 11 i Slocal, 3 tys. za 128 kB pamięci).

Brak szerokiego zainteresowania społeczności uniwersyteckiej korzystaniem z Cm* był spowodowany przede wszystkim małą przestrzenią adresową i powolną arytmetyką. Właściwie każdy poważny problem wymagałby wyjścia poza 64 kB i programowego manipulowania przestrzenią adresową, a to stanowiłoby poważne utrudnienie, jak tego dowiodły liczne doświadczenia z C.mmp, m.in. z programem roz-

poznawania mowy. Gdy więc w pobliżu stał VAX 11/780, choć jednoprocesorowy, to jednak z 32-bitową adresacją i z prawdziwą arytmetyką zmiennopozycyjną, to użytkownicy wybierali właśnie jego.

Eksperymenty programowe pokazały, że problemy związane z niejednorodnością struktury komputera mogą być fascynujące. Dzięki modyfikowalności adresacji można dodatkowo eksperymentować np. z różnymi zasymulowanymi strukturami sieciowymi.

Wydaje się, że byłby możliwy powrót do koncepcji zbliżonej do Cm* , z wykorzystaniem nowych procesorów 32-bitowych (68020, 32032), pamięci 256 kb, macierzy programowalnych i ew. specjalnie wypiekanych układów VLSI. Można by się wtedy przekonać, jak pracuje system złożony np. z tysiąca modułów, z których każdy samodzielnie posiada w przeciwieństwie do LSI 11 pokaźną moc obliczeniową.

Konkurencyjnym rozwiązaniem jest konstrukcja sieci komputerowej (opartej np. o Ethernet) wyposażonej w jeden system operacyjny, pozwalający na automatyczny podział zasobów, w tym także czasu poszczególnych procesorów. W tym kierunku idą obecne prace w CMU.

BIBLIOGRAFIA

Różne aspekty badań związanych z Cm są opisane w kilkudziesięciu raportach CMU. Poniżej zestawiono tylko prace przeglądowe oraz prace publikowane w czasopismach o większym zasięgu.

Fuller, S.H., Jones, A.K., Durham, I. (red.) "Cm* Review", CMU, Jun 1977

Jones, A.K., Schwarz, P. "Experience using multiprocessor systems: a status report", Computing Surveys 11(2), June 1980

Jones, A.K., Gehringer, E.F. "The Cm* multiprocessor project: a research review", CMU, July 1980

Fuller, S.H. et al. "Multi-microprocessors: An overview and working example", Procs. IEEE 66(2), Feb 1978

Jones, A.K. et al. "Programming issues raised by a multiprocessor", Procs. IEEE 66(2), Feb 1978

Deminet, J. "Experience with multiprocessor algorithms", IEEE Trans. on Computers, C-31, Apr 1982

Ousterhout, J.K., Scelza, D.A., Sindhu, P.S. "Medusa: An experiment in distributed operating system structure", Comm. ACM, 23(2), Feb 1980

Jesienna Szkoła PTI
Listopad 1985

UMOWY W ZAKRESIE INFORMATYKI

Jacek Irlik
Centrum Techniki Obliczeniowej
Uniwersytet Śląski
ul. Uniwersytecka 4
40-007 Katowice

1. Wprowadzenie do problematyki umów

Umowa jest powszechnie uważana za najbardziej dogodny instrument kształtowania stosunków gospodarczych. Dlaczego tak jest? Umowa stanowi jedno ze źródeł stosunku zobowiązaniowego, t.j. stosunku, w którym jedna ze stron (wierzyciel) może żądać od drugiej (zwanej dłużnikiem) spełnienia określonego świadczenia i w którym ta druga powinna to świadczenie spełnić (art.353§1 kodeksu cywilnego). Umowy najczęściej rodzą stosunek wzajemny, tzn. każda ze stron jest równocześnie dłużnikiem i wierzycielem w odniesieniu do wzajmnych świadczń, które powinny mieć w zasadzie charakter ekwiwalentny. Stosunek zobowiązaniowy, którego źródłem jest umowa, powstaje i zostaje ukształtowany wolą stron, które z mocy prawa uzyskują właściwą w tym celu kompetencję. Dzięki tej właśnie kompetencji powstały stosunek jest prawnie chroniony oraz rodzi prawem określone sankcje w przypadku jeśli strony postępują niezgodnie z jego treścią. Fakt, że powstanie i ukształtowanie stosunku zobowiązaniowego wynika z woli stron ma ogromne znaczenie. Mogą one mianowicie ukształtować go w sposób najbardziej dla siebie dogodny, odpowiadający najlepiej ich interesowi gospodarczemu. Jak zawiera się umowę? W życiu codziennym zawieramy setki

umów często nie zdając sobie z tego sprawy. Istotą umowy jest konsens - zgodne oświadczenie woli stron umowy dotyczące woli jej zawarcia oraz jej postanowień, z których wynikają wzajemne uprawnienia i obowiązki stron. Częste nieuświadomienie faktu, że właśnie zawarliśmy umowę wynika z tego, że oświadczenie woli zawarcia umowy oraz woli zawarcia w niej określonych postanowień może być wyrażone nie tylko w sposób wyraźny, ale również przez takie zachowanie się, które wolę tę ujawnia w sposób dostateczny (art.60 k.c.). Oświadczenie woli może więc być tzw. oświadczeniem dorozumianym. Może nim być gest, słowo, mimika czy wreszcie po prostu podjęcie czynności, którą czy które umowa przewiduje.

Przy zawarciu umowy - jak wspomnieliśmy - istotny jest konsens, tzn zgodne oświadczenie woli jej zawarcia oraz woli zawarcia w niej określonych postanowień. W jaki sposób doprowadzić można do osiągnięcia takiego konsensu? Najczęstszym sposobem jest postępowanie polegające na tym, że jedna ze stron występuje do drugiej z ofertą, t.j. z oświadczeniem woli zawarcia umowy, w którym określone są jej istotne postanowienia (art.66§1 k.c.), a druga oświadcza, że ofertę przyjmuje bez zastrzeżeń. Należy przy tym zaznaczyć, że przyjęcie oferty z zastrzeżeniem zmiany lub uzupełnienia jej treści jest nową ofertą skierowaną w kierunku odwrotnym (art.68 k.c.). W niektórych przypadkach do zawarcia umowy dochodzi w wyniku rokowań, których celem jest wspólne uzgodnienie wszystkich postanowień umowy (art.72 k.c.).

W jakiej formie zawiera się umowy? W zasadzie można je zawierać w formie dowolnej. Umowę można zawrzeć ustnie, można nadać jej formę pisemną lub inną formę szczególną (np. formę aktu notarialnego). Jeżeli przepisy dotyczące formy oświadczenia woli dla konkretnej sytuacji nie stanowią odmiennie, to wymóg formy pisemnej ma znaczenie jedynie dla celów dowodowych i brak tej formy nie powoduje nieważności umowy. Można sądzić, że właśnie dlatego strony - zawierając umowę ustnie - nie uświadamiają sobie, że umowę zawarły. Należy zaznaczyć, że wymóg formy pisemnej (dla celów dowodowych) zastrzeżony jest jedynie dla niektórych oświadczeń woli, np. tych, któreb dotyczą wartości majątkowej powyżej dziesięciu tysięcy złotych (art.75§1 k.c.). W obrocie uspołecznionym wymóg formy pisemnej jest powszechny i wynika z tzw. ogólnych warunków

umów, stanowionych przez naczelne organy administracji państwowej dla jednostek gospodarki uspołecznionej (art.384§1 k.c.). Wynika on również z obowiązku szczególnej staranności w przypadku gospodarowania mieniem społecznym.

Jakie postanowienia mogą być zawarte w umowie? Powyżej, przy omawianiu postępowania ofertowego, użyto określenia "istotne postanowienia" umowy. Co to są za postanowienia? Postanowienia istotne to te, których brak wywoływałby brak wynikających z nich skutków. Zdanie to stanie się bardziej zrozumiałe, gdy powiemy co to są tzw. "postanowienia nieistotne". Są to mianowicie takie postanowienia, których skutki i tak wynikają z przepisów prawa odnoszących się do określonego typu umowy - brak więc takich postanowień w umowie nie powoduje braku wywoływanych przez nie skutków. Wśród postanowień istotnych wyróżnić można tzw. postanowienia przedmiotowo istotne - essentialia negotii. W historii obrotu gospodarczego wykształcił się mianowicie szereg typów powszechnie zawieranych umów - co znalazło odzwierciedlenie w ustawodawstwie w ten sposób, że w przepisach prawa cywilnego niektóre typy umów ujęte zostają w przpisach szczególnych i określane są powszechnie mianem "umów nazwanych". Umowami takimi są np. umowa sprzedaży, umowa najmu czy umowa o dzieło. Essentialia negotii są takimi postanowieniami, których obecność w umowie pozwala na zakwalifikowanie jej jako określonej umowy nazwanej. Postanowieniami takimi w danej umowie są np. przeniesienie własności rzeczy, jej wydanie oraz zapłata ceny, pozwalające na zakwalifikowanie takiej umowy jako umowy sprzedaży. W umowach można zawierać w zasadzie postanowienia o dowolnej treści, kształtując tym samym treść stosunku zobowiązaniowego dowolnie, zależnie od wspólnego zamiaru stron. Jedynym zastrzeżeniem jest to, że z mocy art.58 k.c. nieważnymi są te postanowienia, które są sprzeczne z prawem lub które mają na celu obejście prawa a także takie, które są sprzeczne z zasadami współżycia społecznego. Wspomniano powyżej o umowach nazwanych. W niektórych jednak umowach spotkać można taki układ postanowień, który nie pozwala zakwalifikować umowy jako żadnej ze znanych umów nazwanych. Umowy takie noszą w nauce prawa miano umów nienazwanych. Jest tak nawet w przypadkach, w których częstość występowania podobnych do siebie umów nienazwanych doprowadził w praktyce do powstania nazwy odpowiedniego rodzaju umowy nienazwanej.

Przykładem takiego nazwanego rodzaju umowy nienazwanej jest umowa know-how, która przewiduje odpłatne udostępnienie wiedzy technicznej o charakterze poufnym - wiedzy, która nie jest przedmiotem praw o charakterze bezwzględnym (np. przedmiotem praw będących skutkiem udzielenia patentu). Dokonując oceny prawnej jakiejkolwiek zawartej umowy należy dążyć do stwierdzenia, czy postanowienia ilustrujące główny zamiar i cel stron tej umowy stanowią essentialia negotii jakiejś umowy nazwanej. Jeśli tak jest to umowę taką można zakwalifikować jako nazwaną i w ocenie jej skutków prawnych posługiwać się odpowiednimi dla właściwej umowy nazwanej przepisami szczególnymi. Kształtowanie zawieranej umowy jako umowy nazwanej pozwala na skorzystanie np. z istniejących wzorów umów (wydawanych przez naczelne organy administracji państwowej i obligatoryjnych dla jednostek gospodarki uspołecznionej). Zakwalifikowanie już zawartej umowy jako nazwanej ma tę korzyść praktyczną, że pozwala na oparcie oceny prawnej o przepisy, które szczegółowo dla danej umowy określają wiele uprawnień i obowiązków stron, a także skutki niewykonania lub nienależytego wykonania wynikających z niej zobowiązań. Nie jest to jednak zawsze możliwe, zwłaszcza w przypadku umów zawieranych w obszarze nowych rodzajów działalności gospodarczej (np. w informatyce) z uwagi na dużą bezwładność przepisów prawa w porównaniu do zmian w charakterze zjawisk zachodzących w takiej działalności.

Parę uwag należy poświęcić niezmiernie istotnej kwestii odpowiedzialności za szkody wynikłe z niewykonania lub nienależytego wykonania zobowiązania umownego. Zasady ogólne są takie, że zgodnie z art.471 k.c. odpowiada za nie ten, którego świadczenie było przedmiotem zobowiązania - chyba, że niewykonanie lub nienależyte wykonanie takiego zobowiązania było następstwem okoliczności, za które nie ponosi on odpowiedzialności. Oznacza to, że ten, któremu miano świadczyć i który poniósł szkodę powinien ją wykazać oraz dowieść, że była ona skutkiem niewykonania lub nienależytego wykonania zobowiązania. Ten natomiast, który miał świadczyć, może uwolnić się od odpowiedzialności jedynie przez udowodnienie, że niewykonanie lub nienależyte wykonanie zobowiązania było następstwem okoliczności, za które nie ponosi on odpowiedzialności. Te zasady ogólne zostają zastąpione przepisami szczególnymi, jeśli mamy do czynienia z umowami nazwanymi. Dzięki temu właśnie - jak

wspomniano wyżej - kształtowanie lub kwalifikowanie umowy jako nazwanej stwarza w kwestii odpowiedzialności sytuację obu stron bardziej jednoznaczną.

Omawiając umowy zawierane w informatyce ujmiemy je w następującym porządku. W pierwszej kolejności omówimy umowy zawierane w związku z tworzeniem i rozpowszechnianiem systemów informatycznych. Będą to umowy zawierane w obrocie oprogramowaniem oraz umowy o dostawę sprzętu komputerowego. Dalej omówione zostaną umowy związane z eksploatacją systemów informatycznych, tzn. umowy, w których świadczone są rozmaite tzw, usługi informatyczne.

2. Umowy zawierane w obrocie oprogramowaniem.

W swej interesującej monografii [1] p.t. "Umowy w zakresie informatyki i ochrona programów komputerowych" B.Czachórska dzieli umowy dotyczące oprogramowania na umowy o wytworzenie oprogramowania oraz umowy o udostępnienie oprogramowania gotowego. Autorka zauważa, że w kraju - jak dotychczas - przeważają sytuacje, w których zawierane zostają umowy o wytworzenie oprogramowania, głównie oprogramowania zindywidualizowanego i tym właśnie umowom poświęca głównie swoje rozważania. Biorąc pod uwagę znaczny wzrost liczby sytuacji, w których przedmiotem umowy jest udostępnienie oprogramowania gotowego, w niniejszym wykładzie poświęcimy więcej uwagi takim właśnie umowom.

A)Umowy o udostępnienie oprogramowania. Obrót oprogramowaniem systemów informatycznych stwarza najczęściej, co wynika z natury jego przedmiotu, wiele rozmaitych wątpliwości o charakterze prawnym. Przyczyną takiej sytuacji jest fakt, że oprogramowanie - będąc przedmiotem obrotu w postaci rozmaitych nośników materialnych (nośniki magnetyczne z zakodowanymi programami, zeszyty z dokumentacją) - jest w istocie swej wytworem intelektualnym, a więc tzw. niematerialnym dobrem prawnym. Nie budzi na ogół wątpliwości fakt, że inna jest treść umowy sprzedaży książki czy kasety z nagraniem utworu muzycznego, a inna - umowy o sprzedaż autorskich praw majątkowych mających za przedmiot utwory, które są w nich utrwalone. Inna jest treść umowy o sprzedaż urządzenia technicznego, a inna -

umowy licencyjnej dotyczącej opatentowanego rozwiązania, w oparciu o które urządzenie to jest skonstruowane. Jak sprawa ta przedstawia się w sytuacjach, w których zawierana umowa dotyczy oprogramowania? Postaramy się udzielić odpowiedzi ogólnej, ale możliwie wyjaśniającej istotę zagadnienia. Przed przystąpieniem do analizy postawionego zagadnienia poczynimy kilka ustaleń terminologicznych. Mówiąc o oprogramowaniu będziemy mieli na myśli następujące jego elementy:

a) programy dla komputera,

b) materiały pomocnicze.

Przez te ostatnie rozumieć będziemy wszelkiego rodzaju dokumentację programów, wyrażoną pismem czy rysunkami i obejmującą rozmaite podręczniki operatora, programisty czy użytkownika. Używając terminu "oprogramowanie" będziemy przy tym mieli na myśli dobro o charakterze niematerialnym, wytwór intelektualny, a więc rozwiązania składające się na treść programów oraz utwory składające się na treść materiałów pomocniczych. Nośniki, w których oprogramowanie jest utrwalone i za pośrednictwem których staje się ono przedmiotem obrotu, nazywać będziemy "materialnymi substratami oprogramowania".

Przechodząc do kwestii związanych z obrotem oprogramowaniem należy na wstępie wyjaśnić jaka jest prawna natura oprogramowania jako dobra niematerialnego, t.j. jakie prawa mające oprogramowanie za przedmiot mogą komukolwiek przysługiwać. Występująca w prawie cywilnym konstrukcja tzw. praw podmiotowych dzieli je na dwa rodzaje - prawa bezwzględne i prawa względne (prawo podmiotowe- kompleks uprawnień przysługujących danemu podmiotowi ze względu na jakiś przedmiot tego prawa w kontekście obowiązku innych podmiotów co do respektowania tych uprawnień). Prawa bezwzględne wynikają z przepisów ustawy i skuteczne są względem jakiegokolwiek innego podmiotu, który tym samym obowiązany jest do ich respektowania. Prawa względne skuteczne są jedynie względem niektórych innych podmiotów i wynikają najczęściej z umownego zobowiązania zaciągniętego w stosunku do podmiotu, który w ten właśnie sposób je uzyskał. Przykładem prawa bezwzględnego jest na przykład prawo własności (art.140 k.c.), które odnosi się do rzeczy, czyli przedmiotów materialnych. Zgodnie z obowiązującym w Polsce prawem jedynymi podmiotowymi prawami bezwzględnymi, mającymi za przedmiot dobra

niematerialne są prawa wynikające z ustawy z dnia 19 października 1972 r. o wynalazczości (Dz.U. 1984 r., nr.33, poz.133), a więc prawo wyniakjące z patentu lub zarejestrowanego wzoru użytkowego oraz prawa autorskie wynikające z ustawy z dnia 10 lipca 1952 r. (Dz.U. nr.34, poz.234). Art.2 p.5 ustawy o wynalazczości wyklucza możliwości patentowania pragramów dla maszyn cyfrowych lub rejestrowania ich jako wzorów użytkowych. Oprogramowanie mogłoby więc ewentualnie uznane jedynie za przedmiot praw autorskich, którym jest "każdy utwór literacki, naukowy i artystyczny ustalony w jakiejkolwiek postaci" (art.1§1 ustawy o prawie autorskim) jeśliby przyjąć, że jest ono utworem tego rodzaju lub takie utwory na nie się składają. W niektórych przypadkach jako utwory naukowe możnaby kwalifikować materiały pomocnicze, a więc jedynie niektóre elementy oprogramowania - i to nie te, na których ochronie najbardziej jego wytwórcom zależy. Za utwory takie nie można jednak uznać samych programów. Teza taka może być najogólniej uzasadniona analogią pomiędzy zapisem programu, a zapisem ustalenia naukowego (formułą, zapisem postępowania w metodzie pomiaru i t.p.). W przypadku utworu naukowego ochronie podlega sam utwór, natomiast zamieszczone w nim zapisy dokonanych ustaleń nie podlegają ochronie prawem autorskim i mogą być swobodnie wykorzystywane. Tezę taką uzasadniał szczegółowo J.Waluszewski [2]. W konsekwencji więc nie można uznać za przedmiot praw autorskich tego elementu oprogramowania, który jest w nim najbardziej istotny i z którego wytworzeniem związana jest większość nakładów ponoszonych w toku opracowywania systemów informatycznych. Ochronę interesu majątkowego osób ponoszących takie nakłady można zapewnić więc jedynie poprzez odpowiednie kształtowanie zawieranych w obrocie oprogramowaniem umów, a więc poprzez tworzenie tą drogą dogodnych podmiotowych praw względnych.

Na początku niniejszej części wykładu zwrócono uwagę na konieczność odróżnienia na przykład umowy sprzedaży kasety z nagraniem utworu muzycznego od umowy przenoszącej majątkowe prawa autorskie do utworu muzycznego utrwalonego na takiej kasecie. Nabywcy własności kasety z takim nagraniem nie przysługuje pełna swoboda wykorzystania przedmiotu swojej własności. Właściciel jakiejkolwiek rzeczy może bowiem korzystać z niej i rozporządzać nią jedynie "w granicach określonych przez ustawy", zgodnie z art.140 k.

c.. W przypadku wspomnianej kasety z nagraniem ustawą ograniczającą swobodę jej właściciela jest mieędzy innymi ustawa o prawie autorskim. Nakłada ona na właściciela szereg ograniczeń. Nie może więc on na przykład skopiować nagrania dla kogoś innego czy odtwarzać nagrania publicznie pobierając za to opłatę (chyba, że zawarł z podmiotem praw autorskich odpowiednią umowę).

Używane w praktyce określenie "sprzedaż oprogramowania" jest prawnie niewłaściwe. Sprzedaż bowiem dotyczyć może wyłącznie rzeczy (art.535 k.c.) lub praw (art.555 k.c.). O sprzedaży praw być mowy w tym wypadku nie może ponieważ oprogramowanie - jak wspomniano - nie jest jako całość przedmiotem praw bezwzględnych. Sprzedaż oprogramowania mogłaby więc jedynie oznaczać sprzedaż rzeczy, a więc sprzedaż materialnych substratów oprogramowania. Nabywca takich substratów miałby jednak w takim przypadku pełną swobodę korzystania z programów i rozporządzania nimi. Mógłby on je kopiować, ulepszać, rozpowszechniać oraz wykonywać wiele innych działań, które najczęściej są w sprzeczności z interesem osoby, która poniosła nakłady na wytworzenie oprogramowania. W odróżnieniu bowiem od utworów będących przedmiotem praw autorskich programów przedmiotem takich praw nie są. Nie ma więc w tym przypadku ograniczeń, którym podlega na przykład nabywca wspomnianej ksety z nagraniem utworu muzycznego czy nabywca książki lub obrazu.

Osoba dysponująca oprogramowaniem i chcąca je rozpowszechniać powinna więc w celu ochrony swojego interesu majątkowego kształtować umowę o udostępnienie oprogramowania w taki sposób, aby zobowiązać kontrahenta do wykorzystywania udostępnionego oprogramowania w sposób ochronę tego interesu zapewniający. Kontrahent ten będzie wówczas odpowiedzialny za niewykonanie lub nienależyte wykonanie przyjętych zobowiązań zgodnie z art.471 k.c.. Zawierana umowa mogłaby być kształtowana w sposób podobny, jak kształtowane są tzw. umowy know-how [3], w ramach których udostępniana jest wiedza techniczna o charakterze poufnym, nie będąca najczęściej przedmiotem praw bezwzględnych.

Zawierając umowę o udostępnienie oprogramowania możnaby więc umieszczać w jej treści postanowienia następujących rodzajów:

a) Postanowienia dotyczące przeniesienia własności materialnych substratów oprogramowania lub wytworzenia takich substratów przez powielenie programów na nośnikach odbiorcy. Postanowienia tego rodzaju pozwalają na stwierdzenie istnienia w umowie o udostępnienie oprogramowania elementów sprzedaży (która dotyczy materialnych substratów oprogramowania) lub elemetów umowy o dzieło (w ramach której takie substraty zostają wykonane). W obu przypadkach daje to podstawę dla odpowiedzialności udostępniającego z tytułu rękojmi za wady oprogramowania (art.556 k.c. lub art.637 i 638 k.c.). Wady fizyczne oprogramowania mogą być

- fizycznymi wadami nośników,
- wadami polegającymi na tym, że programy nie funkcjonują zgodnie z ich opisem. Wady takie można uznać za fizyczne dlatego, że można je uznać za niedoskonałość substratów materialnych. Nie mają one bowiem w takim przypadku właściwości, o której odbiorca był zapewniony (poprzez treść dostarczanych materiałów pomocniczych)(art.556§1 k.c.). Z postanowień tych może ponadto wynikać, że odbiorcy wydane zostaną jedynie niektóre elementy materiałów pomocniczych, wystarczające dla celów eksploatacji oprogramowania (np. bez dokumentacji źródłowej). Wydanie jedynie części materiałów pomocniczych stanowi często dodatkowe zabezpieczenie tajemnicy rozwiązań zawartych w oprogramowaniu - poprzez utrudnienie zapoznania się z nimi.

b)Postanowienia dotyczące zakresu i sposobu wykorzystania udostępnionego oprogramowania przez odbiorcę. Postanowienia tego rodzaju mogą określać na przykład, czy odbiorca uprawniony będzie wyłącznie do wykorzystywania oprogramowania dla własnych potrzeb, a jeśli nie, to określać mogą

- krąg pdodmiotów, które korzystać mogą z udostępnionego oprogramowania,
- uprawnienia tych podmiotów do usługowej, odpłatnej eksploatacji oprogramowania na rzecz innych osób,
- zasady odpłatności i wysokość opłat za wykorzystanie oprogramowania w celu odpłatnego świadczenia usług informatycznych.

c)Postanowienia dotyczące rozporządzania przez odbiorcę udostępnionym oprogramowaniem. W ścisłym związku z tymi postanowienia-

mi są postanowienia nakładające obowiązek zachowania rozwiązań zawartych w oprogramowaniu w tajemnicy. Postanowienia dotyczące rozporządzania oprogramowaniem określają uprawnienia odbiorcy odnośnie do dalszego udostępnienia oprogramowania przez odbiorcę osobom trzecim, a także do dokonywania w nim zmian i ulepszeń.

d)Postanowienia dotyczące uprawnień i obowiązków udostępniającego w odniesieniu do oprogramowania po zawarciu umowy. Mogą one określać na przykład, czy po udostępnieniu oprogramowania udostępniający nadal zachowuje względem oprogramowania swoje dotychczasowe uprawnienia, czy też uprawnienia te przechodzą (w całości lub części) na odbiorcę. Do postanowień tego rodzaju można również zaliczyć takie, które określają obowiązki udostępniającego w zakresie serwisu oprogramowania oraz obowiązki w zakresie udostępniania kolejnych, ulepszonych jego wersji.

Powyższe uwagi pomyślane są jako wskazówka co do tego, na jakie elementy należy zwracać uwagę w przypadku zawierania umów o udostępnienie oprogramowania. Ograniczenie treści tych umów wyłącznie do postanowień takich jak wymienione w p. a) czyni z nich umowy o sprzedaż materialnych substratów oprogramowania i nie nakłada na ich nabywcę żadnych ograniczeń w zakresie korzystania z oprogramowania i rozporządzania nim. Konkretny zestaw postanowień takiej umowy powinien każdorazowo wynikać z jej celu w danej sytuacji.

Jak już wspomniano wyżej, występowanie w umowach o udostępnienie oprogramowania elementów sprzedaży (czy umowy o dzieło) stanowi podstawę odpowiedzialności udostępniającego z tytułu rękojmi za wady oprogramowania. Odpowiedzialności tej nie można wyłączyć ani ograniczyć w stosunkach między jednostkami gospodarki uspołecznionej ani w stosunkach pomiędzy takimi jednostkami a innymi osobami - jeżeli nie pozwala na to przepis szczególny (art.558§1 k. c.). W odniesieniu do umów dotyczących oprogramowania żadnych przepisów szczególnych nie ma - ograniczać ani wyłączać uprawnień z tytułu rękojmi drogą postanowień umownych nie wolno. Jak wspomina w cytowanej pracy B.Czachórska odpowiedzialność ta w odniesieniu do wad oprogramowania nie daje najczęściej satysfakcjonującej ochrony jego odbiorcom, w szczególności ze względu na zbyt krótki o-

kres przewidziany w kodeksie cywilnym dla wykonywania uprawnień z tego tytułu (1 rok). Należy więc dążyć do ukształtowania odpowiedzialności udostępniającego za wady oprogramowania w drodze postanowień w tym zakresie do umowy. Postanowienia takie - zwane zwykle klauzulami gwarancyjnymi - określają rodzaje wad, za które odpowiadać będzie udostępniający w razie ich wykrycia oraz określają szczegółowo obowiązki udostępniającego i uprawnienia odbiorcy wynikające z tej odpowiedzialności. Postanowienia takie nie mogą oczywiście być mniej korzystne dla odbiorcy niż przepisy o rękojmi z uwagi na kodeksowy zakaz jej ograniczania - w obrocie uspołecznionym. Przepisy te, wyrażone w art.560 k.c. dają odbiorcy oprogramowania możność

- odstąpienia od umowy,
- żądania obniżenia ceny

chyba, że udostępniający oświadczy gotowość natychmiastowej wymiany materialnych substratów oprogramowania na wolne od wad lub niezwłocznie wady usunie (jeśli nie są to wady fizyczne nośników, oznacza to poprawienie programów takie, aby zapewnić zgodność ich funkcji z funkcjami opisanymi w dokumentacji). Odbiorca może ponadto, na mocy art.561§2 k.c., żądać od udostępniającego - jeśli to on był wytwórcą oprogramowania - usunięcia wad w wyznaczonym terminie z zagrożeniem, że w razie niedotrzymania odstąpi od umowy. Udostępniający może jednak odmówić, jeśli usunięcie wad byłoby związane z nadmiernymi kosztami.

Odbiorca oprogramowania z wadami ma ponadto - niezależnie od uprawnień przysługujących mu z tytułu rękojmi czy klauzul gwarancyjnych - możność uniważnienia zawartej uprzednio umowy, w ramach której oprogramowanie takie zostało mu udostępnione. Jeśli bowiem programy nie realizują funkcji, które zapowiedziane są w materiałach pomocniczych to odbiorca może powołać się na przepisy o błędzie w oświadczeniu woli (art.84 k.c.) i oświadczyć, że miał miejsce błąd istotny, dotyczący treści umowy, wywołany przez drugą stronę. Był bowiem przekonany, że udostępnione mu będzie oprogramowanie zapewniające realizację określonych funkcji, co nie odpowiada stanowi faktycznemu. Może on wówczas uchylić się od skutków swojego oświadczenia woli, a więc uznać umowę za niezawartą. Ta możliwość wzmacnia jego pozycję.

B)Umowy o wytworzenie oprogramowania. Trudności w ocenie skutków prawnych umów o wytworzenie oprogramowania leżą w nieco innym obszarze w porównaniu z umowami omawianymi poprzednio. W wielokrotnie już powoływanej monografii B.Czachórska kwalifikuje umowy tego rodzaju jako umowy o dzieło (regulowane przepisami art. 627 i następnymi k.c.). Mniej wątpliwości mianowicie budzi kwestia uprawnień względem oprogramowania. Jak twierdzi autorka, w związku z zapłatą przez zamawiającego wynagrodzenia za wykonanie dzieła, całość uprawnień do rezultatu zamówienia przechodzi na zamawiającego. (Wyjątkiem w takiej sytuacji są uprawnienia wykonawcy pracy naukowo-badawczej z mocy przepisów szczególnych dotyczących umów tego rodzaju. Może on udostępniać odpłatnie wyniki takiej pracy również innym osobom.)

Wątpliwości dotyczące kwalifikacji prawnej umów o wytworzenie oprogramowania, a więc dotyczące skutków prawnych zawieranej umowy leżą natomiast gdzie indziej. Zgodnie ze stanowiskiem nauki prawa [4] oraz orzecznictwa, dziełem jest "indywidualnie oznaczony, obiektywnie możliwy i subiektywnie pewny do uzyskania rezultat o charakterze materialnym lub niematerialnym". W przypadku, w którym zamówionym dziełem jest oprogramowanie, można mieć poważne wątpliwości czy przdmiot zamówienia jest w dostatecznym stopniu indywidualnie oznaczony (t.j. czy funkcje oprogramowania są dostatecznie sprecyzowane), nie zawsze też można być w pełni przekonanym o obiektywnej możliwości uzyskania rezultatu odpowiadającego indywidualnemu oznaczeniu (t.j. nie zawsze wiadomo, czy będzie możliwe uzyskanie rozwiązań zgodnych z oznaczeniem funkcji zamawianego oprogramowania). Można więc postulować realizację umowy w dwóch etapach. W pierwszym - przedmiotem umowy mogłoby być przeprowadzenie odpowiednich badań w celu zarówno indywidualnego oznaczenia oprogramowania (określenia jego funkcji) jak i w celu określenia stopnia możliwości jego opracowania. W drugim - świadczenie przyjmującego zamówienie mogłoby być już bez wątpliwości kwalifikowane jako wykonanie dzieła, za który to rezultat odpowiadałby on już bez wątpliwości zgodnie z przepisami dotyczącymi umowy o dzieło. W szczególności odpowiedzialność ta obejmowałaby również wady dzieła i określona byłaby przepisami jak przy sprze-

daży (na mocy art.637 i 638 k.c.). I w tym przypadku mają więc aktualność uwagi na temat instytucji rękojmi zamieszczone w akapicie A).

Należy jeszcze zaznaczyć dla uniknięcia nieporozumień, że co innego należy mieć na myśli mówiąc o elementach umowy o dzieło występujących w umowie o udostępnienie oprogramowania gotowego, a o umowie o wytworzenie oprogramowania jako o umowie o dzieło. W pierwszym przypadku dziełem są materialne substraty oprogramowania. Oprogramowanie już istnieje, a wytworzenie jego substratów polega na wykonaniu kopii substratów już istniejących. Nie ma więc wątpliwości ani co do indywidualnego oznaczenia rezultatu, ani co do pewności jego uzyskania. Dzieło ma w tym przypadku charakter czysto materialny. W drugim przypadku dziełem jest oprogramowanie jako całość. Dzieło to ma w tym przypadku charakter niematerialny - jest wytworem intelektualnym.

3. Umowy o dostawę sprzętu komputerowego.

Umowa o dostawę sprzętu komputerowego jest umową dotyczącą rzeczy i w związku z tym nie stwarza większych trudności w zakresie jej kwalifikacji. Jest to umowa sprzedaży lub dostawy regulowana przepisami kodeksu cywilnego, odnośnie do sprzedaży - art.535 i następne k.c., odnośnie do dostawy - art.605 i następne k.c.. W przypadku, w którym sprzęt dostarczany jest przez jednostki gospodarki uspołecznionej, zastosowanie mają przede wszystkim przepisy tzw. ogólnych warunków umów sprzedaży i umów dostawy wydane przez Radę Ministrów (Uchwała nr.207 RM z dnia 27 września 1982 r. Mon.Pol. nr.26, poz.235). Przepisy tego rodzaju wydawane zostają na podstawie art.2 i art.384 k.c.. Jedynie wówczas, gdy jakieś kwestie nie są uregulowane tymi przepisami, zastosowanie mają przepisy kodeksu cywilnego. Przepisy kodeksu cywilnego mają więc w tych przypadkach charakter - jak się to określa - subsydiarny.

Jest jednak szereg kwestii szczególnych, na które należy zwracać uwagę przy zwieraniu umów dotyczących sprzętu komputerowego. Są to kwestie:

- uzależnienia odpowiedzialności sprzedawcy, dostawcy czy pro-

ducenta z tytułu rękojmi za wady sprzętu czy z tytułu klauzul gwarancyjnych od zapewnienia przez kupującego czy odbiorcę właściwych technicznych warunków eksploatacji oraz właściwej obsługi sprzętu,

- obowiązków sprzedawcy, dostawcy czy producenta w zakresie serwisu.

Jeżeli z dostawą sprzętu komputerowego wiąże się udostępnienie oprogramowania to należy mieć na uwadze wszelkie problemy z tym związane. Jeżeli więc umowa nie zawiera szczególnych postanowień w odniesieniu do oprogramowania to - zgodnie z tym, co powiedziano w p.2 na ten temat - udostępnienie oprogramowania jest jedynie sprzedażą jego materialnych substratów. Umowa taka nie stwarza więc wówczas żadnych praw względnych mających za przedmiot oprogramowanie jako dobro niematerialne i nie chroni interesu jego wytwórcy.

4. Umowy związane z eksploatacją systemów informatycznych.

W wielokrotnie już powoływanej monografii B.Czachórska wymienia następjące rodzaje usług związanych z eksploatacją systemów informatycznych:

a) przygotowanie maszynowych nośników informacji,
b) przetwarzanie danych,
c) czasowe udostępnienie sprzętu komputerowego (w tym udostępnienie w ramach systemów abonenckich),
d) przygotowanie i utrzymywanie baz danych oraz banków danych,
e) szkolenie w zakresie obsługi systemów.

Autorka dochodzi do wniosku, że zawierane w celu świadczenia tych usług umowy mogą być kształtowane czy kwalifikowane według schematów umów nazwanych, co oczywiście ułatwia ocenę prawną ich skutków. Według wspomnianej autorki usługi wymienione w pp. a), b) i d) można kształtować lub kwalifikować jako usługi wykonywane w ramach umowy o dzieło (regulowanej art.627 i następnymi k.c.), a usługi wymienione w p.d) - jako świadczenie w ramach umowy najmu (regulowanej art.659 i następnymi k.c.). Usługi wymienione w p.e) należy oceniać w oparciu o przepisy dotyczące umowy zlecenia (regulowanej art.734 i następnymi k.c.).

Należy podnieść dwa problemy szczegółowe związane z tymi umowami. Pierwszy z nich to problem odpowiedzialności osoby świadczącej usługi w zakresie eksploatacji systemów informatycznych za szkody wynikłe z nieprawidłowego przetwarzania, w szczególności wynikłe z nieterminowego lub błędnego wykonania obliczeń. Drugi - to problem odpowiedzialności tej osoby za szkody wynikłe z faktu utraty danych lub ich uszkodzenia, a także ich ujawnienia, jeśli miały pozostać przedmiotem tajemnicy. Osoba świadcząca usługi odpowiedzialna jest za wszelkie szkody wynikłe bądź z nieprawidłowego przetwarzania bądź z faktu utraty, uszkodzenia lub ujawnienia danych. Jest to odpowiedzialność za szkody wynikłe z niewykonania lub nienależytego wykonania zobowiązania i określona jest art.471 k.c.. Poważne wątpliwości mogą budzić jednak kwestie związane z rodzajem okoliczności, które uwolnić mogą osobę świadczącą te usługi od odpowiedzialności za wynikłe szkody. W pracy czwórki autorów, mającej charakter poradnika dla osób zawierających umowy w zakresie informatyki [5)], postuluje się regulowanie kwestii tej odpowiedzialności drogą odpowiednich postanowień umownych. Postanowienia te miałyby dotyczyć:

- obowiązków osoby świadczącej usługi w zakresie sprawdzenia niezawodności sprzętu, stwierdzenia poprawności oprogramowania, zapewnienia właściwej obsługi procesu przetwarzania czy sposobu ewidencjonowania przebiegu tego procesu dla umożliwienia późniejszego wykrycia przyczyn ewentualnej nieprawidłowości,
- uprawnień odbiorcy usług w zakresie kontroli procesu przetwarzania,
- obowiązków osoby świadczącej usługi w przypadku zaistnienia okoliczności szczególnych, takich jak na przykład poważniejsza awaria sprzętu czy brak zasilania z zewnątrz.

Umowne uregulowanie tych kwestii może stanowić istotny element w przypadku ewentualnych sporów, pomagający w jaśniejszym określeniu sytuacji obu stron. Zakres i stopień szczegółowości tej regulacji zależny jest od woli stron i może wynikać z konkretnej sytuacji.

BIBLIOGRAFIA

1. B.Czachórska, Umowy w zakresie informatyki i ochrona programów komputerowych, Ossolineum 1980,
2. J.Waluszewski, Programy dla maszyn cyfrowych jako przedmiot prawa, Zeszyty Naukowe UJ, PWOWI, 1975, zeszyt 5,
3. B.Gawlik, Umowa know-how, Zeszyty Naukowe UJ, PWOWI, 1974, zeszyt 3,
4. S.Wójcik, Pojęcie umowy o dzieło, Studia cywilistyczne, t.IV, 1963 r.
5. B.Czachórska, J.Irlik, R.Pessel, J.Waluszewski, Umowy w zakresie informatyki, Wyd. ORGMASZ (w przygotowaniu).

Jesienna Szkoła PTI
Mrągowo, listopad 1985

GRAFIKA KOMPUTEROWA

dr Michał Jankowski
Instytut Informatyki
Uniwersytetu Warszawskiego
PKiN p. 850, tel. 268258
00-901 Warszawa

1. Wstęp.

W książce [GI] znajdujemy następującą definicję grafiki komputerowej: "Termin ten oznacza generowanie, przedstawianie, przekształcanie lub analizowanie obiektów graficznych przy pomocy komputera, a także określanie związków między obiektami graficznymi i innymi nie graficznymi rodzajami informacji".

Przez prawie dwadzieścia lat, od zbudowania w 1950 roku pierwszego monitora graficznego, grafika komputerowa była dziedziną uprawianą tylko przez wąską grupę profesjonalistów. Przyczyną był głównie bardzo wysoki koszt urządzeń graficznych. Gwałtowny rozwój elektroniki radykalnie zmienił (i nadal szybko zmienia) ten stan rzeczy. Dzięki mikrokomputerom grafika komputerowa stała się łatwo dostępna - trafiła do fabryk, szkół, biur i prywatnych mieszkań. W krajach rozwiniętych koszt niezłej instalacji nie przekracza przeciętnej pensji, a kłopoty z kupnem wynikają z bogactwa wyboru.

Coraz niższe ceny sprzętu oraz rozwój oprogramowania bardzo poszerzyły zakres zastosowań grafiki komputerowej. Zrewolucjonizowała ona wiele dziedzin - od projektowania samochodów do produkcji filmów rysunkowych i rozrywki (kto nie bawił się grami komputerowy-

mi ?).

Obecnie zamiast żmudnych prac projektowych przy biurku i rysownicy, wykonywania modelu i prób na nim możemy zaprojektować, obejrzeć i zbadać, a jeśli jednym z urządzeń wyjściowych komputera jest sterowana numerycznie obrabiarka, to nawet otrzymać gotowy produkt finalny - wszystko nie odchodząc od komputera.

Dla większości ludzi obraz jest znacznie bardziej komunikatywny niż napisy czy kolumny cyfr. Ten fakt oraz istniejące już możliwości łatwego wprowadzania obrazów, ich komputerowego generowania i przetwarzania powodują, że - jak pisze w artykule [VD] A. van Dam - "Interakcyjna grafika komputerowa szybko staje się standardowym środkiem komunikowania pomiędzy komputerami a ich użytkownikami".

2. Urządzenia graficzne.

Najczęściej oglądamy obrazy komputerowe na ekranie lub kartce papieru. Mogą być one zapisywane także techniką fotograficzną (mikrofilmy, slajdy) i video.

Urządzenia ekranowe to grafoskopy, monitory lub zwykłe odbiorniki TV, których zasadniczą częścią jest lampa katodowa (jednak coraz częściej można już spotkać płaskie ekrany na technologii ciekłych kryształów). Obraz na ekranie jest "malowany" przez strumień elektronów, który wzbudza świecenie luminoforu. W przypadku ekranów kolorowych są to trzy strumienie dla podstawowych kolorów: czerwonego, zielonego i niebieskiego. Intensywność świecenia luminoforu zależy od energii strumienia. Zanika ono po krótkim czasie i dlatego obraz musi być odnawiany co najmniej 30 razy na sekundę. Wymaga to pamiętania cyfrowej reprezentacji obrazu.

Urządzenia ekranowe mogą pracować w trybie wektorowym lub rastrowym. W trybie wektorowym strumień elektronów przesuwany w sposób ciągły pomiędzy dwoma dowolnymi punktami na ekranie rysuje odcinek -wektor, a obraz jest sumą wektorów. Główną wadą tego trybu jest brak możliwości wypełnienia kolorem (stopniem jasności) całych obszarów, otrzymujemy zawsze rysunek kreskowy, co na ogół pozwala przedstawić tylko szkieletową konstrukcję rysowanego obiektu. Jeśli liczba wektorów tworzących obraz jest bardzo duża, to wynikają kło-

poty z pamięcią do przechowywania informacji o obrazie i nadążaniem z jego odnawianiem.

W trybie rastrowym strumień elektronów (tak jak w odbiorniku TV) przebiega kolejne poziome linie wzbudzając z różną intensywnością (kolorem) świecenie poszczególnych najmniejszych elementów ekranu, zwanych potocznie pixlami. Rysunek odcinka jest teraz sumą pixli, przy małej rozdzielczości ekranu przedstawiany jest jako dość grube "schodki". W urządzeniach rastrowych łatwe jest wypełnianie wnętrza obszaru, a więc uzyskiwanie kolorowych plam potrzebnych do realistycznego obrazowania scen. Ponadto liczba miejsc pamięci i czas potrzebny do odnawiania nie zależy od złożoności malowanego obrazu.

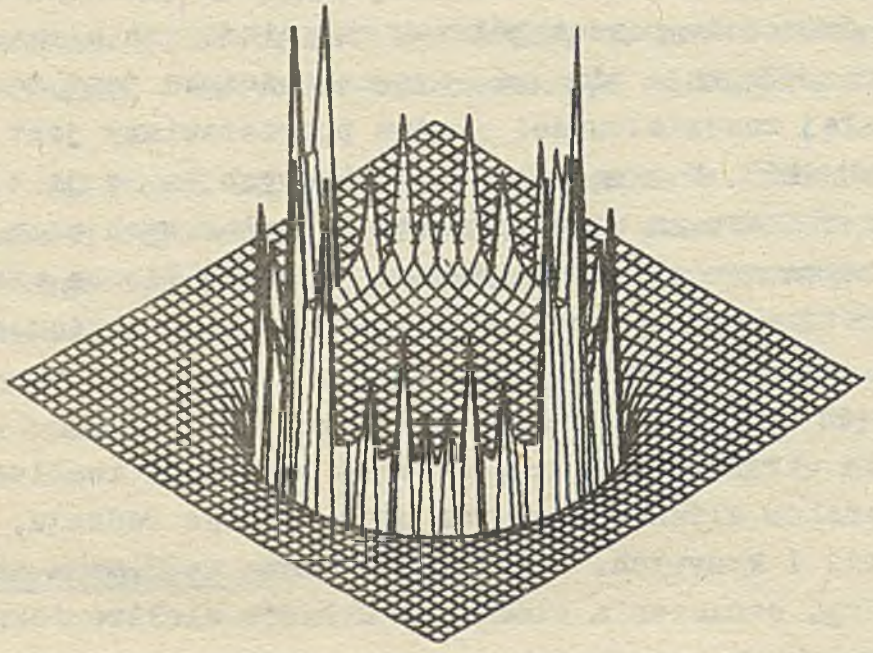
Urządzenia ekranowe mają najczęściej sprzętowo realizowane wyświetlanie znaków alfanumerycznych oraz różnego rodzaju, koloru i jasności linii i krzywych. Zależnie od zastosowań, menu znaków może zawierać np. oznaczenia elementów układów elektronicznych, nuty itp.

Jakość (i cena) monitorów jest bardzo różna. Można wybierać od zwykłych odbiorników TV czarno-białych lub o kilku kolorach i rozdzielczości rzędu kilkuset linii przy najtańszych mikrokomputerach osobistych do monitorów ekranowych o rozdzielczości kilku tysięcy linii i 24 bitach na informację o kolorze dla każdego elementu ekranu.

Jeszcze większa różnorodność występuje wśród urządzeń rysujących (drukujących) na papierze. Najprostsze kreślaki (ang. plotter) cyfrowo-analogowe rysują flamastrami na kartce papieru formatu zeszytowego, potwierdzeniem marnej jakości takich urządzeń jest na przykład rysunek 1. Najbardziej skomplikowane kreślaki malują w wielu kolorach rysunki wielkości kilkunastu metrów i to z dokładnością rzędu 10 μ. Niewiele mniejsza różnorodność występuje wśród drukarek: od mozajkowych do zaczynających się już coraz częściej pojawiać laserowych.

W interakcyjnej grafice komputerowej równie ważne jak wyjściowe są urządzenia umożliwiające wprowadzanie danych graficznych. Należą do nich pióra świetlne, manipulatory i "myszki" (ruszając nimi

zmieniamy położenie kursora na ekranie) oraz dyskretyzatory (ang. digitizer) pozwalające na przenoszenie obrazów ze zdjęć lub rysunków (w tym wykonywanych ręcznie na specjalnych podkładkach) na ekran. Zainteresowanych takimi urządzeniami odsyłam np. do artykułu [BO].



Rys. 1.

3. Zastosowania grafiki komputerowej.

Głównym zastosowaniem grafiki komputerowej jest projektowanie wspomagane komputerowo (ang. CAD - Computer Aided Design). Wyspecjalizowane systemy CAD (patrz [E1] i [KU]) są stosowane w przemyśle samochodowym, lotniczym i okrętowym, do projektowania architektonicznego i urbanistycznego, do projektowania układów elektronicznych (szczególnie VLSI), form przemysłowych i maszyn.

Grafika komputerowa umożliwia symulowanie lotów samolotami i naukę nawigacji. Metody grafiki komputerowej pozwalają na rysowanie map na podstawie zdjęć satelitarnych, są wykorzystywane w medycynie i biologii.

Bardzo ważnym zastosowaniem jest edukacja, w tym np. komputerowe obrazowanie zjawisk fizycznych, procesów chemicznych czy modeli matematycznych. Możliwość interakcyjnego generowania kolorowych obrazów i animacji znacznie uprościła przygotowywanie filmów rysun-

kowych i reklam telewizyjnych. Grafika komputerowa w połączeniu z techniką video pozwala na uzyskiwanie niezwykłych efektów w filmach science-fiction. Znajduje też coraz więcej zastosowań w pracach biurowych. Szalenie popularne są obecnie gry komputerowe, w których najbardziej atrakcyjna jest na ogół strona graficzna. Grafika komputerowa trafiła nawet do salonów kosmetycznych - klientki firmy Elizabeth Arden przed zrobieniem makijażu mogą obejrzeć jego projekt na monitorze.

4. Standardy i języki graficzne.

Od około dziesięciu lat są opracowywane i ciągle ulepszane dwa standardy graficzne: Core Graphics System i Graphical Kernel System ([ENI). CORE traktuje grafikę dwuwymiarową jako szczególny przypadek grafiki trójwymiarowej, podczas gdy GKS operuje prawie wyłącznie na obiektach dwuwymiarowych. Dalsze różnice dotyczą obsługi urządzeń wejścia/wyjścia - CORE kontroluje je indywidualnie a GKS grupuje w stacje. Inne są też sposoby wizualizacji.

Mimo że oba standardy doczekały się wielu implementacji, nie wydaje się by były powszechnie używane w grafice komputerowej. Zarówno GKS, jak i CORE zostały zaprojektowane w epoce urządzeń wektorowych i zawierają zbyt mało mechanizmów pozwalających na pełne wykorzystanie różnorodnych możliwości urządzeń rastrowych, które zdominowały rynek.

Jak dotąd nie ma języka programowania powszechnie używanego w grafice. Język taki powinien ułatwiać generowanie prymitywów graficznych (punkty, linie, krzywe, powierzchnie, znaki itp.), konstruowanie przy ich pomocy złożonych rysunków oraz opis i przekształcenia obrazów. Jednym z niewielu projektów języków graficznych, które doczekały się implementacji jest Pascal/Graph ([BA]). Znacznie częstszym rozwiązaniem jest tworzenie pakietów procedur graficznych, na ogół pisanych w Fortranie - przykładem może być szeroko rozpowszechniony system MOVIE ([CH]). W zastosowaniach edukacyjnych coraz większą popularność zdobywa język Logo.

5. Reprezentacja i modelowanie obiektów graficznych.

W najprostszym przypadku obiekty graficzne mogą być opisywane przy pomocy punktów, odcinków (par punktów-końców) i ścian (uporządkowanych list odcinków). Taki zbiór prymitywów wystarcza do reprezentowania nawet skomplikowanych trójwymiarowych tworów "szkieletowych". Wzbogacenie prymitywów o prostokąty i prostopadłościany (wraz z informacją o wypełniającym je kolorze) umożliwia budowanie scen zawierających "pełne" kolorowe figury i bryły. Przegląd różnych technik konstruowania i korzystania z baz danych graficznych można znaleźć w książce [BL].

W wielu zastosowaniach grafiki komputerowej, np. przy projektowaniu karoserii samochodowych czy przedmiotów codziennego użytku, potrzebne jest zarówno matematyczne modelowanie znanych kształtów (krzywych, powierzchni) jak też interakcyjne kreowanie nowych.

Krzywe przedstawiane są najczęściej w postaci parametrycznej

$$Q(t)=[x(t),y(t),z(t)] , \quad t\in[a,b] .$$

Jeśli znamy n+1 różnych punktów $P_i=P_i(t_i)$, przez które przechodzi dana krzywa (tzn. $P_i=Q(t_i)$ dla $i=0,1,\ldots,n$), to możemy interpolować Q krzywą wielomianową

$$W_n(t)=\sum_{i=0}^{n} P_i \prod_{j=0,j\neq i}^{n} \frac{t-t_j}{t_i-t_j} .$$

Zasadniczą wadą takiego rozwiązania są często bardzo duże oscylacje W_n między punktami P_i. Można się przed tym bronić, przybliżając Q nie jednym wielomianem W_n, na ogół wysokiego stopnia, ale na każdym podprzedziale $[t_i,t_{i+1}]$ określić wielomian niskiego stopnia spełniający warunki interpolacyjne i tak dobrać współczynniki tych różnych wielomianów, aby ich "sklejenie" dawało funkcję regularną na całym przedziale. W ten sposób konstruujemy funkcje sklejane (ang. spline). Pierwsze ich graficzne zastosowanie miało miejsce w 1963 r. w systemie CAD zaprojektowanym przez J.Fergusona dla firmy Boeing. Funkcje sklejane są obecnie powszechnym "narzędziem" w

grafice komputerowej. Przypomnijmy ich definicję

Definicja 1. Funkcję $S(t)$ nazywamy funkcją sklejaną rzędu k, z węzłami $t_0, t_1 \ldots t_n$, jeśli

- w każdym podprzedziale $[t_i,t_{i+1}]$ jest wielomianem stopnia $\leq k-1$,
- funkcja S i jej pochodne rzędu $1,2,\ldots,k-2$ są ciągłe na całym przedziale $[t_0,t_n]$.

W praktyce najczęściej używa się kubicznych funkcji sklejanych ($k=4$), w każdym podprzedziale $[t_i,t_{i+1}]$ funkcja taka jest wielomianem, który możemy zapisać w postaci

$$a_i+b_i(t-t_i)+c_i(t-t_i)^2+d_i(t-t_i)^3 .$$

Warunki interpolacyjne $S(t_i)=a_i=P_i$ i zależności wynikające z ciągłości funkcji S i jej pochodnych S' i S'' prowadzą (por.[JJ]) do układu równań liniowych o macierzy trójdiagonalnej, z którego kosztem $O(n)$ działań wyznaczamy współczynniki funkcji sklejanej.

Często nie chcemy przybliżać krzywej, ale projektować jej kształt i mieć możliwość łatwego wprowadzania zmian. Takie podejście zostało zapoczątkowane przez P.Bézier w systemie UNISURF używanym od 1972 r. przez firmę Renault do projektowania karoserii samochodów. W najprostszym przypadku tzw. krzywa Bézier jest postaci

$$Q(t)=(1-t)^3P_0+3t(1-t)^2P_1+3t^2(1-t)P_2+t^3P_3 , \quad t\in[0,1] ,$$

określające ją punkty P_0,P_1,P_2 i P_3 tworzą tzw. łamaną Bézier. Łatwo sprawdzić, że

$$Q(0)=P_0 , \quad Q(1)=P_3 , \quad Q'(0)=3(P_1-P_0) , \quad Q'(1)=3(P_3-P_2) ,$$

a stąd widać jaki jest wpływ punktów P_i na kształt krzywej. Jeśli zauważymy, że

$$Q(t)=\sum_{i=0}^{3} P_i \binom{3}{i} t^i(1-t)^{3-i}=\sum_{i=0}^{3} P_i\, B_{i,3}(t)$$

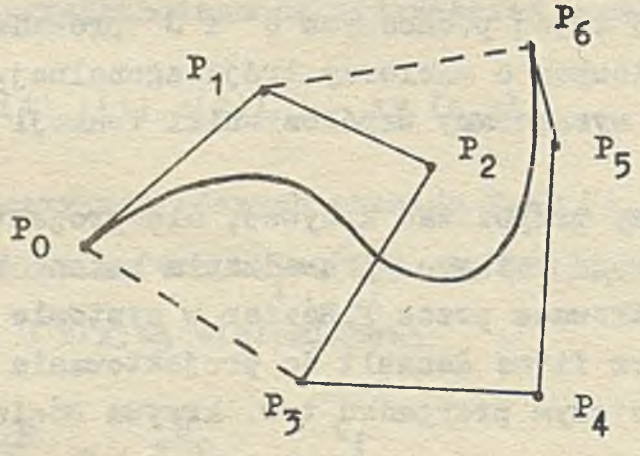
gdzie $B_{i,3}$ są szczególnym przypadkiem wielomianów Bernsteina $B_{i,n}(t) = \binom{n}{i} t^i (1-t)^{n-i}$, to naturalne jest uogólnienie krzywych Bézier dla większej liczby punktów P_i

$$Q(t) = \sum_{i=0}^{n} P_i \, B_{i,n}(t) \ .$$

Z własności wielomianów Bernsteina

$$\sum_{i=0}^{n} B_{i,n}(t) = 1 \ , \quad B_{i,n}(t) \geqslant 0 \quad \text{dla } t \in [0,1]$$

wynika, że krzywa Bézier leży wewnątrz powłoki wypukłej punktów P_i (czyli w części wspólnej wszystkich zbiorów wypukłych zawierających punkty P_i).



Rys. 2. Łamana i krzywa Bézier (linią przerywaną zaznaczono powłokę wypukłą punktów P_i).

Zmieniając punkty P_i możemy poprawiać kształt krzywej Bézier, jednak nawet zmiana tylko jednego punktu powoduje zmianę całej krzywej. Dopiero połączenie idei Bézier z funkcjami sklejanymi dało rozwiązanie wolne od tej wady. Zaproponował je w 1973 r. R.Riesenfeld. Krzywą Q przedstawiamy w bazie $N_{i,k}$ tzw. B-splajnów (patrz [DB]), które można definiować rekurencyjnie.

Definicja 2. Niech $t_0 < t_1 < \ldots < t_n$ będą danymi węzłami. Dalej niech dla k=1

$$N_{i,1}(t) = \begin{cases} 1 \text{ dla } t \in [t_i, t_{i+1}] , \\ 0 \text{ dla pozostałych } t , \end{cases}$$

a dla $k > 1$

$$N_{i,k}(t) = \frac{t-t_i}{t_{i+k-1}-t_i} N_{i,k-1}(t) + \frac{t_{i+k}-t}{t_{i+k}-t_{i+1}} N_{i+1,k-1}(t) .$$

Funkcje $N_{i,k}$ są funkcjami sklejanymi zgodnie z Definicją 1, mają też ważną dodatkową własność - zerują się poza przedziałem $[t_i, t_{i+k}]$

$$N_{i,k}(t) = 0 \quad \text{dla } t \notin [t_i, t_{i+k}] .$$

W konsekwencji zmiany pojedyńczych punktów P_i określających tak zwaną krzywą B-splajnową postaci

$$Q(t) = \sum_{i=0}^{m} P_i N_{i,k}(t) , \quad (m > k)$$

powodują tylko lokalne (częściowe) zmiany jej kształtu.

Omówienie własności funkcji sklejanych, opisy algorytmów i gotowe procedury można znaleźć w książce C. de Boora [DB].

Przedstawione wyżej główne idee przybliżania i modelowania krzywych można przenieść na powierzchnie. Zacznijmy od obrazowego (rys. 3) przedstawienia konstrukcji powierzchni przy pomocy krzywych. Weźmy przedstawienie parametryczne krzywej S(u) w dowolnej bazie np. funkcji F_i, a więc

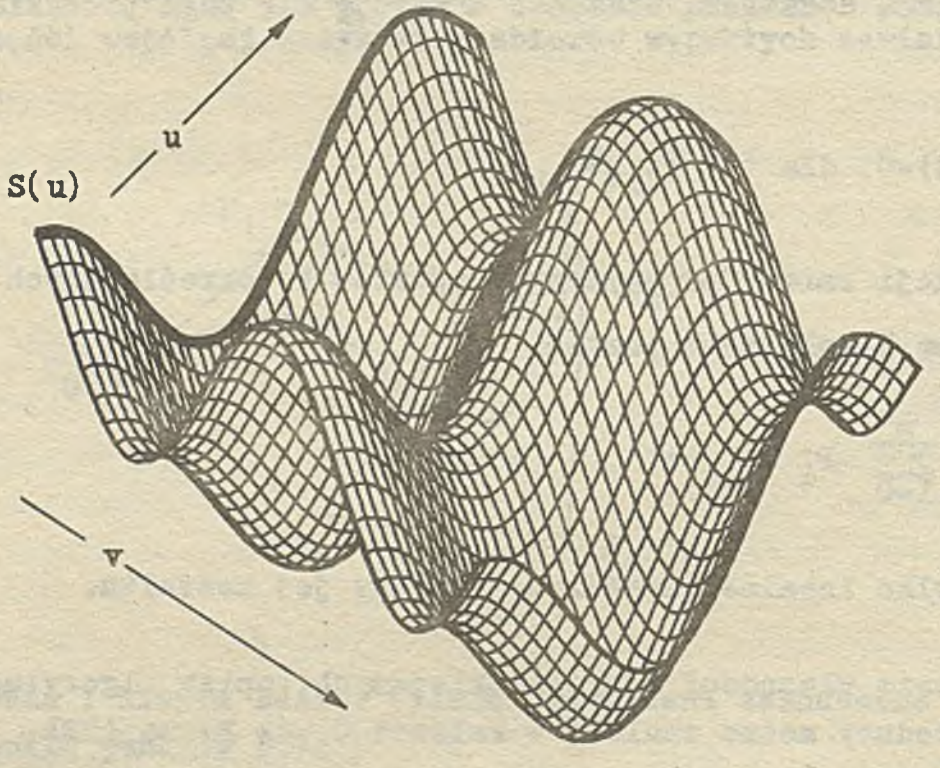
$$S(u) = \sum_{i=0}^{n} P_i F_i(u) .$$

Niech teraz punkty P_i poruszają się w przestrzeni po krzywych

$P_i(v) = \sum_{j=0}^{m} P_{ij}\, G_j(v)$, gdzie G_j są funkcjami bazowymi. Otrzymujemy w ten sposób (nazywany iloczynem tensorowym) powierzchnię

$$S(u,v) = \sum_{i=0}^{n} P_i(v)\, F_i(u) = \sum_{i=0}^{n} \sum_{j=0}^{m} P_{ij}\, F_i(u)\, G_j(v) \ .$$

Biorąc jako funkcje bazowe F_i i G_j wielomiany Bernsteina, funkcje sklejane czy B-splajny, otrzymujemy powierzchnie Bézier, powierzchnie sklejane i powierzchnie B-splajnowe (a dokładniej ich fragmenty -"porcje" określone zakresem zmienności parametrów u i v).



Rys. 3.

W najprostszym przypadku (tzn. n=m=3) powierzchnia Bézier (wtedy $F_i = G_i = B_{i,3}$) ma postać

$$S(u,v) = \sum_{i=0}^{3} \sum_{j=0}^{3} P_{ij} \binom{3}{i} \binom{3}{j} u^i (1-u)^{3-i} v^j (1-v)^{3-j} \ , \quad 0 \le u,v \le 1 \ .$$

Z 16-tu punktów P_{ij} definiujących powierzchnię S tylko cztery

narożne P_{00}, P_{03}, P_{30} i P_{33} leżą na niej. Analogicznie jak w przypadku krzywej Bézier, nietrudno znaleźć związki między stycznymi do powierzchni Bézier a punktami P_{ij}. Przykładowo

$$S_u(0,0)=3(P_{10}-P_{00}) \ , \quad S_v(0,0)=3(P_{01}-P_{00}) \ .$$

Można oczywiście określać fragmenty powierzchni dla różnych zakresów wartości parametrów u i v. Jeśli definiujące je punkty P_{ij} spełniają odpowiednie warunki (patrz [FP1), to łączenie sąsiednich "porcji" powierzchni może być gładkie (gładkość oznacza tu ciągłość funkcji i odpowiednich pochodnych kierunkowych).

Inny, obecnie szeroko stosowany, sposób przybliżania powierzchni przy pomocy krzywych zaproponował w 1967 r. S.A.Coons. Dane są cztery odpowiednio przecinające się krzywe $S(u,0)$, $S(u,1)$, $S(0,v)$ i $S(1,v)$, definiujące brzeg fragmentu powierzchni. Wnętrze tego fragmentu $(0<u,v<1)$ możemy przybliżać sumując interpolację liniową krzywych $S(0,v)$ i $S(1,v)$ po zmiennej u

$$S_1(u,v)=(1-u)S(0,v)+uS(1,v)$$

i interpolację liniową krzywych $S(u,0)$ i $S(u,1)$ po zmiennej v

$$S_2(u,v)=(1-v)S(u,0)+vS(u,1) \ .$$

Suma S_1+S_2 nie spełnia jednak warunków brzegowych, np. dla v=0 dostajemy nie $S(u,0)$ ale

$$(S_1+S_2)(u,0)=S(u,0)+[(1-u)S(0,0)+uS(1,0)] \ .$$

Jeśli jednak od S_1+S_2 odejmiemy interpolację liniową dodatkowych składników brzegowych, to otrzymamy przedstawienie tzw. fragmentu powierzchni Coonsa

$$S(u,v)=[(1-u)\ u]\begin{bmatrix} S(0,v) \\ S(1,v) \end{bmatrix} + [S(u,0)\ S(u,1)]\begin{bmatrix} 1-v \\ v \end{bmatrix}$$

$$- [(1-u)\ u]\begin{bmatrix} S(0,0) & S(0,1) \\ S(1,0) & S(1,1) \end{bmatrix}\begin{bmatrix} 1-v \\ v \end{bmatrix},$$

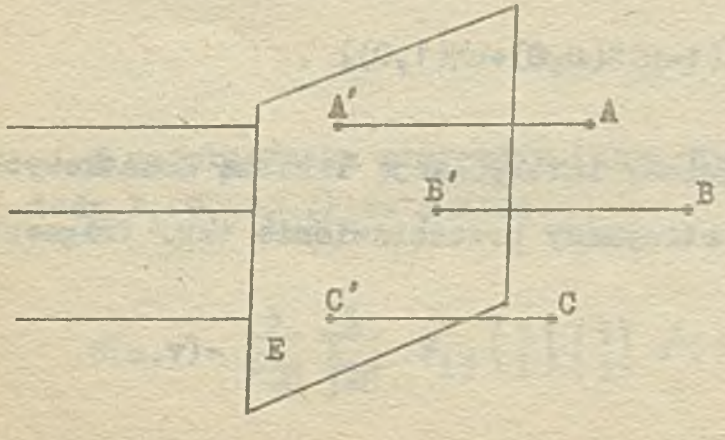
którego brzegami są wyjściowe krzywe.

W praktycznych zastosowaniach przybliżamy kształty złożonych obiektów łącząc fragmenty powierzchni Coonsa. Aby takie łączenie było gładkie, konieczne jest przedstawienie powierzchni Coonsa w ogólniejszej postaci - zawierającej nie tylko krzywe brzegowe ale i odpowiednie pochodne (informacje o kierunkach stycznych do powierzchni na jej brzegach).

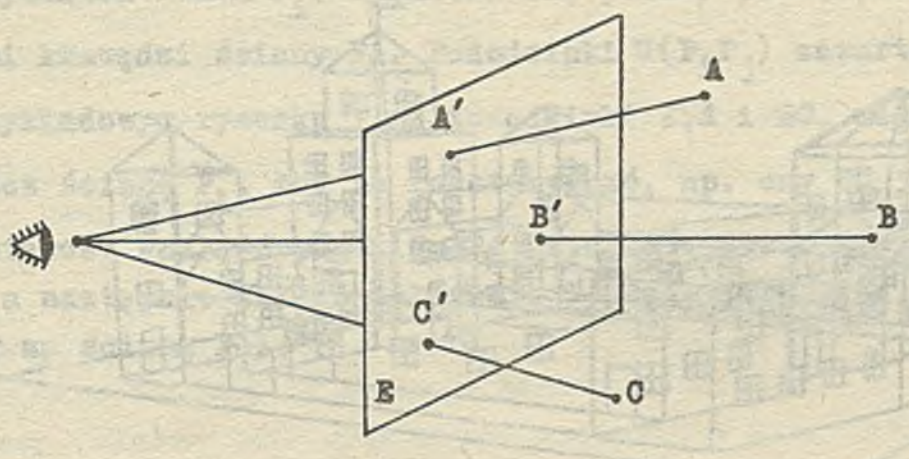
Objętość tego referatu nie pozwala na omówienie innych (często bardzo ciekawych matematycznie, mających związek np. z metodą elementu skończonego czy aproksymacją) metod modelowania powierzchni. Zainteresowanym tą tematyką polecam przeglądowy artykuł [BÖH].

6. Rysowanie obiektów trójwymiarowych.

Najczęściej stosowanym przedstawieniem obiektu trójwymiarowego na dwuwymiarowej płaszczyźnie ekranu lub kartki papieru jest jego rzut równoległy lub środkowy (nazywany też rzutem perspektywicznym). Sposoby rzutowania ilustrują rysunki 4 i 5.



Rys. 4. Rzut równoległy na płaszczyznę E.

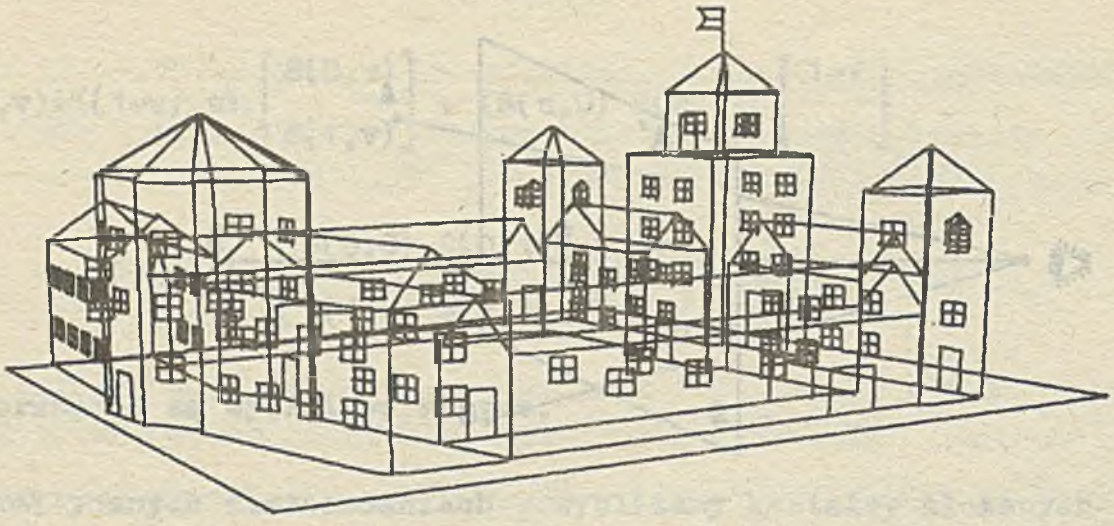


Rys. 5. Rzut środkowy na płaszczyznę E.

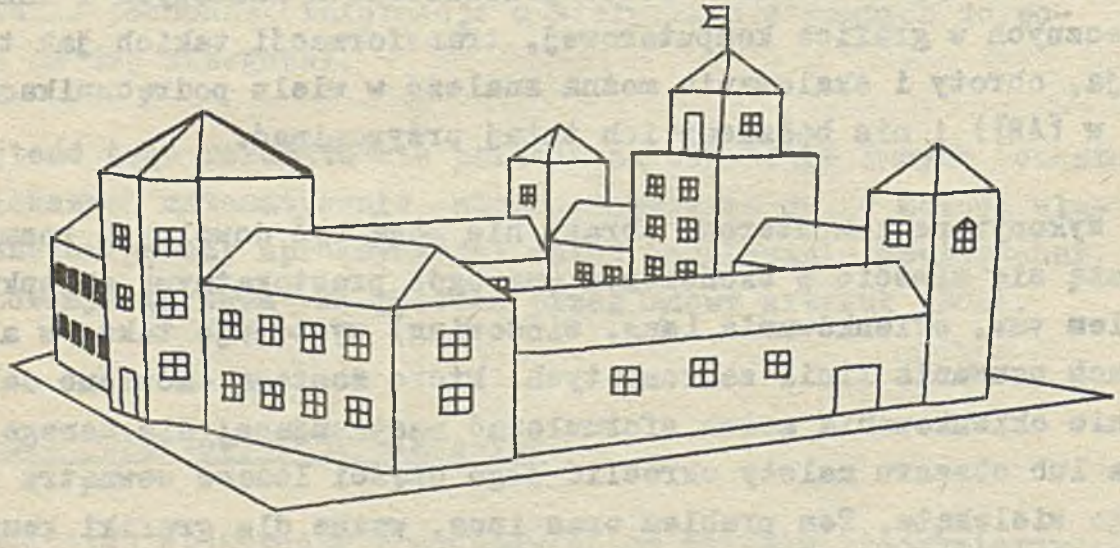
Proste wyprowadzenie wzorów przekształceń rzutowych i innych, użytecznych w grafice komputerowej, transformacji takich jak translacja, obroty i skalowanie można znaleźć w wielu podręcznikach (np. w [AN]) i nie będziemy ich tutaj przypominać.

Wykonywane komputerowo obrazy nie mogą być dowolnych rozmiarów - muszą się mieścić w skończonym (na ogół prostokątnym) okienku. Problem tzw. okienkowania (ang. windowing) występuje także w algorytmach usuwania linii zasłoniętych, które zostaną omówione dalej. Zadanie okienkowania można sformułować następująco: dla danego odcinka lub obszaru należy określić jego części leżące wewnątrz ustalonego wielokąta. Ten problem oraz inne, ważne dla grafiki komputerowej, zadania lokalizacji punktów, wyznaczania powłok wypukłych, triangulacji i wypełniania obszarów są zadaniami rozwijającej się od niedawna dyscypliny - geometrii obliczeniowej [LP].

Rysunki dwuwymiarowych rzutów obiektów trójwymiarowych są często zupełnie nieczytelne, jeśli nie usuniemy linii zasłoniętych (por. rysunki 6 a i 6 b). Rozstrzyganie, czy dana linia (obszar) jest widoczna czy zasłonięta, stanowi jedno z ciekawszych i trudniejszych zadań grafiki komputerowej. Znanych jest wiele jego rozwiązań (patrz artykuł przeglądowy [SU]) i ciągle są publikowane nowe (np. [SCH]).



Rys. 6 a. Projekt domku jednorodzinnego, narysowane linie zasłonięte.

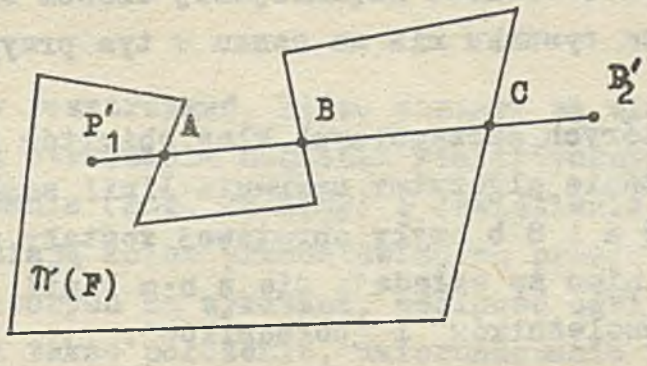


Rys. 6 b. Ten sam projekt, ale z usuniętymi liniami zasłoniętymi.

Koncepcyjnie najprostszym algorytmem wyznaczania linii zasłoniętych wydaje się być algorytm Ricciego, stosowany do obiektów budowanych ze ścian będących wielokątami (np. takich jak na rys. 6). Dokładniej: punkty $P \in R^3$ określamy współrzędnymi (x,y,z), podając punkt początkowy P_i i końcowy P_j definiujemy odcinek P_iP_j, i wreszcie ściana F jest określona ciągiem $P_0P_1, P_1P_2, \ldots, P_{k-1}P_k, P_kP_0$ współpłaszczyznowych i nie przecinających się odcinków (krawędzi).

Oznaczmy przez π przekształcenie rzutowe obiektu trójwymiarowego na płaszczyznę rysunku. W algorytmie Ricciego rzut $\pi(P_iP_j)$ każdej krawędzi P_iP_j porównywany jest kolejno z rzutami $\pi(F_l)$

wszystkich ścian F_1. Wyznacza się punkty przecięcia $\pi(P_iP_j)$ z rzutami krawędzi ściany F_1. Pododcinki $\pi(P_iP_j)$ zawarte w $\pi(F_1)$, na przykładowym rysunku 7 są to odcinki P'_1A i BC, mogą być zasłonięte przez ścianę F_1. Aby to rozstrzygnąć, np. czy BC jest zasłonięty, wystarczy znaleźć punkt $P \in R^3$, którego rzutem jest środek odcinka BC a następnie zbadać położenie P (tzn. określić, czy P leży przed, czy za ścianą F_1).



Rys. 7.

Algorytm Ricciego wymaga co najwyżej $O(n^2 \log n)$ działań, a często tylko $O(n^2)$, gdzie n jest liczbą krawędzi, z których składa się analizowany obiekt.

Innego typu algorytm - oparty o zasadę "dziel i korzystaj" - został zaproponowany przez P.Warnocka. Ekran (kartkę), na którym rysujemy rzut obiektu trójwymiarowego, dzielimy na cztery mniejsze części. Zamiast jednego dużego rysunku, możemy teraz analizować cztery mniejsze, całkowicie niezależne rysunki. Dzieląc ekran podzieliliśmy większość ścian na części, dla sensowności opisu rysunku czasami niezbędne jest też dodanie pewnych nowych krawędzi. W sumie rysunki składają się teraz nie z n krawędzi, ale z $n_1+n_2+n_3+n_4$. Najczęściej jest

$$n < n_1+n_2+n_3+n_4 .$$

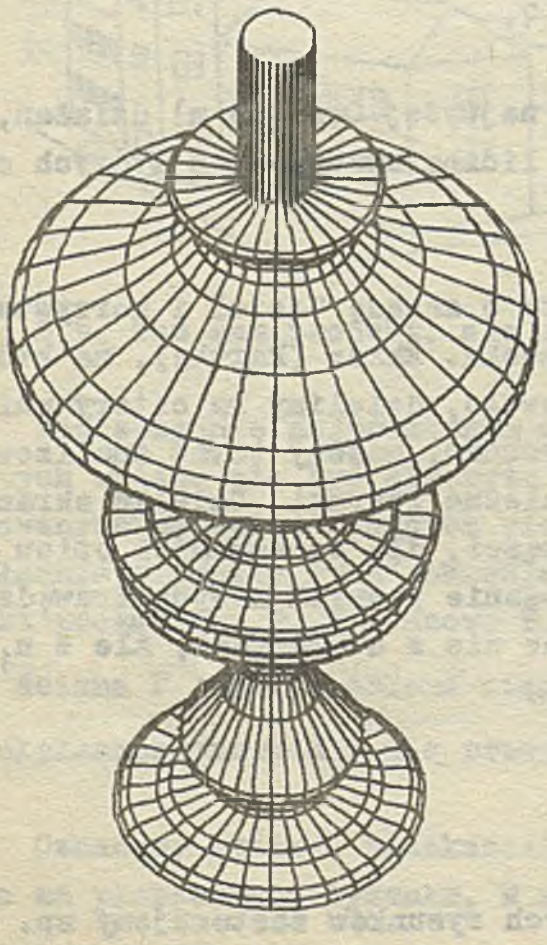
Jeśli do każdego z częściowych rysunków zastosujemy np. algorytm Ricciego, którego koszt na ogół jest proporcjonalny do n_i^2, to

w sumie przeanalizowanie całego rysunku będzie wymagało rzędu $n_1^2+n_2^2+n_3^2+n_4^2$ działań. A często (szczególnie przy złożonych obrazach, kiedy n jest duże) zachodzi nierówność

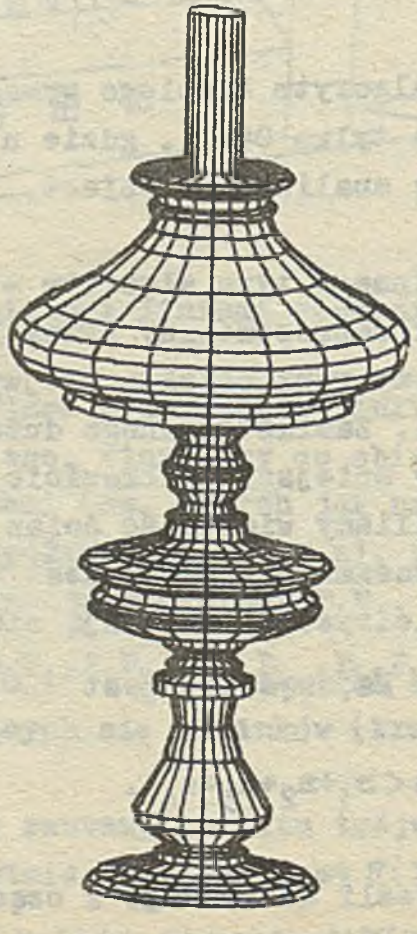
$$n^2 > n_1^2+n_2^2+n_3^2+n_4^2$$

i podział rysunku jest opłacalny. Oczywiście możemy kontynuować takie postępowanie, dzieląc fragmenty rysunku na jeszcze mniejsze części. Przy korzystaniu z urządzeń rastrowych taki podział jest ograniczony przez rozmiar najmniejszej części ekranu (ale dokładniejsza analiza rysunku nie ma sensu w tym przypadku).

Dla niektórych szczególnych klas obiektów trójwymiarowych znane są bardzo tanie algorytmy usuwania linii zasłoniętych. Przykładowo, rysunki 8 a i 8 b bryły obrotowej zostały wykonane kosztem O(n) działań, mimo że składają się z n·m linii, gdzie n i m są liczbami "równoleżników" i "południków".



Rys. 8 a.



Rys. 8 b.

Wydaje się, że zasadę "dziel i korzystaj" należy rozbudować do postaci "dziel złożony rysunek na fragmenty i korzystaj z różnych metod dla ich rysowania". Z drugiej strony trzeba zdawać sobie sprawę z faktu, że dla pewnych scen trójwymiarowych prymitywny algorytm Ricciego jest najtańszy, i nie da się rozwiązać dla nich problemu linii zasłoniętych kosztem mniejszej liczby działań (patrz [SCH]). Oznacza to również, że komputerowe generowanie złożonych obrazów wymaga sprzętu o dużej mocy obliczeniowej. Profesjonalnej grafiki komputerowej nie da się niestety uprawiać na ZX Spectrum.

7. Kolor i światło.

Jeśli potrafimy rozstrzygać, które obszary są widoczne, to następnym krokiem do otrzymania możliwie realistycznych obrazów jest problem cieniowania (ang. shading) i oświetlenia. Stosowane rozwiązania uwzględniają kolor przedstawianych przedmiotów, własności materiałów z których są wykonane, zdolność odbijania i przepuszczania światła a także położenie, ukierunkowanie i rodzaje źródeł światła.

Autor tego referatu ma nadzieję, że niedługo również u nas będzie możliwe praktyczne zajmowanie się tymi problemami.

BIBLIOGRAFIA

[AN] Angell I.O., A Practical Introduction to Computer Graphics, Macmillan Press 1981, wyd. pol. w druku

[BA] Barth W., Dirnberger J., Purgathofer W., The high-level graphics programming language Pascal/Graph, In: [E2], 151-164

[BL] Blaser A. (ed.), Data Base Techniques for Pictorial Application, Springer 1979

[BO] Bo K., Hardware for Computer Graphics and Computer Aided Design, In: [E1], 294-328

[BÖH] Böhm W., Farin G., Kahman J., A survey of curve and surface methods in CAGD, Comp. Aided Geom. Des. 1, 1984, 1-60

[DB] de Boor C., A Practical Guide to Splines, Springer 1978

[CH] Christiansen H.N., Stephenson M.B., Nay B.J., Ervin D.G., Hales R.F., MOVIE.BYU-1981, In: [E2], 57-70

[VD] van Dam A., Computer Software for Graphics, Scientific American, Sept. 1984, 103-113

[E1] Encarnaçao J. (ed.), Computer Aided Design Modelling, Systems Engineering, CAD-Systems, Springer 1980

[E2] Encarnaçao J. (ed.), Eurographics 81, North Holland 1981

[EN] Enderle G., Kansy K., Pfaff G.E., Computer Graphics Programming, Springer 1984

[FP] Faux I.D., Pratt M.J., Computational Geometry for Design and Manufacture, Ellis Horwood 1979, wyd. ros. 1982

[GI] Giloi W., Interactive Computer Graphics, Prentice Hall 1978, wyd. ros. 1981

[JJ] Jankowscy J i M., Przegląd metod i algorytmów numerycznych, WNT 1981

[KU] Kunii T.L. (ed.), Computer Graphics, Springer 1983

[LP] Lee C., Preparata F., Computational Geometry - A Survey, IEEE Trans. on Comp., 1984, vol. C-33, no 12, 1072-1101

[SCH] Schmitt A., Time and space bounds for hidden line and hidden surface algorithms, In: [E2], 43-56

[SU] Sutherland I.E., Sproull R.F., Schumaker R.A., A characterization of ten hidden surface algorithms, Comp. Surv. 6, 1974, 1-55

Jesienna Szkoła PTI
Mrągowo, listopad 1985

SYSTEMY OPERACYJNE DLA MIKROKOMPUTERÓW

Jan Madey
Instytut Informatyki
Uniwersytet Warszawski
PKiN pok.850
00-901 Warszawa
tel. 268-258

1. Wprowadzenie

1.1. Systemy operacyjne

W latach sześćdziesiątych zaczęły dominować na świecie duże systemy komputerowe. W ich skład wchodziły różnorodne urządzenia peryferyjne, masowe pamięci zewnętrzne i na ogół jeden centralny procesor (tzn. jednostka sterująco-przetwarzająca). Składowe systemu były więc bardzo zróżnicowane i w szczególności występowały duże dysproporcje w szybkości ich działania. Urządzenia peryferyjne były w stanie wykonać w najlepszym razie tysiące operacji na sekundę, podczas gdy procesor działał z prędkościa milionów operacji na sekundę. Podobnie, duże różnice występowały w czasie potrzebnym na dostęp (tzn. zapis lub odczyt) do pamięci, w zależności od typu tej pamięci. Pamięć operacyjna była zwykle o rzędy wielkości szybsza od pamięci zewnetrznych, a te z kolei były jeszcze zróżnicowane między sobą, zarówno ze względu na szybkość, jak i tryb dostępu (np. taśma magnetyczna versus bęben). Istotną

cechą tych systemów była możliwość jednoczesnego obsługiwania wielu użytkowników. Było to w pełni uzasadnione ekonomicznie -- pojedynczy użytkownik potrzebował do swojej pracy tylko małego podzbioru zasobów systemu. Jednakże same cechy sprzętu nie wystarczały dla takiego trybu obsługi; konieczne było istnienie specjalnego oprogramowania zarządzającego zasobami komputera, tzw. *systemu operacyjnego*. System operacyjny występował i we wcześniejszym okresie, przy znacznie mniej rozbudowanych i zaawansowanych komputerach. Wydaje się jednak, że dopiero sytuacja z lat sześćdziesiątych spowodowała rozpoczęcie na dużą skalę badań w dziedzinie systemów operacyjnych i ogólnie zajęcie się metodyką projektowania i konstruowania dużych, złożonych programów.

Celem systemu operacyjnego jest stworzenie środowiska, w którym użytkownik może wygodnie wykonywać swoją pracę, w dużym stopniu niezależnie od zewnętrznej sytuacji (w tym od liczby klientów jednocześnie obsługiwanych przez system). W szczególności więc, system operacyjny musi prowadzić odpowiednią politykę rozdziału zasobów komputerowych pomiędzy ubiegających się o nie użytkowników. Ale nie można zapomnieć o względach ekonomicznych -- system operacyjny powinien w ten sposób prowadzić gospodarkę zasobami, aby dbając o interesy indywidualne użytkownika dążyć także do optymalizacji pracy komputera z punktu widzenia wybranych miar wydajności (przepustowość, stopień wykorzystania pewnych zasobów, itd.). W efekcie więc, systemy operacyjne dużych komputerów lat sześćdziesiątych i siedemdziesiątych były bardzo złożonymi i niestety niedoskonałymi programami. Ich wady odbijały się na każdym użytkowniku, podrażając i utrudniając pracę, a koszt ich konstrukcji sięgał niekiedy kilku lat wysiłku wieloosobowych zespołów. Pozytywnym efektem ubocznym takiego stanu rzeczy był rozwój badań w dziedzinie metod projektowania i programowania systemów operacyjnych, a w szczególności rozwój teorii procesów współbieżnych i odpowiednich języków programowania.

1.2. Specyfika mikrokomputerów

W połowie lat siedemdziesiątych nastąpiła prawdziwa rewolucja w zakresie sprzętu komputerowego. Rozwój technologii doprowa-

dził do istotnych zmian jakościowych i ilościowych. Pojawiły się tanie, małe komputery, oparte na mikroprocesorach ośmiobitowych, występujące początkowo w dosyć prostej i ograniczonej konfiguracji (pamięć operacyjna do 64K bajtów, pamięci zewnętrzne na kasetach magnetycznych lub na tzw. dyskach elastycznych, ekran telewizyjny, klawiatura, i niekiedy drukarka). Mikrokomputery takie zaczęły trafiać dosłownie "pod strzechy", t.j. do prywatnych domów, w ręce nie tylko osób obeznanych z informatyką, ale i laików. Tym samym więc zmienił się cel ich stosowania. Uprzednio komputer był wykorzystywany w zakładach przemysłowych, administracji, bankach, ośrodkach badawczych i szkoleniowych, itd., do rozwiązywania na ogół dużych problemów różnej natury. Teraz mikrokomputer stał się narzędziem indywidualnej pracy zawodowej, wspomaga wykonywanie niektórych czynności domowych, ułatwia przyswajanie różnych dziedzin wiedzy, jest inteligentną maszyną do pisania, wyzwala fantazję hobbystów i wreszcie po prostu dostarcza rozrywki.

Jakie były potrzeby odnośnie systemów opercyjnych dla tych pierwszych mikrokomputerów? Przede wszystkim, w zestawach mikrokomputerowych nadal występują różne zasoby, którymi system musi zarządzać. Zmieniła się oczywiście skala problemu, bo (przynajmniej początkowo) zniknęła potrzeba dzielenia zasobów pomiędzy współzawodniczących użytkowników, niemniej jednak sam problem pozostał. W szczególności należało rozwiązać sprawę gospodarki pamięcią zewnetrzną. Następnie, konieczne było wypracowanie metod łatwego podłączania różnorodnych urządzeń, np. drukarek różnych firm. Przy tym, z założenia, podłączenia takie miały być robione przez niefachowców, co nakładało na projektantów systemów określone wymagania. Kolejna sprawa, to tworzenie środowiska dla użytkowników. Chodzi tu zarówno o funkcje systemu jako pośrednika pomiędzy użytkownikiem, a sprzętem i oprogramowaniem, jak i o pośredniczenie na niższym poziomie, tzn. o dostarczanie mechanizmów potrzebnych przy dołączaniu nowych programów systemowych, translatorów, itd. Powstały więc pierwsze systemy operacyjne, konstruowane na ogół w dużym pośpiechu, co było wymuszone sytuacją i potrzebą rynku.

Kolejne lata przyniosły dalszy rozwój technik mikroprocesorowych, a w szczególności rewolucję cenową. Pojawiły się *mikrokompu-*

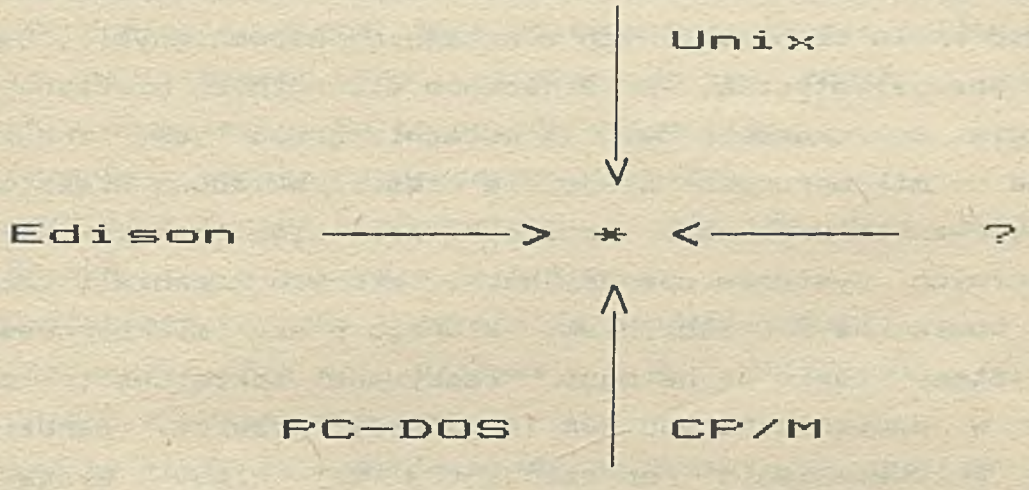
tery osobiste (tzn. przeznaczone dla indywidualnych odbiorców), których parametry są lepsze niż parametry wielu komputerów, stanowiących swego czasu bazę sprzętową różnych ośrodków obliczeniowych. Mikrokomputery takie są już oparte na procesorach 16 bitowych, mają pamięć operacyjną na ogół co najmniej 256K bajtów, pamięć zewnętrzną na dyskach elastycznych dużej pojemności, a dodatkowo często też na dyskach twardych typu Winchester (o pojemności od 10 megabajtów wzwyż), mogą być łączone w lokalne sieci bądź też podłączane do dużych komputerów, a drukarka mozaikowa o możliwościach graficznych staje się ich standardowym wyposażeniem. Przy tym nic nie wzkazuje na to, że zbliżamy się do końca rozwoju technologicznego i zniżki cen. Wręcz przeciwnie, każdy rok przynosi pewne nowości, a w szczególności tanieją istotne komponenty przy jednoczesnym polepszaniu się ich parametrów.

Wszystko to odbija się istotnie na systemach operacyjnych dla mikrokomputerów. Pojawia się na nowo potrzeba rozwiązywania pewnych problemów, które czasowo zniknęły z pola widzenia. W szczególności dotyczy to problematyki współbieżności. Zauważmy bowiem, że nawet jedna osoba pracując na swoim własnym sprzęcie może znacznie skorzystać z faktu, że niektóre czynności mogą być wykonywane jednocześnie, bądź że niektóre zasoby mogą być dzielone. Przypuśćmy na przykład, że korzystamy z mikrokomputera przygotowując książkę. Byłoby wówczas bardzo wygodne, gdybyśmy mogli jednocześnie drukować jeden fragment książki, sprawdzać pod względem poprawności językowej drugi (z pomoca odpowiedniego oprogramowania) oraz pisać kolejny. Obecny sprzęt umożliwia taki tryb pracy, ale musimy dysponować odpowiednim systemem operacyjnym. Inny przykład dotyczy drugiej ze wspomnianych wcześniej sytuacji -- chcielibyśmy podzielić ekran na "okienka", w których byłyby wyświetlane jednocześnie różnego typu informacje. W jednym okienku moglibyśmy na przykład obserwować tekst programu w języku źródłowym, w drugim byłyby umieszczane wyniki tego programu, a w trzecim śledzilibyśmy wartości wybranych zmiennych. Oczywiście okienka powinny mieć zmienne kształty i informacje w nich umieszczane nie musiałyby być ze sobą powiązane. Mamy więc znowu do czynienia z dzieleniem zasobu (ekranu) pomiędzy współbieżne procesy. Ponownie, sytuacja taka jest już możliwa technicznie, ale wymagany jest do tego odpowiedni system operacyjny. Mikrokomputery, jak wspomniano,

mogą być obecnie łączone w sieci, lub też podłączane do istniejących sieci. Zatem typowe zadanie dawnych systemów operacyjnych: umożliwienie grupie osób wygodnej i bezpiecznej współpracy przy założeniu dzielenia zasobów, staje się z powrotem aktualne.

1.3. Kierunki rozwojowe systemów operacyjnych dla mikrokomputerów

Z powyższych rozważań wynika, że systemy operacyjne dla mikrokomputerów są zróżnicowane i podlegają ewolucji. Zastanówmy się obecnie nad trendami w tej dziedzinie. Można wyróżnić cztery podejścia, które są zilustrowane na następującym diagramie:



Strzałka "z góry w dół" oznacza próbę przystosowania systemu operacyjnego, który oryginalnie był opracowany dla komputerów z uprzedniej generacji. Wydaje się, że ten kierunek jest reprezentowany przez jeden tylko, ale za to bardzo ciekawy i ważny system: *Unix*. W dalszej części niniejszego opracowania przedstawiamy zwięźle główne cechy tego systemu (rozdział 3).

Strzałka "z dołu w górę" reprezentuje ewolucję pierwszych prostych systemów operacyjnych dla mikrokomputerów. W grupie tej szczególne znaczenie mają dwa systemy: *CP/M* oraz *PC-DOS*. Pierwszy z nich stał się standardem dla mikrokomputerów ośmiobitowych, a drugi zdominował rynek szesnastobitowy (głównie ze względu na popularność mikrokomputera IBM PC). Systemy te i ich pochodne omawiamy skrótowo w rozdziale 2 poniżej.

Strzałka "z lewej w prawo", to tzw. podejście jednojęzykowe. Idea tego podejścia polega na próbie użycia jednego tylko języka wysokiego poziomu do opracowania całego podstawowego oprogramowania. Co więcej, język ten miałby być również głównym językiem programowania dostępnym dla normalnych użytkowników. Do najbardziej znanych propozycji w tej grupie należy *UCSD Pascal*, *Modula-2* oraz *Edison*. Ostatni z tych języków był projektowany z myślą o mikrokomputerach, które mogą występować w sieci. Jest to ciekawa propozycja, warta bliższego zapoznania się z nią. Poświęcamy jej rozdział 4 niniejszego opracowania.

Ostatnia strzałka reprezentuje systemy, które nie należą do żadnej z wcześniej omawianych klas. Są to więc zarówno systemy uniwersalne, opracowane dla konkretnych typów mikrokomputerów (i mające często wiele cech wspólnych z wcześniej wspomnianymi), bądź też systemy specjalistyczne, przeznaczone dla różnych nietypowych trybów użycia mikrokomputerów. Ta ostatnia grupa jest zresztą bardzo ważna -- mikroprocesor staje się częścią składową większości zautomatyzowanych urządzeń. Konieczne jest więc opracowywanie specjalistycznych systemów operacyjnych, których zadania i cechy są ściśle powiązane z urządzeniem, którego pracą mikroprocesor steruje. Systemy takie są na ogół "zaszywane sprzętowo", tzn. umieszczane w pamięciach typu *ROM* (*Read Only Memory*). Bardziej szczegółowe przedstawienie reprezentantów tej klasy systemów wykracza poza ramy niniejszego opracowania.

2. Proste systemy i ich ewolucja

2.1. System CP/M i pochodne

Skrót "CP/M" pochodzi od *Control Program for Microcomputers*, lub - jak niektóre źródła podają - *Control Program/Monitor*. System ten, opracowany około 1975 roku przez jedną osobę, miał na celu wspomaganie tworzenia środowiska programistycznego opartego na języku PL/M dla mikroprocesorów Intel 8080. W szczególności rozwiązany został problem pamięci zewnętrznej -- przyjęto dyski elastyczne jako standardową pamięć i opracowano dla niej dosyć

efektywny i niezawodny system plików. W skład oryginalnego CP/M wchodziły od razu pewne programy użytkowe, takie jak edytor, czy tzw. "debugger". Wkrótce system ten stał się bardzo popularny wśród użytkowników wczesnych mikrokomputerów, a wiele firm zaczęło opracowywać różnorodne oprogramowanie dopasowane do CP/M. Swoją popularność CP/M zawdzięczał zarówno niezawodności i mocy, jak i łatwości użycia (ang. *friendliness*) oraz przenośności. Powszechnie uważa się, że CP/M jest standardem jeśli chodzi o mikrokomputery oparte o Intel 8080, 8085 lub Zilog-80.

Kolejne lata przynosiły wspomniane uprzednio zmiany w rozwoju mikroprocesorów. Firma *Digital Research Inc.*, sponsor CP/M, wprowadziła na rynek nowe edycje tego systemu, zawierające różne usprawnienia (np. w roku 1984 ukazała się wersja *CP/M 3.0*, zwana też *CP/M Plus*). Niezależnie od tego zaczęły się pojawiać systemy oparte na filozofii CP/M i należące do tej rodziny, ale mające istotnie nowe cechy. Są to przede wszystkim:

* *CP/M-86* : wersja na procesory 16 bitowe,
* *MP/M II* : system wielodostępny z podziałem czasu 8 bitowy,
* *MP/M-86* : j.w., wersja 16 bitowa,
* *Concurrent CP/M-86* : wersja ze współbieżnoscią,
* *Concurrent CP/M with Windows* : j.w. z okienkami,
* *CP/NET* : system sieciowy.

Systemy te miały za zadanie wyjść na przeciw wymaganiom rynku oraz życzeniom użytkowników. I tak, CP/M-86 jest przeznaczony dla mikrokomputerów wykorzystujących procesory Intel 8086 lub 8088, a więc w szczególności IBM PC. Concurrent CP/M pozwala na współbieżne wykonywanie kilku programów przez jedną osobę, natomiast MP/M (w wersjach dla 8 i 16 bitowych mikrokomputerów) umożliwia jednoczesną pracę wielu użytkowników. Ostatni z wymienionych systemów, CP/NET, ma za zadanie łączenie w sieć mikrokomputerów sterowanych systemami CP/M i MP/M. Nowsza wersja systemu Concurrent CP/M, *edycja 3.1*, zastępuje już całkowicie MP/M-86 i jest ponadto zgodna z systemem PC-DOS, tzn. umożliwia wykonywanie programów oryginalnie opracowanych dla tego systemu. Warto jeszcze wspomnieć, że ukazała się wersja CP/M zaszyta sprzętowo wraz z procesorem Z-80 (tzw. *Personal CP/M*), a ponadto można nabyć wkładki do mikrokomputerów opartych o inne procesory, np. *CP/M Gold Card* dla Apple (z wersją CP/M Plus).

2.2. Struktura i funkcje CP/M

System operacyjny CP/M ma strukturę hierarchiczną i składa się z trzech modułów, zwanych skrótowo *CCP, BIOS* oraz *BDOS*.

(a) CCP (od *Concole Command Processor*) jest modułem odpowiedzialnym za konwersację z użytkownikiem. Właśnie ta część systemu ma główny wpływ na ocenę całości z punktu widzenia łatwości i wygody użytkowania. CCP interpretuje komendy wprowadzane z klawiatury, wykonuje niektóre z nich (tzw. wbudowane dyrektywy) oraz ładuje i inicjuje wykonanie programów użytkowych. CCP zleca pozostałym dwóm modułom wykonywanie operacji we/wy oraz operacji na plikach (zbiorach danych).

(b) BIOS (od *Basic Input/Output System*) jest modułem odpowiedzialnym za zarządzanie fizycznymi urządzeniami we/wy i jako jedyny moduł systemu odwołuje się do nich bezpośrednio. Ta część systemu musi więc być dopasowywana do konkretnych instalacji i tym samym stanowi wymienny składnik CP/M.

(c) BDOS (od *Basic Disk Operating System*) jest modułem odpowiedzialnym za zarządzanie pamięcią na dyskach elastycznych. Moduł ten tworzy system plików w CP/M, a w szczególności przekształca poziom logiczny (odwołania za pomocą nazw symbolicznych) w fizyczny (konkretne adresy oraz opisy).

CP/M zajmuje pamięć operacyjną według następującego schematu (poczynając od największych adresów):

* BIOS - około 1.5K bajtów,
* BDOS - około 3.5K bajtów,
* CCP - około 2.0K bajtów,
* obszar dla programów (*TPA - Transient Program Area*),
* parametry systemowe - początkowe 256 bajtów.

Jeżeli program użytownika nie mieści się w obszarze TPA, to jest możliwe częściowe zajęcie obszaru systemowego, lecz program ten sam musi spowodować ponowne wprowadzenie systemu do pamięci.

Dla ilustracji systemu operacyjnego tej klasy podamy obecnie wykaz komend, które są interpretowane przez moduł CCP, wraz z bardzo zwięzłym objaśnieniem ich znaczenia. Komendy wbudowane (wykonywane bez potrzeby sprowadzania programu z pamięci dyskowej) są podkreślone.

efektywny i niezawodny system plików. W skład oryginalnego CP/M wchodziły od razu pewne programy użytkowe, takie jak edytor, czy tzw. "debugger". Wkrótce system ten stał się bardzo popularny wśród użytkowników wczesnych mikrokomputerów, a wiele firm zaczęło opracowywać różnorodne oprogramowanie dopasowane do CP/M. Swoją popularność CP/M zawdzięczał zarówno niezawodności i mocy, jak i łatwości użycia (ang. *friendliness*) oraz przenośności. Powszechnie uważa się, że CP/M jest standardem jeśli chodzi o mikrokomputery oparte o Intel 8080, 8085 lub Zilog-80.

Kolejne lata przynosiły wspomniane uprzednio zmiany w rozwoju mikroprocesorów. Firma *Digital Research Inc.*, sponsor CP/M, wprowadziła na rynek nowe edycje tego systemu, zawierające różne usprawnienia (np. w roku 1984 ukazała się wersja *CP/M 3.0*, zwana też *CP/M Plus*). Niezależnie od tego zaczęły się pojawiać systemy oparte na filozofii CP/M i należące do tej rodziny, ale mające istotnie nowe cechy. Są to przede wszystkim:

* *CP/M-86* : wersja na procesory 16 bitowe,
* *MP/M II* : system wielodostępny z podziałem czasu 8 bitowy,
* *MP/M-86* : j.w., wersja 16 bitowa,
* *Concurrent CP/M-86* : wersja ze współbieżnoscią,
* *Concurrent CP/M with Windows* : j.w. z okienkami,
* *CP/NET* : system sieciowy.

Systemy te miały za zadanie wyjść na przeciw wymaganiom rynku oraz życzeniom użytkowników. I tak, CP/M-86 jest przeznaczony dla mikrokomputerów wykorzystujących procesory Intel 8086 lub 8088, a więc w szczególności IBM PC. Concurrent CP/M pozwala na współbieżne wykonywanie kilku programów przez jedną osobę, natomiast MP/M (w wersjach dla 8 i 16 bitowych mikrokomputerów) umożliwia jednoczesną pracę wielu użytkowników. Ostatni z wymienionych systemów, CP/NET, ma za zadanie łączenie w sieć mikrokomputerów sterowanych systemami CP/M i MP/M. Nowsza wersja systemu Concurrent CP/M, *edycja 3.1*, zastępuje już całkowicie MP/M-86 i jest ponadto zgodna z systemem PC-DOS, tzn. umożliwia wykonywanie programów oryginalnie opracowanych dla tego systemu. Warto jeszcze wspomnieć, że ukazała się wersja CP/M zaszyta sprzętowo wraz z procesorem Z-80 (tzw. *Personal CP/M*), a ponadto można nabyć wkładki do mikrokomputerów opartych o inne procesory, np. *CP/M Gold Card* dla Apple (z wersją CP/M Plus).

2.2. Struktura i funkcje CP/M

System operacyjny CP/M ma strukturę hierarchiczną i składa się z trzech modułów, zwanych skrótowo *CCP, BIOS* oraz *BDOS*.

(a) CCP (od *Concole Command Processor*) jest modułem odpowiedzialnym za konwersację z użytkownikiem. Właśnie ta część systemu ma główny wpływ na ocenę całości z punktu widzenia łatwości i wygody użytkowania. CCP interpretuje komendy wprowadzane z klawiatury, wykonuje niektóre z nich (tzw. wbudowane dyrektywy) oraz ładuje i inicjuje wykonanie programów użytkowych. CCP zleca pozostałym dwóm modułom wykonywanie operacji we/wy oraz operacji na plikach (zbiorach danych).

(b) BIOS (od *Basic Input/Output System*) jest modułem odpowiedzialnym za zarządzanie fizycznymi urządzeniami we/wy i jako jedyny moduł systemu odwołuje się do nich bezpośrednio. Ta część systemu musi więc być dopasowywana do konkretnych instalacji i tym samym stanowi wymienny składnik CP/M.

(c) BDOS (od *Basic Disk Operating System*) jest modułem odpowiedzialnym za zarządzanie pamięcią na dyskach elastycznych. Moduł ten tworzy system plików w CP/M, a w szczególności przekształca poziom logiczny (odwołania za pomocą nazw symbolicznych) w fizyczny (konkretne adresy oraz opisy).

CP/M zajmuje pamięć operacyjną według następującego schematu (poczynając od największych adresów):

* BIOS - około 1.5K bajtów,
* BDOS - około 3.5K bajtów,
* CCP - około 2.0K bajtów,
* obszar dla programów (*TPA - Transient Program Area*),
* parametry systemowe - początkowe 256 bajtów.

Jeżeli program użytownika nie mieści się w obszarze TPA, to jest możliwe częściowe zajęcie obszaru systemowego, lecz program ten sam musi spowodować ponowne wprowadzenie systemu do pamięci.

Dla ilustracji systemu operacyjnego tej klasy podamy obecnie wykaz komend, które są interpretowane przez moduł CCP, wraz z bardzo zwięzłym objaśnieniem ich znaczenia. Komendy wbudowane (wykonywane bez potrzeby sprowadzania programu z pamięci dyskowej) są podkreślone.

KOMENDA	*ZNACZENIE*
ASM	wywołanie asemblera Intel 8080
DDT	wywołanie oprogramowania do testowania
DIR	wyświetlanie zawartości katalogu
DUMP	wyświetlanie pliku w kodzie szesnastkowym
ED	wywołanie edytora
ERA	usunięcie pliku
LOAD	konwersja pliku z postaci szesnastkowej na binarną
MOVCPM	utworzenie nowej kopii CP/M w pamięci
PIP	wymiana informacji pomiędzy urządzeniami
REN	zmiana nazwy pliku
SAVE	zapisanie pliku na dysku
STAT	wyświetlanie statystyki o dyskach, plikach, itd.
SUBMIT	wywołanie sekwencji komend zapisanych na pliku
SYSGEN	utworzenie wersji CP/M na dysku
TYPE	wyswietlanie zawartości pliku tekstowego (ASCII)
USER	wprowadzenie prymitywnego systemu ochrony

2.3. Rzut oka na systemy MS-DOS i PC-DOS

Prace nad systemem operacyjnym dla procesorów 16 bitowych były prowadzone w kilku konkurujących ze sobą firmach, z których na uwagę zasługują *Digital Research* oraz *Microsoft*. Pierwsza z nich sponsoruje rodzinę systemów CP/M, a druga -- rodzinę *MS-DOS* (skrót pochodzi od *Microsoft Disk Operating System*). Nazwa MS-DOS została zastąpiona nazwą *PC-DOS* (od *Personal Computer DOS*) w przypadku systemów zainstalowanych na mikrokomputerze IBM PC.

Historia MS-DOS (mająca istotny wpływ na cechy tego systemu) przedstawia się w dużym skrócie następująco. Gdy w maju 1979 firma *Seattle Computer* wykonała prototyp karty 8086 dla szyny S-100, co oznaczało początek ery mikrokomputerów 16 bitowych, stało się konieczne szybkie opracowanie odpowiedniego systemu operacyjnego. Firma Digital Research spóźniała się z zapowiadaną wersją CP/M-86 i w rezultacie w Seattle Computer opracowano w dwa osobomiesiące system *QDOS 0.10 (Quick and Dirty Operating System)*, który miał dużo cech wspólnych z CP/M. Kolejną wersję tego systemu, o nazwie

86-DOS 1.14, wykupiła w 1981 roku firma Microsoft i przemianowała na *MS-DOS*. Podobieństwo MS-DOS z systemem CP/M wynikało z założeń projektowych -- chodziło o uzyskanie łatwej przenośności programów opracowanych pierwotnie dla mikrokomputerów sterowanych CP/M. System MS-DOS był poddawany ciągłym ulepszeniom. W szczególności *wersja 2.00* z 1983 roku została wzbogacona o kilka ważnych cech zaczerpniętych z systemu Unix: hierarchiczna struktura plików, programy filtrujące, anonimowe bufory (*pipes*). Z kolei zapowiedziana na koniec 1985 roku *wersja 4.0* ma wprowadzić pamięć wirtualną oraz współbieżność. Warto także zauważyć, że w połowie 1984 roku firma Digital Research wprowadziła na rynek system *Concurrent PC-DOS*, który jest zmodyfikowaną wersją Concurrent CP/M with Windows i umożliwia wykonywanie programów opracowanych zarówno dla CP/M-86, jak i dla PC-DOS. Doszło więc do pewnej formy połączenia tych dwóch kierunków rozwoju pierwszych systemów mikrokomputerowych.

3. System Unix i jego wersje

3.1. Historia

Systemu *Unix* jest podobnym fenomenem wśród systemów operacyjnych, jak język *Pascal* wśród języków programowania. W obu wypadkach był to produkt autorstwa jednej lub co najwyżej paru osób, tworzony bez reklamy i specjalnego poparcia. W obu wypadkach hasłem naczelnym była *prostota*, a twórcy świetnie wyczuli przy tym nastroje i potrzeby środowiska informatycznego. W efekcie ich dzieła odniosły prawdziwy sukces, zyskując olbrzymią popularność bez pomocy instytucjonalnych sponsorów. Co więcej, zarówno Pascal jak i Unix stają się obecnie standardami dla mikrokomputerow, mimo że pojawiły się one we wczesnych latach siedemdziesiątych i nie były projektowane dla tego typu sprzętu. Na bazie Pascala powstają obecnie nowe języki; na bazie Unixa nowe systemy operacyjne.

Unix opracowano na przełomie lat sześćdziesiątych i siedemdziesiątych w *Bell Laboratories*, na wewnętrzny użytek firmy. System ten, napisany początkowo w asemblerze na komputer DEC PDP-7,

KOMENDA	*ZNACZENIE*
ASM	wywołanie asemblera Intel 8080
DDT	wywołanie oprogramowania do testowania
DIR	wyświetlanie zawartości katalogu
DUMP	wyświetlanie pliku w kodzie szesnastkowym
ED	wywołanie edytora
ERA	usunięcie pliku
LOAD	konwersja pliku z postaci szesnastkowej na binarną
MOVCPM	utworzenie nowej kopii CP/M w pamięci
PIP	wymiana informacji pomiędzy urządzeniami
REN	zmiana nazwy pliku
SAVE	zapisanie pliku na dysku
STAT	wyświetlanie statystyki o dyskach, plikach, itd.
SUBMIT	wywołanie sekwencji komend zapisanych na pliku
SYSGEN	utworzenie wersji CP/M na dysku
TYPE	wyswietlanie zawartości pliku tekstowego (ASCII)
USER	wprowadzenie prymitywnego systemu ochrony

2.3. Rzut oka na systemy MS-DOS i PC-DOS

Prace nad systemem operacyjnym dla procesorów 16 bitowych były prowadzone w kilku konkurujących ze sobą firmach, z których na uwagę zasługują *Digital Research* oraz *Microsoft*. Pierwsza z nich sponsoruje rodzinę systemów CP/M, a druga -- rodzinę *MS-DOS* (skrót pochodzi od *Microsoft Disk Operating System*). Nazwa MS-DOS została zastąpiona nazwą *PC-DOS* (od *Personal Computer DOS*) w przypadku systemów zainstalowanych na mikrokomputerze IBM PC.

Historia MS-DOS (mająca istotny wpływ na cechy tego systemu) przedstawia się w dużym skrócie następująco. Gdy w maju 1979 firma *Seattle Computer* wykonała prototyp karty 8086 dla szyny S-100, co oznaczało początek ery mikrokomputerów 16 bitowych, stało się konieczne szybkie opracowanie odpowiedniego systemu operacyjnego. Firma Digital Research spóźniała się z zapowiadaną wersją CP/M-86 i w rezultacie w Seattle Computer opracowano w dwa osobomiesiące system *QDOS 0.10* (*Quick and Dirty Operating System*), który miał dużo cech wspólnych z CP/M. Kolejną wersję tego systemu, o nazwie

86-DOS 1.14, wykupiła w 1981 roku firma Microsoft i przemianowała na *MS-DOS*. Podobieństwo MS-DOS z systemem CP/M wynikało z założeń projektowych -- chodziło o uzyskanie łatwej przenośności programów opracowanych pierwotnie dla mikrokomputerów sterowanych CP/M. System MS-DOS był poddawany ciągłym ulepszeniom. W szczególności *wersja 2.00* z 1983 roku została wzbogacona o kilka ważnych cech zaczerpniętych z systemu Unix: hierarchiczna struktura plików, programy filtrujące, anonimowe bufory (*pipes*). Z kolei zapowiedziana na koniec 1985 roku *wersja 4.0* ma wprowadzić pamięć wirtualną oraz współbieżność. Warto także zauważyć, że w połowie 1984 roku firma Digital Research wprowadziła na rynek system *Concurrent PC-DOS*, który jest zmodyfikowaną wersją Concurrent CP/M with Windows i umożliwia wykonywanie programów opracowanych zarówno dla CP/M-86, jak i dla PC-DOS. Doszło więc do pewnej formy połączenia tych dwóch kierunków rozwoju pierwszych systemów mikrokomputerowych.

3. System Unix i jego wersje

3.1. Historia

Systemu *Unix* jest podobnym fenomenem wśród systemów operacyjnych, jak język *Pascal* wśród języków programowania. W obu wypadkach był to produkt autorstwa jednej lub co najwyżej paru osób, tworzony bez reklamy i specjalnego poparcia. W obu wypadkach hasłem naczelnym była *prostota*, a twórcy świetnie wyczuli przy tym nastroje i potrzeby środowiska informatycznego. W efekcie ich dzieła odniosły prawdziwy sukces, zyskując olbrzymią popularność bez pomocy instytucjonalnych sponsorów. Co więcej, zarówno Pascal jak i Unix stają się obecnie standardami dla mikrokomputerow, mimo że pojawiły się one we wczesnych latach siedemdziesiątych i nie były projektowane dla tego typu sprzętu. Na bazie Pascala powstają obecnie nowe języki; na bazie Unixa nowe systemy operacyjne.

Unix opracowano na przełomie lat sześćdziesiątych i siedemdziesiątych w *Bell Laboratories*, na wewnętrzny użytek firmy. System ten, napisany początkowo w asemblerze na komputer DEC PDP-7,

tworzył środowisko ułatwiające programowanie. Głównymi odbiorcami i krytykami systemu byli więc jego twórcy. Wkrótce Unix został ponownie zaprogramowany w specjalnie do tego celu opracowanym języku wysokiego poziomu *C* i zainstalowany na komputerach serii PDP-11. Przy tej okazji wprowadzono do systemu różne modyfikacje i rozszerzenia. W 1973 roku Unix został udostępniony ośrodkom akademickim, co bardzo szybko przyniosło mu olbrzymią popularność. Z czasem różne instytucje zajęły się rozwijaniem i dystrybucją systemu Unix, a w szczególności przenoszeniem systemu na inne komputery. Powstało przy tej okazji wiele różnych wersji i mutacji Unixa, często dosyć istotnie różniących się od siebie. Do najbardziej zaawansowanych należą wersje dla komputerów VAX opracowane przez oddział Uniwersytetu Kalifornijskiego w Berkeley (ostatnia, to *Unix 4.2BSD*; skrót oznaczający wersję pochodzi od *Berkeley Software Distribution*). Ale z naszego punktu widzenia bardziej interesujący jest zwrot ku mikrokomputerom. Już obecnie istnieją instalacje Unixa na systemy 16 bitowe z procesorami Intel 8088 i Motorola 68000, a więc w szczególności na IBM PC. Również omawiane w poprzednim punkcie systemy *-DOS mają wiele cech przejętych z Unixa. Wszystko wskazuje na to, że Unix staje się standardowym systemem operacyjnym dla komputerów 16 i 32 bitowych, a dominującą w tym kontekście wersją jest obecnie tzw. *Unix System V edycja 2, (V.2)* z 1984 roku.

3.2. Struktura i główne cechy

Unix jest systemem wielodostępnym wyposażonym w różne narzędzia wspomagające programowanie i czynności edytorskie. Zbiór takich narzędzi nie stanowi części systemu operacyjnego przy tradycyjnej definicji tego pojęcia. Dlatego też termin "Unix" oznacza znacznie więcej, niż sam system operacyjny -- *Unix, to w rzeczywistości bogate środowisko programistyczne.* Wśród dostępnych programów systemowych są znane edytory (*ed, vi, emacs, ...*), programy redagujące teksty (*troff, scribe, ...*), kompilatory różnych języków wysokiego poziomu (*C, Pascal, Fortran, ...*), programy porównujące pliki (*cmp, diff*), wyszukujące określone sekwencje znaków (*grep, awk*), sortujące (*sort*), obsługujące pocztę elektroniczną (*mail*), itd.

Unix składa się z dwóch warstw: *zewnętrznej i wewnętrznej*. Warstwę zewnętrzną tworzą wspomniane wyżej programy systemowe wraz ze specjalnym programem, interpretatorem komend (zwanym *shell*). Program "shell" jest odpowiednikiem modułu CCP z systemu CP/M odpowiedzialnym za konwersację z użytkownikiem. Istnieje jednak istotna różnica: CCP jest częścią stałą, natomiast "shell" jest częścią wymienialną. Każdy użytkownik Unixa może napisać swój własny interpretator komend i wstawić go w miejsce standardowego. W powszechnym użyciu są obecnie dwa interpretatory, tzw. *Bourne shell* (od nazwiska autora) i *C shell*. Warstwa wewnętrzna Unixa, zwana *jądrem* (ang. *kernel*), realizuje podstawowe funkcje systemu operacyjnego. Jądro zajmuje się więc zarówno operacjami wejścia-wyjścia i obsługą przerwań, jak i implementuje pojęcie *procesu* (bardzo istotne w Unixie), prowadzi gospodarkę pamięcią operacyjną i pomocniczą (implementując w szczególności interesujący *system plików*), itd.

W systemie Unix *plik* jest ciągiem bajtów bez żadnej struktury wewnętrznej. Pliki są grupowane w *katalogi*, które same też są plikami zawierającymi informacje o dostępie do innych plików. W efekcie mamy do czynienia z drzewiastą strukturą hierarchiczną plików, do obsługi której istnieje bogaty zestaw prostych narzędzi. Możemy łatwo tworzyć nowe pliki (i dołączać je w dowolne miejsce), usuwać, kopiować, przesuwać. W prosty sposób wykonuje się także operacje na całych poddrzewach. Plik jest identyfikowany przez *ścieżkę* do niego prowadzącą albo od *korzenia* struktury (ścieżka *bezwzględna*), albo od danego katalogu (ścieżka *względna*). Syntaktycznie jest to ciąg nazw plików-katalogów rozdzielanych kreską ukośną "/". Na przykład:

JM/książka/zadania/z13

identyfikuje plik *z13* będący elementem podkatalogu *zadania*, który jest elementem podkatalogu *książka*, a ten z kolei jest podkatalogiem katalogu *JM*. Ścieżka ta jest względna, bo zaczyna się od *JM*. Natomiast

/user/JM/poczta

jest ścieżką bezwzględną identyfikującą plik *poczta* w katalogu *JM*, który jest podkatalogiem *user*. Pierwsza (początkowa) kreska "/" nie jest rozdzielaczem, ale *nazwą korzenia* całej struktury (konwencja przyjęta w Unixie). W typowej instalacji Unixa istnieje

zwykle dosyć rozbudowana biblioteka plików, o pewnej liczbie standardowych katalogów (*user, lib, bin, dev, ...*) i o dużej liczbie katalogów oraz wychodzących z nich poddrzew tworzonych i dynamicznie zmienianych przez użytkowników. W szczególności jeden plik może być znany pod różnymi nazwami w jednym lub więcej katalogach -- dzięki temu można uniknąć kosztownego kopiowania oraz ułatwić współpracę między użytkownikami. Z innych cech charakterystycznych systemu plików w Unixie należy wymienić następujące:

* jednolite traktowanie plików tekstowych i urządzeń,
* elastyczny system ochrony, pozwalający wyróżniać poziomy,
* mozliwość instalacji biblioteki na kilku pakietach dyskowych.

Powyższe omówienie systemu plików było zrobione z perspektywy użytkownika. Operacje dostępne na tym poziomie są realizowane przez jądro, dysponujące odpowiednim zestawem bardziej prymitywnych narzędzi (których nie będziemy tu przytaczać).

Każda akcja realizowana poza jądrem, to pewien *proces*. Procesem jest wykonywanie dowolnego programu, w tym i programu systemowego (interpretatora komend, edytora, itd.). Procesy mogą być dynamicznie tworzone i niszczone; możliwa jest również synchronizacja osiągana przez *oczekiwanie na zdarzenie* -- jądro systemu zawiera zbiór operacji potrzebnych do tego celu (*fork, execve, wait, exit, kill, ...*). Procesy mogą się ze sobą komunikować poprzez anonimowe bufory (*pipes*). Każdy proces jest jednoznacznie identyfikowany i ma swoją przestrzeń adresową, której część może być dzielona z innymi procesami. Szeregowanie procesów odbywa się w oparciu o dynamicznie wyznaczane priorytety w powiązaniu z podziałem czasu (*round robin*). Jak z tego widać, pojęcie *procesów współbieżnych* odgrywa w Unixie podstawową rolę.

Użytkownik styka się z systemem poprzez interpretator komend, t.j. program "shell", tworzący pewien język wysokiego poziomu. Język ten jest bogaty i elastyczny, niemniej jednak przyjęty zwyczaj stosowania zwięzłych skrótów bywa przedmiotem krytyki, zwłaszcza wśród nowicjuszy. Standardowy program "shell" jest silnym narzędziem, dającym wiele ciekawych i użytecznych możliwości. Na przykład, użytkownik może zainicjować kilka czynności, z których pierwsza odbywa się jawnie, a reszta *w tle*. Innymi słowy jednej osobie może odpowiadać w danej chwili kilka współbieżnych

procesów. Inną interesującą cechą jest mechanizm zmiany pliku do którego mają być przekazane wyniki danego programu, lub z którego mają być wprowadzane dane. Na przykład, jeżeli *prog* jest programem, który oczekuje danych z klawiatury, a wyprowadza wyniki na drukarkę, to bez ingerencji w treść tego programu możemy w prosty sposób zapisać polecenie, aby dane były pobrane z pliku *dane*:

% prog < dane

lub by wyniki były wyprowadzane na plik *wyniki*:

% prog > wyniki

lub też aby zarówno wczytywanie, jak i wypisywanie odbywało się z udziałem tych nowych plików:

% prog < dane > wyniki

(Symbol "%" jest *kursorem* oznaczającym gotowość systemu na przyjęcie kolejnej komendy).

Jak już wspominaliśmy, procesy mogą się komunikować poprzez anonimowe bufory, tworzące tzw. *rurociągi* (ang. *pipelines*). Przykładem może być poniższa komenda:

% prog | filtr | lpr

która oznacza, że wyniki programu *prog* stają się danymi wejściowymi pewnego programu *filtr* (który np. może zastępować wielokrotne spacje pojedynczymi), a wyniki programu *filtr* stanowią z kolei dane wejściowe do standardowego programu *lpr* drukowania na drukarce.

Język interpretatora komend (wersja *Bourne shell* i *C shell*) ma wiele konstrukcji programistycznych spotykanych w normalnych językach programowania wysokiego poziomu, co pozwala na odejście od trybu interakcyjnego. Użytkownik może skomponować program w tym języku, zapisany na jakimś pliku, a następnie zlecić wykonywanie tego pliku przez program "shell". Jest to więc mechanizm bardzo uniwersalny i o dużych możliwościach.

3.3. Xenix, Tunis

Niezależnie od wielu wersji i edycji systemu Unix znane są także systemy, które noszą odmienną nazwę, ale są w rzeczywistości pewnymi mutacjami Unixa. Na szczególną uwagę zasługują dwa z nich: *Xenix* oraz *Tunis*. Poniżej przedstawiamy w dużym skrócie podstawowe fakty o tych dwóch systemach.

Xenix jest produktem firmy Microsoft, o której była mowa w punkcie 2.3 niniejszego opracowania. System ten, wprowadzony na rynek w 1980 roku, jest wersją komercyjną Unixa opracowaną na mikroprocesory szesnastobitowe. Została zmodyfikowana przy tym nieznacznie struktura Unixa. Warstwa zewnętrzna jest w Xenixie rozbita na cztery poziomy, z których każdy wykorzystuje poziom niższy. Pierwszy poziom (bezpośrednio nad jądrem systemu) zawiera program "shell" oraz pewne programy systemowe, występujące w standardowym Unixie. Kolejny poziom, to kompilator języka C, biblioteka podprogramów, edytory i programy redakcyjne oraz narzędzia wspomagające uruchamianie programów (m.in. *debugger*). Na trzecim poziomie znajdują się programy systemowe tworzące różnego rodzaju usprawnienia i rozszerzenia, np. pozwalające zastosować system w sieci. Najwyższy poziom stanowią przede wszystkim kompilatory i interpretatory języków Pascal, Fortran, Cobol i Basic, opracowane przez Microsoft. Niezależnie od wprowadzenia tej strukturalizacji, Xenix ma pewne ulepszenia w stosunku do wersji Unixa na której bazował (tzw. *Unix V7* z 1979 roku). Dotyczy to w szczególności reakcji na błędy sprzętowe i komunikacji między procesami. Ponadto zostały wprowadzone w Xenixie dzielone segmenty danych. Konkurująca z Microsoft firma Digital Research również przenosi Unix na mikroprocesory szesnastobitowe, ale opiera się o nowszą wersję z 1984 roku (wspomniana uprzednio wersja *System V*).

Projekt, którego wynikiem jest system Tunis stanowi przecięcie dwóch podejść z diagramu podanego w punkcie 1.3. Z jednej strony mamy tu bowiem do czynienia z wersją systemu Unix, ale z drugiej strony jest to użycie nowoczesnego języka programowania wysokiego poziomu do tworzenia oprogramowania systemowego. Językiem tym jest *Concurrent Euclid* opracowany na początku lat osiemdziesiątych w Uniwersytecie w Toronto. Celem projektu było wykazanie, że gdy się dysponuje właściwym narzędziem (odpowiednim językiem programowania), to zadania programowania systemowego znacznie się upraszczają. Wydaje się, że teza ta została wykazana. W ośrodku akademickim, głównie siłami studentów, zostało zaprojektowane od nowa jądro Unixa i zaprogramowane w języku Concurrent Euclid, w wyniku czego powstał system Tunis (*Toronto UNIwersity System*). Jest on dostępny na komputerze VAX, a obecnie przenoszony także na 16 bitowe mikrokomputery.

4. System Edison - przykład podejścia jednojęzykowego

4.1. Założenia systemu oraz trochę wstępu

Autor systemu Edison, Per Brinch Hansen, od kilkunastu lat zajmuje się systemami operacyjnymi i językami programowania, zarówno od strony badawczej i metodycznej, jak i praktycznej. Jest on głównym projektantem systemu operacyjnego komputera *RC 4000*, którego prototyp był zainstalowany w Zakładach Azotowych w Puławach, a który to system jest cytowany w każdym poważnym podręczniku o systemach operacyjnych. Rozwijając dalej swoje pomysły Brinch Hansen opracował różnego rodzaju narzędzia do programowania systemowego i współbieżnego. Jest on m.in twórcą tzw. *warunkowych rejonów krytycznych*, z których powstały później *monitory*. Monitor jest wygodnym i silnym narzędziem do programowania współbieżnego. Z jego pomocą można zarówno definiować abstrakcyjne typy danych oraz wyrażać modułowość programu (co ma zalety przy każdym programowaniu), jak i zapewniać wyłączność dostępu do dzielonych obiektów (co jest już typową cechą programowania współbieżnego). Monitory występują w kilku językach programowania, a w szczególności w języku *Concurrent Pascal* autorstwa Brinch Hansena oraz w *Concurrent Euclid* o którym była mowa przy okazji systemu Tunis. Obydwa te języki mają takie same założenia projektowe -- są one *bezpieczne* w tym sensie, że dostarczają wyrafinowanych i złożonych narzędzi oraz narzucają duży rygor na programistę. Dzięki temu wiele błędów można wykryć już na poziomie kompilacji i znacznie łatwiej jest wykazywać poprawność programów. Przy tym w obu językach występują też konstrukcje umożliwiające opisywanie niskopoziomowych funkcji systemu operacyjnego. Wadą tego podejścia jest spora złożoność i objętość tych języków. Parametry pierwszych, względnie prostych mikrokomputerów w sposób naturalny narzuciły ograniczenia na złożoność ich podstawowego oprogramowania. Brinch Hansen ujrzał w tym szansę całkowitego zerwania z przeszłością i zastosowania nowego podejścia, którego podstawową cechą jest *prostota*. Cytując z [1], str.2 :

System oprogramowania komputera osobistego jest prosty, jeżeli można się nauczyć jego używania w 1 dzień, poznać jego szczegóły w 1 miesiąc, a skonstruować go w 1 rok.

Dla spełnienia tego postulatu Brinch Hansen zaproponował użycie jednego języka wysokiego poziomu do opracowania kompletnego podstawowego oprogramowania systemowego. Język ten, *Edison*, jest zdaniem jego twórcy "mniejszy i prostszy niż Pascal, ale mocniejszy od kombinacji języków Pascal i Concurrent Pascal".

4.2. Składowe systemu Edison

System Edison był opracowany oryginalnie dla mikrokomputera PDP 11/23, a następnie zainstalowany też na IBM PC. W skład oryginalnej wersji systemu wchodzą następujące programy napisane w języku Edison:

* *Kompilator języka Edison* { 4200 wierszy }
* *System operacyjny* { 1200 wierszy }
* *Edytor ekranowy* { 500 wierszy }
* *Redaktor tekstu* { 400 wierszy }
* *Program drukujący* { 400 wierszy }
* *Asembler Alva dla PDP 11* { 1600 wierszy }

{ Razem ok. 8300 wierszy }

oraz

* *jądro* napisane w asemblerze { 1800 wierszy }

{ Razem ok. 10100 wierszy }

Omawiając w uprzednich rozdziałach inne systemy operacyjne, zwracaliśmy uwagę w szczególności na interpretator komend i na system plikow. W Edisonie język interpretatora komend jest bardzo prosty i ma cechy "suflera", tzn. system podpowiada, jakie parametry należy podać dla określonej komendy. Jest to konwencja wygodna, gdyż nie ma potrzeby pamiętania wszystkich szczegółów, ale i niekiedy męcząca, przegadana. Jeśli chodzi z kolei o system plików, to przyjmuje się, że konfiguracja sprzętowa jest zawsze wyposażona w dwa napędy dysków elastycznych. Każdy z dysków ma swój własny katalog plików o maksymalnej objętosci 1K słów, który jest kopiowany do obszaru pamięci systemu operacyjnego. Pliki na jednym dysku są organizowane w jeden katalog i same nie mają struktury wewnętrznej. Przy projektowaniu systemu plików były brane pod uwagę parametry napędów (głównie ich ograniczenia pojemnościowe i szybkościowe).

4.3. Język Edison

Edison ma swoje korzenie w następujących językach: *Pascal, Concurrent Pascal* oraz *Modula*. Prostota języka jest uzyskana przez eliminację tych konstrukcji, które nie były niezbędne, oraz przez zastąpienie wielu złożonych bardziej elementarnymi. Tym samym więc Brinch Hansen odszedł od swoich uprzednich idei języka rygorystycznego i przez to bezpiecznego. Wynika to po części z rozwoju metodyki programowania współbieżnego, a po części z konieczności ograniczenia języka do niezbędnego minimum. W efekcie więc w języku Edison nie występują:

- *typy okrojone*
- *typ real*
- *warianty w rekordach*
- *pliki*
- *wskaźniki*
- *instrukcje skoku*
- *instrukcje wyboru* (case)
- *instrukcje "powtarzaj" i "dla"* (repeat, for)
- *instrukcje wiążące* (with)
- *monitory, klasy*
- *...*

A oto lista podstawowych pojęć języka Edison:

* *Typy elementarne:*
  - *standardowe* (int, bool, char)
  - *wyliczeniowe* (enum)
* *Rekordy, tablice, zbiory*
* *Stałe, zmienne, wyrażenia*
* *Konstruktory*
* *Procedury*
* *Moduły i obiekty eksportowane*
* *Instrukcje puste* (skip), *przypisania, procedur*
* *Instrukcje warunkowe* (if), *iteracyjne* (while)
* *Instrukcje współbieżne* (cobegin)
* *Instrukcje synchronizujące* (when)

Dla uzyskania pewnego obrazu języka, przedstawimy teraz prosty przykład modułu definiującego abstrakcyjny typ *bufor* i operacje na nim wykonywalne: *put, get*. Bufor jest jednoelementowy. Operacja

put jest wykonalna tylko gdy bufor jest pusty, natomiast operacja *get* tylko gdy bufor jest zapełniony.

```
module
 var slot : char; full : bool
*proc put(c : char)
 begin
   when not full do
     slot := c; full := true
   end
 end
*proc get(var c : char)
 begin
   when full do
     c := slot; full := false
   end
 end
begin full := false end
```

Bufor jest reprezentowany przez lokalną zmienną (*slot*), która jest używana wyłącznie wewnątrz modułu i nie jest znana na zewnątrz. Podobnie jest ze zmienną roboczą *full*. Natomiast na zewnątrz modułu korzystamy z procedur *put* i *get*, które są *eksportowane* (fakt ten jest zaznaczony symbolem "*"). W module inicjuje się także niektóre obiekty lokalne (zmienna *full*).

Powyższy przykład pokazuje, jak w oparciu o istniejące w Edisonie konstrukcje można uzyskać efekt *monitora*. Dla pełności obrazu należy jeszcze zauważyć, że semantyka instrukcji synchronizującej zapewnia wzajemne wykluczanie przy dostępie do zmiennej dzielonej.

4.4. Uwagi końcowe

W Instytucie Informatyki UW skonstruowano system wielomikroprocesorowy z językiem Edison, oparty na kilku komputerach ZX Spectrum połączonych w sieć. Opis tego systemu jest zamieszczony w trzech pracach magisterskich [5], które były mu poświęcone.

BIBLIOGRAFIA

Literatura związana z konkretnymi systemami operacyjnymi oraz z ogólna problematyką systemów jest bardzo obszerna. W jej skład wchodzą zarówno podręczniki, raporty, dokumentacja konkretnych systemów, jak i różne artykuły naukowe i popularne. W poniższym wykazie są umieszczone tylko takie pozycje, z których korzystano w trakcie przygotowywania niniejszego opracowania. I tak, do tematyki rozdziału 2 szczególnie pomocne były pozycje [2, 3, 4, 8, 9, 10, 12, 13, 14]. Do rozdziału 3 wykorzystano [2, 3, 4, 6, 7, 11], natomiast do rozdziału 4: [1, 5].

1. P.Brinch Hansen, *Programming a Personal Computer*, Prentice-Hall, Englewood Cliffs, New Jersey, (1982).

2. *BYTE*, Volume 10, różne numery, (1985).

3. H.M.Deitel, *An Introduction to Operating Systems*, Addison-Wesley, Reading, Massachusetts, (1984).

4. *Digital Research News*, Volume 3, 4 (1983, 1984).

5. C.Dubnicki, P.Kałamajski, W.Wyglądała, *Projekt Edison: cz.1, 2, 3*, zestaw prac magisterskich, Instytut Informatyki Uniwersytetu Warszawskiego, (1985).

6. R.C.Holt, *Concurrent Euclid, the Unix System, and Tunis*, Addison-Wesley, Reading, Massachusetts, (1983).

7. B.W.Kernighan, R.Pike, *The Unix Programming Environment*, Prentice-Hall, Englewood Cliffs, New Jersey, (1984).

8. C.Larson, "MS-DOS 2.0: An Enhanced 16-Bit Operating System", *BYTE*, 8, 11, (listopad 1983), s.285-289.

9. *Osborne 1 User's Reference Guide*, Osborne Computer Corporation, California (1982).

10. T.Paterson, "An Inside Look at MS-DOS", *BYTE*, 8, 6, (czerwiec 1983), s.230-247.

11. J.L.Peterson, A.Silberschatz, *Operating System Concepts*, Addison-Wesley, Reading, Massachusetts, wydanie 2, (1985).

12. S.Schmitt, "MP/M II", *BYTE*, 8, 3, (marzec 1983), s.190-214.

13. M.Sobczyk, "Dyskowy system operacyjny CP/M. Cz. 1-3", *Informatyka*, nr.2, 3, 4, (1983).

14. R.Taylor, P.Lemmons, "A Comparison of CP/M-86 and MS-DOS", *BYTE*, 7, 7, (lipiec 1982), s.330-356.

Druga Jesienna Szkoła PTI
Mrągowo, listopad 1985

MASZYNY I ALGORYTMY RÓWNOLEGŁE

Maciej M. Sysło
Instytut Informatyki
Uniwersytet Wrocławski
ul. Przesmyckiego 20
51-151 Wrocław, tel. 251271

1. Wstęp

1.1. Maszyny i algorytmy równoległe

Wyróżnić można dwa podstawowe sposoby przyśpieszania komputerów: jeden polega na stosowaniu coraz szybszych elementów elektronicznych, a drugi - na wykorzystaniu współbieżności obliczeń. Pierwszy - był siłą napędową rozwoju komputerów od chwili zbudowania pierwszej elektronicznej maszyny cyfrowej ENIAC w 1946 roku. Drugi zaś - pojawił się dopiero na początku lat 60-tych, najpierw w rozważaniach teoretycznych, gdy zaczęto interesować się algorytmami równoległymi. Pierwszy ze sposobów przyśpieszania obliczeń jest całkowicie uzależniony od dostępnej technologii, natomiast drugi - może być rozwijany niezależnie od sprzętu komputerowego. Pomysł uwspółbieżnienia obliczeń pojawił się niemal w tym samym czasie co konstrukcja pierwszej maszyny. Charles Babbage (1791-1871), konstruktor pierwszej zachowanej do dzisiaj maszyny liczącej, zainspirowany został przykładem obliczeń tablic logarytmicznych, których wykonanie mogłoby pochłonąć jedno całe życie, zostały jednak sporządzone w ciągu kilku lat przez zespół 6 uczonych, 8 wyszkolonych pomocników i około 60 rachmistrzów

umiejących jedynie dodawać i odejmować. Z tego przykładu organizacji obliczeń (współbieżnych, zauważmy!), Ch. Babbage zrealizował jedynie ideę maszyny, która mogłaby zmechanizować wykonywanie takich samych działań. W późniejszych latach swej działalności, zauważył jednak także, że takie obliczenia można by wykonywać za pomocą maszyny, która potrafiłaby produkować wiele wyników w tej samej chwili czasu. Idea ta pojawiła się ponownie dopiero wtedy, gdy rozwój elektroniki przybliżył moment zbudowania pierwszej maszyny równoległej.

Wyjaśnimy w tym miejscu różnicę między określeniami *równoległy* (*parallel*) i *współbieżny* (*concurrent*). Tego pierwszego - używać będziemy przeważnie do obiektów statycznych, drugiego zaś - do procesów. Zatem równoległymi mogą być algorytm lub maszyna. Współbieżnymi są natomiast wykonywane w maszynie procesy lub programy będące implementacjami algorytmów równoległych. Mówimy więc o programowaniu współbieżnym.

Udział coraz szybszego hardware'u (realizującego klasyczne obliczenia sekwencyjne) w przyśpieszaniu obliczeń wyraźnie maleje od pewnego czasu. W ostatnim dziesięcioleciu stało się jasne, że coraz trudniej będzie osiągnąć tempo wzrostu szybkości hardware'u podobne do tego, które obserwowaliśmy w pierwszych dziesięcioleciach. Z drugiej strony, rozwój technologii dużej i wielkiej skali integracji (*VLSI*) doprowadził w ostatnich latach do stałego zmniejszania się rozmiarów, a co najważniejsze, także kosztów produkcji podstawowych, operacyjnych i pamięciowych elementów komputerów. Dysponując nową technologią, dalsze zwiększanie możliwości obliczeń zaczęto osiągać dzięki nowym konstrukcjom maszyn, które są w stanie wykonywać jednocześnie więcej niż jedną operację.

Omówimy teraz pokrótce najważniejsze rodzaje maszyn równoległych, przy czym bardzo często będziemy zamiennie używać określeń maszyna i algorytm. Naszym podstawowym celem jest omówienie algorytmów równoległych, te zaś jednak nierozerwalnie związane są z modelem (lub konkretną maszyną) obliczeń, dla których zostały opracowane.

Algorytm równoległy można zdefiniować jako zbiór niezależnych zadań, które mogą być wykonywane jednocześnie, komunikując się ze sobą nawzajem by korzystać ze swoich wyników obliczeń. Klasyfikacja algorytmów równoległych na ogół pokrywa się z klasyfikacją maszyn równoległych, dla których

zostały opracowane. Wyróżniamy trzy podstawowe cechy, wg których można sklasyfikować zdecydowaną większość algorytmów równoległych: metodę sterowania współbieżnością, wielkość rozdrobnienia zadań i geometrię sieci komunikacyjnej.

Sterowanie współbieżnością jest potrzebne do zapewnienia poprawności obliczeń, w których więcej niż jedno zadanie może być wykonywane równocześnie. W tym celu, wymuszane są współdziałania między różnymi zadaniami. Najogólniej, obliczenia mogą być *synchroniczne* dzięki scentralizowanej kontroli i *asynchroniczne*, gdy np. sterowanie odbywa się za pomocą wspólnych danych.

Stopień rozdrobienia algorytmu równoległego można określić maksymalną wielkością obliczeń wykonywanych przez zadanie zanim ma nastąpić komunikacja z innymi zadaniami. Dla przykładu, algorytmy o bardzo dużym rozdrobnieniu wymagają zwykle dość częstej komunikacji między zadaniami, a zatem stosunkowo dużą część obliczeń zajmuje komunikacja.

Na podstawie metody sterowania współbieżnością i wielkości rozdrobnienia zadań, wyróżnić można trzy podstawowe typy maszyn równoległych: MIMD, SIMD i systoliczne.

Maszyna MIMD (*Multiple Instruction Multiple Data*), zwana także maszyną wieloprocesorową, składa się z pewnej liczby działających asynchronicznie procesorów, z których każdy ma niezależny licznik rozkazów i wykonuje swój własny program korzystając przy tym ze wspólnej dla wszystkich procesorów pamięci. Rozdrobnienie zadań w algorytmach na maszyny MIMD jest zwykle niewielkie i poszczególnym procesorom przydzielane są całe fragmenty obliczeń, np. wywołania procedur. Maszynami MIMD są C.mmp, CM* i Pluribus.

W maszynach SIMD (*Single IMD*) procesor centralny steruje pozostałymi procesorami wysyłając do wszystkich taką samą instrukcję (lub operację), która wykonywana jest na danych umieszczonych w lokalnych pamięciach procesorów. Działanie maszyn SIMD jest więc synchroniczne. Wielkości zadań obliczeniowych dla poszczególnych procesorów są średnie lub małe. Przykładem maszyny SIMD jest Illiac IV składająca się z 64 procesorów.

Gwałtowny rozwój mikroelektroniki w ostatnich latach zrewolucjonizował także konstrukcje komputerów. Technologia wielkiej skali integracji pozwala obecnie umieszczać w jednej *kostce* (*chip*) dziesiątki a nawet setki tysięcy elementów. Doprowadziło to między innymi do powstania całych maszyn (procesorów) w pojedyńczych kostkach, a wśród nich tzw. maszyn systolicznych, które mogą być częściami składowymi innych potężniejszych komputerów. Maszyna systoliczna jest regularną siatką elementów operacyjnych o bardzo wąsko wyspecjalizowanym przeznaczeniu, przez które rytmicznie przepływają dane. Jest to więc maszyna synchroniczna ze stałym bardzo dużym rozdrobnieniem zadań.

Jak łatwo zauważyć, procesory (dalej oznaczane czasem w skrócie przez P) w maszynach MIMD są zwykle pełnymi jednostkami operacyjnymi wyposażonymi we wszystkie podstawowe operacje. W maszynach SIMD, procesory ograniczone są zwykle do wykonywania tylko pewnego podzbioru operacji. W maszynach systolicznych natomiast, procesory wykonują najczęściej tylko jedną operację arytmetyczną (i kilka przesyłań). Z tego względu procesory w maszynach systolicznych a czasem także i w maszynach SIMD nazywane są *elementami operacyjnymi* (*processing element*), w skrócie PE.

Sieć połączeń między procesorami w maszynie równoległej może przyjmować różnorodną postać. Bezpośrednie połączenia między procesorami określają komunikację między zadaniami wykonywanymi przez poszczególne procesory. Dla przykładu, procesory mogą być umieszczone w wierzchołkach drzewa binarnego, którego krawędzie wyznaczają możliwe drogi bezpośredniej komunikacji między procesorami. Elementy operacyjne mogą tworzyć jedno- lub wielowymiarową sieć, np. liniową, kwadratową czy sześciokątną.

Powyższa taksonomia maszyn równoległych, której schemat zaczerpnięty został od Kunga [16], ma swoje odbicie w algorytmach równoległych. Zatem, dla przykładu, algorytm równoległy opracowany dla maszyny SIMD z siecią kwadratową nazywamy algorytmem SIMD z siecią kwadratową itp. Podobnie definiujemy algorytmy systoliczne i MIMD.

Reasumując, w algorytmach systolicznych zadania dla poszczególnych elementów operacyjnych są bardzo proste a komunikacja między PE dość częsta. Algorytmy systoliczne są implementowane zwykle bezpośrednio w postaci odpowiedniego hardware'u. Z algorytmami MIMD sytuacja jest niemal odwrotna -

są one przeznaczone do wykonywania w systemach wieloprocesorowych o ogólnym przeznaczeniu. Algorytmy SIMD sklasyfikować można pomiędzy dwoma pozostałymi typami. Na ogół, każdy algorytm systoliczny ma swojego odpowiednika w postaci algorytmu SIMD, w którym poszczególne zadania mają mniejsze rozdrobnienie. Nie każdy jednak algorytm SIMD może być zrealizowany przez maszynę systoliczną.

1.2. Złożoność i efektywność algorytmów równoległych

Ocena efektywności algorytmów (i maszyn) równoległych powinna dostarczyć nam pewnej miary przyśpieszenia obliczeń osiągniętego dzięki użyciu wielu procesorów. Idealnie, maszyna z p procesorami powinna redukować czas obliczeń p razy. Dla wielu problemów nie udało się jednak dotychczas tego osiągnąć.

Definicja złożoności czasowej (lub czasu obliczeń) algorytmu równoległego jest naturalnym rozszerzeniem określenia złożoności algorytmu sekwencyjnego. Funkcja $f(n)$, gdzie n jest rozmiarem problemu, jest złożonością algorytmu równoległego, jeśli czas od chwili rozpoczęcia pracy przez pierwszy z procesorów maszyny do chwili zakończenia pracy przez wszystkie procesory nie przewyższa $f(n)$ dla każdego zadania problemu o rozmiarze n. Dla przykładu, algorytm sekwencyjny obliczający sumę n liczb ma złożoność $O(n)$, gdyż wykonuje n-1 dodawań. Jeśli dodawania mogą być wykonywane współbieżnie i dysponujemy $\lceil n/\log n \rceil$ procesorami, to suma n liczb może być policzona w $2\lceil \log n \rceil$ krokach za pomocą maszyny drzewiastej (por. § 2.3). Zatem, równoległy algorytm dodawania n liczb ma złożoność $O(\log n)$ przy użyciu $n/\log n$ procesorów.

Liczbę procesorów tworzących maszynę równoległą oznaczać będziemy przez p, a funkcję złożoności algorytmu równoległego dla takiej maszyny - przez $t_p(n)$. Zatem $t_1(n)$ jest złożonością algorytmu sekwencyjnego. Dla wielu maszyn, problemów i algorytmów - funkcja $t_p(n)$ jest określona tylko dla pewnych p, a czasem tylko dla jednej wartości p zależnej od rozmiaru problemu n (por. poprzedni akapit). Jeśli p nie jest ograniczone to mówimy o *nieograniczonej równoległości* (*unbounded parrallelism*). Iloraz $s_p(n) = t_1(n)/t_p(n)$ nazywamy *przyśpieszeniem* (*speedup*) algorytmu równoległego zaś $e_p(n) = s_p(n)/p$ - *efektywnością* (*efficiency*) algorytmu. W definicji przyśpieszenia, za $t_1(n)$

przyjmuje się najczęściej złożoność najlepszego algorytmu sekwencyjnego. Przyśpieszenie jest miarą ulepszenia algorytmu sekwencyjnego przez algorytm równoległy, zaś efektywność określa stopień wykorzystania równoległości. Inną miarą efektywności algorytmu równoległego uwzględniającą czas obliczeń i liczbę procesorów jest *koszt* (*cost*) algorytmu, który definiujemy $c_p(n) = p \cdot t_p(n)$. Oczywiste są następujące nierówności $s_p(n) \leq p$ czyli $t_1(n) \leq p t_p(n) = c_p(n)$ oraz $e_p(n) \leq 1$, gdyż algorytm równoległy o złożoności $t_p(n)$ może być symulowany przez algorytm sekwencyjny w czasie $O(c_p(n))$. Zauważmy, że $e_1(n) = 1$, zatem naszym celem niekoniecznie jest optymalizacja $e_p(n)$ przez wybór odpowiedniej liczby procesorów. Dla wspomnianego wyżej równoległego algorytmu obliczania sumy n liczb mamy $t_p(n) = 2\lceil \log n \rceil$, gdzie $p = \lceil n/\log n \rceil$, zatem $s_p(n)$ jest rzędu $p/2$, a $e_p(n)$ - w przybliżeniu 1/2, czyli procesory wykorzystywane są przez ten algorytm średnio w 50 %.

Algorytm równoległy, dla którego $s_p(n) = p$, ma *optymalny* współczynnik przyśpieszenia obliczeń. (Jeśli $s_p(n)$ jest $O(p)$, to mówimy o przyśpieszeniu optymalnym z dokładnością do stałego współczynnika proporcjonalności.) Powodem tego, że wiele algorytmów równoległych nie osiąga optymalnego przyśpieszenia może być np.: (i) problem nie daje się podzielić na podproblemy o prawie tych samych rozmiarach, (ii) korzystanie z zasobów pamięci (zwykle wspólnych) wydłuża pracę procesorów, (iii) zapewnienie synchronizacji między procesorami pochłania dodatkowy czas.

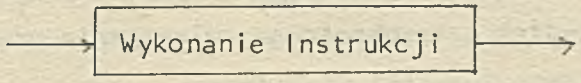
Maszyny równoległe z wielomianową liczbą procesorów oraz opracowane dla nich algorytmy, nie są w stanie zmienić statusu problemów NP-zupełnych, tj. najtrudniejszych problemów kombinatorycznych. W dalszych rozdziałach przedstawiamy szereg *polylogarytmicznych* algorytmów równoległych, tj. o czasie działania $O(\log^k n)$, gdzie k jest stałą. Niestety nie każdy problem w klasie P, tj. problem rozwiązywalny wielomianowym algorytmem sekwencyjnym, ma polylogarytmiczny algorytm równoległy, np. programowanie liniowe, maksymalny przepływ w sieci, i porządkowanie wierzchołków grafu zgodnie z metodą przeglądania DFS. O dwóch ostatnich problemach udowodniono (patrz [22]), że należą do klasy problemów P-zupełnych, która przejęła rolę NP-zupełności w obliczeniach współbieżnych.

Złożoności obliczeniowej problemów w modelach obliczeń współbieżnych poświęcone są prace [12], [25] i wspomniana wyżej [22].

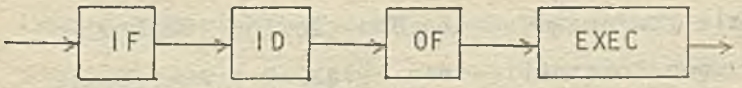
1.3. Przetwarzanie potokowe

Jednym z najwcześniej zastosowanych podejść do współbieżności obliczeń było *przetwarzanie potokowe* (*pipelining*). Najogólniej mówiąc, potokowość obliczeń jest realizowana przez podział wielokrotnie powtarzanego procesu sekwencyjnego na podprocesy, z których każdy wykonywany jest przez niezależny moduł pracujący równocześnie z innymi modułami.

Jako pierwszą ilustrację, rozpatrzmy proces wykonania instrukcji, w którym wyróżnić można przynajmniej cztery podprocesy: pobranie instrukcji (IF), zdekodowanie operacji (ID), pobranie argumentów (OF) oraz wykonanie (EXEC). Rysunek 1 ilustruje potokowe przetwarzanie instrukcji. Ten typ



Procesor niepotokowy



Procesor potokowy

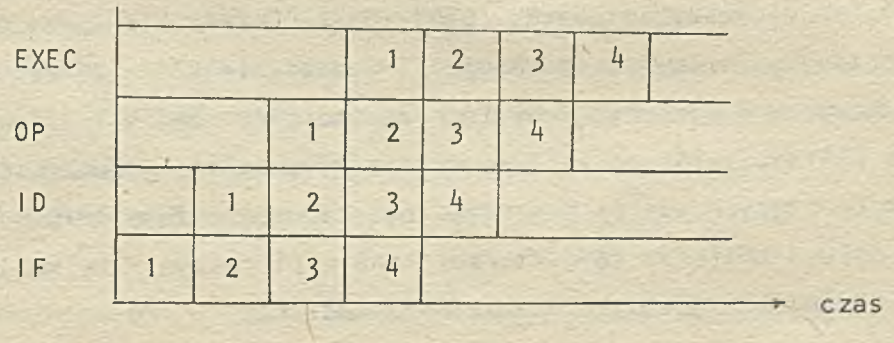


Diagram czasowy

Rys. 1. Potokowe wykonanie instrukcji.

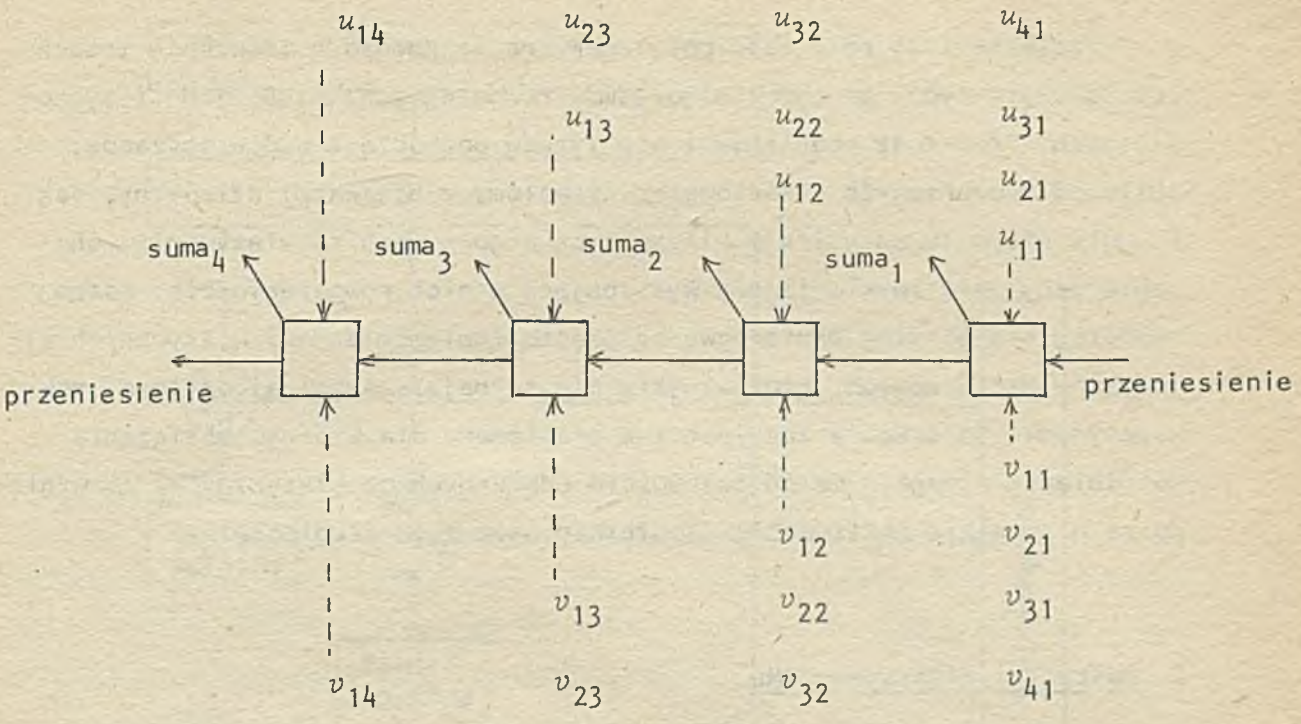
potokowości realizowany jest na poziomie systemu operacyjnego. Do niższego poziomu zaliczyć można potokowe jednostki arytmetyczne. Dla przykładu, w procesie dodawania dwóch liczb zmienno-pozycyjnych wyróżnić można następujące podprocesy, które mogą być realizowane potokowo: (1) wydzielenie cech (wykładników) obu liczb, (2) porównanie cech, (3) przesunięcie mantysy

jednej z liczb dla zrównania cech, (4) dodanie mantys, (5) normalizacja sumy, (6) badanie nadmiaru lub niedomiaru. Oba typy potokowości zostały zastosowane już w latach 60-tych w maszynie IBM 360/91, a obecnie realizowane są w większości superkomputerów, np. w TI ASC, CDC 6600, CDC STAR 100, Amdahl 470 V/6 i CRAY 1 (te dwa ostatnie uznawane są za maszyny czwartej generacji).

Jedną z podstawowych miar oceny działania systemu jest jego współczynnik *przepustowości* (*throughput rate*), definiowany jako liczba wyników (lub wykonanych instrukcji) w jednostce czasu. Potokowość obliczeń jest jedną z metod zwiększenia przepustowości systemów. Powróćmy do przykładu z Rys. 1. Jeśli czas wykonania instrukcji przez procesor niepotokowy wynosi $T = t_1+t_t+t_3+t_4$ i $t = \max\{t_1,t_2,t_3,t_4\}$, to odpowiednie współczynniki przepustowości wynoszą $1/T$ i $1/t$. W szczególności, jeśli wszystkie t_i są sobie równe, to potokowość jest źródłem czterokrotnego zwiększenia przepustowości.

Uzasadnieniem użycia potokowości jest przyjęcie założenia, że ten sam ciąg operacji (instrukcji) będzie powtarzał się bardzo często. Idealną sytuacją dla przetwarzania potokowego są obliczenia związane z wektorami, np. dodawanie dwóch wektorów. Komputery, zdolne w sposób potokowy wykonywać operacje na całych wektorach nazywa się *komputerami wektorowymi* (*vector computers*). Takimi maszynami są np. STAR 100 i CRAY-1. Dla uproszczenia operacji przesyłania zawartości pamięci, w maszynie STAR 100 zakłada się, że wektor utworzony może być jedynie przez kolejne komórki pamięci, zaś w maszynie CRAY-1, składowe wektora mogą znajdować się w komórkach pamięci o numerach tworzących postęp arytmetyczny. Zauważmy, że w drugim przypadku wiersze i kolumn macierzy są wektorami, zaś w pierwszym - tylko jeden typ linii, w zależności od sposobu pamiętania macierzy.

Rysunek 2 przedstawia potokowy sumator wektorów o składowych naturalnych. Sumator ten tworzy sumę dwóch wektorów $(U_1,U_2,U_3,\ldots)$ i $(V_1,V_2,V_3,\ldots)$, gdzie $U_j = u_{j1}u_{j2}\ldots u_{jk}$ i $V_j = v_{j1}v_{j2}\ldots v_{jk}$ są binarnymi reprezentacjami składowych.



Rys. 2. Potokowy sumator wektorów.

Czas wykonania operacji wektorowej składa się z dwóch członów: początkowego opóźnienia δ oraz czasu pojawiania się kolejnych wyników τ. Zatem czas wykonania operacji na wektorach długości N wynosi $\sigma+N\tau$. Dla przykładu, w maszynie STAR 100 dla dodawania wektorów mamy $\tau = .6$ i $\sigma = 71$, zaś dodawanie dwóch skalarów trwa $t = 13$ (czasy te podane są w jednostkach cyklu podstawowego, który wynosi 40 nanosekund). Zatem w tym przypadku, zastąpienie N operacji skalarnych przez jedną operację wektorową zrealizowaną potokowo jest opłacalne począwszy już od $N = 6$ [w ogólnym przypadku, dla $N > \sigma/(t-\tau)$]. Co więcej, czym dłuższe wektory, tym mniejszy wpływ wartości σ na czas obliczeń, gdyż najczęściej $\sigma \gg t$.

Potokowy sumator wektorów z Rys. 2 można uznać za maszynę SIMD a nawet może być zrealizowany przez układ systoliczny. Naszym celem było jednak naszkicowanie ogólnej idei przetwarzania potokowego, które historycznie było pierwszą zrealizowaną koncepcją obliczeń współbieżnych. W dalszych fragmentach tego opracowania ilustrujemy zastosowanie potokowości w maszynach SIMD a zwłaszcza w algorytmach systolicznych.

Przetwarzanie potokowe omówione jest szczegółowo w [21].

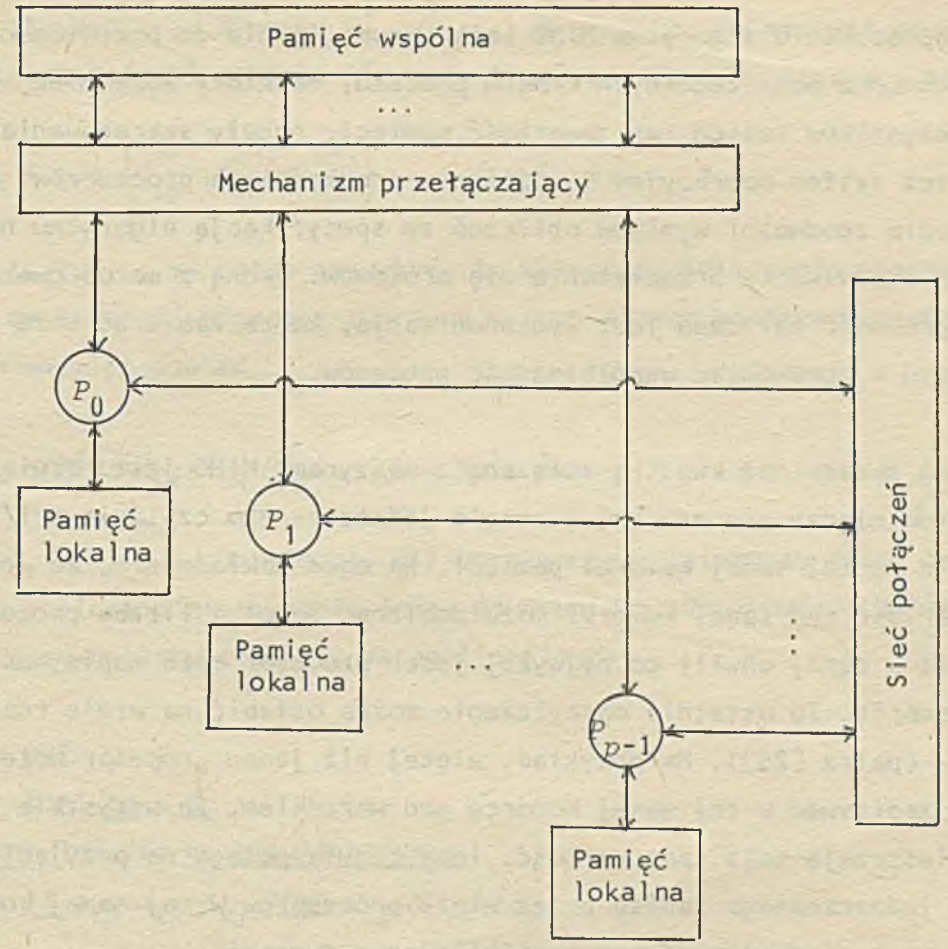
Następne trzy rozdziały poświęcamy na szczegółowe omówienie trzech podstawowych typów maszyn i algorytmów równoległych: MIMD, SIMD i systolicznych. Przykłady problemów i algorytmów pochodzą z dwóch obszarów: obliczeń macierzowych i sortowania. Problemy z pierwszej dziedziny, tak w swoim sformułowaniu jak i klasycznych algorytmach rozwiązywania, charakteryzują się zwykle jawnie występującą w nich równoległością. Zatem równoległe algorytmy macierzowe są często implementacjami klasycznych algorytmów macierzowych, które zwykle nie są najszybszymi algorytmami sekwencyjnymi. Sortowanie zaś jest tym problemem, dla którego obliczenia współbieżne wymagają metod całkowicie odmiennych od klasycznych, ze względu na niewielkie możliwości uwspółbieżniania tych drugich.

2. Maszyny i algorytmy MIMD

2.1. Maszyny MIMD

Maszyna MIMD, zwana także asynchronicznym *wieloprocesorem* (*multiprocessor*) składa się z wielu niezależnych procesorów, które mają dostęp do wspólnej pamięci oraz wspólnych urządzeń I/O poprzez centralny mechanizm przełączający (*switching network*). Dopuszcza się, że każdy z procesorów dysponuje lokalną pamięcią, do której nie ma dostępu żaden inny procesor. Działanie maszyny MIMD jest sterowane przez jeden system operacyjny umożliwiający komunikację między procesorami na różnych poziomach. Rysunek 3 przedstawia schemat maszyny MIMD. Wysokie koszty budowy sieci przełączającej ograniczają liczbę procesorów składających się na maszynę MIMD. Maszyny, a których piszemy dalej, mają odpowiednio 16 i 8 procesorów.

Różny stopień zwartości sieci połączeń między procesorami pozwala włączać do rodziny maszyn MIMD także systemy rozproszone a nawet sieci komputerowe. Wykluczamy jednak te obiekty z naszych rozważań, gdyż składają się one zwykle z wielu samodzielnych komputerów, podczas gdy w naszym rozumieniu komputer równoległy jest pojedyńczą maszyną wyposażoną w wiele procesorów.



Rys. 3. Schemat maszyny MIMD.

Pierwszą maszyną typu MIMD była D-825 zbudowana przez Burroughs w 1962 roku. Najbardziej znanymi maszynami MIMD są C.mmp i HEP. Maszyna C.mmp została zbudowana i zainstalowana w Uniwersytecie Carnegie-Mellon w 1974 roku. Składa się z 16 minikomputerów DEC PDP-11 połączonych ze wspólną pamięcią za pomocą kraty przełączającej (*crossbar switch network*). Największym odstępstwem od przyjętego wzoru maszyny MIMD jest w C.mmp podłączenie urządzeń I/O do poszczególnych procesorów do ich wyłącznego użytku. Maszyna HEP (Heterogeneous Element Processor) zbudowana przez Denelcor pod koniec lat 70-tych składa się z ośmiu identycznych procesorów.

Algorytm na maszynę typu MIMD (zwany w skrócie algorytmem MIMD) można przedstawić jako zbiór współpracujących ze sobą procesów, które mogą być wykonywane jednocześnie. Należy odróżnić pojęcie procesu od pojęcia procesora. *Proces* oznacza wykonanie pewnych obliczeń przez jeden procesor, zaś procesor jest jednostką hardware'u zdolną wykonywać procesy. Przyporządkowaniem procesów procesorom zajmuje się system operacyjny. Podstawowa trud-

ność w opracowaniu algorytmu MIMD leży w tym, iż nie do przewidzenia jest czas wykonania poszczególnych kroków procesu, na który może mieć wpływ szereg czynników takich jak zawartość pamięci, reguły szeregowania stosowane przez system operacyjny i różnice w prędkościach procesorów. Dla zapewnienia zgodności wyników obliczeń ze specyfikacją algorytmu należy zadbać o odpowiednie przeplatanie się procesów. Jedną z metod gwarantujących poprawność obliczeń jest synchronizacja, która zabierać może jednak dużo czasu i ograniczać współbieżność procesów.

Inną zasadniczą kwestią związaną z maszynami MIMD jest rozwiązanie konfliktów między procesorami w czasie jednoczesnego czytania z i/lub zapisywania do tej samej komórki pamięci. Na ogół zakłada się, że jednocześnie zawartość tej samej komórki może pobierać dowolna liczba procesorów, natomiast w danej chwili co najwyżej jeden procesor może zapisywać w tej samej komórce. To ostatnie ograniczenie można osłabić na wiele różnych sposobów (patrz [25]). Na przykład, więcej niż jeden procesor może jednocześnie zapisywać w tej samej komórce pod warunkiem, że wszystkie procesory umieszczają taką samą wartość. Inna reguła polega na przyjęciu, że efektem jednoczesnego zapisu przez wiele procesorów w tej samej komórce jest to co zapisywał procesor o najmniejszym numerze.

2.2. Współbieżna realizacja algorytmu Warshalla-Floyda

W tym paragrafie opiszemy realizację algorytmu Warshalla-Floyda na maszynę HEP (patrz [5]). Zakładamy, że nie więcej niż jeden procesor może pobierać równocześnie zawartość tej samej komórki, natomiast dwa procesy nie mogą ani zapisywać, ani czytać i zapisywać w tej samej komórce jednocześnie.

Algorytm Warshalla-Floyda wyznacza długości najkrótszych dróg między każdą parą wierzchołków w sieci. Niech $G_d = (V,A;d)$ będzie siecią, w której $d: A \to R$ jest funkcją długości łuków. O sieci G_d zakładamy jedynie, że nie zawiera cykli ujemnej długości, gdyż w przeciwnym przypadku problem najkrótszej drogi nie miałby skończonego rozwiązania. Dla uproszczenia rozważań można przyjąć, że $A = V \times V$, a wtedy gdy para (i,j) nie jest łukiem w G, to przyjmujemy $d_{ij} = +\infty$. Niech $D = (d_{ij})$ będzie macierzą odległości między wierzchołkami $V = \{1,2,\ldots,n\}$. Algorytm Warshalla-Floyda (WF) wy-

znacza ciąg macierzy $D^0 = D$, D^1, D^2,..., D^n takich, że macierz długości najkrótszych dróg D^* spełnia $D^* = D^n$. Algorytm WF ma następującą postać

$$d_{ij}^k = \min \left\{ d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1} \right\}, \quad i,j = 1, 2, \ldots, n$$

dla $k = 1, 2, \ldots, n$. Łatwo zauważyć, że wykonując obliczenia wierszami, kolejne macierze D^k mogą być tworzone w tych samych komórkach pamięci. Algorytm ma więc postać

$$d_{ij} = \min \{ d_{ij}, d_{ik} + d_{kj} \}, \quad i,j = 1, 2, \ldots, n$$

dla $k = 1, 2, \ldots, n$. Co więcej, dla ustalonego k, elementy k-tej macierzy mogą być obliczane niezależnie jeden od drugiego. Otrzymujemy zatem następujący równoległy algorytm Warshalla-Floyda:

Algorytm PWF

for k:=1 to n do

for $1 \le i,j \le n$ do simultaneously

T_{kij}: if $d_{ij} > d_{ik} + d_{kj}$ then $d_{ij} := d_{ik} + d_{kj}$

Do udowodnienia poprawności algorytmu PWF, zastosujemy teorię rozwiniętą dla projektowania i kontroli współbieżnych procesów w systemach operacyjnych (por. Rozdz. 2 w [3]). Wprowadźmy najpierw niezbędne pojęcia. *System zadań* (*task system*) $C = (T, \le)$ składa się ze zbioru zadań (obliczeniowych) i relacji (częściowego porządku) $\le$, gdzie $T < T'$ oznacza, że wykonywanie zadania T' można rozpocząć dopiero po zakończeniu zadania T. Z każdym zadaniem $T \in T$ związane są dwa podzbiory pamięci, *dziedzina* (*domain*) D_T oraz *zakres* (*range*) R_T. Zadanie T pobiera wartości z komórek w D_T a zapisuje w komórkach z R_T. Dwa zadania T i T' są *niekolidujące* (*noninterfering*), jeśli albo $T < T'$, albo $T' < T$, albo $D_T \cap R_{T'} = R_T \cap D_{T'} = R_T \cap R_T = \emptyset$. Zadania w T są *nawzajem niekolidujące* (*mutually noninterfering*), jeśli każda para zadań w T jest niekolidująca. Dowodzi się, że każdy system składający się z nawzajem niekolidujących zadań jest *zdeterminowany* (*determinate*), to znaczy, dla ustalonego początkowego stanu pamięci, jakakolwiek kolejność wykonania zadań kończy się identycznym stanem pamięci. Powróćmy do algorytmu PWF i określmy podzbiory zadań, z których każdy składa się z nawzajem niekolidujących zadań. Wtedy, na podstawie podanego wyżej faktu, algorytmy WF i PWF będą wyznaczały identyczne macierze D. Zadanie T_{kij}, dla $k,i,j = 1, 2, \ldots, n$, jest systemem zadań, w którym $T_k = \{T_{kij} : i,j = 1, 2, \ldots, n\}$

i relacja $\leq$ jest pusta $\emptyset$. Zatem $D_{kij} = \{M_{ij}, M_{ik}, M_{kj}\}$, $R_{kij} = \{M_{ij}\}$ oraz jeśli G_d nie ma ujemnych cykli, to $R_{kkj} = R_{kik} = \emptyset$, gdzie M_{ij} jest komórką pamięci przechowującą d_{ij}. Łatwy dowód bezkolizyjności zadań w zbiorze T_k pozostawiamy Czytelnikowi.

Algorytm PWF może być wykonywany przez maszynę MIMD, która dysponuje n^2 procesorami. Niestety, istniejące maszyny MIMD dysponują bardzo ograniczoną liczbą procesorów. Przedstawimy więc realizację algorytmu PWF dla maszyny z p procesorami mając na myśli przede wszystkim HEP, gdzie $p = 8$. Załóżmy więc, że maszyna może wykonywać co najwyżej p procesów jednocześnie. Komunikacja między procesorami odbywa się za pomocą wspólnej pamięci. Zakładamy, że maszyna może wykonywać następujące operacje create, lock i unlock. Jeśli proces P_1 wykonuje instrukcję "create proces P_1", to P_2 rozpoczyna swoje działanie i P_1 jest nadal kontynuowany. Dla każdej komórki pamięci X po wykonaniu przez proces P_1 instrukcji "lock X", każdy inny proces, który usiłuje odczytać, zapisać lub wykonać "lock X" musi poczekać, aż P_1 wykona instrukcję "unlock X". Algorytm PWF-HEP wykorzystuje p-1 razy polecenie create by utworzyć p współbieżnych procesów (MASTER i (p-1) WORKERS) i używa poleceń lock i unlock, by zapewnić synchronizację między procesami. Proces WORKER(l) wyznacza l-ty, (l+p)-ty, (l+2p)-ty,.. wiersze kolejnych macierzy D^k. Zmienna SYN gwarantuje, że żaden proces nie może przejść do liczenia elementów macierzy D^{k+1}, jeśli któryś z procesów nie zakończył jeszcze wyznaczania elementów macierzy D^k.

Algorytm PWF-HEP

```
Proces MASTER
    SYN:=0;
    for l:=1 to p-1 do create WORKER(l);
    execute WORKER(p).
Proces WORKER(l)
    for k:=1 to n do begin
      for i:=l step p until n do
        if d_ik<∞ then
          for j:=1 to n do execute T_kij;
      lock SYN; SYN:=SYN+1; unlock SYN;
L1:   if SYN<p·k then go to L1
    end.
```

Dla poprawnej interpretacji działania algorytmu PWF-HEP zauważmy, że instrukcja zwiększania *SYN* o 1 w trzeciej od końca linii wykonywana jest przez dokładnie jeden procesor. Instrukcja zaś w następnej linii wykonywana jest przez wszystkie procesy i jej efektem jest równoczesne zakończenie wszystkich procesów.

Eksperymenty obliczeniowe przeprowadzone z algorytmem PWF-HEP na maszynie HEP wykazały jego dużą efektywność zwłaszcza dla sieci pełnych i przypadków, gdy liczba wierzchołków w sieci była wielokrotnością liczby procesów, gdyż wtedy wszystkie procesory są obciążone prawie równomiernie (patrz szczegóły w [5]).

Podobne podejście zostało zastosowane w szeregu pracach Autorów z CSD WSU w Pullman do rozwiązywania problemów algebraicznych i równań różniczkowych na maszynie HEP. Polecamy zwłaszcza pracę [19], gdzie system zadań związanych z rozwiązywaniem układu równań liniowych, bardziej rozbudowany niż powyżej, dopuszcza optymalizację stopnia współbieżności poprzez konstrukcję tzw. maksymalnie równoległej relacji poprzedzania między zadaniami.

2.2. Sortowanie na maszynach MIMD

Algorytmy sortujące dla maszyn MIMD w większości przypadków najpierw wyznaczają *rząd* (*rank*) lub po prostu numer każdego elementu, a następnie przesuwają elementy w odpowiednie miejsca. Obliczanie numeru danego elementu odbywa się przez wyliczenie (*enumeration*) liczby wszystkich elementów mniejszych od niego. Ze względu na wymaganą dość dużą liczbę procesorów, są to algorytmy mało praktyczne. Dla ilustracji opiszemy algorytm podany przez Hirschberga [10] o złożoności $O(\log n)$, który wymaga jednak $O(n^{1.5})$ procesorów o wolnym dostępie do wspólnej pamięci. Odstępstwem od zdefiniowanego przez nas modelu maszyny MIMD jest założenie o synchronicznym działaniu procesorów, por. Kroki 3 i 6 w opisie algorytmu.

W Krokach 2 i 5 wykonywane jest sumowanie elementów ciągów $\sqrt{n}$-elementowych. Ogólnie, suma n elementów a_i może być znaleziona na synchronicznej maszynie MIMD z n procesorami za pomocą następującego algorytmu.

Algorytm PSUMA

for i:=0 to $\lceil \log n \rceil$-1 do
  for $1 \le j \le n-2^i$ do simultaneously
    $a_j := a_j + a_{j+2^i}$;
{a_1 jest sumą n elementów a_i}.

Złożoność algorytmu PSUMA wynosi $\lceil \log n \rceil$. Inna metoda sumowania n liczb na maszynie MIMD wymaga jedynie $n/\log n$ procesorów, a jej czas działania wynosi $2\lceil \log n \rceil$. Metoda ta składa się z dwóch kroków. W pierwszym kroku, każdy z procesorów znajduje sumę przydzielonego mu podciągu $\log n$-elementowego, a w następnym kroku, znalezione sumy częściowe kumulowane są metodą binarną. Zachęcamy Czytelnika do udowodnienia poprawności tej metody oraz podanej złożoności (por. [9]).

Algorytm Hirschberga
{Zakładamy, że wszystkie elementy ciągu $a_1, a_2, \ldots, a_n$ są różne, a n jest kwadratem liczby naturalnej.}

Krok 1. Podzielić zbiór n liczb na $\sqrt{n}$ grup złożonych z $\sqrt{n}$ liczb każda.

Następujące operacje wykonywać równolegle w każdej grupie elementów.

Krok 2. Dla każdego j, określić rząd $\mathrm{COUNT1}_j$ elementu a_j w grupie, do której należy, tj. liczbę tych i, dla których $a_i < a_j$ plus 1. W tym celu, każdemu j przydzielamy $\sqrt{n}$ procesorów, zatem $n \cdot n^{1/2}$ procesorów jest potrzebnych dla wszystkich elementów. Zauważmy, że wszystkie porównania między elementami a_i mogą być współbieżnie wykonane w jednostce czasu, a wyniki porównań mogą być skumulowane w czasie $\lceil \log \sqrt{n} \rceil$.

Krok 3. Każdą grupę uporządkować zgodnie z wartościami COUNT1, tj. jednocześnie wykonać $a_{\mathrm{COUNT1}_j} := a_j$. Porządek ten może być znaleziony w stałym (tj. niezależnym od n) czasie i nie pojawi się żaden konflikt pamięci, gdyż wartości COUNT1 są różne (w grupach) dla różnych elementów.

Krok 4. Przydzielić każdemu elementowi a_j zbiór $\sqrt{n}$ procesorów. Procesory te równocześnie wykonują binarne przeszukiwanie uporządkowanych $\sqrt{n}$ grup i określają następujące wartości

$$\text{COUNT2}_{j,y} = \begin{cases} j, & \text{jeśli } a_j \text{ należy do grupy } y, \\ \text{liczba elementów } a_i \text{ w grupie } y \text{ takich, że } a_i < a_j. \end{cases}$$

Wartości $\text{COUNT2}_{j,y}$ mogą być znalezione dla wszystkich elementów a_j w czasie $\lceil \log \sqrt{n} \rceil$ przez $n \cdot n^{1/2}$ procesorów.

Krok 5. Używając $\sqrt{n}$ procesorów dla każdego elementu j, wyznaczyć rząd COUNT_j elementu a_j w całym ciągu, gdzie $\text{COUNT}_j = \sum_y \text{COUNT2}_{j,y}$. Krok ten może być współbieżnie zrealizowany w czasie $\lceil \log \sqrt{n} \rceil$.

Krok 6. Na podstawie wartości COUNT, utworzyć poszukiwane uporządkowanie. Może to być wykonane współbieżnie w stałym czasie.

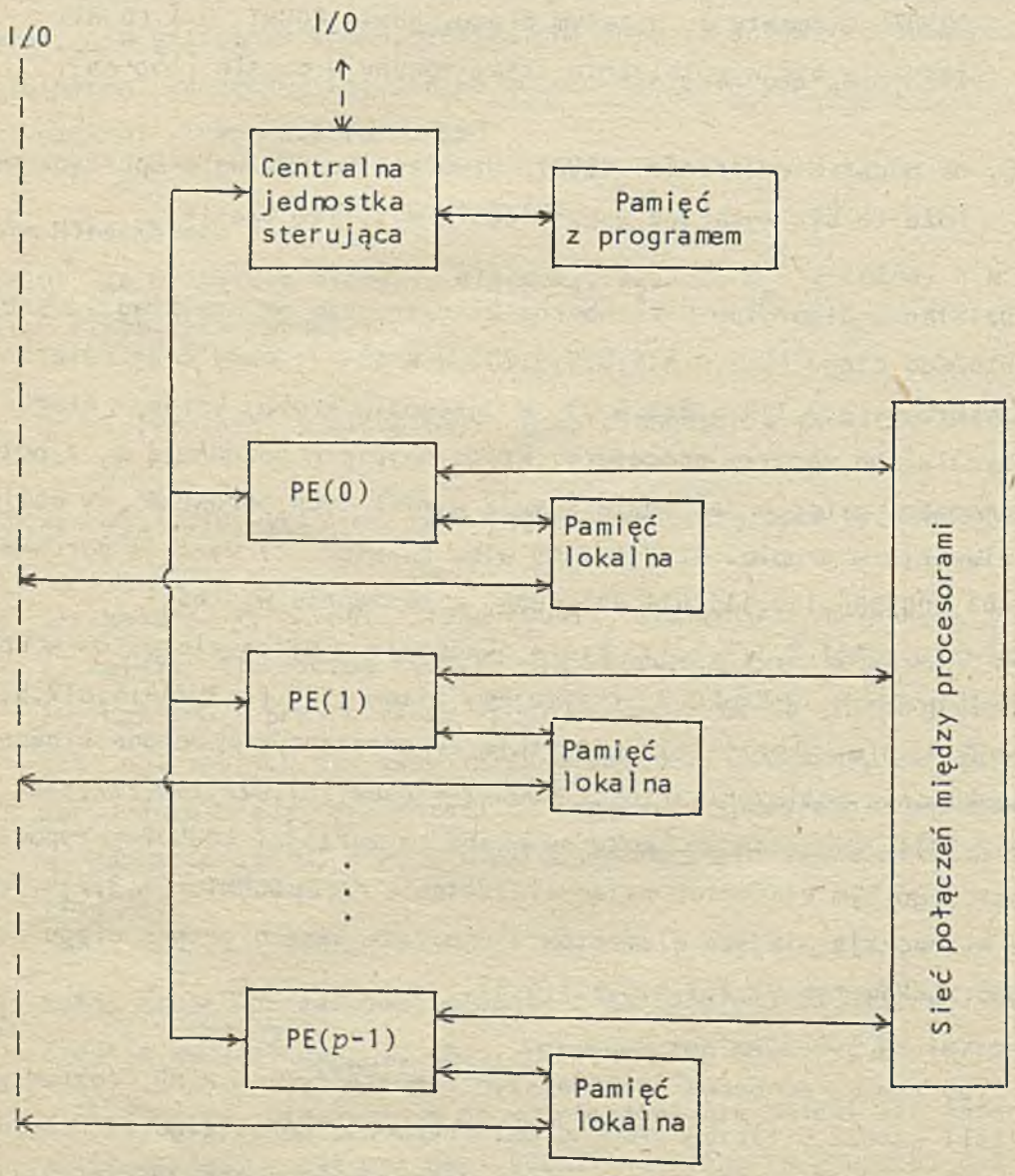
Działanie algorytmu Hirschberga zilustrujemy na przykładzie 9-cio elementowego ciągu (3,7,1,4,6,8,9,5,2). W Kroku 1, cały ciąg dzielony jest na trzy grupy (3,7,1|4,6,8|9,5,2). W następnym kroku, każdemu elementowi a_j przydzielone są trzy procesory, które najpierw porównują a_j z pozostałymi elementami grup, a następnie sumują wyniki tych porównań, by obliczyć rząd elementu w grupie. Otrzymujemy więc najpierw 27 wyników porównań (001,101,000|000,100,110|011,001,000), a następnie wartości COUNT1 = (2,3,1|1,2,3|3,2,1), które określają miejsca elementów w uporządkowanych grupach. W Kroku 3 otrzymujemy zatem ciąg (1,3,7|4,6,8|2,5,9). Krok 4 ponownie wykorzystuje wszystkie 27 procesorów by metodą binarnego przeszukiwania policzyć wartości COUNT2 = (100,201,322|211,222,332|101,212,333). W następnym kroku sumowane są wartości COUNT2 przyporządkowane poszczególnym elementom dając w rezultacie rzędy COUNT = (1,3,7|4,6,8|2,5,9), które wyznaczają miejsca elementów z uporządkowanych grup w ciągu całkowicie uporządkowanym (1,2,3,4,5,6,7,8,9).

Algorytm Hirschberga jest najszybszym algorytmem MIMD, rozrzutnym jednak jeśli chodzi o liczbę procesorów. Preparata podał algorytm równie szybki, ale wymagający tylko $n \log n$ procesorów. Valiant a później Gavril podali algorytmy MIMD wykorzystujące $n/2$ procesorów o czasach działania $O(\log n \log\log n)$ i $O(\log^2 n)$, odpowiednio. Szczegółowe opisy tych algorytmów znaleźć można w pracach przeglądowych [2] i [18].

3. Maszyny i algorytmy SIMD

3.1. Maszyny SIMD

Maszyna SIMD, zwana także *siatką procesorów* (*array processors*), składa się ze zbioru procesorów (oznaczanych w skrócie przez PE), które zsynchronizowane są jednym strumieniem instrukcji (operacji) wysyłanym i sterowanym przez jednostkę centralną (patrz Rys. 4). Procesory wykonują



Rys. 4. Schemat maszyny SIMD.

instrukcje na danych przechowywanych we własnej (lokalnej) pamięci operacyjnej. Zatem, maszyna SIMD może wykonywać ciąg tych samych instrukcji jednocześnie na wielu różnych danych. O każdym z PE zakładamy, że zna i może rozpoznać swój numer. Ponadto, istnieje możliwość stosowania maski, która umożliwia uaktywnianie tylko pewnych procesorów. Najczęściej, zbiór procesorów aktywnych określany jest za pomocą prostego kryterium wyboru, którym może być warunek logiczny umieszczany w algorytmach po instrukcji. Dana instrukcja wykonywana jest przez te procesy, których numery spełniają kryterium. Dla przykładu, jeśli algorytm ma zwiększać zawartość rejestru A(i) w procesorach o parzystym numerze i, to odpowiedni warunek może mieć postać ($i_0 = 0$), gdzie i_0 jest najmniej znaczącym bitem w binarnym rozwinięciu i.

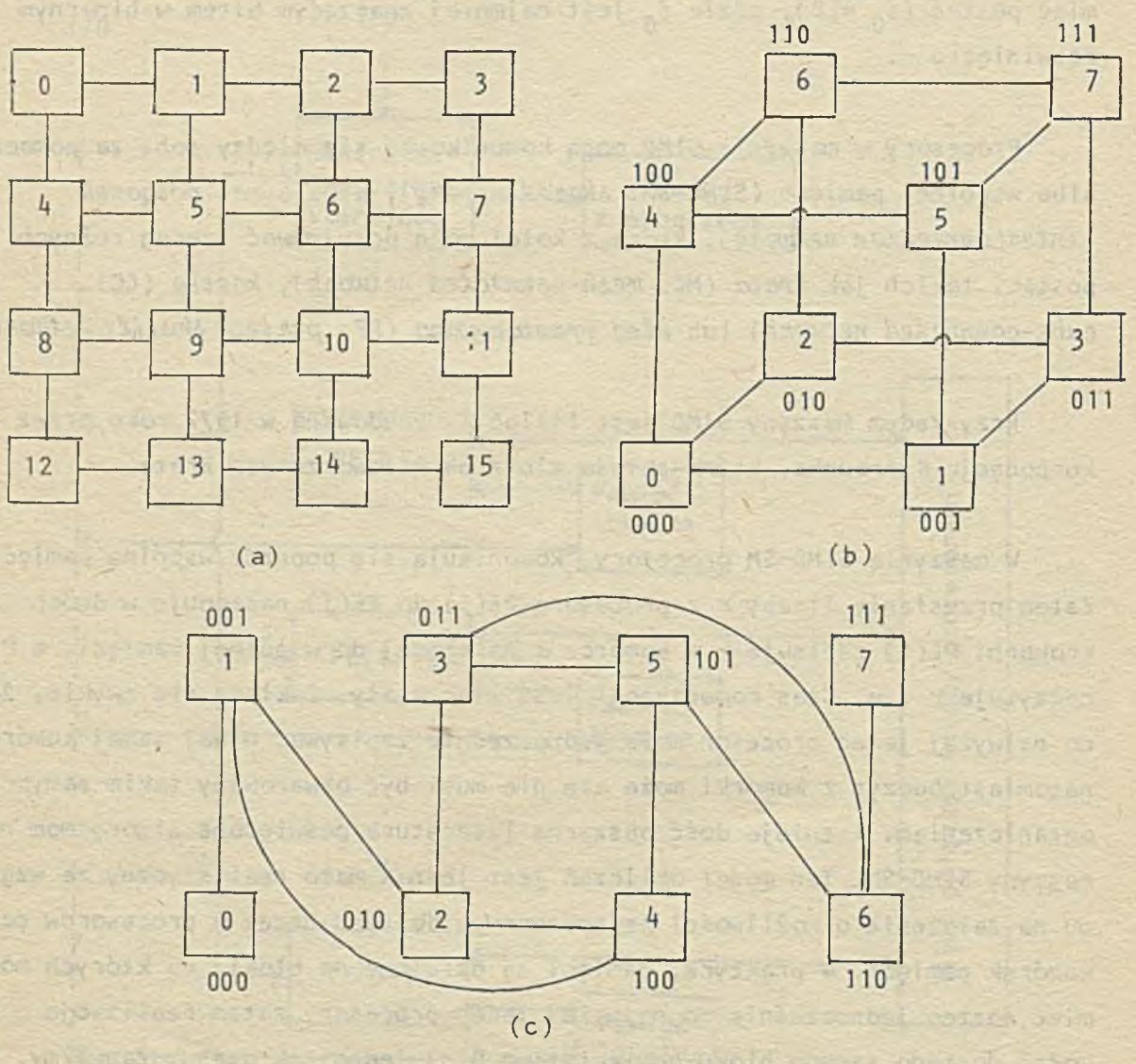
Procesory w maszynie SIMD mogą komunikować się między sobą za pomocą albo wspólnej pamięci (SIMD-SM: *shared memory*), albo *sieci połączeń* (*interconnection network*), która z kolei może przyjmować szereg różnych postaci takich jak *krata* (MC: *mesh-connected network*), *kostka* (CC: *cube-connected network*) lub *sieć przetasowana* (PF: *perfect shuffle network*).

Przykładem maszyny SIMD jest Illiac IV zbudowana w 1972 roku przez korporację Burroughs, która składa się z 64 PE tworzących kratę.

W maszynie SIMD-SM procesory komunikują się poprzez wspólną pamięć. Zatem przesłanie liczby r z procesora PE(i) do PE(j) następuje w dwóch krokach: PE(i) zapisuje r w komórce c należącej do wspólnej pamięci, a PE(j) odczytuje r z c. Czas komunikacji jest więc stały. Zakłada się zwykle, że co najwyżej jeden procesor może jednocześnie zapisywać w tej samej komórce, natomiast odczyt z komórki może ale nie musi być obwarowany takim samym ograniczeniem. Istnieje dość obszerna literatura poświęcona algorytmom na maszyny SIMD-SM. Ten model obliczeń jest jednak mało realistyczny ze względu na założenie o możliwości jednoczesnego dostępu przez p procesorów do p komórek pamięci. W praktyce, pamięci są dzielone na bloki, do których może mieć dostęp jednocześnie co najwyżej jeden procesor, zatem realizacja q odwołań do tego samego bloku pamięci trwa $O(q)$ jednostek czasu. Zauważmy, że maszynę SIMD-SM można uważać za szczególną wersję zsynchronizowanej maszyny MIMD.

Omówimy teraz szczegółowo wspomniane wyżej sieci połączeń między procesorami.

W maszynie SIMD-MC, procesory są ułożone w k-wymiarową tablicę $(n_0,n_1,\ldots,n_{k-1})$, gdzie n_i jest rozmiarem i-tego wymiaru oraz $p = \prod_{i=0}^{k-1} n_i$. Procesor PE w pozycji $P(i_0,\ldots,i_{k-1})$ połączony jest z procesorami w pozycjach $P(i_0,\ldots,i_j\pm1,\ldots,i_{k-1})$ dla $0 \le j \le k$, jeśli tylko istnieją. Zatem, prawie każdy procesor połączony jest z $2k$ innymi procesorami. Rysunek 5 (a) przedstawia SIMD-MC z 16 procesorami dla $k = 2$.



Rys. 5. Przykłady sieci połączeń w maszynach SIMD.

Załóżmy, że $p = 2^q$ i niech $i = [i_{q-1},\ldots,i_0]$ będzie rozwinięciem binarnym liczby i, $i\in[0,p-1]$. Oznaczmy przez $\bar{i}_j$ dopełnienie bitu i_j, i zdefiniujmy $i^{(b)} = [i_{q-1},\ldots,i_{b+1},\bar{i}_b,i_{b-1},\ldots,i_0]$. W maszynie SIMD-CC, zakła-

damy, że każdy procesor i połączony jest z procesorami PE($i^{(b)}$), gdzie $0 \leq b < q$. Zatem, każdy PE(i) połączony jest z $\log p = q$ innymi procesorami. Rysunek 5 (b) przedstawia SIMD-CC z 8 procesorami (tj. $q = 3$).

Niech p, q, i, $i^{(b)}$ będą zdefiniowane jak wyżej. Zdefiniujmy $S(i)$ (*shuffle*) oraz $US(i)$ (*unshuffle*) jako liczby naturalne o rozwinięciach $[i_{q-2}, i_{q-3}, \ldots, i_0, i_{q-1}]$ oraz $[i_0, i_{q-1}, i_{q-2}, \ldots, i_2, i_1]$, odpowiednio. W maszynie SIMD-PS, procesor PE(i) połączony jest z procesorami o numerach $i^{(0)}$, $S(i)$ oraz $US(i)$. Każdy procesor w tym modelu połączony jest z trzema procesorami. Rysunek 5 (c) przedstawia komputer SIMD-PS z 8 procesorami.

3.2. Mnożenie macierzy

Działania na macierzach, ze względu na jawnie tkwiącą w nich równoległość operacji, stanowią najobszerniejszą i najpopularniejszą grupę zadań realizowanych na maszynach SIMD. W tym paragrafie, skupimy swoją uwagę na wyznaczaniu iloczynu dwóch dowolnych macierzy. Działanie to ma bardzo wiele zastosowań. Szereg problemów teorii grafów, dla których metody macierzowe nie dostarczają najefektywniejszych algorytmów sekwencyjnych, ma bardzo efektywne algorytmy równoległe wykorzystujące iloczyn dwóch macierzy (lub pewną jego modyfikację). Do problemów takich należą: wyznaczanie długości najkrótszych dróg w sieci, wyznaczanie przechodniego domknięcia, obliczanie promienia, średnicy i centrum grafu, generowanie drzewa częściowego metodą rozszerzania i topologiczne sortowanie acyklicznego digrafu (szczegóły można znaleźć w [4]).

Iloczyn dwóch macierzy $C = A \cdot B$ stopnia n może być policzony w czasie $O(n)$ na wyidealizowanym komputerze SIMD-SM z n^2 procesorami. Znane są także algorytmy równoległe o czasie działania $O(\log n)$. Algorytm będący realizacją klasycznej metody mnożenia potrzebuje $n^3/\log n$ procesorów, a współbieżna implementacja algorytmu Strassena - wymaga $n^{\log 7}/\log n$ procesorów. Wszystkie trzy algorytmy osiągają optymalne (z dokładnością do stałego współczynnika) przyśpieszenie.

Przedstawimy teraz algorytm Cannona mnożenia dwóch macierzy $A = (a_{ij})$ i $B = (b_{ij})$ stopnia n na maszynie SIMD-MC z $k = 2$, o której zakładamy dla

uproszczenia, że wyposażona jest dodatkowo w połączenia odpowiednich procesorów brzegowych, tak aby elementy w wierszach i kolumnach siatki procesorów mogły być przesuwane cyklicznie. Zakładamy, że każdy procesor PE(i,j) ma trzy rejestry $A(i,j)$, $B(i,j)$ oraz $C(i,j)$, dane macierze A i B umieszczone są odpowiednio w rejestrach A i B, a wynik ma być pozostawiony w rejestrach C. Algorytm składa się z dwóch faz. W pierwszej fazie, elementy i-tego wiersza macierzy A przesuwane są i razy cyklicznie w lewo, a elementy j-tej kolumny macierzy B - j razy cyklicznie w górę. W rezultacie otrzymujemy $A(i,j) = a_{i,(j+i) \bmod n}$ oraz $B(i,j) = b_{(i+j) \bmod n,j}$, a zatem $A(i,j)*B(i,j)$ jest składnikiem sumy tworzącej c_{ij}. W drugiej fazie, każde przesunięcie elementów w rejestrach A i B „nakłada" odpowiednie elementy macierzy A i B na siebie dając kolejne składniki elementów iloczynu. W poniższym opisie algorytmu, symbol := oznacza podstawienie wykonywane w jednym procesorze, zaś ← jest poleceniem przesłania informacji po bezpośrednim połączeniu sieci komunikacyjnej.

Algorytm Cannona

```
begin
  for l:=1 to n-1 do begin
    A(i,j)←A(i,(j+1)mod n);    (i ≥ l)
    B(i,j)←B((i+1)mod n,j)     (j ≥ l)
  end ;
  C(i,j):=A(i,j)*B(i,j);
  for l:=1 to n-1 do begin
    A(i,j)←A(i,(j+1)mod n);
    B(i,j)←B((i+1)mod n,j);
    C(i,j):=C(i,j)+A(i,j)*B(i,j)
  end
end
```

Rysunek 6 ilustruje algorytm Cannona na przykładzie macierzy stopnia 3.

Faza 1						Faza 2											
a_{00}	a_{01}	a_{02}	b_{00}	b_{11}	b_{22}	a_{01}	a_{02}	a_{00}	b_{10}	b_{21}	b_{02}	a_{02}	a_{00}	a_{01}	b_{20}	b_{01}	b_{12}
a_{11}	a_{12}	a_{10}	b_{10}	b_{21}	b_{02}	a_{12}	a_{10}	a_{11}	b_{20}	b_{01}	b_{12}	a_{10}	a_{11}	a_{12}	b_{00}	b_{11}	b_{22}
a_{22}	a_{20}	a_{21}	b_{20}	b_{01}	b_{12}	a_{20}	a_{21}	a_{22}	b_{00}	b_{11}	b_{22}	a_{21}	a_{22}	a_{20}	b_{10}	b_{21}	b_{02}

Rys. 6. Nakładanie się macierzy w kolejnych krokach algorytmu Cannona.

Całkowity czas komunikacji między procesorami wynosi w algorytmie Cannona $4(n-1)$ jednostek, zaś całkowita złożoność algorytmu jest $O(n)$. Kosztem zwiększenia stałej w oszacowaniu złożoności, algorytm Cannona może być zmodyfikowany dla maszyny SIMD-MC bez dodatkowych połączeń procesorów brzegowych (por. [4]). Klasyczny algorytm mnożenia dwóch macierzy ma bardzo prostą implementację w postaci algorytmu systolicznego (por. § 4.1.3).

Opiszemy teraz algorytm mnożenia dwóch macierzy dla maszyny SIMD-CC z n^3 procesorami, gdzie $n = 2^q$. Dla uproszczenia oznaczeń załóżmy, że procesory tworzą siatkę $n\times n\times n$, czyli PE(i,j,k) ma numer in^2+jn+k, gdzie $i,j,k\in[0,n-1]$. Zatem, jeśli $r_{3q-1},\ldots, r_0$ jest binarnym rozwinięciem numeru procesora PE(i,j,k), to $i = r_{3q-1},\ldots, r_{2q}$, $j = r_{2q-1},\ldots, r_q$ oraz $k = r_{q-1},\ldots, r_0$. Tak jak poprzednio, zakładamy, że każdy procesor ma trzy rejestry A, B oraz C, na początku $A(0,j,k) = a_{jk}$, $B(0,j,k) = b_{jk}$, zaś wynik ma być umieszczony w $C(0,j,k) = c_{jk}$, dla $j,k\in[0,n-1]$. Algorytm składa się z dwóch faz. W pierwszej fazie, elementy macierzy A i B rozsyłane są po procesorach tak, że $A(l,j,k) = a_{jl}$ oraz $B(l,j,k) = b_{lk}$. Zaś w drugiej fazie kumulowane są elementy iloczynu.

Algorytm mnożenia macierzy dla maszyny SIMD-CC.

```
begin
  for l:=3q-1 downto 2q do begin
    A(m^(l))←A(m);          (m_l = 0)
    B(m^(l))←B(m)           (m_l = 0)
  end; {A(i,j,k)=a_jk, B(i,j,k)=b_jk dla i∈[0,n-1]}
  for l:=q-1 downto 0 do {kopiowanie A(i,j,i) do A(i,j,*)}
    A(m^(l))←A(m);          (m_l = m_2q+1)
  for l:=2q-1 downto q do {kopiowanie B(i,i,k) do B(i,*,k)}
    B(m^(l))←B(m);          (m_l = m_q+l)
  C(m):=A(m)*B(m); {C(i,j,k)=a_ji b_ik dla i,j,k∈[0,n-1]}
  for l:=2q to 3q-1 do
    C(m)←C(m)+C(m^(l))
end
```

Prześledźmy jedynie działanie ostatniej instrukcji for, która w każdym rejestrze $C(0,j,k)$ kumuluje sumę $c_{jk} = \sum_{i=0}^{n-1} a_{ji}b_{ik}$. Ponieważ zmienna sterują-

ca l przebiega od $2q$ do $3q-1$, zmiana bitu w $m^{(l)}$ następuje na pozycji pierwszego indeksu. Dla $l = 2q$, otrzymujemy $C(i,j,k) \leftarrow C(i,j,k)+C(i+1,j,k)$ dla parzystych i oraz $C(i,j,k) \leftarrow C(i,j,k)+C(i-1,j,k)$ dla nieparzystych i. Dla $l = 2q+1$, mamy $C(i,j,k) \leftarrow C(i,j,k)+C(i+2,j,k)$, gdy drugi najmniej znaczący bit w rozwinięciu i jest 0, oraz $C(i,j,k) \leftarrow C(i,j,k)+C(i-2,j,k)$, w przeciwnym przypadku. Zauważmy, że dla $l = 2q$, rejestr $C(2,j,k)$ otrzymuje wartość $C(2,j,k)+C(3,j,k)$, a zatem po dwóch iteracjach rejestr $C(0,j,k)$ zawiera sumę $C(0,j,k)+C(1,j,k)+C(2,j,k)+C(3,j,k)$, czyli liczba składników w $C(0,j,k)$ podwajana jest w każdej iteracji. W rezultacie, po q iteracjach rejestr $C(0,j,k)$ zawiera sumę $\sum_{i=0}^{n-1} C(i,j,k) = c_{jk}$.

Powyższy algorytm dla maszyny SIMD-CC z n^3 procesorami ma złożoność $O(q)$, a więc $O(\log n)$. Praca [4] zawiera szereg innych algorytmów mnożenia macierzy dla maszyn SIMD-CC, a wśród nich algorytm dla n^2 procesorów o złożoności $O(n)$ oraz dla $n^2 m$ procesorów o złożoności $O(n/m+\log m)$, gdzie m spełnia $1 \le m \le n$. Oba algorytmy są optymalne, w drugim przypadku należy przyjąć $m = n/\log n$. Podobna sytuacja jest także dla maszyn SIMC-PS (por. [4]), odpowiednie algorytmy są jednak bardziej skomplikowane. Zamiast nich, sieć PS zastosujemy w algorytmie sortowania (patrz § 4.2.3).

3.3. Algorytm naprzemiennego sortowania

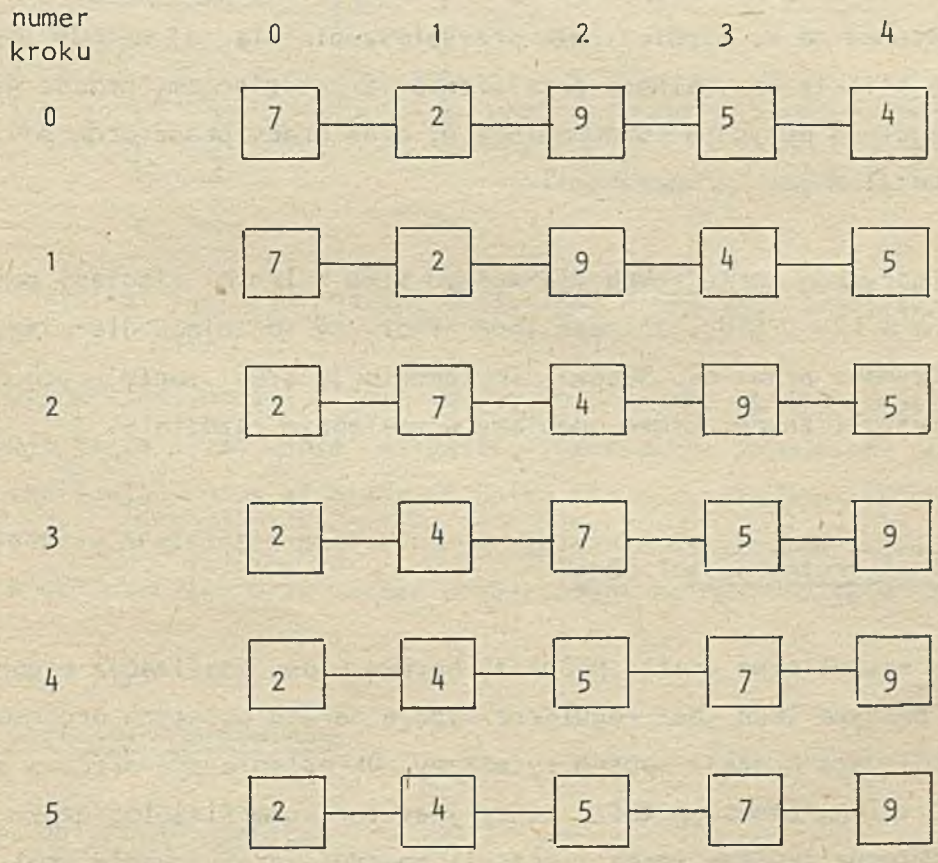
Dla posortowania n liczb, struktura połączeń między procesorami w maszynie SIMD może być nawet liniowa, taką maszynę oznaczać będziemy w skrócie przez SIMD-L (*linear*). Działanie maszyny SIMD-L zilustrujemy na przykładzie uwspółbieżnienia algorytmu sekwencyjnego, którego czas działania wynosi $O(n^2)$. Rozpatrzmy *metodę bąbelkową* (*bubble sort*), w której porównywane są i, w razie potrzeby, zamieniane miejscami elementy sąsiednie. W pierwszej iteracji algorytmu, element największy umieszczany jest na końcu ciągu, następnie drugi największy itd. Liczba porównań w tej metodzie wynosi co najwyżej $\sum_{i=1}^{n-2} (n-i)$, a więc jest rzędu n^2. Odmianą metody bąbelkowej jest metoda *naprzemiennego sortowania* (*odd-even transportation sort*) składająca się z n przeplatających się nawzajem faz nieparzystych i parzystych, z których każda wykonuje $n/2$ porównań. W każdej z faz, element aktywny x_i porównywany jest ze swoim prawym sąsiadem x_{i+1} i jeśli $x_i > x_{i+1}$, to elementy te zamieniane są

miejscami. W fazie nieparzystej, aktywne są elementy o wskaźnikach nieparzystych, a w fazie parzystej - elementy o wskaźnikach parzystych. Przyjmijmy, że element x_i znajduje się w i-tym procesorze maszyny SIMD-L w rejestrze o tej samej nazwie. Procesor aktywny porównuje zawartość swojego rejestru x z zawartością rejestru x w następnym procesorze, i w razie konieczności zamienia miejscami zawartości. Poniższy algorytm wymaga n procesorów, a jego czas działania wynosi $O(n)$.

Algorytm naprzemiennego sortowania

```
for l:=1 to n do
  if x_i>x_{i+1} then EXCHANGE(x_i,x_{i+1})   (i_0 = l_0)
{i_0(l_0) oznacza najmniej znaczący bit w rozwinięciu binarnym i(l)}
```

Rysunek 7 ilustruje działanie algorytmu naprzemiennego sortowania na przykładzie ciągu pięcioelementowego.



Rys. 7. Ilustracja działania algorytmu naprzemiennego na maszynie SIMD-L.

Podobnie, można uwspółbieżnić algorytm sortujący, który wykorzystuje drzewo binarne jako strukturę połączeń między elementami i wynikami porównań.

Jako rezultat otrzymujemy inny równoległy algorytm sortujący n liczb w czasie $O(n)$ przy użyciu $O(n)$ procesorów. Szczegółową realizację tego algorytmu pozostawiamy Czytelnikowi.

Metoda naprzemiennego sortowania może być uogólniona na przypadek k liniowo połączonych procesorów, z których każdy może przechowywać w swojej pamięci m liczb. Załóżmy, że $n = km$ i niech każdy z procesorów ma w swojej pamięci m liczb. Na początku, każdy procesor sortuje swój podciąg jedną z klasycznych metod sekwencyjnych. W następnych k krokach, uaktywaniane są naprzemiennie parzyste i nieparzyste procesory. Procesor aktywny otrzymuje podciąg z procesora następnego, scala oba podciągi w jeden ciąg uporządkowany, dzieli otrzymany ciąg na dwie równe części i drugą z nich odsyła do procesora następnego. Złożoność całego algorytmu jest $O(m \log m)+O(km)$, gdzie pierwszy składnik jest oszacowaniem czasu współbieżnego posortowania podciągów w procesorach, a drugi - złożonością kroku iteracyjnego. Oszacowanie złożoności można wyrazić także w postaci $O(n \log n)/k+O(n)$, a zatem dla n dużych w stosunku do k, współczynnik przyśpieszenia dla tej metody jest bliski k. To bliskie optymalnego prześpieszenie zawdzięczamy przede wszystkim temu, że dla n dużych w porównaniu z k, czas pracy procesorów dominuje czas komunikacji między procesorami.

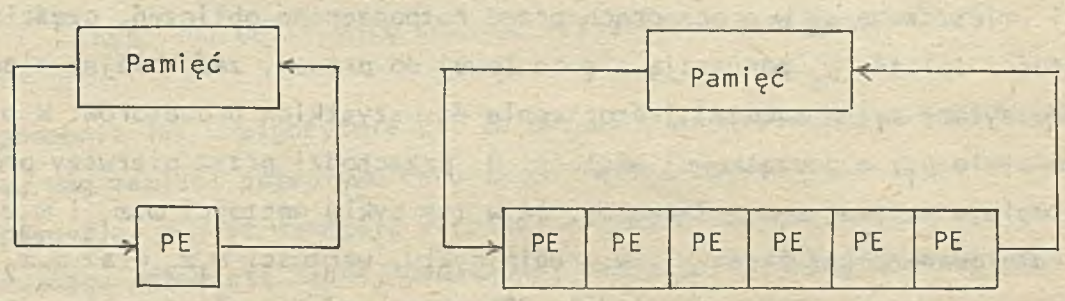
Opis algorytmów sortujących dla maszyn SIMD z innymi sieciami połączeń znaleźć można w [2] i [18]. Jeszcze inne algorytmy sortujące dla maszyn SIMD mogą być otrzymane przez odpowiednią implementację sieci sortujących oraz algorytmów systolicznych, które omawiamy w następnym rozdziale.

4. Maszyny i algorytmy systoliczne

Maszyny systoliczne uważać można za hardware'owe realizacje algorytmów (obliczeń). Maszyna taka jest regularną siecią bardzo prostych procesorów (PE), w której dane krążą w sposób rytmiczny. Określenie *systoliczna* pochodzi od angielskiego słowa *systole*, które jest terminem fizjologicznym oznaczającym rytmiczne skurcze serca i arterii powodujące pulsowanie krwi w ciele. Pierwsze maszyny systoliczne, będące realizacjami operacji macierzowych w technologii VLSI, zostały opracowane pod koniec lat 70-tych przez H.T. Kunga z Uniwersytetu Carnegie-Mellon. Komercyjne maszyny systoliczne pojawiły się na początku lat 80-tych.

Procesory w maszynach systolicznych wykonują najczęściej jedynie bardzo proste operacje. Informacje między procesorami przepływają w sposób potokowy, zaś komunikacja ze światem zewnętrznym odbywa się tylko za pomocą procesorów, które należą do brzegu całego układu. Zatem, jedynie brzegowe procesory mogą być portami I/O w czysto-systolicznych układach. Maszyny systoliczne, w których dostępne są dla świata zewnętrznego także niektóre nie-brzegowe procesory nazywać będziemy pseudo-systolicznymi.

Podstawowym zastosowaniem maszyn systolicznych są obliczenia, w których liczba operacji zdecydowanie przewyższa liczbę danych, zaś dane wielokrotnie występują w operacjach tego samego typu. Maszyny systoliczne zawdzięczają więc swoją dużą efektywność wielokrotnemu wykorzystaniu danych wprowadzonych do układu. Rysunek 8 przedstawia schemat maszyny systolicznej.



Rys. 8. Ewolucja maszyny klasycznej w maszynę systoliczną.

Ocena efektywności algorytmów systolicznych, oprócz czasu obliczeń, obejmuje także oszacowanie całkowitej powierzchni zajmowanej przez układ VLSI realizujący dany algorytm. Z kolei, oszacowanie powierzchni powinno uwzględniać przestrzeń potrzebną do zapisania wszystkich liczb występujących w obliczeniach oraz obszar przeznaczony na realizację procesorów (PE).

4.1. Obliczenia macierzowe

4.1.1. Obliczenia splotów

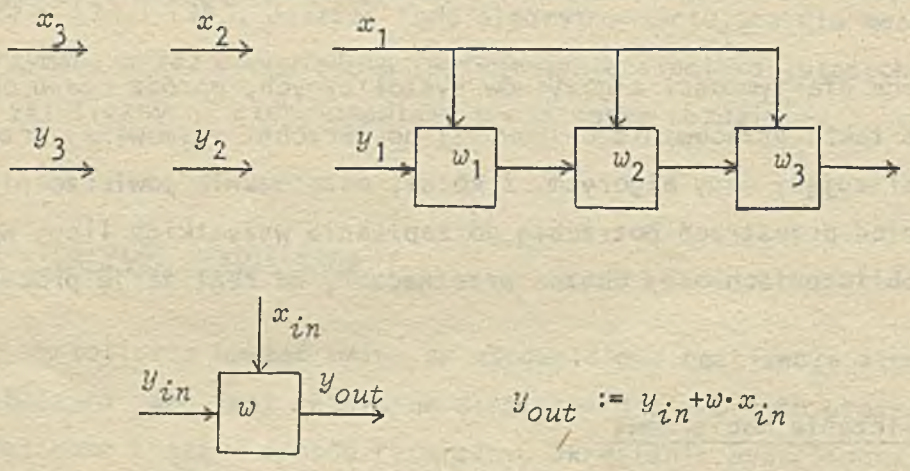
Klasycznym przykładem algorytmu systolicznego jest metoda obliczania *splotów* (*convolution*). Załóżmy, że dany jest ciąg wag w_1, w_2,..., w_k. Dla ciągu danych wejściowych x_1, x_2, x_3,..., x_n należy wyznaczyć ciąg y_1, y_2, ..., y_{n+1-k} zdefiniowany następująco

$$y_i = w_1 x_i + w_2 x_{i+1} + \ldots + w_k x_{i+k-1}.$$

Obliczanie splotów pojawia się w bardzo wielu standardowych podprogramach numerycznych takich jak: dopasowywanie wzorca, korelacja, interpolacja, obliczanie wartości wielomianów, dyskretna transformacja Fouriera, mnożenie i dzielenie wielomianów. Najpierw, opiszemy trzy algorytmy semi-systoliczne (C1-C3), a następnie - dwa algorytmy czysto-systoliczne (C4-C5). Dla większej przejrzystości prezentacji, opisami algorytmów są przykładowe sieci systoliczne dla $k = 3$.

C1. Rozsyłanie danych - ruch wyników - wagi w miejscu.

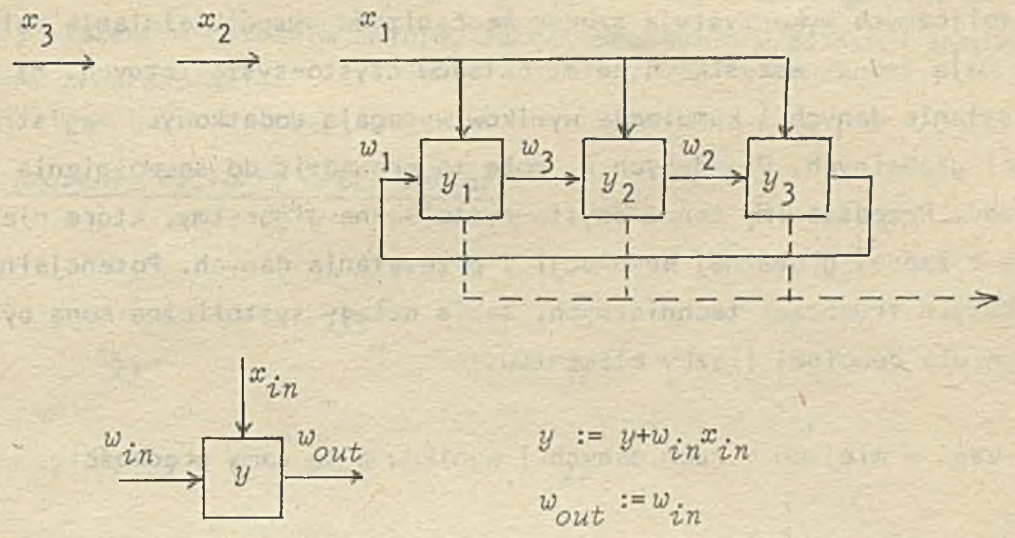
W algorytmie systolicznym, którego schemat przedstawiono na Rys. 9, wagi umieszczane są w procesorach przed rozpoczęciem obliczeń, częściowe wartości splotów y_i poruszają się od lewej do prawej, zaś kolejne elementy x_i rozsyłane są (*broadcast*) jednocześnie do wszystkich procesorów. W pierwszym cyklu y_1, o początkowej wartości 0, przechodzi przez pierwszy procesor i kumuluje wartość $w_1 x_1$. Zauważmy, że w tym cyklu wartości $w_2 x_1$ i $w_3 x_1$ nie są kumulowane przez żaden y_i. W drugim cyklu, wartości $w_1 x_2$ oraz $w_2 x_2$ kumulowane są przez y_2 i y_1 odpowiednio. Począwszy od trzeciego cyklu, poprawne wartości y_1, y_2,... zaczynają pojawiać się na wyjściu trzeciego procesora.



Rys. 9. Obliczanie splotów - systoliczny algorytm C1.

C2. Rozsyłanie danych - ruch wag - wyniki w miejscu.

W algorytmie przedstawionym na Rys. 10, wartości y_i kumulowane są w procesorach, przez które przepływa cyklicznie ciąg wag. Dane x_i rozsyłane

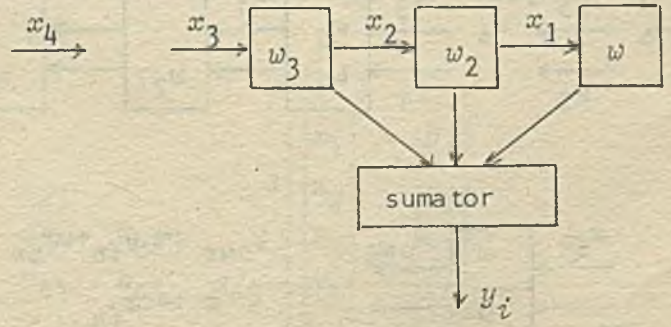


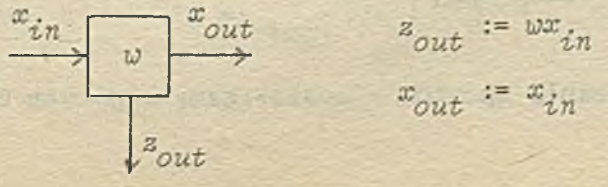
Rys. 10. Obliczanie splotów - systoliczny algorytm C2.

są podobnie jak w algorytmie C1. Za przewagę algorytmu C2 nad C1 można uznać opływ wag zamiast przepływu rezultatów, które często liczone są ze zwiększoną precyzją, a więc wymagają większej liczby bitów niż wagi. Rozwiązanie C1 nie wymaga natomiast żadnej dodatkowej magistrali (lub globalnej sieci) do zbierania wyników.

C3. Kumulacja wyników - ruch danych - wagi w miejscu.

Rysunek 11 przedstawia algorytm, w którym wszystkie składniki każdego splotu liczone są równocześnie, a następnie wysyłane są do sumatora, gdzie następuje ich kumulacja. Jeśli k jest stosunkowo duże, to sumator może mieć postać drzewa procesorów, w którym kolejne kumulacje realizowane są potokowo.



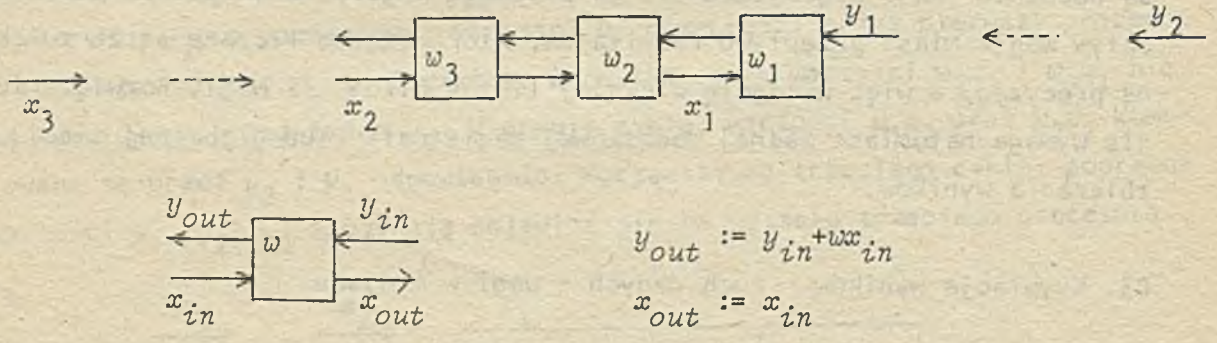


Rys. 11. Obliczanie splotów - systoliczny algorytm C3.

Przedstawione trzy metody liczenia splotów za pomocą algorytmów semi-systolicznych wykorzystują szereg mechanizmów uwspółbieżniania obliczeń, nie mają jednak wszystkich zalet układów czysto-systolicznych. Na przykład, rozsyłanie danych i kumulacja wyników wymagają dodatkowych magistrali lub sieci globalnych. Dla dużych k, może to prowadzić do spowolnienia zegara układu. Przedstawimy teraz czysto-systoliczne algorytmy, które nie korzystają z żadnej globalnej kumulacji i przesyłania danych. Potencjalnie, bez większych trudności technicznych, takie układy systoliczne mogą być rozszerzane dla dowolnej liczby elementów.

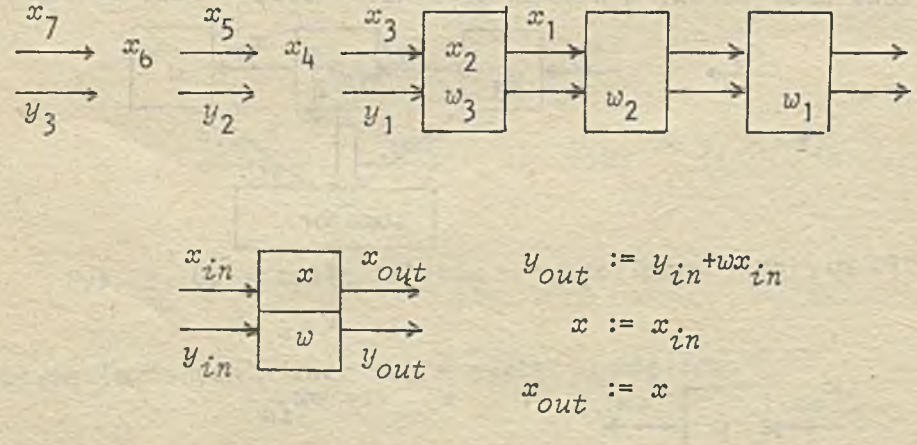
C4. Wagi w miejscu - ruch danych i wyników z tą samą prędkością.

W algorytmie zilustrowanym na Rys. 12, wagi umieszczone są w procesorach przed rozpoczęciem obliczeń, a dane i wyniki przepływają przez procesory w odstępach dwóch cykli między kolejnymi elementami. Słabą stroną tego rozwiązania jest tylko w połowie wykorzystywana moc (liczba) procesorów (średnio 50 %).



Rys. 12. Obliczanie splotów - systoliczny algorytm C4.

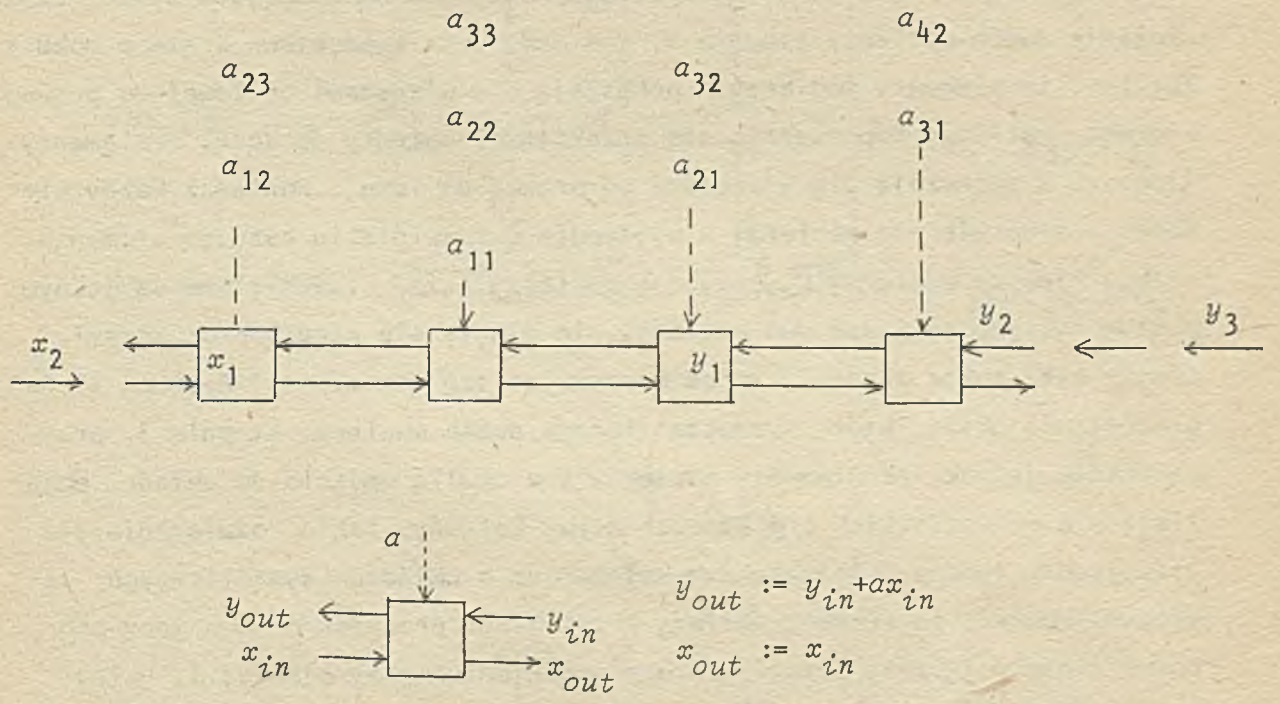
C5. Wagi w miejscu - ruch danych i wyników.



Rys. 13. Obliczanie splotów - systoliczny algorytm C5.

Algorytm przedstawiony na Rys. 13, nie ma tej wady algorytmu C4, o której piszemy w ostatnim zdaniu, dzięki przesyłaniu danych i wyników z różnymi prędkościami.

4.1.2. Mnożenie macierzy przez wektor



Rys. 14. Algorytm systoliczny dla wyznaczania iloczynu macierzy wstęgowej przez wektor (Kung i Leiserson, 1978).

Rysunek 14 przedstawia schemat algorytmu systolicznego z liniowym układem procesorów, który wyznacza elementy iloczynu Y wstęgowej macierzy A przez wektor X, tj.

$$\begin{vmatrix} a_{11} & a_{12} & & & & \\ a_{21} & a_{22} & a_{23} & & & \\ a_{31} & a_{32} & a_{33} & a_{34} & & \\ & a_{42} & a_{43} & a_{44} & a_{45} & \\ & 0 & \cdots & & & \end{vmatrix} \begin{vmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \vdots \end{vmatrix} = \begin{vmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ \vdots \end{vmatrix}$$

Przypuśćmy, że macierz A ma wstęgę szerokości w (powyżej, $w = 4$). Wtedy, jak łatwo sprawdzić, układ składający się z w procesorów oblicza n kolejnych elementów wektora Y w czasie $2n+w$.

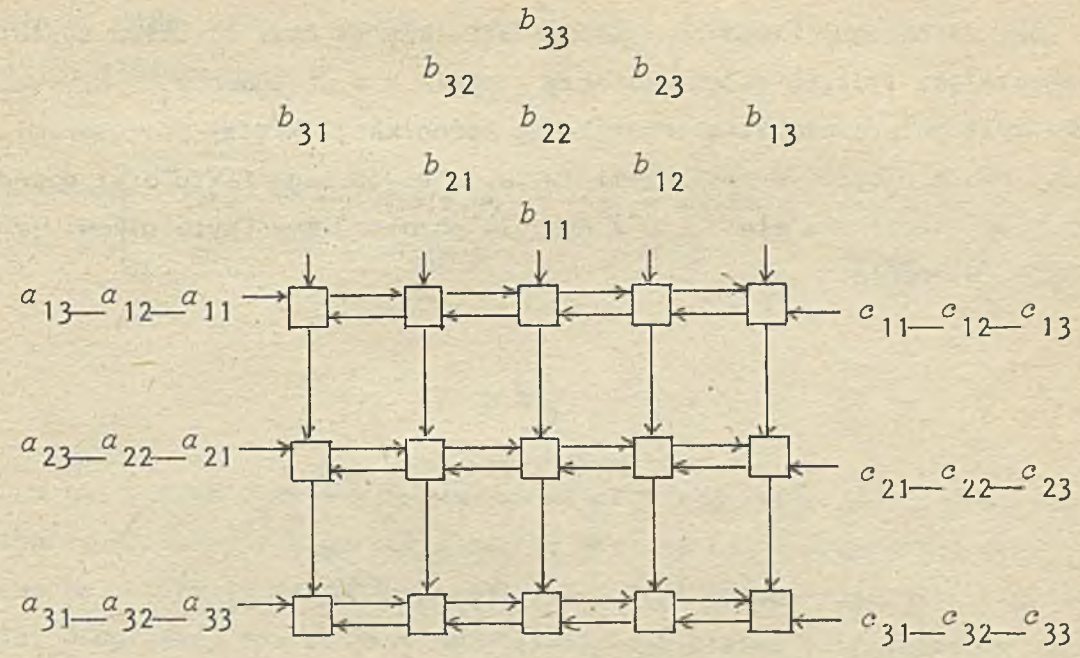
4.1.3. Mnożenie dwóch macierzy

Najbardziej naturalną organizacją procesorów wykonujących systoliczne mnożenie dwóch macierzy stopnia n, $C = A \cdot B$, jest kwadratowa krata o boku n. Załóżmy, że elementy macierzy A poruszają się wierszami od lewej do prawej, elementy macierzy B poruszają się przekątnymi od góry do dołu, a elementy iloczynu C poruszają się wierszami od prawej do lewej. Ponieważ każdy element i-tego wiersza macierzy A występuje w rozwinięciu każdego elementu i-tego wiersza macierzy C, elementy macierzy A i C przedzielone są jednym pustym cyklem, by żadne dwa elementy nie minęły się między procesorami. (Pusty cykl można zastąpić zerem otrzymując ten sam efekt.) Rysunek 15 (a) przedstawia układ, który wyznacza iloczyn dwóch macierzy stopnia 3, przy założeniu jednak, że elementy macierzy B z chwilą wejścia do układu, pojawiają się we wszystkich procesorach danej kolumny. Takie rozwiązanie nie jest jednak zgodne z podstawowym założeniem o układach systolicznych. Zakładamy bowiem, że element wysłany przez jeden procesor osiąga inny procesor z nim połączony bezpośrednio z opóźnieniem przynajmniej 1. Układ z Rys. 15 (a) może być jednak łatwo sprowadzony do układu czysto-systolicznego przez opóźnienie każdego elementu macierzy B o 1 między kolejnymi procesorami i cofnięcie i-tych wierszy macierzy A i C o i-1 cykli. Tak zmodyfikowany układ jest już czysto-systoliczny, patrz Rys. 15 (b).

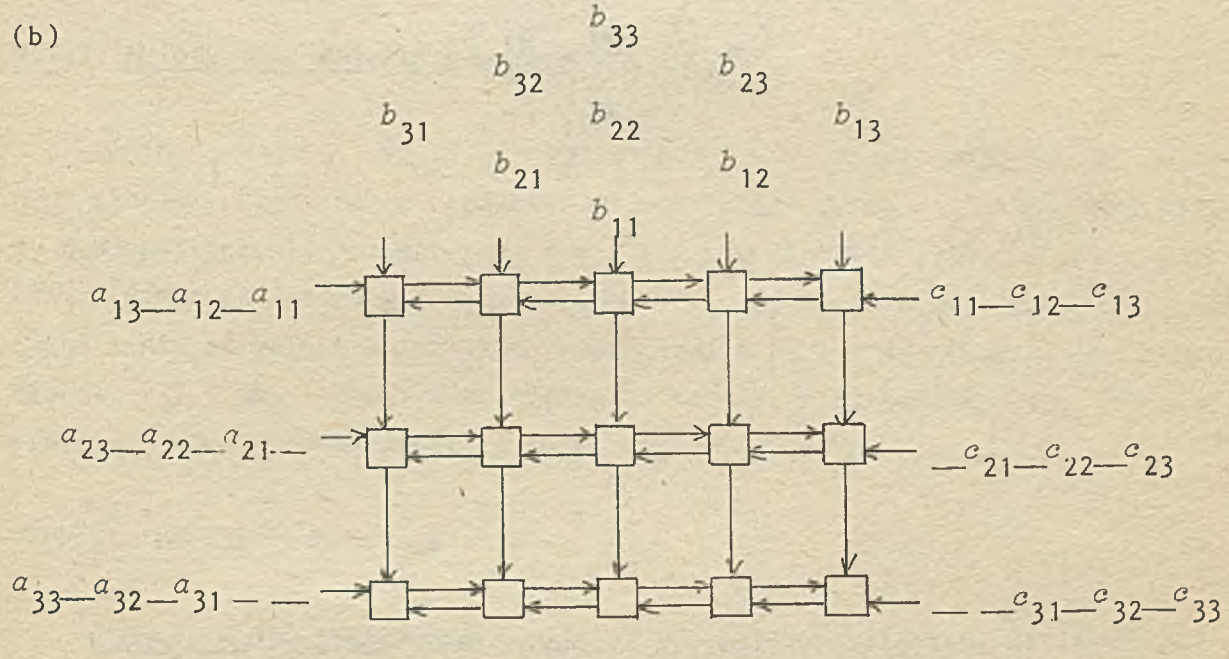
Przedstawiony układ systoliczny mnoży dwie macierze stopnia n w czasie $O(n)$ i zajmuje powierzchnię $O(n^2 \log m)$, gdzie n^2 jest rzędem obszaru zajmowanego przez n^2 procesorów, zaś $\log m$ jest oszacowaniem górnym rozmiaru (tj. liczby bitów) liczb występujących w obliczeniach.

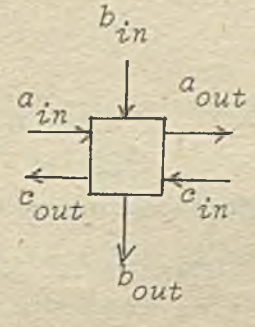
Powyższy układ systoliczny może być wykorzystany do mnożenia macierzy boolowskich stopnia n na powierzchni $O(n^2)$ i w tym samym czasie $O(n)$. Ten algorytm z kolei może być zastosowany do wyznaczania macierzy przechodniego domknięcia binarnej relacji (lub grafu) w czasie $O(n \log n)$. Szereg innych układów systolicznych dla problemów macierzowych i grafowych przedstawionych jest w książce Ullmana [24], Rozdz. 5.

(a)



(b)





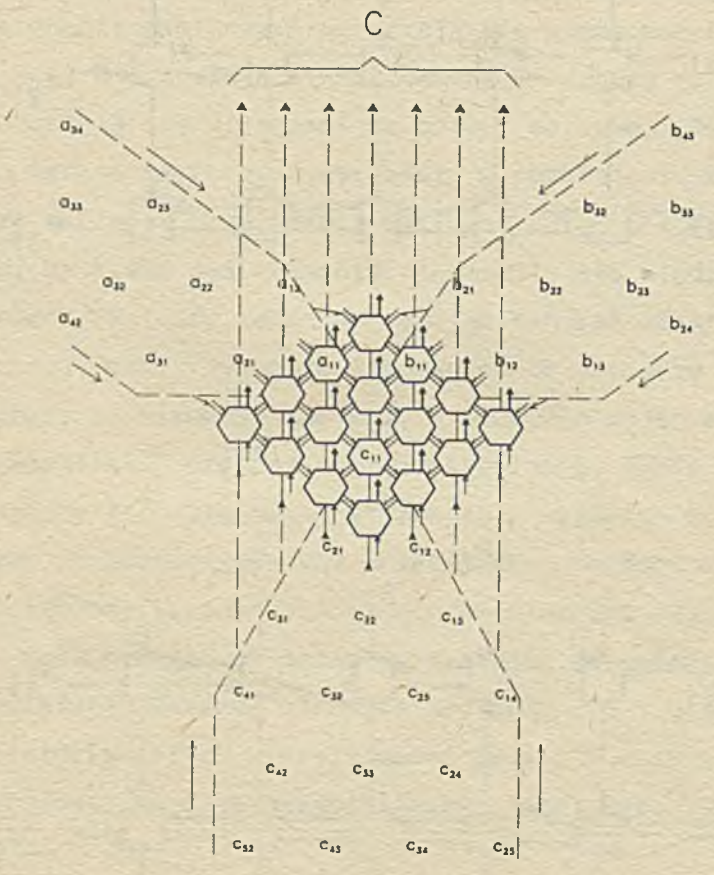
$a_{out} := a_{in}$

$b_{out} := b_{in}$

$c_{out} := c_{in} + a_{in} b_{in}$

Rys. 15. Systoliczny algorytmy mnożenia dwóch macierzy.

Na zakończenie tego punktu przedstawiamy na Rys. 16 układ systoliczny wyznaczający iloczyn dwóch macierzy stopnia n o wstęgach w_1 i w_2, który wykorzystuje sześciokątną organizację komunikacji między procesorami. Czas działania algorytmu wynosi $3n+\min\{w_1,w_2\}$. Dowód tego faktu oraz dowód poprawności działania algorytmu z Rys. 16 pozostawiamy Czytelnikowi jako proste ćwiczenie.

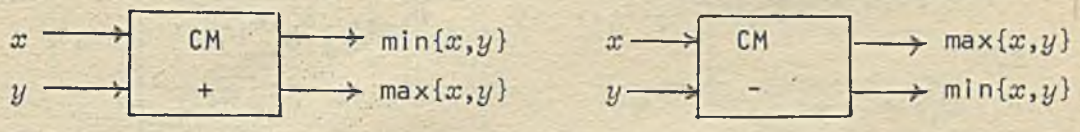


Rys. 16. Systoliczny algorytm mnożenia dwóch macierzy wstęgowych.

4.2. Sieci sortujące

Systoliczne algorytmy porządkowania ciągu liczb nazywane są powszechnie *sieciami sortującymi* (*sorting network*). Podstawowym elementem sieci sortującej jest *moduł porównania*, który dla dwóch danych liczb na wejściu, na wyjściu podaje minimum i maximum, patrz Rys. 17. Ze względu na swoją

prostotę, moduły takie mogą być produkowane w technologii VLSI i tworzyć bardziej złożone układy.



Rys. 17. Moduły porównań.

W następnych punktach opisujemy dwa algorytmy sortujące pochodzące od Batchera. Pierwszy z algorytmów oparty jest na innym algorytmie tego samego autora, który łączy dwa uporządkowane ciągi metodą naprzemiennego scalania. Drugi zaś, oparty jest na własnościach ciągów bitonicznych. Obie sortujące sieci Batchera są najszybszymi znanymi równoległymi algorytmami sortującymi.

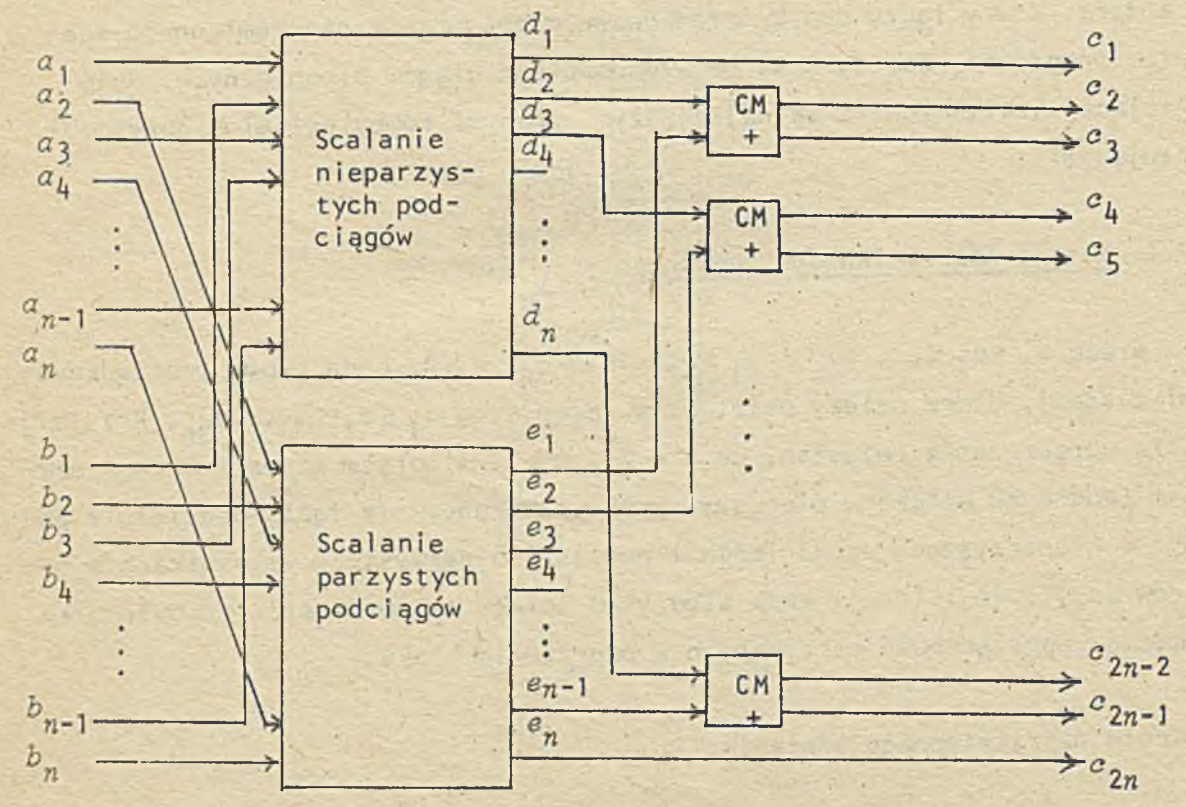
4.2.1. Metoda naprzemiennego scalania

Niech $a_1 \le a_2 \le \ldots \le a_n$ i $b_1 \le b_2 \le \ldots \le b_n$, $n \ge 2$, będą dwoma uporządkowanymi ciągami, które należy połączyć w jeden ciąg $c_1 \le c_2 \le \ldots \le c_{2n}$. Przyjmijmy dla uproszczenia rozważań, że $n = 2^k$, $k \ge 1$. Algorytm *naprzemiennego scalania* (*odd-even merge*) w pierwszym kroku rekurencyjnie łączy oddzielnie podciągi o nieparzystych wskaźnikach i podciągi o parzystych wskaźnikach z obu ciągów $\{a_j\}$ i $\{b_j\}$. Drugi krok algorytmu polega na porównaniu odpowiednich elementów uporządkowań otrzymanych w poprzednim kroku.

Algorytm naprzemiennego scalania.

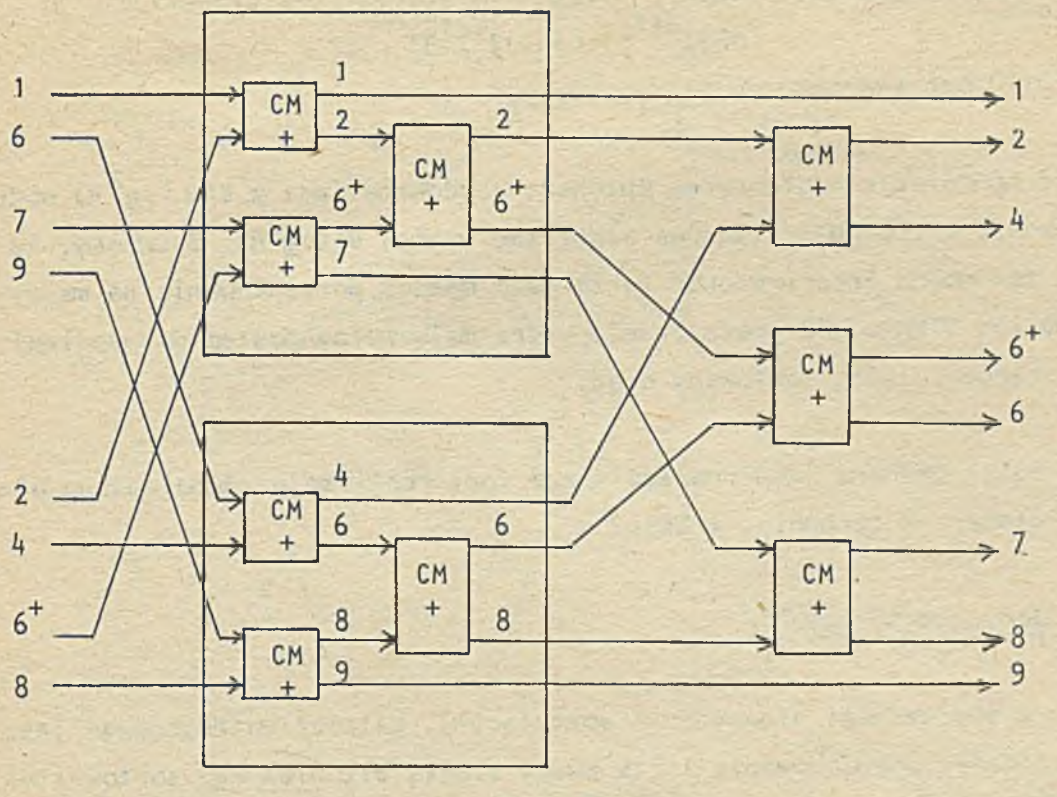
```
procedure ODD_EVEN_MERGE(n;a_1,a_2,...,a_n;b_1,b_2,...,b_n;c_1,c_2,...,c_2n);
begin
  if n=2 then begin
    if a_1≤b_1 then begin d_1:=a_1; d_2:=b_1 end
    else begin d_1:=b_1; d_2:=a_1 end;
    if a_2≤b_2 then begin e_1:=a_2; e_2:=b_2 end
    else begin e_1:=b_2; e_2:=a_2 end end
  else begin
```

```
    ODD_EVEN_MERGE(n/2;a_1,a_3,...,a_{n-1};b_1,b_3,...,b_{n-1};d_1,d_2,...,d_n);
    ODD_EVEN_MERGE(n/2;a_2,a_4,...,a_n;b_2,b_4,...,b_n;e_1,e_2,...,e_n)
  end;
  c_1:=d_1; c_{2n}:=e_n;
  for i:=1 to n-1 do begin
    c_{2i}:=min{d_{i+1},e_i};
    c_{2i+1}:=max{d_{i+1},e_i}
  end
end
```



Rys. 18. Schemat działania algorytmu naprzemiennego scalania.

Rysunek 18 przedstawia ogólny schemat powyższego algorytmu, a na Rys. 19 pokazano przykładową sieć dla $n = 4$. Zauważmy, że na ogół jest to algorytm niestabilny, gdyż element $a_2 = 6$ występuje na początku (w danych) przed $b_3 = 6^+$, a w uporządkowanym ciągu ich kolejność jest odwrotna.



Rys. 19. Przykład sortowania metodą naprzemiennego scalania.

Łatwo policzyć rozwiązując odpowiednie równanie rekurencyjne, że sieć naprzemiennego scalania zbudowana jest z $O(n \log n)$ modułów porównań, zaś czas potrzebny do scalenia dwóch uporządkowanych ciągów w jeden wynosi $O(\log n)$.

4.2.2. Równoległy algorytm Batchera

Wykorzystamy teraz powyższy algorytm do skonstruowania sieci porządkującej dowolny ciąg liczb. Przyjmijmy, że należy uporządkować ciąg a_1, a_2, ..., a_n, gdzie dla uproszczenia mamy $n = 2^l$, $l > 0$. Równoległy algorytm Batchera wykonuje l kroków dla $i = 0, 1, \ldots, l-1$, a w każdym z nich stosując algorytm naprzemiennego scalania łączy wszystkie 2^{k-i-1} pary uporządkowanych podciągów długości 2^i.

Algorytm Batchera

```
for i:=0 to l-1 do
  for 1≤j≤2^(l-i-1) do simultaneously
```

$$\text{ODD\_EVEN\_MERGE}(2^i; a[j2^{i+1}+1:j2^{i+1}+2^i]; a[j2^{i+1}+2^i+1:(j+1)2^{i+1}];$$
$$a[j2^{i+1}+1:(j+1)2^{i+1}])$$

{$a[g:h]$ oznacza ciąg $a_g, a_{g+1}, \ldots, a_h$}

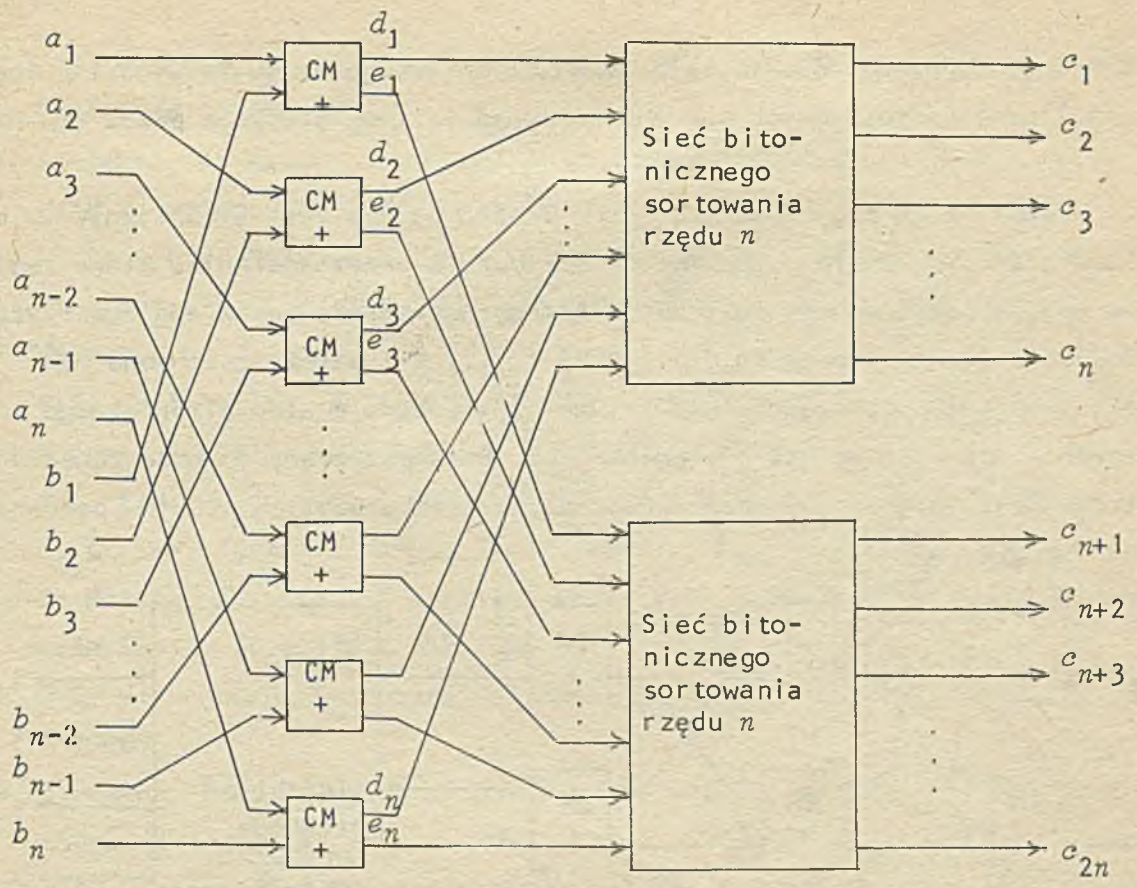
Sieć równoległego algorytmu Butchera zbudowana jest z $O(n \log^2 n)$ modułów porównań, a złożoność czasowa algorytmu wynosi $O(\log^2 n)$. Zauważmy, że algorytm ten można interpretować także jako metodę porządkowania na maszynach MIMD lub SIMD z $n/2$ procesorami, które mają wolny dostęp do wspólnej pamięci przechowującej sortowany ciąg.

W książce Ullmana [24] znaleźć można inną realizację równoległego algorytmu Batchera w technologii VLSI.

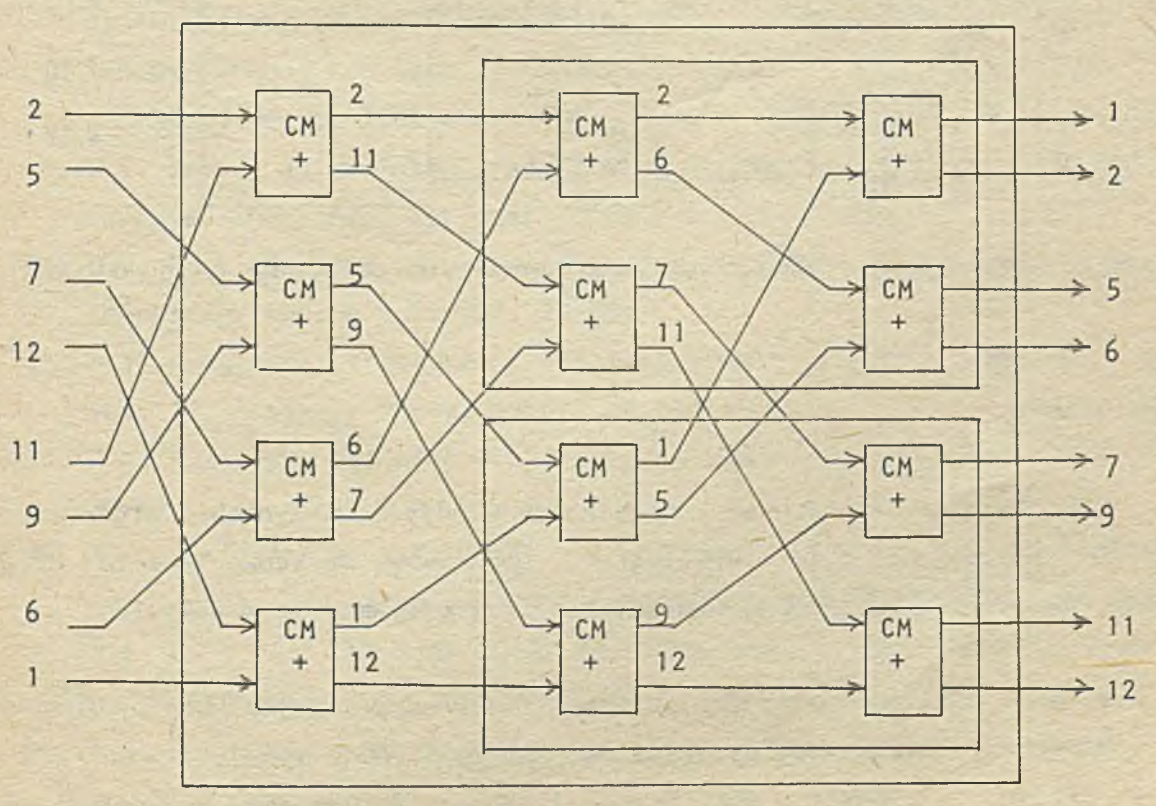
4.2.3. Bitoniczne sortowanie

Wraz z poprzednimi algorytmami sortującymi, Batcher zaproponował jeszcze inny schemat porządkowania liczb zwany siecią bitonicznego sortowania. Ciąg liczb $a_1, a_2, \ldots, a_n$ jest *bitoniczny* (*bitonic*), jeśli składa się z co najwyżej dwóch podciągów monotonicznych, tj. jeśli istnieje j, $1 \le j \le n$, takie, że $a_1 \le a_2 \le \ldots \le a_j \ge a_{j+1} \ge \ldots \ge a_n$. Sieć bitonicznego sortowania porządkuje ciąg bitoniczny. Zauważmy, że dla $j = 1$ lub $j = n$, ciąg bitoniczny jest monotoniczny. Każdy podciąg ciągu bitonicznego jest również bitoniczny, a także jeśli $a_1 \le a_2 \le \ldots \le a_n$ i $b_1 \le b_2 \le \ldots \le b_m$, to ciąg $(a_1, a_2, \ldots, a_n, b_m, \ldots, b_1)$ jest bitoniczny. Wynika stąd, że sieć bitonicznego sortowania może być stosowana do scalania dwóch uporządkowanych ciągów.

Załóżmy, że $a_1, a_2, \ldots, a_{2n}$ jest ciągiem bitonicznym i przyjmijmy, że $n = 2^k$, $k \ge 0$. Batcher udowodnił, że ciągi $d_1, d_2, \ldots, d_n$ i $e_1, e_2, \ldots, e_n$, gdzie $d_i = \min\{a_i, a_{n+i}\}$ oraz $e_i = \max\{a_i, a_{n+i}\}$ dla $i = 1, 2, \ldots, n$, są także bitoniczne i co więcej $\max\{d_i\} \le \min\{e_i\}$. Ta własność ciągów bitonicznych może być rekurencyjnie zastosowana do ciągów $d_1, d_2, \ldots, d_n$ oraz $e_1, e_2, \ldots, e_n$. W rezultacie, otrzymujemy algorytm bitonicznego sortowania, którego sieć przedstawiona jest schematycznie na Rys. 20, zaś Rys. 21 ilustruje realizację tej sieci dla $2n = 8$, w której schemat połączeń między modułami porównań w dwóch sąsiednich kolumnach wyznaczony jest przez wartość funkcji SHUFFLE. Przyjmijmy, że wyjścia z/i wejścia do modułów oznaczone są liczbami od 0 do $2n-1$. Wtedy wyjście i połączone jest z wejściem SHUFFLE(i) = $= (i_{m-1}, i_{m-2}, \ldots, i_0, i_m)$, gdzie $i_m, i_{m-1}, \ldots, i_0$ jest binarną reprezenacją i.



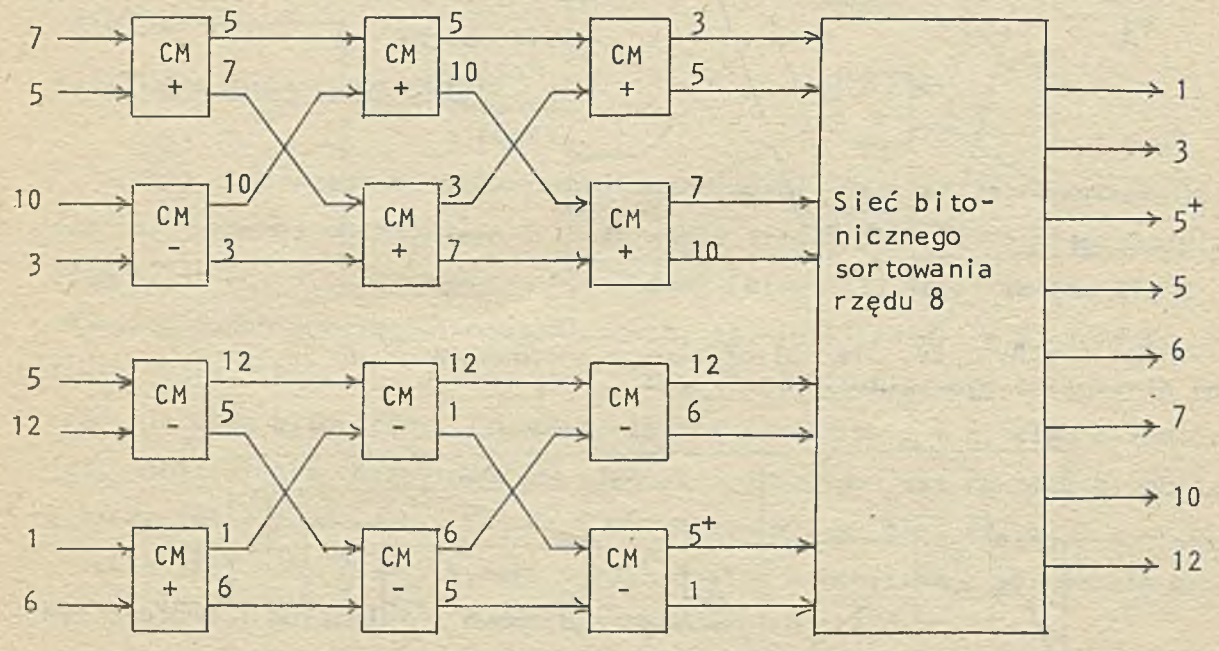
Rys. 20. Sieć bitonicznego sortowania rzędu $2n$.



Rys. 21. Przykładowa sieć bitonicznego sortowania rzędu 8.

Czas potrzebny do posortowania ciągu bitonicznego rzędu $2n$ wynosi $O(\log n)$, zaś liczba zastosowanych modułów porównań jest ograniczona przez $O(n \log n)$.

Jeśli ciąg $a_1, a_2, \ldots, a_n$, $n = 2^k$ $(k \geq 0)$ nie jest bitoniczny, to można wtedy zastosować klasyczną metodę sortowania przez scalanie, które realizowane jest przy pomocy sieci bitonicznego sortowania odpowiedniego rzędu. Dokładniej, w i-tym kroku dla $i = 0, 1, \ldots, k-1$ należy zastosować 2^{k-i-1} sieci sortujących rzędu 2^{i+1}. Rysunek 22 ilustruje sieć bitonicznego sortowania dowolnego ciągu 8-elementowego. Czas potrzebny do posortowania n liczb jest ograniczony przez $O(\log^2 n)$, a liczba użytych modułów porównań wynosi $O(n \log^2 n)$.



Rys. 22. Przykładowa sieć bitonicznego sortowania dowolnego ciągu długości 8.

BIBLIOGRAFIA

Czytelnik zainteresowany pogłębieniem wiedzy o maszynach i algorytmach równoległych znaleźć może w bibliografii podstawowe opracowania z tej dziedziny. Nie wszystkie pozycje wymienione niżej cytowane są w tekście.

1. A.V. Aho, J.E. Hopcroft, J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974 (PWN, 1983).
2. D. Bitton, D.J. DeWitt, D.K. Hsiao, J. Menon, A taxonomy of parallel sorting, *Computing Surveys* 16 (1984), 287-318.

3. E.G. Coffman, Jr., P.J. Denning, *Operating Systems Theory*, Prentica-Hall, Englewood Cliffs, 1973.
4. E. Dekel, D. Nassimi, S. Sahni, Parallel matrix and graph algorithms, *SIAM J. Comput.* 10 (1981), 657-675.
5. N. Deo, C.Y. Pang, R.E. Lord, Two parallel algorithms for shortest path problems, TR CS-80-059, Computes Science Department, WSU, Pullman (WA), 1980.
6. P.E. Enslow, Jr., Multiprocessor organization - A survey, *Computing Surveys* 9 (1977), 103-129.
7. P.E. Enslow, Jr., *Systemy Cyfrowe Wieloprocesorowe*, WNT, Warszawa, 1978.
8. J. Gruska, *Systolic Computations*, Springer-Verlag, Berlin, 1985.
9. D. Heller, A survey of parallel algorithms in numerical linear algebra, *SIAM Review* 20 (1978), 740-777.
10. D.S. Hirschberg, Fast parallel sorting algorithms, *Comm. ACM* 21 (1978), 657-666.
11. K. Hwang, F.A. Brigger, *Computer Architecture and Parallel Processing*, McGraw-Hill, 1984.
12. D.S. Johnson, The NP-completeness column: an ongoing guide, *J. Algorithms* 4 (1983), 189-203 and 286-300, 5 (1984), 595-609.
13. G.A.P. Kindervater, J.K. Lenstra, Parallel algorithms in combinatorial optimization: an annotated bibliography, in: O'hEigeartaigh, J.K. Lenstra, A.H.G. Rinnooy Kan (eds.), *Combinatorial Optimization: Annotated Bibliographies*, Wiely, 1985.
14. V.I. Kotov, J. Mikloško (red.), *Algorithms, Software and Hardware of Parallel Computers*, Springer-Verlag, Berlin, 1984.
15. D.J. Kuck, A survey of parallel machine organization and programming, *Computing Surveys* 9 (1977), 29-59.
16. H.T. Kung, The structure of parallel algorithms, in: *Advances in Computers*, vol. 19, Academic Press, 1980, str. 65-112.
17. H.T. Kung, Why systolic architecture, *Computer Magazine*, Styczeń 1982, 37-46.
18. S. Laksmivaraham, S.K. Dhall, L.L. Miller, Parallel sorting algorithms, in: *Advances in Computers*, vol. 23, Academic Press, 1984, str. 295-354.
19. R.E. Lord, J.S. Kowalik, S.P. Kumar, Solving linear algebraic equations on an MIMD computer, *J. ACM* 30 (1983), 103-117.
20. M.J. Quinn. N. Deo, Parallel graph algorithms, *Computing Surveys* 16 (1984), No. 3.

21. C.V. Ramamoorthy , H.F. Li, Pipeline architecture, *Computing Surveys* 9 (1977), 61-102.

22. J.H. Reif, Depth-first search is inherently sequential, *Information Processing Letters* 20 (1985), 229-234.

23. M. Satyanarayanan, *Multiprocessors*, Prentice-Hall, Englewood Cliffs, 1980.

24. J.D. Ullman, *Computational Aspects of VLSI*, Computer Science Press, 1984.

25. U. Vishkin, Synchronous parallel computation - a survey, maszynopis, 1983.

Druk. Wyd. "ALFA". Nakł. 200. Zam.3185/85