ZESZYTY NAUKOWE POLITECHNIKI ŚLĄSKIEJ 1997
Seria: INFORMATYKA z. 33 Nr kol. 1377

Adam GROS, Krzysztof PIERZCHAŁA
Politechnika Śląska, Instytut Informatyki

WSADOWE PRZETWARZANIE RÓWNOLEGŁE W SIECI KOMPUTEROWEJ NA PRZYKŁADZIE PROGRAMU MAKE[1]

Streszczenie. W artykule zaprezentowano wersję programu make umożliwiającą równoległą realizację zadań w sieci komputerowej. Pokazano podstawy wykorzystywania programu make oraz dokonanano porównania różnych jego wersji. Przeanalizowano możliwość realizacji programu make przy użyciu różnych systemów pozwalających rozpraszać obliczenia w sieci komputerowej (RPC, Linda, PVM). Zaprezentowano niektóre szczegóły implementacyjne programu oraz dokonano oceny efektywności jego działania.

NETWORK PARALLEL BATCH PROCESSING BY EXAMPLE OF MAKE PROGRAMME

Summary. The make programme version that makes possible parallel tasking in computer networks is presented in the paper. The basic rules of using make and comparison of its different versions are presented. Possibilities of realization make programme using different systems for distributing processing in computer networks (RPC, Linda, PVM) are analyzed. Some implementation details and effectivity evaluation are discussed.

[1]Pracę wykonano w ramach grantu KBN nr 3 P406 011 04.

1. Wprowadzenie

Program make jest jednym z narzędzi spotykanych w systemie operacyjnym UNIX. Jest to narzędzie programistyczne, służące do zautomatyzowania procesu generowania i zarządzania modułami pośrednimi i ich bibliotekami oraz programami wykonywalnymi projektu programowego. Make umożliwia utrzymanie spójności projektu programowego oraz eliminuje niepotrzebne rekompilacje modułów, na które nie wpłynęły zmiany kodu źródłowego.

Program make wykonuje operacje na podstawie pliku wejściowego zawierającego opis projektu programowego. Jednorazowe napisanie pliku z opisem projektu zwalnia programistę z obowiązku "ręcznego" uruchamiania procesów kompilacji i łączenia modułów po dokonaniu zmian w kodzie źródłowym tworzonej aplikacji. Proces powstawania aplikacji z wykorzystaniem programu make przybiera następującą postać:

myślenie — modyfikacja plików źródłowych projektu — wydanie polecenia "**make**" — testowanie ...

Wprawdzie zamiast kolejno wydawać za każdym razem wszystkie polecenia, można by zapisać je w skrypcie powłoki i wykonywać tylko ten skrypt. Jeśli jednak w celu aktualizacji modułów projektu zastosuje się program make, to wykonywane będą tylko te polecenia, które są konieczne. W celu stwierdzenia, które polecenia muszą być wykonane, a które nie, make używa metody sprawdzania zależności plików źródłowych i wynikowych. Polega ona na porównaniu czasów ostatniej modyfikacji tych plików. Jeśli plik wynikowy nie istnieje lub choć jeden z jego plików źródłowych jest "młodszy" od niego, to plik wynikowy jest aktualizowany – wykonywane są przypisane mu polecenia. Struktura zależności między plikami może mieć postać hierarchiczną – plik wynikowy może być plikiem źródłowym dla innego pliku.

Najważniejszą rzeczą, która wchodzi w skład opisu projektu, jest specyfikacja zależności między modułami oraz polecenia, które służą do przekształcania jednych modułów w drugie. Zapis w pliku wejściowym jest następujący:

```
cel :   [zależność] ...
        [polecenie]
        ...
```

i znaczy to, że plik wynikowy (zwany także celem) o nazwie podanej przed dwukropkiem, powstaje z plików źródłowych (zależności) o nazwach zapisanych po znaku ":" (dwukropka), w wyniku wykonania zapisanych w kolejnych liniach poleceń.

Jeśli nie podano w wywołaniu programu inaczej, make rozpoczyna przetwarzanie od pierwszego celu występującego w pliku wejściowym (cel ten zwany jest celem głównym). Cel uaktualniany jest w następujący sposób:

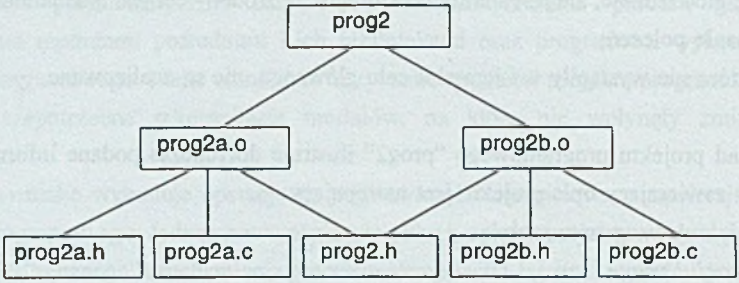
- uaktualniane są jego źródła występujące po prawej stronie znaku dwukropka; kolejne źródła stają się celami – sprawdzanie zależności odbywa się rekurencyjnie;
- po przetworzeniu wszystkich źródeł – jeśli któreś z nich było uaktualniane lub plik cel nie istnieje albo istnieje, ale jest starszy od któregoś ze źródeł – cel jest uaktualniany poprzez wykonanie poleceń;
- cele, które nie wystąpiły w hierarchii celu głównego, nie są analizowane.

Przykład projektu programowego "prog2" ilustruje dotychczas podane informacje. Plik wejściowy zawierający opis projektu jest następujący:

```
# plik wejściowy programu make
# dla aplikacji "prog2"
prog2 : prog2a.o prog2b.o
    cc prog2a.o prog2b.o -o prog2
prog2a.o : prog2a.c prog2a.h prog2.h
    cc -c prog2a.c -o prog2a.o
prog2b.o : prog2b.c prog2b.h prog2.h
    cc -c prog2b.c -o prog2b.o
```

Celem głównym jest cel "prog2", a hierarchię jego zależności przedstawia rys. 1. W tym przypadku make zaczyna od celu "prog2", który ma zależności. Zanim sprawdzi, czy cel ten należy uaktualnić, sprawdza kolejno, czy aktualizacji wymagają jego pliki źródłowe "prog2a.o" oraz "prog2b.o". Te z kolei mają swoje zależności, więc i ich weryfikacja zostaje odłożona. Zależności celów "prog2a.o" i "prog2b.o", to pliki źródłowe (także nagłówkowe) w języku C (pliki z rozszerzeniem ".c" oraz ".h"). Opisów celów o takich nazwach nie ma w pliku wejściowym, więc make przyjmuje, że są to istniejące pliki w systemie plików i zapamiętuje czas ich ostatniej modyfikacji (jest to uproszczony opis zachowania programu – pomija on bowiem wykorzystanie dopasowywania reguł domyślnych). Następnie wraca do odłożonego celu (z rozszerzeniem ".o") i analizuje jego stan. Jeżeli plik o takiej nazwie nie istnieje albo istnieje, ale przynajmniej jeden z jego wcześniej analizowanych plików źródłowych jest od niego nowszy, wtedy cel ten zostaje uznany za nieaktualny. Aby cel zaktualizować, wykonywane są odpowiadające mu polecenia (w tym przypadku kompilacja plików źródłowych w języku C do plików ".o"), po czym zapamiętany zostaje czas aktualizacji. Jeżeli analizowany cel nie został uznany za nieaktualny, nie są wykonywane żadne operacje. Po sprawdzeniu obu celów (nazwy plików z rozszerzeniami ".o") program wraca do analizy celu głównego. Zostanie on uznany za nieaktualny, jeżeli była uaktualniana któraś z jego zależności. Jeśli zależności nie wymagały aktualizacji, wtedy o statusie celu głównego i konieczności jego utworzenia decyduje zasada podobna, jak dla celów "prog2a.o"

i "prog2b.o": jeżeli plik o takiej nazwie ("prog2") nie istnieje albo istnieje, ale przynajmniej jeden z jego wcześniej analizowanych plików źródłowych jest od niego nowszy, to cel ten zostaje uznany za nieaktualny.



Rys. 1. Hierarchia zależności celów w projekcie "prog2"
Fig. 1. Target dependency hierarhy in "prog2" project

Powyżej przedstawiona została podstawowa cecha programu make, mianowicie umiejętność uaktualniania celu głównego na drodze selektywnego wykonywania tylko tych działań, które są niezbędne. Zaprezentowany sposób opisu projektu w pliku wejściowym jest identyczny dla wielu wersji programu make, o ile nie korzysta się z dodatkowych możliwości charakterystycznych dla konkretnego programu. Make należy do tych narzędzi systemu operacyjnego UNIX, które są ciągle ulepszane. Powstają ciągle jego nowsze wersje, które są często niezgodne z poprzednimi. Nie istnieje standard, który określałby, jakie funkcje powinien mieć make i w jaki sposób byłyby one dostępne (zapisane w pliku wejściowym). Dlatego istnieje wiele wersji tego programu. Niektóre są rozprowadzane wraz z systemami operacyjnymi. Istnieją też wersje programu make dostępne dla większej liczby odmian systemu UNIX oraz innych systemów operacyjnych. W poszczególnych wersjach różny może być zbiór udostępnianych funkcji oraz format pliku wejściowego. W celu wyboru funkcji udostępnianych przez równoległą wersję programu make (identyfikowaną dalej przez **makep**) porównano następujące istniejące wersje tego programu:

- **dmake** wersja 3.8 – dostępny w sieci Internet, autorem jest Denis Vadura;
- **GNU make** wersja 3.71 i 3.72 – dostępny w sieci Internet, powstały w ramach projektu GNU;
- **SunOS make** – dostarczany wraz z systemem operacyjnym SunOS;
- **BSD pmake** – rozprowadzany z systemem operacyjnym LINUX;
- **Borland make** wersja 3.6 i 3.7 – dostępny w pakiecie Borland C++ 3.1 oraz Borland C++ 4.02;
- **gymake** wersja 1.2 i 1.4 – dostępny w sieci Internet, autorem jest Greg Yachuk;
- **pdmake** – dostępny w sieci Internet, autorem jest Paul Homchick.

Zestawienie funkcji udostępnianych przez poszczególne wersje oraz wybranych do implementacji w programie makep prezentuje tabela 1. Znak "+" oznacza dostępność danej funkcji, a znak "–" jej brak. W przypadku reguł domyślnych litera "S" oznacza możliwość korzystania z reguł opartych na porównaniu przyrostka (ang. suffix rule), a "P" z reguł opartych na porównaniu wzorca (ang. pattern rule). Dodatkowo w tabeli zawarto informację, w jakim systemie operacyjnym działa dany program.

Tabela 1

Zestawienie funkcji dostępnych w poszczególnych wersjach programu make

Funkcje	dmake	gnu make	SunOS make	BSD pmake	Borland make	gymake	pdmake	makep
Definiowanie zmiennych	+	+	+	+	+	+	+	+
Włączanie plików	+	+	+	+	+	–	–	+
Przetwarzanie warunkowe	+	+	–	+	+	–	–	–
Kontrola modyfikacji poleceń	+	+	+	–	+	–	–	–
Reguły domyślne	S, P	S, P	S, P	S	S	S	S	S, P
Specjalna obsługa bibliotek	+	+	+	–	–	–	–	–
Dostępny w systemach operacyjnych	różne	UNIX	SunOS	UNIX	DOS	UNIX, DOS	DOS	UNIX

2. Analiza możliwości realizacji programu make wykonującego operacje w sposób równoległy z uwzględnieniem środowiska sieci komputerowej

2.1. Przyjęte założenia i cele

Program make, jak powiedziano wcześniej, znajduje głównie zastosowanie w procesie generowania z kodu źródłowego kodu wynikowego projektów programowych. Jest on odpowiedzialny za przeprowadzenie kompilacji i łączenia plików wchodzących w skład projektu. W przypadku dużych projektów programowych, gdzie liczba plików sięga dziesiątek

lub nawet setek, a ich objętość mierzy się w dziesiątkach megabajtów, może trwać to bardzo długo.

Celem pracy było stworzenie programu, który pełniłby funkcję analogiczną do wyszczególnionych wcześniej programów make. Przy czym powinien on wykonywać niektóre operacje w sposób równoległy, wykorzystując w tym celu sumaryczną moc komputerów połączonych siecią. W wyniku wykorzystania tego programu uzyska się skrócenie czasu potrzebnego na przekształcenie plików źródłowych projektu programowego w ich postać wynikową. Potencjalnie przyspieszenie, jakiego można oczekiwać, jest równe stosunkowi sumarycznej mocy komputerów połączonych siecią do mocy jednego komputera.

Przyjęto, że równoległa wersja programu make powinna spełniać następujące założenia:

- program ma być funkcjonalnym odpowiednikiem "klasycznych" programów typu make;
- sposób użytkowania tworzonego programu powinien być maksymalnie zbliżony do już istniejących programów tego typu – zgodny interfejs użytkownika;
- zdolność akceptacji opisów projektów (plików wejściowych) stworzonych dla innych wersji programu make;
- równoległe wykonywanie operacji w dowolnej wersji systemu operacyjnego UNIX, bez potrzeby wykorzystania rozproszonych systemów operacyjnych (takich jak np. Chorus, Mach czy QNX);
- możliwość wykorzystania programu zarówno w sieci komputerów, jak i na pojedynczym komputerze.

2.2. Wybór operacji realizowanych równolegle

W skład projektu programowego wchodzą pliki źródłowe, z których po skompilowaniu i połączeniu otrzymuje się działający program. Pomiędzy plikami źródłowymi aplikacji, plikami pośrednimi powstającymi podczas przekształcania kodu źródłowego w program wykonywalny oraz wynikowymi plikami wykonywalnymi występują pewne zależności. Przykładowo, aby uzyskać program wykonywalny, należy połączyć pliki obiektowe (".o"). Pliki obiektowe uzyskuje się poprzez kompilację odpowiednich plików zawierających kod źródłowy w pewnym języku programowania. Czasami przed kompilacją plików źródłowych należy je jeszcze przetworzyć pewnym preprocesorem (np. "m4"), albo też pliki te tworzone są przy pomocy generatorów, jak np. "lex", "yacc". Niektóre z tych operacji muszą być wykonywane w odpowiedniej kolejności, przykładowo łączenie plików jest możliwe dopiero po kompilacji. Natomiast inne mogłyby być wykonywane równocześnie, np. kompilacja niezależnych plików źródłowych. Informacje o zależnościach pomiędzy plikami oraz sposób przekształcania jednych w drugie zawarty jest w pliku wejściowym programu make zawierającym opis danego projektu.

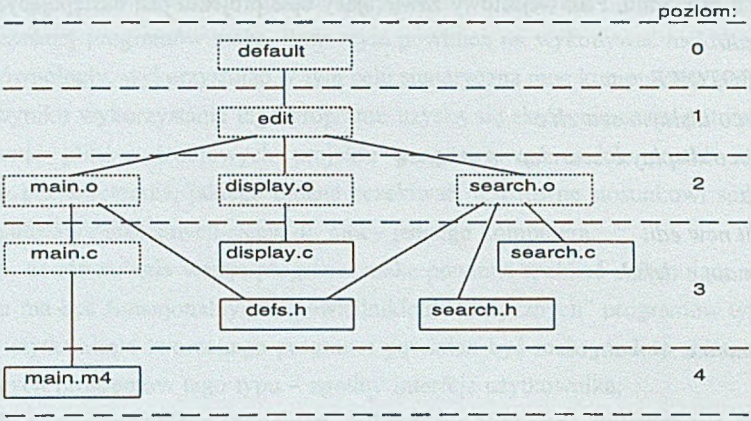
Spróbujmy teraz znaleźć operacje, które mogą być realizowane równocześnie. Dokonane zostanie to na podstawie konkretnego przykładu, jednak nie ogranicza to w niczym ogólności rozwiązania problemu. Plik wejściowy zawierający opis projektu jest następujący:

```
default : edit
    echo GOTOWE
edit : main.o dislay.o search.o
    cc main.o display.o search.o -o edit.new
    rm edit
    mv edit.new edit
main.o : main.c defs.h

    cc -c main.c -o main.o

main.c : main.m4
    rm main.c
    m4 main.m4 > main.c
display.o : display.c
    cc -c display.c -o display.o
search.o : search.c search.h defs.h

    cc -c search.c -o search.o
```

Jest to projekt prostego edytora tekstów. Wykonywalnym plikiem wynikowym jest "edit", a plikami istniejącymi w systemie plików są: "main.m4", "defs.h", "display.c", "search.h" i "search.c".

Pojedynczą operacją wykonywaną przez program make w celu aktualizacji projektu jest jedna linia polecenia służąca do uaktualnienia danego celu. Aktualizacja niektórych celów wymaga wprawdzie wykonania kilku poleceń, jednak ich kolejność w ogólnym przypadku nie jest dowolna (patrz opis celów: "edit" i "main.c"). Tak więc aktualizacja jednego celu musi być wykonana w sposób sekwencyjny, według kolejności wystąpień poleceń w opisie danego celu.

Zależności pomiędzy poszczególnymi plikami projektu przedstawia rys. 2. Tak przedstawiona struktura zależności między celami wygląda jak drzewo. Możemy określić korzeń, którym jest cel główny, oraz wyróżnić gałęzie, w których zachowana jest opisana poniżej hierarchia zależności. Nie jest to jednak "prawdziwe" drzewo, gdyż niektóre z "gałęzi" mogą się ze sobą łączyć. Struktura ta więc jest bliższa grafowi zorientowanemu.



Rys. 2. Zależności pomiędzy plikami projektu edytora
Fig. 2. Editor project files dependencies

Operacją, która może podlegać zrównoleglenia, jest uaktualnienie jednego celu, czyli wykonanie jego wszystkich poleceń. Odpowiada to węzłowi grafu z rys. 2. Problemem jest optymalny wybór operacji do równoległego wykonania przy zachowaniu wymaganych zależności.

Na rys. 2 wszystkie pliki zakwalifikowano do pewnego poziomu. Numer poziomu odzwierciedla odległość plików na tym poziomie od celu głównego. Jak widać, zależności między celami mają taki charakter, że cel będący na poziomie n może zależeć bezpośrednio jedynie od celów na poziomie $n+1$, ale pośrednio także od celów z poziomów o numerach większych od n. Cele, które są na jednym poziomie, nie zależą od siebie oraz cel o wyższym numerze poziomu nie zależy od celu będącego na poziomie o numerze niższym. W ten sposób dochodzimy do konkluzji, że operacjami, które na pewno mogą być wykonywane równocześnie, są aktualizacje celów będących na jednakowym poziomie, przy czym zaczynamy od poziomu o największym numerze. Jeśliby jednak dobrze się przyjrzeć rysunkowi 2, to widać, że algorytm oparty tylko na tej regule nie jest optymalny. Można przecież równocześnie dokonać aktualizacji niektórych celów z poziomu 2 i 3, np. "display.o", "search.o", "main.c".

2.3. Możliwość realizacji programu w poszczególnych systemach przetwarzania rozproszonego i równoległego

W tej części zostaną zarysowane możliwości, a właściwie podjęte próby stworzenia programu makep w kilku systemach programowych umożliwiających przetwarzanie rozproszone i równoległe z wykorzystaniem sieci komputerów.

Na początek zapoznajmy się z algorytmem stosowanym przez "klasyczne" programy make, które wykonują aktualizację poszczególnych celów w sposób sekwencyjny. Poniżej przedstawiony algorytm jest formalnym zapisem tego co zostało powiedziane wcześniej, w związku z wyjaśnianiem zasady działania programu make. Algorytm przedstawiony jest w umownym języku programowania wysokiego poziomu o składni zbliżonej do języka PASCAL:

```
{ aktualizuje 'cel' po uprzedniej aktualizacji jego zależności, zwraca czas modyfikacji celu }
aktualizacja_celu_1(cel)
begin
  if 'cel'_ ma zależności then
     for zależności
        aktualizacja_celu_1(zależność);
  endfor
     if (not istnieje_plik(cel))
        or (istnieje_plik(cel) and jedna z zależności jest nowsza) then
           wykonaj_polecenia_celu(cel);
        return czas aktualizacji celu;
     else
        return czas modyfikacji pliku 'cel'_;
     endif
  else
     if istnieje_plik(cel) then
        return czas modyfikacji pliku 'cel'_;
     else
        if 'cel'_ ma polecenia then
           wykonaj_polecenia_celu(cel);
           return czas aktualizacji celu;
        else
           Błąd( "Brak pliku i poleceń do utworzenia
                     celu: ", cel);
        endif
     endif
```

```
    endif
end
```

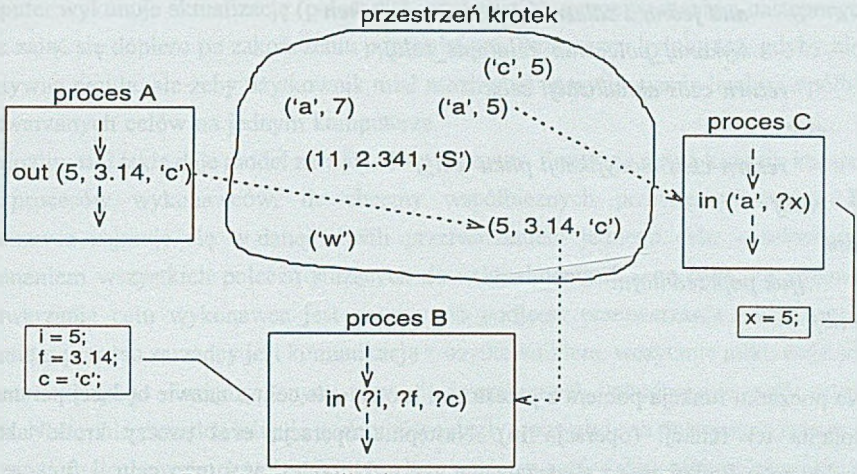
Funkcja "aktualizacja_celu_1" jest wywoływana przez program make jedynie dla celu głównego. W ciele tej funkcji następuje rekurencyjne wywołanie jej samej dla celów będących zależnościami przetwarzanego celu. Kolejne zależności są aktualizowane sekwencyjnie, jedna po drugiej. Po uaktualnieniu wszystkich zależności podejmowana jest decyzja, czy przetwarzany cel jest aktualny, czy też należy wykonać przypisane mu polecenia. Ostatecznie funkcja zwraca czas aktualizacji celu lub czas ostatniej modyfikacji pliku o takiej nazwie, w zależności od tego, czy cel wymagał uaktualnienia, czy też nie. Jeśli przetwarzany cel nie miał zależności, to polecenia uaktualniające są wykonywane w zależności od tego, czy plik o takiej nazwie istnieje. W ten sposób rzeczywista aktualizacja celów następuje "od dołu do góry" z zachowaniem hierarchii celów (rys. 2).

Powyżej przedstawiony algorytm sekwencyjny można by łatwo zrównoleglić, gdyby zastąpić sekwencyjne wywoływanie funkcji aktualizującej zależności równoczesną aktualizacją zależności. Algorytm taki w używanej powyżej notacji, poszerzonej o możliwość wyrażenia współbieżnego wykonywania funkcji, jest następujący:

```
{ aktualizuje 'cel'_ po uprzedniej równoległej aktualizacji jego zależności, zwraca czas
modyfikacji celu }
aktualizacja_celu_2(cel)
begin
    if 'cel'_ ma zależności then
        do parallel for zależności
            aktualizacja_celu_2(zależność);
        endpar
        if (not istnieje_plik(cel))
            or (istnieje_plik(cel) and jedna z zależności jest nowsza) then
                wykonaj_polecenia_celu(cel);
            return czas aktualizacji celu;
        else
            return czas modyfikacji pliku 'cel'_;
        endif
    else
        ...{jak poprzednio}...
    endif
end
```

Powyższy algorytm można by zaimplementować wykorzystując do tego system Linda. W systemie tym procesy odpowiadają funkcjom i mogą być tworzone dynamicznie, a tego

właśnie wymaga ten algorytm. System Linda symuluje w sieci komputerów komputer wieloprocesorowy ze wspólną pamięcią. Podstawowym pojęciem w modelu Lindy jest krotka (ang. tuple). Krotka to ciąg danych (o określonej lub nieokreślonej wartości), z których każda ma określony typ. Ciąg typów poszczególnych elementów krotki tworzy jej sygnaturę. W Lindzie wszystkie procesy realizują się w środowisku zwanym przestrzenią krotek (ang. tuple space). Przestrzeń krotek to nic innego jak współdzielona pamięć asocjacyjna. Z przestrzeni tej proces pobiera dane w formie krotek (operacją in) i do tej przestrzeni wysyła wyniki, także w formie krotek (operacja **out**). Operacja **in** pobiera krotkę (usuwa ją z przestrzeni krotek) o podanej sygnaturze i wartościach podanych pól (rys. 3). Do tworzenia procesów współbieżnych (krotek aktywnych) służy operacja **eval**.



Rys. 3. Komunikacja poprzez przestrzeń krotek
Fig. 3. Communication using tupple space

W programie opartym na systemie Linda można wyróżnić dwie części. Pierwsza, to program główny odpowiedzialny za wczytanie pliku wejściowego, przetworzenie go (między innymi rozwinięcie zmiennych), a następnie umieszczenie w postaci krotek opisów poszczególnych celów w przestrzeni krotek i wywołanie funkcji odpowiadającej zamieszczonej powyżej funkcji "aktualizacja_celu_2" dla celu głównego. Proces odpowiadający tej części jest tylko jeden.

Drugą częścią jest właśnie funkcja odpowiadająca przedstawionemu powyżej algorytmowi. Jej postać przystosowana dla Lindy jest następująca:

{ aktualizuje 'cel'_ po uprzedniej równoległej aktualizacji jego zależności, zwraca czas modyfikacji celu, wersja dla Lindy }

***function** aktualizacja_celu_2L(cel : string) : int*

begin

```
    in(cel,?opis_celu);
    if 'cel'_ ma zależności then
        for zależności
            eval(zależność,aktualizacja_celu_2L(zależność));
        endfor
        for zależności
            in (zależność, ?czas);
        endfor
        if (not istnieje_plik(cel))
            or (istnieje_plik(cel)
                and jedna z zależności jest nowsza) then
                wykonaj_polecenia_celu(opis_celu);
            return czas aktualizacji celu;
        else
            return czas modyfikacji pliku 'cel'_;
        endif
    else
        ...{jak poprzednio}...
    endif
end
```

Na początku funkcja pobiera z przestrzeni krotek opis celu o nazwie będącej parametrem wywołania tej funkcji (operacja **in**). Następnie operacją **eval** tworzy krotki aktywne odpowiadające funkcjom przetwarzającym wszystkie zależności tego celu – funkcje te są wykonywane współbieżnie. Po zakończeniu funkcji każda z tych krotek aktywnych zamieni się w krotkę bierną. Na wszystkie te krotki funkcja czeka wykonując operację **in** dla wszystkich zależności. Ciąg dalszy jest taki jak dla algorytmu sekwencyjnego.

Tak ogólnie zaprezentowany sposób implementacji równoległego programu make w Lindzie wygląda obiecująco, szczególnie prostota przejścia od znanego algorytmu sekwencyjnego do algorytmu równoległego. Jednak praktyczna realizacja napotyka problemy. Przykładowo, powyższy algorytm nie uwzględnia sytuacji, gdy jeden cel jest zależnością większej liczby celów (wtedy któraś z operacji **in** nigdy się nie skończy, gdyż krotkę, na którą czeka, wziął ktoś inny). Innym problemem jest przekazywanie przez przestrzeń krotek opisów celów, gdyż struktury o stałych polach tutaj nie wystarczą (zmienna ilość zależności i poleceń dla różnych celów).

Z jednej strony zaletą algorytmu jest to, że wykonuje maksymalną możliwą w danej chwili ilość operacji równolegle. Z drugiej jednak strony prowadzi to do niekontrolowanego wzrostu liczby procesów, co przy ograniczonej liczbie jednostek przetwarzających prowadzi do ich

nadmiernego obciążenia. Doświadczalnie stwierdzono, że uruchomienie równocześnie na przykład dwóch procesów kompilacji na jednym komputerze nie przynosi praktycznie żadnego zysku, a większa liczba takich procesów powoduje przeciążenie systemu. Podobny problem stanowi brak możliwości specyfikowania w Lindzie komputera, na którym ma być powoływany nowy proces. Implementacja programu makep w Lindzie została zaniechana głównie z tych ostatnich powodów.

W trakcie kolejnych prób implementacji programu make, wykonującego operacje równolegle, założono maksymalne wykorzystanie dostępnych komputerów bez ich przeciążania. Przyjęto, że byłoby optymalnie, gdyby makep używał wszystkich komputerów, wykonując na każdym z nich w danej chwili tylko jedną operację. Znaczy to, że jeden komputer wykonuje aktualizację (polecenia) jednego celu, a przetwarzaniem następnego celu może zająć się dopiero po zakończeniu poprzedniego. Pożyteczne byłoby też, gdyby nie była to sztywna reguła, ale żeby użytkownik miał możliwość specyfikowania liczby współbieżnie przetwarzanych celów na jednym komputerze.

Możliwości takie daje model zarządca i wykonawcy. Powołuje się na każdym komputerze tyle procesów wykonawców, ile chcemy współbieżnych przetwarzań celów. Każdy wykonawca zajmuje się w danej chwili przetwarzaniem jednego celu – sekwencyjnym wykonaniem wszystkich poleceń służących do uaktualnienia danego celu. Po zakończeniu przetwarzania celu wykonawca jest gotowy do podjęcia przetwarzania następnego celu. Zadaniem procesu zarządcy jest komunikacja z użytkownikiem, wczytanie pliku wejściowego i przetworzenie go – między innymi rozwinięcie zmiennych i zbudowanie grafu zależności celów. Następnie zarządca inicjuje system równoległy i powołuje na dostępnych komputerach odpowiednią liczbę wykonawców. Po czym korzystając z grafu zależności celów wybiera cele, które należy uaktualnić i wysyła ich polecenia do wolnych wykonawców. Decyzja o tym, czy dany cel jest aktualny, czy też należy wykonać jego polecenia, należy do procesu zarządcy. Wykonawcy otrzymują do przetwarzania tylko te cele, które należy uaktualnić. Po rozesłaniu pracy do wszystkich wykonawców zarządca czeka, aż któryś z nich skończy i ponownie korzystając z grafu zależności odnajduje cel do uaktualnienia, który wysyła do tego wykonawcy. Trwa to aż do momentu, gdy zarządca przeanalizuje aktualność celu głównego i jeśli potrzeba, przekaże do wykonania jego polecenia. Na zakończenie odwołuje wykonawców i wykonuje prace związane z zamknięciem systemu równoległego. W takim rozwiązaniu wykonawcy odpowiedzialni są jedynie za przyjęcie i sekwencyjne wykonanie przesłanych im poleceń. Cała "inteligencja" – optymalny wybór celów do równoczesnego przetwarzania – spoczywa na zarządcy. Opis takiej funkcji wyszukującej w grafie zależności cele do przetwarzania znajduje się w następnej części.

Zauważmy, że tak skonstruowany program ma charakter aplikacji rozproszonej, która sama decyduje o wykorzystaniu poszczególnych komputerów sieci do wykonania zadań. Możliwa

jest na przykład klasyfikacja dostępnych maszyn według ich mocy obliczeniowej i wysyłanie zadań w pierwszej kolejności do maszyn mocniejszych. Jeśli powołamy na każdym komputerze po jednym wykonawcy, to możemy mówić o "prawdziwie" równoległym przetwarzaniu celów – liczba jednostek przetwarzających jest równa liczbie zadań. Inaczej sytuacja wyglądała podczas realizacji programu w Lindzie, gdzie rozproszona realizacja programu wynikała z rozproszonego charakteru systemu Linda, a rozdział zadań na poszczególne komputery był przypadkowy.

Próbę implementacji takiego modelu podjęto przy wykorzystaniu mechanizmu RPC oraz systemu PVM.

W RPC rolę wykonawców pełnią procesy zwane w terminologii RPC serwerami. RPC nie dostarcza mechanizmu zdalnego uruchamiania serwerów, więc należy tego dokonać przy użyciu systemowych operacji udostępnianych przez UNIX. RPC umożliwia wykonanie zadania na innym komputerze poprzez zdalne wywołanie procedury. Odbywa się to synchronicznie – wywołanie zdalnej procedury powoduje wstrzymanie procesu wywołującego aż do wykonania tej procedury i zwrotu przez nią wyniku. W tym przypadku wynikiem jest to, czy wykonanie poleceń zakończyło się sukcesem, czy porażką (np. błędy w trakcie kompilacji plików źródłowych). Takie wstrzymanie procesu zarządcy na czas przetworzenia danego celu niweczy równoczesne przetwarzanie celów. Można wprawdzie do obsługi wywołania zdalnej procedury utworzyć proces potomny (funkcją systemową "*fork*"), który będzie wykonywany współbieżnie z procesem rodzica (zarządcy). Jednak powstanie wtedy poważny problem synchronizacji procesu potomnego i procesu rodzica (kiedy wywołać systemową funkcję "*wait*") oraz komunikacji – odebrania w odpowiednim momencie wyników przetwarzania celu od procesu potomnego. Trzeba stwierdzić, że implementacja w RPC modelu jeden zarządca (w terminologii RPC klient) i wiele równocześnie pracujących wykonawców (serwery) napotyka duże problemy. Problemy te można rozwiązać dopiero na niskim poziomie programowania systemowego. Poza tym korzystanie z RPC wymaga programowania bezpośrednio związanego z mechanizmami sieciowymi.

Zupełnie inaczej wygląda sprawa w przypadku skorzystania z systemu PVM. Uruchamianie nowych zadań oraz komunikacja między nimi jest dostępna na wysokim poziomie abstrakcji, przy czym nie traci się możliwości sterowania sposobem rozproszenia części składowych aplikacji. PVM tworzy z sieci komputerów wirtualny komputer wieloprocesorowy, w którym komunikacja między zadaniami oparta jest na przesyle komunikatów. PVM udostępnia funkcję powołującą nowe zadanie na dowolnym komputerze wchodzącym w skład komputera wirtualnego, przy czym możliwe jest wyspecyfikowanie jego nazwy, typu architektury lub pozostawienie wyboru systemowi PVM. System udostępnia także funkcje, które zwracają informacje o konfiguracji maszyny wirtualnej, na przykład o mocy obliczeniowej poszczególnych komputerów, dzięki czemu otrzymuje się możliwość

optymalnego wykorzystania poszczególnych maszyn. Komunikacja między zadaniami w systemie PVM opiera się na przesyle komunikatów. Proces zarządcy może wysłać, bez wstrzymywania siebie samego, komunikat zawierający polecenia do uaktualnienia celu i w ten sam sposób otrzymać informację o zakończeniu przetwarzania celu wraz z informacją o ewentualnych błędach.

PVM umożliwia tworzenie aplikacji równoległych na wysokim poziomie abstrakcji. Daje przy tym możliwość sterowania rozproszeniem części składowych aplikacji. W wyniku powyższej analizy podjęto decyzję o zaimplementowaniu programu makep według modelu zarządca i wykonawcy przy użyciu systemu PVM.

3. Implementacja programu make wykonującego operacje w sposób równoległy

W poprzedniej części zostały przedstawione teoretyczne koncepcje programu make, który wykonywałby operacje w sposób równoległy. Zarysowano tam kierunek, w którym zmierzać powinno rozwiązanie praktyczne. Tu podane zostaną pewne szczegółowe informacje dotyczące programu w jego konkretnej implementacji.

3.1. Użyte narzędzia oraz systemy informatyczne

W konkretnej implementacji programu makep wykorzystano:

- system operacyjny SunOS zarządzający komputerami SUN pracującymi w sieci;
- kompilator języka C o nazwie "gcc" w wersji 2.5 rozprowadzany w ramach projektu GNU;
- system umożliwiający tworzenie aplikacji równoległych i rozproszonych PVM w wersji 3.3, powstały w Oak Ridge National Laboratory.

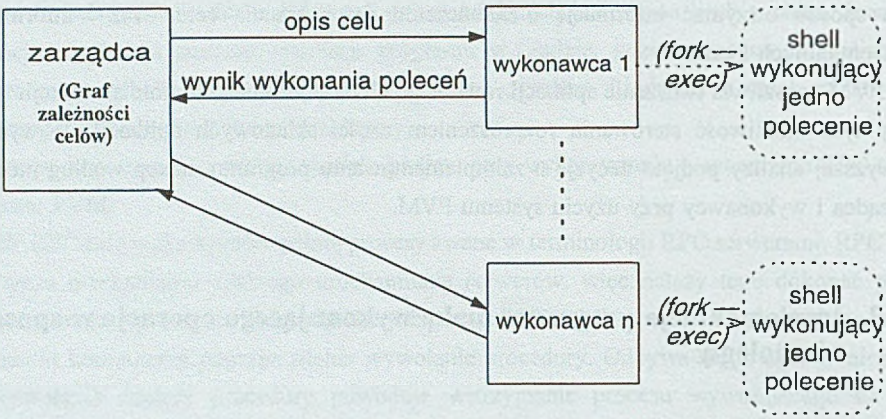
3.2. Zarys struktury wewnętrznej aplikacji makep

Zrealizowana aplikacja nosi nazwę makep i w rzeczywistości składa się z dwóch programów wykonywalnych oraz z jednego pliku dodatkowego:

- makep – program zarządcy, można uruchamiać go z linii poleceń;
- makep_w – program wykonawcy, uruchamiany przez zarządcę na wszystkich wykorzystywanych komputerach; nie może być wykonywany samodzielnie;
- default.mkp – plik zawierający predefiniowane zmienne i reguły domyślne.

Program zarządcy zajmuje się współpracą z użytkownikiem, analizą argumentów wywołania i plików wejściowych oraz rozsyłaniem celów do przetworzenia i odbiorem

wyników od wykonawców. Programy wykonawców odbierają opisy celów, wykonują ich polecenia (za pomocą shell'a) i przesyłają do zarządcy wynik tej operacji (informację, czy wystąpił błąd wykonania poleceń). Obrazowo przedstawia to rysunek 4.



Rys. 4. Struktura programu makep
Fig. 4. Makep programme structure

3.2.1. Program zarządcy

W pracy zarządcy można wyróżnić następujące fazy działania:

- analiza argumentów linii wywołania;
- analiza wartości zmiennych środowiskowych, z których program korzysta;
- wczytanie i analiza pliku z predefiniowanymi zmiennymi i regułami domyślnymi;
- wczytanie zawartości środowiska;
- wczytanie i analiza plików wejściowych użytkownika;
- utworzenie grafu zależności celów;
- inicjowanie przetwarzania równoległego;
- aktualizacja celów zgodnie z hierarchią zależności;
- prace związane z zakończeniem przetwarzania równoległego;
- zwrócenie do systemu kodu błędu;

Niektóre z tych faz zostaną omówione teraz nieco dokładniej.

3.2.1.1. Wczytanie i analiza plików

Podczas wczytywania i analizy plików program tworzy bazę zmiennych, bazę opisów celów oraz bazę definicji reguł domyślnych. Baza reguł domyślnych zorganizowana została jako lista liniowa, dlatego aby przy szukaniu reguł zachować kolejność, w jakiej były definiowane. Natomiast w przypadku zmiennych i opisów celów nie ma takiego wymogu. Dlatego ze względu na szybszy dostęp w trakcie odwołań do zmiennych i celów bazy te

zostały zorganizowane w postaci tablic mieszających z podwieszonymi listami (mieszanie otwarte).

3.2.1.2. Utworzenie grafu zależności celów

W koncepcji zarządcy i wykonawców przed przystąpieniem do aktualizacji celów musi być utworzony graf zależności celów. Graf ten jest budowany na podstawie zależności między celami, wyspecyfikowanymi w plikach wejściowych – w momencie budowy grafu opisy celów są już w bazie celów i stamtąd są pobierane. Brany jest także pod uwagę fakt istnienia plików o nazwach identycznych z nazwami celów i czasy ostatniej modyfikacji tych plików. Sposób, w jaki brane są pod uwagę te warunki, został już opisany.

Każdy z węzłów grafu odpowiada opisowi jednego celu. Część tych opisów jest brana z pliku wejściowego, a część powstaje dopiero w trakcie działania programu, w wyniku dopasowywania reguł domyślnych. Jako że program umożliwia równoczesne przetwarzanie wielu celów, konieczne było wprowadzenie do węzła dodatkowej informacji o aktualnym stanie danego celu. Każdy cel może być w jednym ze stanów (w poniższy sposób stany te są określane w programie):

- T_MakingNow – polecenia tego celu są obecnie wykonywane przez wykonawcę;
- T_Made – cel został już zaktualizowany;
- T_ReadyToMake – cel nie został jeszcze zaktualizowany, ale jest już gotowy do aktualizacji – jego wszystkie zależności są już zaktualizowane;
- T_NotReadyToMake – cel nie może być jeszcze analizowany, gdyż przynajmniej jedna z jego zależności nie została jeszcze zaktualizowana.

W fazie tworzenia grafu zależności celów, każdy z węzłów może być w jednym z trzech ostatnich stanów.

3.2.1.3. Inicjowanie przetwarzania równoległego

W fazie inicjalizacji przetwarzania równoległego proces zarządcy zgłasza się do systemu PVM (funkcją "pvm_mytid") i otrzymuje numer identyfikacyjny (w systemie PVM zadania są identyfikowane przez unikalny numer, tzw. tid – ang. task identifier). Następnie pobiera od systemu informacje o konfiguracji komputera wirtualnego i tworzy wewnętrzną tablicę tych komputerów, posortowaną według ich szybkości. Na każdym z tych komputerów powołuje programy wykonawców, którym przesyła aktualny katalog roboczy oraz zawartość środowiska, którą wykonawcy powinni przekazać wykonywanym poleceniom. Zarządca identyfikuje wykonawców przez ich numery (tid), które otrzymuje w momencie powołania wykonawców. Zarządca przechowuje informację, na którym komputerze jest dany wykonawca, dzięki czemu może wysyłać pracę zawsze najszybszemu wolnemu wykonawcy.

3.2.1.4. Aktualizacja celów zgodnie z hierarchią zależności

Aktualizacja celów dokonywana jest według grafu zależności. Sposób, w jaki program aktualizuje cały graf, jest następujący:

- znajdź i wyślij wszystkim wykonawcom (zaczynając od najszybszego) opisy celów, których polecenia należy wykonać; jeżeli nie można znaleźć wystarczającej ilości takich celów, to część wykonawców (tych wolniejszych) pozostanie "bezrobotna"; stan celów w grafie, których opisy wysłano, ustawiany jest na T_MakingNow;
- wykonuj w pętli (dopóki nie zaktualizowano celu głównego) oczekiwanie na komunikat i obsłużenie go; jest kilka rodzajów komunikatów, jednak poniższy opis dotyczy jedynie obsługi komunikatu o zakończeniu wykonywania poleceń (od wykonawcy):
- oznacz w grafie cel, który zaktualizował wykonawca jako T_Made;
- znajdź i wyślij bezrobotnym wykonawcom (zaczynając od najszybszego) opisy celów, których polecenia należy wykonać i ustaw stan tych celów w grafie na T_MakingNow.

Kluczowym zagadnieniem w przedstawionej dotychczas koncepcji równoczesnego aktualizowania wielu celów jest funkcja "znajdź". Funkcja ta ma za zadanie odszukać za każdym razem w grafie zależności celów jeden cel, który może być w danej chwili aktualizowany. Zajmuje się ona także uaktualnianiem stanu grafu. Taką funkcję zaimplementowano w programie zarządcy. Jej uproszczona wersja wygląda następująco:

```
/*****************************************************************/
/* poniższa funkcja operuje na dwóch polach węzła grafu:
   state - określa stan celu (opisane wyżej),
   dep_tab - tablica wskazań do węzłów w grafie będących zależnościami tego celu.
*/
/*****************************************************************/

/* 't_p' musi wskazywać na węzeł w grafie zależności celów;
funkcja szuka rekurencyjnie w grafie począwszy od celu wskazywanego przez 't_p' celu, który
można w tej chwili uaktualnić i zwraca wskazanie do niego; jeżeli w tym podgrafie nie ma
takiego celu, zwraca NULL;

przy okazji uaktualnia stan grafu:
jeśli jakiś cel na drodze poszukiwań jest 'T_NotReadyToMake' i wszystkie jego zależności są
już zaktualizowane, to ustawia jego stan na 'T_ReadyToMake' i zwraca wskazanie do niego.
*/
```

```
struct target * find_target_2do (struct target * t_p )
{
 if ( t_p->state == T_Made )
 /* tzn., że wszystko w tym podgrafie jest już uaktualnione */
 {
 return NULL;          /* porażka - UP */
 }

 if ( t_p->state == T_MakingNow )
 /* tzn., że ten cel jest obecnie aktualizowany */
 {
 return NULL;          /* porażka - UP */
 }

 if ( t_p->state == T_ReadyToMake )
 /* tzn., że ten cel nadaje się w tej chwili do aktualizacji */
 {
 return t_p;           /* (!) sukces - UP */
 }

 /* jeśli doszedł tutaj, tzn., że stan celu wskazywanego przez 't_p' jest 'T_NotReadyToMake',
    a to znaczy, że cel ten ma zależności, które należy sprawdzić, czy są gotowe do aktualizacji
    albo czy są już zaktualizowane ('T_Made') */

 { /* rekurencyjne poszukiwanie w podgrafach będących zależnościami celu 't_p': */

 int dep_made_f = 1; /* będzie '1' po pętli 'for' jeśli wszystkie zależności celu są już
                        zaktualizowane */
 unsigned i;         /* numer zależności */

 /* dla wszystkich zależności celu wskazywanego przez 't_p': */

 for ( i = 0 ; t_p->dep_tab[i] != NULL ; i++ )
 /* szukaj, aż któraś będzie gotowa do zaktualizowania */
 {
 struct target *t_2do_p = find_target_2do (t_p->dep_tab[i]);
```

```
if ( t_2do_p != NULL ) /* znaleziono */
return t_2do_p;        /* (!) sukces - UP */

/* skasuj flagę, jeśli zależność nie jest uaktualniona: */
if ( t_2do_p->state != T_Made )
dep_made_f = 0;
} /* for */
}

/* jeśli doszedł tutaj, tzn., że w żadnym podgrafie tego celu nie można nic w tej chwili
zaktualizować, ale jeśli wszystkie zależności tego celu są już uaktualnione, to cel ten jest
gotowy do aktualizacji: */

if ( dep_made_f )
{
t_p->status = T_ReadyToMake; /* <- uaktualnia stan grafu */
return t_p;        /* (!) no i sukces - UP */
}

/* jeśli doszedł tutaj tzn., że ani ten cel, ani nic w jego podgrafach nie nadaje się w tej chwili
   do aktualizacji */
return NULL;
} /* find_target_2do */

/*****************************************************************/
```

Funkcja "find_target_2do" z parametrem wskazującym na węzeł odpowiadający celowi głównemu wywoływana jest przez zarządcę w miejsce funkcji zapisanej powyżej jako znajdź.

3.2.1.5. Prace związane z zakończeniem przetwarzania równoległego

Po zakończeniu aktualizacji zarządca wysyła wykonawcom komunikat, żeby zakończyli działanie i opuszcza system równoległy (funkcją "pvm_exit").

3.2.2. Program wykonawcy

W pracy wykonawcy można wyróżnić następujące fazy działania:

- odbiór od zarządcy katalogu roboczego oraz zawartości środowiska, które będzie przekazywane wykonywanym poleceniom;
- dopóki nie odebrano komunikatu o zakończeniu pracy (w pętli):

- odebranie opisu celu;
- wykonanie sekwencyjnie wszystkich poleceń;
- odesłanie do zarządcy informacji o tym, czy polecenia zostały wykonane bezbłędnie.

Jak napisano wcześniej, wykonywanie poleceń jednego celu odbywa się w sposób sekwencyjny. Wykonanie jednego polecenia realizuje funkcja "do_cmd" poprzez uruchomienie shell'a, wykonującego te polecenie, jako procesu potomnego. W systemie UNIX uruchomienie jednego programu z drugiego osiąga się za pomocą pary funkcji systemowych "*fork*" i "*exec*".

Jak wcześniej zasygnalizowano, zadania w systemie PVM są identyfikowane przez unikalne numery, które przydziela PVM. Dowolny proces UNIX'owy może stać się zadaniem systemu PVM. Następuje to w momencie wywołania dowolnej funkcji z biblioteki PVM. W czasie tego pierwszego odwołania do PVM proces otrzymuje numer. Tak dzieje się w przypadku procesów, które uruchamiane są jako normalne procesy UNIX'owe i same zgłaszają się do systemu PVM (takim procesem jest np. proces zarządcy).

Drugim sposobem uruchomienia zadania pracującego w systemie PVM jest uruchomienie go specjalną funkcją ("pvm_spawn") przez istniejące już zadanie PVM'a. W tym przypadku numer przydzielany jest nowemu zadaniu właśnie w tym momencie, a zadanie powołujące nowe zadania zna ich numery. W ten sposób proces zarządcy uruchamia procesy wykonawców i na podstawie otrzymanych numerów identyfikuje ich.

Pierwotna funkcja "do_cmd" do wykonania polecenia uruchamiała proces potomny (funkcją "*fork*"). I w tym miejscu pojawił się problem. Oba te procesy są identyfikowane w systemie PVM przez ten sam numer, a co gorsza, gdy proces potomny skończy się, PVM przestaje "widzieć" proces rodzica (wykonawcę) jako zadanie PVM – wysłanie komunikatu do zadania o tym numerze (mimo że ono istnieje) kończy się błędem w rodzaju "brak zadania o takim numerze" (nazwano to efektem znikających wykonawców).

Do rozwiązania powyższego problemu wykorzystano pewną cechę systemu PVM. Oprócz tego, że dowolny proces UNIX'owy może stać się zadaniem PVM (jak opisano to wyżej), jest możliwa także sytuacja odwrotna. Zadanie PVM może opuścić w dowolnej chwili system przez wykonanie funkcji "pvm_exit". W przypadku opuszczenia przez zadanie systemu PVM, pierwsze następne wywołanie funkcji systemu nada temu zadaniu nowy numer.

Problem znikających wykonawców praktycznie rozwiązano w ten sposób, że tuż przed powołaniem procesu potomnego wykonawca opuszcza system PVM i natychmiast po tym ponownie do niego wchodzi otrzymując nowy numer. W ten sposób, w momencie powoływania procesu potomnego, proces wykonawcy nie jest zadaniem PVM i problem znika. Dodatkowo, jako że proces wykonawcy zmienia swój identyfikator, musi więc go przesyłać do zarządcy, aby ten mógł się z nim dalej komunikować. Ostatecznie funkcja "do_cmd" wygląda następująco:

```
/*****************************************************************/

int do_cmd ( char *shell, char *cmd )
{
 int status;

 pvm_exit (); /* <- opuść system PVM */

 switch ( fork () )
 {
 case -1   :   /* błąd systemowy: */
 SYS_FATAL ("fork") ;        /* EXIT */

 case 0    :   /* potomek: */
 /* wykonaj shell'a: */
 execlp ( shell, shell, "-c", cmd, NULL );

 /* z funkcji 'execlp' nie ma powrotu ! */
 SYS_FATAL ("execlp") ;        /* EXIT */

 default   :   /* rodzic: */
 tid = pvm_mytid (); /* <- wejdź ponownie do PVM */
 /* i prześlij swój nowy numer do zarządcy: */

 . . .

 /* czekaj na zakończenie potomka: */
 if ( wait ( &status ) == -1 )
 SYS_FATAL ("wait") ;        /* EXIT */

 return status;     /* zwróć kod wykonania polecenia */
 } /* switch ( fork ) */

} /* do_cmd */

/*****************************************************************/
```

4. Ocena efektywności programu MAKEP

W rozdziale tym dokonano oceny efektywności programu makep umożliwiającego wykonywanie operacji równolegle w sieci komputerów. Zawarto także porównanie z programem make dostarczanym wraz z systemem operacyjnym komputerów SUN.

4.1. Metoda i środowisko pomiarów

Badania nad efektywnością programu makep przeprowadzono w sieci komputerów SUN dostępnej w Instytucie Informatyki Politechniki Śląskiej. W skład sieci wchodzi siedem komputerów, z czego do badań wykorzystano cztery. Badania przeprowadzono w warunkach normalnego użytkowania sieci – na części komputerów były uruchomione procesy innych użytkowników. Do badań wykorzystano następujące komputery: sun10, classic1, classic2, classic3.

Wszystkie komputery wchodzące w skład sieci korzystały z wspólnego, sieciowego systemu zbiorów (NFS), w wyniku czego dostęp do zbiorów z poszczególnych komputerów odbywał się za pośrednictwem sieci.

Testy przeprowadzono dla trzech projektów, z których dwa pierwsze zostały specjalnie przygotowane:

- projekt 1 – wykonanie aktualizacji projektu polega na skompilowaniu szesnastu plików źródłowych w języku C, między którymi nie ma zależności – wszystkie kompilacje mogą być wykonane równocześnie; plik wejściowy z opisem projektu jest następujący:

```
# kompilacja - współpraca z dyskiem
a: a1 a2 a3 a4 a5 a6 a7 a8 a9 a10 a11 a12 a13 a14 a15 a16
.SUFFIXES:
.SUFFIXES: .c
.c:
    gcc -c -o $@ $*.c
    rm -f $@
```

- projekt 2 – wykonanie aktualizacji projektu polega na szesnastokrotnym wykonaniu jednego programu "test1", który operuje na dynamicznych strukturach danych (listach) w pamięci, bez odwoływania się do dysku; między celami nie ma zależności – wszystkie wykonania programu "test1" mogą odbywać się równocześnie; plik wejściowy z opisem projektu jest następujący:

```
# wykonanie programu - bez współpracy z dyskiem
a: a1 a2 a3 a4 a5 a6 a7 a8 a9 a10 a11 a12 a13 a14 a15 a16
.SUFFIXES:
```

.SUFFIXES: .c

.c:

test1

- projekt 3 – jest to rzeczywisty projekt programowy pochodzący z sieci Internet; wykonanie aktualizacji tego projektu polega na przeprowadzeniu kompilacji i łączenia plików źródłowych pakietu programowego "GNU Make" w celu otrzymania postaci wykonywalnej; w tym przypadku o możliwości równoczesnego przetwarzania celów decydowały rzeczywiste zależności między plikami.

W trakcie badań mierzono całkowity czas potrzebny na wykonanie aktualizacji projektu. Pomiary dokonane zostały poleceniem systemowym UNIX'a "time". Polecenie "time" mierzy czas wykonania programu będącego jego argumentem. Składnia tego polecenia jest następująca:

time program

Każdy pomiar dokonywany był trzykrotnie w celu uniezależnienia się od chwilowych zmian obciążenia komputerów.

4.2. Wyniki badań

Na początku wykonane zostały pomiary czasu wykonania aktualizacji wszystkich projektów na każdym komputerze z osobna. W tym przypadku, gdy do wykonania zadania angażowany był tylko jeden komputer, aktualizacji projektów dokonano wykorzystując zarówno program makep, jak i Sun make.

Nie wszystkie z wykorzystywanych komputerów mają jednakową moc. Komputery "classic" są słabsze od komputera "sun10". W celu umożliwienia dalszej analizy przyporządkowano każdemu komputerowi moc względną. Jako moc jednostkową przyjęto moc najsilniejszej maszyny. Moc względna pozostałych maszyn wskazuje, jaką częścią mocy jednostkowej dysponuje dany komputer. Jako że dla każdego projektu stosunek ten jest nieco inny, obliczenie mocy względnej przeprowadzono dla każdego z nich osobno.

Kolejne pomiary przeprowadzono już tylko dla programu makep. Stworzono cztery razy komputer wirtualny, który składał się kolejno z 1, 2, 3 i 4 komputerów dostępnych w sieci. Komputery dołączano w następującej kolejności: "sun10", "classic1", "classic2", "classic3". Dla każdego takiego komputera wirtualnego mierzono czas wykonania aktualizacji każdego z projektów. Wyniki zawierają tabele 2, 3 i 4.

Tabela 2

Aktualizacja projektu 1 na komputerach wirtualnych

Liczba komputerów	Moc	Czas wykonania [s]				Przyspieszenie	Sprawność [%]
		1	2	3	średnia		
1	1.00	46	46	45	45.7	1.00	100
2	1.68	29	29	29	29.0	1.58	94
3	2.33	22	22	23	22.3	2.05	88
4	2.98	20	19	20	19.7	2.32	78

Tabela 3

Aktualizacja projektu 2 na komputerach wirtualnych

Liczba komputerów	Moc	Czas wykonania [s]				Przyspieszenie	Sprawność [%]
		1	2	3	średnia		
1	1.00	31	31	31	31.0	1.00	100
2	1.80	20	19	19	19.3	1.61	89
3	2.47	16	15	15	15.3	2.03	82
4	3.26	12	12	12	12.0	2.58	79

Tabela 4

Aktualizacja projektu 3 na komputerach wirtualnych

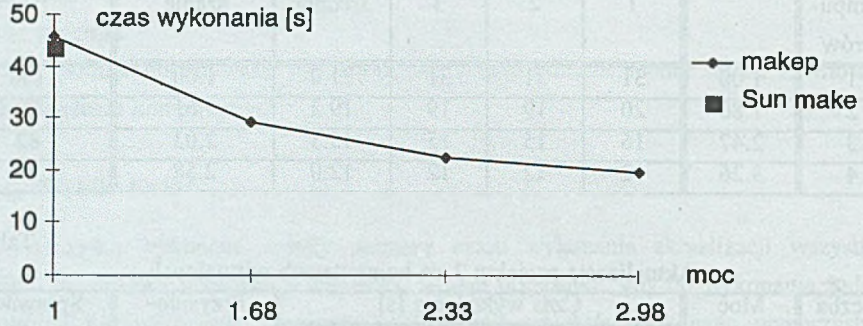
Liczba komputerów	Moc	Czas wykonania [s]				Przyspieszenie	Sprawność [%]
		1	2	3	średnia		
1	1.00	87	86	87	86.7	1.00	100
2	1.75	57	58	57	57.3	1.51	86
3	2.42	48	48	49	48.3	1.80	74
4	3.12	41	40	40	40.3	2.15	69

Kolumna pierwsza tych tabel określa liczbę komputerów wchodzących w skład komputera wirtualnego, a kolumna druga ich sumaryczną moc względną. Względna moc komputera wirtualnego określa równocześnie maksymalne przyspieszenie jakie można uzyskać. Kolejne kolumny zawierają wyniki trzech pomiarów oraz średni czas wykonania aktualizacji. Przedostatnia kolumna zawiera wartość przyspieszenia,jakie uzyskano w danej konfiguracji, w stosunku do wykonania aktualizacji danego projektu na jednym komputerze, o mocy jednostkowej ("sun10"). Kolumna ostatnia ukazuje sprawność danej konfiguracji komputera wirtualnego. Sprawność określona jest następującym wzorem:

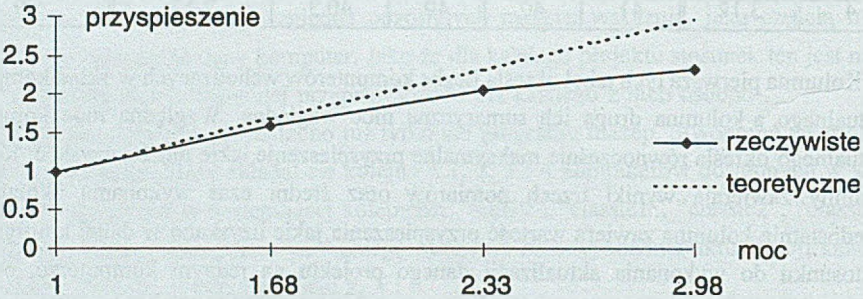
$$sprawność = \frac{osiągnięte\ przyspieszenie}{moc\ komputera\ wirtualnego} * 100\%$$

Rysunki 5 do 13 zawierają wykresy sporządzone na podstawie powyżej zamieszczonych tabel, osobno dla każdego projektu:

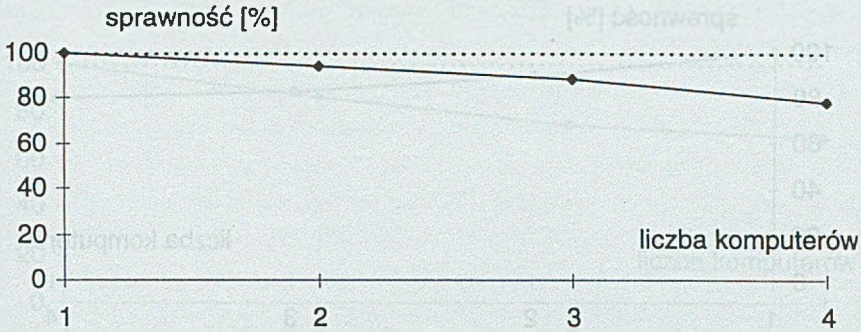
- czas wykonania aktualizacji projektu w zależności od mocy komputera wirtualnego, oraz dla porównania, czas wykonania aktualizacji na jednym komputerze ("sun10") przez Sun make;
- przyspieszenie wykonania aktualizacji projektu w zależności od mocy komputera wirtualnego;
- sprawność komputera wirtualnego względem liczby komputerów dla danego projektu.



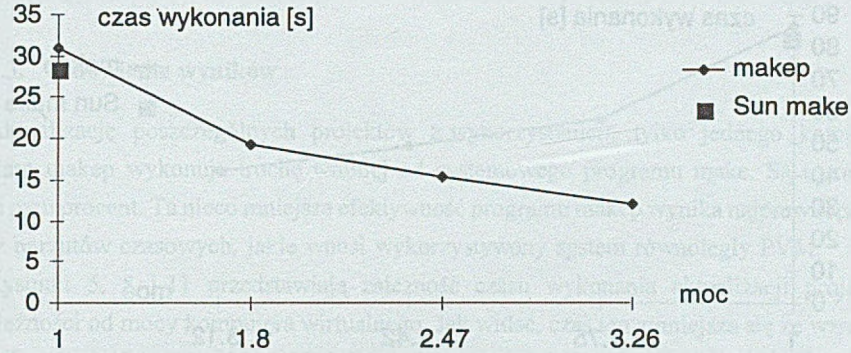
Rys. 5. Czas wykonania aktualizacji projektu 1
Fig. 5. Project 1 actualization time



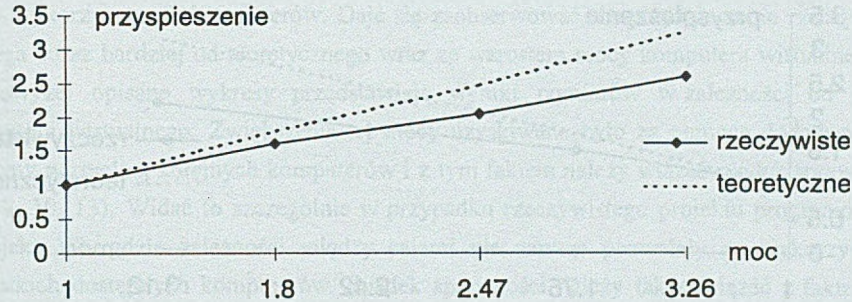
Rys. 6. Przyspieszenie wykonania aktualizacji projektu 1
Fig. 6. Project 1 aktualization speed-up



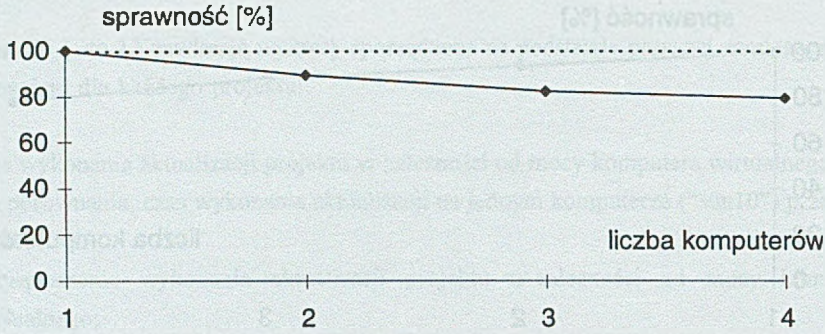
Rys. 7. Sprawność komputera wirtualnego dla projektu 1
Fig. 7. Project 1 virtual computer performance



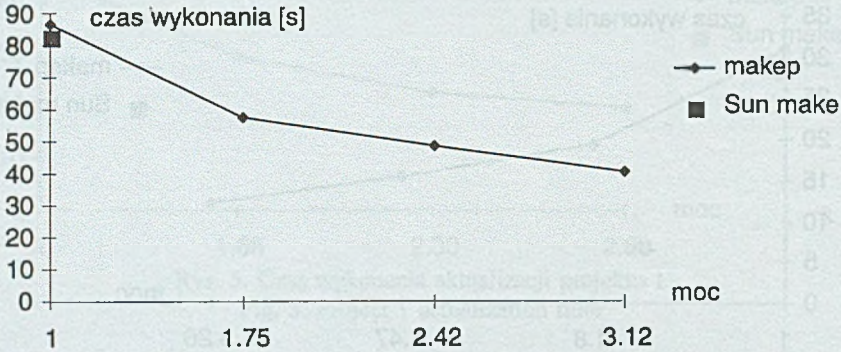
Rys. 8. Czas wykonania aktualizacji projektu 2
Fig. 8. Project 2 actualization time



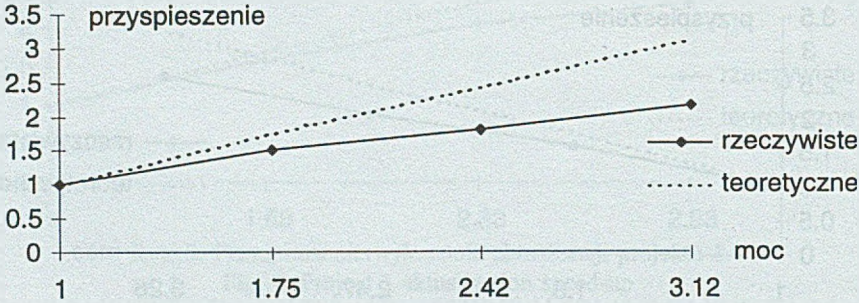
Rys. 9. Przyspieszenie wykonania aktualizacji projektu 2
Fig. 9. Project 2 aktualization speed-up



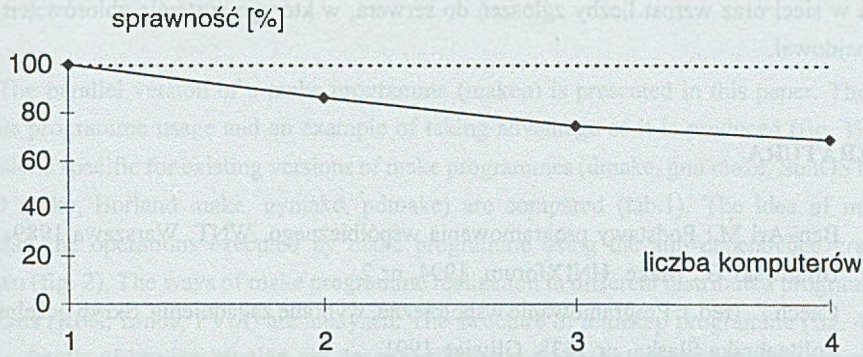
Rys. 10. Sprawność komputera wirtualnego dla projektu 2
Fig. 10. Project 2 virtual computer performance



Rys. 11. Czas wykonania aktualizacji projektu 3
Fig. 11. Project 3 actualization time



Rys. 12. Przyspieszenie wykonania aktualizacji projektu 3
Fig. 12. Project 3 actualization speed-up



Rys. 13. Sprawność komputera wirtualnego dla projektu 3
Fig. 13. Project 3 virtual computer performance

4.3. Omówienie wyników

Aktualizację poszczególnych projektów z wykorzystaniem tylko jednego komputera program makep wykonuje trochę wolniej od systemowego programu make. Są to różnice rzędu paru procent. Ta nieco mniejsza efektywność programu makep wynika najprawdopodobniej z narzutów czasowych, jakie wnosi wykorzystywany system równoległy PVM.

Rysunki 5, 8 i 11 przedstawiają zależność czasu wykonania aktualizacji projektów w zależności od mocy komputera wirtualnego. Jak widać, czas ten zmniejsza się ze wzrostem mocy, czego należało oczekiwać, jednak nie jest to spadek czysto liniowy. Największy skok obserwujemy przy zwiększeniu liczby komputerów z jednego do dwóch.

Rysunki 6, 9 i 12 prezentują rzeczywiste przyspieszenie, jakie uzyskano zwiększając moc komputera wirtualnego oraz przyspieszenie teoretycznie możliwe wynikające z sumarycznej mocy wykorzystanych komputerów. Daje się zaobserwować, że przyspieszenie rzeczywiste odbiega coraz bardziej od teoretycznego wraz ze wzrostem mocy komputera wirtualnego.

Powyżej opisane wykresy przedstawiają wyniki pomiarów w zależności od mocy komputera wirtualnego. Zwiększenie tej mocy uzyskiwane było za pomocą dodawania do maszyny wirtualnej kolejnych komputerów i z tym faktem należy wiązać spadek sprawności (rys. 7, 10, 13). Widać to szczególnie w przypadku rzeczywistego projektu programowego ("projekt 3"), gdzie zależności między celami nie zawsze pozwalały na wykorzystanie wszystkich dostępnych komputerów. Spadek sprawności należy także wiązać z faktem, iż wykonanie każdej operacji wymagało uruchomienia programu znajdującego się w sieciowym systemie zbiorów (NFS), co przy zwiększającej się ilości wykonawców powodowało wzrost

ruchu w sieci oraz wzrost liczby zgłoszeń do serwera, w którego systemie zbiorów ten plik się znajdował.

LITERATURA

[1] Ben-Ari M.: Podstawy programowania współbieżnego. WNT, Warszawa 1989.

[2] Blinowski G.: Make. UNIXforum, 1994, nr 2.

[3] Czech Z. (red.): Programowanie współbieżne, wybrane zagadnienia. Skryp uczelniany, Politechnika Śląska, nr 1638, Gliwice 1991.

[4] Gabassi M., Dupouy B.: Przetwarzanie rozproszone w systemie UNIX. LUPUS, Warszawa 1995.

[5] Goszyński R., Tuszyński M.: Programowanie w systemie UNIX. UNIXforum, 1993–1994.

[6] Instrukcje laboratoryjne z przedmiotu Architektura Komputerów.

[7] Kernighan B. W., Ritchie D. M.: Język C. WNT, Warszawa 1988.

[8] Pająk A., Wigura A.: Makrogeneratory, asemblery i konsolidatory. PWN, Warszawa 1983.

[9] Rochkind M. J.: Programowanie w systemie Unix dla zaawansowanych. WNT, Warszawa 1993.

[10] Silvester P. P.: System operacyjny Unix. WNT, Warszawa 1990.

[11] Dokumentacja do systemu operacyjnego SunOS.

[12] Weiss Z., Gruźlewski T.: Programowanie współbieżne i rozproszone. WNT, Warszawa 1993.

[13] Wirth N.: Algorytmy + struktury danych = programy. WNT, Warszawa 1989.

[14] Wolisz A.: Podstawy lokalnych sieci komputerowych – Oprogramowanie sieciowe i usługi sieciowe. WNT, Warszawa 1992, t. 2.

Dokumentacja elektroniczna

[15] The GNU Make Manual. Maj 1994.

[16] Geist A. i inni: PVM 3 USER'S GUIDE AND REFERENCE MANUAL. Tennessee, Oak Ridge National Laboratory 1994 (ORNL/TM-12187).

[17] Manual pages for: BSD pmake, GNU make, Sun make, PVM 3.

Recenzent: Doc. dr hab. inż. Tadeusz Czachórski

Wpłynęło do Redakcji 29 listopada 1996 r.

Abstract

The parallel version of a make programme (makep) is presented in this paper. The idea of the programme usage and an example of taking advantage of it is produced (fig. 1). The functions specific for existing versions of make programmes (dmake, gnu make, SunOS make, BSD make, Borland make, gymake, pdmake) are compared (tab.1). The idea of making parallel the operations executed by make programme using the aim dependence graph is shown (fig. 2). The ways of make programme realization in different distributed programming systems (RPC, Linda, PVM) are analyzed. The structure of a makep programme (fig. 4) and some details of implementation are described. The term of parallel computer's performance is introduced. The effectiveness (the time of execution, acceleration and efficiency) of makep programme depending on parallel computer's performance is analyzed (tab. 2–4; fig. 5–13).