STUDIA INFORMATICA 2001
Volume 22 Number 3 (45)

Hafedh ZGHIDI
Politechnika Śląska, Instytut Informatyki

SYSTEM UMOŻLIWIAJĄCY GENEROWANIE APLIKACJI RÓWNOLEGŁYCH WYKORZYSTUJĄCYCH MECHANIZMY ZDALNEGO WYWOŁANIA PROCEDUR

Streszczenie. Niniejsza praca przedstawia próbę stworzenia modułu umożliwiającego generowanie aplikacji równoległych wykorzystujących mechanizmy zdalnego wywołania procedur, jak również przykład jego zastosowania. Takie rozwiązanie ma na celu ułatwienie i przyspieszenie procesu opracowania programów równoległych.

A SYSTEM ENABELING PARALLEL APPLICATIONS GENERATION USING REMOTE PROCEDURE CALL MECHANISMS

Summary. The paper presents a program that enables the creation of parallel programs based on remote procedure call mechanisms. Using this tool, programmers can accelerate the process of parallel programs generation. An example of how to use this tool is also presented.

1. Wstęp

Rosnące zapotrzebowanie na coraz szybsze i bardziej efektywne przetwarzanie danych doprowadziło do powstania wachlarza systemów umożliwiających opracowanie programów rozproszonych bądź równoległych. Istnieje szereg narzędzi i systemów umożliwiających tworzenie i opracowanie aplikacji równoległych. Do dostępnych narzędzi można zaliczyć: PCN jako system i język programowania równoległego, system Piranha umożliwiający tworzenie komputera wirtualnego poprzez połączenie komputerów sieciowych, system Paradise wykorzystujący model wirtualnej pamięci współdzielonej oraz inne systemy i narzędzia, takie jak RMS i biblioteka P4 [1, 2]. Do najpopularniejszych narzędzi zaliczyć należy

system PVM, system MPI i system Linda oraz mechanizmy zdalnego wywołania procedur (RPC, ang.Remote Procedure Call).

Niektóre dostępne narzędzia, takie jak wymieniane wcześniej systemy Linda lub PVM, w porównaniu z mechanizmami RPC, są dla programisty bardziej przyjazne i dużo łatwiejsze w stosowaniu. Jako samodzielne systemy oferują szereg funkcji lub procedur, których wykorzystanie zapewnia transmisję danych, wykonanie i synchronizację zdalnych procesów. Niniejsza praca przedstawia próbę opracowania modułu wspomagającego, umożliwiającego generowanie szeregu funkcji i modułów ułatwiających korzystanie z mechanizmów RPC oraz organizujących działanie programu równoległego. Moduł APPLGEN, odgrywający rolę generatora aplikacji równoległych, tworzy kody źródłowe modułów aplikacji równoległej opartej na wywołaniach RPC w trybie nieblokującym lub w trybie rozgłaszania. Oba warianty wykorzystania RPC charakteryzują się dużym stopniem efektywności [3,4,7,11] i pozwalają na wywołanie procedur zdalnych w trybie synchronicznym, co umożliwia równoległe wykonanie programów.

2. Mechanizmy zdalnego wywołania procedur

Mechanizmy RPC [3,4,5,6,7] są wysokopoziomowymi protokołami komunikacyjnymi, pozwalającymi na tworzenie aplikacji sieciowych poprzez specjalnie przygotowane procedury. Procedury te zasłaniają niskopoziomowe sieciowe mechanizmy, ułatwiając tym samym zadanie programistom.

RPC implementuje logiczny model klient/serwer. Za pomocą RPC klienci wywołują procedury, które zgłaszają żądania wykonania usługi do serwera. Gdy na komputerze serwera pojawia się żądanie, serwer przyjmuje je, wykonuje zadanie i wysyła odpowiedź. Głównym celem programowania za pomocą RPC jest możliwość wykonania aplikacji w sieciowym systemie komputerowym, gdzie dowolny komputer może grać rolę serwera, klienta lub równocześnie serwera i klienta. Aplikacje te korzystają z mechanizmów RPC, by uniknąć szczegółów związanych z programowaniem w sieci, umożliwiając tym samym klientom usługi sieciowe, bez świadomości istnienia i funkcjonowania sieci. Programy napisane z wykorzystaniem RPC mogą być implementowane w różnych językach programowania (C, Pascal ...), uruchamiane na takich samych komputerach bądź na różnych komputerach o odmiennej architekturze i mogą zapewniać komunikację między procesami na tym samym komputerze lub na różnych komputerach.

Zaletą wywołania procedur zdalnych jest to, że procedury te są wykonywane w przestrzeni adresowej komputera obsługującego (serwera), co pozwala na wykorzystanie ich mocy obliczeniowej i odciążenie komputera klienta.

Mechanizmy RPC można wykorzystać w trybie blokującym, nieblokującym, zwrotnym, wsadowym oraz rozgłaszania. Równoległa realizacja programów z wykorzystaniem RPC jest możliwa pod warunkiem, że mechanizmy te wykorzystane są w trybie synchronicznym. Wymaga to jednak stosowania dodatkowych mechanizmów wspomagających zapewniających synchronizację obliczeń. Jest to konieczne w przypadku, kiedy wyniki wykonania programów zdalnych są potrzebne do dalszego wykonania programu równoległego. W pracy jako mechanizmy synchronizacji wykorzystano pamięć dzieloną, semafory oraz sygnały [9,10].

3. Organizacja obliczeń równoległych

Przedstawione w pracy rozwiązanie oparte jest na dynamicznym modelu podziału zadań. Polega to na tym, że komputery zdalne po wykonaniu zadania zgłaszają swoją gotowość do wykonania kolejnych obliczeń poprzez wysłanie do komputera sterującego (lokalnego) swojego identyfikatora w postaci wcześniej przydzielonego im numeru. Alternatywnym rozwiązaniem jest tzw. statyczny podział zadań, gdzie liczba przeznaczonych do wykonania zadań przez poszczególne zdalne komputery jest obliczana na początku programu równoległego przez komputer sterujący i jest na ogół wynikiem dzielenia liczby wszystkich zadań przez liczby zdalnych komputerów (zakładając, że liczba zadań jest podzielna przez liczbę komputerów) [7]. Obie metody mają swoje wady i zalety.

Wadą pierwszej metody jest to, że konieczna jest całkowita niezależność pomiędzy zdalnie wykonywanymi zadaniami. Dodatkowo metoda ta powoduje wzrost liczby operacji transmisji i synchronizacji danych. Zaletą natomiast jest równomierny podział zadań na poszczególne zdalne komputery. Liczba wykonywanych zadań przez wykorzystane komputery zależy jedynie od ich mocy obliczeniowej i ich aktualnego obciążenia. Taka metoda podziału zadań pozwala wykorzystywać słabsze komputery do wykonania części obliczeń.

Zaletą drugiej metody jest mniejsza liczba operacji transmisji i synchronizacji danych. Wadą zaś jest wzrost czasu obliczeń zdalnych w przypadku wykorzystania słabszych komputerów, co kończy się zwykle rezygnacją z ich wykorzystania do obliczeń zdalnych, a więc nie wszystkie dostępne zasoby sieci komputerowej są wykorzystane.

4. Etapy generowania programu równoległego

Opracowanie programu równoległego z wykorzystaniem mechanizmów RPC składa się z kilku etapów. Poza opracowaniem algorytmu równoległego (koncepcji rozwiązania)

zadaniem użytkownika jest wykonanie następujących czynności pozwalających na skompletowanie wszystkich komponentów programu równoległego:

a) określenie specyfikacji procedur zdalnych. Polega ona na generowaniu pliku źródłowego dla kompilatora RPCGEN (plik z rozszerzeniem .x zawierający opis wszystkich procedur zdalnych [3,7]),

b) uruchomienie kompilatora RPCGEN w celu generowania plików źródłowych pieńka klienta i pieńka serwera, pliku nagłówkowego oraz filtrów XDR [3,4,5] ,

c) generowanie kodu źródłowego programu klienta, który musi zawierać wywołania funkcji odpowiadające za:
   - nawiązanie połączenia z serwerem,
   - inicjację procedur zdalnych,
   - przesłanie danych i wywoływanie procedur obliczeniowych, synchronizację obliczeń, odbioru wyników.

d) generowanie kodu źródłowego programu serwera zawierającego treści wszystkich procedur zdalnych, które zapewniają:
   - wykonanie obliczeń,
   - zwrócenie wyników obliczeń do komputera klienta.

Aby ułatwić zadanie użytkownikowi, opracowany moduł APPLGEN zapewnia samodzielne wykonanie etapów b), c) i d). Etap a) jest zastępowany innym etapem, dużo łatwiejszym. Przy wykorzystaniu modułu APPLGEN etapy opracowania programu równoległego są następujące:

a) definiowanie funkcji zdalnych,

b) wywołanie generatora "*readinf*", który generuje:
   - plik ze specyfikacją procedur zdalnych, na podstawie czynności z etapu a),
   - gotowy kod źródłowy programu klienta,
   - gotowy kod źródłowy programu serwera,
   - wszystkie potrzebne mechanizmy synchronizacji.

Przy zastosowaniu modułu rola użytkownika sprowadza się jedynie do umieszczenia w odpowiednim miejscu generowanego automatycznie kodu źródłowego programu serwera ciała procedury obliczeniowej, przeznaczonej do wykonania na zdalnych komputerach. Użytkownik musi również umieścić w odpowiednim miejscu generowanego automatycznie kodu programu klienta rozkazy umożliwiające wywołanie procedur zdalnych oraz odbiór i przetwarzanie wyników. Miejsca te są wskazane odpowiednim komentarzem.

5. Opis modułu

Podczas projektowania modułu wzięto pod uwagę fakt, że musi on zapewnić duży stopień elastyczności. Użytkownik może generować większą liczbę procedur zdalnych o dowolnej liczbie parametrów wejściowych i wyjściowych. W tym celu zdefiniowano następujące zmienne i nadano im wartości początkowe, które można w razie potrzeby zmieniać:

MAXLEN 50: maksymalna długość nazwy procedury zdalnej,
MAXINP 10: maksymalna liczba parametrów wejściowych procedury zdalnej,
MAXOUT 10: maksymalna liczba parametrów wyjściowych procedury zdalnej,
MAXFUN 20: maksymalna liczba procedur zdalnych,
MAXCOMP 10: maksymalna liczba komputerów zdalnych.

Parametry te są definiowane w pliku nagłówkowym o nazwie *common.h*. W przypadku gdy wymagana jest inna konfiguracja, wystarczy te parametry odpowiednio ustawić.
Pierwszym zadaniem modułu jest sprawdzenie listy procedur i ich parametrów wejściowych i wyjściowych, które użytkownik zamierza zdalnie wywołać. Opis tych procedur użytkownik umieszcza w pliku tekstowym, którego nazwę podaje jako parametr wejściowy generatora *readinf*. Plik ten zastępuje tradycyjny plik ze specyfikacja RPC i ma następującą strukturę:

```
{                                          /* Początek definicji procedury zdalnej */
FNAME     nazwa_procedury_zdalnej
INPUT     typ_parametru_wejściowego  nazwa_parametru_wejściowego
INPUT     typ_parametru_wejściowego  nazwa_parametru_wejściowego
...
OUTPUT    typ_parametru_wyjściowego  nazwa_parametru_wyjściowego
OUTPUT    typ_parametru_wyjściowego  nazwa_parametru_wyjściowego
...
}                                          /* Koniec definicji procedury zdalnej */
```

Poniżej przedstawimy przykład pliku opisującego dwie procedury zdalne:

```
{
FNAME     f1
INPUT     int     nrcomp
INPUT     double  pi
INPUT     char    znak
OUTPUT    int     nrproc
OUTPUT    int     wynik
}
{
FNAME     f2
INPUT     float   tabA<>
INPUT f   loat    tabB[100]
OUTPUT    string  nazwa
}
```

Parametry wejściowe i wyjściowe mogą być typu prostego (takie jak typ zmiennych w języku C) lub typu akceptowanego przez RPCGEN (tablica znaków "string", tablica o zmiennej długości itp.[3]). Struktura pliku konfiguracyjnego jest bardzo prosta i nie wymaga od użytkownika samodzielnego definiowania typów pośrednich w celu przekazywania do procedury zdalnej zmiennych o typie złożonym (takich jak tablice wielowymiarowe, wskaźniki do wskaźników itd).

Słowo kluczowe FNAME oznacza, że procedura zdalna ma być wywołana w trybie nieblokującym. W przypadku gdy procedura zdalna ma być wywołana w trybie rozgłoszeniowym, należy słowo FNAME zastąpić słowem BROAD.

Po sprawdzeniu i przygotowaniu listy procedur zdalnych i ich parametrów, kolejnym krokiem modułu jest generowanie plików ze specyfikacją procedur zdalnych dla kompilatora RPCGEN.

W celu zapewnienia równoległej realizacji programu wykorzystuje się dwa serwery RPC. Jeden serwer RPC uruchamiany jest na komputerach zdalnych i zapewnia wywołanie procedur obliczeniowych związanych z rozwiązywanym zadaniem. Drugi serwer RPC uruchamiany jest na komputerze lokalnym (sterującym) i zapewnia przejście przez kolejne etapy programu równoległego. W tym celu generowane są dwa pliki, z których pierwszy, *pwork.x*, opisuje procedury przeznaczone do wykonania na zdalnych komputerach, a drugi, *waik.x*, specyfikuje procedury drugiego serwera RPC, uruchamianego na komputerze lokalnym, odbierającego wyniki procedur zdalnych. Dodatkowo umieszczone są w nim definicje mechanizmów zapewniających synchronizację, pozwalające na przejście przez kolejne etapy programu równoległego. Definicje te są generowane bez udziału użytkownika.

Do synchronizacji procesów zdalnych i programu klienta można wykorzystać pamięć dzieloną (**PD**), semafory i sygnały. Mechanizmy te zostały zbadane i wykorzystane z powodzeniem w poprzednich pracach [7,11]. Kolejnym krokiem modułu APPLGEN jest definiowanie struktury obszaru pamięci dzielonej. W celu ułatwienia operacji odbioru wyników wszystkich procedur zdalnych, strukturę obszaru pamięci dzielonej zaprojektowano w sposób przedstawiony na rysunku 1. Składa się ona z parametrów wyjściowych wszystkich definiowanych procedur oraz dodatkowo z nazwy procedury zdalnej (pole *funcname*) i numeru komputera zdalnego (pole *nrcomp*). Dodatkowe pola ułatwiają pobieranie wyników działania wybranej procedury zdalnej wykonywanej na określonym oddalonym komputerze. Po generowaniu plików ze specyfikacją RPCGEN zostaje uruchomiony kompilator RPCGEN. Jeśli w wyniku kompilacji powstają błędy, działanie modułu jest przerywane i użytkownik jest informowany o konieczności sprawdzenia pliku z definicją procedur zdalnych. Jeśli wynik działania RPCGEN jest poprawny, powstają kody źródłowe programów pieńka klienta i serwera, pliki nagłówkowe oraz pliki z filtrami XDR [3, 4].

nrcomp	*funcname*	*p1_1*	...	*p1_n*	...	*pn_1*	...	*pn_n*

nrcomp – nr zdalnego komputera,
funcname – nazwa procedury zdalnej,
$p1\_1..p1\_n_i$ – parametry wejściowe procedury zdalnej 1
$p1\_1..p1\_n_i$ – parametry wejściowe procedury zdalnej n,

Rys. 1. Struktura obszaru pamięci dzielonej
Fig. 1. Shared memory structure

5.1. Etapy tworzenia programu klienta

Kolejnym etapem funkcjonowanie modułu jest generowanie pliku zawierającego kod źródłowy programu klienta *wclient.c*. Etap ten wykonywany jest w kilku następujących krokach:

1. W treści programu klienta umieszczane są komendy dołączania plików nagłówkowych systemowych, użytkownika i generowanych przez RPCGEN.
2. Umieszczane są definicje zmiennych globalnych wykorzystanych w programie klienta.
3. Generowana jest funkcja służąca do przygotowania listy komputerów, które mogą być wykorzystane zdalnie i ustalająca komputer lokalny, który pełni rolę komputera sterującego. W celu zapewnienia poprawnego działanie tej funkcji, podczas działania programu równoległego użytkownik musi przygotować plik z konfiguracją sieci o nazwie *servers* o następującej strukturze:

```
SERVER      nazwa komputera zdalnego
SERVER      nazwa komputera zdalnego
...
LOCALCOMP nazwa komputera sterującego
```

Przykład pliku *servers*:

```
SERVER      classic1
SERVER      classic2
LOCALCOMP   sun10
```

Rolą tej funkcji jest zwiększenie przenośności generowanego programu równoległego, poprzez uniezależnienie go od środowiska sieciowego, w którym zostaje uruchomiony. W przypadku uruchomienia programu równoległego w innej sieci, gdzie nazwa komputerów jest inna, wystarczy zmienić zawartość pliku *servers*.

4. Generowane są funkcje zapewniające tworzenie mechanizmów synchronizacji w postaci obszarów pamięci dzielonej (**PD**) oraz semafory zapisu (**SW**) i odczytu (**SR**) do/z **PD**. W celu zapewnienia stabilnej pracy programu równoległego

tworzonych jest tyle obszarów pamięci, ile jest komputerów zdalnych. W celu organizacji dostępu do obszarów **PD** przewidziane są dwa semafory, zapewniające wykonanie operacji zapisu i odczytu do/z obszaru **PD** przez współdziałające programy. Użytkownik ma również możliwość korzystania z mechanizmów sygnałów jako narzędzia do synchronizacji [9,10].

5. Generowana jest funkcja zapewniająca wysyłanie adresów obszarów **PD**, identyfikatorów semaforów i numeru procesu klienta do programu serwera uruchomionego na komputerze lokalnym. Wykonanie tej funkcji jest kluczową fazą podczas działania programu równoległego, gdyż zapewnia otrzymanie wyników procedur zdalnych i wykonanie wszystkich etapów programu równoległego. Podczas jej generowania wzięto pod uwagę konieczność wywołania jej w trybie blokującym.
6. Generowana jest funkcja zapewniająca wysłanie do zdalnych komputerów ich numerów oraz nazwy komputera sterującego. Funkcja ta również działa w trybie blokującym. Nazwa komputera sterującego umożliwia komputerom zdalnym identyfikację tego komputera w celu wysłania wyników obliczeń. Numer komputera zdalnego wykorzystany jest przez program klienta w celu odbioru wyników działania procedury zdalnej wykonywanej na konkretnym zdalnym komputerze.
7. Generowana jest funkcja zapewniająca nawiązywanie komunikacji z wszystkimi zdalnymi komputerami dla wszystkich procedur zdalnych zdefiniowanych przez użytkownika. Procedury te przewidziane są do realizacji obliczeń równoległych.
8. Generowane są kolejne funkcje zapewniające wywołanie procedur zdalnych i przekazywanie im odpowiednich wartości parametrów wejściowych. W celu zapewnienia równoległej pracy programu, procedury zdalne są wywoływane w trybie asynchronicznym (nieblokującym lub rozgłaszania).
9. Odbiór wyników wywoływanych zdalnie procedur jest wykonywany na żądanie użytkownika. W tym celu w kodzie treści programu klienta generowanych jest tyle funkcji odbierających wyniki wywołań zdalnych, ile zostało wywołanych procedur zdalnych. Funkcje te zapewniają połączenia z odpowiednimi obszarami **PD** i odbiór odpowiedniej liczby parametrów zdefiniowanych jako wynik procedury zdalnej. Ponieważ odbiór wyników może się wiązać z ich dalszym przetwarzaniem, umieszczony jest również komentarz dla użytkownika, informujący go o możliwości wpisania odpowiedniego kodu.
10. Po wygenerowaniu funkcji odpowiadających za odbiór wyników, generowana jest funkcja zapewniająca zerwanie połączeń z komputerami zdalnymi i usuwanie wszystkich tworzonych obszarów **PD** i semaforów.

11. Ostatnim etapem podczas tworzenia programu klienta jest generowanie głównej funkcji programu (*main()*), która wywołuje funkcje odpowiadające za wczytanie konfiguracji sieci (etap 3), tworzenie mechanizmów synchronizacji (etap 4), wysyłanie adresów obszarów **PD** i identyfikatorów semaforów (etap 5), wysyłanie numerów komputerów zdalnych i nazwy komputera lokalnego (etap 6) oraz wywołanie funkcji odpowiadającej za nawiązywanie połączenia z komputerami zdalnymi. Następnie umieszczony jest komentarz informujący użytkownika o konieczności wpisania kodu zapewniającego realizację obliczeń zdalnych. Kod ten zależny jest wyłącznie od użytkownika i musi zawierać odwołania do funkcji zapewniających uruchomienie procedur zdalnych (etap 8) i odbiór wyników (etap 9). Ostatnim elementem generowanej głównej funkcji programu jest wywołanie funkcji zapewniającej zakończenie obliczeń zdalnych (etap 10).

5.2. Etapy tworzenia programu serwera

Po wygenerowaniu programu klienta *wclient.c*, zadaniem modułu APPLGEN jest przygotowanie kodu źródłowego programu serwera *wproc.c*. Podobnie jak w przypadku programu klienta, jest on tworzony w następujących etapach:

1. W treści programu serwera umieszczone są komendy dołączania plików nagłówkowych systemowych, użytkownika i generowanych przez RPCGEN.
2. Umieszczone są definicje zmiennych globalnych wykorzystanych w programie serwera, takie jak numer komputera i nazwa komputera lokalnego.
3. Umieszczony jest kod procedury zdalnej potwierdzającej odbiór nazwy komputera lokalnego.
4. Generowane są kody źródłowe procedur zdalnych przeznaczonych do wykonania obliczeń równoległych. Definiowana jest struktura umożliwiająca odbiór parametrów wejściowych procedury oraz zmienne wykorzystywane podczas wysyłania do komputera lokalnego wyników obliczeń. Zależnie od przeznaczenia procedury, użytkownik musi wprowadzić właściwy kod zapewniający wykonanie obliczeń. W tym celu umieszczony jest odpowiedni komentarz zachęcający. Jako że procedury te mogą wykonywać dowolne zadanie obliczeniowe, nie ma możliwości automatycznego generowania kodu obliczeniowego. Ponieważ procedury te są wykonywane w trybie asynchronicznym, wynik ich działania jest przesyłany do komputera lokalnego za pomocą mechanizmów RPC i zapisywany w odpowiednim obszarze **PD**. W tym celu generowane są instrukcje umożliwiające wywołanie zdalnej usługi na komputerze lokalnym, który staje się dla nich komputerem zdalnym. W tej fazie programu równoległego następuje zamiana ról komputerów. Komputer lokalny

staje się serwerem, a komputery zdalne zmieniają się w klienta. Rolą nowego serwera jest odbiór wyników procedur zdalnych i umieszczenie ich w odpowiednich obszarach **PD**.

5.3. Etapy tworzenia programu drugiego serwera

Po wygenerowaniu programu serwera *wproc.c* zadaniem modułu APPLGEN jest przygotowanie kodu źródłowego programu drugiego serwera *waikproc.c*. Serwer ten, jak już wspomniano, uruchomiony będzie na komputerze sterującym i zapewni odbiór wyników procedur zdalnych i zapisywanie tych wyników do odpowiednich obszarów **PD**. Generowanych jest tyle funkcji, ile jest procedur zdalnych. Program serwera *waikproc.c* tworzony jest w następujących etapach:

1. W treści programu *waikproc.c* umieszczane są komendy dołączania plików nagłówkowych systemowych, użytkownika i generowanych przez RPCGEN.
2. Umieszczane są definicje zmiennych globalnych wykorzystanych w programie serwera do przechowywania adresów obszarów **PD**, identyfikatorów semaforów i numeru procesu programu klienta.
3. Generowany jest kod procedury odbierającej adresy obszarów **PD**, identyfikatory semaforów i numeru procesu klienta oraz zwracającej potwierdzenie ich odbioru.
4. Generowane są kody procedur, których rolą jest: odbiór wyniku odpowiedniej procedury zdalnej, zajęcie semafora zapisu (**SW**) do obszaru **PD**, zapis wyników do obszaru **PD** oraz zwolnienie **SW**. Prawidłowe działanie tych procedur jest warunkiem poprawnego funkcjonowanie programu równoległego. Dlatego wywołane są one w trybie blokującym. Jeśli użytkownik wykorzystuje mechanizmy sygnałów do synchronizacji, procedury te mogą zapewnić budzenie procesu klienta wysyłając mu sygnał pobudki SIGALRM [10].

5.4. Moduły dodatkowe

W celu zapewnienia poprawnego działania programu równoległego opracowano dodatkowo kilka modułów pomocniczych, umożliwiających wykonanie operacji związanych z mechanizmami synchronizacji i obsługą błędów. Stanowią one szereg funkcji bibliotecznych, które można wykorzystać również w innych programach.

Pierwszy moduł *pamoper.c* zapewnia operowanie na obszarach pamięci dzielonej i umożliwia wykonanie takich operacji, jak: tworzenie, kasowanie, odczyt i zapis do obszarów **PD**.

Drugi moduł *semaphs.c* zapewnia operowanie na semaforach binarnych i umożliwia wykonanie takich operacji, jak: tworzenie, usuwanie, zajmowanie i zwolnienie semafora.

Trzeci moduł *charoper.c* umożliwia przetwarzanie zawartości plików tekstowych. Analiza zawartości plików tekstowych ma miejsce podczas sprawdzania pliku użytkownika zawierającego opis procedur zdalnych oraz pliku konfiguracji sieciowej *servers*.

Czwarty moduł *errors.c* zapewnia obsługę błędów i komunikatów informacyjnych użytkownika. Wyróżnia się następujące typy błędów i komunikatów:

FATAL - wystąpienie tego typu błędu oznacza natychmiastowe zakończenie działania programu,

ERROR - błąd tego typu informuje o powstaniu błędu, program może być wykonywany dalej, jeśli zaistniały błąd nie powoduje zakłócenia jego działania,

WARNING - informacja tego typu przestrzega użytkownika przed możliwością wystąpienia błędu,

INFO - komunikat tego typu ma charakter informacyjny,

SUCCESS - komunikat tego typu ma również charakter informacyjny, ułatwia monitorowanie działania programu.

Do kategorii FATAL można zaliczyć takie zdarzenia, jak: brak możliwości utworzenia obszaru **PD**, niemożność implementacji semafora lub nieudana próba wywołania procedury zdalnej.

Do kategorii ERROR można zaliczyć takie zdarzenia, jak brak możliwości nawiązywania połączenia z komputerem zdalnym, przepełnienie dysku.

W kategorii WARNING można wyświetlić np. komunikat informujący o zbyt dużej liczbie zdalnych komputerów, co może powodować niemożność tworzenia odpowiedniej liczby obszarów **PD** itd.

W kategorii INFO można wyświetlić komunikaty informujące o rozpoczęciu/zakończeniu obliczeń, o czasie wykonania procedury zdalnej lub lokalnej itd.

W kategorii SUCCESS można przykładowo wyświetlić komunikaty informujące o udanej próbie wywołania procedury zdalnej, powodzeniu operacji zapisu wyników obliczeń itd.

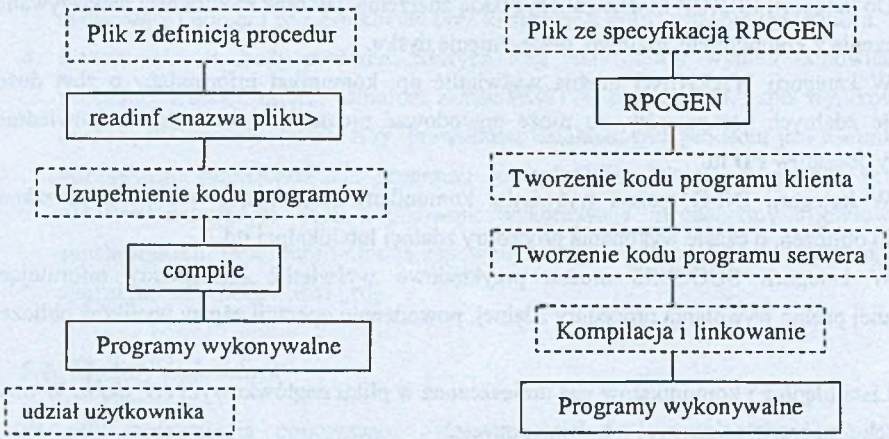
Lista błędów i komunikatów jest umieszczona w pliku nagłówkowym *err_def.h*. W razie potrzeby można ją rozszerzyć o kolejne pozycje.

Kody źródłowe powstałe w wyniku uruchomienia modułu wspomagającego (w wyniku uruchomienia programu *readinf*), po ich uzupełnieniu przez użytkownika we wskazanych miejscach, należy skompilować i linkować wraz z modułami bibliotecznymi oraz modułami generowanymi przez kompilator RPCGEN. Otrzymujemy wtedy kody wykonywalne programów klienta i serwerów. Użytkownik do tego celu może wykorzystać przygotowany skrypt *compile*.

6. Użytkowanie i przykład zastosowania modułu APPLGEN

Z poziomu powłoki systemowej należy uruchomić program *readinf* podając jako parametr nazwę pliku z definicjami procedur zdalnych. W wyniku działania programu powstają kody źródłowe plików ze specyfikacją RPCGEN, kody programów generowanych przez kompilator RPCGEN i kody źródłowe programów klienta i serwerów. Wszystkie te programy wraz z modułami synchronizującymi należy skompilować i linkować wykorzystując wyżej wymieniony skrypt *compile*. Rysunek 2 przedstawia udział użytkownika w generowaniu programu równoległego, przy tradycyjnym wykorzystaniu mechanizmów RPC i modułu *readinf*.

W celu oceny poprawności działania modułu, przedstawimy przykład jego wykorzystania do opracowania równoległego programu mnożenia macierzy kwadratowych. Idea rozwiązania tego zadania polega na podzieleniu jednej z macierzy na odpowiednie podmacierze (wektory) i ich równoległym mnożeniu z drugą macierzą na zdalnych komputerach. Po zakończeniu obliczeń na wszystkich zdalnych komputerach, na komputerze sterującym należy odebrać i wyświetlić wyniki mnożenia. Przyjmijmy dla uproszczenia, że maksymalny stopień macierzy jest 20.



Rys. 2. (a) Tworzenie programu równoległego z wykorzystaniem modułu APPLGEN
(b) Standardowe tworzenie programu równoległego z wykorzystaniem RPC
Fig. 2. (a) Parallel program generation using APPLGEN module
(b) Traditional parallel program generation using RPC

Jak już wspomniano, rolą użytkownika jest utworzenie pliku opisującego procedury zdalne. W naszym przykładzie należy zdefiniować jedną procedurę *matmult*. Opis ten został umieszczony w pliku *matdat.def* o następującej treści:

```
{                                                    /* początek definicji procedury zdalnej */
FNAME     matmult                                    /* nazwa procedury zdalnej */
INPUT     long int    matsize                        /* rozmiar macierzy */
INPUT     long int    colcount   /* liczba kolumn podmacierzy (rozmiar wektora) */
INPUT     float       matdat[400]                    /* wartości elementów macierzy */
INPUT     float       vecdat[200]                    /* wartości elementów podmacierzy */
OUTPUT    double      resvec[200]  /* wynik mnożenia macierzy z wektorem */
}                                                    /* koniec definicji procedury zdalnej */
```

W wyniku wykonania z poziomu systemu komendy: *readinf matdat.def* otrzymujemy kolejno:

1) pliki ze specyfikacją RPCGEN *pwork.x* i *waik.x*,
2) kod programu klienta *wclient.c*,
3) kod programu serwera wykonującego obliczenia zdalne *wproc.c*,
4) kod programu drugiego serwera uruchamianego na komputerze lokalnym *waikproc.c*,

Po uruchomieniu kompilatora RPCGEN kolejno z plikiem *pwork.x* i *waik.x*, otrzymujemy pliki: *pwork.h, pwork_clnt.c, pwork_svc.c, pwork_xdr.c, waik.h, waik_clnt.c, waik_svc.c i waik_xdr.c*.

Kody źródłowe wszystkich generowanych plików umieszczone są w ostatnim rozdziale pracy.

7. Ocena modułu

Opracowany moduł APPLGEN jest użytecznym narzędziem umożliwiającym przyspieszenie procesu tworzenia programów równoległych z wykorzystaniem mechanizmów RPC. W odróżnieniu od innych systemów i narzędzi, nie stanowi on języka programowania ani zbioru funkcji bibliotecznych umożliwiających równoległe wykonywanie zadań. Generuje on skomplikowane fragmenty kodów źródłowych programów klienta i serwera, gdzie wymagana jest znajomość podstawowych funkcji RPC. Nie oznacza to całkowitej eliminacji udziału użytkownika w procesie tworzenia aplikacji równoległej. Do jego obowiązków należy stworzenie prostego pliku z opisem procedur oraz uzupełnienie brakujących fragmentów kodów programów.

Moduł ten nie jest wolny od pewnych niedoskonałości. Narzuca on schemat działania programu równoległego. Opiera się on na dynamicznym przydziale zadań, wymagającym niezależności pomiędzy kolejnymi wykonywanymi zdalnie procedurami. W praktyce istnieje wiele rozwiązań wykorzystujących odmienne schematy działania [2,7, 11].

Mechanizmy synchronizacji wykorzystują obszary pamięci dzielonej (**PD**). Tworzonych jest tyle obszarów, ile jest zdalnych komputerów. W przypadku rozbudowanej sieci i wykorzystania większej liczby komputerów, ograniczenia sprzętowe i systemowe mogą

uniemożliwiać tworzenie wymaganej liczby obszarów **PD**. Sztywna struktura obszaru **PD** składa się z parametrów wyjściowych wszystkich definiowanych procedur, co nie zawsze jest optymalne. Może to być przyczyną przedłużenie czasu dostępu do **PD** i nieefektywnego wykorzystania pamięci komputerów. Generowane pliki programu klienta, serwerów i specyfikacji RPCGEN mają zawsze tę samą nazwę. Jest to pewne ograniczenie podczas wykorzystania modułu do generowania różnych programów. Podobnie jak nazwy programów, procedury zdalne mają zawsze te same numery. Może to powodować konflikt podczas rejestracji różnych usług zdalnych pod tym samym numerem u demona **Portmap** [3]. Ograniczenia te ujawniły się dopiero po wielokrotnym użyciu modułu. Ich uniknięcie nie stanowi dużego problemu.

Podsumowując można stwierdzić, że moduł, mimo pewnych ograniczeń, spełnia wszystkie założenia i realizuje swoją podstawową rolę jako narzędzie ułatwiające pracę programisty korzystającego z mechanizmów zdalnego wywołania procedur.

8. Kody źródłowe generowanych plików

Czcionką *pochyłą* zaznaczono fragment dodatkowego kodu, który użytkownik musi wprowadzić w celu inicjacji obliczeń i inicjalizacji danych początkowych. Czcionką podkreśloną zaznaczono fragment kodu, w którym użytkownik może przetwarzać wyniki zdalnie wywoływanych procedur. Czcionką **pogrubioną** zaznaczono fragment kodu, który zapewnia wywołanie procedur zdalnych i odbiór wyników. W tym fragmencie muszą być odwołania do funkcji automatycznie generowanych do tego celu.

8.1. Pliki ze specyfikacją RPCGEN *pwork.x* i *waik.x*

```
/* Automatic pwork.x generation file */
typedef struct servinit *sinit;
typedef struct inputs_0 *inp_0;
struct inputs_0 {
                    int matsize;
                    int colcount;
                    float matdat[400];
                    float vecdat[200];
                    };
struct servinit {
                int nrcomp;
                char localcomp[30];
};
program PROGRS_0 {
        version VERSS_0 {
        void MATMULT( inp_0 ) = 1;
        } = 1;
} = 0x21000000;
```

```
/* Automatic work.x generation file */
typedef struct localstr *locstr;
typedef struct outputs_0 *outp_0;
struct localstr {
                    int IdMem[20];
                    int IdSemW;
                    int IdSemR;
                    int NrProc;
                    };
struct outputs_0 {
                    int NrComp;
                    double resvec[200];
                    };
program LOCALSVAR {
        version LOCALSVERS {
        int INITIALS( locstr ) = 1;
                    } = 1;
} = 0x41000000;
```

```
program SERVERSIN {
        version SERVERSVERS {
        int SERIN( sinit ) = 1;
        } = 1;
} = 0x51000000;
```

```
program PROGRC_0 {
        version VERSC_0 {
        int WAIK_0( outp_0 ) = 1;
                } = 1;
} = 0x31000000;
```

8.2. Kod programu klienta wclient.c

```
/* wclient.c automatic client program generation */

#include <stdio.h>
#include <rpc/rpc.h>
#include <signal.h>

#include "pwork.h"
#include "waik.h"
#include "common.h"
#include "globals.h"
#include "err_def.h"

CLIENT *clf0[MAXCOMP];

char servertab[MAXCOMP][MAXLEN], localcomp[MAXLEN];
int rsid, wsid, mats, compcount;
int smid[MAXCOMP], sizetab[MAXCOMP];

void
net_conf()                                  /* Sprawdzenie konfiguracji sieci, etap 3 */
{
 FILE *hinp;
 char line[MAXLEN], *lptr;
 int i=0;
 if( ( hinp = fopen( "servers", "r" ) ) == NULL )
  show_error( FILE_OPEN_ERROR, "servers" );
 (char*)lptr = (char *)calloc( MAXLEN, 1 );
 if( lptr == NULL ) show_error( MEMORY_ALLOC_ERROR );
 while( !feof( hinp ) )                     /* przetwarzanie zawartości pliku servers */
 {
  line[0] = '\0';
  fgets( line, MAXLEN, hinp );
  if( strstr( line, "SERVER" ) )
  {
   line[ strlen( line ) - 1 ] = '\0';
   strcpy( lptr, line );
   skip_word( &lptr );
   skip_spaces( &lptr );
   strcpy( servertab[i], lptr );
   i+=1;
   if( i > MAXCOMP )  { show_error( TOO_M_SER ); i-=1; }
  }
  if( strstr( line, "LOCALCOMP" ) )
  {
   line[ strlen( line ) - 1 ] = '\0';
   strcpy( lptr, line );
   skip_word( &lptr );
   skip_spaces( &lptr );
   strcpy( localcomp, lptr );
  }
```

```
 }
 fclose( hinp );
 free((char *)lptr);
}

void
sync()                                          /* Tworzenie mechanizmów synchronizacji, etap 4 */
{
 int i;

 for( i=0; i<compcount; i++ )
  if( !( smid[i] = tworz( SMKEY + i*100, sizeof( struct memostr )))) show_error( SHARED_MEM_ERROR );
 if( ( wsid = tworzsem( SWKEY ) ) < 0 )
 {
  for( i=0; i<compcount; i++ )
   kasuj( smid[i], sizeof( struct memostr ) );
  show_error( SEM_IMP_ERR );
 }
 zwolnsem( wsid );
 if( ( rsid = tworzsem( SRKEY ) ) < 0 )
 {
  for( i=0; i<compcount; i++ )
   kasuj( smid[i], sizeof( struct memostr ) );
  kassem( wsid );
  show_error( SEM_IMP_ERR );
 }
 for( i=0; i<compcount; i++ )
  printf( "SharedMemId = %d ", smid[i] );
 printf( "WriteSemId = %d ReadSemId = %d\n", wsid, rsid);
}

void
local_init()                    /* Wysłanie adresów obszarów PD i identyfikatory semaforów, etap 5 */
{
 CLIENT *loccl;
 locstr locsend;
 int i, *res;
 (char *)locsend = (char *)malloc( sizeof( struct localstr ) );
 if( locsend == NULL ) show_error( MEMORY_ALLOC_ERROR );
 for( i=0; i<compcount; i++ )
  locsend->IdMem[i] = smid[i];                          /* adresy obszarów pamięci dzielonej*/
 locsend->IdSemW = wsid;                                /* identyfikator semafora zapisu */
 locsend->IdSemR = rsid;                                /* identyfikator semafora odczytu */
 locsend->NrProc = getpid();                            /* numer procesu klienta */
 loccl = clnt_create( localcomp, LOCALSVAR, LOCALSVERS, "tcp" );
 if( loccl == NULL )
 {
  printf( "Serwer Call %s Error while Init, exiting...", localcomp );
  exit( 0 );
 }
 res = initials_1( &locsend, loccl );                   /* wywołanie blokujące */
 if( res == NULL )
 {
  clnt_perror( loccl, localcomp );
  exit( 0 );
 }
 free( (char*)locsend );
 clnt_control( loccl, CLSET_FD_CLOSE, (char *)0 );
 clnt_destroy( loccl );
}
```

```
void
server_init()                    /* Wysłanie nazwy komputera lokalnego i nr komputera zdalnego, etap 6 */
{
 CLIENT *clinit[MAXCOMP];
 sinit sendinit;
 int i, *res;

 (char *)sendinit = (char *)malloc( sizeof( struct servinit ) );
 if( sendinit == NULL ) show_error( MEMORY_ALLOC_ERROR );
 strcpy( sendinit->localcomp, localcomp);
 for( i=0; i<compcount; i++ )
 {
  sendinit->nrcomp = i;
  clinit[i] = clnt_create( servertab[i], SERVERSIN, SERVERSVERS, "tcp" );
  if( clinit[i] == NULL ) show_error( SERVER_C_ERROR);
  res = serin_1( &sendinit, clinit[i] );                    /* wywołanie blokujące */
  if( res == NULL )
  {
   clnt_perror( clinit[i], servertab[i] );
   exit( 0 );
  }
  clnt_control( clinit[i], CLSET_FD_CLOSE, (char *)0 );
  clnt_destroy( clinit[i] );
 }
 free( (char*)sendinit );
}

void
pworkinit()                      /* Nawiązanie połączenia z komputerami zdalnymi, etap 7 */
{
 int i;
 struct timeval tv;
 tv.tv_sec = 0; tv.tv_usec = 0;
 for( i=0; i<compcount; i++ )
 {
  clf0[i] = clnt_create( servertab[i], PROGRS_0, VERSS_0, "tcp" );
  if( clf0[i] == NULL ) show_error( SERVER_C_ERROR );
  clnt_control( clf0[i], CLSET_TIMEOUT, &tv );
 }
}

void
rcall_0( inputp_0, inputp_1, inputp_2, inputp_3, compnr )    /* Wywołanie procedury zdalnej, etap 8 */
int inputp_0;
int inputp_1;
float inputp_2[400];
float inputp_3[200];
int compnr;
{
 int i;
 void *result;
 inp_0 sendinp;
 struct rpc_err err;
 (char *)sendinp = (char *)malloc( sizeof( struct inputs_0 ) );
 if ( sendinp == NULL ) show_error( MEMORY_ALLOC_ERROR );
 sendinp->matsize = inputp_0;
 sendinp->colcount = inputp_1;
 for( i=0; i<inputp_0*inputp_0; i++ )
 {
  sendinp->matdat[i] = inputp_2[i];
```

```
 }
 for( i=0; i<inputp_0*inputp_1; i++ )
 {
  sendinp->vecdat[i] = inputp_3[i];
 }
 result = matmult_1( &sendinp, clf0[compnr] );                    /* Wywołanie nieblokujące */
 if( result == NULL )
 {
  clnt_geterr( clf0[compnr], &err );
  if (err.re_status != RPC_TIMEDOUT )
  {
   clnt_perror( clf0[compnr], servertab[compnr] );
   exit(0);
  }
 }
 free( (char *) sendinp );
}

void
getres_0( compcount )                                    /* Odbiór wyników procedury zdalnej, etap 9 */
int compcount;
{
 int i, j, compnr;
 struct memostr memst;
 char fname[MAXLEN];
 strcpy( fname, "matmult" );
 for( i=0; i<compcount; i++ )
 {
  if( !odbierz( smid[i], &memst, sizeof( struct memostr ) ) ) show_error( MEMO_READ_ERROR);
  if( !strcmp( memst.funcn, fname ) )
  {
   compnr = memst.nrcomp;
   printf( "Result get from function %s\n on computer %d\n", fname, compnr );
   /* PLEASE ENTER YOUR CODE */                              /* Przetwarzanie wyników */
   for( i=0; i<mats*sizetab[i]; i++ )
    printf( "%lf\n", memst.resvec[i] );
   /* CODE ENTERED */
  }
 }
}

void
work_end()      /* Zerwanie połączenia z komputerami zdalnymi i usuwanie obszarów PD i semaforów, etap 10*/
{
 int i;
 for( i=0; i<compcount; i++ )
  kasuj( smid[i], sizeof( struct memostr ) );
 kassem( rsid );
 kassem( wsid );
 for( i=0; i<compcount; i++ )
 {
  clnt_control( clf0[i], CLSET_FD_CLOSE, (char *)0 );
  clnt_destroy( clf0[i] );
 }
 printf( "Work ended \n" );
}

void
pobudka(){}                                              /* Budzenie procesu klienta */
```

```
void
main( argc, argv )                                   /* Funkcja główna programu klienta, etap 11 */
int argc;
char *argv[];
{
 int i, j, k=0;
 int rest, vect;
 int obtained=0, sended=0;
 float *bigmat, *litmat;
 if( argc < 3 ) show_error( BAD_USAGE );
 compcount = atoi( argv[1] );
/*
 signal( SIGALRM, pobudka());     /* Możliwość korzystania z sygnałów do budzenia procesu lokalnego */
*/
/*
 dodatkowy kod związany z specyfikacją zadania wprowadzony przez użytkownika
 (inicjacja zmiennych użytkownika, obliczenie liczby kolumn wektorów, inicjacja wartości
elementów macierzy i wektorów ...)
 */
 mats = atoi( argv[2] );
 for( i=0; i<compcount; i++ ) sizetab[i]=0;
 rest = mats % compcount;
 vect = mats / compcount;
 for( i=0; i<compcount; i++ ) sizetab[i] = vect;
 i=0;
 for( j=0; j<rest; j++)
 {
   sizetab[i] = vect + rest;
   i+=1;
   if( i>compcount) i=0;
 }
/* koniec kodu dodatkowego */
 net_conf();                                                    /* Odwołanie do etapu 3 */
 sync();                                                        /* Odwołanie do etapu 4 */
 local_init();                                                  /* Odwołanie do etapu 5 */
 server_init( compcount );                                      /* Odwołanie do etapu 6 */
 pworkinit();                                                   /* Odwołanie do etapu 7 */
 /* PLEASE ENTER YOUR CODE */      /* Wskazanie miejsca do umieszczenie treści procedury zdalnej */
 (float *)bigmat = (float *)malloc( mats*mats*sizeof( float ) );
 for( i=0; i<mats*mats; i++ ) bigmat[i] = ( float )( i+1 );
 for( i=0; i<compcount; i++ )
 {
  (float *)litmat = (float *)malloc( mats*sizetab[i]*sizeof( float ) );
  for( j=0; j<sizetab[i]*mats; j++ )
  {
   k+=1;
   litmat[j] = j+k;
  }
  rcall_0( mats, sizetab[i], bigmat, litmat, i );               /* Odwołanie do etapu 8 */
  sended+=1;
  free( (float *) litmat );
 }
 free( (float *) bigmat );
 printf( "sended = %d  Obtained = %d \n", sended, obtained );
 while( obtained < sended )
 {
  zajmsem( rsid );                                  /* Korzystanie z mechanizmów synchronizacji */
  getrres_0( compcount );                                       /* Odwołanie do etapu 9 */
  obtained += 1;
  printf( "Obtained %d \n", obtained );
```

```
  zwolnsem( wsid );
 }
 /* END CODE ENTERED */
 work_end();                                                    /* Odwołanie do etapu 10 */
}
```

8.3. Kod programu serwera wykonującego obliczenia zdalne wproc.c

```
/* Remote user procedure call generation */

#include <stdio.h>          /* Dołączanie plików nagłówkowych systemowych, RPC i użytkownika, etap 1 */
#include <rpc/rpc.h>
#include "pwork.h"
#include "waik.h"
#include "common.h"
#include "globals.h"
#include "err_def.h"
int nrcomp;                                                     /* Definiowanie zmiennych globalnych, etap 2 */
char localcomp[MAXLEN];

int
*serin_1( input )                          /* Odbiór numeru komputera i nazwy komputera lokalnego, etap 3 */
sinit *input;
{
 static int res;
 nrcomp = (*input)->nrcomp;
 strcpy( localcomp, (*input)->localcomp );
 res = 1;
 return( &res );                                                /* Potwierdzenie odbioru */
}

void
*matmult_1( input )                                /* Kod procedury wykonującej obliczenia zdalne, etap 4 */
inp_0 *input;
{
 outp_0 output;
 CLIENT *cl;
 int *res;
 /* PLEASE ENTER YOUR CODE */
 int i, j, k, msize, colcount;
 float *mA, *mB, *wskA, *wskB, pom;
 double *wskC, *mW, elewyn;
 msize = (*input)->matsize;
 colcount = (*input)->colcount;
 (float *)mA = (float *)malloc( msize*msize*sizeof(float) );
 (float *)mB = (float *)malloc( msize*colcount*sizeof(float) );
 (double *)mW = (double *)malloc( msize*colcount*sizeof(float) );
 if (mA == NULL || mB == NULL || mW == NULL)
  printf ("Serwer: Brak pamieci \n");

 for( i=0; i<msize*msize; i++ )
  *(mA + i) = (*input)->matdat[i];
 for( i=0; i<msize*colcount; i++ )
 {
  *(mB + i) = (*input)->vecdat[i];
  *(mW + i) = 0;
 }
 for( i=0; i<msize-1; i++ )
```

```
  for( j=i+1; j<msize; j++ )
  {
   pom = *(mA + i*msize + j);
   *(mA + i*msize + j) = *(mA + j*msize + i);
   *(mA + j*msize + i) = pom;
  }
 wskC = mW;
 for( i=0; i<msize; i++)
  for (j=0; j<colcount; j++)
  {
   elewyn = 0;
   wskA = mA + (i * msize);
   wskB = mB + (j * msize);
   for (k=0; k<msize; k++)
    elewyn += *(wskA++) * *(wskB++);
   *(wskC++) = (double)elewyn;
  }

 /*USER CODE END*/
 (char *)output = (char *)malloc( sizeof( struct outputs_0 ));
 if( output == NULL ) show_error( MEMORY_ALLOC_ERROR );
 for( i=0; i<msize*colcount; i++ )
  output->resvec[i] = *(mW + i);                          /* Enter your output values */
 output->NrComp = nrcomp;
 free((float *)mA);
 free((float *)mB);
 free((double *)mW);

 cl = clnt_create( localcomp, PROGRC_0, VERSC_0, "tcp" );   /* wysyłanie za pomocą mechanizmów RPC */
 if( cl == NULL )                                           /* wyniku obliczeń do komputera sterującego */
 {
  printf( "Server Call Error %s, exiting...", localcomp );
  show_error( SERVER_C_ERROR );
 }
 res = waik_0_1( &output, cl );      /* wywołanie procedury zdalnej dostępnej na komputerze sterującym */
 if( res == NULL )          /* zamiana roli, komputer zdalny jest klientem, a komputer lokalny serwerem */
 {
  clnt_perror( cl, localcomp );
  exit( 0 );
 }
 free( (char *)output );
 clnt_control( cl, CLSET_FD_CLOSE, (char *)0 );
 clnt_destroy( cl );
 return( NULL );
}
```

8.4. Kod programu drugiego serwera uruchamianego na komputerze lokalnym waikproc.c

```
#include <stdio.h>                  /* Dołączanie plików nagłówkowych systemowych i użytkownika, etap 1*/
#include <rpc/rpc.h>
#include <signal.h>
#include "waik.h"
#include "globals.h"
#include "err_def.h"
```

```
int IdSemR, IdSemW, NrProc, IdMem[20];                    /* Definicja zmiennych globalnych, etap 2 */
int *initials_1( input )                  /* Odbiór adresów obszarów PD i identyfikatory semaforów, etap 3 */
locstr *input;
{
 int i;
 static int res;
 IdSemW = (*input)->IdSemW;
 IdSemR = (*input)->IdSemR;
 NrProc = (*input)->NrProc;
 for( i=0; i<20; i++ )   IdMem[i] = (*input)->IdMem[i];
 res = 1;
 return( &res );                                                    /* Potwierdzenie odbioru */
}

int
*waik_0_1( input )                        /* Zapis do obszarów PD wyniku obliczeń zdalnych, etap 4 */
outp_0 *input;
{
 static int res;
 struct memostr wstr;
 int i;
 for( i=0; i<200; i++ )   wstr.resvec[i] = (*input)->resvec[i];
 strcpy( wstr.funcn, "matmult" );
 wstr.nrcomp = (*input)->NrComp;
 zajmsem( IdSemW );
 if ( !przeslij( IdMem[(*input)->NrComp], &wstr, sizeof( struct memostr )))
  show_error( SHMEMO_WR_ERROR );
 zwolnsem( IdSemR );
 /*
 kill( NrProc, SIGALRM );                          /* Możliwość korzystania z sygnałów - sygnał pobudki */
 */
 res = 1;
 return( &res );
}
```

LITERATURA

1. Środowiska programowania rozproszonego w sieciach komputerowych. Praca zbiorowa pod redakcją K. Zielińskiego, Kraków 1994.
2. Systemy umożliwiające realizację algorytmów równoległych w sieciach komputerowych. Praca zbiorowa pod redakcją S. Kozielskiego. Politechnika Śląska, Skrypty uczelniane nr 1975, Gliwice 1996.
3. Network Programming Guide. Sun Microsystems 1990.
4. Gabassi M., Dupouy B.: Przetwarzanie rozproszone w systemie Unix. LUPUS, Warszawa 1995.
5. Weiss Z., Gruźlewski T.: Programowanie współbieżne i rozproszone. Wydawnictwo Naukowo-Techniczne, Warszawa 1993.
6. http://www.eti.pg.gda.pl/nkowalcz/publikacje/rpc.htm
7. Zghidi H.: Efektywność obliczeń równoległych w sieci komputerowej przy wykorzystaniu mechanizmów zdalnego wywołania procedur. Rozprawa doktorska. Politechnika Śląska, Gliwice 2000.
8. Gabassi M., Dupouy B.: Przetwarzanie rozproszone w systemie Unix. LUPUS, Warszawa 1995.
9. Stevens W. Richard: Programowanie zastosowań sieciowych w systemie Unix. Wydawnictwo Naukowo-Techniczne, Warszawa 1995.
10. Rochkind M. J.: Programowanie w systemie Unix dla zaawansowanych. Wydawnictwo Naukowo-Techniczne, Warszawa 1993.
11. Zghidi H.: Wpływ metody wykorzystania mechanizmów zdalnego wywołania procedur na czas realizacji równoległych zadań. ZN Pol. Śl. s. Informatyka, nr 1356, Gliwice 1997.
12. Rochkind M. J.: Programowanie w systemie Unix dla zaawansowanych. Wydawnictwo Naukowo-Techniczne, Warszawa 1993.

Recenzent: Prof. dr hab. inż. Tadeusz Czachórski

Wpłynęło do Redakcji 30 marca 2001 r.

Abstract

Although Remote Procedure Call (RPC) mechanisms are one of the most efficient tools based on message passing model, they seem to be too hard to use. In comparison to other systems enabling parallel programming, such as PVM, MPI, and Linda they are less popular.

These systems, as independent tools, offer more utilities enabling communication, synchronization and data transfer. One of RPC disadvantages is that additional mechanisms are needed to synchronize parallel program phases.

The presented program is a useful tool that can be used to elaborate parallel programs using RPC mechanisms. This tool generates the most complicated parts of a parallel application and different synchronization mechanisms. This enables programmers to accelerate parallel programs generation and make RPC easier to use.