

SPARC

Napisz fun w asm sparc, która oblicza sumę liczb naturalnych od 1 do a; a >= 1
f(5) = 1 + 2 + 3 + 4 + 5

```
////////// SPARC
```

```
.global fun
```

```
.proc 4
```

```
! %l0 - a
! %l1 - licznik
! %o0 - suma
! cmp kompilowane do subcc
```

```
fun:
```

```
save %sp, -96, %sp
```

```
! przełączenie okien
```

```
mov %i0, %l0
```

```
! przepisujemy do locali, bo tak
```

```
mov 1, %l1
```

```
! wpisujemy 1 do licznika
```

```
mov 0, %o0
```

```
! suma = 0
```

```
petla:
```

```
cmp %l1, %l0
```

```
! porównujemy licznik i a
```

```
bg koniec
```

```
! jeżeli licznik większy to koniec
```

```
nop
```

```
add %o0, %l1, %o0
```

```
! dodajemy do sumy wynik
```

```
add 1, %l1, %l1
```

```
! inkrementujemy licznik
```

```
ba petla
```

```
nop
```

```
koniec:
```

```
ret
```

```
! powrót
```

```
restore
```

```
! przywrócenie stanu okien
```

```
.global fun
```

```
.proc 4
```

```
! %i0 - *tab
! %i1 - rozmiar
! %o0 - suma - później wynik
! %l1 - licznik
! cmp kompilowane do subcc
```

```
fun:
```

```
save %sp, -96, %sp
```

```
! przełączenie okien
```

```
! suma = 0
```

```
mov 0, %o0
```

```
! jeżeli rozmiar = 0 to zwroc 0
```

```
cmp %i1, 0
```

```
be zwroc_zero
```

```
nop
```

```
! for(int i = 0; i < rozmiar; i++)
```

```
petla:
```

```
cmp %i1, %l1
```

```
! sprawdzamy, czy licznik jest równy rozmiar
```

```
bge koniec
```

```
! jeżeli tak to skacz do koniec
```

```
nop
```

```
! if(tab[i] > 0)
```

```
ld [%i0 + %l1], %l0
```

```
! ładujemy dane z tablicy pod adresem: *tab +
```

```
licznik do rejestru %l0
```

```
add %l1, 1, %l1
```

```
! inkrementacja licznika
```

```
cmp %l0, 0
```

```
ble petla
```

```
! jeżeli tab[i] mniejsze od 0 skacz do petli
```

```
nop
```

```
add %l0, %o0
```

```
! jeżeli większe to dodaj do sumy i skok do
```

```
petli
```

```
be petla
```

```
nop
```

```
zwroc_zero:
    mov 0, %o0                ! jesli rozmiar 0 to zwroc 0
    ba wyjdz
    nop

koniec:
    udiv %o0, %i1, %o0        ! suma / rozmiar

wyjdz:
    ret                        ! powrót
    restore                    ! przywrócenie stanu okien
```

Symbol	Argumenty	Komentarz
LD	[adres], rej	Pobranie wartości z pamięci do rejestru. Adres może być pojedynczym rejestrem, sumą dwóch rejestrów, lub sumą rejestru i stałej.
ST	rej, [adres]	Zapis wartości z rejestru do pamięci pod wskazany adres. Adres jak dla instrukcji LD.
MOV	rej1/wart, rej2	Zapisanie stałej lub wartości z rej1 do rejestru rej2
SWAP	[adres], rej	Zamiana wartości rejestru ze wskazaną komórką pamięci
NOP		Nie robi nic
AND ANDcc OR ORcc XOR XORcc XNOR XNORcc	rej1, rej2/wart, rej3	Wykonanie operacji bitowej na podanych argumentach i zapisanie wyniku w trzecim operandzie. Dodanie przyrostka „cc” powoduje oprócz obliczenia wyniku również ustawienie odpowiednich flag w rejestrze stanu (cc = condition codes)
SLL SRL SRA	rej1, rej2/wart, rej3	Przesunięcia logiczne w lewo lub prawo oraz przesunięcie arytmetyczne w prawo
ADD ADDcc ADDX ADDXcc	rej1, rej2/wart, rej3	Dodawanie bez lub z przeniesieniem (X). Ustawienie flag jeśli „cc”.
SUB SUBcc SUBX SUBXcc	rej1, rej2/wart, rej3	Odejmowanie bez lub z przeniesieniem (X). Ustawienie flag jeśli „cc”.
UMUL SMUL UMULcc SMULcc	rej1, rej2/wart, rej3	Mnożenie bez lub ze znakiem.
UDIV SDIV UDIVcc SDIVcc	rej1, rej2/wart, rej3	Dzielenie ze znakiem lub bez znaku.
Bxx	etykieta	Skok warunkowy gdzie xx jest skrótem warunku z języka angielskiego, np.: ba = branch always, bne = branch not equal, bg = branch greater, itd.
CALL	etykieta	Wywołanie podprogramu.
RET RETL		Powrót z podprogramu. Gdy okna rejestrów były przełączone instrukcją SAVE stosujemy rozkaz RET . Jeżeli stan okien jest taki w chwili wywołania CALL , to stosujemy RETL .
SAVE	%sp, wndSize, %sp	Przełączenie okien rejestrów oraz rezerwacja miejsca na stosie na wypadek wyczerpania puli rejestrów procesora. Typowa składnia to: SAVE %sp, -96, %sp
RESTORE		Operacja odwrotna do SAVE . Przywrócenie pierwotnego stanu okien rejestrów.

PVM

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "pvm3.h"

main()
{
    int i, j;
    int tidmaster;
    int ilhost, ilarch;
    struct pvmhostinfo *info;
    int *tid;
    int idp;
    int k;
    int tab[200][3];
    int msgtid;
    srand(time(0));

    for (i = 0; i < 200; i++)                //uzupelnienie tablicy losowymi
wartosciami
    {
        tab[i][0] = rand()%100;
        tab[i][1] = rand()%100;
        tab[i][2] = 0;
    }

    if ((tidmaster = pvm_mytid()) < 0)        //zarejestrowanie procesu
zarzadcy w pvm
    {
        pvm_perror("enroll");
        exit(1);
    }

    pvm_config(&ilhost, &ilarch, &info);      //uzyskanie danych o maszynie pvm

    tid = (int*) calloc (ilhost, sizeof(int)); //rezerwowanie pamieci dla
idyntyfikatorow procesow potomnych

    for (i = 0; i < ilhost; i++)              //rozdzielenie pierwszych zadan
miedzy wszystkimi procesami pochodnymi
    {
                                                                    //powolanie
procesu potomnego
        pvm_spawn("/home/pvm/pvm3/sekcja1234/bin/SUN4SOL2/hello_other", 0, PvmTaskHost,
info[i].hi_name, 1, &tid[i]);
        pvm_initsend(PvmDataDefault);        //inicjalizacja bufora
        pvm_pkint(&i, 1, 1);                  //przeslanie do bufora indeksu
elementow do zsumowania
        pvm_pkint(&tab[i][0], 1, 1);          //przeslanie do bufora pierwszego
elementu do zsumowania
        pvm_pkint(&tab[i][1], 1, 1);          //przeslanie do bufora drugiego
elementu do zsumowania
        pvm_send(tid[i], 100);                //przeslanie danych do
procesu
    }
    while (i < 200)                            //petla wykonuje sie
dopoki sa jeszcze niewyslane dane
    {
        pvm_recv(-1, 200);                    //odebranie wyniku
        pvm_upkint(&k, 1, 1);                  //rozpakowanie indeksu
elementu
        pvm_upkint(&tab[k][2], 1, 1);          //rozpakowanie wyniku
        pvm_upkint(&msgtid, 1, 1);             //rozpakowanie tid, z ktorego
procesu wynik odebrano
        pvm_initsend(PvmDataDefault);        //inicjalizacja bufora
        pvm_pkint(&i, 1, 1);
    }
}
```

```

        pvm_pkint(&tab[i][0], 1, 1);
        pvm_pkint(&tab[i][1], 1, 1);
        pvm_send(msgtid, 100); //przeslanie kolejnych
danych do wolnego procesu pochodnego
    }

    for (i = 0; i < ilhost; i++)
    {
        pvm_recv(-1, 200); //odebranie ostatnich
wynikow
        pvm_upkint(&k, 1, 1);
        pvm_upkint(&tab[k][2], 1, 1);
        pvm_upkint(&msgtid, 1, 1);
    }

    for (i = 0; i < 200; i++)
        printf("%d. %d + %d = %d\n", i+1, tab[i][0], tab[i][1], tab[i][2]); //wypisanie
wynikow

    for (i = 0; i < ilhost; i++)
        pvm_kill(tid[i]); //zniszczenie procesow
pochodnych

    pvm_exit(); //wyrejestrowanie z
pvm
    exit(0);
}

```

```

#include <stdio.h>
#include "pvm3.h"

```

```

main()
{
    int masterid; //tid procesu macierzystego
    int a,b;
    int index;
    int wynik;
    int mytid;
    masterid = pvm_parent(); //pobranie tid procesu macierzystego

    while(1)
    {
        pvm_recv(-1, 100); //odebranie z bufora danych do obliczen
        pvm_upkint(&index, 1, 1); //rozpakowanie indeksu aktualnego elementu
        pvm_upkint(&a, 1, 1); //rozpakowanie pierwszego elementu
        pvm_upkint(&b, 1, 1); //rozpakowanie drugiego elementu

        wynik = a+b; //obliczenie wyniku
        mytid = pvm_mytid(); //pobranie tid aktualnego procesu
        pvm_initSend(PvmDataDefault); //zainicjalizowanie bufora
        pvm_pkint(&index, 1, 1); //spakowanie indeksu wyniku
        pvm_pkint(&wynik, 1, 1); //spakowanie wyniku
        pvm_pkint(&mytid, 1, 1); //spakowanie tid procesu
        pvm_send(masterid, 200); //wyslanie do procesu macierzystego wyniku
    }
}

```

```

int main()
{
    int ilhost, ilarch, info;
    int msgbufid, msglen, msgtag, sender;
    int *tid; //tablica ID hostów

    pvm_config(&ilhost, &ilarch, &info);
    tid = (int*) calloc(ilhost, sizeof(int));
    for (i=0; i<ilhost; ++i)

```

```

{
pvm_spawn(SCIEZKA, 0, PvmTaskHost, info[i].hi_name, 1, &tid[i]);
pvm_initsend(PvmDataDefault);
// jakieś przygotowanie danych, pvm_pkint itp.
pvm_send(tid[i],100);
}
for (;;) // w tej pętli iterujemy po tych danych
// których nie wysłaliśmy wcześniej
{
// odbieramy dane od jakiegoś hosta, który już skończył
// robotę
msgbufid = pvm_recv(-1, 200);
// ostatni parametr, tj. &sender, sprawia, że w zmiennej
// sender mamy ID procesu, od którego odebraliśmy dane
pvm_bufinfo(msgbufid, &msglen, &msgtag, &sender);
// następnie odbieramy dane, czyli pvm_pkuint itp.

// i przygotowujemy nowe dane do hosta
pvm_initsend(PvmDataDefault);
/* pvm_pkint itp. */

// odesłaliśmy do sendera
pvm_send(sender, 100);

}
// z powyższej pętli wychodzimy, gdy wysłaliśmy już wszystkie dane,
// teraz odbieramy resztkę
for (i=0; i<ilhost; ++i)
{
msgbufid = pvm_recv(-1, 200);
pvm_bufinfo(msgbufid, &msglen, &msgtag, &sender);
/* pvm_upkint itp.*/

// sender nie będzie już potrzebny
pvm_kill(sender);
}
// ładnie kończymy sesję z pvm
pvm_exit();
return 0;
}

```

```

int info = pvm_pkbyte (char *cp, int nitem, int stride)
int info = pvm_pkcplx (float *xp, int nitem, int stride)
int info = pvnt_pkdcplx (double *zp, int nitem, int stride)
int info = pvm_pkdouble (double *dp, int nitem, int stride)
int info = pvm_pkfloat (float *fp, int nitem, int stride)
int info = pvm_pkint (int *np, int nitem, int stride)
int info = pvm_pklong (long *np, int nitem, int stride)
int info = pvm_pkshort (short *np, int nitem, int stride)
int info = pvm_pkstr (char *cp)

```

```

int info = pvm_upkbyte (char *cp, int nitem, int stride)
int info = pvm_upkcplx (float *xp, int nitem, int stride)
int info = pvm_upkdcplx (double *zp, int nitem, int stride)
int info = pvm_upkdouble (double *dp, int nitem, int stride)
int info = pvm_upkfloat (float *fp, int nitem, int stride)
int info = pvm_upkint (int *np, int nitem, int stride)
int info = pvm_upklong (long *np, int nitem, int stride)
int info = pvm_upkshort (short *np, int nitem, int stride)
int info = pvm_upkstr (char *cp)

```

JavaSpaces

Producent-konsument

```
public class ProducentKonsument {
    int defaultLease=100000;
    int id=1; //Identyfikator aktualnie produkowanego/konsumowanego wpisu
    //Konstruktor
    public ProducentKonsument()
    {
        //Wyszukanie serwisu typu lookup
        Space.discoverLookupServices(null);
    }
    //Metoda implementująca działanie producenta
    public void producer()
    {
try {

        //Pobranie referencji do przestrzeni JavaSpace
        JavaSpace space=Space.getSpace(spaceName);
        while(true)
        {
            //Wyprodukowanie wpisu
            Data data=produce();
            //Zapisanie wpisu do przestrzeni JavaSpace
            space.write(data,null,defaultLease);
        }
    } catch(Exception ex) //Przechwycenie wyjątków mogących się pojawić }
    podczas
    {
        System.out.println(ex); //wypisanie komunikatu o wyjątku
    }
    //Wyprodukowanie wpisu
    private Data produce()
    {
        Data data=new Data();
        data.desc="Porcja";
        data.id=new Integer(id++);
        return Data;
    }
    //Metoda implementująca proces konsumenta
    public void consumer()
    {
        try {
            //Pobranie referencji do przestrzeni JavaSpace
            JavaSpace space=Space.getSpace(spaceName);
            //Wzorzec wpisu do pobrania
            Data data=new Data();
            int id=1;
            while(true)
            {
                //Pole desc nie będzie dopasowywane co do wartości
                data.desc=null;
                //Pole id będzie dopasowywane również co do wartości
                data.id=new Integer(id++);
                //Pobranie wpisu
                data=(Data)space.take(data,null,defaultLease);
                //Skonsumowanie wpisu
                consume(data);
            }
        }
    }
}
```



```

    }
    } catch(Exception ex)
    {
        System.out.println(ex);
    }
}
//Metoda konsumująca wpis
private void consume(Data data)
{
    System.out.println("Konsumowana "+data.desc+" "+data.id.toString());
}
//Główna funkcja aplikacji
public static void main(String[] args)
{
    ProducentKonsumentt pk=new ProducentKonsument();
    if(args[0].equalsIgnoreCase("producent"))
        pk.producer();
    else if(args[0].equalsIgnoreCase("Konsument"))
        pk.consumer();
}
}

```

Nadzorca-wykonawca

5.2.1 Klasa wpisu z opisem podzadania

```

public class Task implements Entry {
    public Integer parameter; //liczba strzałów do wykonania w ramach zadania
}

```

5.2.2 Klasa wpisu z wynikami podzadania

```

public class Result implements Entry {
    public Integer hits; // liczba trafionych strzałów
    public Integer misses; //liczba nietrafionych strzałów
}

```

5.2.3 Główna klasa aplikacji

```

//Implementacja procesu nadzorcy
public class PIDistApp {
    int defaultLease=100000;
    //Konstruktor
    public PIDistApp()
    {
        //Wyszukanie serwisu typu lookup
        Space.discoverLookupServices(null);
    }
    public void supervisor(int numOfShots,int shotPerTask)
    {
        //pobranie referencji do obiektu przestrzeni JavaSpace
        JavaSpace space=Space.getSpace("spaceName");
        try {
            Task task=new Task(); //wpis z opisem podzadania
            //Rozesłanie podzadań
            for(int shot=0;shot<=numOfShots;shot+=shotPerTask)
            {

```

```

        task.parameter=new Integer((shot+shotPerTask)<=numOfShots
        ?(shotPerTask):(numOfShots-shot));
space.write(task,null,defaultLease);
}
int totalHits=0;
int totalMisses=0;
Result result=new Result();
//zbieranie wyników podzadań
for(int shot=0;shot<=numOfShots;shot+=shotPerTask) {
result.hits=null;
    result.misses=null;
result=(Result)space.take(result,null,defaultLease);
totalHits+=result.hits.intValue();
    totalMisses+=result.misses.intValue();
        System.out.println("Wyniki częściowe: PI:"+
        (4*(double)totalHits/(double)(totalHits+totalMisses)));
}

    System.out.println("Wyniki końcowe: PI:"+
    (4*(double)totalHits/(double)(totalHits+totalMisses)));
task.parameter=null;
//Wysłanie pigułki trującej, informującej procesy wykonawców o zakończeniu obliczeń
space.write(task,null,defaultLease);
} catch(Exception ex) //Przechwycenie ewentualnych wyjątków
{
System.out.println(ex);
}
}
//Implementacja procesu wykonawcy
public void worker()
{
try {
//pobranie referencji do przestrzeni JavaSpace
        JavaSpace space=Space.getSpace("spaceName");
//pobranie referencji do obiektu menadżera transakcji
TransactionManager txnManager=Space.getTransactionManager();
while(true)
{
Task task=new Task(); //wpis z opisem podzadania
task.parameter=null;
//Stworzenie nowej transakcji
        Transaction transaction=TransactionFactory.create(txnManager,
        defaultLease).transaction;

//Pobranie zadamoa
        task=(Task)space.take(task,transaction,defaultLease);

//Jeśli pigułka trująca
        if(task.parameter==null) {
transaction.abort();
return;
}
//wykonanie podzadania
Result result=work(task.parameter);
Zwrócenie rezultatów
space.write(result,transaction,defaultLease);
/Zatwierdzenie transakcji
        transaction.commit();
}
} catch(Exception ex)
{
System.out.println(ex);
}
}

```

```

//Implementacja wykonania podzadania
public Result work(Integer param)
{
    int numOfShots=param.intValue(); //Liczba strzałów w ramach podzadania
    int hits=0; //strzały trafione
    int misses=0; //strzały nietrafione
    Random random=new Random();
    for(int i=0;i<numOfShots;i++)
    {
        // r=1
        double x=random.nextDouble();
        double y=random.nextDouble();
        if((x*x+y*y)<=1)
            hits++;
        else
            misses++;
    }
    Result result=new Result(); //Obiekt wyników
    result.hits=new Integer(hits);
    result.misses=new Integer(misses);
    return result;
}
//Główna metoda aplikacji
public static void main(String[] args)
{
    PIDistApp app=new PIDistApp ();
    if(args[0].equalsIgnoreCase("Nadzorca"))
        app.supervisor(Integer.parseInt(arg[1]), Integer.parseInt(args[2]));
    else if(args[0].equalsIgnoreCase("Wykonawca"))
        app.worker();
}
}

```

Dany jest czarno-biały obraz przechowywany w obiekcie klasy Image. Obiekt ten udostępnia metody:

- public int width() - zwraca szerokość obrazu w pikselach,
- public int height() - zwraca wysokość obrazu w pikselach,
- public Image clip(int x1, int y1, int x2, int y2) - zwraca wycinek obrazka o zadanym rozmiarze,
- public boolean isBlack(int x, int y) - bada, czy piksel o współrzędnych x, y jest czarny (zwraca true) czy biały (zwraca false).

Napisać program wykorzystujący JavaSpaces, który zapewniając odpowiedni podział zadania na mniejsze części, obliczy ilość białych pikseli. Wynik wypisać na ekranie.

Należy zakończyć procesy wykonawców i nie pozostawiać zbędnych krotek w przestrzeni.

Oceniania będzie efektywność rozwiązania.

Założenia:

- zawsze uruchamiany jest najpierw proces nadzorcy
- ilość procesów wykonawców jest nieznana (zawsze jednak > 0)
- typ Image implementuje interfejs java.io.Serializable

Rozwiązanie powinno zawierać:

- deklaracje potrzebnych zmiennych
- ciała funkcji nadzorcy i wykonawcy (nie trzeba definiować konstruktora, ani funkcji main)
- deklaracje klas używanych wpisów (krotek)

- komentarze

Rozwiązanie

```

=====
=====

```

Plik Task.java:

```

import net.jini.space.*;
import net.jini.core.entry.*;
import net.jini.core.lease.*;

```

```

import java.util.*;
import net.jini.core.transaction.server.*;
import net.jini.core.transaction.*;
import java.rmi.*;
/** Klasa zadania */
public class Task implements Entry {
    public Image image;
}
Plik Result.java
import net.jini.space.*;
import net.jini.core.entry.*;
import net.jini.core.lease.*;
import java.util.*;
import net.jini.core.transaction.server.*;
import net.jini.core.transaction.*;
import java.rmi.*;
/** Klasa wyniku */
public class Result implements Entry {
    public Integer res;
}
Plik MainClass.java
import net.jini.space.*;
import net.jini.core.entry.*;
import net.jini.core.lease.*;
import java.util.*;
import net.jini.core.transaction.server.*;
import net.jini.core.transaction.*;
import java.rmi.*;
public class MainClass {
    JavaSpace space;
    int defaultLease=100000;
    // zmienna definiująca ilość podziałów (w % całości)
    private int g = 10;
    public MainClass(String spaceName) {
        Space.discoverLookupServices(null);
        space=Space.getSpace(spaceName);
    }
    /**
     * Proces nadzorcy.
     * 1) jeśli istnieje, usuwa trującą pigułkę z poprzedniego wykonania
     * 2) dzieli obrazek na podzadania zgodnie z pewną założoną granulacją i rozsyła (i zlicza)
     zadania
     * 3) zbiera wyniki podwykonawców, wypisuje wynik globalny,
     * 4) wysyła trującą pigułkę
     */
    public void supervisor() {
        Image obraz = new Image();
        try {
            // 1) usunięcie trującej pigułki, jeśli została po poprzednim wykonaniu
            space.takeIfExists(new Task(), null, defaultLease);
            // 2) Rozesłanie podzadań
            int taskCounter = 0;
            int hx = (int) Math.ceil((obraz.width() * g) / 100);
            int hy = (int) Math.ceil((obraz.height() * g) / 100);
            for(int i = 0; i < obraz.width(); i += hx) {
                for (int j = 0; j < obraz.height(); j += hy) {
                    int x1 = i;
                    int y1 = j;
                    int x2 = i + hx - 1;
                    int y2 = j + hy - 1;
                    if ((obraz.width() - x2) < hx) x2 = obraz.width() - 1;
                    if ((obraz.height() - y2) < hy) y2 = obraz.height() - 1;
                    Task task = new Task(); // wpis z opisem podzadania
                    task.image = obraz.clip(x1, y1, x2, y2);
                    space.write(task, null, defaultLease);
                    taskCounter++;
                }
            }
        }
    }
}

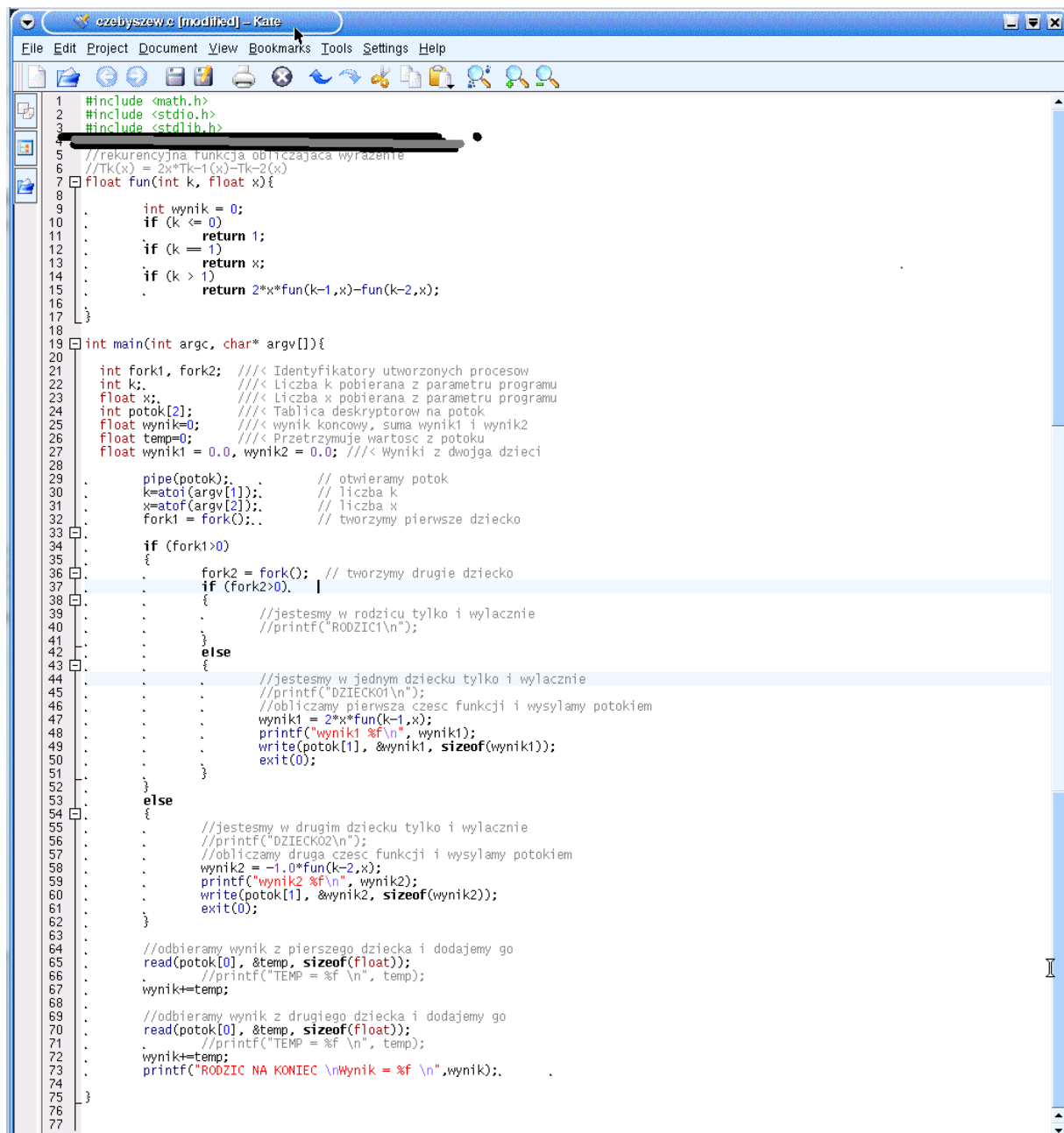
```

```

}
// 3) zbieranie wyników podzadań
int whitePxs = 0;
Result resPattern = new Result();
for(int i = 0; i < taskCounter; i++)
whitePxs += ((Result) space.take(resPattern, null, defaultLease)).res.intValue();
// 4) wysłanie trującej pigułki
Task pp = new Task();
space.write(pp, null, defaultLease);
System.out.println("Znaleziono " + whitePxs + " białych pikseli w obrazku");
} catch (Exception ex) {
System.out.println(ex);
}
}
}
/**
 * Proces wykonawcy
 *
 * 1) pobiera zadanie z przestrzeni
 * 2) sprawdzenie, czy to trująca pigułka. jeśli tak, koniec pracy
 * 3) odczytanie parametrów zadania i zliczenie białych pikseli
 * 4) utworzenie i przesłanie wyniku
 */
public void worker() {
try {
TransactionManager txnManager=Space.getTransactionManager();
while(true) {
// Stworzenie nowej transakcji
Transaction transaction = TransactionFactory.create(txnManager,
defaultLease).transaction;
Task task = new Task();
// 1) Pobranie zadania
task = (Task) space.take(task, transaction, defaultLease);
// 2) Zakoncz prace
if(task.image == null) {
transaction.abort();
return;
}
// 3) Wykonanie podzadania
int pixels = 0;
for (int i = 0; i < task.image.width(); i++)
for (int j = 0; j < task.image.height(); j++)
if (!task.image.isBlack(i, j))
pixels++;
// 4) utworzenie obiektu wyniku
Result result = new Result();
result.res = new Integer(pixels);
// Zwrócenie rezultatów
space.write(result, transaction, defaultLease);
// Zatwierdzenie transakcji
transaction.commit();
}
} catch (Exception ex) {
System.out.println(ex);
}
}
}
/**
 * Metoda uruchamiająca aplikację
 *
 * @param args[1] -- jeśli uruchomiony z N, wtedy nadzorca, jeśli z W, wtedy wykonawca.
 */
public static void main(String[] args) {
MainClass mainClass=new MainClass(args[0]);
if(args[1].equalsIgnoreCase("N"))
mainClass.supervisor();
else if(args[1].equalsIgnoreCase("W"))
mainClass.worker();
}
}

```

MOSIX



```
1 #include <math.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 //rekurencyjna funkcja obliczajaca wyrazenie
6 //Tk(x) = 2xTk-1(x)-Tk-2(x)
7 float fun(int k, float x){
8     int wynik = 0;
9     if (k <= 0)
10         return 1;
11     if (k == 1)
12         return x;
13     if (k > 1)
14         return 2*x*fun(k-1,x)-fun(k-2,x);
15 }
16
17
18
19 int main(int argc, char* argv){
20
21     int fork1, fork2; ///< Identyfikatory utworzonych procesow
22     int k; ///< Liczba k pobierana z parametru programu
23     float x; ///< Liczba x pobierana z parametru programu
24     int potok[2]; ///< Tablica deskryptorow na potok
25     float wynik=0; ///< wynik koncowy, suma wynik1 i wynik2
26     float temp=0; ///< Przechowywanie wartosci z potoku
27     float wynik1 = 0.0, wynik2 = 0.0; ///< Wyniki z dwojga dzieci
28
29     pipe(potok); // otwieramy potok
30     k=atoi(argv[1]); // liczba k
31     x=atof(argv[2]); // liczba x
32     fork1 = fork(); // tworzymy pierwsze dziecko
33
34     if (fork1>0)
35     {
36         fork2 = fork(); // tworzymy drugie dziecko
37         if (fork2>0)
38         {
39             //jestesmy w rodzicu tylko i wylacznie
40             //printf("RODZIC\n");
41         }
42         else
43         {
44             //jestesmy w jednym dziecku tylko i wylacznie
45             //printf("DZIECKO1\n");
46             //obliczamy pierwsza czesc funkcji i wysylamy potokiem
47             wynik1 = 2*x*fun(k-1,x);
48             printf("wynik1 %f\n", wynik1);
49             write(potok[1], &wynik1, sizeof(wynik1));
50             exit(0);
51         }
52     }
53     else
54     {
55         //jestesmy w drugim dziecku tylko i wylacznie
56         //printf("DZIECKO2\n");
57         //obliczamy druga czesc funkcji i wysylamy potokiem
58         wynik2 = -1.0*fun(k-2,x);
59         printf("wynik2 %f\n", wynik2);
60         write(potok[1], &wynik2, sizeof(wynik2));
61         exit(0);
62     }
63
64     //odbieramy wynik z pierwszego dziecka i dodajemy go
65     read(potok[0], &temp, sizeof(float));
66     //printf("TEMP = %f \n", temp);
67     wynik+=temp;
68
69     //odbieramy wynik z drugiego dziecka i dodajemy go
70     read(potok[0], &temp, sizeof(float));
71     //printf("TEMP = %f \n", temp);
72     wynik+=temp;
73     printf("RODZIC NA KONIEC \nWynik = %f \n",wynik);
74 }
75
76
77
```

```

3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <math.h>
6
7  // obliczamy pole pojedynczego trojkata sqrt()*0.5
8  // argumenty: od ktorego i do ktorego ma liczyc
9  double funkcja(int ktory)
10 {
11     double zmienna = 0.5*sqrt(ktory);
12     return zmienna;
13 }
14
15 int main(int argc, char* argv[])
16 {
17     double wynik = 0;
18     double ostateczny = 0;
19     int potok[2];
20     int i;
21     int j;
22     int ilosc;
23     int temp;
24     int forkk;
25     double wynik2 = 0;
26     int n = atoi(argv[1]);
27     int k = atoi(argv[2]);
28     printf("Parametry N:%d, K:%d\n\n", n, k);
29     pipe(potok); //tworzenie potoku
30
31     // ile czesci ma liczyc kazdy proces
32     ilosc = n / k;
33     for(i = 0; i < k; i++)
34     {
35         // tworzenie procesu
36         forkk = fork();
37         if(forkk > 0)
38         {
39             //rodzic
40         }
41         else
42         {
43             //dziecko - oblicza wartosci pol trojkatow
44             // składowych od i*ilosc do (i+1)*ilosc
45             // jezeli w ilosc (n/k) została reszta z dzielenia
46             // sprawdzamy czy jesteśmy w ostatnim przedziale
47             // i obliczamy od i*ilosc do n
48             wynik2 = 0.0;
49             temp = i == k-1 ? n : ((i+1)*ilosc);
50             for(j=i*ilosc; j<temp; j++)
51             {
52                 wynik2 += funkcja(j+1);
53             }
54             write(potok[1], &wynik2, sizeof(wynik2));
55             exit(0);
56         }
57     }
58
59     // zbieranie danych z potoku
60     for(i = 0; i < k; i++)
61     {
62         read(potok[0], &wynik, sizeof(wynik));
63         ostateczny += wynik;
64     }
65     printf("WYNIK: %f", ostateczny);
66
67     return 0;
68 }

```

RPC

9. Przykład zastosowania dla prostego algorytmu

W celu wyjaśnienia zasad działania mechanizmów zdalnego wywołania procedur oraz sposobu ich wykorzystania przedstawimy przykład ich zastosowania do zdalnego obliczenia sumy elementów tablicy (typu całkowitego). Liczba elementów tablicy i liczba wykorzystywanych komputerów są zmienne - podaje je użytkownik jako parametry wejściowe programu.

Rozwiązanie tego algorytmu z wykorzystaniem RPC rozpoczniemy od określenia parametrów wejściowych i wyjściowych zdalnej procedury. Należy jednak podkreślić, że przykład ten nie ilustruje równoległego rozwiązywania zadań, lecz jedynie sposób wykorzystania mechanizmów RPC do wykonania zdalnych procedur.

Parametry wejściowe programu, określające długość tablicy i liczbę zdalnych komputerów są zmienne. Parametrem każdej procedury zdalnej jest tablica (wektor, uporządkowany zbiór liczb) o elementach całkowitych, stanowiąca część tablicy podstawowej. Wynikiem zdalnie wywołanej procedury jest liczba typu całkowitego (int). Zadaniem każdego ze zdalnych komputerów jest obliczenie sumy odpowiedniego fragmentu tablicy podstawowej (podzbioru liczb). Ponieważ zarówno liczba elementów tablicy podstawowej, jak i liczba zdalnie wykorzystywanych komputerów zależne są od użytkownika, to liczba zdalnie sumowanych elementów jest zmienna i należy ją określić na początku działania programu.

Po określeniu parametrów wej/wyj procedury zdalnej należy napisać plik ze specyfikacją RPC, w którym określamy nazwę zdalnie wywołanej procedury, jej parametry wejściowe i wyjściowe, numer procedury oraz numer programu serwera.

```
/*  
** suma.x  
** Plik ze specyfikacją RPC, do zdalnego obliczenia sumy elementów fragmentu tablicy.  
*/
```

```
typedef int tabl<>; /* Tablica o nieokreślonej długości */  
program SUMPROG /* Nazwa programu */  
{  
    version SUMVERS /* Nazwa wersji */  
{  
    int SUM( tabl ) = 1; /* Nawa, nr i typ procedury zdalnej */  
} = 1; /* Nr wersji */  
} = 0x32100118; /* Nr programu */
```

W wyniku wywołania polecenia `rpcgen suma.x` otrzymujemy: plik nagłówkowy `suma.h`, pieniek klienta (`suma_clnt.c`), pieniek serwera (`suma_svc.c`) i filtr XDR (`suma_xdr.c`).

Plik nagłówkowy zawiera definicję funkcji `sum_1`, która w wyniku działania zwraca wskaźnik na liczbę całkowitą. Procedury RPC jako parametr wywołania lub wynik działania, zawsze przekazują adres obiektu, który jest ich parametrem lub wynikiem. Dodatkowo, jak już wspomniano, jako parametry wej/wyj mogą one mieć tylko jeden obiekt. Zadaniem programisty jest teraz opracowanie dwóch pozostałych plików zawierających moduł programu serwera realizującego obliczenia (`suma_proc.c`) i moduł programu klienta (`rsuma.c`).

```
/*  
* suma_proc.c  
* Procedura zdalna do obliczenia sumy elementów tablicy.  
*/  
#include <rpc/rpc.h> /* Dołączenie bibliotek systemowych */  
#include <stdio.h>  
#include "suma.h" /* Dołączenie pliku nagłówkowego */  
int *sum_1( val ) /* Ciało procedury zdalnej */  
    tabl val; /* Parametr wejściowy procedury */  
{  
    /* Definiowanie zmiennej zawierającej wynik obliczeń - musi być zawsze static */  
    static int res;  
    int i;  
    res = 0;  
    for(i=0; i<val.tabl_len; i++)  
        res+=val.tabl_val[i]; /* Sumowanie elementów tablicy */  
    return( &res ); /* Wysyłanie wyniku obliczeń */  
}
```


Moduł programu serwera zawiera ciało procedury zdalnej, wykonującej obliczenie sumy elementów tablicy. Moduł ten należy skompilować i skonsolidować (linkować) z pieńkiem serwera i filtrem XDR. Otrzymujemy wtedy wykonywalny program serwera. W tym celu należy wykonać z poziomu powłoki systemowej komendę:

cc -o sumasvr sumaproc.c suma_svc.c suma_xdr.c

Następnie należy przygotować plik zawierający kod programu klienta (rsuma.c).

```
/*
 * rsuma.c
 * program klienta do obliczenia sumy elementów tablicy
 */
#include <stdio.h>
#include <rpc/rpc.h>
#include <time.h>
#include <sys/timeb.h>
#include "suma.h"
/* Lista komputerów zdalnych */
char *tabkomp[]={ "classic1", "classic2", "classic3", "ipx", "sun10" };
void main (argc, argv)
int argc;
char *argv[];
{
    CLIENT *cl; /* Identyfikator klienta */
    /* Struktura do ustawienia czasu oczekiwania na odpowiedź od serwera */
    struct timeval tv;
    tabl vec; /* Parametr wejściowy procedury zdalnej */
    int *result; /* Wynik procedury zdalnej */
    int dl, lk, ile, reszta; /* Zmienne pomocnicze */
    int i, j, tabrozm[5], suma=0;
    if( argc < 3 )
    {
        printf( "Wywołanie:: rsuma <dlugosc tablicy> <liczba komputerow>\n" );
        exit( 0 );
    }
    dl = atoi( argv[1] ); /* Długość tablicy podstawowej */
    lk = atoi( argv[2] ); /* Liczba zdalnych komputerów */
    if( lk > 5 )
    {
        printf( "Maksymalna liczba komputerow 5\n");
        lk = 5;
    }
    ile = dl / lk;
    reszta = dl % lk;
    for( i=0; i<lk; i++ ) /* Obliczenie liczby elementów */
        tabrozm[i] = ile; /* do sumowania na poszczególnych */
    if( reszta ) /* zdalnych komputerach */
        for( i=0; i<reszta; i++ )
            tabrozm[i] += 1;
    tv.tv_sec = 10; /* Czas oczekiwania na odpowiedź = 10 sekund. */
    tv.tv_usec = 0; /* Po upływie tego czasu i braku odpowiedzi jest zgłaszany błąd */
    for( i=0; i<lk; i++ )
    {
        vec.tabl_len = tabrozm[i]; /* Długość tablicy */
        /* Rezerwowanie pamięci do przechowywania wartości elementów tablicy */
        (int *)vec.tabl_val = (int *)malloc( tabrozm[i]*sizeof(int) );
        /* Inicjowanie danych wejściowych - losowo */
        for( j=0; j<tabrozm[i]; j++ )
            vec.tabl_val[j] = (int)(rand()/50000000);
        /* Próba nawiązywania połączenia z komputerem zdalnym - protokół TCP/IP */
        cl = clnt_create( tabkomp[i], SUMPROC, SUMVERS, "tcp" );
        if (cl == NULL) /* Nie można się łączyć z komputerem zdalnym */

```

```

{
    clnt_pcreateerror( tabkomp[i] ); /* Wyświetlenie komunikatu o błędzie */
    exit( 0 ); /* Zakończ działania programu */
}
clnt_control( cl, CLSET_TIMEOUT, &tv ); /* Ustawienie czasu oczekiwania */
result = sum_1( &vec, cl ); /* Wywołanie procedury zdalnej i odbiór wyniku */
if( result == NULL ) /* Błąd podczas wywołania procedury zdalnej */
{
    clnt_perror( cl, tabkomp[i] ); /* Wyświetlenie komunikatu o błędzie */
    exit( 0 ); /* Zakończ działania programu */
}
/* Wyświetl wynik sumowania z komputera zdalnego */
printf( "Suma komputera %s = %d\n", tabkomp[i], *result );
suma += *result; /* Oblicz sumę wszystkich wyników zdalnych */
clnt_destroy( cl ); /* Usuń identyfikator klienta */
}
printf( "Suma wszystkich elementów tablicy = %d\n", suma );
}

```

Moduł programu klienta zawiera procedury zapewniające przygotowanie wartości elementów tablicy podstawowej, określenie liczby elementów do sumowania przez poszczególne zdalne komputery, połączenie się z komputerami zdalnymi, wysyłanie danych, wywoływanie procedury zdalnej i odbieranie wyników. Moduł ten należy skompilować i linkować z pieńkiem klienta i filtrem XDR. Otrzymujemy wtedy wykonywalny program klienta. W tym celu należy wykonać z poziomu powłoki systemowej komendę:

```
# cc -o rsuma rsuma.c suma_clnt.c suma_xdr.c.
```

W celu użytkowania programu należy uruchomić w tle program serwera na wszystkich zdalnych komputerach, wykonując komendę:

```
# sumasvr &.
```

Po uruchomieniu procesu serwera na komputerach zdalnych, możemy sprawdzić działanie programu wykonując na komputerze lokalnym polecenie:

```
# rsuma 100 4.
```

Następuje sumowanie 100 losowo generowanych elementów tablicy na czterech komputerach (po 25 na każdym zdalnym komputerze).

Powyższy przykład ilustruje możliwość wykorzystania RPC do obliczeń zdalnych w trybie asynchronicznym. Uniemożliwia to wykonanie obliczeń równoległych.

Jednym ze sposobów wykorzystania RPC w trybie synchronicznym są wywołania nieblokujące.

Jak już wspomniano, wynikiem wywołania procedury zdalnej w trybie nieblokującym jest wartość NULL. Oznacza to, że procedury te nie mogą zwracać wyniku ich działania do komputera klienta. Konieczne staje się, więc wykorzystanie dodatkowej procedury zdalnej, która pozwala na otrzymanie wyników wykonywanych obliczeń. Wykorzystanie RPC w trybie synchronicznym pozwalającym na zrównoleglenie obliczeń wymaga, więc wprowadzenie kilku zmian w kodzie programu klienta, serwera i w pliku ze specyfikacją RPCGEN.

```

/*
** suma.x
** Plik ze specyfikacją RPC, do zdalnego obliczenia sumy elementów fragmentu tablicy.
** wersja 2
*/
typedef int tabl<>; /* Tablica o nieokreślonej długości */
program SUMPROG /* Nazwa programu */
{
    version SUMVERS /* Nazwa wersji */
    {
        void SUM( tabl ) = 1; /* Nawa, nr i typ procedury zdalnej */
    } = 1; /* Nr wersji */
} = 0x32100118; /* Nr programu */
program RESPROG /* Nazwa programu */
{
    version RESVERS /* Nazwa wersji */
    {
        int RES( void ) = 1; /* Nazwa, nr i typ dodatkowej procedury zdalnej */
    }
}

```

```

} = 1; /* Nr wersji */
} = 0x32100119; /* Nr programu */
/*
 * suma_proc.c
 * Procedury zdalne do obliczenia sumy elementów tablicy i otrzymania wyników obliczeń.
 */
#include <rpc/rpc.h>
#include <stdio.h>
#include "len.h"
int suma; /* Zmienna globalna do przechowywania wyniku procedury obliczeniowej */
void *sum_1( val )
tabl val;
{
    int i;
    suma = 0;
    for(i=0; i<val.tabl_len; i++)
        suma+=val.tabl_val[i]; /* Wykonanie obliczeń */
    return( NULL ); /* Zwróć wartość NULL */
}
int *res_1() /* Druga procedura zdalna - zwraca wynik obliczeń (zmienna suma) */
{
    static int res; /* Musi być zawsze static */
    res = suma;
    return( &suma );
}
/*
 * rsuma.c
 * program klienta do obliczenia sumy elementów tablicy
 */
#include <stdio.h>
#include <rpc/rpc.h>
#include "len.h"
char *tabkomp[]={ "classic1", "classic2", "classic3", "sun4", "ultra5" };
int main( argc, argv )
int argc;
char *argv[];
{
    CLIENT *cl;
    struct timeval tv;
    tabl vec;
    int i, j, dl, lk, ile, reszta, tabrozm[5], suma=0;
    void *result; /* Wynik pierwszej procedury zdalnej (obliczeniowej) */
    int *gres; /* Wynik drugiej procedury zdalnej (wynik obliczeń pierwszej procedury) */
    struct rpc_err err; /* Struktura kodów błędów RPC */
    if( argc < 3 )
    {
        printf( "Wywołanie:: rsum <dlugosc tablicy> <liczba komputerow>\n" );
        exit( 0 );
    }
    dl = atoi( argv[1] );
    lk = atoi( argv[2] );
    if( lk > 5 )
    {
        printf( "Maksymalna liczba komputerow 5\n" );
        lk = 5;
    }
    ile = dl / lk;
    reszta = dl % lk;
    for( i=0; i<lk; i++ )
        tabrozm[i] = ile;

```

```

if( reszta )
for( i=0; i<reszta; i++ )
tabrozm[i]++;
tv.tv_sec = 0; /* Zerowanie wartość czasu oczekiwania na wynik procedury zdalnej */
tv.tv_usec = 0; /* Wywołanie w trybie nieblokującym */
for( i=0; i<lk; i++ )
{
vec.tabl_len = tabrozm[i];
(int *)vec.tabl_val = (int *)malloc(tabrozm[i]*sizeof(int) );
for( j=0; j<tabrozm[i]; j++ )
vec.tabl_val[j] =(int)(rand())/50000000);
/* Próba nawiązywania połączenia z komputerem zdalnym - protokół TCP/IP */
cl = clnt_create( tabkomp[i], SUMPROG, SUMVERS, "tcp" );
if( cl == NULL ) /* Nie można się łączyć z komputerem zdalnym */
{
clnt_pcreateerror( tabkomp[i] );
exit( 0 );
}
clnt_control( cl, CLSET_TIMEOUT, &tv ); /* Ustawienie czasu oczekiwania */
result = (void *)sum_1( &vec, cl ); /* Wywołanie procedury zdalnej */
if( result == NULL );/* Błąd podczas wywołania procedury zdalnej */
{
clnt_geterr( cl, &err ); /* Sprawdź kod błędu */
if( err.re_status != RPC_TIMEDOUT ) /* Ignoruj błąd TIMEOUT */
{
clnt_perror( cl, tabkomp[i] );
printf( "Server call error \n" );
exit( 0 );
}
}
clnt_destroy( cl );
}
tv.tv_sec = 30; /* Czas oczekiwania na odpowiedź = 30 sekund. */
tv.tv_usec = 0; /* Po upływie tego czasu i braku odpowiedzi jest zgłaszany błąd */
for( i=0; i<lk; i++ )
{
/* Próba nawiązywania połączenia z komputerem zdalnym w celu wywołania drugiej */
/* procedury zdalnej - protokół TCP/IP- */
cl = clnt_create( tabkomp[i], RESPROG, RESVERS, "tcp" );
if( cl == NULL ) /* Nie można się łączyć z serwerem zdalnym */
{
clnt_pcreateerror( tabkomp[i] );
exit( 0 );
}
clnt_control( cl, CLSET_TIMEOUT, &tv ); /* Ustawienie czasu oczekiwania */
gres = res_1( &result, cl ); /* wywołanie procedury zdalnej */
if( gres == NULL ) /* Błąd podczas wywołanie procedury zdalnej */
{
clnt_perror( cl, tabkomp[i] );
printf( "Server call error \n" );
exit( 0 );
}
printf( "Suma komputera %s = %d\n", tabkomp[i], *gres );
suma += *gres; /* Przygotuj wynik końcowy */
clnt_destroy( cl );
}
printf( "Suma wszystkich elementow tablicy = %d\n", suma );
}

```

W celu zapewnienia działanie programu w trybie nieblokującym, należy jeszcze zmienić wartość czasu odpowiedzi w kodzie programu klienta (w tym przypadku plik suma_clnt.c). Polega to na

dodanie następującego wiersza:
static struct timeval MTIMEOUT = {0, 0};
i zmianie ustawienie czasu oczekiwania w procedurach nie blokujących:
z TIMEOUT na MTIMEOUT (również w kodzie programu klienta).

```
/*
**minmax.x
**Plik ze specyfikacją RPC do szukania minimum i maximum w macierzach
*/
typedef float mac<>;
program MINMAX
{
    version MINMAX1
    {
        void MINMAXPROC(mac) = 1;
    } = 0x32100123;
}
program MINMAXRESP
{
    version MINMAXRESP1
    {
        mac MINMAXRESPPROC(void) = 1;
    } = 1;
} = 0x32100124;

client
/*
 * This is sample code generated by rpcgen.
 * These are only templates and you can use them
 * as a guideline for developing your own functions.
 */

#include "minmax.h"
#include <stdio.h>
#include <stdlib.h> /* getenv, exit */

int tab_we_size;
float * tab_we;

void
minmax_1(host,from,size)
    char *host;
    float *from;
    int size;
{
    CLIENT *clnt;
    void *result_1;
    mac minmaxproc_1_arg;
    int i,len=10;

    minmaxproc_1_arg.mac_len=size;
    minmaxproc_1_arg.mac_val = from;

#ifdef DEBUG
```

```

        clnt = clnt_create(host, MINMAX, MINMAX1, "netpath");
        if (clnt == (CLIENT *) NULL) {
            clnt_pcreateerror(host);
            exit(1);
        }
#endif /* DEBUG */

        result_1 = minmaxproc_1(&minmaxproc_1_arg, clnt);
        if (result_1 == (void *) NULL) {
            clnt_perror(clnt, "call failed");
        }
#endif /* DEBUG */
        clnt_destroy(clnt);
#endif /* DEBUG */
    }

void
minmaxresp_1(host)
    char *host;
{
    CLIENT *clnt;
    mac *result_1;
    char * minmaxrespproc_1_arg;

#ifdef DEBUG
    clnt = clnt_create(host, MINMAXRESP, MINMAXRESP1, "netpath");
    if (clnt == (CLIENT *) NULL) {
        clnt_pcreateerror(host);
        exit(1);
    }
#endif /* DEBUG */

    result_1 = minmaxrespproc_1((void *)&minmaxrespproc_1_arg, clnt);
    if (result_1 == (mac *) NULL) {
        clnt_perror(clnt, "call failed");
    }
    else
    {
        printf("min: %f\nmax: %f\n", result_1->mac_val[0], result_1->mac_val[1]);
    }
#ifdef DEBUG
    clnt_destroy(clnt);
#endif /* DEBUG */
}

main(argc, argv)
    int argc;
    char *argv[];
{
    char *host;
    int komputerow;
    int a,b,i;
    float * temp_tab;

    if (argc < 2) {
        printf("usage: %s server_host\n", argv[0]);
        exit(1);
    }

```

```

    }
    host = argv[1];

    tab_we_size=argc-3;
    tab_we = (float*)malloc(tab_we_size*sizeof(float));
    for (i=0;i<tab_we_size;i++)
        tab_we[i]=atof(argv[3+i]);

    komputerow=atoi(argv[2]);

    b=tab_we_size/komputerow;
    for (a=0;a<komputerow-1;a++)
    {
        temp_tab=&tab_we[b*a];
        printf("%d %d\n",temp_tab,b);
        minmax_1(host,temp_tab,b);
    }
    temp_tab=&tab_we[b*(komputerow-1)];
    minmax_1(host,temp_tab,tab_we_size-b*(komputerow-1));
    printf("%d %d\n",temp_tab,tab_we_size-b*(komputerow-1));
    for (a=0;a<komputerow;a++)
        minmaxresp_1(host);

    free(tab_we);
}

```

server

```

/*
 * This is sample code generated by rpcgen.
 * These are only templates and you can use them
 * as a guideline for developing your own functions.
 */

#include "minmax.h"
#include <stdio.h>
#include <stdlib.h> /* getenv, exit */
#include <signal.h>
static float wynik_min,wynik_max;
void *
minmaxproc_1(argp, rqstp)
    mac *argp;
    struct svc_req *rqstp;
{
    static char * result;

    /*
     * insert server code here
     */

    int i = 0;
    float min = argp->mac_val[0],max = argp->mac_val[0];
    for(i = 1; i < argp->mac_len;i++)
    {
        if(min > argp->mac_val[i])
            min = argp->mac_val[i];
        else if(max < argp->mac_val[i])
            max = argp->mac_val[i];
    }
}

```

```

    }
    wynik_min = min;
    wynik_max = max;
    return((void *) &result);
}

mac *
minmaxrespproc_1(argp, rqstp)
    void *argp;
    struct svc_req *rqstp;
{
    static mac result;

    /*
     * insert server code here
     */
    result.mac_len=2;
    result.mac_val=malloc(8);
    result.mac_val[0]=wynik_min;
    result.mac_val[1]=wynik_max;

    return (&result);
}

```