Obliczenia równoległe i rozproszone

Praca zbiorowa pod redakcją

Andrzeja Karbowskiego

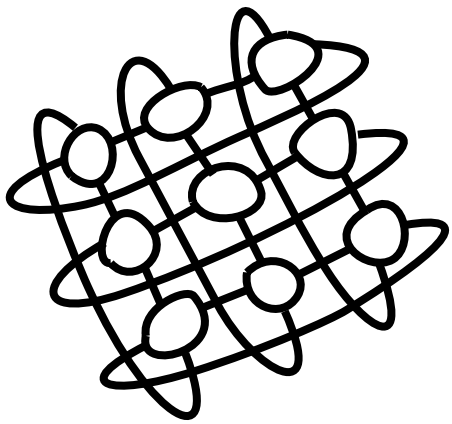
i Ewy Niewiadomskiej-Szynkiewicz

Oficyna Wydawnicza Politechniki Warszawskiej, 2001

Spis treści

Część II

NARZĘDZIA OPROGRAMOWANIA



Rozdział 5

Narzędzia do programowania równoległego na maszynach wieloprocesorowych z pamięcią wspólną

Michał Warchoł, Andrzej Karbowski, Ewa Niewiadomska-Szynkiewicz

5.8.2. Biblioteka PSL do realizacji obliczeń wielowątkowych

Biblioteka PSL (*Parallel Support Library*), występująca na maszynach równoległych z pamięcią wspólną firmy Cray, jest klasycznym narzędziem programowania równoległego wykorzystującym wątki.

Tak, jak w przypadku dyrektyw, liczbę procesorów biorących udział w obliczeniach ustala się za pomocą zmiennej środowiskowej, tutaj nazywa się ona `PARALLEL`. Można, na przykład, wykorzystać polecenie:

```
$ export PARALLEL=<liczba_procesorów>
```

Rozdzielenie sterowania, czyli właściwe zrównoleglenie, następuje poprzez wywołanie procedury

```
call PSL_pcall(sub, nargs, arg1, arg2, ...)
```

gdzie `sub` to nazwa procedury realizowanej przez wątek, `nargs` to liczba jej parametrów formalnych (argumentów), zaś `arg1, arg2, ..., arg<nargs>` to kolejne parametry przekazywane do wątków.

Wcześniej system PSL musi być zainicjowany przez

```
call PSL_init()
```

Jedno wywołanie procedury `PSL_pcall` powołuje taką liczbę wątków, jaką wartość ma zmienna `PARALLEL`. Liczbę uruchamianych wątków można zmienić poprzez wywołanie procedury:

```
call PSL_setcpus(n)
```

Ponadto, można odczytać, ile aktualnie uruchomiłoby się wątków, wywołując funkcję typu `Integer*4 PSL_getcpus()`, oraz ile maksymalnie wątków można uruchomić (tzn. ile jest procesorów) – `PSL_tcpus()`. Procedura wykonywana równolegle ma postać:

```
subroutine sub(lcpu, ncpus, arg1, arg2, ...)
integer*4 lcpu, ncpus
⋮
```

gdzie `lcpu` jest identyfikatorem numerycznym wątku (numerem procesora, na którym jest wykonywany), a `ncpus` liczbą wszystkich uruchomionych instancji procedury `sub`.

W bibliotece PSL zrealizowano trzy mechanizmy synchronizacyjne:

1. Zamek

```
call PSL_bcs(lcpu)
⋮
call PSL_ecs(lcpu)
```

2. Semafor

```
call PSL_psem(id_sem)
⋮
call PSL_vsem(id_sem)
```

gdzie `psem` odpowiada operacji `wait vsem signal`, a `id_sem` jest numerycznym identyfikatorem semafora.

3. Bariera

```
call PSL_barrier(lcpu)
```

Przy pisaniu programów wykorzystujących bibliotekę PSL, tak jak przy wszystkich innych aplikacjach wielowątkowych zawierających wywołania procedur i funkcji standardowych, należy pamiętać o sprawdzeniu, czy są wielowątkowo bezpieczne (ang. *MT-safe*). Chodzi o poprawność działania procedury, niezależnie od kontekstu jej wywołania. Dotyczy to szczególnie procedur z pamięcią, np. generatorów zakłóceń losowych. Jeśli jakaś procedura nie jest MT-safe, to należy zastosować jej odpowiednik spełniający ten warunek.

Przykład 5.12

Przykładowy program w Fortranie 90 korzystający z pamięci wspólnej (zmienne `suma1`, `suma2`), do której dostęp jest chroniony za pomocą zamka.

Program liczy sumę numerów procesorów biorących udział w obliczeniach oraz tę sumę powiększoną o liczbę procesorów, korzystając w pierwszym przypadku jedynie ze zmiennej globalnej, a w drugim również z lokalnych.

Program ten warto uruchomić kilka razy, żeby się przekonać, co jest losowe przy wykonaniu, a co nie (obydwa wyniki końcowe nie powinny być losowe).

pamiec_ws.f

```
module Pamiec_wspolna
    real :: suma1, suma2
end module
!------------------------------------------------------------------------
program test
    Use Pamiec_wspolna
    integer*4 ::  lba_proc, PSL_tcpus, par_we
    external sub

    call PSL_init()
    lba_proc = PSL_tcpus()
    print*, 'liczba procesorow: ',lba_proc
    suma1 = 0.; suma2 = 0.
    par_we = 1.
    call PSL_pcall(sub, 1, par_we)
    print*, ' Suma koncowa_1: ',suma1,' Suma koncowa_2: ', suma2
end
!------------------------------------------------------------------------
subroutine sub(lsub, ncpus, par)
    Use Pamiec_wspolna
    integer*4 :: lsub, ncpus, zlok, par

    zlok = par + lsub
    call PSL_bcs(lsub)
    suma1 = suma1 + lsub
    suma2 = suma2 + zlok

    print*, 'Zglasza sie procesor: ',lsub,', suma_wy_1:  ', &
                           suma1, ' # zm. lok.: ', zlok, &
                           ', suma_wy_2:  ', suma2
    call PSL_ecs(lsub)
end
```

□

Rozdział 6

Narzędzia do programowania na maszynach z pamięcią lokalną oraz w sieciach komputerowych

Michał Warchoł, Andrzej Karbowski, Ewa Niewiadomska-Szynkiewicz

6.3. Pakiet PVM

Pakiet PVM (*Parallel Virtual Machine*) to środowisko oprogramowania do realizacji programów rozproszonych na maszynach wieloprocesorowych z pamięcią lokalną, wspólną[2] oraz w heterogenicznych sieciach maszyn sekwencyjnych i równoległych. Środowisko to zostało opracowane w Oak Ridge National Laboratory. PVM jest narzędziem służącym głównie do obliczeń wykonywanych w sieci maszyn roboczych. Komputery połączone siecią tworzą tzw. równoległą maszynę wirtualną. Poszczególne komputery mogą pracować pod nadzorem różnych systemów operacyjnych, a nawet mieć różne architektury sprzętowe. PVM podczas instalacji „dostosowuje się" do systemu, w którym działa. Oznacza to, że do komunikacji między procesorami na maszynie równoległej wykorzystuje mechanizmy oprogramowania tej maszyny, a nie interfejs z warstwy sieciowej.

Z punktu widzenia programisty, PVM jest biblioteką procedur i funkcji służących głównie do przesyłania komunikatów między procesami. Oprócz funkcji komunikacyjnych dostarcza również wielu funkcji pomocniczych do

[2] W przypadku maszyn z pamięcią wspólną naturalne jest stosowanie mechanizmu wątków, a nie PVM, aczkolwiek istnieją wersje PVM dla takich maszyn.

zarządzania, synchronizacji procesów, itp. PVM ma interfejs dla języków C i Fortran.

W skład systemu PVM wchodzą:

- program konsoli do zarządzania maszyną wirtualną – program `pvm`,
- program demon(y), umożliwiający współdziałanie poszczególnych programów (procesów) składających się na aplikację rozproszoną – plik `pvmd3`,
- biblioteka funkcji podstawowych PVM – zbiór `libpvm3.a`,
- biblioteka funkcji operujących na grupach procesów – zbiór `libgpvm3.a`.

Aby uruchomić aplikację PVM, należy wykonać następujące operacje:

1. Zainstalować pakiet na danym komputerze. Pakiet musi być zainstalowany także na każdym z komputerów wchodzących w skład maszyny wirtualnej. W przypadku używania wspólnego systemu plików, wystarczy jedna instalacja dla każdej z architektur używających go komputerów.
2. Zdefiniować w procesie dwie zmienne środowiska: `PVM_ARCH`, określającą architekturę danego komputera, i `PVM_ROOT`, podającą katalog, w którym pakiet został zainstalowany. W przypadku wspólnego systemu plików, najlepiej jest zdefiniować w pliku `.cshrc` zmienną `PVM_ROOT`, a następnie wywołać (za pomocą komendy `source`) komendy pliku `$PVM_ROOT lib cshrc.stub`, które właściwie ustawią zmienną `PVM_ARCH`.

Po zainstalowaniu pakietu w katalogu `$PVM_ROOT lib/$PVM_ARCH` zostaną umieszczone pliki `libpvm3.a`, `libfpvm3.a` i `libgpvm3.a`, a w katalogu `$PVM_ROOT/bin/$PVM_ARCH` pliki `pvmd3` i `pvmgs`.

Wszystkie informacje dotyczące systemu PVM można znaleźć na stronie `http://www.epm.ornl.gov/pvm/pvm_home.html`.

6.3.1. Maszyna wirtualna

Kluczowym zagadnieniem przy pisaniu aplikacji używających pakietu PVM jest pojęcie maszyny wirtualnej. Jak już było wspomniane, jest to zbiór komputerów, połączonych siecią, na których będą realizowane obliczenia rozproszone. Na każdym komputerze, wchodzącym w skład maszyny wirtualnej, uruchomiony jest program *demon*, który pośredniczy w komunikacji między procesami użytkownika oraz zapewnia automatyczną konwersję danych między różnymi architekturami. Dla PVM, sieć maszyn stanowi jedną – równoległą maszynę. Maszynę wirtualną konfiguruje każdy użytkownik niezależnie. W danej chwili pakiet może być jednocześnie używany przez kilku użytkowników mających różne maszyny wirtualne. Ich procesy nie przeszkadzają sobie wzajemnie.

Procesy pakietu można podzielić na dwie grupy: procesy użytkownika i procesy demonów. Jak już mówiono, demony to procesy działające po

jednym na każdej fizycznej maszynie i integrujące je we wspólną maszynę wirtualną. Jeden z nich – działający na komputerze, na którym uruchomiono pierwszy proces użytkownika – pełni rolę nadrzędną: zarządza maszyną wirtualną, przydziela identyfikatory procesów i zarządza grupami procesów. W chwili dołączenia nowego komputera do maszyny wirtualnej jest na niej uruchamiany demon za pomocą polecenia `rsh`. Ten demon z kolei uruchamia procesy użytkownika.

Przed uruchomieniem aplikacji PVM należy zainicjować maszynę wirtualną – skonfigurować ją i uruchomić. Istnieje na to kilka sposobów. Można uruchomić program konsoli, ręcznie dodając stacje robocze do maszyny wirtualnej. Drugim sposobem jest podanie w linii wywołania pliku konfiguracyjnego. Demony uruchamiają się automatycznie w tle. Wykorzystuje się do tego mechanizm `rsh`. Maszynę wirtualną można modyfikować wywołując takie funkcje, jak: `pvm_addhost` i `pvm_delhost`.

6.3.2. Konsola pakietu

Konsolę PVM uruchamia się za pomocą polecenia

`pvm [`*`hostfile`*`]`

gdzie `hostfile` jest opcjonalną nazwą pliku konfiguracyjnego, zawierającego początkowy skład maszyny wirtualnej. Po rozpoczęciu działania konsola sprawdza, czy na danej maszynie jest uruchomiony proces demona i ewentualnie uruchamia go. Następnie pokazuje znak zachęty

`pvm>`

i czeka na polecenia ze standardowego wejścia. Najczęściej używane komendy to:

`add` *`hostname`* `[`*`hostname`*`]...` – dodaje podane komputery do maszyny wirtualnej;

`delete` *`hostname`* `[`*`hostname`*`]...` – usuwa podane komputery z maszyny wirtualnej; działające w tych komputerach procesy użytkownika są zabijane;

`conf` – wypisuje konfigurację maszyny wirtualnej (nazwy węzłów, identyfikatory procesów demonów, typ architektury i względne prędkości);

`halt` – przerywa działanie wszystkich procesów użytkownika i demonów oraz kończy działanie procesu konsoli;

`quit` – kończy działanie procesu konsoli (pozostałe procesy działają nadal);

`spawn` – uruchamia nowy proces użytkownika na wybranym komputerze wchodzącym w skład maszyny wirtualnej;

`help [nazwa_komendy]` – wyświetla opis podanej komendy wraz z możliwymi argumentami; pominięcie nazwy komendy spowoduje wyświetlenie nazw wszystkich dostępnych komend.

W chwili wywołania konsoli PVM konfiguracja maszyny wirtualnej jest określona przez plik konfiguracyjny podany jako argument. Każda linia tego pliku opisuje jeden komputer. Poza tym dopuszczalne są linie komentarza.

Linie określające komputer zawierają nazwę komputera oraz mogą dodatkowo zawierać opcje o postaci *nazwa_opcji=wartość*. Częściej używane opcje to:

`lo=username` – informuje, jaka jest nazwa użytkownika na danym komputerze;

`so=hasło` – podaje hasło użytkownika na danym komputerze. Opcje `lo` i `so` zwykle występują razem i powodują, że PVM do uruchomienia demona na zdalnym komputerze zamiast usługi `rsh` użyje `rexec`.

`ep=katalogi` – podaje pakietowi katalogi, które należy przeszukiwać w celu uruchomienia procesów użytkownika. Jeśli opcja nie wystąpi, to będzie przeszukiwany tylko katalog `$HOME/pvm3/bin/$PVM_ARCH` (zmienne środowiska są oczywiście rozwijane przez demon na komputerze, na którym ma zostać uruchomiony proces, a nie na komputerze, który wykonuje funkcję `pvm_spawn`);

`sp=liczba` – określa względną szybkość komputera w porównaniu z innymi komputerami, w ramach maszyny wirtualnej. Przy uruchamianiu procesu użytkownika bez określenia konkretnego komputera pakiet sam dokona wyboru, posługując się podanymi szybkościami i liczbą procesów już działających w ramach maszyny wirtualnej.

6.3.3. Tworzenie procesów użytkownika

Procesy użytkownika są uruchamiane przez demona jednego z programów z katalogu `$HOME/pvm3/bin/$PVM_ARCH` lub innego, określonego przez opcję `ep`. Demon tworzy procesy użytkownika realizując komendę konsoli `spawn` lub realizując funkcję `pvm_spawn`, wywołaną przez jeden z już uruchomionych procesów użytkownika.

Aby komunikować się z innymi procesami działającymi w ramach maszyny wirtualnej, proces powinien wywoływać funkcje z biblioteki PVM. Nazwy funkcji tej biblioteki, przeznaczone do użycia w programach napisanych w języku C, mają przedrostek `pvm_`, natomiast procedury przeznaczone do użycia z programów w języku Fortran mają przedrostek `pvmf`. W dalszej części będziemy używać nazw odpowiadających językowi C, chociaż istnieją ich dokładne odpowiedniki dla języka Fortran. W plikach wykorzystujących bibliotekę PVM należy oczywiście dołączyć plik nagłówkowy biblioteki:

```
#include <pvm3.h>
```

Każdy proces użytkownika jest identyfikowany w maszynie wirtualnej za pomocą unikalnego numeru `tid`. Jest on przydzielany w chwili wywołania przez proces dowolnej funkcji PVM. Dopiero od tej chwili proces „istnieje" dla maszyny wirtualnej i można się z nim komunikować. Najczęściej pierwszą funkcją pakietu wywoływaną przez proces użytkownika jest

```
int tid = pvm_mytid(void)
```

zwracająca `tid` wywołującego ją procesu. Identyfikator jest globalny (może być przesłany do innego procesu i tam nadal będzie identyfikował ten sam proces) i może być używany do czasu wywołania przez identyfikowany proces funkcji

```
int info = pvm_exit(void)
```

informującej pakiet, że wywołujący ją proces kończy z nim współpracę. Dalsze użycie `tid`, odpowiadającego takiemu procesowi, będzie powodowało wystąpienie błędu.

Jak wspomniano wcześniej, procesy użytkownika są tworzone na skutek wykonania komendy `spawn` konsoli lub funkcji

```
int numt = pvm_spawn(char *task,
                     char **argv,
                     int  flags,
                     char *where,
                     int  ntasks,
                     int  *tids)
```

gdzie poszczególne argumenty mają następujące znaczenie:

`task` – nazwa programu zawierającego kod procesu;

`argv` – tablica wskazań do poszczególnych argumentów programu (łącznie z nazwą jako pierwszym elementem);

`flags` – zestaw bitów określających opcje; najczęściej przyjmowane wartości to:

- `PvmTaskDefault` – PVM może sam wybrać fizyczną maszynę, na której proces zostanie uruchomiony. W tym przypadku wartość argumentu `where` jest ignorowana,
- `PvmTaskHost` – argument `where` określa nazwę fizycznej maszyny, na której proces zostanie uruchomiony;

`where` – nazwa fizycznej maszyny, na której proces zostanie uruchomiony;

`ntasks` – liczba uruchamianych kopii procesu; nie wszystkie kopie muszą być uruchomione na tej samej maszynie;

`tids` – tablica o rozmiarze `ntasks`, w którą funkcja wpisze identyfikatory uruchomionych procesów. Jeśli przy uruchamianiu któregoś z procesów wystąpił błąd, to odpowiednia pozycja będzie zawierała kod błędu.

Funkcja zwraca liczbę faktycznie uruchomionych procesów. W niektórych przypadkach zwrócona wartość może być ujemna i oznacza wspólny kod błędu dla wszystkich prób utworzenia procesów.

Identyfikator `tid` procesu przodka można uzyskać wywołując funkcję

```
int tid = pvm_parent(void)
```

Gdy przodek nie istnieje, funkcja zwraca wartość `PvmNoParent`.

Proces użytkownika można zakończyć za pomocą funkcji

```
int status = pvm_kill(int tid)
```

która w odróżnieniu od funkcji `pvm_exit()` kończy również działanie procesu. Nie należy jej używać w celu zakończenia aktualnego procesu. W większości implementacji funkcja wysyła do podanego procesu sygnał `SIGTERM`. Zwraca wartość 0 lub kod błędu.

6.3.4. Przesyłanie komunikatów

Najważniejszą usługą udostępnianą przez PVM jest możliwość przesyłania komunikatów. Aby przesłać komunikat, należy wykonać następujące operacje:

1) utworzyć bufor nadawczy,
2) wpisać dane do bufora nadawczego,
3) wysłać treść bufora, nadając mu identyfikator, do jednego lub wielu procesów,
4) zwolnić, ewentualnie, bufor nadawczy,
5) utworzyć bufor odbiorczy,
6) odebrać komunikat do danego bufora odbiorczego,
7) odczytać jego treść (rozpakować go),
8) zwolnić, ewentualnie, bufor odbiorczy.

W procesie, w danej chwili, może istnieć wiele buforów, ale tylko jeden z nich jest aktywnym buforem nadawczym i tylko jeden jest aktywnym buforem odbiorczym. Operują na nich funkcje wpisujące dane do bufora, wysyłające komunikat, odbierające komunikat i rozpakowujące go.

Do tworzenia bufora służy funkcja

```
int bufid = pvm_mkbuf(int encoding)
```

zwracająca identyfikator utworzonego bufora. Argument `encoding` określa sposób kodowania danych umieszczanych w buforze i może przyjmować następujące wartości:

`PvmDataDefault` – kodowanie XDR;
`PvmDataRaw` – brak kodowania;

`PvmDataInPlace` – w buforze są umieszczane jedynie wskaźniki do danych, które są odczytywane dopiero przy wysyłaniu komunikatu. Umożliwia to uniknięcie niepotrzebnego kopiowania danych.

Bufor można zwolnić za pomocą funkcji

```
int status = pvm_freebuf(int bufid)
```

podając identyfikator bufora jako argument `bufid`.

Utworzony bufor nie jest jeszcze aktywny. Należy go uaktywnić wywołując funkcję

```
int oldbuf = pvm_setsbuf(int bufid)
```

ustawiającą bufor o identyfikatorze `bufid` jako aktywny bufor nadawczy, lub funkcję

```
int oldbuf = pvm_setrbuf(int bufid)
```

ustawiającą aktywny bufor odbiorczy. Obie funkcje zwracają identyfikator poprzedniego aktywnego bufora nadawczego/odbiorczego. Identyfikator ten można również uzyskać wołając funkcję

```
int bufid = pvm_getsbuf(void)
```

lub funkcję

```
int bufid = pvm_getrbuf(void)
```

W chwili uruchomienia procesu są w nim automatycznie tworzone po jednym aktywnym buforze nadawczym i odbiorczym (buforów tworzonych automatycznie nie trzeba zwalniać). Tak więc, w większości przypadków, nie ma potrzeby korzystania z opisanych wyżej funkcji. Można stale używać tych samych buforów, utworzonych podczas uruchamiania procesu. Należy jedynie pamiętać, by przed umieszczaniem w buforze nadawczym nowego komunikatu usunąć jego poprzednią zawartość. Służy do tego funkcja

```
int bufid = pvm_initsend(int encoding)
```

W przeciwnym przypadku nowe dane zostaną dołączone do poprzednich. Dodatkowo, funkcja `pvm_initsend` umożliwia zmianę sposobu kodowania danych. Dane z bufora odbiorczego nie muszą być usuwane przed odbiorem komunikatu. Funkcja odbierająca komunikat zrobi to automatycznie.

Jak już wspomniano, w większości aplikacji nie ma potrzeby tworzenia nowych buforów. Jest to konieczne wtedy, gdy chcemy jednocześnie umieszczać dane dla kilku różnych komunikatów oraz gdy tworzymy bibliotekę procedur i nie możemy używać buforów wykorzystywanych przez użytkownika biblioteki.

Każdy bufor może przechować wiele porcji danych. Każda porcja danych jest tablicą danych jednego typu. Do dodawania porcji do bufora służą funkcje:

```
int status = pvm_packf(const char *fmt, ...)

int status = pvm_pkbyte(char *data,
                        int   nitem,
                        int   stride)

int status = pvm_pkcplx(float *data,
                        int   nitem,
                        int   stride)

int status = pvm_pkdcplx(double *data,
                         int    nitem,
                         int    stride)

int status = pvm_pkdouble(double *data,
                          int    nitem,
                          int    stride)

int status = pvm_pkfloat(float *data,
                         int   nitem,
                         int   stride)

int status = pvm_pkint(int *data,
                       int nitem,
                       int stride)

int status = pvm_pkuint(unsigned int *data,
                                int nitem,
                                int stride)

int status = pvm_pkushort(unsigned short *data,
                                  int   nitem,
                                  int   stride)

int status = pvm_pkulong(unsigned long *data,
                                 int  nitem,
                                 int  stride)

int status = pvm_pklong(long *data,
                        int  nitem,
                        int  stride)

int status = pvm_pkshort(short *data,
                         int    nitem,
                         int    stride)

int status = pvm_pkstr(char *sp)
```

Przedstawione funkcje (poza `pvm_packf` i `pvm_pkstr`) umieszczają w aktualnym buforze nadawczym tablicę elementów jednego typu. Znaczenie argumentów jest następujące:

`fmt` – wyrażenie o formacie podobnym do stosowanego przez funkcję `printf`;

`data` – wskazanie na jeden element lub tablicę elementów danego typu. Długość tablicy zależy od wartości argumentów `nitem` i `stride`;

`nitem` – liczba elementów danego typu do umieszczenia w buforze;

`stride` – krok, z jakim kolejne elementy są umieszczane w tablicy `data`;

`sp` – wskaźnik do napisu umieszczanego w buforze.

Funkcja `pvm_pkstr` umieszcza w buforze tekst wskazywany przez argument `sp`. Najbardziej złożona jest funkcja `pvm_packf`. Interpretuje ona kolejne znaki wskazywane przez `fmt` i w zależności od ich znaczenia odczytuje kolejne argumenty wywołania. Argument `fmt` powinien zawierać jedną lub wiele sekwencji

```
%<ilosc><skok><modyfikator><typ>
```

powodujących wstawienie po jednej porcji danych do bufora. Dodatkowo, na początku, może być umieszczona sekwencja `'%+'`, powodująca, że kodowanie poprzedzone zostanie wykonaniem `initsend`. Zinterpretowany wówczas argument powinien mieć typ `int` i powinien określać sposób kodowania. Każdej porcji danych odpowiada przynajmniej jeden argument, określający dane do umieszczenia w buforze. W przypadku pakowania jednego elementu jest to wartość, a w przypadku wielu elementów – wskazanie na tablicę wartości. Fragment `<ilosc>` jest liczbą całkowitą, określającą ilość elementów umieszczanych w buforze. Może też przyjmować wartość `'*'` – wówczas zostanie zinterpretowany jako argument typu `int`, podający liczbę elementów. Podobną postać ma `<skok>`, określający, co ile pozycji kolejne elementy są umieszczane w tablicy, z tym że jest poprzedzony kropką. Fragment `<modyfikator><typ>` określa typ danych do spakowania. Może przyjmować następujące wartości:

`'c'` – pakowanie bajtów (argument określający dane ma typ `char` lub `char*`),

`'d'` – pakowanie liczb całkowitych (argument określający dane ma typ `int` lub `int*`),

`'ud'` – pakowanie liczb całkowitych bez znaku (argument określający dane ma typ `unsigned int` lub `unsigned int*`),

`'hd'` – pakowanie liczb całkowitych krótkich (argument określający dane ma typ `short` lub `short*`),

`'ld'` – pakowanie liczb całkowitych długich (argument określający dane ma typ `long` lub `long*`),

‘uhd’ – pakowanie liczb całkowitych krótkich bez znaku (argument określający dane ma typ `unsigned short` lub `unsigned short*`),
‘uld’ – pakowanie liczb całkowitych długich bez znaku (argument określający dane ma typ `unsigned long` lub `unsigned long*`),
‘f’ – pakowanie liczb rzeczywistych (argument określający dane ma typ `float` lub `float*`),
‘lf’ – pakowanie liczb rzeczywistych podwójnej precyzji (argument określający dane ma typ `double` lub `double*`),
‘x’ – pakowanie liczb zespolonych (argument określający dane ma typ `float*`),
‘lx’ – pakowanie liczb zespolonych podwójnej precyzji (argument określający dane ma typ `double*`),
‘s’ – pakowanie tekstu (argument określający dane ma typ `char*`).
Jeśli pakowanie przebiegnie pomyślnie, zwracana jest wartość 0.

Do wysłania komunikatu najczęściej używa się funkcji

```
int status = pvm_send(int tid,
                      int msgtag)
```

wysyłającej zawartość aktualnego bufora nadawczego do procesu o identyfikatorze `tid`. Komunikat zostaje opatrzony etykietą `msgtag`. Po powrocie z procedury wysyłania komunikatu aktywny bufor można zwolnić lub użyć do umieszczenia w nim nowych danych.

Dany komunikat można także wysłać do wielu procesów za pomocą funkcji

```
int status = pvm_mcast(int *tids,
                       int ntasts,
                       int msgtag)
```

Tablica `tids` o rozmiarze `ntasks` zawiera identyfikatory procesów, które otrzymają komunikat.

Jeśli przesyłany komunikat zawiera tylko jedną porcję danych, to można też użyć funkcji

```
int status = pvm_psend(int   tid,
                       int   msgtag,
                       void  *data,
                       int   count,
                       int   type)
```

Funkcja pakuje `count` danych o adresie początkowym `data` do bufora, a następnie wysyła komunikat. Używa własnego bufora, stąd nie wpływa na inne bufory. Argument `type` określa typ pakowanych danych. Może on przyjmować następujące wartości:
`PVM_STR` – pakowany jest jeden tekst (wartość `count` jest ignorowana),
`PVM_BYTE` – pakowane są bajty,

`PVM_SHORT` – pakowane są liczby całkowite krótkie,
`PVM_INT` – pakowane są liczby całkowite,
`PVM_LONG` – pakowane są liczby całkowite długie,
`PVM_USHORT` – pakowane są liczby całkowite krótkie bez znaku,
`PVM_UINT` – pakowane są liczby całkowite bez znaku,
`PVM_ULONG` – pakowane są liczby całkowite długie bez znaku,
`PVM_FLOAT` – pakowane są liczby rzeczywiste,
`PVM_DOUBLE` – pakowane są liczby rzeczywiste podwójnej precyzji,
`PVM_CPLX` – pakowane są liczby zespolone,
`PVM_DCPLX` – pakowane są liczby zespolone podwójnej precyzji.

Komunikat jest najczęściej odczytywany za pomocą funkcji

```
int bufid = pvm_recv(int tid,
                     int tagmsg)
```

Funkcja czeka na komunikat o etykiecie `tagmsg`, od procesu o identyfikatorze `tid`. Komunikat jest umieszczany w aktywnym buforze odbiorczym i zwracany jest identyfikator tego bufora. Przed umieszczeniem danych bufor jest czyszczony. Funkcja może też odczytywać komunikat od dowolnego procesu (w tym celu należy podstawić `tagmsg = -1`) lub o dowolnej etykiecie (należy wówczas podstawić `tid = -1`).

Funkcja `pvm_recv` ma nieograniczony czas oczekiwania na komunikat. Istnieje jej wersja, o nazwie `pvm_trecv`, oczekująca tylko przez zadany okres czasu, podany jako dodatkowy argument. Jeśli w podanym okresie komunikat nie nadejdzie, zwracana jest wartość zerowa.

Szczególnym przypadkiem funkcji `pvm_trecv` jest funkcja `pvm_nrecv`, która w ogóle nie czeka na komunikat. Jeśli w chwili jej wywołania komunikat już przybył, zostanie on umieszczony w buforze i zostanie zwrócony identyfikator bufora, jeśli zaś nie – zwrócona zostanie wartość zero.

Bardzo podobne działanie do `pvm_nrecv` ma funkcja `pvm_probe`. Nie odczytuje ona jednak komunikatu, a jedynie pobiera jego parametry (nadawcę, etykietę i długość w bajtach). Do odczytu tych parametrów służy funkcja `pvm_bufinfo`.

Przed rozpakowaniem komunikatu należy najpierw wywołać jedną z wcześniej opisanych funkcji, która odczyta komunikat. Do rozpakowywania komunikatów służy zestaw funkcji o analogicznych nazwach do tych, które służą do pakowania komunikatów.

```
int status = pvm_unpackf(const char *fmt, ...)

int status = pvm_upkbyte(char *data,
                         int  nitem,
                         int  stride)
```

```
int status = pvm_upkcplx(float *data,
                         int   nitem,
                         int   stride)

int status = pvm_upkdcplx(double *data,
                          int    nitem,
                          int    stride)

int status = pvm_upkdouble(double *data,
                           int     nitem,
                           int     stride)

int status = pvm_upkfloat(float *data,
                          int   nitem,
                          int   stride)

int status = pvm_upkint(int *data,
                        int nitem,
                        int stride)

int status = pvm_upkuint(unsigned int *data,
                                 int nitem,
                                 int stride)

int status = pvm_upkushort(unsigned short *data,
                                  int   nitem,
                                  int   stride)

int status = pvm_upkulong(unsigned long *data,
                                 int  nitem,
                                 int  stride)

int status = pvm_upklong(long *data,
                         int  nitem,
                         int  stride)

int status = pvm_upkshort(short *data,
                          int   nitem,
                          int   stride)

int status = pvm_upkstr(char *sp)
```

Znaczenie argumentów jest identyczne jak dla funkcji pakujących. Jedyna różnica jest w funkcji `pvm_unpackf`, która nawet w przypadku pojedynczego elementu oczekuje argumentu w postaci wskazania, a nie wartości. Funkcje po udanym rozpakowaniu zwracają wartość zerową.

6.3.5. Grupy procesów

Grupa jest zbiorem procesów, któremu przypisano wspólną nazwę tekstową. Jeden proces może należeć do dowolnej liczby grup. Grupa nie może być pusta – jeśli ostatni proces opuści grupę, grupa przestaje istnieć. Procesy należące do wspólnej grupy są ponumerowane. Numery te nie mają nic wspólnego z identyfikatorami `tid`. Gdy proces jest wstawiany do pewnej grupy, grupa przydziela mu numer. Numery te są odzyskiwane w chwili, gdy proces opuszcza grupę i mogą być przydzielone innym procesom. Jeśli jednak do danej grupy nikt się nie przyłączy, to w numeracji procesów w danej grupie wystąpią luki.

Do wstawienia procesu do grupy służy funkcja

```
int inum = pvm_joingroup(char *group)
```

dołączająca wywołujący ją proces do grupy o nazwie wskazywanej przez argument `group`. Funkcja zwraca numer dołączonego procesu w grupie.

Proces może opuścić grupę, do której należy, wywołując funkcję

```
int status = pvm_lvgroup(char *group)
```

i podając nazwę opuszczanej grupy jako argument `group`.

O fakcie dołączenia się nowego procesu do grupy oraz o fakcie opuszczenia grupy inni członkowie grupy nie są informowani. Mogą jednak uzyskać informacje o członkach grupy w danej chwili. Funkcja

```
int size = pvm_gsize(char *group)
```

zwraca aktualną liczbę członków grupy `group`. Identyfikatory `tid` członków grupy można poznać używając funkcji

```
int tid = pvm_gettid(char* group,
                     int   inum)
```

podając numer procesu w grupie jako argument `inum`. Jeśli numeracja w grupie jest ciągła i członkowie się nie zmieniają, to identyfikatory `tid` wszystkich członków grupy można poznać wywołując funkcję `pvm_gettid` i podając jako argument `inum` kolejne numery od 0 do wartości zwróconej przez funkcję `pvm_gsize` pomniejszonej o jeden. Jeśli w numeracji występują luki, to również można uzyskać indentyfikatory członków. Należy zrezygnować z górnego ograniczenia i zliczać pomyślne wywołania funkcji `pvm_gettid` (w przypadku trafienia w lukę funkcja `pvm_gettid` zasygnalizuje błąd). Jeśli jednak procesy mogą w każdej chwili opuścić grupę i w grupie mogą wystąpić luki w numeracji, to bez dodatkowej komunikacji w chwili dołączania się do grupy i opuszczania grupy nie można poznać jej wszystkich członków.

Grupy zostały zaimplementowane po to, by wykonywać na nich pewne operacje. Najczęściej wykonywaną operacją jest wysłanie komunikatu do wszystkich członków danej grupy. Realizuje to funkcja

```
int status = pvm_bcast(char *group,
                       int  msgtag)
```

przy czym funkcja nie wysyła komunikatu do procesu wywołującego ją, nawet jeśli jest on członkiem grupy `group`.

Innym typem operacji na grupie jest grupowa operacja redukcji. Polega ona na dostarczeniu przez wszystkich członków grupy analogicznych danych (np. wektorów o jednakowych rozmiarach), wykonaniu na tych danych pewnych operacji (np. sumowania) i zwróceniu wyniku do wyróżnionego członka grupy. Do takich operacji służy funkcja

```
int status = pvm_reduce(void  (*func)(),
                        void  *data,
                        int   count,
                        int   datatype,
                        int   msgtag,
                        char  *group,
                        int   root)
```

Powinna być ona wywołana przez wszystkich członków grupy `group`. Argumenty wywołań we wszystkich procesach powinny być jednakowe, za wyjątkiem argumentu `data`, wskazującego na dane, na których ma być wykonana operacja. Typ tych danych określa argument `datatype` o wartościach analogicznych do argumentu `type` funkcji `pvm_psend` (nie można jednak podać typu tekstowego ani liczby bez znaku). Liczbę danych dostarczanych przez każdy z procesów określa argument `count`. Argument `func` określa operację do wykonania. Może przyjmować następujące wartości:
`PvmMin` – wyznaczenie minimum,
`PvmMax` – wyznaczenie maksimum,
`PvmSum` – wyznaczenie sumy,
`PvmProduct` – wyznaczenie iloczynu.

Operacja na danych jest wykonywana na każdej z `count` danych niezależnie (po jednej z każdego procesu), po czym otrzymuje się `count` rezultatów. Rezultaty są wpisywane pod adresem `data` w procesie o numerze w grupie, równym `root`. Argument `msgtag` ma znaczenie pomocnicze, określa etykietę nadawaną komunikatom używanym podczas operacji i powinien być inny niż etykiety innych komunikatów, które nie były jeszcze odebrane.

Na grupie danych można też wykonywać własną operację, podając jako argument `func` wskazanie na funkcję o prototypie

```
void func(int  *datatype,
          void *x,
```

```
            void *y,
            int  *num,
            int  *status)
```

Nieco podobne działanie do `pvm_reduce` ma funkcja

```
int status = pvm_gather(void  *result,
                        void  *data,
                        int   count,
                        int   datatype,
                        int   msgtag,
                        char  *group,
                        int   root)
```

Jedyną różnicą jest to, że na danych nie jest wykonywana żadna operacja, są one natomiast łączone w kolejności wyznaczonej przez rosnące numery procesów w grupie i kopiowane pod adres[3] `result` w procesie o numerze `root`. Argument `result` w innych procesach jest ignorowany.

Przeciwieństwem funkcji `pvm_gather` jest funkcja

```
int status = pvm_scatter(void  *result,
                         void  *data,
                         int   count,
                         int   datatype,
                         int   msgtag,
                         char  *group,
                         int   root)
```

Funkcja ta dzieli wektor o adresie `data`, przekazany przez proces o numerze `root` (tylko w tym procesie argument `data` ma znaczenie), na wiele jednakowych kawałków i kopiuje każdy z nich do obszarów o adresach określonych argumentami `result`. Argument `count` określa liczbę danych w każdym z kawałków.

W bibliotece PVM brak jest funkcji realizujących synchronizację. Wynika to z faktu, iż przy przesyłaniu komunikatów synchronizacja jest zazwyczaj wykonywana automatycznie podczas oczekiwania na komunikat. W niektórych przypadkach warto jednak czekać, aż wszystkie procesy dojdą do pewnego punktu w algorytmie. Realizuje to funkcja

```
int status = pvm_barrier(char *group,
                         int  count)
```

oczekująca aż `count` członków grupy `group` wykona ją. We wszystkich wywołaniach argument `count` musi mieć jednakową wartość.

[3] Kopiowanie do obszaru pamięci wskazywanego przez `result`.

6.3.6. Przykładowe zadanie obliczeniowe – system PVM

W celu prezentacji omówionych mechanizmów programowania równoległego z wykorzystaniem pakietu PVM przedstawimy programy realizujące zadanie obliczeniowe, polegające na rozwiązaniu układu równań liniowych. Zadanie zostało przedstawione w rozdz. 5.7, w którym zamieszczono również kod programów realizujących obliczenia równoległe z użyciem systemu IPC oraz wątków. Różnica polega na tym, że w środowiskach z pamięcią lokalną i-ty procesor (komputer) zmienia swój podwektor x_i, zgodnie z iteracją:

$$x_i := f_i(x_1^i, x_2^i, \ldots, x_{i-1}^i, x_i, x_{i+1}^i, \ldots, x_p^i) \tag{6.1}$$

gdzie x_j^i, $j \neq i$, jest estymatą podwektora j-tego procesora x_j przechowywaną w buforze i-tego procesora, otrzymaną w rezultacie ostatniej transmisji z j do i (tak zwaną fotografią x_j posiadaną przez i-ty procesor). Od czasu tamtej transmisji j-ty procesor mógł oczywiście wykonać wiele innych iteracji typu (6.1) oraz wiele transmisji do innych procesorów.

W rozwiązywaniu zadania, tak samo jak w implementacjach omówionych w rozdz. 5.7, uczestniczą dwa typy procesów:

- Jeden proces inicjujący obliczenia. Zajmuje się wczytaniem danych, dekompozycją problemu, uruchomieniem i inicjalizacją procesów obliczeniowych oraz przechowywaniem wyników pośrednich.
- Wiele procesów obliczeniowych realizujących iteracje algorytmu (6.1).

Realizacja algorytmu – pakiet PVM

W tej wersji rozwiązania wykorzystany został pakiet PVM. Komunikacja między procesami jest realizowana poprzez przesyłanie komunikatów. Ponieważ w omawianej wersji programu procesy nie mają dostępu do wspólnych danych, więc w komunikacji między nimi pośredniczy program inicjujący, który przechowuje aktualną wartość wektora x. Podobnie do wersji, w których do komunikacji wykorzystano pamięć dzieloną, *proces inicjujący* rozpoczyna działanie od odczytania danych, następnie dekomponuje zadanie i inicjuje działanie procesów obliczeniowych. Tablice zawierające dane są w tym przypadku przechowywane w lokalnej pamięci procesu inicjującego. Tak więc ten proces musi aktywnie uczestniczyć w przekazywaniu aktualnych wersji wektora x do procesów obliczeniowych.

Inaczej niż w poprzednim przykładzie, inicjowane są także procesy obliczeniowe. Parametry nie są im przekazywane podczas wywołania. Już po uruchomieniu, proces inicjujący przesyła im komunikaty zawierające liczbę procesów roboczych (`p`), numer bieżącego procesu (`n`) oraz współrzędne obliczanego podwektora wektora x. Po uruchomieniu procesów roboczych i prze-

słaniu im początkowych danych, proces inicjujący obsługuje komunikaty przychodzące od procesów obliczeniowych. Są to:

`UPDATE` – przesłanie przez proces obliczeniowy nowej wersji przetwarzanego podwektora wektora x;

`NEWX` – żądanie przesłania procesowi obliczeniowemu aktualnego wektora x. W tym miejscu można by ograniczyć liczbę przesyłanych danych, przesyłając do procesu obliczeniowego tylko te części wektora x, które zostały zmienione w stosunku do poprzedniej iteracji. W tym celu z każdym komunikatem `UPDATE` proces obliczeniowy powinien przesyłać dodatkowo znacznik czasu, który byłby zapamiętywany przez proces inicjujący. Następnie, w odpowiedzi na żądanie `NEWX`, proces otrzymałby tylko te części wektora, z którymi jest związany znacznik czasowy większy niż jego aktualny.

`STOP_TEST` – proces obliczeniowy przesyła swój znacznik zakończenia obliczeń i otrzymuje w odpowiedzi aktualne wartości znaczników zakończenia innych procesów. Na ich podstawie, podobnie jak w przykładzie omawianym w rozdz. 5.7, podejmowana jest decyzja o zakończeniu obliczeń.

Procesy obliczeniowe są w zasadzie identyczne jak w przykładzie z wykorzystaniem systemu IPC (rozdz. 5.7). Różnią się tylko sposobem pobierania przez nie początkowych danych o sobie i części przetwarzanego zadania oraz sposobem dostępu do danych innych procesorów (aktualnej wartości wektora x oraz znacznika końca obliczeń). Zamiast synchronizowanych semaforami odwołań do pamięci dzielonej przesyłane są odpowiednie komunikaty do procesu inicjującego.

Poniżej przedstawiamy kod programów rozwiązujących nasze zadanie. Zostały one napisane przez mgr. inż. Piotra Bolka z Instytutu Automatyki i Informatyki Stosowanej PW.

defsPVM.h

```
#define MAXSIZE 300
#define XSIZE   sizeof(double)*MAXSIZE
#define ASIZE   XSIZE*MAXSIZE

/* Typy komunikatów */
#define INIT_SIZE       1
#define INIT_B          2
#define INIT_A          10000
#define STOP_TEST       3
#define UPDATE          4
#define NEWX            5
#define END_OF_TASK     6

void freemem();
void getmem();
```

```
void readA();
void readb();
void init();
void work();
```

workPVM.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <pvm3.h>
#include "defsPVM.h"

#define delta_x 0.01              /* dokładność obliczeń        */
#define MAXITER 500               /* maksymalna liczba iteracji */

int     size;                     /* rozmiar zadania                 */
double A[MAXSIZE][MAXSIZE];       /* macierz układu równań           */
double b[MAXSIZE];                /* wektor b                        */
int    *local_stop;               /* znaczniki osiągnięcia lokalnego */
                                  /* testu zatrzymania               */
double D[MAXSIZE];                /* macierz diagonalna D            */
double xl[MAXSIZE];               /* obliczany podwektor wektora x   */
double x[MAXSIZE];                /* lokalna kopia wektora x         */
                                  /* z poprzedniej iteracji          */
int     beg, end;                 /* fragment problemu rozwiązywany  */
                                  /* w bieżącym zadaniu              */
int     p;                        /* liczba procesów                 */
int     n;                        /* numer bieżącego procesu         */
int     mytid, maintid;

int
main(int argc, char **argv)
{
#ifdef DEBUG
    printf("Proces %d (%d,%d)\n", n, beg, end);
#endif
    init();
    work();
    free(local_stop);
    pvm_exit();
    return 0;
}

void
init()
{
    int i, j;

    mytid = pvm_mytid();
    maintid = pvm_parent();
    printf("my = %x, main = %x\n", mytid, maintid);
```

```
    pvm_recv(maintid, INIT_SIZE);
    pvm_upkint(&p, 1, 1);          /* liczba procesów */
    pvm_upkint(&n, 1, 1);          /* numer procesu   */
    pvm_upkint(&beg, 1, 1);
    pvm_upkint(&end, 1, 1);
    pvm_upkint(&size, 1, 1);
    printf("p = %d, n = %d, beg = %d, end = %d, size = %d\n",
                                       p, n, beg, end, size);

    local_stop = malloc (p * sizeof(int));

    pvm_recv(maintid, INIT_B);
    pvm_upkdouble(b, size, 1);
    for(i = beg; i <= end; i++)
    {
        pvm_recv(maintid, INIT_A + i);
        pvm_upkdouble(&A[i][0], size, 1);
    }

#ifdef DEBUG
    for(i = beg; i <= end; i++)
    {
        printf("%d\t", i);
        for(j = 0; j < size; j++)
            printf("%.1f ", A[i][j]);
        printf("\n");
    }
    for(j = 0; j < size; j++)
        printf("%.1f ", b[j]);
    printf("\n");
#endif
}

/*   Obliczenia   */

void work()
{
    int i, j, k;
    int koniec;

    for(i = beg; i <= end; i++)
        D[i] = 1/A[i][i];

/* Odczytanie nowego wektora    */
        pvm_initsend(PvmDataDefault);
        pvm_send(maintid, NEWX);
        pvm_recv(maintid, NEWX);
        pvm_upkdouble(x, size, 1);

        for(k = 0; k < MAXITER; k++)
        {
```

```
                                    /* x_k = A_k * x^{i-1} */
            for(i = beg; i <= end; i++)
            {
                xl[i] = 0;
                for(j = 0; j < size; j++)
                    xl[i] += A[i][j] * x[j];
            }
                                    /* x_k = x_k - b_k */
            for(i = beg; i <= end; i++)
                xl[i] -= b[i];
                                    /* x_k = D_k * x_k */
            for(i = beg; i <= end; i++)
                xl[i] *= D[i];
                                /* x_k^{i} = x_k^{i-1} - x_k */
            for(i = beg; i <= end; i++)
                xl[i] = x[i] - xl[i];

/* Zapisanie nowej wersji przetwarzanego podwektora wektora x */
            pvm_initsend(PvmDataDefault);
            pvm_pkint(&beg, 1, 1);
            pvm_pkint(&end, 1, 1);
            pvm_pkdouble(&xl[beg], end-beg+1, 1);
            pvm_send(maintid, UPDATE);

/* Test zatrzymania -- lokalny.
   Sprawdzenie zmiany przetwarzanego podwektora wektora x */
            koniec = 1;
            for(i = beg; i <= end; i++)
            {
                if(fabs(x[i] - xl[i]) > delta_x)
                {
                    local_stop[n] = 0  /* Test zatrzymania nie */
                                       /* osiągnięty...        */
                    koniec = 0;
                    break;             /* ...dalej             */
                }
            }
            pvm_initsend(PvmDataDefault);
            pvm_pkint(&n, 1, 1);
            pvm_pkint(&local_stop[n], 1, 1);
            pvm_send(maintid, STOP_TEST);
            pvm_recv(maintid, STOP_TEST);
            pvm_upkint(local_stop, p, 1);

/* Odczytanie nowego wektora    */
            pvm_initsend(PvmDataDefault);
            pvm_send(maintid, NEWX);
            pvm_recv(maintid, NEWX);
            pvm_upkdouble(x, size, 1);
/* Jeśli osiągnięty został lokalny test zatrzymania, to
   sprawdzenie, czy inne procesy też osiągnęły test zatrzymania.
```

```
   Jeśli tak -- koniec obliczeń -- znaleziono rozwiązanie.     */
           if(koniec)
           {
               local_stop[n] = 1;
               for(i = 0; i < p; i++)
               {
                 if(!local_stop[i])
                 {                      /* Nie wszystkie procesy    */
                     koniec = 0;        /* osiągnęły lokalny test   */
                     break;             /* zatrzymania -- dalej     */
                 }
               }
           }
           if(koniec) /* Wszystkie procesy osiągnęły test zatrzymania */
               break; /* KONIEC obliczeń!                             */
       }
       pvm_initsend(PvmDataDefault);
       pvm_pkint(&k, 1, 1);
       pvm_send(maintid, END_OF_TASK);
#ifdef DEBUG
       printf("End of %d after %d iterations\n", n, k);
       for(i = beg; i <= end; i++)
           printf("x%d -> %f\n", i, xl[i]);
#endif
}
```

mainPVM.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pvm3.h>
#include "defsPVM.h"

int     size;                   /* rozmiar zadania                */
double  A[MAXSIZE][MAXSIZE];    /* macierz układu                 */
double  b[MAXSIZE];             /* wektor b                       */
double  xo[MAXSIZE];            /* wektor x                       */
int     *local_stop;            /* tablica znaczników osiągnięcia */
                                /* lokalnego testu zatrzymania    */
int     *worker;                /* tablica tid procesów roboczych */
int     mytid;
int     p;                      /* liczba procesów                */
int     debug = 0;

main(int argc, char **argv)
{
    int i;

    if(argc < 2)
        p = 2;
```

```
    else
        p = atoi(argv[1]);
    if(argc == 3)
        debug = 1;
    getmem();
    readA();
    readb();
    init();
    work();
    freemem();
    pvm_exit();
    printf("size = %d ==>\n", size);
    for(i = 0; i < size; i++)
        printf("x%.2d ==> %f\n", i, xo[i]);
    return 0;
}

/* Alokacja tablicy znaczników lokalnych testów zatrzymania
   i tablicy identyfikatorów procesów roboczych             */
void
getmem()
{
    local_stop = (int *) malloc(p * sizeof(int));
    worker = (int *) malloc(p * sizeof(int));

/*   Nikt jeszcze nie skończył obliczeń */
    memset(local_stop, 0, p * sizeof(int));
}

/* Zwolnienie tablicy znaczników lokalnych testów zatrzymania
   i tablicy identyfikatorów procesów roboczych               */
void
freemem()
{
    free(local_stop);
    free(worker);
}

/*   Wczytanie macierzy A   */
void
readA()
{
    int  i, j;
    FILE *afile;

    afile = fopen("A.in", "r");
    fscanf(afile, "%d", &size);
    for(i = 0; i < size; i++)
        for(j = 0; j < size; j++)
            fscanf(afile,"%lf", &A[i][j]);
    fclose(afile);
```

```
/*   Wczytanie wektora b   */
void
readb()
{
    int  i;
    FILE *bfile;

    bfile = fopen("b.in", "r");
    fscanf(bfile, "%d", &size);
    for(i = 0; i < size; i++)
        fscanf(bfile,"%lf", &b[i]);
    fclose(bfile);
}
/* Podział problemu, uruchomienie
                        i inicjalizacja procesów obliczeniowych  */
void
init()
{
    int i, j, k, n;
    int rest;
    int beg, end;
    int result;
#ifdef DEBUG
    printf("size = %d\n", size);
#endif
    mytid = pvm_mytid();
    pvm_catchout(stdout);
/* Podział zadania i uruchomienie procesów roboczych */
    if(debug)
        result = pvm_spawn("work", NULL, PvmTaskDebug, NULL, p, worker);
    else
        result = pvm_spawn("work", NULL, PvmTaskDefault, NULL, p, worker);
    if(result != p)
    {
       printf("error!\n");
       pvm_exit();
       exit(1);
    }
    k = size / p;
    rest = size - k * p;
    for(i = 0; i < p; i++)
    {
        if(i < rest)
        {
            beg = i * (k + 1);
            end = (i + 1) * (k + 1) - 1;
        }
        else
        {
            beg = rest * (k + 1) + (i - rest) * k;
```

```
            end = rest * (k + 1) + (i - rest + 1) * k - 1;
        }
#ifdef DEBUG
        printf("p = %d <%d> (%d,%d)\n", i, end - beg + 1, beg, end);
#endif
        pvm_initsend(PvmDataDefault);
        pvm_pkint(&p, 1, 1);
        pvm_pkint(&i, 1, 1);
        pvm_pkint(&beg, 1, 1);
        pvm_pkint(&end, 1, 1);
        pvm_pkint(&size, 1, 1);
        pvm_send(worker[i], INIT_SIZE);
        pvm_initsend(PvmDataDefault);
        pvm_pkdouble(b, size, 1);
        pvm_send(worker[i], INIT_B);
        for(j = beg; j <= end; j++)
        {
            pvm_initsend(PvmDataDefault);
            pvm_pkdouble(A[j], size, 1);
            pvm_send(worker[i], INIT_A + j);
        }
    }
}

void
work()
{
    int active;
    int bufid;
    int tid;
    int msgtype;
    int bytes;
    int beg, end;
    int process;
    int iter;

    active = p;
    do {
        bufid = pvm_recv(-1,-1);
        pvm_bufinfo(bufid, &bytes, &msgtype, &tid);
        switch(msgtype)
        {
            case UPDATE:
                                    /* printf("UPDATE\n");      */
                pvm_upkint(&beg, 1, 1);
                pvm_upkint(&end, 1, 1);
                pvm_upkdouble(&xo[beg], end - beg + 1, 1);
                break;
            case NEWX:
                                    /* printf("NEWX\n");        */
                pvm_initsend(PvmDataDefault);
```

```
                pvm_pkdouble(xo, size, 1);
                pvm_send(tid, NEWX);
                break;
            case STOP_TEST:
                                    /* printf("STOP_TEST\n");   */
                pvm_upkint(&process, 1, 1);
                pvm_upkint(&local_stop[process], 1, 1);
                pvm_initsend(PvmDataDefault);
                pvm_pkint(local_stop, p, 1);
                pvm_send(tid, STOP_TEST);
                break;
            case END_OF_TASK:
                                    /* printf("END_OF_TASK\n"); */
                pvm_upkint(&iter, 1, 1);
                printf("End of %x after %d iterations\n", tid, iter);
                active--;
                break;
        }
    }while(active);
}
```

6.5. HPF – High Performance Fortran

HPF *High Performance Fortran* – język dyrektyw zrównoleglających dla maszyn z pamięcią lokalną – jest rozszerzeniem Fortranu 90, opracowanym z inicjatywy firmy Digital przez konsorcjum kilkunastu firm, uniwersytetów i laboratoriów badawczych, takich jak: Convex, Cray, Digital, Fujitsu, HP, IBM, Lahey, Intel. Pierwsza wersja HPF została opublikowana w maju 1993 roku.

HPF jest prostszym narzędziem niż języki dyrektyw zrównoleglających dla maszyn z pamięcią wspólną MIC lub OpenMP (patrz rozdz. 5.8.1). Dotyczy on tylko jednego aspektu równoległości – a mianowicie równoległości danych (ang. *data parallelism*). Umożliwia to wykonanie tej samej operacji na różnych częściach złożonych struktur danych, głównie tablic. Z punktu widzenia programisty, programy stosujące zasadę równoległości danych charakteryzują się:

- jednowątkowością,
- globalną przestrzenią nazw (komunikacja między poszczególnymi procesami jest niewidoczna, jest to więc do pewnego stopnia realizacja idei wirtualnej maszyny z pamięcią wspólną),
- synchronicznym wykonywaniem poszczególnych instrukcji, tzn. przed przejściem do następnej instrukcji procesory muszą zakończyć operacje związane ze swoją częścią struktur danych.

Zadaniem programisty jest podzielenie struktur danych i ewentualnie wskazanie kompilatorowi, które fragmenty kodu (np. pętle) mogą być wykonywane równolegle. Cały wysiłek związany z tworzeniem aplikacji równoległej spoczywa na kompilatorze, który tłumaczy kod źródłowy na program typu SPMD (*Single Program Multiple Data*), wykonywany przez poszczególne procesory.

W programie, pisanym przez programistę, wystąpią podstawowe dyrektywy HPF-u realizujące partycję i alokację danych. Są to następujące dyrektywy:

- deklaracja n-wymiarowej tablicy procesorów o nazwie `nazwa_tab_proc`:

  ```
  !HPF$ PROCESSORS nazwa_tab_proc(p1, p2, ..., pn)
  ```

  p_i oznacza tutaj liczbę procesorów wirtualnych wzdłóż i-tej współrzędnej

– rozkład tablic na tablicę procesorów:

```
!HPF$ DISTRIBUTE lista_tablic ONTO nazwa_tab_proc
```

– wyrównanie tablic `tablica1` i `tablica2`:

```
!HPF$ ALIGN tablica1 WITH tablica2
```

Dyrektywa ta służy do wzajemnego ułożenia tablic, tak aby dzieląc między procesory jedną z nich automatycznie uzyskiwać podział i przydział do procesorów części drugiej.
Tablica może być zadeklarowana tradycyjnie, przez podanie typu elementów, rozmiaru oraz nazwy, czyli:

```
real, dimension(n1, n2, ..., nw)  ::  nazwa_tab_danych
```

lub za pomocą wzorca w następujący sposób:

```
!HPF$ TEMPLATE nazwa_wzorca(n1, n2, ..., nw)
```

Zaletą wzorców jest to, że nie wymagają one pamięci operacyjnej.

Pierwsze dwie dyrektywy służą do rozdzielenia jednego lub wielu obiektów na abstrakcyjną tablicę procesorów. Powiązanie procesorów wirtualnych z fizycznymi zależy od implementacji i nie jest określone w języku, tzn. programista nie ma na nie wpływu. W szczególności, procesorów wirtualnych może być więcej lub mniej niż fizycznych (czyli $p_1 \cdot p_2 \cdot \ldots \cdot p_n \neq p$). Liczbę procesorów w maszynie można odczytać za pomocą bezparametrowej funkcji `NUMBER_OF_PROCESSORS()`.

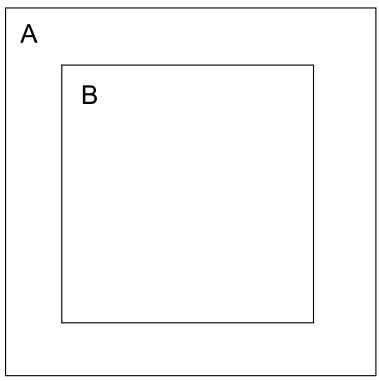
Mamy zatem dwuetapową alokację – na procesory wirtualne (poprzedzoną ewentualnie wyrównaniem) i fizyczne.

Przedstawimy teraz kilka przykładów pokazujących wykorzystanie dyrektyw HPF.

Przykład 6.2

Zdefiniowanie wzajemnego przyporządkowania elementów macierzy A i B

```
real, dimension(16,16) :: A
real, dimension(10,10) :: B
!HPF$ ALIGN B(I,J) WITH A(I + 3, J + 3)
```

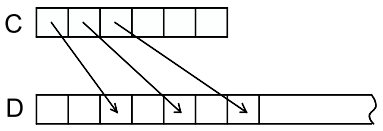


□

Przykład 6.3

Zdefiniowanie wzajemnego przyporządkowania elementów wektorów C i D

```
!HPF$ ALIGN C(I) WITH D(2 * I + 1)
```

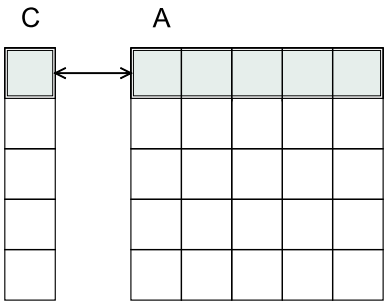


□

Przykład 6.4

Zdefiniowanie wzajemnego przyporządkowania elementów macierzy A i wektora C

```
!HPF$ ALIGN A(:,*) WITH C(:)
```

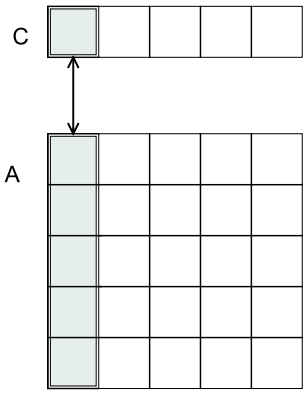


* – oznacza rzutowanie według danego wymiaru. □

Przykład 6.5

Zdefiniowanie wzajemnego przyporządkowania elementów wektora C i macierzy A

```
!HPF$ ALIGN C(:) WITH A(*,:)
```



□

W dyrektywie `DISTRIBUTE`, dla każdej współrzędnej tablicy możliwe są trzy sposoby rozkładu:

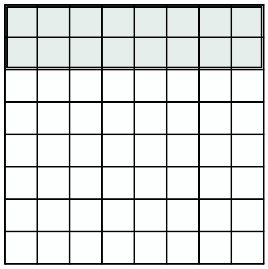
1) `*` – rzutowanie (nie ma rozkładu),
2) `BLOCK(mi)` – rozkład blokowy; m_i oznacza liczbę elementów w spójnym obszarze, według współrzędnej i, przyporządkowanych temu samemu procesorowi wirtualnemu; domyślnie przyjmuje się $m_i = n_i/p_i$,
3) `CYCLIC(mi)` – rozkład cykliczny; domyślnie zakładamy $m_i = 1$.

Przykład 6.6

Przydział elementów tablicy A do 4 procesorów. Zaznaczono część tablicy przypisaną do pierwszego procesora wirtualnego.

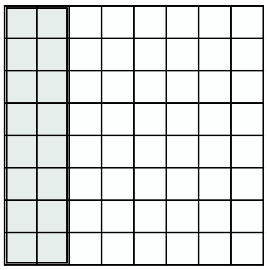
a)

```
real, dimension(8, 8) ::  A
!HPF$ PROCESSORS P(4)
!HPF$ DISTRIBUTE A(BLOCK, *) ONTO P
```



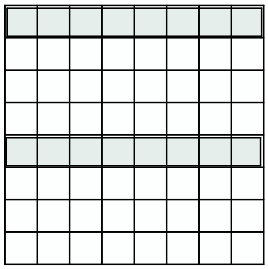
b)

```
real, dimension(8, 8) ::  A
!HPF$ PROCESSORS P(4)
!HPF$ DISTRIBUTE A(*, BLOCK) ONTO P
```



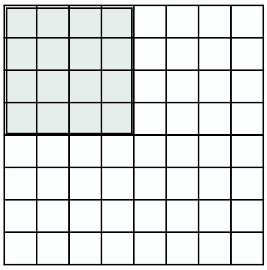
c)

```
real, dimension(8, 8) ::  A
!HPF$ PROCESSORS P(4)
!HPF$ DISTRIBUTE A(CYCLIC, *) ONTO P
```



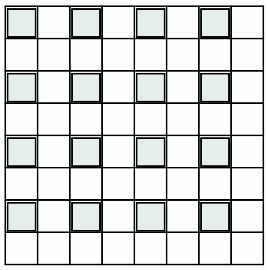
d)

```
real, dimension(8, 8) :: A
!HPF$ PROCESSORS P(2, 2)
!HPF$ DISTRIBUTE A(BLOCK, BLOCK)
                              ONTO P
```



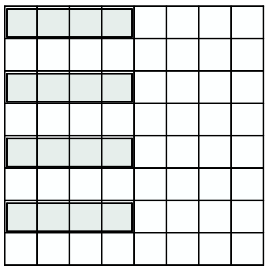
e)

```
real, dimension(8, 8) ::  A
!HPF$ PROCESSORS P(2, 2)
!HPF$ DISTRIBUTE A(CYCLIC, CYCLIC)
                              ONTO P
```



f)

```
real, dimension(8,8) ::  A
!HPF$ PROCESSORS P(2, 2)
!HPF$ DISTRIBUTE A(CYCLIC, BLOCK)
                              ONTO P
```

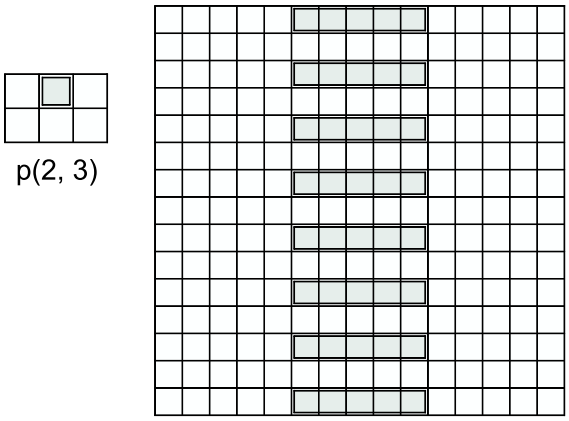


□

Przykład 6.7

Przydział elementów tablicy `A` do 6 procesorów. Zaznaczono część tablicy przypisaną do procesora `P(1, 2)`

```
real, dimension(15, 15)  :: A
!HPF$ PROCESSORS P(NUMBER_OF_PROCESSORS()/3, 3)
!HPF$ DISTRIBUTE  A(CYCLIC, BLOCK) ONTO P
```



□

W większości przypadków kompilator sam rozpoznaje instrukcje, które można wykonywać równolegle. Mogą jednak wystąpić takie sytuacje, jak w przykładzie poniżej

```
do i = 1, n
    B(index(i)) = A(i)
enddo
```

Kompilator nie będzie miał tutaj pewności, czy wartości współrzędnej `index(i)` dla różnych `i` są różne. Ta część kodu zostanie wykonana równolegle, jeśli pętle poprzedzimy dyrektywą `INDEPENDENT`, a więc nasz kod przyjmie następującą postać:

```
!HPF$ INDEPENDENT
do i = 1, n
    B(index(i)) = A(i)
enddo
```

Dyrektywy `INDEPENDENT` mogą być zagnieżdżone. Ukazuje to następujący przykład:

```
!HPF$ INDEPENDENT
do i = 1, n1                ! pętla po i - niezależna
!HPF$ INDEPENDENT
    do j = 1, n2                ! pętla po j - niezależna
        do k = 1, n3            ! wewnętrzna pętla - zależna
```

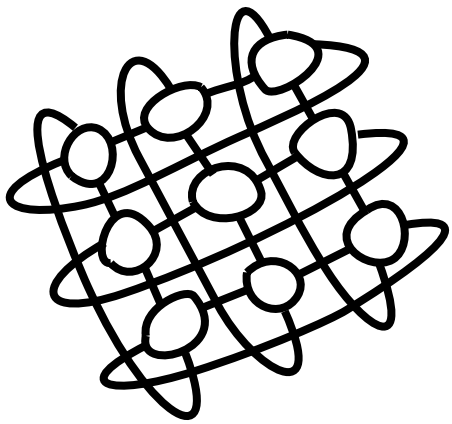
```
            A(i, j) = A(i, j) + B(i, j, k) * C(i, j)
        enddo
    enddo
enddo
```

Na zakończenie omawiania HPF-u zwróćmy uwagę na jego zalety i wady.

Podstawową zaletą jest prostota programowania, mimo że jest to narzędzie dla maszyn z pamięcią lokalną. Wadą natomiast jest to, iż jego zastosowanie ogranicza się do zrównoleglania drobnoziarnistego (ze względu na częstotliwość synchronizacji zadań – po każdej instrukcji). Tak więc przynosi on efekty tylko wtedy, gdy rozważamy bardzo duże zadania i dysponujemy maszynami z dużą liczbą procesorów. Narzędzie to nie nadaje się do implementacji algorytmów z równoległością gruboziarnistą, gdy można wyodrębnić duże fragmenty algorytmu wykonywane niezależnie. Za pomocą HPF-u nie można realizować równoległych zadań.

Część III

METODY OBLICZENIOWE



Rozdział 8

Algorytmy synchroniczne

Andrzej Karbowski, Krzysztof Malinowski, Ewa Niewiadomska-Szynkiewicz

8.3. Programowanie nieliniowe

Rozważamy następujące zadanie optymalizacji statycznej

$$\min_{x \in \mathbb{R}^n} f(x), \qquad f : \mathbb{R}^n \to \mathbb{R} \tag{8.8}$$

W metodach programowania matematycznego wielokrotnie obliczamy wartość funkcji celu oraz wyznaczamy kierunek poszukiwań. Wiele algorytmów korzysta z gradientu i hesjanu funkcji, można je więc w łatwy sposób zrównoleglić. Przedstawimy dwa przykładowe algorytmy.

8.3.1. Metody Newtona i zmiennej metryki

Omówimy teraz możliwość zrównoleglenia metod optymalizacji, korzystających z gradientu funkcji oraz hesjanu lub ich aproksymacji [4, 8].

Rozważmy zadanie (8.8) dla przypadku, gdy optymalizowana funkcja celu $f \in \mathbb{C}^2(\mathbb{R}^n)$. Korzystamy z warunku koniecznego optymalności, który staje się warunkiem wystarczającym, gdy rozważamy funkcje wypukłe

$$\nabla f(x) = 0$$

W celu rozwiązania zadania można zastosować podejście analogiczne do stosowanego w przypadku rozwiązywania układu równań nieliniowych. Należy jedynie zamiast $F(x)$ we wzorze (8.7) podstawić gradient $\nabla f(x)$ (czyli $F(x) \equiv \nabla f(x)$) oraz zamiast $\nabla F(x)$ hesjan $\nabla^2 f(x)$ (czyli $\nabla F(x) \equiv \equiv \nabla^2 f(x)$). Gradient oraz hesjan są wykorzystywane do wyznaczenia nowego kierunku poszukiwań minimum funkcji.

Dla uproszczenia notacji w dalszych rozważaniach będziemy stosować następujące oznaczenia gradientu i hesjanu: $\nabla f(x) = g(x)$ i $\nabla^2 f(x) = G(x)$.

Często nie mamy analitycznej postaci gradientu i hesjanu. Możemy wówczas zastosować ich aproksymację za pomocą różnic skończonych. W przypadku metod zmiennej metryki można zastosować aproksymację hesjanu lub jego odwrotności.

Rozważmy trzy przypadki: pierwszy – gdy rozwiązujemy zadanie, w którym możemy analitycznie wyznaczyć gradient funkcji; drugi – gdy gradient jest niedostępny i musi być aproksymowany oraz metody zmiennej metryki.

1. Gradient jest dostępny analitycznie

Załóżmy, że wykonujemy k-tą iterację algorytmu. Realizujemy następujące zadania:

– Obliczyć $g(x) = \frac{\partial f}{\partial x}$ w punktach $x^{(k)} + h_j e_j$, $j = 1, \ldots, \dim x$. Obliczyć poszczególne kolumny macierzy hesjanu $G^{(k)}$. Kolumny liczymy równolegle, korzystając odpowiednio ze wzoru

$$G_j^{(k)} = \frac{g(x^{(k)} + h_j e_j) - g(x^{(k)})}{h_j}$$

lub

$$G_j^{(k)} = \frac{g(x^{(k)}) - g(x^{(k)} - h_j e_j)}{h_j}$$

lub

$$G_j^{(k)} = \frac{g(x^{(k)} + h_j e_j) - g(x^{(k)} - h_j e_j)}{2h_j}$$

– Rozwiązać układ równań

$$G^{(k)}d^{(k)} = -g^{(k)}$$

Można zastosować tu równoległe metody rozwiązywania (patrz rozdz. 8.1).
– Wyznaczyć $x^{(k+1)} = x^{(k)} + d^{(k)}$

2. Gradient nie jest dostępny analitycznie

Załóżmy, że wykonujemy k-tą iterację algorytmu. Realizujemy następujące zadania:
– Wyznaczyć aproksymację gradientu $g(x) = \frac{\partial f}{\partial x}$, korzystając ze wzoru:

$$g_i(x^{(k)}) = \frac{f(x^{(k)} + h_i e_i) - f(x^{(k)})}{h_i}, \quad i = 1, \ldots, n$$

lub

$$g_i(x^{(k)}) = \frac{f(x^{(k)}) - f(x^{(k)} - h_i e_i)}{h_i}, \quad i = 1, \ldots, n$$

lub

$$g_i(x^{(k)}) = \frac{f(x^{(k)} + h_i e_i) - f(x^{(k)} - h_i e_i)}{2h_i}, \quad i = 1, \ldots, n$$

Obliczenia wartości funkcji $f(x^{(k)} \pm h_i e_i)$, $i = 1, \ldots, n$, mogą być realizowane równolegle.
– Obliczyć aproksymacje kolejnych elementów macierzy hesjanu

$$G_{ij}^{(k)} = \frac{f(x^{(k)} + h_i e_i + h_j e_j) - f(x^{(k)} + h_i e_i) - f(x^{(k)} + h_j e_j) + f(x^{(k)})}{h_i h_j},$$

$$i \neq j,\ i, j = 1, \ldots, n$$

oraz

$$G_{ii}^{(k)} = \frac{f(x^{(k)} + h_i e_i) - 2f(x^{(k)}) + f(x^{(k)} - h_i e_i)}{(h_i)^2},$$

Obliczenia wartości funkcji $f(x^{(k)} + h_i e_i + h_j e_j)$, $i, j = 1, \ldots, n$, również mogą być realizowane równolegle.
– Rozwiązać układ równań

$$G^{(k)}d^{(k)} = -g^{(k)}$$

Można tu, oczywiście, zastosować równoległe metody rozwiązywania przedstawione w rozdz. 8.1.
– Wyznaczyć $x^{(k+1)} = x^{(k)} + d^{(k)}$

3. Metody zmiennej metryki

Do najbardziej efektywnych metod optymalizacji należą metody zmiennej metryki, w których w poszczególnych iteracjach uaktualnia się odwrotność hesjanu (lub hesjan) poprzez pewną poprawkę ΔG^{-1} (lub ΔG) wyznaczaną na podstawie gradientu funkcji. Dla uproszczenia notacji, w dalszych rozważaniach będziemy stosować następujące oznaczenia odwrotności hesjanu i poprawki odwrotności hesjanu: $G^{-1} = V$ i $\Delta G^{-1} = \Delta V$. Najbardziej popularne schematy liczenia poprawek to:

– *Schemat Davidona–Fletchera–Powella (DFP)*

$$\Delta V_{DFP}^{(k+1)} = \frac{p^{(k)}(p^{(k)})^T}{(p^{(k)})^T r^{(k)}} - \frac{V^{(k)} r^{(k)} (r^{(k)})^T V^{(k)}}{(r^{(k)})^T V^{(k)} r^{(k)}} \tag{8.9}$$

– *Schemat Broydena–Fletchera–Goldfarba–Shanno (BFGS)*

$$\begin{aligned} \Delta V_{BFGS}^{(k+1)} = & \left(1 + \frac{(r^{(k)})^T V^{(k)} r^{(k)}}{(p^{(k)})^T r^{(k)}}\right) \frac{p^{(k)}(p^{(k)})^T}{(p^{(k)})^T r^{(k)}} - \\ & - \frac{p^{(k)}(r^{(k)})^T V^{(k)} + V^{(k)} r^{(k)} (p^{(k)})^T}{(p^{(k)})^T r^{(k)}} \end{aligned} \tag{8.10}$$

gdzie $p^{(k)} = x^{(k+1)} - x^{(k)}$ oraz $r^{(k)} = g^{(k+1)} - g^{(k)}$.

Na początku algorytmu zmiennej metryki przyjmuje się następujące przybliżenie odwrotności macierzy hesjanu $V^{(0)} = I$. Opiszemy teraz kolejną, załóżmy że k-tą, iterację algorytmu. Realizujemy następujące zadania:

– Wyznaczyć kierunek $d^{(k)}$, wykonując mnożenie

$$d^{(k)} = -V^{(k)} g^{(k)}$$

Można tu ewentualnie wykorzystać proste zrównoleglenia typu rozbicia pętli.

– Wyznaczyć $x^{(k+1)} = x^{(k)} + \alpha^{(k)} d^{(k)}$

– Korzystając z wybranego schematu (8.9) lub (8.10), wyznaczyć poprawkę $\Delta V^{(k+1)}$ oraz nową aproksymację odwrotności hesjanu

$$V^{(k+1)} = V^{(k)} + \Delta V^{(k+1)}$$

Analogicznie można zaproponować alternatywną metodę, polegającą na wyznaczaniu poprawki hesjanu [8, 76].

– *Davidona–Fletchera–Powella*

$$\begin{aligned} \Delta G_{DFP}^{(k+1)} = & \left(1 + \frac{(p^{(k)})^T G^{(k)} p^{(k)}}{(p^{(k)})^T r^{(k)}}\right) \frac{r^{(k)}(r^{(k)})^T}{(p^{(k)})^T r^{(k)}} - \\ & - \frac{r^{(k)}(p^{(k)})^T G^{(k)} + G^{(k)} p^{(k)} (r^{(k)})^T}{(p^{(k)})^T r^{(k)}} \end{aligned} \tag{8.11}$$

– *Broydena–Fletchera–Goldfarba–Shanno*

$$\Delta G_{BFGS}^{(k+1)} = \frac{r^{(k)}(r^{(k)})^T}{(r^{(k)})^T p^{(k)}} - \frac{G^{(k)} p^{(k)} (p^{(k)})^T G^{(k)}}{(p^{(k)})^T G^{(k)} p^{(k)}} \tag{8.12}$$

Korzystając z wybranego schematu (8.11) lub (8.12), wyznaczamy aproksymację hesjanu

$$G^{(k+1)} = G^{(k)} + \Delta G^{(k+1)} \tag{8.13}$$

oraz kierunek, rozwiązując równolegle układ równań

$$G^{(k+1)} d^{(k+1)} = -g^{(k+1)}$$

Zwróćmy uwagę, że metoda Newtona w wersji z aproksymacją hesjanu za pomocą różnic skończonych, zrównolegla się lepiej niż metody zmiennej metryki. Jest to związane z możliwością równoległego wyznaczania od razu poszczególnych elementów macierzy hesjanu, czyli wiekszą ziarnistością obliczeń.

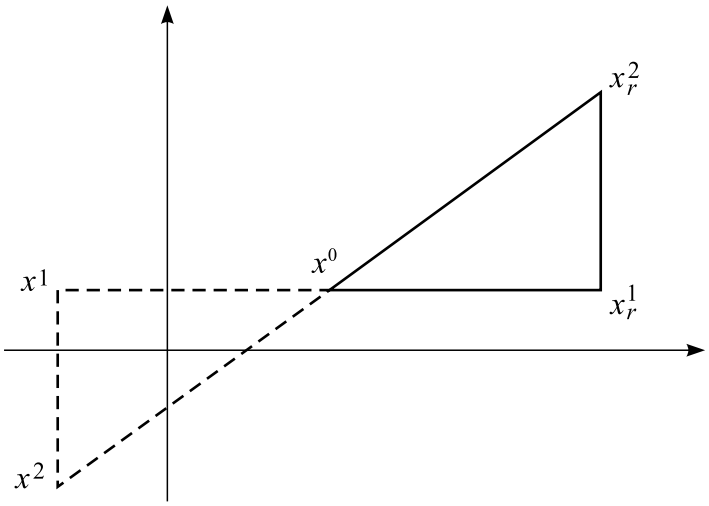
8.3.2. Metoda sympleksu nieliniowego

Niekiedy wyznaczenie wartości funkcji trwa bardzo długo lub funkcja celu ma punkty nieróżniczkowalności. W takich przypadkach opieranie się na estymatach gradientu i hesjanu jest niecelowe i warto zastosować algorytmy, które podczas poszukiwań ekstremum funkcji wykorzystują jedynie wartość funkcji celu. Do grupy tej należy metoda sympleksu nieliniowego Neldera–Meada [58]. Ideą algorytmu jest zdefiniowanie wielościanu (sympleksu) o $n+1$ wierzchołkach w przestrzeni $\mathbb{R}^n$ i jego odpowiednie przekształcanie, polegające na wykonywaniu operacji odbicia, ekspansji i kontrakcji względem środka sympleksu (centroidu). Omawiana metoda stawia bardzo małe wymagania odnośnie optymalizowanej funkcji. Wartości funkcji mogą być generowane przez *czarną skrzynkę*; wystarczy założenie, że rozwiązanie istnieje. Wadą metody jest powolna zbieżność, szczególnie odczuwalna w przypadku zadań wielowymiarowych. Przyspieszenie można uzyskać zrównoleglając obliczenia. W środowisku równoległym większą efektywność daje wariant metody sympleksu nieliniowego zaproponowany przez Torczon (poszukiwanie wielokierunkowe) [28]. Podstawowa różnica – w stosunku do podejścia Neldera–Meada – polega na tym, że w kolejnych krokach metody zmienia się tylko rozmiar przekształcanego sympleksu, kształt pozostaje ten sam. Poniżej przedstawiamy algorytm metody Torczon.

KROK 1. Utworzyć $(n+1)$-wymiarowy sympleks w przestrzeni $\mathbb{R}^n$. Obliczyć wartości funkcji celu w jego wierzchołkach.

KROK 2. Wyznaczyć wierzchołek najlepszy $x^0 = \arg \min_{l=1,\ldots,n+1} f(x^l)$, gdzie l oznacza numer wierzchołka.

KROK 3. Operacja *odbicia*: utworzyć nowy sympleks $\{x^0, x_r^1, \ldots, x_r^n\}$ odbijając wierzchołki x^l, $l = 1, \ldots, n$, względem wierzchołka najlepszego x^0 (rys. 8.1)



Rys. 8.1. Operacja odbicia

$$x_r^l = 2x^0 - x^l$$

Sprawdzić, czy operacja odbicia zakończyła się sukcesem, tzn. czy wśród wierzchołków nowego sympleksu znajduje się wierzchołek spełniający nierówność

$$\min_{l=1,\ldots,n} f(x_r^l) < f(x^0)$$

Jeżeli odbicie zakończyło się sukcesem, wykonywana jest operacja ekspansji powodująca zwiększenie sympleksu (KROK 4), w przeciwnym przypadku realizowana jest kontrakcja zmniejszająca sympleks (KROK 5).

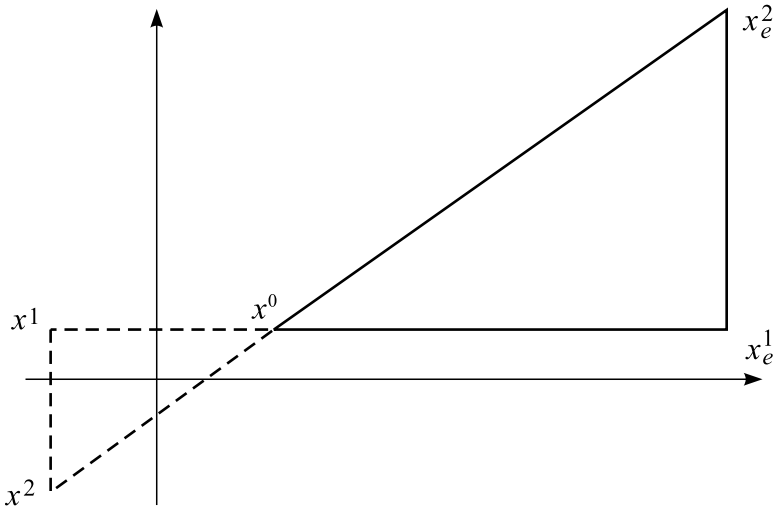
KROK 4. Operacja *ekspansji*: utworzyć nowy sympleks $\{x^0, x_e^1, \ldots, x_e^n\}$ powstały w wyniku powiększenia sympleksu otrzymanego w rezultacie odbicia (rys. 8.2)

$$x_e^l = \alpha x_r^l + (1 - \alpha)x^0 \qquad l = 1, \ldots, n$$

gdzie α oznacza współczynnik ekspansji. Operacja ekspansji kończy się sukcesem, gdy

$$\min_{l=1,\ldots,n} f(x_e^l) < \min_{l=1,\ldots,n} f(x_r^l)$$

Jeżeli ten warunek jest spełniony, to w miejsce starego sympleksu podstawiany jest nowy, czyli $\{x^0, x_e^1 \ldots, x_e^n\}$, i przechodzi się do KROKU 2 (tzn. już dla nowego sympleksu ponownie wyznaczany jest najlepszy wierzchołek, a następnie wykonywane jest odbicie pozostałych względem niego).



Rys. 8.2. Operacja ekspansji

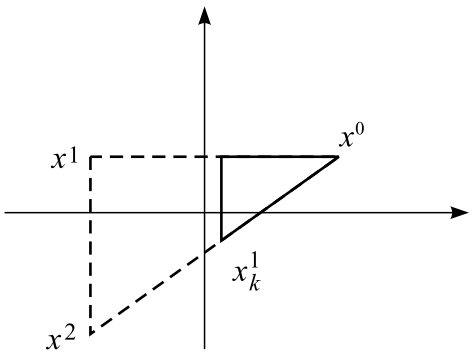
Jeżeli ekspansja dała wynik negatywny, przechodzi się do KROKU 2 z sympleksem $\{x^0, x_r^1 \ldots, x_r^n\}$.

KROK 5. Operacja *kontrakcji*: jeżeli odbicie (wykonane w KROKACH 2, 3) zakończyło się niepowodzeniem, to dokonujemy zmniejszenia sympleksu – kontrakcji. Nowy sympleks $\{x^0, x_k^1, \ldots, x_k^n\}$ powstaje w wyniku zastosowania na sympleksie $\{x^0, x^1, \ldots, x^n\}$ jednokładności o środku x_0 (tzn. w najlepszym wierzchołku) i skali – współczynniku kontrakcji $\beta \in (0, 1)$, (rys. 8.3), czyli

$$x_k^l = (1 - \beta)x^0 + \beta x^l \quad l = 1, \ldots, n$$

Następnie przechodzi się do KROKU 2, a więc ponownie wykonywane jest odbicie, tylko już nowego sympleksu $\{x^0, x_k^1 \ldots, x_k^n\}$, będącego wynikiem kontrakcji.

W metodzie Torczon w każdym kroku jednocześnie przekształca się n wierzchołków sympleksu, dlatego też naturalne jest jej zrównoleglenie. Ponieważ metoda została zaprojektowana z myślą o zrównolegleniu, możemy mówić o zrównolegleniu zadania. Ponadto, w związku z tym że wszystkie operacje wykonywane są względem tego samego wierzchołka, poszczególne



Rys. 8.3. Operacja kontrakcji

procesory mogą wyznaczać jednocześnie wartości funkcji w punktach powstałych w wyniku odbicia, ekspansji i kontrakcji wierzchołków, zanim będzie wiadomo, która z operacji zostanie zrealizowana. Wzrost przyspieszenia obliczeń jest najbardziej widoczny w przypadku, gdy $p > n$, czyli gdy dysponujemy większą liczbą procesorów niż wymiarowość przestrzeni. W takim przypadku niewykorzystane procesory mogą dokonywać dodatkowych obliczeń wartości funkcji celu, przewidując wyniki możliwych, przyszłych operacji.

8.4. Dekompozycja zadań optymalizacji

Przejdziemy teraz do omówienia metod dekompozycji złożonego zadania optymalizacji na kilka mniejszych zadań, które mogą być rozwiązywane przez różne procesory. Przedstawimy metodę bezpośrednią w wersji równouprawnionej i hierarchicznej oraz metodę cen w wersji hierarchicznej. Szczegółowy opis rozważanych metod Czytelnik znajdzie w pracach [6, 25, 26].

8.4.1. Algorytmy typu Jacobiego i Gaussa–Seidela

Na początku będziemy rozważać najprostszą metodę dekompozycji, polegającą na podzieleniu wektora zmiennych decyzyjnych x na p części, i optymalizacji wskaźnika jakości względem poszczególnych podwektorów.

Zakładamy, że rozwiązujemy zadanie

$$\min_{x \in X} f(x) \tag{8.14}$$

przy czym

$$X = X_1 \times X_2 \times \ldots \times X_p \tag{8.15}$$

czyli

$$x = (x_1, x_2, \ldots, x_p) \tag{8.16}$$

gdzie $x_i \in \mathbb{R}^{n_i}$, $i = 1, \ldots, p$. Założenie o braku ograniczeń łącznych między przestrzeniami X_i, $i = 1, \ldots, p$, może się wydać założeniem zbyt zawężającym klasę rozwiązywanych zadań. Weźmy jednak pod uwagę to, że i tak ograniczenia typu funkcyjnego są uwzględniane najczęściej poprzez funkcję kary, dla której iterowany jest współczynnik w kolejnych krokach. W każdym z tych kroków natomiast rozwiązywane jest zadanie bez ograniczeń funkcyjnych, czyli typu (8.14)–(8.15).

Zadanie powyższe będziemy rozwiązywać metodą kolejnych minimalizacji funkcji f względem poszczególnych składowych x_i wektora x. Wyróżnimy dwa typy algorytmów: Jacobiego oraz Gaussa–Seidela. Oznaczając kolejne iteracje przez k, możemy te algorytmy opisać w sposób następujący:

– algorytm Jacobiego:

$$x_i^{(k+1)} = \arg \min_{x_i \in X_i} f\left(x_1^{(k)}, \ldots, x_{i-1}^{(k)}, x_i, x_{i+1}^{(k)}, \ldots, x_p^{(k)}\right) \tag{8.17}$$

– algorytm Gaussa–Seidela:

$$x_i^{(k+1)} = \arg \min_{x_i \in X_i} f\left(x_1^{(k+1)}, \ldots, x_{i-1}^{(k+1)}, x_i, x_{i+1}^{(k)}, \ldots, x_p^{(k)}\right) \tag{8.18}$$

Algorytm Jacobiego polega na tym, że nowe wartości podwektora x_i, czyli $x_i^{(k+1)}$, dla każdego i, są uzyskane na podstawie tej samej informacji, a więc mogą być wyznaczone niezależnie od siebie. W algorytmie Gaussa–Seidela, do wyznaczenia nowej wartości x_i wykorzystywane są poprzednie wartości podwektorów $x_{i+1}, \ldots, x_p$, oraz już nowe wartości podwektorów $x_1, \ldots, x_{i-1}$. Algorytm Gaussa–Seidela jest więc algorytmem sekwencyjnym.

O zbieżności algorytmu Gaussa–Seidela mówi następujące twierdzenie [6]:

Twierdzenie 8.1.

Załóżmy, że funkcja $f : \mathbb{R}^n \to \mathbb{R}$ jest różniczkowalna w sposób ciągły oraz wypukła na X. Załóżmy ponadto, że, dla każdego i, f jest ściśle wypukłą funkcją x_i, gdy pozostałe składowe wektora x są stałe. Niech $x^{(k)}$ będzie ciągiem generowanym przez algorytm Gaussa–Seidela. Wówczas $x^{(k)}$ dąży w granicy do punktu minimalizującego f na X.

Inne twierdzenie wiąże zbieżność obydwu algorytmów, zarówno Jacobiego (8.17) jak i Gaussa–Seidela (8.18), z własnościami przybliżenia liniowego funkcji f, a mianowicie [6]:

Twierdzenie 8.2.

Niech $f : \mathbb{R}^n \to \mathbb{R}$ będzie funkcją różniczkowalną w sposób ciągły, γ zaś pewnym współczynnikiem dodatnim. Załóżmy, że odwzorowanie: $h : X \to \mathbb{R}^n$,

zdefiniowane jako

$$h(x) = x - \gamma \cdot \nabla f(x) \tag{8.19}$$

jest zwężające w blokowej normie maksimum

$$\|x\|_{\infty}^{w} = \|(x_1, \ldots, x_p)\|_{\infty}^{w} = \max_{i=1,\ldots,p} \frac{\|x_i\|_i}{w_i} \tag{8.20}$$

gdzie, dla każdego i, $\|.\|_i$ jest normą euklidesową w $\mathbb{R}^{n_i}$, a każdy współczynnik w_i jest liczbą dodatnią. Wówczas istnieje tylko jeden wektor $\hat{x}$, który minimalizuje funkcję f na X. Ponadto, ciągi $x^{(k)}$ generowane przez algorytmy (8.17) lub (8.18) zbiegają geometrycznie do $\hat{x}$.

Warunek zwężenia odwzorowania h (8.19) jest spełniony, na przykład, gdy hesjan funkcji f jest ograniczony, czyli gdy istnieje taka stała K, że:

$$\frac{\partial^2 f}{\partial x_i \partial x_j} \leqslant K \quad \forall x \in \mathbb{R}^n, \quad \forall i, j \tag{8.21}$$

oraz spełniony jest w nim warunek dominacji głównej diagonali dla pewnego wektora liczb dodatnich $[w_1, w_2, \ldots, w_n]$ (najczęściej przyjmuje się $w_i = 1 \ \ \forall i$)

$$w_i \cdot \frac{\partial^2 f}{\partial x_i^2} > \sum_{j \neq i} w_j \cdot \left| \frac{\partial^2 f}{\partial x_i \partial x_j} \right| \tag{8.22}$$

Wówczas, jeżeli przyjmiemy odpowiednio mały współczynnik γ, a dokładniej

$$0 < \gamma < \frac{1}{K} \tag{8.23}$$

to odwzorowanie h jest zwężające w normie maksimum.

Na przykład, dla funkcji kwadratowej

$$f(x) = \frac{1}{2} x' A x - b' x \tag{8.24}$$

warunek dominacji głównej diagonali (8.22) będzie miał postać

$$a_{ii} > \sum_{j \neq i} |a_{ij}| \quad \forall i \tag{8.25}$$

Wystarczy wówczas zastosować krok

$$\gamma < \frac{1}{a_{ii}} \quad \forall i \tag{8.26}$$

Uwaga 1.

Warunek zwężenia w blokowej normie maksimum oznacza, iż istnieje taki współczynnik $\alpha \in (0, 1)$, że

$$\|h(x) - h(y)\|_\infty^w \leqslant \alpha \|x - y\|_\infty^w \qquad \forall x, y \in X \tag{8.27}$$

W szczególności, dla punktu stałego odwzorowania h, oznaczmy go jako $\hat{x}$, otrzymamy (pomińmy dla uproszczenia wektor wag w) dla każdego zadania lokalnego $i = 1, \ldots, p$

$$\|h_i(x) - \hat{x}_i\|_i \leqslant \alpha \|x - \hat{x}\|_\infty \qquad \forall x, y \in X \tag{8.28}$$

gdzie $\|.\|_i$ jest normą euklidesową w $\mathbb{R}^{n_i}$.

Innymi słowy, po wykonaniu iteracji lokalnej, każdy podwektor lokalny będzie miał bliżej do swojego podwektora $\hat{x}$ rozwiązania (punktu stałego $\hat{x}$), niż najgorszy z podwektorów (tzn. najdalszy od swojego podwektora rozwiązania) w poprzedniej iteracji. □

Uwaga 2.

Funkcje, w których hesjan jest zdominowany diagonalnie stanowią podklasę w zbiorze wszystkich funkcji wypukłych. Wynika to z twierdzenia Gerszgorina[1] oraz równoważności między dodatnim znakiem wartości własnych i dodatnią określonością w klasie rzeczywistych macierzy symetrycznych [33].

A zatem, do wszystkich funkcji ściśle wypukłych można zastosować twierdzenie 8.1, czyli zdekomponowany algorytm optymalizacji typu Gaussa–Seidela, a tylko do tych funkcji wypukłych, w których zachodzi warunek dominacji głównej diagonali – twierdzenie 8.2 i algorytm Jacobiego.

O tym, że nie wszystkie funkcje wypukłe mają hesjan zdominowany diagonalnie, świadczy następujący przykład. Rozważmy formę kwadratową $f(x) = \frac{1}{2}x'Ax$ z macierzą

$$A = \begin{bmatrix} 3 & 2 & 2 \\ 2 & 3 & 2 \\ 2 & 2 & 3 \end{bmatrix} \tag{8.29}$$

Choć funkcja jest wypukła (hesjan, czyli macierz A, dodatnio określony), nie jest spełniony warunek dominacji głównej diagonali. □

Możemy również wziąć pod uwagę metody hybrydowe, które łączą cechy metod Jacobiego i Gaussa–Seidela. Na przykład, możemy podzielić p podwektorów wektora x na dwie grupy: $(x_1, \ldots, x_l)$ i $(x_{l+1}, \ldots, x_p)$ oraz wykonywać iteracje:

$$x_i^{(k+1)} = \arg \min_{x_i \in X_i} f\left(x_1^{(k)}, \ldots, x_{i-1}^{(k)}, x_i, x_{i+1}^{(k)}, \ldots, x_p^{(k)}\right) \tag{8.30}$$

[1] Wszystkie wartości własne macierzy A leżą w zbiorze będącym sumą teoriomnogościową kul $K\left(a_{ii}, \sum_{j \neq i} |a_{ij}|\right)$.

dla $i = 1, \ldots, l$, oraz

$$x_i^{(k+1)} = \arg \min_{x_i \in X_i} f\left(x_1^{(k+1)}, \ldots, x_l^{(k+1)}, x_{l+1}^{(k)}, \ldots \right. \\ \left. \ldots, x_{i-1}^{(k)}, x_i, x_{i+1}^{(k)}, \ldots, x_p^{(k)}\right) \tag{8.31}$$

dla $i = l+1, \ldots, p$. A zatem każda grupa podwektorów jest modyfikowana zgodnie z metodą Jacobiego, ale modyfikacje w drugiej grupie biorą już pod uwagę wyniki obliczeń w pierwszej grupie. Twierdzenie 8.2 zachowuje oczywiście ważność również i dla takich procesów obliczeniowych.

Zauważmy, że tego typu proces obliczeniowy, w którym wektor x może być podzielony w dowolny sposób, a podwektory x_i, $i = 1, \ldots, p$, są wyliczane zarówno według iteracji Jacobiego (niezależnie od siebie), jak i Gaussa–Seidela (wykorzystując najświeższe dane od innych zadań), jest już bardzo bliski obliczeniom asynchronicznym. W rozdziale 9.1 poświęconym tym obliczeniom wykażemy, że warunki zbieżności są identyczne jak w twierdzeniu 8.2.

8.4.2. Hierarchiczne metody optymalizacji

Omówimy teraz metody optymalizacji, w których wskaźnik jakości jest funkcją $p-1$ wskaźników cząstkowych, a wektor zmiennych decyzyjnych x możemy podzielić na dwa podwektory $x = [y, v]$, gdzie y składa się z podwektorów zadań cząstkowych, a v reprezentuje zmienne wspólne. Prezentowane techniki znajdują głównie zastosowanie w zadaniach, w których rozważa się systemy zbudowane z wielu podsystemów, tzw. systemy złożone. Podsystemy mogą być powiązane wzajemnie w różny sposób: bezpośrednio, gdy występują między nimi połączenia fizyczne, tzn. wyjścia jednych podsystemów są wejściami innych, lub pośrednio – poprzez wspólny cel działania systemu albo globalne ograniczenia fizyczne. W niektórych systemach fizycznych występują oba rodzaje powiązań, w innych jedno z nich. Przykładem systemu o powiązaniach bezpośrednich może być system wodno-gospodarczy, którego działanie polega na zabezpieczeniu dostaw wody na danym obszarze geograficznym kraju, czy też system produkcyjny, np. rafineria ropy naftowej, gdzie półprodukty są przesyłane między różnymi instalacjami. Problemami w których występują ograniczenia na zasoby, są np.: inwestowanie w reklamę danego produktu w różnych mediach przy ograniczonym kapitale, czy optymalizacja procesu produkcji przy ograniczonej pojemności magazynów.

Nasze zadanie optymalizacji dla systemu złożonego z $(p-1)$ podsystemów, sformułowane w przestrzeni

$$X = \left\{ x \in \mathbb{R}^n : \; g_i(x_i) \leqslant 0, \quad r_i(x_i) = 0, \quad i = 1, \ldots, p-1, \right. \\ \left. \sum_{i=1}^{p-1} z_i(x_i) \leqslant w, \quad \sum_{i=1}^{p-1} h_i(x_i) = s \right\} \tag{8.32}$$

będzie miało postać

$$\min_{x \in X} \left[f(x) = \Psi(f_1(x_1), f_2(x_2), \ldots, f_{p-1}(x_{p-1})) \right] \tag{8.33}$$

gdzie w oznacza wektor wspólnych zasobów, s – wektor dodatkowych stałych wejść (np. w systemach produkcyjnych jest to surowiec pobierany z zewnętrznego źródła, który uzupełnia zapotrzebowanie instalacji), a Ψ jest funkcją monotonicznie rosnącą względem każdego argumentu. Wszystkie występujące w opisie funkcje mogą być wektorami.

Równościowe i nierównościowe ograniczenia addytywne pokazują powiązania podsystemów. Zanim wyjaśnimy ich znaczenie, podzielmy zmienne podsystemów na dwie grupy $x_i = [y_i, v_i]$, gdzie y_i oznacza lokalne zmienne decyzyjne, a v_i wejścia interakcyjne (wiążące podsystemy). Dla większości systemów wzajemne oddziaływanie podsystemów związane z połączeniami fizycznymi opisuje równanie

$$v_i = \sum_{j=1}^{p-1} H_{ji} u_j + s_i \tag{8.34}$$

gdzie H_{ji} oznacza lokalny operator interakcji, v_i – wektor wejść interakcyjnych i-tego podsystemu, s_i – wektor dodatkowych wejść (danych), u_j – wektor wyjść j-tego podsystemu, czyli wynik odwzorowania reprezentującego funkcję podsystemu $u_j = r_j(y_j, v_j)$.

Ograniczenia wynikające ze wspólnych zasobów formułujemy w postaci $\sum_{i=1}^{p-1} z(y_i, v_i) \leqslant w$. Ograniczenia $g_i(y_i, v_i) \leqslant 0$ są dodatkowymi ograniczeniami na zmienne wejściowe każdego podsystemu.

Jeżeli przyjmiemy, że wskaźnik jakości f w zadaniu optymalizacji (8.33) jest funkcją $p-1$ wskaźników cząstkowych, z których każdy jest zależny od zmiennych y_i i v_i, oraz że ograniczenia występujące w sformułowaniu zadania mogą być częściowo zdekomponowane, to do rozwiązania zadania możemy zastosować hierarchiczne metody optymalizacji.

8.4.3. Optymalizacja hierarchiczna metodą bezpośrednią

Rozważamy następujące zadanie optymalizacji

$$\min_{x\in X} f(x) \tag{8.35}$$

Wskaźnik jakości jest tu monotoniczną funkcją wskaźników cząstkowych, z których każdy jest zależny od dwóch podwektorów $[y_i, v]$

$$x = [y,\ v] = [y_1, y_2, \ldots, y_{p-1}, v] \tag{8.36}$$

gdzie podwektor v powtarza się we wszystkich wskaźnikach cząstkowych, to znaczy

$$\min_{(y,v)\in YV} \Big[f(y,v) = \Psi\left(f_1(y_1,v), f_2(y_2,v), \ldots \right. \\ \left. \ldots, f_{p-1}(y_{p-1},v)\right) + f_0(v) \Big] \tag{8.37}$$

Załóżmy dodatkowo, iż $y_i \in \mathbb{R}^{n_i},\ i = 1,\ldots,p-1,\ v \in \mathbb{R}^{n_v}$ oraz że dla każdego i, a także dla wszystkich ustalonych skalarów

$$z_1 = \bar{z}_1, \quad z_2 = \bar{z}_2, \ldots, \quad z_{i-1} = \bar{z}_{i-1}, \quad z_{i+1} = \bar{z}_{i+1}, \ldots, \quad z_{p-1} = \bar{z}_{p-1}$$

funkcja jednej zmiennej $z_i \in \mathbb{R}$

$$\Psi\left(\bar{z}_1, \bar{z}_2, \ldots, \bar{z}_{i-1}, z_i, \bar{z}_{i+1}, \bar{z}_{p-1}\right)$$

jest monotonicznie rosnąca.

Jednocześnie zakłada się, że ograniczenia są częściowo zdekomponowane, to znaczy zbiór dopuszczalny X ma postać

$$X = YV = \Big\{ (y_1, y_2, \ldots, y_{p-1}, v) : v \in V, \quad (y_i, v) \in YV_i, \\ \forall i = 1,\ldots,p-1 \Big\} \tag{8.38}$$

Zadanie takie można rozwiązać, rozwiązując $p-1$ zadań lokalnych

$$\min_{y_i \in Y_i(v)} f_i(y_i, v), \quad i = 1,\ldots,p-1 \tag{8.39}$$

gdzie zbiór $Y_i(v)$ jest wycinkiem zbioru YV_i dla ustalonego v, to znaczy

$$Y_i(v) = \{y_i : (y_i, v) \in YV_i\} \tag{8.40}$$

a następnie rozwiązując zadanie koordynacji

$$\min_{v\in V\cap V_0} \Big[f(\hat{y}(v), v) = \Psi\Big(f_1(\hat{y}_1(v), v),\ f_2(\hat{y}_2(v), v), \ldots \\ \ldots,\ f_{p-1}(\hat{y}_{p-1}(v), v)\Big) + f_0(v) \Big] \tag{8.41}$$

przy czym zbiór V_0 jest zbiorem zmiennych koordynacyjnych v, dla których wszystkie wycinki lokalne $Y_i(v)$ są niepuste, to znaczy:

$$V_0 = \{v : \ Y_i(v) \neq \emptyset \quad \forall i = 1, \ldots, p-1\} \tag{8.42}$$

a $\hat{y}_i(v)$, $i = 1, \ldots, p-1$, są rozwiązaniami zadań lokalnych.

Jeśli chodzi o warunki stosowalności metody bezpośredniej, to poza oczywistymi warunkami zapewniającymi, zgodnie z twierdzeniem Weierstrassa, istnienie rozwiązania zadania (8.37), a mianowicie by funkcja $f(.,.)$ była ciągła, a obszar $X = YV$ zwarty, często wymaga się również by [25]:

- funkcja $f(.,.)$ oraz zbiór YV były wypukłe – zapewnia to wypukłość optymalizowanej funkcji $f(\hat{y}(v), v)$ w zadaniu koordynacji (8.41);
- zbiór YV miał postać iloczynu kartezjańskiego: $YV = Y \times V$ (czyli, aby ograniczenia na y i v były niezależne) – zapewnia to ciągłość funkcji $f(\hat{y}(v), v)$ w zadaniu (8.41);
- zbiór YV miał postać iloczynu kartezjańskiego: $YV = Y \times V$, funkcja $f(.,.)$ była ściśle wypukła oraz w całej dziedzinie istniała ciągła pochodna $\frac{\partial f}{\partial v}$ – zapewnia to różniczkowalność funkcji $f(\hat{y}(.),.)$, przy czym

$$\nabla f_v \left(\hat{y}(v), v\right) = \left.\frac{\partial f}{\partial v}(y, v)\right|_{y=\hat{y}(v)} \tag{8.43}$$

Gdy spełnione są powyższe warunki, do rozwiązania wszystkich p zadań optymalizacji (zarówno $p-1$ zadań lokalnych, jak i zadania koordynacji) mogą być użyte efektywne gradientowe metody optymalizacji (np. największego spadku, gradientów sprzężonych lub zmiennej metryki).

Zauważmy, że oznaczając przez k kolejne iteracje, możemy przedstawić hierarchiczny algorytm metody bezpośredniej (8.39)–(8.41) w następujący sposób:

- zadania lokalne

$$y_i^{(k+1)} = \arg \min_{y_i \in Y_i(v^{(k)})} f_i(y_i, v^{(k)}), \quad i = 1, \ldots, p-1 \tag{8.44}$$

- zadanie koordynacji

$$\begin{aligned} v^{(k+1)} &= \arg \min_{v \in V \cap V_0} \Big[f(y^{(k+1)}, v) = \\ &= \Psi \left(f_1(y_1^{(k+1)}, v), f_2(y_2^{(k+1)}, v), \ldots, f_{p-1}(y_{p-1}^{(k+1)}, v) \right) + f_0(v) \Big] \end{aligned} \tag{8.45}$$

Jest to nic innego jak algorytm typu (8.18).

Klasyczną metodę bezpośrednią optymalizacji hierarchicznej można więc potraktować jako algorytm hybrydowy Gaussa–Seidela/Jacobiego. Między grupą zadań lokalnych oraz zadaniem koordynacji realizowany jest algorytm typu Gaussa–Seidela. Same zadania dolnego poziomu (8.44) rozwiązywane

są, dzięki monotoniczności wskaźnika jakości względem wskaźników lokalnych, poprzez algorytm typu Jacobiego, czyli niezależnie od siebie. Zadanie koordynacji (8.45), tak jak w metodzie Gaussa–Seidela, jest rozwiązywane po otrzymaniu od zadań lokalnych optymalnych wartości podwektorów $\hat{y}_i(v^{(k)}) = y_i^{(k+1)}$, $i = 1, \ldots, p-1$, czyli po wykorzystaniu przez nie poprzedniej wartości wektora zmiennych koordynujących $v^{(k)}$. Zadania lokalne są następnie rozwiązywane dla nowego wektora zmiennych koordynujących $v^{(k+1)}$, itd.

Specyfika metody bezpośredniej polega jedynie na dekompozycji zadania minimalizacji funkcji Ψ, właśnie dzięki jej monotoniczności względem poszczególnych argumentów, na wiele minimalizacji, argumentów tej funkcji, czyli funkcji $f_i(y_i, v)$. Zamiast jednego zadania o wymiarze $\sum_{i=1}^{p-1} n_i$ rozwiązuje się $p-1$ zadań, z których każde ma wymiar n_i. Może to być opłacalne w tych metodach optymalizacji, w których nakłady obliczeniowe rosną silniej niż liniowo z wymiarem zadania. Jednocześnie, zadania te mogą być rozwiązywane niezależnie, czyli, na przykład, przez różne procesory maszyny równoległej. Daje to dodatkową korzyść z zastosowania metody bezpośredniej.

8.4.4. Optymalizacja hierarchiczna metodą cen

W metodzie koordynacji bezpośredniej w sformułowaniu zadania nadrzędnego występują ograniczenia na zmienne wiążące. Ta niedogodność oraz ograniczenie możliwości stosowania metod gradientowych na poziomie koordynatora utrudniają rozwiązanie zadania optymalizacji. W metodzie cen powyższe problemy nie występują, gdyż nie rozpatruje się zmiennych v jako zmiennych decyzyjnych zadania nadrzędnego. Funkcją celu jest w tym przypadku funkcja Lagrange'a, zmiennymi decyzyjnymi zadania koordynatora są mnożniki Lagrange'a, mające znaczenie cen.

Podobnie jak w metodzie bezpośredniej przyjmujemy, że wskaźnik jakości w zadaniu (8.33) jest funkcją $p-1$ wskaźników cząstkowych, z których każdy jest zależny od zmiennych y i v. Pierwszą różnicą w stosunku do metody bezpośredniej jest wymaganie, by funkcja celu była addytywna, tj. $f(x) = \sum_{i=1}^{p-1} f_i(x_i)$. Każdy wskaźnik cząstkowy oraz odpowiadające mu ograniczenia są zależne od dwóch podwektorów y_i, v_i, gdzie $y_i \in \mathbb{R}^{n_{y_i}}$, a $v_i \in \mathbb{R}^{n_{v_i}}$, $i = 1, \ldots, p-1$. Możemy więc zapisać $x = [y, v] = [y_1, y_2, \ldots, y_{p-1}, v_1, v_2, \ldots, v_{p-1}]$, $f(y, v) = \sum_{i=1}^{p-1} f_i(y_i, v_i)$. Zbiory rozwiązań dopuszczalnych zadań cząstkowych są następujące $YV_i =$

$= \{(y_i, v_i) : \quad g_i(y_i, v_i) \leqslant 0,\ r_i(y_i, v_i) = 0,\ y_i \in \mathbb{R}^{n_{y_i}},\ v_i \in \mathbb{R}^{n_{v_i}}\}$. Tak więc, w przeciwieństwie do metody bezpośredniej, dla każdego indeksu i, zarówno w lokalnych funkcjach celu f_i jak i ograniczeniach g_i, występują różne wektory v_i. W tym ujęciu eliminujemy zmienne wyjściowe podsystemów u_i, $i = 1, \ldots, p-1$. Nie tracimy przez to ogólności rozważań, gdyż we wzorze (8.34) zawsze możemy rozszerzyć wektor v_i o te zmienne u_j, $j = 1, \ldots, p-1$, dla których $H_{ji} \neq 0$. Wówczas funkcje podsystemów $u_j = r_j(y_j, v_j)$ staną się dodatkowymi ograniczeniami lokalnymi typu $r_i(x_i) = 0$.

Uwzględniając równania wzajemnej interakcji podsystemów (8.34), otrzymujemy ograniczenia globalne na zmienne wiążące v_i. Przyjmijmy, że muszą one spełniać warunki

$$\sum_{i=1}^{p-1} A_{ji} v_i = s_j, \qquad j = 1, \ldots, m \tag{8.46}$$

gdzie m jest liczbą powiązań między podsystemami. Macierz A_{ji} utworzona została na podstawie macierzy H_{ji} z uwzględnieniem ewentualnego rozszerzenia wektora zmiennych wiążących v_i.

Nie możemy również pominąć ograniczeń globalnych na zasoby. Tak więc zbiór dopuszczalny rozwiązań ma postać

$$X = YV = \left\{ (y_i, v_i) : (y_i, v_i) \in YV_i, \quad \forall i = 1, \ldots, p-1, \quad \sum_{i=1}^{p-1} z_i(y_i, v_i) \leqslant \ w, \right.$$
$$\left. \sum_{i=1}^{p-1} A_{ji} v_i = s_j, \quad \forall j = 1, \ldots, m \right\}$$

gdzie $YV_i = \{(y_i, v_i) :\ g_i(y_i, v_i) \leqslant 0,\ r_i(y_i, v_i) = 0\}$.

Wykorzystamy teraz założenie o addytywnej postaci funkcji celu oraz rozdzielności zmiennych (y_i, v_i) i obszarów YV_i. Przyporządkowując równaniom (8.46) mnożniki Lagrange'a $\mu_1, \mu_2, \ldots, \mu_m$ oraz globalnym ograniczeniom nierównościowym wektor mnożników $\lambda \geqslant 0$, formułujemy funkcję Lagrange'a koordynatora

$$\begin{aligned} L(y, v, \lambda, \mu) &= \sum_{i=1}^{p-1} f_i(y_i, v_i) + \lambda^T \left(\sum_{i=1}^{p-1} z_i(y_i, v_i) - w \right) + \\ &\quad + \sum_{j=1}^{m} \mu_j^T \left(\sum_{i=1}^{p-1} A_{ji} v_i - s_j \right) = \\ &= \sum_{i=1}^{p-1} \left(f_i(y_i, v_i) + \lambda^T z_i(y_i, v_i) + \sum_{j=1}^{m} \mu_j^T A_{ji} v_i \right) - \lambda^T w - \\ &\quad - \sum_{j=1}^{m} \mu_j^T s_j \end{aligned} \tag{8.47}$$

Załóżmy, że funkcja (8.47) ma punkt siodłowy w rozważanym obszarze YV i można go wyznaczyć dokonując najpierw minimalizacji funkcji (8.47) względem zmiennych $(y_i, v_i) \in YV_i$, $i = 1, \ldots, p-1$, a następnie maksymalizacji względem μ i $\lambda \geqslant 0$. Możemy więc sformułować następujące zadanie

$$\max_{\lambda \geqslant 0,\, \mu} \left\{ \min_{(y_i, v_i) \in YV_i} \left[L(y_1, \ldots, y_{p-1}, v_1, \ldots, v_{p-1}, \lambda, \mu) = \right. \right. \tag{8.48}$$

$$\left. \left. = \sum_{i=1}^{p-1} \left(f_i(y_i, v_i) + \lambda^T z_i(y_i, v_i) + \sum_{j=1}^{m} \mu_j^T A_{ji} v_i \right) - \lambda^T w - \sum_{j=1}^{m} \mu_j^T s_j \right] \right\}$$

Rozwiązanie zadania (8.48) sprowadza się teraz do znalezienia punktu siodłowego $(\hat{y}, \hat{v}, \hat{\lambda}, \hat{\mu})$ spełniającego warunek

$$L(\hat{y}, \hat{v}, \lambda, \mu) \leqslant L(\hat{y}, \hat{v}, \hat{\lambda}, \hat{\mu}) \leqslant L(y, v, \hat{\lambda}, \hat{\mu}) \tag{8.49}$$

przy założeniu, że (y_i, v_i), $i = 1, \ldots, p-1$, są dopuszczalne.

Zadanie optymalizacji (8.48) można rozwiązać, rozwiązując $p-1$ zadań lokalnych

$$\min_{(y_i, v_i) \in YV_i} \left[L_i(y_i, v_i, \lambda, \mu) = f_i(y_i, v_i) + \lambda^T z_i(y_i, v_i) + \sum_{j=1}^{m} \mu_j^T A_{ji} v_i \right] \tag{8.50}$$

oraz zadanie koordynacji

$$\max_{\lambda \geqslant 0,\, \mu} \left[\varphi(\lambda, \mu) = \sum_{i=1}^{p-1} L_i(\hat{y}_i, \hat{v}_i, \lambda, \mu) - \lambda^T w - \sum_{j=1}^{m} \mu_j^T s_j \right] \tag{8.51}$$

które polega na doborze mnożników Lagrange'a μ i λ poprzez maksymalizację względem μ i λ rezultatów zadań lokalnych powiększonych o liniowe funkcje mnożników. Rozwiązaniem zadania koordynatora jest maksimum globalne, gdyż funkcja celu jest wklęsła. Możemy to w łatwy sposób wykazać.

Dla uproszczenia rozważań przyjmijmy $\eta = [\lambda, \mu]$ i $w = 0$ oraz $s = 0$. Funkcja celu koordynatora ma wówczas postać

$$\varphi(\eta) = \sum_{i=1}^{p-1} \min_{(y_i, v_i) \in YV_i} L_i(y_i, v_i, \eta) \tag{8.52}$$

Funkcja $\varphi(\eta)$ jest wklęsła wtedy i tylko wtedy, gdy zachodzi

$$\forall \eta_1, \eta_2 \quad \forall \rho \in [0, 1] \;\; \varphi\Big(\rho\eta_1 + (1-\rho)\eta_2 \Big) \geqslant \rho\varphi(\eta_1) + (1-\rho)\varphi(\eta_2) \tag{8.53}$$

Ponieważ funkcje celu zadań lokalnych L_i są liniowymi funkcjami η, więc możemy zapisać

$$\begin{aligned}
\varphi\left(\rho\eta_1+(1-\rho)\eta_2\right) &= \sum_{i=1}^{p-1} \min_{(y_i,v_i)\in YV_i} L_i(y_i,v_i,\rho\eta_1+(1-\rho)\eta_2) = \\
&= \sum_{i=1}^{p-1} \min_{(y_i,v_i)\in YV_i} \Big[\rho L_i(y_i,v_i,\eta_1)+(1-\rho)L_i(y_i,v_i,\eta_2)\Big] \geqslant \\
&\geqslant \rho \sum_{i=1}^{p-1} \min_{(y_i,v_i)\in YV_i} L_i(y_i,v_i,\eta_1) + \\
&\quad + (1-\rho)\sum_{i=1}^{p-1} \min_{(y_i,v_i)\in YV_i} L_i(y_i,v_i,\eta_2) = \\
&= \rho\varphi(\eta_1)+(1-\rho)\varphi(\eta_2)
\end{aligned}$$

co kończy dowód.

Z powyższych przekształceń wynika również, że każda funkcja lokalna $\min\limits_{(y_i,v_i)\in YV_i} L_i(y_i,v_i,\eta_i)$ jest wklęsła.

Na koniec kilka słów o tym, jak rozwiązywać zadania praktyczne. Prace rozpoczynamy od sformułowania zadania optymalizacji tak, by mogło być ono rozwiązane metodą cen. Przyjmijmy, że z wektora występujących w zadaniu zmiennych x wyodrębniliśmy już wszystkie zmienne v wiążące podsystemy. Na pozostałe zmienne składają się zmienne lokalne podsystemów y_i, $i=1,\ldots,p-1$. Wektor x podzieliliśmy więc na następujące podwektory $x=[v,y_1,\ldots,y_{p-1}]$. Sformułowanie zadania dla metody cen wymaga, by funkcje każdego podsystemu zależały od różnych wektorów v_i. Można to w łatwy sposób otrzymać zastępując wspólne zmienne wiążące v przez $p-1$ wektorów równych sobie zmiennych $v_1,v_2,\ldots,v_p$ (gdzie $v_1=v_2=\ldots=v_p$ jest to powielenie zmiennych). Sformułujmy zdanie (8.48) przy założeniu takiej postaci zmiennych wiążących.

$$\begin{aligned}
\max_{\lambda\geqslant 0,\,\mu} \Bigg\{ & \min_{(y_1,v_1)\in YV_1} \Big[f_1(y_1,v_1)+\lambda^T z_1(y_1,v_1)+\mu_1^T v_1\Big] + \\
& + \min_{(y_2,v_2)\in YV_2} \Big[f_2(y_2,v_2)+\lambda^T z_2(y_2,v_2)-\mu_1^T v_2+\mu_2^T v_2\Big] + \ldots \\
& \ldots + \min_{(y_{p-1},v_{p-1})\in YV_{p-1}} \Big[f_{p-1}(y_{p-1},v_{p-1})+\lambda^T z_{p-1}(y_{p-1},v_{p-1})-\mu_{p-1}^T v_{p-1}\Big] - \\
& - \lambda^T w - \sum_{j=1}^{m} \mu_j^T s_j \Bigg\}
\end{aligned}$$

Przedstawiliśmy sformułowanie zadania koordynatora i zadań lokalnych dla systemu, w którym podsystemy są wzajemnie powiązane poprzez połączenia fizyczne oraz dodatkowo występują globalne ograniczenia fizyczne. Rozpatrzymy teraz dwa szczególne przypadki systemów: systemy powiązane i systemy o wspólnych zasobach.

PRZYKŁAD 8.1. Systemy powiązane.

Podsystemy oddziałują na siebie poprzez fizyczne połączenia i nie ma innych ograniczeń globalnych. Zadanie koordynatora i zadania lokalne przyjmują następującą postać:

– zadanie koordynatora

$$\max_{\mu}\left[\varphi(\mu)=\sum_{i=1}^{p-1}L_i(\hat{y}_i,\hat{v}_i,\mu)-\sum_{j=1}^{m}\mu_j^T s_j\right] \tag{8.54}$$

– i-te zadanie lokalne

$$\min_{(y_i,v_i)\in YV_i}\left[L_i(y_i,v_i,\mu)=f_i(y_i,v_i)+\sum_{j=1}^{m}\mu_j^T A_{ji}v_i\right] \tag{8.55}$$

$$YV_i=\left\{(y_i,v_i):\quad g_i(y_i,v_i)\leqslant 0,\ \ r_i(y_i,v_i)=0,\ \ y_i\in\mathbb{R}^{n_{y_i}},\ v_i\in\mathbb{R}^{n_{v_i}}\right\}$$

W zadaniu pierwotnym mogą wystąpić dodatkowo ograniczenia na zmienne wiążące ($d_i(v_i)\leqslant 0$). W sformułowaniu dla metody cen ograniczenia te będą uwzględniane w zadaniach lokalnych, zbiór rozwiązań dopuszczalnych i-tego zadania będzie wówczas następujący: $YV_i = \{(y_i,v_i):\quad r_i(y_i,v_i)=0,\ g_i(y_i,v_i)\leqslant 0,\ d_i(v_i)\leqslant 0\}$. □

PRZYKŁAD 8.2. Systemy o wspólnych zasobach.

Podsystemy nie są wzajemnie połączone, wielkością wiążącą są wspólne zasoby w. Zadanie koordynatora i zadania lokalne przyjmują następującą postać

– zadanie koordynatora

$$\max_{\lambda\geqslant 0}\left[\varphi(\lambda)=\sum_{i=1}^{p-1}L_i(\hat{y}_i,\hat{v}_i,\lambda)-\lambda^T w\right] \tag{8.56}$$

– i-te zadanie lokalne

$$\min_{(y_i,v_i)\in YV_i}\left[L_i(y_i,v_i,\lambda)=f_i(y_i,v_i)+\lambda^T z_i(y_i,v_i)\right] \tag{8.57}$$

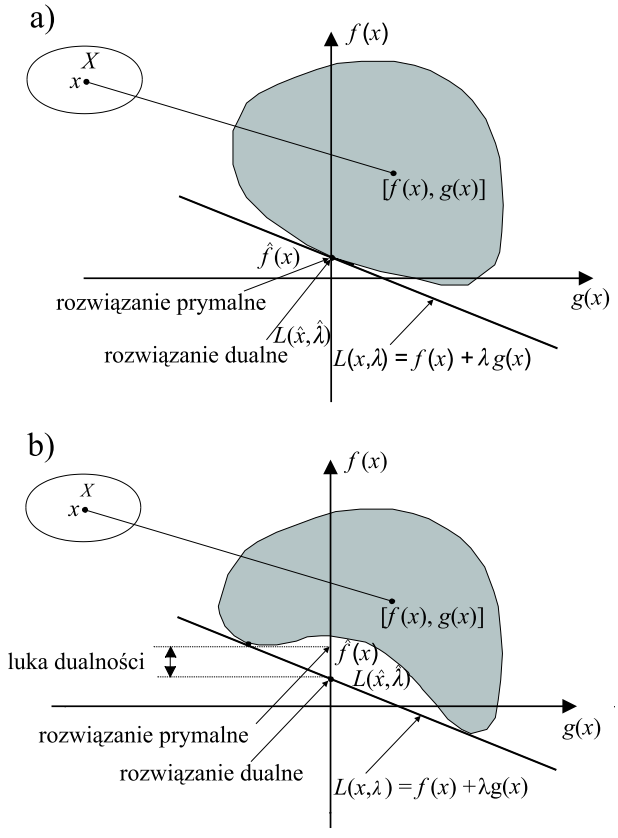
$$YV_i=\left\{(y_i,v_i):\quad g_i(y_i,v_i)\leqslant 0,\ \ r_i(y_i,v_i)=0,\ \ y_i\in\mathbb{R}^{n_{y_i}},\ v_i\in\mathbb{R}^{n_{v_i}}\right\}$$

□

Przejdźmy teraz do omówienia warunków stosowalności metody cen. Poza warunkami, by funkcja celu $f(.)$ oraz funkcje ograniczeń $g(.)$ i $z(.)$ były ciągłe, $f(.)$ addytywna oraz dla każdego λ i μ zadania lokalne (8.50) miały jednoznaczne rozwiązania $(\hat{x}_i(\lambda,\mu) = \hat{y}_i(\lambda,\mu), \hat{v}_i(\lambda,\mu))$, wymaga się istnienia punktu siodłowego funkcji (8.47). Jeżeli punkt siodłowy istnieje, to jest on rozwiązaniem zadania optymalizacji. Twierdzenie odwrotne nie jest niestety prawdziwe, może istnieć rozwiązanie zadania, a nie istnieć punkt siodłowy, czyli

$$\max_{\lambda,\mu} \min_{x\in\mathbb{R}^n} L(x,\lambda,\mu) < \min_{x\in X} f(x) \tag{8.58}$$

Mówimy wówczas o wystąpieniu luki dualności (rys. 8.4). Pokażemy to na dwóch prostych przykładach: P1 – luka dualności nie występuje, P2 – luka dualności występuje. W celu uproszczenia rozważań nie będziemy uwzględniać addytywnych ograniczeń równościowych i nierównościowych.



Rys. 8.4. Ilustracja luki dualności: a) luka dualności nie występuje, b) luka dualności występuje

P1. Punkt siodłowy istnieje

$$\min_x f(x) = 0{,}5(x_1^2 + x_2^2)$$
$$g(x) = x_1 - 1 \leqslant 0, \quad x \in \mathbb{R}^2$$

Rozwiązaniem zadania jest punkt $\hat{x} = (0, 0)$, $f(\hat{x}) = 0$.

Sformułujmy funkcję Lagrange'a $L(x, \lambda) = 0{,}5(x_1^2 + x_2^2) + \lambda(x_1 - 1)$, $\lambda \geqslant 0$. Korzystając z warunków koniecznych optymalności, wyznaczamy funkcję dualną $L_D(\lambda) = \min_{x \in \mathbb{R}^2} L(x, \lambda) = -0{,}5\lambda^2 - \lambda$. Rozwiązanie zadania dualnego $\max_{\lambda \geqslant 0} L_D(\lambda)$ otrzymujemy dla $\hat{\lambda} = 0$ i $\hat{x} = (0, 0)$.

P2. Luka dualności

$$\min_{x \in X} f(x) = -x^2$$
$$g(x) = 2x - 1 \leqslant 0, \quad x \in [\tfrac{1}{10},\ \tfrac{9}{10}]$$

Rozwiązaniem zadania jest punkt $\hat{x} = \frac{1}{2}$, $f(\hat{x}) = -\frac{1}{4}$.

Sformułujmy funkcję Lagrange'a $L(x, \lambda) = -x^2 + \lambda(2x - 1)$, $\lambda \geqslant 0$. Funkcja Lagrange'a jest wklęsła, tak więc będzie zawsze przyjmować najmniejsze wartości na ograniczeniach kostkowych, czyli odpowiednio w punktach $x = \frac{1}{10}$ lub $x = \frac{9}{10}$, w zależności od wartości λ. Funkcje Lagrange'a $L(x, \lambda)$ dla $x = \frac{1}{10}$ i $x = \frac{9}{10}$ są następujące: $L(\frac{1}{10}, \lambda) = -\frac{1}{100} - \frac{4}{5}\lambda$, a $L(\frac{9}{10}, \lambda) = -\frac{81}{100} + \frac{4}{5}\lambda$. W tym przypadku metoda dualna nie daje poprawnego rozwiązania, ponieważ wartość funkcji Lagrange'a dla $\hat{\lambda}$ w jednym z tych punktów będzie zawsze mniejsza od $f(\frac{1}{2})$. Punkt siodłowy nie istnieje. Występuje luka dualności.

Powróćmy do głównego toku rozważań, a konkretnie do możliwości równoległej realizacji metody cen. Jeżeli założymy, że dysponujemy p procesorami, to naturalne jest, by zadania lokalne i nadrzędne były rozwiązywane przez różne procesory. Oznaczając przez k kolejny numer iteracji, możemy algorytm metody cen przedstawić w następujący sposób:

– zadania lokalne

$$(y_i^{(k+1)},\, v_i^{(k+1)}) = \arg \min_{(y_i,\, v_i) \in Y V_i^{(k)}} L_i(y_i,\, v_i, \lambda^{(k)},\, \mu^{(k)}) \quad i = 1, \ldots, p-1 \tag{8.59}$$

– zadanie koordynacji

$$(\lambda^{(k+1)},\, \mu^{(k+1)}) = \arg \max_{\lambda \geqslant 0,\, \mu} \Bigg[\varphi(\lambda,\, \mu) = \sum_{i=1}^{p-1} L_i \left(y_i^{(k+1)},\, v_i^{(k+1)},\, \lambda,\, \mu \right) - \lambda^T w - \sum_{j=1}^{m} \mu_j^T s_j \Bigg] \tag{8.60}$$

Metodę cen, podobnie jak metodę bezpośrednią można potraktować jako algorytm hybrydowy Gaussa–Seidela/Jacobiego. Między grupą zadań lokalnych a zadaniem koordynacji jest realizowany algorytm typu Gaussa––Seidela, same zaś zadania lokalne są rozwiązywane niezależnie od siebie, poprzez algorytm Jacobiego. Zadanie koordynacji (8.60) jest rozwiązywane po otrzymaniu optymalnych wartości podwektorów $\hat{y}_i(\lambda^{(k)}, \mu^{(k)}) = y_i^{(k+1)}$ i $\hat{v}_i(\lambda^{(k)}, \mu^{(k)}) = v_i^{(k+1)}$, $i = 1, \ldots, p-1$. Zadania lokalne (8.59) są następnie rozwiązywane dla nowego wektora mnożników $\lambda^{(k+1)}$ i $\mu^{(k+1)}$, itd.

Stosowanie metody cen jest w porównaniu z metodą bezpośrednią opłacone wprowadzeniem dodatkowych zmiennych λ i μ, zwiększeniem wymiarowości zadań lokalnych oraz, ewentualnie, zwiększeniem liczby ograniczeń w zadaniach lokalnych (dochodzą ograniczenia na zmienne v, $d_i(v_i) \leqslant 0$). W zamian za to uzyskujemy łatwe do rozwiązania zadanie nadrzędne. W prosty sposób można wyznaczyć gradient funkcji koordynatora. Jego części wyrażają się wzorami:

$$\nabla_{\mu_j} \varphi(\lambda, \mu) = \left[\sum_{i=1}^{p-1} A_{ji} \hat{v}_i - s_j \right]^T \tag{8.61}$$

$$\nabla_{\lambda} \varphi(\lambda, \mu) = \left[\sum_{i=1}^{p-1} z_i(\hat{v}_i, \hat{y}_i) - w \right]^T \tag{8.62}$$

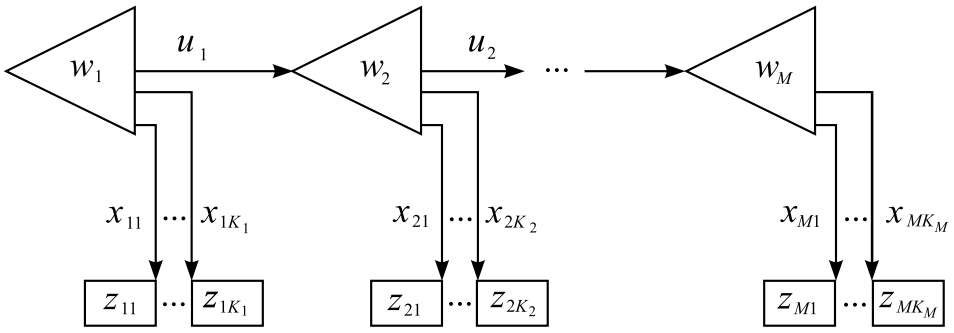
Rozwiązanie zadania koordynatora nie wymaga dużego nakładu obliczeń, a główny wysiłek obliczeniowy jest związany z zadaniami lokalnymi, które mogą być rozwiązywane jednocześnie. Równoległa realizacja metody umożliwia skrócenie czasu obliczeń w każdej iteracji algorytmu z $t^{(k)} = \sum_{i=1}^{p-1} t_i^{(k)} + t_k^{(k)}$ do $t^{(k)} = \max_{i=1\ldots,p-1} t_i^{(k)} + t_k^{(k)}$, gdzie $t_i^{(k)}$ oznacza czas rozwiązania i-tego zadania lokalnego, $t_k^{(k)}$ czas rozwiązania zadania koordynatora. W związku z tym, że $t_k^{(k)} \ll t_i^{(k)}$, zrównoleglenie metody cen powinno przynieść większe efekty niż w przypadku metody bezpośredniej.

8.4.5. Przykłady zastosowania metod optymalizacji hierarchicznej

Przedstawimy teraz dwa przykłady zastosowania obu metod optymalizacji hierarchicznej. Rozważymy system powiązany oraz system o wspólnych zasobach.

PRZYKŁAD 8.3. System powiązany.

Rozważamy kaskadę M zbiorników wodnych przedstawioną na rys. 8.5. W systemie jest K odbiorców wody. Każdy ze zbiorników zasila odpowiednio K_i odbiorców, gdzie i oznacza numer zbiornika $i = 1, \ldots, M$, tak więc $K = \sum_{i=1}^{M} K_i$. Zapasy wody i-tego zbiornika oznaczymy przez w_i, objętość wypływu z i-tego zbiornika do k-tego odbiorcy przez x_{ik}, objętość wypływu z i-tego zbiornika do następnego zbiornika w sieci przez x_{K+i} (czyli $x_{K+i} = u_i$), zaś zapotrzebowanie k-tego odbiorcy pobierającego wodę z i-tego zbiornika przez z_{ik}. Przyjmiemy założenie, że odpływ



Rys. 8.5. System zbiorników wodnych

z i-tego zbiornika jest równy dopływowi do $(i+1)$-tego zbiornika oraz dopływ do pierwszego zbiornika i odpływ z ostatniego zbiornika w systemie są zerowe. Naszym zadaniem jest zapewnienie, w miarę możliwości realizacji potrzeb odbiorców zbiorników. Możemy więc sformułować zadanie optymalizacji, w którym funkcja celu jest „karą" za niedostosowanie dostaw wody do zapotrzebowań, a ograniczenia wynikają z limitowanej pojemności zbiorników. Zmiennymi decyzyjnymi są wypływy ze zbiorników x.

$$\min_x \left[f(x) = \sum_{i=1}^{M} \sum_{k=1}^{K_i} (z_{ik} - x_{ik})^2 \right] \tag{8.63}$$

$$\begin{aligned} g_i(x) &= \sum_{k=1}^{K_i} x_{ik} + x_{K+i} - x_{K+i-1} - w_i \leqslant 0, \quad i = 1, \ldots, M \\ & x_i \geqslant 0, \quad x_{ik} \geqslant 0, \quad k = 1, \ldots, K_i, \quad i = 1, \ldots, M \end{aligned} \tag{8.64}$$

Zastosujmy do rozwiązania zadania metodę bezpośrednią i metodę cen.

Metoda bezpośrednia

Zaczniemy od podziału zmiennych na lokalne y i koordynujące v. Ze względu na postać zadania podział ten jest oczywisty, przyjmiemy $x =$

$= [y, v] = [y_1, y_2, \ldots, y_M, v]$, gdzie $y_i = [y_{i1}, y_{i2}, \ldots, y_{iK_i}] = [x_{i1}, x_{i2}, \ldots, x_{iK_i}]$ jest wektorem wypływów z i-tego zbiornika do jego odbiorców oraz $v = [u_1, u_2, \ldots, u_M] = [x_{K+1}, x_{K+2}, \ldots, x_{K+M}]$ wektorem odpływów ze zbiorników do kolejnych zbiorników w kaskadzie. Możemy więc sformułować zadania lokalne i koordynacji:

– zadanie lokalne i-te

$$\min_{y_{i1}, y_{i2}, \ldots, y_{iK_i}} \left[f_i(y_i, v) = \sum_{k=1}^{K_i} (z_{ik} - y_{ik})^2 \right] \tag{8.65}$$

$$g_i(y_i, v) = \sum_{k=1}^{K_i} y_{ik} + v_i - v_{i-1} - w_i \leqslant 0, \quad y_{ik} \geqslant 0, \quad k = 1, \ldots, K_i$$

– zadanie koordynacji

$$\min_{v} \left[\sum_{i=1}^{M} f_i(\hat{y}_i(v), v) \right] \tag{8.66}$$
$$v_i \geqslant 0, \quad v_i - w_i - v_{i-1} \leqslant 0, \quad i = 1, \ldots, M$$

Metoda cen

Przejdźmy teraz do metody cen i dokonajmy podziału zmiennych. W przeciwieństwie do metody bezpośredniej, dla każdego zbiornika i, zarówno w częściowych funkcjach celu f_i jak i ograniczeniach g_i, wystąpią różne wektory v_i. Przyjmiemy $x = [y, v] = [y_1, y_2, \ldots, y_M, v_1, v_2, \ldots, v_M]$, gdzie $y_i = [y_{i1}, y_{i2}, \ldots, y_{iK_i}] = [x_{i1}, x_{i2}, \ldots, x_{iK_i}]$ jest wektorem wypływów z i-tego zbiornika do jego odbiorców. Każda zmienna wiążąca v_i jest wektorem, $v_i = [v_{i1}, v_{i2}] = [x_{K+i-1}, x_{K+i}]$, gdzie v_{i1} oznacza dopływ do i-tego zbiornika, a v_{i2} odpływ z tego zbiornika do następnego w kaskadzie. Możemy więc sformułować zadania lokalne i koordynacji:

– zadanie lokalne i-te

$$\min_{(y_i, v_i)} \left[L_i(y_i, v_i, \mu) = \sum_{k=1}^{K_i} (z_{ik} - y_{ik})^2 - \mu_{i-1} v_{i1} + \mu_i v_{i2} \right] \tag{8.67}$$

$$g_i(y_i, v_i) = \sum_{k=1}^{K_i} y_{ik} - v_{i1} + v_{i2} - w_i \leqslant 0, \quad y_{ik} \geqslant 0, \quad k = 1, \ldots, K_i,$$
$$v_{i1} \geqslant 0, \quad v_{i2} \geqslant 0$$

– zadanie koordynacji

$$\max_{\mu} \left[\varphi(\mu) = \sum_{i=1}^{M} L_i(\hat{y}_i, \hat{v}_i, \mu) \right] \tag{8.68}$$

□

PRZYKŁAD 8.4. System o wspólnych zasobach.

Pewna firma przeznacza fundusze wielkości w_g na kampanię reklamową N produktów w M mediach. Skuteczność reklamy, określana wzrostem dochodów uzyskanych ze sprzedaży danego produktu, jest funkcją poniesionych na nią nakładów

$$f_i(x) = \alpha_i \prod_{j=1}^{M} x_{ij}^{\beta_{ij}} \tag{8.69}$$

gdzie x_{ij} oznacza nakłady finansowe na reklamę i-tego produktu w j-ty sposób, α_i – współczynnik skalujący, a β_{ij} – macierz współczynników elastyczności, a określających wrażliwość konsumentów i-tego produktu na j-tą reklamę.

Nakłady na reklamę produktu są ograniczone, $w_{i_{\min}} \leqslant \sum_{j=1}^{M} x_{ij} \leqslant w_{i_{\max}}$. Ograniczenia dotyczą również poszczególnych mediów, $w_{ij_{\min}} \leqslant x_{ij} \leqslant w_{ij_{\max}}$. Dodatkowo, mamy ograniczenie globalne związane z funduszem przeznaczonym na kampanię reklamową $\sum_{i=1}^{N}\sum_{j=1}^{M} x_{ij} \leqslant w_g$. Celem firmy jest, oczywiście maksymalizacja zysków.

Zapiszmy zadanie optymalizacji sformułowane w zbiorze

$$X = \left\{ x_{ij} : w_{i_{\min}} \leqslant \sum_{j=1}^{M} x_{ij} \leqslant w_{i_{\max}}, w_{ij_{\min}} \leqslant x_{ij} \leqslant w_{ij_{\max}}, \sum_{i=1}^{N}\sum_{j=1}^{M} x_{ij} \leqslant w_g \right\}$$

$$\max_{x_{ij}\in X} \sum_{i=1}^{N} \alpha_i \prod_{j=1}^{M} x_{ij}^{\beta_{ij}} \tag{8.70}$$

Wymiar zadania optymalizacji wynosi $N \cdot M$. Możemy je rozwiązać poprzez rozwiązanie N zadań lokalnych o wymiarze M oraz zadania koordynatora.

Metoda bezpośrednia

Zadanie lokalne polega na maksymalizacji zysków uzyskanych z reklamy i-tego produktu we wszystkich M mediach. Celem koordynatora jest optymalizacja rozdziału środków na reklamę każdego produktu w taki sposób, by nie zostało naruszone globalne ograniczenie na zasoby, jakim jest fundusz przeznaczony na kampanię reklamową w_g. Przyjmijmy następującą dekompozycję wektora zmiennych występujących w zadaniu: $x = [y_1, \ldots, y_N, v]$, gdzie zmienna v jest N-wymiarowym wektorem, którego elementy reprezentują nakłady przewidziane na reklamę poszczególnych produktów w różnych mediach. Zmienna lokalna każdego podsystemu $y_i = [y_{i1}, y_{i2}, \ldots, y_{iM}]$ jest M-wymiarowym wektorem, którego elementy oznaczają nakłady na reklamę

i-tego produktu w poszczególnych mediach. Zapiszmy zadania lokalne i koordynacji:

– zadanie lokalne i-te

$$\max_{y_i}\left[f_i(y_i)=\alpha_i\prod_{j=1}^{M}y_{ij}^{\beta_{ij}}\right] \tag{8.71}$$

$$w_{ij_{\min}}\leqslant y_{ij}\leqslant w_{ij_{\max}},\ \ j=1,\ldots,M$$

$$\sum_{j=1}^{M}y_{ij}\leqslant v_i$$

– zadanie koordynacji

$$\max_{v}\left[\varphi(v)=\sum_{i=1}^{N}f_i\left(\hat{y}_i(v)\right)\right] \tag{8.72}$$

$$w_{i_{\min}}\leqslant v_i\leqslant w_{i_{\max}},\ \ i=1,\ldots,N$$

$$\sum_{i=1}^{N}v_i\leqslant w_g$$

gdzie $\hat{y}_i(v)$ oznacza rozwiązanie i-tego zadania lokalnego dla zadanych przez koordynatora zmiennych v.

Metoda cen

Zastosujmy metodę cen. Zadanie lokalne będzie polegać na maksymalizacji zysków uzyskanych z reklamy i-tego produktu we wszystkich mediach. Celem koordynatora będzie zagwarantowanie spełnienia globalnego ograniczenia na zasoby – nieprzekroczenie funduszu przeznaczonego na kampanię reklamową, poprzez ustalanie odpowiedniej ceny za jednostkę pobranych zasobów finansowych. Ponieważ nie ma zmiennych, które nie występują w ograniczeniach globalnych, przyjmiemy, że wszystkie zmienne występujące w zadaniu są zmiennymi wiążącymi, a więc $v=x$;

– zadanie lokalne i-te

$$\max_{v_{ij}}\left[\alpha_i\prod_{j=1}^{M}v_{ij}^{\beta_{ij}}-\lambda\sum_{j=1}^{M}v_{ij}\right] \tag{8.73}$$

$$w_{i_{\min}}\leqslant\sum_{j=1}^{M}v_{ij}\leqslant w_{i_{\max}},\ \ w_{ij_{\min}}\leqslant v_{ij}\leqslant w_{ij_{\max}},\ \ j=1,\ldots,M$$

– zadanie koordynacji

$$\min_{\lambda \geqslant 0} \left[\varphi(\lambda) = \sum_{i=1}^{N} \alpha_i \prod_{j=1}^{M} \hat{v}_{ij}^{\beta_{ij}} - \lambda \left(\sum_{i=1}^{N} \sum_{j=1}^{M} \hat{v}_{ij} - w_g \right) \right] \tag{8.74}$$

Jednostka nadrzędna określa optymalną cenę za zużycie zasobu (funduszu na reklamę). W ten sposób optymalizowany jest rozdział środków między poszczególne produkty i media. Możemy natychmiast zauważyć, że w przypadkach, gdy spełnione jest ograniczenie globalne na zasoby, zadanie nadrzędne ustali cenę zerową ($\lambda = 0$). Jeżeli ograniczenie zostanie naruszone, koordynator będzie modyfikował cenę w taki sposób, by minimalizować różnicę $\sum_{i=1}^{N} \sum_{j=1}^{M} v_{ij} - w_g$. W celu wyznaczenia nowej wartości λ może skorzystać z gradientu $\nabla_\lambda \varphi(\lambda) = w_g - \sum_{i=1}^{N} \sum_{i=1}^{M} \hat{v}_{ij}$. $\square$

8.5. Programowanie dynamiczne

Rozważamy zadanie sterowania optymalnego ze wskaźnikiem jakości typu Bolzy:

$$\min_{u_0, u_1, \ldots, u_{N-1}} \sum_{k=0}^{N-1} g_k(x_k, u_k) + g_N(x_N) \tag{8.75}$$

gdzie sterowania u_k, $k = 0, 1, \ldots, N-1$ należą do zbioru $U_k(x_k)$, a stan x_k, $k = 1, 2, \ldots, N$, spełnia równanie stanu

$$x_{k+1} = f_k(x_k, u_k) \tag{8.76}$$

Zgodnie z zasadą optymalności Bellmana, optymalny koszt J_k dojścia od etapu k do końca czyli do N, przyjmując że układ na etapie k znajdował się w stanie x, wyznacza się według algorytmu:

$$J_k(x) = \min_{u \in U_k(x)} \left[g_k(x, u) + J_{k+1}(f_k(x, u)) \right] \tag{8.77}$$

Przyjmujemy $J_N(.) \equiv g_N(.)$. Wartość $J_k(x)$ wyznaczamy dla każdego $x \in \overline{X}$, gdzie $\overline{X}$ oznacza dyskretną aproksymację (siatkę) zbioru X.

Zadanie (8.77) można rozwiązywać niezależnie dla każdego $x \in \overline{X}$, możemy zatem podzielić zbiór $\overline{X}$ na p podzbiorów $\overline{X}_p$: $\overline{X} = \overline{X}_1 \cup \ldots \cup \overline{X}_p$ i każdemu podzbiorowi X_r przydzielić procesor r. Jest to tzw. metoda

równoległego stanu, zapewniająca dużą ziarnistość poszczególnych zadań równoległych. Przy przechodzeniu do kolejnego kroku (czyli poprzedniego etapu) musi nastąpić synchronizacja i ewentualna (w środowiskach z pamięcią lokalną) wymiana informacji (stablicowanych funkcji $J_k(x)$ dla $x \in \overline{X}_i$) między procesorami.

Rozdział 9

Algorytmy asynchroniczne

Andrzej Karbowski, Ewa Niewiadomska-Szynkiewicz

9.1.2.2. Układy równań nieliniowych

Poszukujemy rozwiązania układu równań nieliniowych

$$F(x) = 0 \tag{9.57}$$

gdzie $F : \mathbb{R}^n \longrightarrow \mathbb{R}^n$ jest funkcją różniczkowalną. O gradiencie $\nabla F(x)$ funkcji $F(x)$, czyli jej macierzy jakobianowej, zakładamy, iż jest on ograniczony, a dla pewnego zestawu wag również zdominowany diagonalnie. Dokładniej, jeśli oznaczymy pochodną cząstkową $\frac{\partial F_i}{\partial x_j}$ i-tej współrzędnej funkcji F względem j-tego argumentu jako $\nabla_j F_i$, to zakładamy iż

$$\nabla_i F_i(x) \leqslant K, \quad \forall i \quad \forall x \in \mathbb{R}^n \tag{9.58}$$

oraz że istnieje wektor wag dodatnich $w = [w_1, w_2, \ldots, w_n]$, dla którego zachodzi

$$\nabla_i F_i(x) w_i > \sum_{j \neq i} |\nabla_j F_i(x)| \, w_j \quad \forall i, \quad \forall x \in \mathbb{R}^n \tag{9.59}$$

Innymi słowy, zakładamy, że macierz utworzona z kolumn macierzy jakobianowej funkcji F przemnożonych przez odpowiednie wagi dodatnie jest zdominowana diagonalnie w całej dziedzinie. Założenie dodatniego znaku pochodnych cząstkowych $\nabla_i F_i(x) = \frac{\partial F_i}{\partial x_i}$ nieco zawęża klasę rozważanych zadań. Ważny jest tutaj w zasadzie stały znak pochodnej $\nabla_i F_i(x)$ w całej dziedzinie; jeśli ta pochodna jest ujemna, wystarczy w miejsce funkcji $F_i(x)$ w układzie równań (9.57) podstawić $-F_i(x)$.

Do algorytmu rozwiązania układu równań (9.57) dojdziemy, mnożąc go przez stałą dodatnią γ oraz dodając do obydwu stron wektor x. Otrzymamy w ten sposób algorytm:

$$x := h(x) = x - \gamma F(x) \tag{9.60}$$

Udowodnimy teraz, że dla kroku γ spełniającego warunek

$$0 < \gamma < \frac{1}{K} \tag{9.61}$$

odwzorowanie h jest zwężające w normie maksimum. W tym celu dla ustalonej współrzędnej i oraz wektorów $x, y \in \mathbb{R}^n$ zdefiniujmy funkcję $g_i : [0, 1] \to \mathbb{R}$

$$g_i(t) = t x_i + (1 - t) y_i - \gamma F_i \left(t x + (1 - t) y \right) \tag{9.62}$$

Zauważmy, że funkcja g_i jest różniczkowalna, a dla i-tej współrzędnej funkcji h możemy napisać

$$\begin{aligned} |h_i(x) - h_i(y)| &= |g_i(1) - g_i(0)| = \\ &= \left| \int_0^1 \frac{\mathrm{d}g_i(t)}{\mathrm{d}t} \mathrm{d}t \right| \leqslant \int_0^1 \left| \frac{\mathrm{d}g_i(t)}{\mathrm{d}t} \right| \mathrm{d}t \leqslant \max_{t\in[0,1]} \left| \frac{\mathrm{d}g_i(t)}{\mathrm{d}t} \right| \end{aligned} \tag{9.63}$$

Spróbujmy teraz oszacować moduł pochodnej $\frac{\mathrm{d}g_i}{\mathrm{d}t}$. Korzystając z definicji funkcji g_i (9.62) otrzymamy

$$\begin{aligned} \left| \frac{\mathrm{d}g_i(t)}{\mathrm{d}t} \right| &= \left| x_i - y_i - \gamma \sum_{j=1}^{n} \nabla_j F_i \Big(tx + (1-t)y \Big) (x_j - y_j) \right| = \\ &= \left| w_i \frac{x_i - y_i}{w_i} - \gamma \sum_{j=1}^{n} w_j \nabla_j F_i \Big(tx + (1-t)y \Big) \frac{x_j - y_j}{w_j} \right| = \\ &= \left| w_i \frac{x_i - y_i}{w_i} - \gamma w_i \nabla_i F_i \Big(tx + (1-t)y \Big) \frac{x_i - y_i}{w_i} - \right. \\ &\qquad \left. - \gamma \sum_{j\neq i} w_j \nabla_j F_i \Big(tx + (1-t)y \Big) \frac{x_j - y_j}{w_j} \right| \leqslant \\ &\leqslant \left| w_i - \gamma w_i \nabla_i F_i \Big(tx + (1-t)y \Big) \right| \left| \frac{x_i - y_i}{w_i} \right| + \\ &\quad + \sum_{j\neq i} \left| \gamma w_j \nabla_j F_i \Big(tx + (1-t)y \Big) \right| \left| \frac{x_j - y_j}{w_j} \right| \leqslant \\ &\leqslant \left(\sum_{j\neq i} \left| w_i - \gamma w_i \nabla_i F_i \Big(tx + (1-t)y \Big) \right| + \right. \\ &\qquad \left. + \sum_{j\neq i} \left| \gamma w_j \nabla_j F_i \Big(tx + (1-t)y \Big) \right| \right) \max_{j=1,\ldots,n} \left| \frac{x_j - y_j}{w_j} \right| \end{aligned} \tag{9.64}$$

Założyliśmy że wszystkie wagi $w_i,\ i = 1, \ldots, n$ oraz krok γ są dodatnie. Uwzględniając definicję ważonej normy maksimum (9.23), oszacowanie (9.64) możemy zapisać w następujący sposób:

$$\begin{aligned} \left| \frac{\mathrm{d}g_i(t)}{\mathrm{d}t} \right| \leqslant & \left(\left| w_i - \gamma w_i \nabla_i F_i \Big(tx + (1-t)y \Big) \right| + \right. \\ & \left. + \gamma \sum_{j\neq i} w_j \left| \nabla_j F_i \Big(tx + (1-t)y \Big) \right| \right) \|x - y\|_\infty^w \end{aligned} \tag{9.65}$$

Wykorzystamy teraz założenie, że krok $\gamma > 0$ jest ograniczony i spełnia warunek $\gamma < \frac{1}{K}$, gdzie, zgodnie z (9.58), K jest górnym oszacowaniem pochodnych cząstkowych $\nabla_i F_i$. Wówczas otrzymamy $\forall z \in \mathbb{R}^n$:

$$|w_i - \gamma w_i \nabla_i F_i(z)| = w_i \, |1 - \gamma \nabla_i F_i(z)| = w_i - w_i \gamma \nabla_i F_i(z) \tag{9.66}$$

Uwzględniając (9.66) w (9.65) uzyskamy

$$\begin{aligned} \left| \frac{\mathrm{d}g_i(t)}{\mathrm{d}t} \right| \leqslant & \left(w_i - \gamma w_i \nabla_i F_i \left(tx + (1-t)y \right) + \right. \\ & \left. + \gamma \sum_{j \neq i} w_j \left| \nabla_j F_i \left(tx + (1-t)y \right) \right| \right) \|x - y\|_\infty^w = \\ = & \left(w_i - \gamma \left[w_i \nabla_i F_i (tx + \left(1 - t)y \right) - \right. \right. \\ & \left. \left. - \sum_{j \neq i} w_j \left| \nabla_j F_i \left(tx + (1-t)y \right) \right| \right] \right) \|x - y\|_\infty^w \end{aligned} \tag{9.67}$$

Zauważmy teraz, że zgodnie z (9.59) wyrażenie zawarte w nawiasie kwadratowym jest dodatnie. Stąd w (9.67) otrzymamy ostatecznie

$$\left| \frac{\mathrm{d}g_i(t)}{\mathrm{d}t} \right| < w_i \, \|x - y\|_\infty^w \tag{9.68}$$

co, biorąc pod uwagę (9.63) oraz to, że oszacowanie (9.68) obowiązuje dla każdej współrzędnej $i = 1, \ldots, n$, oznacza, iż

$$\max_{i=1,\ldots,n} \frac{|h_i(x) - h_i(y)|}{w_i} = \|h(x) - h(y)\|_\infty^w < \|x - y\|_\infty^w \tag{9.69}$$

To kończy dowód, że h jest odwzorowaniem zwężającym w ważonej normie maksimum.

A zatem do rozwiązania układu równań nieliniowych (9.57) z funkcją F spełniającą warunki (9.58), (9.59) można wykorzystać algorytm (9.60), gdzie krok γ spełnia warunek (9.61), zrealizowany w wersji całkowicie asynchronicznej.

9.1.2.3. Optymalizacja bez ograniczeń metodą największego spadku

Zacznijmy od zadania optymalizacji funkcji kwadratowej, to znaczy

$$\min_{x \in \mathbb{R}^n} f(x) = \frac{1}{2} x' A x - b' x \tag{9.70}$$

Przy założeniu, że macierz formy kwadratowej A jest symetryczna (oczywiście, nie zmniejsza to ogólności rozważań, bo dla każdej formy kwadratowej można znaleźć jej macierz symetryczną) oraz dodatnio określona, czyli że funkcja f jest wypukła, warunkiem koniecznym i wystarczającym optymalności jest zerowanie się gradientu

$$\nabla f(x) = Ax - b = 0 \tag{9.71}$$

Zauważmy, że otrzymaliśmy nic innego, jak układ równań liniowych (9.28), którego sposób rozwiązania rozważaliśmy wcześniej. Przypomnijmy, należy zastosować algorytm (9.34)–(9.35) (dzieląc w dowolny sposób wektor x), w którym macierz D jest wyznaczona według wzoru (9.29), a współczynnik kroku γ spełnia warunki (9.54).

Takie same jak poprzednio będą też warunki zbieżności, z tą tylko różnicą, że – ze względu na dodatnią określoność macierzy A – elementy należące do głównej przekątnej muszą być dodatnie (inaczej wartość formy kwadratowej $x'Ax$ dla wektorów jednostkowych nie byłaby dodatnia). W związku z tym, w warunku bezwzględnej dominacji (9.53) po lewej stronie należy pominąć moduł i wówczas warunek ten przyjmie postać:

$$\exists\, w \in \mathbb{R}^n_+ : \quad a_{ii}\, w_i > \sum_{j \neq i} |a_{ij}|\, w_j \quad \forall i \tag{9.72}$$

Dodatkowo, ze względu na założoną symetrię macierzy A jest to równoważne

$$\exists\, w \in \mathbb{R}^n_+ : \quad a_{ii}\, w_i > \sum_{j \neq i} |a_{ji}|\, w_j \quad \forall i \tag{9.73}$$

czyli ważonej dominacji kolumnowej głównej przekątnej. Warunek ten jest w oczywisty sposób spełniony dla macierzy symetrycznych z dominacją głównej przekątnej. Jest on również spełniony dla pewnych macierzy symetrycznych i dodatnio określonych, w których ta dominacja nie występuje, np. dla

$$A = \begin{bmatrix} 1 & 1{,}5 & 1 \\ 1{,}5 & 6 & 1 \\ 1 & 1 & 8 \end{bmatrix} \tag{9.74}$$

jeśli przyjmiemy wagi $w_1 = 3$, $w_2 = w_3 = 1$. Niestety, nie dla wszystkich macierzy symetrycznych i dodatnio określonych można taki zestaw wag znaleźć. Łatwo sprawdzić, że jest to niemożliwe dla macierzy

$$A = \begin{bmatrix} 3 & 2 & 2 \\ 2 & 3 & 2 \\ 2 & 2 & 3 \end{bmatrix} \tag{9.75}$$

Ze wzoru (9.72) wynika, że identycznie jak w układach równań liniowych, w części wierszy (lub, w omawianym zadaniu, kolumn) musi wystąpić dominacja głównej przekątnej tak silna, by skompensować brak tej dominacji w innych wierszach (tutaj również kolumnach).

Analogicznie można postąpić z zadaniem optymalizacji nieliniowej bez ograniczeń

$$\min_{x\in\mathbb{R}^n} f(x) \tag{9.76}$$

gdzie $f : \mathbb{R}^n \to \mathbb{R},\ f \in C^2(\mathbb{R}^n)$. O funkcji f zakładamy, że jest wypukła, dzięki czemu warunek zerowania się gradientu tej funkcji jest konieczny i wystarczający dla osiągnięcia minimum. A zatem zadanie optymalizacji może być sprowadzone do rozwiązania układu równań nieliniowych

$$\nabla f(x) = 0 \tag{9.77}$$

Wystarczy teraz porównać to równanie z równaniem (9.57). Jeśli przyjmiemy

$$F(x) \equiv \nabla f(x) \tag{9.78}$$

to będziemy mogli zastosować algorytm (9.60) do otrzymania rozwiązania zadania optymalizacji (9.76). Będzie on miał tutaj postać

$$x := h(x) = x - \gamma \nabla f(x) \tag{9.79}$$

Warunki zbieżności będziemy badać przyjmując

$$\nabla F(x) \equiv \nabla^2 f(x) \tag{9.80}$$

Jak wynika z poprzednich rozważań, żeby odwzorowanie $h(x)$ było zwężające, muszą być spełnione warunki ograniczenia (9.58) oraz ważonej dominacji głównej przekątnej (9.59) macierzy jakobianowej funkcji F, czyli hesjanu funkcji optymalizowanej f, a także warunek ograniczenia kroku (9.61). Warunki te przyjmą tutaj postać:

$$\exists K: \quad \frac{\partial^2 f}{\partial x_i^2} \leqslant K \quad \forall x \in \mathbb{R}^n, \quad \forall i \tag{9.81}$$

$$\sum_{j\neq i} \left| \frac{\partial^2 f}{\partial x_i \partial x_j} \right| w_j < \frac{\partial^2 f}{\partial x_i^2}\, w_i, \quad \forall i \tag{9.82}$$

oraz

$$0 < \gamma < \frac{1}{K} \tag{9.83}$$

9.1.2.4. Zadania optymalizacji z ograniczeniami

Rozważamy zadanie optymalizacji

$$\min_{x \in X} f(x) \tag{9.84}$$

gdzie $X = \prod_{i=1}^{p} X_i$ jest jest iloczynem kartezjańskim niepustych, domkniętych i wypukłych zbiorów $X_i \subset \mathbb{R}^{n_i}$, $i = 1, \ldots, p$, f zaś – funkcją wypukłą i różniczkowalną w $\mathbb{R}^n$, $n = \sum_{i=1}^{p} n_i$.

Przyjmijmy, że algorytm optymalizacji ma postać ogólną

$$x := h(x) \tag{9.85}$$

Załóżmy, że odwzorowanie h ma jeden punkt stały $\hat{x}$, będący rozwiązaniem zadania (9.84), oraz jest *pseudokontrakcją* w blokowej normie maksimum, czyli

$$\exists \alpha \in (0,1): \ \|h(x) - \hat{x}\|_{B\infty} \leqslant \alpha \|x - \hat{x}\|_{B\infty}, \quad \forall x \in X \tag{9.86}$$

Blokowa norma maksimum $\|.\|_{B\infty}$ jest zdefiniowana następująco:

$$\|x\|_{B\infty} = \max_{i=1,\ldots,p} \|x_i\|_i \tag{9.87}$$

gdzie $\|.\|_i$ jest dowolną normą w $\mathbb{R}^{n_i}$.

Definiując zbiór $X(k)$

$$X(k) = \left\{ x \in \mathbb{R}^n : \|x - \hat{x}\|_{B\infty} \leqslant \alpha^k \cdot \|x(0) - \hat{x}\|_{B\infty} \right\} \tag{9.88}$$

otrzymujemy zwężającą rodzinę zbiorów $\{X(k)\}$ w $\mathbb{R}^n$. A zatem, obowiązuje twierdzenie 9.1 o zbieżności asynchronicznej, czyli odwzorowanie (9.85) może być realizowane przy dowolnej partycji wektora x w wersji całkowicie asynchronicznej.

Warunek pseudokontrakcji (9.86) jest dość często spełniony w algorytmach optymalizacji (np. w algorytmach gradientowych z projekcją na ograniczenia), należy jednak pamiętać, że nie wystarczy, by dany algorytm optymalizacyjny był algorytmem spadku – musi też być algorytmem zbliżenia (po każdej iteracji) do rozwiązania przynajmniej w tej podprzestrzeni X_i, w której odległość ta była największa.

9.1.2.5. Sieci neuronowe z pamięcią (Hopfielda)

W wielu zastosowaniach sieci neuronowych, na przykład przy rozpoznawaniu obrazów, optymalizacji kombinatorycznej, poszukuje się takiego punktu $x \in \mathbb{R}^n$, że

$$x_i = \sigma\left(\sum_{j=1}^{n} w_{ij}x_j + u_i\right) \tag{9.89}$$

gdzie $\sigma(.)$ to funkcja sigmoidalna (ciągła, rosnąca monotonicznie od -1 do 1) taka, że

$$0 < \frac{\mathrm{d}\sigma}{\mathrm{d}y} < 1 \tag{9.90}$$

We wzorze (9.89) $w_{ij}, u_i, \;\; i,j = 1,\ldots,n$, są danymi liczbami, przy czym $\forall i \;\; \sum\limits_{j=1}^{n} |w_{ij}| \leqslant 1$. Łatwo się przekonać, iż algorytm oparty na równaniu (9.89) jest odwzorowaniem zwężającym w normie maksimum.

9.1.2.6. Markowowskie procesy decyzyjne

Będziemy poszukiwać optymalnej polityki decyzyjnej na horyzoncie nieskończonym, w przypadku, gdy kryterium oceny jest wartością oczekiwaną zdyskontowanego kosztu całkowitego:

$$\min_{\pi}\left\{J(\pi, x_0) = \lim_{N\to\infty} \mathop{\mathrm{E}}_{w}\left(\sum_{k=0}^{N} \alpha^k \cdot g(x_k, u_k, w_k)\right)\right\} \tag{9.91}$$

gdzie x_k jest stanem, u_k – sterowaniem, w_k – zakłóceniem losowym na k-tym etapie, $w = \{w_0, w_1, w_2, \ldots\}$ – ciągiem wszystkich zakłóceń, π zaś jest polityką sterowania (ciągiem reguł decyzyjnych $\{\mu_0(.), \mu_1(.), \ldots,\}$; każda reguła $\mu_k(.)$ jest funkcją z przestrzeni stanów w przestrzeń sterowań, czyli inaczej funkcją przyporządkowującą zmierzonemu stanowi decyzję)

$$\pi = \{\mu_0(.), \mu_1(.), \mu_2(.), \ldots\} \tag{9.92}$$

$$u_k = \mu_k(x_k) \in U(x_k), \quad k = 0, 1, 2, \ldots \tag{9.93}$$

$$x_{k+1} = f(x_k, u_k, w_k), \quad k = 0, 1, 2, \ldots \tag{9.94}$$

Współczynnik α spełnia warunek:

$$0 < \alpha < 1 \tag{9.95}$$

i w związku z tym nazywany jest współczynnikiem dyskonta. Zakładamy, że zakłócenia w_k należą do skończonego zbioru D oraz mają identyczne rozkłady $P(.|x_k, u_k)$ określone na tym zbiorze.

Zakładamy również, że koszt etapowy jest jednostajnie ograniczony, tzn.

$$0 \leqslant g(x,u,w) \leqslant M, \quad \forall (x,u,w) \in S \times C \times D \tag{9.96}$$

Założenie nieskończonego horyzontu decyzyjnego jest pewną idealizacją matematyczną, stosowaną jednak często w zadaniach, w których liczba etapów jest bardzo duża, a ścisłe określenie chwili końcowej byłoby rzeczą nienaturalną. Z sytuacjami takimi mamy do czynienia, np. w systemach środowiskowych i w sieciach telekomunikacyjnych.

Oznaczmy dla dowolnej funkcji $J : S \to \mathbb{R}$ oraz $\forall x \in S$

$$h(J)(x) = \min_{u \in U(x)} \mathop{\mathrm{E}}_{w} \{g(x,u,w) + \alpha J\left(f(x,u,w)\right)\} \tag{9.97}$$

Często przyjmuje się, że zbiory stanów i sterowań są skończone (zbiór zakłóceń może być skończony, ale nie musi). Oznacza to, iż możemy reprezentować układ dynamiczny (9.94) jako sterowany łańcuch Markowa. Taki proces sterowania nazywany bywa markowowskim procesem decyzyjnym (ang. *Markov Decision Process*). Oznaczmy:

$$S = \{1, 2, \ldots, n\} \tag{9.98}$$

co odpowiada ponumerowaniu wszystkich możliwych punktów przestrzeni stanów, czyli w reprezentacji dyskretnej, kombinacji poziomów poszczególnych współrzędnych wektora stanu oraz

$$p^{ij}(u) = P(x_{k+1} = j | x_k = i, u_k = u), \quad i,j \in S, \ u \in U(i) \tag{9.99}$$

Prawdopodobieństwa przejścia $p^{ij}(u)$ mogą być dane *a priori*, a jeśli nie są, to wyliczamy je korzystając z równania stanu (9.94) i rozkładu prawdopodobieństwa zakłoceń $P(.|x,u)$ w następujący sposób

$$p^{ij}(u) = P(W^{ij}(u)|i,u) \tag{9.100}$$

gdzie

$$W^{ij}(u) = \{w \in D | f(i,u,w) = j\} \tag{9.101}$$

Dla uproszczenia przyjmiemy tutaj, że koszt etapowy nie zależy od zakłóceń.

Odwzorowanie (9.97) można teraz zapisać jako

$$h(J)(i) = \min_{u \in U(i)} \left[g(i,u) + \alpha \sum_{j=1}^{n} p^{ij}(u) J(j) \right] \tag{9.102}$$

Nietrudno udowodnić, że odwzorowanie h dane wzorem (9.97) lub (9.102) jest zwężające w normie maksimum w przestrzeni funkcji jednostajnie ograniczonych. W związku z tym, możemy zastosować twierdzenie

Banacha o punkcie stałym. Wynika z niego, że odwzorowanie h ma dokładnie jeden punkt stały, pewną funkcję $\hat{J}$, i że ten punkt można uzyskać za pomocą algorytmu kolejnych przybliżeń

$$J := h(J) \tag{9.103}$$

Oczywiście, algorytm ten może być realizowany przy dowolnej partycji przestrzeni stanów i w wersji asynchronicznej.

Reguła sterowania $\hat{\mu}(.)$ – odpowiadająca funkcji optymalnej $\hat{J}(.)$, taka że

$$\hat{\mu}(x) = \arg \min_{u \in U(x)} \mathrm{E}_w \left\{ g(x,u,w) + \alpha \hat{J}\left(f(x,u,w)\right) \right\} \tag{9.104}$$

lub w przypadku reprezentacji układu jako sterowanego łańcucha Markowa:

$$\hat{\mu}(i) = \arg \min_{u \in U(i)} \left[g(i,u) + \alpha \cdot \sum_{j=1}^{n} p^{ij}(u) J(j) \right] \tag{9.105}$$

określi nam optymalną politykę sterowania

$$\hat{\hat{\pi}} = \{\hat{\mu}(.), \hat{\mu}(.), \hat{\mu}(.), \ldots\} \tag{9.106}$$

9.1.3. Odwzorowania zachowujące porządek

Zakładamy że:
1) funkcja $h : \mathbb{R}^n \to \mathbb{R}^n$ jest ciągła,
2) funkcja h zachowuje porządek[2], tzn.

$$x \leqslant y \Rightarrow h(x) \leqslant h(y) \tag{9.107}$$

3) istnieje jeden punkt stały $\hat{x}$ odwzorowania h w X,
4) istnieją dwa takie wektory $\underline{x}, \bar{x} \in X$, że:

$$\underline{x} \leqslant h(\underline{x}) \leqslant h(\bar{x}) \leqslant \bar{x} \tag{9.108}$$

$$\lim_{k \to \infty} h^k(\underline{x}) = \lim_{k \to \infty} h^k(\bar{x}) = \hat{x} \tag{9.109}$$

gdzie h^k oznacza k-krotne złożenie odwzorowania h.

Definiujemy tutaj następujący ciąg zbiorów $X(k)$:

$$X(k) = \left\{ x : h^k(\underline{x}) \leqslant x \leqslant h^k(\bar{x}) \right\} \tag{9.110}$$

[2] W książce [6] nazywa się tę własność – niezbyt ściśle – monotonicznością.

W pierwszym z nich stosuje się „hop count metrics", to znaczy zlicza po prostu wszystkie elementarne odcinki od komputera do komputera. W drugim protokole brana jest pod uwagę „network delay metrics", to znaczy mierzy się czas przesyłania informacji. W stanie aktywnym komunikaty służące do optymalizacji tablic routingu (najkrótszych ścieżek) są wysyłane przez każdy z komputerów do wszystkich najbliższych sąsiadów co 30 sekund.

9.1.3.2. Rozwiązywanie układów równań różniczkowych zwyczajnych – zagadnienie Cauchy'ego

Rozpatrujemy układ dynamiczny składający się z p połączonych podukładów. Niech $x_i(t) \in \mathbb{R}^{n_i}$ będzie wektorem stanu i-tego podukładu w chwili t. Zakładamy, że mamy dany stan początkowy $x_i(0)$ dla każdego podukładu oraz że dynamikę opisuje układ równań:

$$\frac{\mathrm{d}x_i(t)}{\mathrm{d}t} + D_i(t)x_i(t) = \sum_{j=1}^{p} B_{ij}(t)x_j(t) + u_i(t), \quad i = 1, \ldots, p \tag{9.119}$$

$D_i(t)$ jest tutaj macierzą o wymiarze $n_i \times n_i$ opisującą wewnętrzną dynamikę i-tego podsystemu, $B_{ij}(t)$ to macierze o wymiarach $n_i \times n_j$ opisujące interakcje, u_i zaś jest znanym wektorem o wymiarze n_i.

Zakładamy, że wszystkie elementy $D_i(.)$, $B_{ij}(.)$ oraz $u_i(.)$ są ciągłymi i ograniczonymi funkcjami czasu na przedziale $[0, T]$. To gwarantuje, że układ równań (9.119) ma jednoznaczne rozwiązanie w tym przedziale.

Rozważamy zatem układy równań różniczkowych zwyczajnych liniowych, niestacjonarnych i niejednorodnych.

Do rozwiązania takiego zadania można zastosować algorytm całkowicie asynchroniczny, który nosi nazwę algorytmu chaotycznej relaksacji. Polega on na rozwiązaniu równania różniczkowego (9.119) przy podstawieniu po prawej stronie otrzymanych estymat trajektorii podsystemów. Załóżmy, że

$$x_i(.) \in X_i \quad i = 1, \ldots, p \tag{9.120}$$

gdzie X_i to zbiór funkcji ciągłych na przedziale, na którym poszukujemy rozwiązania równania różniczkowego.

Odwzorowanie h_i definiujemy w następujący sposób

$$h_i : y_i(.) = h_i\left(x_1(.), \ldots, x_m(.)\right) \tag{9.121}$$

gdzie $y_i(.)$ jest rozwiązaniem równania różniczkowego

$$\frac{\mathrm{d}y_i(t)}{\mathrm{d}t} + D_i(t)y_i(t) = \sum_{j=1}^{p} B_{ij}(t) \cdot x_j(t) + u_i(t) \tag{9.122}$$

dla danego $y_i(0) = x_i(0)$. Wystarczy zatem mieć metodę numeryczną rozwiązywania równań różniczkowych dla podsystemu.

Można udowodnić zbieżność punktową $x_i(.)$ do rozwiązania układu równań różniczkowych (9.119) (w dowodzie twierdzenia 9.1 o zbieżności asynchronicznej nie zakładaliśmy skończonej wymiarowości przestrzeni X_i).

9.1.3.3. Rozwiązywanie układów równań różniczkowych zwyczajnych – zagadnienie dwugraniczne

Zamiast warunków początkowych dla wszystkich współrzędnych wektora funkcji $x(.)$ mamy tutaj warunki początkowe dla części współrzędnych (zbiór I) i końcowe dla pozostałych (zbiór J).

W tym przypadku można również stosować algorytm relaksacyjny. Równania dla $x_i(.),\ i \in I$ – są całkowane w czasie w przód, a dla $x_j(.),\ j \in J$ – w tył.

Algorytm asynchroniczny jest zbieżny, gdy spełniony jest dodatkowo warunek dominacji diagonali w macierzy

$$D = \begin{bmatrix} D_1 & & & \\ & D_2 & & \\ & & \ddots & \\ & & & D_p \end{bmatrix} \tag{9.123}$$

Twierdzenie 9.3.

Załóżmy, że elementy diagonali macierzy D są dodatnie dla tych podsystemów, dla których są dane warunki początkowe, i ujemne dla tych, dla których są dane warunki końcowe. Załóżmy, że istnieje wektor $w > 0$ oraz pewien $\varepsilon > 0$ takie, że

$$\begin{aligned} |D_{i,ll}(t)|\,(1-\varepsilon)w_{i,l} > & \sum_{\{m|m\neq l\}} |D_{i,lm}(t)|\,w_{i,m} + \\ & + \sum_{j=1}^{p} \sum_{m=1}^{n_j} |B_{ij,lm}(t)|\,w_{j,m} \quad \forall i,j,l,t \end{aligned} \tag{9.124}$$

Wówczas dane zadanie dwugraniczne ma jednoznaczne rozwiązanie, a ciąg generowany przez algorytm całkowicie asynchroniczny jest zbieżny do tego rozwiązania.

9.1.3.4. Zadania optymalizacji przepływów w sieciach z nieliniowymi funkcjami kosztów

Przykład ten – zarówno sformułowanie zadania jak i podstawowe własności – zaczerpnięto z pracy [6]. Nieco inne są oznaczenia i uzasadnienie poprawności algorytmów.

Rozpatrujemy graf skierowany, oznaczając jako N zbiór węzłów oraz jako A zbiór łuków. Z każdym łukiem $(i,j) \in A$ jest związana ściśle wypukła funkcja $a_{ij}(f_{ij})$ kosztu realizacji przepływu f_{ij} od węzła i do j oraz kostkowe ograniczenia na ten przepływ

$$b_{ij} \leqslant f_{ij} \leqslant c_{ij} \tag{9.125}$$

Z kolei z każdym węzłem $i \in N$ jest związane dane zasilanie $s_i > 0$, zapotrzebowanie $s_i < 0$, WST. Naszym celem jest taki rozkład przepływów między węzłami, by realizować wszystkie zapotrzebowania przy minimalnym koszcie łącznym przepływów, czyli

WST:
lub dany węzeł jedynie pośredniczy, tzn. tworzy strukturę sieci nie będąc ani źródłem, ani odbiorcą s_i=0

$$\min_f \sum_{(i,j)\in A} a_{ij}(f_{ij}) \tag{9.126}$$

$$\sum_{\{j|(j,i)\in A\}} f_{ji} + s_i = \sum_{\{j|(i,j)\in A\}} f_{ij}, \quad \forall i \in N \tag{9.127}$$

$$b_{ij} \leqslant f_{ij} \leqslant c_{ij}, \quad \forall\,(i,j) \in A \tag{9.128}$$

gdzie f to wektor wszystkich przepływów. Równania (9.127) wynikają z bilansu przepływów w węzłach (tzw. I prawa Kirchhoffa). O zapotrzebowaniach i zasilaniach s_i zakładamy, że się bilansują w sieci, czyli:

$$\sum_{i\in N} s_i = 0 \tag{9.129}$$

Żeby rozwiązać powyższe zadanie, sformułujmy dla niego funkcję Lagrange'a, ale uwzględniając jedynie ograniczenia bilansowe (9.127). Jeśli mnożnik Lagrange'a odpowiadający równaniu i-tego węzła oznaczymy jako μ_i, funkcja ta będzie następująca

$$\begin{aligned} L(f,\mu) = &\sum_{(i,j)\in A} a_{ij}(f_{ij}) + \\ &+ \sum_{i\in N} \mu_i \left(\sum_{\{j|(j,i)\in A\}} f_{ji} + s_i - \sum_{\{j|(i,j)\in A\}} f_{ij} \right) \end{aligned} \tag{9.130}$$

Dokonując prostych przekształceń, funkcję Lagrange'a (9.130) można przedstawić w następujący sposób:

$$\begin{aligned} L(f,\mu) = &\sum_{(i,j)\in A} a_{ij}(f_{ij}) + \sum_{i\in N} \mu_i \sum_{\{j|(j,i)\in A\}} f_{ji} - \\ &- \sum_{i\in N} \mu_i \sum_{\{j|(i,j)\in A\}} f_{ij} + \sum_{i\in N} \mu_i s_i = \end{aligned} \tag{9.131}$$

$$\mu_1 = r \tag{9.134}$$

$$= \sum_{(i,j)\in A} a_{ij}(f_{ij}) + \sum_{j\in N} \mu_j \sum_{\{i|(i,j)\in A\}} f_{ij} -$$

$$- \sum_{i\in N} \mu_i \sum_{\{j|(i,j)\in A\}} f_{ij} + \sum_{i\in N} \mu_i s_i = \qquad \text{(131 cd.)}$$

$$= \sum_{(i,j)\in A} a_{ij}(f_{ij}) + \sum_{j\in N} \sum_{\{i|(i,j)\in A\}} \mu_j f_{ij} -$$

$$- \sum_{i\in N} \sum_{\{j|(i,j)\in A\}} \mu_i f_{ij} + \sum_{i\in N} \mu_i s_i =$$

$$= \sum_{(i,j)\in A} a_{ij}(f_{ij}) + \sum_{(i,j)\in A} \mu_j f_{ij} - \sum_{(i,j)\in A} \mu_i f_{ij} + \sum_{i\in N} \mu_i s_i =$$

$$= \sum_{(i,j)\in A} \left[a_{ij}(f_{ij}) - (\mu_i - \mu_j)\, f_{ij}\right] + \sum_{i\in N} \mu_i s_i$$

A zatem funkcja dualna w zadaniu (9.126)–(9.128) (patrz rozdz. 8.4.4) będzie miała postać

$$L_D(\mu) = \min_{b \leqslant f \leqslant c} \left\{ \sum_{(i,j)\in A} \left[a_{ij}(f_{ij}) - (\mu_i - \mu_j)\, f_{ij}\right] + \sum_{i\in N} \mu_i s_i \right\} =$$

$$= \sum_{(i,j)\in A} \left\{ \min_{b_{ij} \leqslant f_{ij} \leqslant c_{ij}} a_{ij}(f_{ij}) - (\mu_i - \mu_j)\, f_{ij} \right\} + \sum_{i\in N} \mu_i s_i \qquad (9.132)$$

Zgodnie z teorią dualności, rozwiązanie zadania wyjściowego (9.126)–(9.128) można uzyskać rozwiązując zadanie dualne

$$\max_{\mu\in\mathbb{R}^{\overline{\overline{N}}}} L_D(\mu) \qquad (9.133)$$

gdzie $\overline{\overline{N}}$ oznacza moc zbioru N, czyli liczbę węzłów. Zauważmy, że optymalne mnożniki Lagrange'a nie są wyznaczone jednoznacznie, gdyż można do każdego z nich dodać tę samą stałą, a ze względu na postać funkcji dualnej (9.132) oraz warunek (9.129) wartość tej funkcji się nie zmieni. W związku z tym, warto ustalić jedną ze współrzędnych wektora μ i na przykład przyjąć

$$\mu_1 = r \qquad (9.134)$$

gdzie r jest dowolną niezerową stałą rzeczywistą. Niech M będzie zbiorem wszystkich wektorów mnożników Lagrange'a, w których pierwsza współrzędna spełnia warunek (9.134). Oznaczmy teraz jako L_{ij} składnik funkcji dualnej odpowiadający łukowi (i,j)

$$L_{ij}(\mu_i - \mu_j) = \min_{b_{ij} \leqslant f_{ij} \leqslant c_{ij}} a_{ij}(f_{ij}) - (\mu_i - \mu_j)\, f_{ij} \tag{9.135}$$

Jest to, jak widać, zadanie optymalizacji skalarnej, odrębne dla każdego łuku, które łatwo jest rozwiązać, często nawet analitycznie. Funkcję dualną możemy teraz przedstawić w następujący sposób

$$L_D(\mu) = \sum_{(i,j)\in A} L_{ij}(\mu_i - \mu_j) + \sum_{i\in N} \mu_i s_i \tag{9.136}$$

Zgodnie z teorią dualności (patrz rozdz. 8.4.4) funkcja L_D oraz wszystkie funkcje L_{ij} są wklęsłe. Kolejne współrzędne gradientu funkcji L_D są równe:

$$\begin{aligned} \frac{\partial L_D}{\partial \mu_i} &= -\sum_{\{j|(j,i)\in A\}} L'_{ji}(\mu_j - \mu_i) + \sum_{\{j|(i,j)\in A\}} L'_{ij}(\mu_i - \mu_j) + s_i = \\ &= \sum_{\{j|(j,i)\in A\}} f_{ji} - \sum_{\{j|(i,j)\in A\}} f_{ij} + s_i \end{aligned} \tag{9.137}$$

Zdefiniujemy teraz dla każdego $i \in N$ odwzorowanie typu „punkt w zbiór" M_i, które przypisuje każdemu wektorowi cen μ przedział wszystkich cen, które maksymalizują funkcję dualną względem i-tej ceny μ_i, to znaczy:

$$\begin{aligned} M_i(\mu) &= \left\{ \xi : \frac{\partial L_D(\mu_1, \mu_2, \ldots, \mu_{i-1}, \xi, \mu_{i+1}, \ldots, \mu_{\overline{N}})}{\partial \mu_i} = 0 \right\} = \\ &= \left\{ \xi : \sum_{\{j|(j,i)\in A\}} L'_{ji}(\mu_j - \xi) = \sum_{\{j|(i,j)\in A\}} L'_{ij}(\xi - \mu_j) + s_i \right\} \end{aligned} \tag{9.138}$$

Można udowodnić, że zbiór $M_i(\mu)$ jest przedziałem domkniętym. Oznaczmy teraz jako $\underline{M}_i(\mu)$ oraz $\overline{M}_i(\mu)$ odpowiednio, lewy i prawy kraniec tego przedziału. Okazuje się, że obydwie te funkcje zachowują porządek. Przedstawimy dowód tej własności dla $\overline{M}_i$.

Weźmy dowolne $\underline{\mu}, \overline{\mu} \in M$, takie, że $\underline{\mu} \leqslant \overline{\mu}$. Ponieważ funkcje L_{ij} oraz L_{ji} są wklęsłe, więc ich pochodne L'_{ij} oraz L'_{ji} są funkcjami malejącymi, czyli

$$L'_{ji}\left(\overline{\mu}_j - \overline{M}_i(\underline{\mu})\right) \leqslant L'_{ji}\left(\underline{\mu}_j - \overline{M}_i(\underline{\mu})\right), \quad \forall (j,i) \in A \tag{9.139}$$

oraz

$$L'_{ij}\left(\overline{M}_i(\underline{\mu}) - \underline{\mu}_j)\right) \leqslant L'_{ij}\left(\overline{M}_i(\underline{\mu}) - \overline{\mu}_j)\right), \quad \forall (i,j) \in A \tag{9.140}$$

Stąd

$$\begin{aligned}\sum_{\{j|(i,j)\in A\}} L'_{ij}\left(\overline{M}_i(\underline{\mu}) - \overline{\mu}_j)\right) - \sum_{\{j|(j,i)\in A\}} L'_{ji}\left(\overline{\mu}_j - \overline{M}_i(\underline{\mu})\right) + s_i \geqslant \\ \geqslant \sum_{\{j|(i,j)\in A\}} L'_{ij}\left(\overline{M}_i(\underline{\mu}) - \underline{\mu}_j)\right) - \sum_{\{j|(j,i)\in A\}} L'_{ji}\left(\underline{\mu}_j - \overline{M}_i(\underline{\mu})\right) + s_i\end{aligned} \tag{9.141}$$

Zauważmy teraz, że z definicji (9.138) zbiorów $M_i(\mu)$, $i = 1, \ldots, \overline{\overline{N}}$, wynika, iż prawa strona nierówności (9.141) musi być równa zeru. Ponieważ z drugiej strony wartość wyrażenia

$$\sum_{\{j|(i,j)\in A\}} L'_{ij}(\xi - \overline{\mu}_j) - \sum_{\{j|(j,i)\in A\}} L'_{ji}(\overline{\mu}_j - \xi) + s_i$$

jest ujemna dla każdego $\xi > \overline{M}_i(\overline{\mu})$ (łatwo to sprawdzić w podobny sposób jak wyżej, korzystając z monotoniczności funkcji L'_{ij}), więc musi zachodzić $\overline{M}_i(\underline{\mu}) \leqslant \overline{M}_i(\overline{\mu})$. A zatem funkcja wektorowa $\overline{M}(\mu)$ zachowuje porządek.

W analogiczny sposób można przedstawić dowód zachowywania porządku przez funkcję $\underline{M}(\mu)$. A zatem algorytm iteracji mnożników Lagrange'a o postaci

$$\mu_i := \begin{cases} r & i = 1 \\ \gamma \underline{M}_i(\mu) + (1-\gamma)\overline{M}_i(\mu) & i = 2, 3, \ldots, \overline{\overline{N}} \end{cases} \tag{9.142}$$

będzie zbieżny w wersji całkowicie asynchronicznej dla każdego $\gamma \in [0, 1]$.

Łatwo sprawdzić, że również algorytm

$$\mu_i := \begin{cases} r & i = 1 \\ \gamma \cdot \mu_i + (1-\gamma) \arg \min\limits_{\xi \in M_i(\mu)} |\xi - \mu_i| & i = 2, 3, \ldots, \overline{\overline{N}} \end{cases} \tag{9.143}$$

definiuje odwzorowanie zachowujące porządek i w związku z tym ma zagwarantowaną zbieżność w realizacji całkowicie asynchronicznej.

9.1.4. Inne odwzorowania

W tym podrozdziale przedstawimy algorytmy numeryczne, które warto jest realizować w wersji całkowicie asynchronicznej (tzn. mające zagwarantowaną zbieżność w tej implementacji), a które wykraczają poza przedstawiony wyżej model Bertsekasa i Tsitsiklisa.

9.1.4.1. Metody zmiennej metryki

W rozdziale 8.3.1 mówiliśmy o możliwościach zrównoleglenia metod Newtona i zmiennej metryki, przy czym rozważaliśmy podejście synchroniczne. Można podać również asynchroniczne wersje omawianych metod, gdzie każdy z procesów wykonuje operacje różnego typu. W pracy [18] zaproponowano asynchroniczną wersję metody zmiennej metryki z aproksymacją hesjanu realizowaną zgodnie z formułą BFGS (8.12)–(8.13) oraz przedstawiono sposób realizacji algorytmu.

Poszukujemy rozwiązania zadania $\min_{x\in\mathbb{R}^n} f(x)$. Wersja sekwencyjna proponowanego algorytmu optymalizacji jest następująca (patrz rozdz. 8.3.1):

Iteracja k-ta

- Mając dany punkt $x^{(k)}$ i macierz $G^{(k)}$ będącą aproksymacją hesjanu, wyznacz kierunek poszukiwań $d^{(k)}$, rozwiązując układ równań $G^{(k)}d^{(k)} = -g(x^{(k)})$, gdzie $g(x^{(k)})$ oznacza gradient.
- Oblicz współczynnik kroku $\alpha^{(k)}$, nowy punkt $x^{(k+1)} = x^{(k)} + \alpha^{(k)}d^{(k)}$ i gradient w tym punkcie.
- Wyznacz nową macierz $G^{(k+1)}$ korzystając z formuły BFGS, (8.12)–(8.13).

Asynchroniczna wersja metody zakłada podział algorytmu na cztery zadania wykonywane asynchronicznie przez różne procesy. Procesy komunikują się w układzie producent-konsument, przy czym zakładamy, że proces i jest producentem, a proces j – konsumentem. Niech $\zeta_i^{(k_j)}$ będzie wyznaczaną asynchronicznie zmienną. Reprezentuje ona wartość zmiennej ζ obliczoną przez i-ty proces, wysłaną do j-ego procesu i wykorzystaną przez j-ty proces w k_j iteracji.

Inicjalizacja: $x = x^{(0)}$, $g = g^{(0)} = g(x^{(0)})$, $d = -g^{(0)}$, $G = I$.

ZADANIE 1:

Wyznaczenie macierzy $G_1^{(k_1+1)}$, przy wykorzystaniu formuły BFGS (8.12)–(8.13) dla aktualnych wartości wejść $x_4^{(k_1+1)}$ i $g_4^{(k_1+1)}$.

ZADANIE 2:

Faktoryzacja macierzy $G_1^{(k_2)}$ aktualnie dostępnej na wejściu (np. wykorzystanie metody Cholesky'ego). Wyznaczenie macierzy trójkątnej L:

$$G_1^{(k_2)} = L_2^{(k_2)}[L_2^{(k_2)}]^T$$

ZADANIE 3:

Wyznaczenie kierunku $d_3^{(k_3)}$ dla aktualnych wartości wejść: gradientu $g_4^{(k_3)}$ i macierzy $L_2^{(k_3)}$ po faktoryzacji, poprzez rozwiązanie układu równań liniowych

$$L_2^{(k_3)}[L_2^{(k_3)}]^T d_3^{(k_3)} = -g_4^{(k_3)}$$

ZADANIE 4:

Wyznaczenie nowego punktu $x_4^{(k_4+1)}$ oraz gradientu $g_4^{(k_4+1)}$ dla aktualnej wartości wejścia – kierunku $d_3^{(k_4)}$ i obliczonych w poprzedniej iteracji k_4 wartości $x_4^{(k_4)}$ i $g_4^{(k_4)}$.

Wykonywane są następujące obliczenia:

- sprawdzenie, czy kierunek $d_3^{(k_4)}$ jest kierunkiem poprawy (tj. czy spełnia warunek $[g_4^{(k_4)}]^T d_3^{(k_4)} < 0$); w przeciwnym przypadku przyjmuje się $d_3^{(k_4)} = -g_4^{(k_4)}$,
- wyznaczenie współczynnika kroku

$$\alpha^{(k_4)} = \max\{\overline{\alpha}^{(k_4)}, \overline{\alpha}^{(k_4)}/2, \overline{\alpha}^{(k_4)}/4, \ldots\}$$

gdzie $\overline{\alpha}^{(k_4)} \geqslant 0$ oraz $\alpha^{(k_4)}$ spełnia warunek

$$f(x_4^{(k_4)} + \alpha^{(k_4)} d_3^{(k_4)}) - f(x_4^{(k_4)}) \leqslant \rho\, \alpha^{(k_4)} [g_4^{(k_4)}]^T d_3^{(k_4)}$$

przy czym $\rho \in (0, 1/2)$,

- wyznaczenie nowego punktu

$$x_4^{(k_4+1)} = x_4^{(k_4)} + \alpha^{(k_4)} d_3^{(k_4)}$$

i obliczenie gradientu funkcji $g_4^{(k_4+1)}$ w tym punkcie.

Jako kryterium zatrzymania obliczeń proponuje się jeden z testów STOP-u: $||g_4^{(k_4)}|| \leqslant \varepsilon_1$ lub $|f(x_4^{(k_4-p)}) - f(x_4^{(k_4)})| < \varepsilon_2$, gdzie ε_1 i ε_2 oznaczają przyjętą dokładność, p liczbę kroków, po której porównuje się wyniki obliczeń.

Autorzy [18] przedstawiają dowód zbieżności oraz rezultaty eksperymentów numerycznych proponowanej metody. Badania wykonali dla następujących wartości parametrów: $\varrho = 0{,}01$, $p = 30$. Wyniki testów wskazują na wzrost efektywności algorytmu w stosunku do jego sekwencyjnej realizacji.

9.1.4.2. Równoległe metody uczenia statycznych sieci neuronowych

Jednym z najbardziej efektywnych narzędzi aproksymacji funkcji jest obecnie sieć neuronowa typu *feed-forward* (nazywana również *back-propagation*, lub typu perceptron). Składa się ona z wielu komórek (neuronów) tworzonych przez szeregowo połączone: sumator o wielu wejściach (z nastawianymi współczynnikami) i moduł statyczny o charakterystyce nieliniowej. Najczęściej jest to funkcja ciągła podobna do dystrybuanty, to znaczy monotonicznie rosnąca od 0 do 1 w przedziale $(-\infty, +\infty)$. W publikacjach na temat sieci neuronowych funkcja taka nazywana jest sigmoidem lub funkcją sigmoidalną.

W sieciach neuronowych wyjście każdego z neuronów może być jednym z wejść dla pewnej liczby innych neuronów. W sieciach typu *feed-forward* połączenia te zapewniają jednokierunkowy przepływ informacji (unilateralność), czyli nie ma sprzężeń zwrotnych i pamięci. Ponadto, wszystkie neurony są pogrupowane w warstwy, przy czym neurony tej samej warstwy mają jednakowe wejścia, a ich wyjścia połączone są z tymi samymi neuronami warstwy następnej. Wyróżnia się w związku z tym warstwę wejściową (dla niej wejściami są sygnały zewnętrzne, czyli argumenty funkcji, którą sieć ma aproksymować), wyjściową (wartości funkcji aproksymującej) oraz, między nimi, tzw. warstwy ukryte. Udowodniono, że za pomocą sieci składającej się przynajmniej z dwóch warstw (o odpowiedniej liczbie neuronów), przy czym zewnętrzna warstwa może być liniowa, czyli składać się wyłącznie z sumatorów, można dowolnie dokładnie aproksymować każde odwzorowanie nieliniowe [21], [37]. Cały kłopot polega na doborze struktury sieci (liczby neuronów w poszczególnych warstwach oraz, ewentualnie, liczby warstw), a także współczynników, przez które mnożone są wejścia (dla każdego neuronu z osobna), czyli tzw. wag synaptycznych (nastaw sieci).

Pierwszy z wymienionych problemów rozwiązuje się heurystycznie – nie ma obecnie jeszcze ścisłej procedury projektowej, drugi zaś za pomocą metod obliczeniowych optymalizacji, przedstawionych poniżej. Do uczenia tzw. wsadowego mogą być wykorzystane dowolne metody optymalizacji przedstawione w tym rozdziale. Do uczenia z węzłami losowanymi można wykorzystać metodę całkowicie asynchroniczną, realizowaną w architekturze typu *wielu producentów – jeden konsument.*

Aproksymacja funkcji za pomocą sieci neuronowej

Niech $f : \mathbb{R}^n \to \mathbb{R}^m$ będzie funkcją, którą chcemy aproksymować na podstawie jej znajomości w pewnych – przyjmijmy dla ustalenia uwagi r – punktach przestrzeni $\mathbb{R}^n$, a $\hat{f}(., w)$ funkcją aproksymującą, uzyskaną za pomocą sieci neuronowej, dla danego wektora wag synaptycznych w. Optymalne wartości wektora wag wyznacza się rozwiązując zadanie

$$\min_w \left\{ C(w) = \| f(.) - \hat{f}(. , w) \| = \sum_{i=1}^{r} \sum_{j=1}^{m} [f_j(x^i) - \hat{f}_j(x^i , w)]^2 \right\} \quad (9.144)$$

gdzie $\| . \|$ oznacza normę, $f_j(.)$, $\hat{f}_j(. , .)$ – j-tą współrzędną funkcji aproksymowanej i aproksymującej, a $x^i - i$-ty węzeł aproksymacji.

Wartość $\hat{f}(x, w)$ funkcji aproksymującej jest otrzymywana na wyjściu sieci, gdy na wejściu podawany jest wektor x. Zakładamy, że sieć składa się z L warstw, każda zaś warstwa z n_s neuronów, gdzie $s = 1, \ldots, L$ – indeks warstwy.

Neurony tworzące pierwszą warstwę możemy zatem opisać w następujący sposób:

– 1-sza warstwa, q-ta komórka, $q = 1, \ldots, n_1$

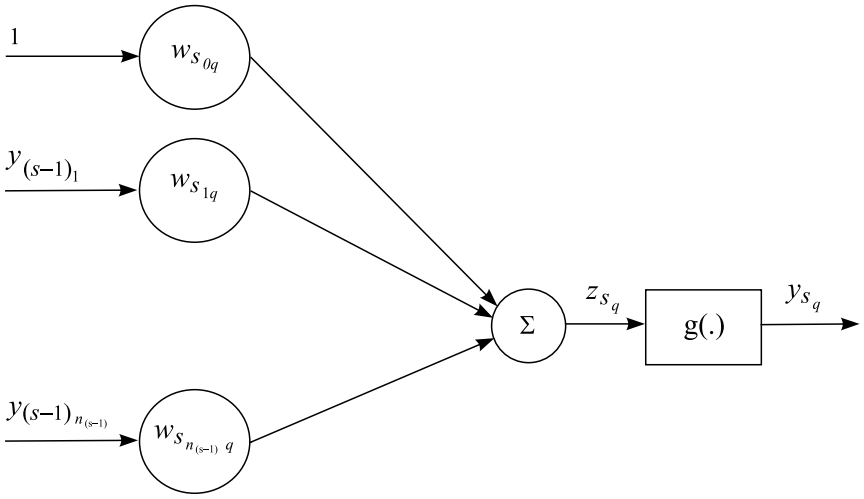
$$z_{1_q}(x, w) = \sum_{p=1}^{n} w_{1_{pq}} \cdot x_p + w_{1_{0q}} \tag{9.145}$$

$$y_{1_q}(x, w) = g\left(z_{1_q}(x, w)\right) \tag{9.146}$$

gdzie: $w_{1_{0q}}$ – składnik stały (można go traktować jako wagę dla wejścia stałowartościowego równego 1),
$z_{1_q}(x, w)$ – wyjście sumatora q-tej komórki,
$y_{1_q}(x, w)$ – wyjście q-tej komórki warstwy wejściowej,
$g(.)$ – funkcja sigmoidalna.

Jak już wspomniano, w następnych warstwach (wszystkich ukrytych i wyjściowej) wejściami są wyjścia wszystkich komórek warstwy poprzedniej.

Przyjmijmy, że opisujemy teraz q-tą komórkę s-tej warstwy (rys. 9.1) i oznaczamy przez $w_{s_{0q}}$ – składnik stały, $z_{s_q}(x, w)$ – wyjście sumatora q-tej komórki warstwy s-tej, $y_{s_q}(x, w)$ – wyjście q-tej komórki warstwy s-tej.



Rys. 9.1. Pojedynczy neuron (q-ta komórka warstwy s-tej)

– s-ta warstwa (wszystkie ukryte i wyjściowa), $s = 2, \ldots, L$, q-ta komórka, $q = 1, \ldots, n_s$

$$z_{s_q}(x, w) = \sum_{p=1}^{n_{s-1}} w_{s_{pq}} \cdot y_{s-1_p}(x, w) + w_{s_{0q}} \tag{9.147}$$

$$y_{s_q}(x, w) = g\left(z_{s_q}(x, w)\right) \tag{9.148}$$

Wyjście sieci będzie zaś opisane równaniem

– wyjście

$$\hat{f}(x,w) = \operatorname{diag}(w_{f_1}, w_{f_2}, \ldots, w_{f_m}) \cdot \begin{bmatrix} y_{L_1}(x,w) \\ y_{L_2}(x,w) \\ \vdots \\ y_{L_m}(x,w) \end{bmatrix} \tag{9.149}$$

lub, gdy warstwa zewnętrzna nie zawiera funkcji sigmoidalnych

$$\hat{f}(x,w) = \operatorname{diag}(w_{f_1}, w_{f_2}, \ldots, w_{f_m}) \cdot \begin{bmatrix} z_{L_1}(x,w) \\ z_{L_2}(x,w) \\ \vdots \\ z_{L_m}(x,w) \end{bmatrix} \tag{9.150}$$

Oczywiście, wszystkie wagi $\{w_{f_1}, w_{f_2}, \ldots, w_{f_m}\}$ są również elementami wektora zmiennych decyzyjnych w.

W powyższych równaniach $g(.)$ oznacza funkcję sigmoidalną. Może to być, na przykład, funkcja logistyczna

$$g(\xi) = \frac{1}{1 + \mathrm{e}^{-\xi}} \tag{9.151}$$

lub tangens hiperboliczny

$$g(\xi) = \operatorname{th}\xi = \frac{\mathrm{e}^{\xi} - \mathrm{e}^{-\xi}}{\mathrm{e}^{\xi} + \mathrm{e}^{-\xi}} \tag{9.152}$$

Optymalizacja nastaw sieci

Do optymalizacji nastaw sieci (wag synaptycznych oraz wag wyjściowych) najczęściej stosuje się metody gradientowe [15]. Gradient funkcji $C(w)$ wyznacza się korzystając ze wzorów

$$\frac{\partial C}{\partial w_{s_{pq}}} = \frac{\partial C}{\partial \hat{f}} \cdot \frac{\partial \hat{f}(x,w)}{\partial w_{s_{pq}}} \tag{9.153}$$

Biorąc pod uwagę (9.144):

$$\frac{\partial C}{\partial w_{s_{pq}}} = -2 \sum_{i=1}^{r} \sum_{j=1}^{m} \left[f_j(x^i) - \hat{f}_j(x^i, w) \right] \frac{\partial \hat{f}_j}{\partial w_{s_{pq}}}(x^i, w) \tag{9.154}$$

$$\frac{\partial C}{\partial w_{f_q}} = \frac{\partial C}{\partial \hat{f}} \begin{bmatrix} 0 \\ 0 \\ \vdots \\ y_{L_q}(x,w) \\ \vdots \\ 0 \\ 0 \end{bmatrix} \tag{9.155}$$

lub

$$\frac{\partial C}{\partial w_{f_q}} = \frac{\partial C}{\partial \hat{f}} \begin{bmatrix} 0 \\ 0 \\ \vdots \\ z_{L_q}(x,w) \\ \vdots \\ 0 \\ 0 \end{bmatrix} \tag{9.156}$$

$$\frac{\partial \hat{f}(x,w)}{\partial w_{s_{pq}}} = \frac{\partial \hat{f}(x,w)}{\partial z_{s_q}} \cdot \frac{\partial z_{s_q}(x,w)}{\partial w_{s_{pq}}} = \delta_{s_q}(x,w) \cdot y_{(s-1)_p}(x,w) \tag{9.157}$$

gdzie

$$\delta_{s_q}(x,w) = \frac{\partial \hat{f}(x,w)}{\partial z_{s_q}} \tag{9.158}$$

oraz dla każdej warstwy $s \in \{1,2,\ldots,L\}$

$$y_{s_0}(.,.) \equiv 1 \tag{9.159}$$

Wartości zmiennych δ_{s_q} mogą być wyznaczone z równania propagacji wstecznej (ang. *back-propagation*), czyli następującego schematu rekursywnego (wynikającego bezpośrednio ze struktury sieci) (9.145)–(9.148)

$$\begin{aligned} \delta_{s_q}(x,w) &= \frac{\partial \hat{f}(x,w)}{\partial z_{s_q}} = \sum_{u=1}^{n_{s+1}} \frac{\partial \hat{f}(x,w)}{\partial z_{(s+1)_u}} \cdot \frac{\partial z_{(s+1)_u}(x,w)}{\partial z_{s_q}} = \\ &= \sum_{u=1}^{n_{s+1}} \delta_{(s+1)_u}(x,w) \cdot \frac{\partial z_{(s+1)_u}(x,w)}{\partial y_{s_q}} \cdot \frac{\partial y_{s_q}(x,w)}{\partial z_{s_q}} = \\ &= \sum_{u=1}^{n_{s+1}} \delta_{(s+1)_u}(x,w) \cdot w_{(s+1)_{qu}} \cdot g'\left(z_{s_q}(x,w)\right) \end{aligned} \tag{9.160}$$

z warunkami końcowymi (patrz (9.147)–(9.150)):

$$\delta_{L_q}(x,w) = \frac{\partial \hat{f}(x,w)}{\partial z_{L_q}} = w_{f_q} \cdot g'\left(z_{L_q}(x,w)\right), \quad q = 1,\ldots,m \tag{9.161}$$

lub

$$\delta_{L_q}(x,w) = \frac{\partial \hat{f}(x,w)}{\partial z_{L_q}} = w_{f_q}, \quad q = 1,\ldots,m \tag{9.162}$$

Jako strategia uczenia może być zastosowana tzw. „reguła delta", wykorzystująca algorytm największego spadku, gdzie wektory wag w kolejnych iteracjach wyrażone są za pomocą równania

$$w^{(k+1)} - w^{(k)} = \Delta w^{(k)} = -\alpha^{(k)} \cdot \frac{\partial C}{\partial w}(w^{(k)}) \tag{9.163}$$

czyli

$$w^{(k+1)} = w^{(k)} - \alpha^{(k)} \cdot \frac{\partial C}{\partial w}(w^{(k)}) \tag{9.164}$$

przy czym $\alpha^{(k)}$ oznacza krok. Można też wykorzystać dowolną inną gradientową metodę programowania nieliniowego.

Często bierze się pod uwagę inną definicję błędu aproksymacji niż we wzorze (9.144). Wiąże się ona z interpretacją wejść sieci, czyli pierwszego zestawu argumentów funkcji $\hat{f}(.,.)$, jako zmiennej losowej. Wówczas skończoną sumę po realizacjach zastępuje operator wartości oczekiwanej i zadanie ma postać

$$\min_{w \in W} \{C(w) = \mathop{\mathrm{E}}_{x} \left\{ \|f(x) - \hat{f}(x,w)\|^2 \right\} = \int_x \|f(x) - \hat{f}(x,w)\|^2 \, \mathrm{d}F(x) \tag{9.165}$$

Norma w tym przypadku oznacza normę euklidesową wektora w $\mathbb{R}^m$, $F(x)$ – dystrybuantę zmiennej losowej x, zaś W – pewien zbiór zwarty w $\mathbb{R}^l$, gdzie

$$l = n_1(n+1) + \sum_{s=2}^{L-1} n_s(n_{s-1}+1) + m(n_L+1) \tag{9.166}$$

jest liczbą wszystkich wag synaptycznych sieci neuronowej.

Dla uproszczenia wzorów oznaczmy teraz funkcję podcałkową jako $\varepsilon(x,w)$, czyli

$$\varepsilon(x,w) = \|f(x) - \hat{f}(x,w)\|^2 \tag{9.167}$$

Przy takim podejściu możemy zastosować metodę gradientu stochastycznego [41], [65] i modyfikować wagi synaptyczne po każdej wylosowanej realizacji wejść i wyjść sieci (tzn. argumentów i wartości funkcji). Oczywiście, pod warunkiem, że krok $\alpha^{(k)}$ jest odpowiednio mały. Obowiązuje bowiem następujące twierdzenie (które podajemy za Pierwozwanskim [63]).

Twierdzenie 9.4.
Jeżeli

$$C(w) = \int_x \varepsilon(x, w) \mathrm{d}F(x)$$

jest funkcją wypukłą i różniczkowalną, to warunki:

$$\sum_{k=1}^{\infty} \alpha^{(k)} = \infty \tag{9.168}$$

oraz

$$\sum_{k=1}^{\infty} \left[\alpha^{(k)}\right]^2 < \infty \tag{9.169}$$

wystarczają, by ciąg wag $w^{(k)}$ generowany zgodnie z algorytmem

$$w^{(k+1)} = \Pi_W \left[w^{(k)} - \alpha^{(k)} \cdot \frac{\partial \varepsilon(x^{(k)}, w^{(k)})}{\partial w} \right] \tag{9.170}$$

gdzie Π_W oznacza operator projekcji na zbiór W, był zbieżny z prawdopodobieństwem jeden do rozwiązania zadania (9.165).

Przykładowym ciągiem współczynników kroku $\alpha^{(k)}$ spełniającym warunki (9.168) i (9.169) jest ciąg proponowany przez Darkena i Moody'ego w ich algorytmie „Szukaj a potem zbiegaj" (*Search-Then-Converge Strategy*) [15] wyrażony wzorem:

$$\alpha^{(k)} = \gamma_0 \frac{1}{1 + \frac{k}{k_0}} \quad k_0 \gg 1, \;\; \gamma_0 > 0 \tag{9.171}$$

Ze wzoru (9.171) widać, że dla $k \ll k_0$ występują istotne zmiany wag przy kroku o wielkości około γ_0, które dla $k > k_0$ maleją stopniowo do zera.

Zaletą takiego podejścia jest rozpatrywanie, z uwagi na mechanizm losowy, większej liczby i rozmaitości realizacji (węzłów aproksymacji) niż tylko r (wzór (9.144)) założonych z góry, uwzględnianych w metodzie wsadowej.

Rozproszona implementacja algorytmu uczenia sieci

W sposób naturalny wyróżnić można w tym zadaniu dwa problemy obliczeniowe:

– dostarczanie sieci w przeciągu całego procesu uczenia próbek danych (realizacji, węzłów aproksymacji), czyli wartości funkcji $f(x^{(k)})$,

– iteracyjny proces uczenia, polegający na zmianie wartości nastaw sieci, czyli wag $w^{(k)}$, po uprzednim wyznaczeniu gradientu

$$\frac{\partial \varepsilon(x^{(k)}, w^{(k)})}{\partial w} = \frac{\partial \varepsilon}{\partial \hat{f}} \cdot \frac{\partial \hat{f}(x^{(k)}, w^{(k)})}{\partial w} = \tag{9.172}$$

$$= 2\left(\hat{f}(x^{(k)}, w^{(k)}) - f(x^{(k)})\right) \frac{\partial \hat{f}(x^{(k)}, w^{(k)})}{\partial w}$$

Zauważmy, że do wyznaczenia tego gradientu potrzebujemy jedynie wartości funkcji aproksymowanej $f(x^{(k)})$ w węźle $x^{(k)}$. Na ogół jest ona otrzymywana z symulatora skomplikowanych zjawisk fizycznych, czyli samo jej wyznaczenie zajmuje większość czasu obliczania gradientu $\frac{\partial \varepsilon(x^{(k)}, w^{(k)})}{\partial w}$. A zatem może się tym zająć proces uczenia się.

Cechą charakterystyczną tej metody jest to, że kolejna iteracja w procesie uczenia może być wykonana tylko wtedy, gdy dostarczona jest paczka danych od symulatora. Widać więc, że jest to klasyczny układ procesów typu producent – konsument. Jeżeli dodamy do tego, że konsument jest procesem o przynajmniej rząd wielkości szybszym niż producent, to znaczy zużywa odpowiednio mniej czasu procesora, to koncepcja systemu obliczeniowego narzuci się w sposób naturalny.

Procesy producentów mogą pracować niezależnie od siebie i równolegle na różnych procesorach. Jedynym warunkiem jest zapewnienie w procesie konsumenta (poprzez mechanizm ochrony sekcji krytycznej), by w danym momencie tylko jedna realizacja mogła zmieniać wektor wag. Ze względu na to, że proces uczenia sieci jest wielokrotnie szybszy niż proces produkcji węzłów, liczba producentów determinuje prędkość procesu uczenia.

9.2. Obliczenia częściowo asynchroniczne

9.2.1. Model Bertsekasa–Tsitsiklisa obliczeń częściowo asynchronicznych

Obliczenia takie są czymś pośrednim między obliczeniami synchronicznymi i całkowicie asynchronicznymi.

Przyjmuje się następujące założenia:

1) systematycznej pracy procesorów
   Istnieje taka liczba naturalna B, że dla każdego $i = 1, \ldots, p$ i każdego $t \geqslant 0$ przynajmniej jeden element zbioru $\{t, t+1, \ldots, t+B-1\}$ należy do T^i;

2) aktualności fotografii własnej

$$\tau_i^i(t) = t \quad \forall i, t \in T^i \tag{9.173}$$

3) względnej aktualności fotografii cudzych

$$t - B < \tau_j^i(t) \leqslant t \tag{9.174}$$

Stała B może być interpretowana jako miara asynchronizmu.

Są dwa rodzaje takich obliczeń:

1. Wymagające znajomości (oszacowania) stałej stopnia asynchronizmu B; wówczas:
   - dane przekazywane między komputerami muszą mieć znacznik czasu (numeru iteracji) w skali systemu, występuje zatem coś w rodzaju zegara globalnego,
   - stała B ma wpływ na współczynnik kroku w kolejnych iteracjach algorytmu $x := h(x)$.
2. Nie wymagające znajomości stałej B.
   Jest to bardzo bliskie obliczeniom całkowicie asynchronicznym. Różnica polega na istnieniu gwarancji, że jest pewien czas skończony B (być może nieznany), po którym na pewno nastąpi wymiana informacji. Czas ten jest niezależny od numeru iteracji. W obliczeniach całkowicie asynchronicznych dopuszczony był np. stały wzrost opóźnienia między kolejnymi iteracjami – tutaj nie. Oczywiście, różnica ta ma raczej charakter teoretyczny – w praktyce żadne obliczenia nie trwają nieskończenie długo.

Różnice między obliczeniami z całkowitym i częściowym asynchronizmem dobrze objaśni następujący przykład.

Przykład 9.1. Rozważmy iterację

$$x := Ax \tag{9.175}$$

gdzie

$$A = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} \end{bmatrix} \tag{9.176}$$

Zakładamy, że obydwa procesory przekazują nowe fotografie w czasach t_k, $k = 1, 2, \ldots$, oraz że nie ma opóźnień w transmisji, między tymi czasami zaś wykonywane są iteracje lokalne:

$$x_1(t+1) = \frac{x_1(t)}{2} + \frac{x_2(t_k)}{2} \tag{9.177}$$

$$x_2(t+1) = \frac{x_1(t_k)}{2} + \frac{x_2(t)}{2} \tag{9.178}$$

Iteracja (9.175) jest zbieżna synchronicznie w jednym kroku do

$$\hat{x} = \begin{bmatrix} \bar{x} \\ \bar{x} \end{bmatrix}, \quad \bar{x} = \frac{x_{10} + x_{20}}{2} \tag{9.179}$$

a w wersji asynchronicznej otrzymujemy:

$$x_1(t_{k+1}) = \left(\frac{1}{2}\right)^{t_{k+1}-t_k} x_1(t_k) + \left[1 - \left(\frac{1}{2}\right)^{t_{k+1}-t_k}\right] x_2(t_k) \tag{9.180}$$

$$x_2(t_{k+1}) = \left(\frac{1}{2}\right)^{t_{k+1}-t_k} x_2(t_k) + \left[1 - \left(\frac{1}{2}\right)^{t_{k+1}-t_k}\right] x_1(t_k) \tag{9.181}$$

Stąd

$$\begin{aligned} |x_2(t_{k+1}) - x_1(t_{k+1})| &= \left[1 - 2\left(\frac{1}{2}\right)^{t_{k+1}-t_k}\right] |x_2(t_k) - x_1(t_k)| = \\ &= (1 - \varepsilon_k)\, |x_2(t_k) - x_1(t_k)| \end{aligned} \tag{9.182}$$

gdzie oznaczyliśmy $\varepsilon_k = 2\left(\frac{1}{2}\right)^{t_{k+1}-t_k}$.

Widać, że błąd $|x_2(t_k) - x_1(t_k)|$ maleje. Niemniej jednak zbieżność do $\hat{x}$ jest zagwarantowana tylko wtedy, gdy zachodzi

$$\prod_{k=1}^{\infty} (1 - \varepsilon_k) = 0 \tag{9.183}$$

Jeżeli opóźnienia $t_{k+1} - t_k$ dobierze się tak, że $\varepsilon_k < \frac{1}{k^2}$, to okazuje się, iż

$$\prod_{k=1}^{\infty} (1 - \varepsilon_k) > 0 \tag{9.184}$$

i nie ma zbieżności. Widać więc, że algorytm ten nie jest zbieżny w wersji całkowicie asynchronicznej. Zbieżność częściowo asynchroniczna zachodzi, gdy

$$2 \cdot \left(\frac{1}{2}\right)^{B} > \frac{1}{k^2} \Rightarrow B < 1 - 2\log_{\frac{1}{2}} k \tag{9.185}$$

czyli dla każdego skończonego B. □

Teoria zbieżności dla odwzorowań częściowo asynchronicznych polega na bezpośrednim wykorzystaniu twierdzenia Lapunowa o stabilności dla układów z czasem dyskretnym. Wykorzystuje się w niej definicję wektora „stanu”

$z(t) \in X^B$ algorytmu w czasie t, jako całej informacji, która ma szansę być przechowywana w buforach i wpływać na obliczenia

$$z(t) = (x(t), x(t-1), \ldots, x(t-B+1)) \tag{9.186}$$

Informacji z chwil wcześniejszych nie uwzględnia się, gdyż, zgodnie z założeniami, fotografie wcześniejsze niż z czasu $t-B+1$ są usuwane z systemu.

Wektor $x(t)$ jest zdefiniowany jako

$$x(t) = (x_1(t), x_2(t), \ldots, x_p(t)) \tag{9.187}$$

gdzie $x_i(t)$ jest podwektorem posiadanym przez i-ty procesor w chwili t, otrzymanym przez ten procesor w rezultacie wykonania iteracji (9.9).

Załóżmy, że odwzorowanie h, realizowane przez procesory zgodnie z algorytmem (9.9), jest ciągłe, a $\hat{X} \subset X$ jest niepustym zbiorem jego punktów stałych, czyli zbiorem rozwiązań układu równań

$$x = h(x) \tag{9.188}$$

Niech $\hat{Z}$ będzie zbiorem wszystkich takich wektorów $\hat{z} \in X^B$, że $\hat{z} = (\hat{x}, \hat{x}, \hat{x}, \ldots, \hat{x})$, gdzie $\hat{x}$ występuje B razy.

Twierdzenie 9.5.

Załóżmy, że istnieje ciągła funkcja $d : X^B \to [0, \infty)$ oraz chwila $\hat{t} > 0$, takie, że dla wszystkich ciągów $\{z(t)\}$ otrzymanych w rezultacie zastosowania algorytmu (9.9) spełnione są dwa warunki:

- $\forall t > 0 \;\; d(z(t+1)) \leqslant d(z(t))$,
- $\forall z(0) \notin \hat{Z} \;\; d(z(\hat{t})) < d(z(0))$.

Wówczas każda granica ciągu $\{z(t)\}$ należy do $\hat{Z}$.

Funkcja d jest więc pewną odmianą funkcji Lapunowa. Na przykład, w przestrzeniach $\mathbb{R}^{Bn}$ przyjmuje się często

$$d(z) = \inf_{\hat{z} \in \hat{Z}} \|z - \hat{z}\|_\infty \tag{9.189}$$

Wystarczy wówczas, by h było odwzorowaniem z pewnymi cechami zwężenia, na przykład, nieekspansywne, czyli

$$\|h(x) - \hat{x}\|_\infty \leqslant \|x - \hat{x}\|_\infty \tag{9.190}$$

Odwzorowanie to nie musi być zwężające – w takim przypadku istniałby tylko jeden punkt stały i warto by było zastosować algorytm całkowicie asynchroniczny.

Poniżej przedstawimy kilka przykładów obliczeń zbieżnych w wersji z częściowym asynchronizmem.

9.2.2. Przykłady zadań, które mogą być rozwiązane za pomocą obliczeń częściowo asynchronicznych

9.2.2.1. Rozwiązywanie układów równań liniowych

Rozważmy układ równań liniowych:

$$Ax = b \tag{9.191}$$

$$x := x - \gamma(Ax - b), \quad \gamma \in (0, 1) \tag{9.192}$$

Aby zagwarantować zbieżność dla każdego skończonego B, wystarczy, żeby A była macierzą ze słabą dominacją diagonali

$$\sum_{j \neq i} |a_{ij}| \leqslant |a_{ii}| \quad \forall i \tag{9.193}$$

9.2.2.2. Sieci neuronowe z pamięcią (Hopfielda)

Równanie i-tego wyjścia

$$x_i = \sigma\left(\sum_{j=1}^{n} w_{ij} x_j + u_i\right) \tag{9.194}$$

gdzie $w_{ij}, u_i, i, j = 1, \ldots, n$ to dane liczby, przy czym $\sum\limits_{j=1}^{n} |w_{ij}| \leqslant 1$. Dla dowolnej funkcji sigmoidalnej ciągłej, rosnącej monotonicznie od -1 do 1, np.

$$\sigma(y) = \frac{2}{1 + \mathrm{e}^{-2y}} - 1 \tag{9.195}$$

oraz dla dowolnego czasu opóźnienia $B < \infty$ odwzorowanie

$$x_i = (1 - \gamma) x_i + \gamma\sigma\left(\sum_{j=1}^{n} w_{ij} x_j + u_i\right), \quad \gamma \in (0, 1) \tag{9.196}$$

jest zbieżne do punktu stałego.

9.2.2.3. Algorytmy porozumieniowe

W algorytmach porozumieniowych (ang. *agreement algorithms*) każdy z procesorów iteruje sumę ważoną współrzędnych wektora x, przy czym ma własny zestaw wag

$$x_i = \sum_{j=1}^{n} a_{ij} x_j \tag{9.197}$$

gdzie

$$\sum_{j=1}^{n} a_{ij} = 1 \quad \forall i \tag{9.198}$$

Szukamy rozwiązania, w którym następuje porozumienie, tzn. takiego punktu $\hat{x}$, że

$$\hat{x}_i = \sum_{j=1}^{n} a_{ij}\hat{x}_j \tag{9.199}$$

Do rozwiązania dla każdego $B < \infty$ prowadzi iteracja

$$x_i(t+1) = \begin{cases} \sum\limits_{j=1}^{n} a_{ij} x_j\left(\tau_j^i(t)\right) & t \in T^i \\ x_i(t) & t \notin T^i \end{cases} \tag{9.200}$$

Algorytmy te można wykorzystywać, na przykład, do liczenia estymaty wartości oczekiwanej procesu, z którego niezależne pomiary współrzędnych są obsługiwane przez poszczególne procesory lub granicznego prawdopodobieństwa dla stacjonarnych łańcuchów Markowa.

9.2.2.4. Równoważenie obciążenia w sieciach komputerowych lub w komputerze równoległym

Niech:
- L – łączne obciążenie (praca do wykonania),
- p – liczba procesorów,
- $x_i(t) \geqslant 0$ – obciążenie i-tego procesora w chwili t,
- $N(i)$ – zbiór sąsiadów i-tego procesora,
- $s_{ij}(\mathrm{t})$ – transfer części pracy od i do j w chwili t.

Przyjmujemy

$$\sum_{i=1}^{p} x_i(0) = L \tag{9.201}$$

ale

$$x_i(0) \neq \frac{L}{p}, \quad i = 1, \ldots, p \tag{9.202}$$

czyli że procesory nie są obciążone równomiernie. Zadanie polega na przekazaniu części pracy od bardziej do mniej obciążonych procesorów.

Algorytm działa w ten sposób, że procesory porównują na bieżąco obciążenie swoje i swoich sąsiadów. Jeśli procesor oceni, że jego obciążenie jest mniejsze niż sąsiadów, to nic nie robi, jeśli jednak stwierdzi, że jego obciążenie jest większe, to przekazuje część swojej pracy najmniej obciążonemu sąsiadowi. Zakłada się, że transfer jest nie mniejszy niż część $\alpha \in (0, 1)$ nadwyżki, przy czym α jest ustalona dla wszystkich procesorów oraz całego

procesu obliczeniowego. Zatem przyjmując, że dany procesor ma indeks i, a l jest indeksem najmniej obciążonego sąsiada

$$l = \arg \min_{k \in N(i)} x_k(\tau_k^i(t)) \tag{9.203}$$

napiszemy

$$s_{il}(t) \begin{cases} = 0 & x_i(t) \leqslant x_l\left(\tau_l^i(t)\right) \\ \geqslant \alpha\left(x_i(t) - x_l\left(\tau_l^i(t)\right)\right) & x_i(t) > x_l\left(\tau_l^i(t)\right) \end{cases} \tag{9.204}$$

Można również przekazywać pozostałą część obciążenia innym sąsiadom, pod warunkiem, że są oni mniej obciążeni. Wszystkie transfery od i-tego procesora do sąsiadów $j \in N(i)$ muszą jednak spełniać warunki

$$x_i(t) - \sum_{k \in N(i)} s_{ik}(t) \geqslant x_j\left(\tau_j^i(t)\right) + s_{ij}(t), \quad \forall j \in N(i) \tag{9.205}$$

Służą one do tego, by uniknąć dużych transferów oraz tworzenia niezrównoważenia przeciwnego znaku, co mogłoby prowadzić do oscylacji.

Weźmy teraz pod uwagę to, że mamy do czynienia z obliczeniami z częściowym asynchronizmem. Transfer obciążenia nie jest więc natychmiastowy, wiemy jedynie, że obciążenie $s_{ij}(t)$ wysłane z i do j jest odebrane przez j przed czasem $t + B$. Oznaczmy w związku z tym:

$r_{ij}(t)$ – ilość obciążenia (pracy) wysłanego od i do j otrzymanego przez j w chwili t,

$v_{ij}(t)$ – ilość obciążenia (pracy) wysłanego od i do j, ale nie otrzymanego przez j przed t.

Obciążenie procesorów będzie się zmieniało zgodnie z równaniem (jest to jednocześnie algorytm równoważenia obciążenia)

$$x_i(t+1) = x_i(t) - \sum_{j \in N(i)} s_{ij}(t) + \sum_{j \in N(i)} r_{ji}(t), \qquad t \in T^i \tag{9.206}$$

Biorąc pod uwagę przyjęte założenia, będziemy mieli

$$v_{ij}(t) = \sum_{\tau=0}^{t-1} \left(s_{ij}(\tau) - r_{ij}(\tau)\right) \tag{9.207}$$

oraz

$$\sum_{i=1}^{p} \left(x_i(t) + \sum_{j \in N(i)} v_{ij}(t) \right) = L \tag{9.208}$$

Nietrudno udowodnić następujące twierdzenie.

Twierdzenie 9.6.

$$\lim_{t\to\infty} x_i(t) = \frac{L}{p} \qquad \forall i \tag{9.209}$$

9.2.2.5. Optymalizacja gradientowa bez ograniczeń

Rozwiązujemy zadanie

$$\min_{x\in\mathbb{R}^n} f(x) \tag{9.210}$$

Okazuje się, że większość gradientowych algorytmów optymalizacji (m.in. gradientów sprzężonych, zmiennej metryki) ma zapewnioną zbieżność przy implementacji równoległej z częściowym asynchronizmem, jeśli krok w kierunku poprawy nie jest zbyt duży. W tej klasie algorytmów ograniczenie kroku zależy zarówno od normy hesjanu funkcji f, jak i od wielkości opóźnienia B.

Dokładniej, przyjmijmy następujące założenia odnośnie funkcji f oraz algorytmu optymalizacji:

$$x := h(x) = x + \gamma s \tag{9.211}$$

czyli i-ty procesor realizuje algorytm

$$x_i(t+1) = x_i(t) + \gamma s_i(t) \tag{9.212}$$

gdzie $s \in \mathbb{R}^n$, jest kierunkiem poprawy.

Założenia:

1. Funkcja f jest nieujemna, czyli

$$f(x) \geqslant 0 \qquad \forall x \in \mathbb{R}^n \tag{9.213}$$

2. Funkcja f jest różniczkowalna w sposób ciągły, a jej gradient jest ciągły w sensie Lipschitza, czyli istnieje taka stała K_1, że

$$\|\nabla f(x) - \nabla f(y)\| \leqslant K_1 \|x - y\| \qquad \forall x, y \in \mathbb{R}^n \tag{9.214}$$

   Jeśli funkcja należy do klasy $C_2(\mathbb{R}^n)$, to warunek ten jest równoważny warunkowi ograniczenia hesjanu (8.21) i za stałą K_1 można przyjąć wartość liczby K.

3. Dla każdego $i = 1, \ldots, p$ oraz dla każdego $t \in T^i$ spełniony jest warunek poprawy lokalnej

$$s_i(t)' \frac{\partial f}{\partial x_i}(x^i(t)) \leqslant -\frac{\|s_i(t)\|^2}{K_3} \tag{9.215}$$

4. Wektor kierunku poprawy nie jest zbyt mały, to znaczy istnieje taka stała dodatnia K_2, że:

$$\|s_i(t)\| \geqslant K_2 \left\| \frac{\partial f}{\partial x_i}(x^i(t)) \right\| \quad \forall i \ \ \forall t \in T^i \tag{9.216}$$

Można udowodnić, że jeśli krok γ w algorytmie (9.211) spełnia warunek

$$\gamma < \gamma_0(B) = \frac{1}{K_3 K_1 (1 + B + nB)} \tag{9.217}$$

to algorytm ten w realizacji częściowo asynchronicznej jest zbieżny do punktu $\hat{x}$, w którym

$$\nabla f(\hat{x}) = 0 \tag{9.218}$$

9.2.2.6. Optymalizacja gradientowa z ograniczeniami

Rozważamy zadanie

$$\min_{x \in X} f(x) \tag{9.219}$$

przyjmując, że zbiór dopuszczalny X ma postać

$$X = \prod_{i=1}^{p} X_i \tag{9.220}$$

gdzie $X_i \subset \mathbb{R}^{n_i}$. W tym przypadku poszczególne procesory będą realizowały algorytm:

$$x_i(t+1) := F_i(x^i(t)) = \left[x_i(t) - \gamma \frac{\partial f}{\partial x_i} \left(x^i(t) \right) \right]^+ \tag{9.221}$$

w którym „+” oznacza rzutowanie na ograniczenia (ortogonalne, w normie euklidesowej, czyli $[x]^+ = \arg\min_{z \in X} \|z - x\|_2$).

Można udowodnić [6], że jeśli funkcja f jest wypukła i nieujemna, jej gradient zaś spełnia warunek Lipschitza (9.214), to algorytm (9.221) w realizacji częściowo asynchronicznej jest zbieżny dla

$$\gamma < \gamma_0(B) = \frac{1}{K_1 (1 + B + nB)} \tag{9.222}$$

gdzie B jest stopniem asynchronizmu mierzonym maksymalną liczbą iteracji między aktualizacjami fotografii, a K_1 – stałą Lipschitza gradientu funkcji f.

9.2.2.7. Algorytmy z gradientem stochastycznym

Algorytmy takie mają najczęściej zastosowanie przy identyfikacji obiektów. Są one postaci

$$x(t+1) = x(t) - \gamma(t) \cdot (\nabla F(x(t)) + w(t)) \tag{9.223}$$

gdzie $w(t)$ jest zmienną losową niezależną od $x(t)$. Wyrażenie w nawiasie może być interpretowane jako szum obserwacyjny gradientu. Algorytm powyższy jest zbieżny dla współczynników kroku spełniających warunki:

$$\sum_{t=1}^{\infty} \gamma(t) = \infty \tag{9.224}$$

$$\sum_{t=1}^{\infty} \gamma^2(t) < \infty \tag{9.225}$$

Na przykład, dla

$$\gamma(t) = \frac{1}{t} \tag{9.226}$$

9.3. Metody optymalizacji globalnej

Omówimy kilka różnych technik, które są stosowane w zadaniach optymalizacji statycznej, gdy funkcja celu lub zbiór rozwiązań dopuszczalnych są niewypukłe. Znaczna część rozważanych metod powstała w wyniku kombinacji aparatu matematyki oraz heurystyki. W niektórych metodach poszukiwanie rozwiązania jest realizowane w wyniku przekształceń deterministycznych, w innych – istotnym elementem są operacje losowe. Zbieżność do rozwiązania w skończonej liczbie kroków jest gwarantowana tylko dla wąskiej klasy metod, a dla większej grupy możemy przedstawić dowody zbieżności w sensie probabilistycznym, tj. wykazać, że z założonym prawdopodobieństwem osiągniemy poszukiwane rozwiązanie.

Jedną z powszechnie dostrzeganych cech algorytmów optymalizacji globalnej są duże wymagania związane z nakładem obliczeń i zasobami komputera. Z drugiej strony znaczną ich część charakteryzuje naturalna równoległość i to z dopuszczeniem implementacji asynchronicznej. Realizacja równoległa może istotnie wpłynąć na niezawodność i efektywność algorytmów.

W tym rozdziale ograniczymy się do przedstawienia krótkiego przeglądu wybranych technik globalnych, zwracając szczególną uwagę na sposoby ich

zrównoleglenia. W przypadku tych metod można zaproponować różne warianty zrównoleglania obliczeń, podejścia częściowo asynchroniczne lub całkowicie asynchroniczne. Osoby zainteresowane algorytmami optymalizacji globalnej zachęcamy do zapoznania się z literaturą [1, 32, 38, 39, 54, 74].

9.3.1. Metody deterministyczne

Wiele metod deterministycznych to metody siatki (*grid methods*). Korzystając z kryterium Hausdorffa poszukuje się minimum globalnego na siatce punktów. Najprostsza odmiana tej metody tworzy siatkę równomierną. Ulepszeniem pierwotnej wersji są techniki podziałów nierównomiernych. Metody siatki gwarantują zbieżność do rozwiązania w skończonej liczbie iteracji, przy założeniu że funkcja spełnia warunek Lipschitza

$$\exists L > 0 \qquad \forall x_1, x_2 \in X \quad |f(x_1) - f(x_2)| \leqslant L||x_1 - x_2|| \tag{9.227}$$

Korzystając z tego warunku, można wyznaczyć minimalną odległość d punktów tworzących siatkę równomierną w przestrzeni $\mathbb{R}^n$, która umożliwia wyznaczenie minimum globalnego z dokładnością ε, $d = \varepsilon/(L\sqrt{n})$.

Metody siatki równomiernej są nieefektywne, gdyż poszukiwanie rozwiązania jest prowadzone w sposób pasywny i nie bierze się pod uwagę informacji zdobytych w czasie działania algorytmu. Znacznie efektywniejsze są techniki siatek nierównomiernych. Przykładem są metody podziału i ograniczeń Gourdina i Meewella–Mayne'a [50, 51].

Przyjrzyjmy się metodzie Meewella–Mayne'a. Wykorzystuje się tu aproksymację funkcji celu zaproponowaną przez Pijavskiego i Shuberta [74]. Nowe punkty siatki wyznaczane są sekwencyjnie w wyniku rozwiązania zadań minimaksowych formułowanych we wszystkich komórkach siatki $C^l, l = 1, \ldots, K$. Funkcja celu $f(x)$ jest wewnątrz każdej komórki aproksymowana za pomocą funkcji kawałkami liniowej $h^l(x, y^l)$, gdzie y^l oznacza zbiór wierzchołków l-tej komórki siatki (przyjmuje się $h^l(x, y^l) \leqslant f(x)$, $x \in C^l$, $X = \bigcup_{l=1,\ldots,K} C^l$). W każdej iteracji dzielona jest komórka, w której minimum funkcji h^l jest najmniejsze. W wyniku podziału powstaje co najwyżej 2^n nowych komórek. Zagęszczenie podziału występuje tam, gdzie funkcja celu optymalizacji przyjmuje najmniejszą wartość. Przyspiesza to działanie metody w stosunku do podejścia z siatką równomierną.

Wyznaczenie hiperpłaszczyzn podpierających funkcję celu wymaga wyznaczenia wartości funkcji celu w węzłach siatki, a przede wszystkim oszacowania stałej Lipschitza. Jest to poważna wada metody. W przypadku gdy funkcja celu jest różniczkowalna, do wyznaczenia stałej L wykorzystuje

się maksymalne pochodne cząstkowe funkcji; jeżeli nie dysponujemy analityczną postacią funkcji (np. wynik symulacji itp.), jedynym sposobem jest wykonanie eksperymentu numerycznego, polegającego na losowym wybieraniu par punktów z dziedziny oraz wyznaczeniu L z zależności (9.227).

Wersja sekwencyjna algorytmu Meewella–Mayne'a jest następująca, przy założeniu, że wykonujemy k-tą iterację algorytmu Meewella–Mayne'a.

KROK 1. W każdym wierzchołku l-tej komórki wyznaczyć funkcję podpierającą

$$h^l(x, y^l) = \max_{m=1,\ldots,2^n} \left[f(y_m^l) + (x - y_m^l)^T L \sum_{j=1}^{n} u_j(m) e_j \right] \tag{9.228}$$

gdzie: m – numer wierzchołka $m = 1, \ldots, 2^n$, e_j – wektor n-elementowy z j-tą współrzędną równą jeden, $u_j(m) = 2c_j(m) - 1$, $c(m)$ – reprezentacja binarna liczby $m-1$, gdzie $m-1 = \sum_{i=0}^{n-1} 2^i c^{n-i}(m)$ i $c^i(m) \in \{0, 1\}$.

KROK 2. W każdej komórce $l = 1, \ldots, K$ rozwiązać zadanie programowania liniowego

$$\hat{x}^l = \arg \min_{x \in C^l} h^l(x, y^l) \tag{9.229}$$

stosując np. metodę sympleksu liniowego.

KROK 3. Wyznaczyć nowy punkt podziału

$$x^* = \arg \min_{l=1,\ldots,K} h^l(\hat{x}^l, y^l) \tag{9.230}$$

Jeżeli punkt x^* jest węzłem siatki – zakończyć działanie, jeżeli nie – podzielić komórkę l^* w punkcie x^*.

Jedyny sensowny sposób zrównoleglenia algorytmu polega na dekompozycji dziedziny na podobszary i równoległym uruchomieniu p instancji programów na tych podobszarach. Działanie procesów nie wymaga synchronizacji. Procesy mogą przekazywać sobie wzajemnie najlepsze, znalezione dotychczas rozwiązania, tzn. najmniejszą znalezioną dotychczas wartość funkcji celu $\hat{f}_p$. Pozwala to na usuwanie na bieżąco z obszaru poszukiwań poszczególnych procesów tych komórek, dla których $h_p^l(\hat{x}^l, y^l) \geqslant \min_{p=1,\ldots,P} \hat{f}_p$. Ponadto z przeprowadzonych eksperymentów numerycznych wynika, że można uzyskać istotne przyspieszenie włączając w pewnych etapach działania wybraną, efektywną metodę poszukiwania ekstremum lokalnego (np. Newtona, BFGS, itd.). Obliczenia wykonywane tymi metodami mogą być również realizowane przez różne procesy.

9.3.2. Metody niedeterministyczne

9.3.2.1. Metody poszukiwania losowego

Metody niedeterministyczne można podzielić na niezależne i błądzące. Metody niezależne prowadzą poszukiwania niezależnie w wielu punktach dziedziny. Punktem wyjścia do tych technik była metoda Monte Carlo, zwana także metodą prostych poszukiwań losowych. Polega ona na losowaniu punktów ze zbioru dziedziny funkcji przy zastosowaniu rozkładu równomiernego i przyjmowaniu aktualnie najlepszej wartości funkcji jako oszacowania optimum. Metoda może być uruchamiana w systemie złożonym z wielu procesorów, dla podobszarów dziedziny poszukiwań. Podczas obliczeń między procesorami przekazywane są najlepsze dotychczasowe rozwiązania. Dodatkowo, z wylosowanych punktów uruchamiane są klasyczne metody optymalizacji lokalnej (otrzymujemy wówczas metodę wielostartową). Zamiast generatorów losowych można tu również zastosować generatory quasi-losowe, tzw. $L_P\tau$ ciągi [66].

Do metod niezależnych są też zaliczane metody sterowanego przeszukiwania losowego CRS [42, 67]. Różne warianty tych metod są oznaczane kolejnymi numerami. W algorytmach CRS2 i CRS3 stosuje się przekształcenia zaproponowane w metodzie sympleksu nieliniowego Neldera–Meada [58]. Losowany jest zbiór początkowy P punktów o liczności zależnej od wymiaru zadania n (liczność $NP(n+1)$, NP – stała, np. 10), w których wyznacza się wartość funkcji celu. Zbiór P w miarę upływu czasu podlega przekształceniom.

Załóżmy, że wykonujemy k-tą iterację algorytmu CRS2.

KROK 1. Wyznacz punkty: najlepszy $x^l = \arg \min_{i=1,\ldots,P} f(x^i)$ i najgorszy $x^h = \arg \max_{i=1,\ldots,P} f(x^i)$ w zbiorze P.

KROK 2. Wylosuj ze zbioru P n punktów i utwórz $(n+1)$-wymiarowy sympleks $\{x^l, x^1, x^2, \ldots, x^n\}$, w którym pierwszym wierzchołkiem jest najlepszy punkt ze zbioru P. Wyznacz środek sympleksu $\overline{x} = \left(x_l + \right.$ $\left. + \sum_{i=1}^{n-1} x^i \right)/n$.

KROK 3. Odbij punkt x^n względem środka sympleksu $\overline{x}$

$$x_r = 2\overline{x} - x^n$$

KROK 4. Sprawdź, czy punkt x_r należy do zbioru dopuszczalnego X. Jeżeli punkt z odbicia jest dopuszczalny i spełnia warunek $f(x_r) < f(x^h)$, to w miejsce punktu x^h w zbiorze P wstaw wynik odbicia, tj. punkt x_r,

i wróć do pierwszego kroku. W przeciwnym razie wylosuj nowy sympleks i powtórz kolejne kroki.

W przypadku metody CRS3 dodatkowo wymaga się sortowania zbioru P. Jeżeli wynik odbicia trafi do zbioru składającego się z 10% najlepszych punktów P, wówczas uruchamiana jest metoda optymalizacji lokalnej Neldera–Meada. Po jej zakończeniu następuje powrót do algorytmu CRS2.

Najefektywniejszym sposobem zrównoleglenia algorytmów CRS jest uruchomienie niezależnych instancji programu na różnych procesorach. Wykorzystuje się tu strukturę gwiazdy. Poszczególne procesy po wykonaniu obliczeń dla swoich zbiorów punktów P^p przekazują wyniki do wybranego procesu, który wybiera rozwiązanie najlepsze.

Metody błądzące symulują losowo zakłócony ruch obiektu dynamicznego. Przykładem takiej metody może być algorytm symulowanego wyżarzania (ang. *simulated annealing*) [23, 39, 44, 66] wykorzystujący analogię do powolnego stygnięcia i krystalizacji substancji. Nie będziemy jej tutaj omawiać.

9.3.2.2. Algorytmy ewolucyjne

Algorytmy ewolucyjne (algorytmy genetyczne, strategie ewolucyjne) są odpornymi, prostymi metodami, które mogą być stosowane, między innymi, w zadaniach poszukiwania ekstremum globalnego [1, 32, 54]. Nie nakładają one żadnych dodatkowych wymagań odnośnie optymalizowanej funkcji, której wartości mogą być generowane przez *czarną skrzynkę*; wystarczy założenie, że rozwiązanie istnieje. Ich zasadniczą wadą jest (podobnie jak w przypadku większości innych metod niedeterministycznych) brak gwarancji zbieżności w skończonej liczbie iteracji oraz powolna zbieżność. Pomysł technik ewolucyjnych pochodzi z symulacji procesu ewolucji żywych organizmów. Stan algorytmu opisuje zbiór odpowiednio zakodowanych łańcuchów (binarnie bądź za pomocą liczb rzeczywistych), będących analogiem kodów genetycznych (chromosomów). Do każdego łańcucha jest przypisana miara przystosowania (w zadaniach optymalizacji funkcja celu). Multizbiór łańcuchów (populacja) jest losowo wybierany z przestrzeni poszukiwań i poprzez mechanizmy wyboru, mutacji i (czasami) rekombinacji ewoluuje w kierunku minimum globalnego funkcji celu. W ten sposób uzyskujemy odwzorowanie zwężające, a więc algorytm optymalizacji.

Elementarny algorytm genetyczny [32] jest skonstruowany z następujących operacji:

– Inicjalizacja – polega na wylosowaniu początkowej populacji P. Zmienne zadania optymalizacji muszą być zakodowane w postaci skończonego ciągu znaków z pewnego skończonego alfabetu, np. w postaci łańcucha binarnego: 01101 – osobnik 1, 11000 – osobnik 2.

- Selekcja – kopiowanie z poprzedniej populacji do nowej może odbywać się według różnych reguł. Warunkiem jest jednak, aby prawdopodobieństwo wyboru osobników o większej mierze przystosowania było większe (przykład – „koło ruletki").
- Krzyżowanie – zazwyczaj przebiega w dwóch etapach. Najpierw kojarzymy losowo w pary ciągi z populacji rodziców. Następnie, w sposób losowy, z jednakowym prawdopodobieństwem, wybieramy miejsce przecięcia k spośród $l-1$ początkowych pozycji w ciągu kodowym (np. dla osobników 01101 i 11100 oraz $k=3$ otrzymujemy osobniki 01100 i 11101).
- Mutacja – polega na losowej zmianie jednego bitu ciągu z populacji (odwrócenie bitu).

W strategiach ewolucyjnych stosuje się kodowanie za pomocą liczb rzeczywistych. Losowe modyfikacje osobników są realizowane zgodnie z rozkładem normalnym o wartości oczekiwanej równej zeru [1, 54].

Jak już wspomnieliśmy, algorytmy ewolucyjne można w sposób naturalny zrównoleglić. Zwiększa to efektywność algorytmów. Zrównoleglenie może być realizowane w sposób synchroniczny i asynchroniczny.

Najprostszym podejściem jest wariant synchroniczny, w którym operuje się na jednej wspólnej populacji (algorytm bez podziału dziedziny). Procesy podrzędne równolegle wyznaczają wartości funkcji przystosowania różnych osobników wchodzących w skład populacji. Proces nadrzędny koordynuje działania procesów podrzędnych. Odpowiada on za selekcję oraz wykonuje operacje genetyczne. Sama realizacja algorytmu jest ściśle związana z dostępną architekturą. Takie podejście ma istotne wady: komunikacja jest częsta, a więc powinna być realizowana szybko, przyspieszenie można osiągnąć tylko wtedy, gdy nakład na komunikację jest mały w porównaniu z nakładem na obliczanie wartości funkcji celu, niezawodność algorytmu zależy bezpośrednio od niezawodności procesu nadrzędnego.

Bardziej efektywny wydaje się być wariant asynchroniczny algorytmu (z podziałem dziedziny), gdzie wiele procesów wykonuje autonomicznie operacje genetyczne dla rozproszonej populacji i komunikuje się między sobą w celu wymiany informacji (np. części osobników). Takie algorytmy nazywamy podpopulacyjnymi. Ewoluujące niezależnie od siebie populacje mogą się składać z osobników tego samego lub różnych typów oraz mogą różnić się także funkcją przystosowania. Zazwyczaj między funkcjami występują wówczas pewne zależności.

Przykładem algorytmu podpopulacyjnego jest algorytm wyspowy [1]. Wersja równoległa takiego algorytmu jest następująca. Poszczególne procesy realizują obliczenia, których celem jest wyznaczenie rozwiązania zadania optymalizacji. Stosowany jest algorytm ewolucyjny, przy czym każdy

z procesów operuje na innej populacji osobników. Procesy komunikują się między sobą w celu wymiany pewnych osobników. Osobniki mogą zostać wprowadzone do populacji danego procesu oraz przesłane do innych procesów. Stosowane są dwa modele wymiany osobników:

– osobnik przesyłany do innych procesów jest automatycznie usuwany z populacji macierzystego procesu (tzw. *emigracja*),
– osobnik pozostaje w populacji bazowej, do innych procesów przesyłana jest jego kopia (tzw. *imigracja*).

Proponuje się różne warianty działania algorytmu, np. osobniki są rozsyłane do wszystkich pozostałych procesów lub tylko do procesów wybranych (sąsiadów). Oczywiście, realizacja algorytmu jest zdeterminowana przez dostępną architekturę, niemniej jednak takie podejście pozwala na istotne zredukowanie nakładu na komunikację.

9.4. Warunki stopu

Warunki stopu w przypadku algorytmów asynchronicznych są dużo bardziej skomplikowane niż w przypadku algorytmów synchronicznych.

Można przyjąć, że obliczenia powinny być zakończone, gdy:

– wszystkie fotografie są aktualne, tzn.

$$x_j\left(\tau_j^i(t)\right) \equiv x_j(t) \quad \forall j = 1, \ldots, p \tag{9.231}$$

– nowa iteracja procesora j nie zmienia x_j, $\quad \forall j = 1, \ldots, p$.

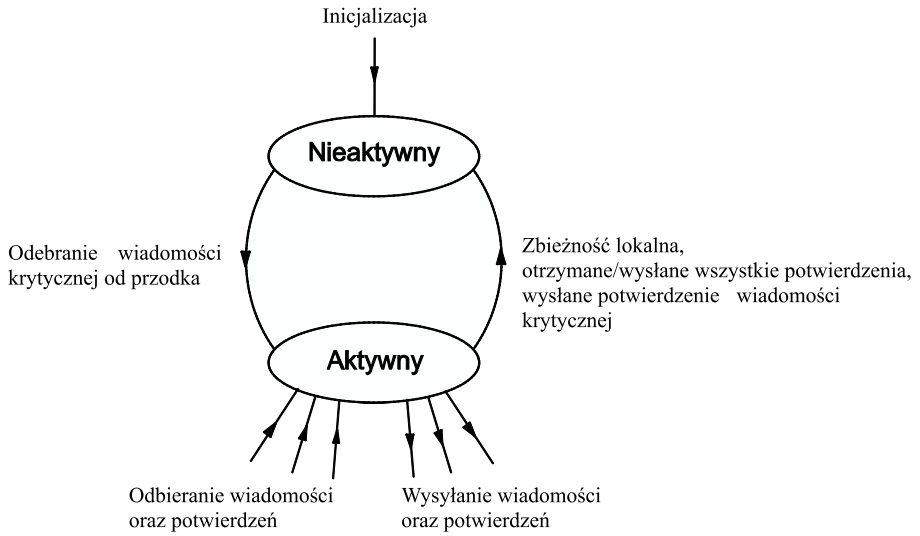
Są różne sposoby sprawdzania tych warunków. Przedstawimy tutaj dwa z nich, zaprojektowane dla środowiska sieciowego.

1. Z potwierdzaniem wszystkich wiadomości

Zakłada się dodatkowo, że komputery, poza pakietami z fotografiami swoich podwektorów, wysyłają pakiety potwierdzeń (ang. *acknowledgements*). Komputer (procesor) może być w dwóch stanach: aktywnym i nieaktywnym. Do stanu aktywnego procesor przechodzi, gdy dostanie pierwszy komunikat od dowolnego innego procesora (przodka). Taki komunikat będziemy nazywać wiadomością krytyczną, która nie wymaga natychmiastowego potwierdzenia. W tym stanie procesor wykonuje obliczenia i komunikuje się z innymi. Potomek może wrócić do stanu nieaktywnego po wysłaniu do przodka potwierdzenia wiadomości krytycznej (rys. 9.2). Może to nastąpić, gdy spełnione są następujące warunki:

a) komputer (procesor) wykorzystał w iteracjach lokalnych wszystkie komunikaty, które otrzymał,

b) nowa iteracja lokalna nie zmienia podwektora lokalnego,
c) komputer (procesor) wysłał potwierdzenia otrzymania wszystkich komunikatów, które dostał (poza wiadomością krytyczną),
d) komputer (procesor) otrzymał potwierdzenia otrzymania wszystkich komunikatów, jakie wysłał.



Rys. 9.2. Graf stanów procesora związanych z warunkiem stopu

2. Z potwierdzaniem tylko zbieżności lokalnej

Wadą metody 1. jest duża liczba potwierdzeń, ponieważ wymagane jest potwierdzanie otrzymania każdego komunikatu z danymi bieżącymi. Ponadto, algorytm zablokuje się, gdy jakiś komunikat przepadnie, co jednak w sieci czasami może się zdarzyć.

Z tego punktu widzenia lepszy jest algorytm detekcji zbieżności globalnej, gdzie procesory przesyłają komunikaty organizacyjne tylko wtedy, gdy któryś z nich stwierdzi zbieżność lokalną. Wówczas przepytuje on inne, czy ich obliczenia również zbiegły (zakładamy, że wcześniej nie dostał takiego zapytania od innego procesora; jeśli tak, to odpowiada i przechodzi w stan nieaktywny). Jeżeli wszystkie procesory potwierdzą swoją zbieżność, to będziemy mieli do czynienia ze zbieżnością globalną.

Rozdział 10

Symulacja rozproszona

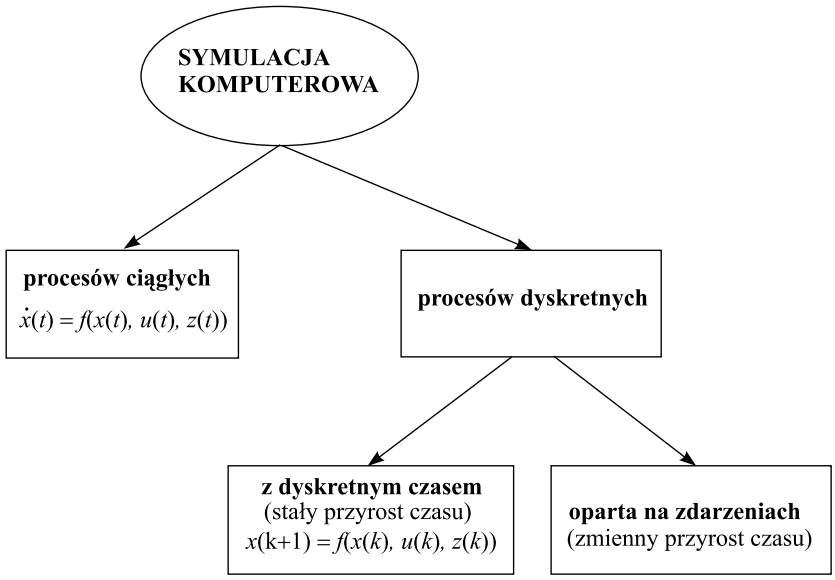
Ewa Niewiadomska-Szynkiewicz

10.1. Wprowadzenie – symulacja komputerowa

Przy prowadzeniu badań nad systemami rzeczywistymi oraz projektowaniu układów sterowania tymi systemami konieczne jest wykonanie wielu eksperymentów badawczych w celu dostarczenia wiedzy o procesach zachodzących w systemie oraz sposobach wpływania na ich przebieg. W większości przypadków niemożliwe jest wykonanie badań na rzeczywistym systemie. W tej sytuacji stosuje się symulację komputerową, a więc wykonuje eksperyment z modelem. W ostatnich latach nastąpił gwałtowny rozwój w dziedzinie technik i narzędzi do symulacji [43, 81]. Symulacja stanowi jedno z podstawowych narzędzi wykorzystywanych we wszystkich obszarach nauki i techniki. Celem symulacji jest uzyskanie odpowiedzi na pytanie, jak system będzie się zachowywał w określonej sytuacji, tzn. jakie będą jego sygnały wyjściowe dla określonych sygnałów wejściowych. Wykonując eksperymenty symulacyjne, możemy:

- zrozumieć działanie systemu,
- uzyskać estymaty wartości wskaźników opisujących jakość działania systemu,
- ocenić poprawność projektu układu sterowania, wskazać kierunki jego poprawy.

Eksperymenty symulacyjne poprzedza budowa modelu symulacyjnego systemu. W związku z tym, że budując model staramy się, by był on odpowiednikiem fizycznego systemu (z zadaną i możliwą do osiągnięcia dokładnością), wyniki badań eksperymentalnych wykonanych z modelem można z pewnym przybliżeniem przenieść na rzeczywisty system. O tym, jak daleko idące wnioski można wyciągać co do zachowania się systemu na podstawie badań wykonanych z modelem, decyduje poziom zgodności modelu z systemem rzeczywistym. Nie jest natomiast istotny sposób formułowania modelu i wykorzystane do jego budowy narzędzia techniczne. W przypadku gdy mówimy o symulacji komputerowej, model systemu jest oczywiście kodowany w postaci programów komputerowych, a symulacja jest wykonywana w komputerze. Możemy więc podać bardzo ogólną definicję symulacji komputerowej [13]: *Symulacja komputerowa to eksperyment wykonywany w komputerze.*



Rys. 10.1. Rodzaje metod symulacyjnych

Istnieją dwie klasy metod symulacyjnych (rys. 10.1):

Symulacja procesów ciągłych

Eksperymenty są wykonywane z modelami ciągłymi, w których w skończonym okresie czasu zmienne stanu zmieniają się nieskończenie często (czas jest ciągły i wartości zmiennych wejściowych, wyjściowych i stanu są określone dla każdej chwili czasu). Są one zazwyczaj opisywane za pomocą równań różniczkowych zwyczajnych $\dot{x}(t) = f(x(t), u(t), z(t))$, gdzie $x(t)$ ozna-

cza wektor zmiennych stanu, $u(t)$ – wektor zmiennych sterujących, a $z(t)$ – wektor wejść swobodnych.

Symulacja procesów dyskretnych

Zakłada się, że zmiany stanu następują w dyskretnych chwilach czasu, a więc w skończonym okresie czasu i że liczba wartości przyjmowanych przez zmienne stanu jest ograniczona. W zależności od sposobu zmian czasu możemy mówić o dwóch rodzajach symulacji dyskretnej:

- symulacja z czasem dyskretnym (ang. *discrete-time simulation*) – symulacja ze stałym przyrostem czasu, charakteryzująca się stałym krokiem czasowym. Do opisu modelu wykorzystywane są równania różnicowe postaci $x(k+1) = f(x(k), u(k), z(k))$, gdzie k oznacza kolejne chwile dyskretne.
- symulacja oparta na zdarzeniach (ang. *discrete event simulation*) – symulacja ze zmiennym przyrostem czasu, charakteryzująca się zmiennym krokiem czasowym.

W przypadku symulacji z czasem dyskretnym zmiany stanu procesu następują w stałych odstępach czasu, a więc dokładnie wiemy, kiedy te zmiany nastąpią. Podstawowymi elementami symulacji są stan procesu i czas. Należy podkreślić, że gdy w komputerze symulowane są procesy ciągłe, przyrost czasu symulowanego powinien być możliwie jak najmniejszy. Jego wielkość zależy od maszyny, na której jest wykonywany eksperyment. Podczas symulacji procesów dyskretnych czas symulowany zmienia się ze stałym krokiem. Wielkość tego kroku nie zależy od konkretnej realizacji sprzętowej.

Symulacja ze zmiennym krokiem (oparta na zdarzeniach) charakteryzuje się tym, że zmiany stanu systemu następują w nieregularnych odstępach czasu. Zazwyczaj nie wiemy *a priori*, kiedy taka zmiana nastąpi. Podstawowymi elementami symulacji są stan systemu, zdarzenia i czas. Chwile wystąpienia zdarzeń są nieodłączną częścią fizycznego systemu. Zdarzenia i stan są elementami dualnymi – zmiana stanu następuje w wyniku wystąpienia jakiegoś zdarzenia.

Zwróćmy uwagę na jeszcze jeden istotny element symulacji – skale czasowe, jakimi operujemy podczas wykonywania eksperymentu. W symulacji komputerowej czas symulacji procesów fizycznych jest równy czasowi obliczeń przeprowadzonych z modelem procesu i różni się od czasu, w jakim przebiega proces w rzeczywistości. Rozróżniamy więc dwa rodzaje czasów:

- czas symulacji – czas bieżący, w którym wykonywany jest eksperyment symulacyjny (czas w komputerze),
- czas fizyczny lub czas symulowany – czas przebiegu procesu w rzeczywistym systemie, ustalony przez badacza (podany w liczbach rzeczywistych lub całkowitych); nie jest związany z czasem bieżącym wykonywania eksperymentu symulacyjnego.

Prowadząc eksperymenty, należy w odpowiedniej kolejności symulować procesy fizyczne, tak by były zachowane związki przyczynowo-skutkowe w systemie.

Dalszą część niniejszego rozdziału poświęcimy sposobom realizacji eksperymentów symulacyjnych z wykorzystaniem maszyny jednoprocesorowej oraz w środowisku równoległym bądź rozproszonym.

10.2. Sekwencyjna symulacja dyskretna

Systemy rzeczywiste mają zazwyczaj charakter złożony. Można w nich wyodrębnić wiele podsystemów oddziałujących na siebie wzajemnie w pewnych chwilach czasu. System fizyczny może być więc reprezentowany przez graf powiązań podsystemów $G = (N, A)$, w którym N jest zbiorem węzłów (i-temu węzłowi odpowiada i-ty podsystem S_i), a A jest zbiorem łuków (i, j) (gdzie i i j mogą przyjmować wartości $1, \ldots, N$) łączących poszczególne węzły. Istnienie w grafie połączenia (i, j) oznacza, że podsystem S_j wpływa na stan podsystemu S_i. Oddziaływanie to można zapisać za pomocą zależności funkcyjnej

$$v_{ji}(t) = g_{ji}(x_j(t)) \qquad (i, j) \in A \tag{10.1}$$

gdzie: t – symulowany czas,
$v_{ji}(t)$ – oddziaływanie podsystemu S_j na podsystem S_i w chwili t,
$x_j(t)$ – stan podsystemu S_j,
$g_{ji} : X_j \rightarrow V_{ji}$, przy czym X_j – zbiór stanów podsystemu S_j, V_{ji} – zbiór oddziaływań S_j na S_i.

Model symulacyjny systemu fizycznego złożonego z wielu podsystemów może być konstruowany w postaci zbioru pewnych procesów logicznych PL_i, gdzie $i = 1, \ldots, N$, symulujących procesy fizyczne PF_i przebiegające w podsystemach S_i. Skoncentrujmy się na symulacji systemu, gdzie każdy proces fizyczny opisywany jest przez zbiór zdarzeń, a każdemu zdarzeniu przypisany jest czas jego wystąpienia. Symulacja takiego systemu opiera się na uporządkowanej liście zdarzeń, które wystąpią w przyszłości. Poszczególne zdarzenia są objęte relacjami zależności, co oznacza, że dane zdarzenie musi być poprzedzone innym. Zależność zdarzeń odzwierciedla naszą wiedzę o kolejności zjawisk zachodzących w symulowanym systemie fizycznym. Przystępując do realizacji eksperymentu symulacyjnego musimy zagwarantować to, by czas zdarzenia poprzedzającego wystąpienie innego zdarzenia był mniejszy. Można sformułować następujący algorytm symulacji sekwencyjnej:

symulacja sekwencyjna

```
GVT = 0
globalna lista zdarzeń = <...,(tk, mk),...>@
            //  mk - k-te zdarzenie,
            //  tk - czas wystąpienia k-tego zdarzenia
while(Warunek stopu symulacji nie jest spełniony)
{
    Sekwencyjna symulacja PFi,

    - usunięcie z listy zdarzeń zdarzenia k
      o znaczniku czasu: t = min tk; k = 1, ..., K,
                          K - liczba wszystkich zdarzeń
    - obsługa zdarzenia k
    - aktualizacja listy zdarzeń:
        - zapisanie do listy zdarzeń nowych zdarzeń
          będących rezultatem wykonania zdarzenia k
    - aktualizacja czasu: GVT = t
}
```

W symulacji sekwencyjnej wymienione powyżej operacje są wykonywane sekwencyjnie – jedna po drugiej – przez pojedynczy procesor komputera. Podejście sekwencyjne gwarantuje, że żadne zdarzenie nie zostanie zrealizowane przed zdarzeniem poprzedzającym je na liście zdarzeń, tzn., że żaden proces fizyczny PF_i nie otrzyma danych od procesów, które generują te dane w późniejszym terminie. Jest to realizowane dzięki zastosowaniu globalnej listy zdarzeń, na której operują wszystkie procesy logiczne PL oraz globalnego zegara GVT (ang. *Global Virtual Time*), wyznaczającego chwile wystąpienia zdarzeń.

10.3. Rozproszona symulacja dyskretna

W przypadku systemów rzeczywistych, składających się z wielu powiązanych wzajemnie podsystemów, często występują okresy czasu, w których poszczególne podsystemy działają niezależnie. Synchronizacja ich pracy następuje w momencie wymiany danych. Wykonywanie eksperymentów symulacyjnych przez pojedynczy procesor komputera może okazać się mało efektywne. W każdym cyklu symulacji może zostać zrealizowane i usunięte z listy zdarzeń tylko jedno zdarzenie. O czasie trwania eksperymentu decydują procesy logiczne, które wykonują swoje obliczenia najdłużej. Naturalne wydaje się w tej sytuacji zastosowanie technik zrównoleglenia obliczeń. Wykorzystanie wielu maszyn, połączonych w sieć, lub komputerów wieloprocesorowych zwiększa dostępne moce obliczeniowe. Można wymienić wiele czynników de-

cydujących o przewadze symulacji rozproszonej nad sekwencyjną realizacją na pojedynczej maszynie. Na przykład:

1. Realizacja rozproszona odzwierciedla rzeczywistą strukturę badanego systemu, składającego się z wielu powiązanych podsystemów, w których równolegle przebiegają różne procesy. Umożliwia to badanie wielu zjawisk zachodzących w systemach złożonych, takich jak: synchronizacja, zakleszczenia obliczeń, efektywna komunikacja, zarządzanie pamięcią, itd.
2. Skrócony jest czas obliczeń. Ma to szczególne znaczenie w sytuacjach gdy:
   - czas trwania eksperymentu jest bardzo długi i – dodatkowo – konieczne jest wielokrotne uruchomienie symulacji dla różnych danych wejściowych,
   - symulacja jest wykonywana w czasie rzeczywistym, podczas pracy interakcyjnej.
3. Rozwiązywane zadanie jest zbyt duże, by mogło być wykonane w pojedynczej maszynie (zbyt duże wymagania odnośnie zasobów komputera).
4. Symulacja może być dopasowana do istniejącej bazy sprzętowej, np. gdy dostępnych jest tylko kilka maszyn, to kilka procesów może być sekwencyjnie symulowanych w jednej maszynie.

Idea symulacji rozproszonej została zaproponowana przez Chandy'ego w 1977 roku i, niezależnie, przez Bryanta. W rozproszonej symulacji dyskretnej (RSD) zakłada się, że poszczególne procesy logiczne PL są wykonywane na różnych maszynach lub procesorach i porozumiewają się między sobą wysyłając komunikaty z odpowiadającymi im znacznikami czasowymi (tab. 10.3).

Tabela 10.1

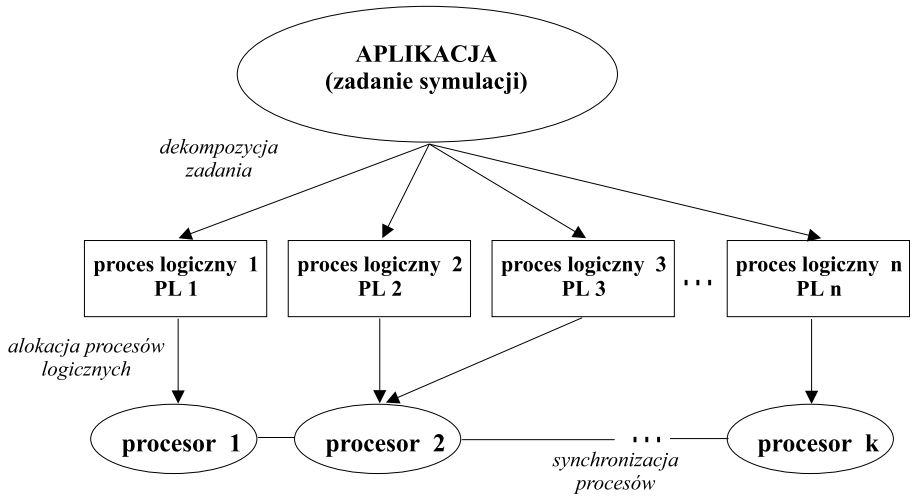
Postać komunikatu przesyłanego między procesami PL w RSD

czas wysłania komunikatu
nadawca komunikatu
odbiorca komunikatu
treść komunikatu
status (*pole nie zawsze wykorzystywane*)

Odpowiada to sytuacji rzeczywistej, w której poszczególne procesy fizyczne komunikują się między sobą. Związki przyczynowo-skutkowe procesów fizycznych są zachowane dzięki uwzględnianiu znaczników czasowych przychodzących komunikatów. Rezygnujemy z globalnej listy zdarzeń i – w przypadku symulacji asynchronicznej – z globalnego zegara. Każdy z procesów logicznych dysponuje swoją lokalną listą zdarzeń. Zdarzenia są uporządkowane zgodnie z przypisanymi im znacznikami czasowymi. Poszcze-

gólne procesy logiczne PL wykonują swoje obliczenia niezależnie, tak długo, jak długo niezależnie działają symulowane przez nie procesy fizyczne PF. Następnie PL są wstrzymywane w celu odebrania komunikatów z danymi od innych, powiązanych z nimi procesów logicznych, symulując w ten sposób interakcje zachodzące między fizycznymi podsystemami. Nie ma znaczenia, na jakiej platformie jest wykonywany eksperyment symulacyjny, czy jest to sieć komputerów, czy komputer wieloprocesorowy – koncepcja jest taka sama. Implementacja, w przypadku różnych platform może być oczywiście różna. Jest to zazwyczaj związane z koniecznością stosowania różnych narzędzi programowych zrównoleglenia obliczeń. Efektywność symulacji będzie również zależeć od realizacji sprzętowej.

Realizacja symulacji w wersji rozproszonej wymaga rozwiązania trzech podstawowych problemów (rys. 10.2): dekompozycji zadania, przydziału zadań do procesorów komputera wieloprocesorowego lub komputerów w sieci i dostarczenia mechanizmów synchronizacji obliczeń.



Rys. 10.2. Realizacja symulacji rozproszonej

10.3.1. Dekompozycja zadania symulacji

Przystępując do realizacji eksperymentu symulacyjnego w środowisku równoległym lub rozproszonym rozpoczyna się pracę od dekompozycji zadania obliczeniowego na zbiór mniejszych zadań, które mogą być wykonywane przez różne procesy (procesy logiczne PL), w szczególności na różnych procesorach. Proponowane są trzy podstawowe podejścia: podział dziedziny, tzn. bazy danych, na której jest wykonywany eksperyment symulacyjny

(ang. *domain decomposition* lub *data distribution*), podział funkcjonalny (ang. *functional decomposition* lub *algorithmic distribution*), dekompozycja realizowana dynamicznie (ang. *irregular decomposition*).

1. Dekompozycję wykorzystującą podział bazy danych stosuje się w zadaniach operujących na dużym zbiorze danych, gdy o czasie obliczeń w znacznym stopniu decyduje częste odwoływanie się do tego zbioru. Idea tego podejścia polega na takiej dekompozycji zadania i bazy danych, by poszczególne procesy logiczne mogły niezależnie operować na przypisanych im podzbiorach danych. Taka realizacja wpłynie w istotny sposób na przyspieszenie obliczeń w przypadku zadań zajmujących się głównie przetwarzaniem danych, w których różne operacje mogą być wykonywane równolegle na podzbiorach danych. Nie będzie ona jednak efektywna, np. w przypadku symulacji systemów kolejkowych.
2. Dekompozycję funkcjonalną stosuje się w zadaniach charakteryzujących się znaczną złożonością algorytmiczną, tam gdzie o czasie trwania obliczeń decyduje przede wszystkim złożoność obliczeniowa, a nie częste sięganie do zbioru danych. Zrównoleglenia dokonuje się na poziomie algorytmu. Poszczególne procesory wykonują różne operacje realizujące odpowiednie części algorytmu obliczeniowego. Podział winien być dokonany w taki sposób, aby różne procesory wykonywały zadania o podobnej złożoności obliczeniowej.
3. W przypadku dekompozycji zadania realizowanej dynamicznie podział na podzadania odbywa się w trakcie trwania eksperymentu symulacyjnego. Na bieżąco wprowadzane są zmiany w dekompozycji, gdy okazuje się, że dokonany wcześniej rozdział nie zapewnia równomiernego obciążenia procesorów. Oczywiste jest, że taki sposób dekompozycji powinien spowodować największe przyspieszenie obliczeń, jest on jednak najtrudniejszy w realizacji.
   Szczegółowe informacje, dotyczące sposobu dekompozycji zadań obliczeniowych, można znaleźć w literaturze, np. w pracach [9, 11].

10.3.2. Alokacja procesów logicznych

Procesy obliczeniowe, realizujące zadania powstałe w wyniku dekompozycji, są przydzielane do procesorów, w których będą wykonywane. Odpowiednia alokacja procesów logicznych ma często decydujący wpływ na efektywność symulacji. Po pierwsze, obliczenia muszą być w miarę możliwości równomiernie rozłożone między procesory. Po drugie, uwzględniając topologię systemu komputerowego, należy grupować procesy logiczne symulujące podsystemy fizyczne, które często wymieniają między sobą dane, by redukować nakład

czasu na komunikację. Alokacja procesów może być realizowana na dwa sposoby: statycznie i dynamicznie.

Podejście statyczne zakłada, że przydział do procesorów jest realizowany przed przystąpieniem do obliczeń. W przypadku, gdy liczba zadań jest równa liczbie procesorów, każde zadanie jest realizowane przez inny procesor. W sytuacji, gdy nie dysponujemy odpowiednio dużą liczbą procesorów, zadania są odpowiednio grupowane i każdy procesor wykonuje kilka zadań.

W podejściu dynamicznym przydział zadań do procesorów odbywa się na bieżąco, w trakcie trwania obliczeń. Procesy obliczeniowe oczekują w kolejce na przydział procesora. Opóźnienie, związane z czasem oczekiwania, zależy od liczby realizowanych procesów obliczeniowych i priorytetów nadanych procesom obliczeniowym.

Można zastosować również podejście pośrednie, łączące oba sposoby alokacji: statyczny i dynamiczny. Przed rozpoczęciem symulacji dokonujemy wstępnego przydziału procesów logicznych do procesorów, zakładając, że przydział ten może ulec zmianie podczas wykonywania eksperymentu. Na bieżąco sprawdza się obciążenie procesorów i dokonuje zmiany alokacji procesów w celu maksymalnego wykorzystania mocy obliczeniowej. Mówimy wówczas o migracji procesów. Często zdarza się, że zmiana przydziału procesów jest wymuszona, np. awarią procesora.

10.3.3. Synchronizacja obliczeń

W przypadku symulacji komputerowej realizowanej w środowisku rozproszonym, gdy celem wykonywanego eksperymentu jest naśladowanie procesów przebiegających w świecie rzeczywistym, możliwości zrównoleglenia obliczeń są ograniczone przez związki przyczynowo-skutkowe występujące w symulowanym systemie. Przyjmijmy, że wystąpienie zdarzeń E_1 i E_2 powoduje zmianę stanu systemu x. Jeżeli w rzeczywistości zdarzenie E_1 poprzedza zdarzenie E_2, to podczas symulacji obliczenia związane ze zdarzeniem E_1 muszą zostać wykonane przed obliczeniami związanymi z wystąpieniem zdarzenia E_2. Analogią jest tu składanie (superpozycja) przekształceń, które nie muszą być przemienne, tzn. $f_1(f_2(x)) \neq f_2(f_1(x))$. Wykonanie obliczeń w kolejności niezgodnej z wystąpieniem zdarzeń powoduje wystąpienie błędu przyczynowo-skutkowego. Tak więc sekwencyjna metoda symulacji nie może być w sposób automatyczny zaadaptowana w rozproszonym, asynchronicznym środowisku sprzętowym. Konieczne jest zastosowanie specjalnych mechanizmów synchronizacji obliczeń, które zagwarantują poprawny przebieg eksperymentu, zabezpieczą przed wystąpieniem błędów przyczynowo-skutkowych i zakleszczeniem programów.

Podobnie jak w przypadku obliczeń rozproszonych wyróżnia się dwa podejścia:
- symulację synchroniczną,
- symulację asynchroniczną.

Kryterium tego podziału jest sposób synchronizacji obliczeń. Brzmi to nieco dziwnie, ale w symulacji asynchronicznej konieczne jest zastosowanie pewnych mechanizmów synchronizacji obliczeń zabezpieczających przed wystąpieniem błędów przyczynowo-skutkowych. Termin *symulacja asynchroniczna* oznacza więc co innego niż asynchroniczne obliczenia. Wybrane techniki, stosowane do synchronizacji procesów obliczeniowych podczas eksperymentu symulacyjnego, są opisane w następnym podrozdziale.

10.4. Symulacja synchroniczna

W podejściu synchronicznym stosowany jest mechanizm globalnej synchronizacji procesów obliczeniowych. Może być on zrealizowany za pomocą globalnego zegara GVT (ang. *Global Virtual Time*) [43]. Listy zdarzeń są natomiast rozproszone – każdy z procesów logicznych operuje na lokalnej liście zdarzeń. Różnica w stosunku do symulacji wykonywanej tradycyjnie w sposób sekwencyjny polega na równoległym uruchamianiu procesów symulujących zdarzenia występujące w tej samej chwili czasu. Symulacja może być realizowana dwoma sposobami:

Symulacja sterowana zegarem: stan zegara zmienia się o takt i równolegle obsługiwane są zdarzenia z chwili równej stanowi zegara. Wszystkie procesy logiczne otrzymują informacje o stanie globalnego zegara.

Symulacja sterowana zdarzeniami: zegar przyjmuje wartości znaczników czasowych kolejnych zdarzeń. W tym przypadku proces zarządzający musi znać znaczniki czasowe pierwszych zdarzeń z list wszystkich procesów logicznych w celu ustalenia czasu zegara globalnego.

Można więc sformułować następujący algorytm symulacji rozproszonej wykonywanej w sposób synchroniczny:

symulacja synchroniczna

```
GVT = 0
lokalne listy zdarzeń procesów PLi = <...,(tk,mk),...>; i = 1,...,N

while(Warunek stopu symulacji nie jest spełniony)
{
      Równoległa symulacja procesów PFi
       - usunięcie z list zdarzeń procesów PLi
         zdarzeń o znaczniku czasu: t = GVT
       - obsługa usuniętych zdarzeń
```

```
            - aktualizacja list zdarzeń procesów PLi:
              - zapisanie do list procesów PLi nowych zdarzeń
                będących rezultatem wykonania usuniętych zdarzeń
            - aktualizacja czasu: GVT = min ti
              gdzie ti - znaczniki czasu pierwszych zdarzeń
              z lokalnych list zdarzeń procesów PLi
    }
```

Możliwości zrównoleglenia obliczeń w przypadku symulacji synchronicznej są bardzo ograniczone – równolegle są wykonywane tylko zdarzenia o znaczniku czasowym zgodnym z czasem wskazywanym przez globalny zegar. Synchronizacja globalna zabezpiecza przed wystąpieniem błędów przyczynowo-skutkowych oraz przed zakleszczeniem programów. Procesy logiczne PL symulujące procesy fizyczne PF są z nimi dokładnie zsynchronizowane. O kolejności uruchomienia procesów logicznych decyduje kolejność wystąpienia symulowanych przez nie zjawisk. Dla każdych dwóch procesów PF_i i PF_j, których czasy rozpoczęcia spełniają zależność $t_i < t_j$, czasy uruchomienia symulujących je procesów logicznych PL_i i PL_j spełniają zależność $t_{Li} < t_{Lj}$. Jednocześnie są symulowane tylko te procesy, które zachodzą w tej samej chwili. Oczywiście, inne są czasy biegu procesów w rzeczywistym systemie (czas symulowany), a inne są czasy obliczeń wykonanych przez ich symulatory (czas symulacji). Taka realizacja symulacji jest mało efektywna.

10.5. Symulacja asynchroniczna

W podejściu asynchronicznym następuje odejście od globalnej synchronizacji procesów. Dopuszcza się sytuacje, w których kolejność rozpoczęcia obliczeń przez symulatory jest inna niż kolejność wystąpienia zjawisk w fizycznym systemie, tzn. mogą wystąpić sytuacje, w których dla dwóch procesów PF_i i PF_j, takich że $t_i < t_j$, czasy uruchomienia symulujących je procesów PL_i i PL_j spełniają zależność $t_{Li} \geqslant t_{Lj}$. Aby eksperyment symulacyjny był wykonany poprawnie, konieczne jest zabezpieczenie obliczeń przed wystąpieniem błędu przyczynowo-skutkowego, gdy proces logiczny PL_2, symulujący zdarzenie zależne od zdarzenia symulowanego przez proces PL_1, wykona swoje obliczenia przed procesem PL_1. Zegar globalny w symulacji synchronicznej zabezpieczał przed wystąpieniem błędu przyczynowo-skutkowego. W symulacji asynchronicznej przyjmuje się następujące założenia:

- Każdy proces logiczny PL ma swoją lokalną, uporządkowaną listę zdarzeń. Lista ta jest aktualizowana w czasie symulacji.
- Każdy proces logiczny PL ma swój lokalny zegar LVT (ang. *Local Virtual Time*). Krok taktowania każdego z zegarów lokalnych może być różny.

– Postać komunikatu przesyłanego między procesami PL przedstawiono w tabeli 10.3.
– Podczas eksperymentów wykorzystuje się specjalne protokoły synchronizacji [9, 40, 56, 60]. Można je podzielić na trzy grupy: techniki konserwatywne, optymistyczne i hybrydowe.

10.5.1. Techniki konserwatywne

Idea technik konserwatywnych polega na unikaniu sytuacji wystąpienia błędu przyczynowo-skutkowego. Opracowano wiele metod. Ich wspólną cechą jest przyjęcie założenia, że na każdym etapie obliczeń wykonywane są tylko procesy obliczeniowe realizujące zdarzenia bezpieczne. Metody te różnią się sposobem identyfikowania takich zdarzeń.

10.5.1.1. Algorytm CMB

Chandy, Misra i Bryant są autorami jednego z pierwszych protokołów synchronizacji, oznaczmy go przez CMB [12, 56]. Zakłada on, że w danej chwili czasowej żaden proces obliczeniowy nie zostanie uruchomiony, dopóki nie będzie gwarancji, że nie otrzyma on danych z wcześniejszej chwili, od przekazujących do niego komunikaty procesów. Załóżmy, że nasz model symulacyjny składa się z PL_i $(i = 1, \ldots, N)$ procesów logicznych symulujących działanie procesów fizycznych PF_i $(i = 1, \ldots, N)$. Każdy proces logiczny PL_i symuluje działanie procesu fizycznego PF_i do chwili t tylko wtedy, gdy zna on stan początkowy i wszystkie komunikaty, które proces fizyczny PF_i otrzyma do chwili t. Oczywiste jest, że wszystkie komunikaty, które zostaną dostarczone po chwili t $(t_j > t)$, nie wpłyną na stan procesu fizycznego PF_i w chwili t. Tak więc, jeżeli w rzeczywistości wiadomość m zostanie przesłana przez proces PF_j do procesu PF_i w chwili t, to podczas symulacji komunikat postaci (t, m) zawierający wiadomość m zostanie przesłany od procesu logicznego PL_j do PL_i. Zakładamy przy tym, że jeżeli proces logiczny PL_j wysyła sekwencję komunikatów $\{\ldots(t_k, m_k), (t_{k+1}, m_{k+1})\ldots\}$, wówczas zachodzi $t_k < t_{k+1}$, itd. Z tego założenia wynika, że jeżeli proces logiczny PL_i otrzyma od procesu PL_j komunikat (t, m), oznacza to, że zna on wszystkie komunikaty przesłane przez proces fizyczny PF_j do procesu PF_i do chwili t włącznie. Znacznik czasowy każdego następnego komunikatu będzie większy. Przy takim założeniu możemy mieć gwarancję, że każdy proces logiczny PL_i zna komunikaty pochodzące od wszystkich powiązanych z nim procesów do chwili $T_i = \min\limits_{j=1,\ldots,J} t_j$. Przyjmijmy, że T_i oznacza stan zegara lokalnego PL_i, a więc $LVT_i = T_i$. Tak więc proces logiczny PL_i może bezpiecznie symulować działanie procesu fizycznego PF_i

do chwili LVT_i. Podstawowy algorytm działania PL_i, który zaproponowali Chandy i Misra, można przedstawić następująco:

symulacja asynchroniczna (algorytm CMB)

```
LVTi=0
lista zdarzeń procesu PLi = <...,(tk,mk),...>

 // Proces logiczny PLi zna komunikaty przesłane
 // do procesu fizycznego PFi
 // przez wszystkie procesy PFj, j = 1, ..., J, do chwili LVTi.

while(Warunek stopu symulacji nie jest spełniony)
{
  Symulacja procesu PFi do chwili LVTi:
  - usunięcie z listy zdarzeń procesu PLi,
    zdarzeń o znacznikach czasu t < LVTi
  - obsługa usuniętych zdarzeń,
  - aktualizacja listy zdarzeń procesu PLi i
    wysłanie komunikatów z wynikami obliczeń,
    <(t1, m1), (t2, m2), ... ,(tk, mk)>, gdzie t1 < t2, ..., < tk

    if(Przyszły nowe komunikaty od wszystkich procesów j = 1, ...,J)
    {
       - umieszczenie nowych zdarzeń na liście,
       - aktualizacja zegara procesu PLi, LVTi = min tj, j = 1,
...,J
    }
}
```

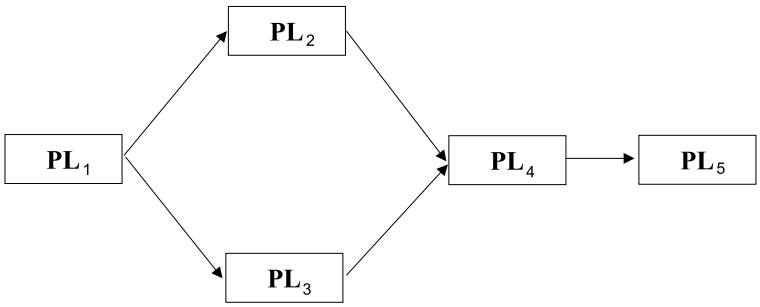
Zakleszczenie obliczeń

Przedstawiony powyżej protokół komunikacji nie gwarantuje, że w każdym przypadku obliczenia zakończą się sukcesem. Omówimy to na przykładach.

Przykład 10.1. Wstrzymanie obliczeń

Rozważamy system złożony z pięciu procesów fizycznych PF, którego schemat jest prezentowany na rys. 10.3. Zakładamy, że procesy logiczne PL symulujące poszczególne procesy fizyczne są wykonywane przez różne procesory. Proces PF_1 przesyła komunikaty z danymi do jednego z dwóch procesów PF_2 lub PF_3. Po upływie pewnego czasu proces PF_2 lub PF_3, w zależności od tego, który z nich otrzymał komunikaty od PF_1, przekazuje te komunikaty do procesu PF_4. Rozważmy przypadek, w którym proces PF_1 przesyła wszystkie komunikaty do procesu PF_2 (może to zrobić, gdyż to on wybiera kanał, którym zostaną przesłane dane). W ten sposób proces logiczny PL_4, symulujący działanie procesu PF_4, nigdy nie otrzyma komunikatu od procesu PL_3. Zgodnie z opisanym podstawowym algorytmem CMB stan zegara lokalnego procesu PL_4 wyznaczany jest jako $LVT_4 = \min(t_2, t_3)$, gdzie t_2 i t_3 oznaczają znaczniki czasu komunikatów

od procesów PL_2 i PL_3. Czas t_3 nie zmieni się i stale będzie równy zeru. Obliczenia zostaną wstrzymane, żaden komunikat nie zostanie przesłany do procesu PL_5.

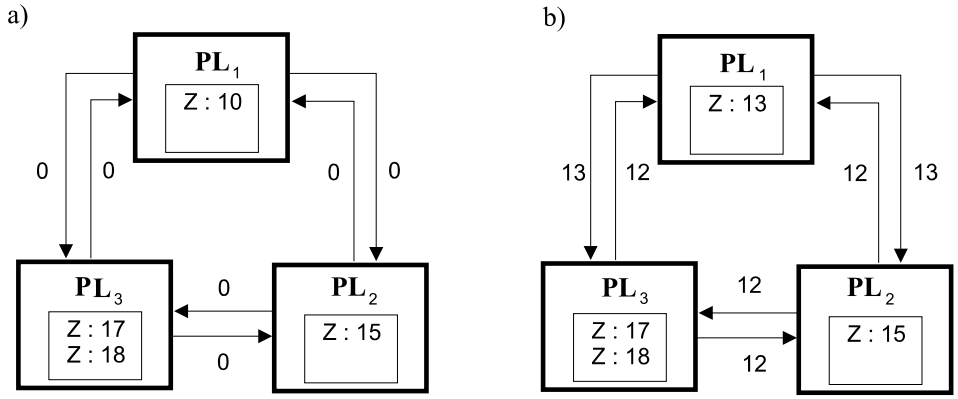


Rys. 10.3. Przypadek wstrzymania obliczeń

□

Przykład 10.2. Cykliczne oczekiwanie

Rozważamy system złożony z trzech procesów fizycznych, przedstawiony na rys. 10.4 a. Procesy są wykonywane przez różne procesory i działają w pętli.



Rys. 10.4. Symulacja z zastosowaniem mechanizmu CMB: a) inicjacja obliczeń, b) obsługa pierwszego zdarzenia z listy zdarzeń

Zakłada się dwukierunkową komunikację między powiązanymi procesami. Rysunek przedstawia aktualne lokalne listy zdarzeń, np. (Z:10) oznacza wystąpienie w chwili $t_1 = 10$ zdarzenia Z. Liczby przypisane łączom są znacznikami czasowymi ostatnich komunikatów, przesyłanych tymi połączeniami. Początkowe czasy przypisane łączom są równe 0. Oznacza to, że nie były wysłane jeszcze żadne komunikaty. Pierwsze zdarzenie, które powinno być obsłużone, ma znacznik czasowy 10. W związku z tym, że procesy

logiczne PL_i dysponują tylko wiedzą lokalną (nie znają list zdarzeń pozostałych procesów), żaden komunikat nie zostanie wysłany, gdyż proces logiczny PL_1 nie wyśle żadnego komunikatu do PL_2, zanim nie otrzyma danych od PL_3, PL_3 nie wyśle danych, dopóki nie otrzyma danych od PL_2, itd. Proces obliczeniowy znajdzie się w stanie cyklicznego oczekiwania. □

Metody rozwiązania problemu zakleszczeń

Zaproponowano kilka sposobów rozwiązania problemu zakleszczeń. Charakterystyczne dla sytuacji, w których dochodzi do wstrzymania obliczeń, jest to, że czas generowania przez procesy logiczne komunikatów nie ulega zmianie. Aby kontynuować obliczenia, wystarczy dostarczyć procesom dodatkowych informacji, tak więc w przykładzie 10.1 jest to wiadomość, że proces PL_3 nigdy nie prześle danych do procesu PL_4, a w przykładzie 10.2, że pierwsze zdarzenie wystąpi w chwili 10 i jego obsługa uruchomi działanie systemu. Pierwszym pomysłem na rozwiązanie zakleszczeń było zastosowanie komunikatów z pustymi wiadomościami (ang. *null messages*) [56]. Wysłanie przez proces PL_i do PL_j komunikatu $(t, null)$ oznacza, że proces PL_i nie prześle do PL_j wiadomości z czasem mniejszym niż t. Każdy następny przesłany komunikat (t^*, m^*) będzie spełniać warunek $t^* > t$. Komunikaty $(t, null)$ nie mają swojego odpowiednika w systemie rzeczywistym. Są wykorzystywane w systemie logicznym do informowania o braku nowych danych w chwili t. Reakcja procesów na komunikaty zawierające puste wiadomości jest identyczna jak na komunikaty z danymi. Wyznaczany jest nowy stan procesu PL_i, aktualizowany czas lokalnego zegara i wysyłane do związanych procesów komunikaty zawierające np. wyniki przeprowadzonych obliczeń.

Przykład 10.3. Zastosowanie mechanizmu CMB z pustymi komunikatami.

Powróćmy do przykładu 10.2 (rys. 10.4 a), w którym wystąpił problem zakleszczenia obliczeń. Zbadamy efekt zastosowania mechanizmu z pustymi komunikatami.

Załóżmy, że minimalny czas obsługi zdarzenia Δt_i jest równy 1 (czyli $\Delta t_i \geqslant 1$), tak więc znacznik czasowy komunikatu wysłanego w wyniku obsługi zdarzenia z chwili T_i spełnia warunek $t \geqslant T_i + 1$. W związku z tym, w chwili 0 wiadomo, że żaden z procesów nie otrzyma komunikatu ze znacznikiem czasowym $t < 1$. Każdy proces logiczny może wysłać do procesów odbierających od niego dane pusty komunikat $(1, null)$. Symulacja się rozpoczyna. Co okres czasu $\Delta t = 1$ wysyłane są puste komunikaty, aż do osiągnięcia chwili wystąpienia pierwszego zdarzenia $t = t_1 = 10$. Każdy z procesów wyśle więc 10 pustych komunikatów. Przyjmijmy, że czas obsługi pierwszego zdarzenia z listy jest równy 3, a więc do chwili $t = 13$ włącznie proces PL_1 będzie prowadził obliczenia i w tym czasie nie będzie wysyłał żadnych komunikatów. Proces PL_1 wysyła więc pusty komunikat

$(13, null)$ i przystępuje do obliczeń. W wyniku obsługi zdarzenia z chwili $t = 10$, w chwili $t = 13$ na liście zdarzeń procesu PL_1 pojawi się nowe zdarzenie. □

Mechanizm CMB z pustymi komunikatami może być z powodzeniem stosowany do symulacji systemów, w których dochodzi do częstej wymiany danych między procesami. Efektywność mechanizmu gwałtownie spada w przypadku, gdy komunikacja między procesami jest sporadyczna. Podczas symulacji takiego systemu konieczne jest przesyłanie wielu dodatkowych, pustych komunikatów. Zwiększa to obciążenie sieci komputerowej i wydłuża czas obliczeń.

Zaproponowano pewne modyfikacje protokołu CMB. Jedną z przyczyn wysyłania tak wielu pustych komunikatów jest fakt, iż niosą one bardzo ubogą informację. Poszczególne procesory mając dostęp do swojej listy zdarzeń mogą dołączyć do przesyłanej pustej wiadomości znacznik czasowy najwcześniejszego zdarzenia ze swojej listy zdarzeń t_i lub informacje o czasie wysłania przez siebie nowego komunikatu $t = t_i + \Delta t_i$, związanego z zakończeniem obsługi zdarzenia z chwili t_i (jeśli znany jest czas obsługi zdarzenia Δt_i). Takie podejście zostało nazwane *Carrier Null Message*. Zakłada ono, że przesłane puste komunikaty zawierają listę procesów, do których dotarł pusty komunikat i znaczniki czasowe najwcześniejszych zdarzeń z lokalnych list zdarzeń tych procesów. W przykładzie 10.2 (rys. 10.4) wiadomo jest, że pierwsze zdarzenie obsługiwane przez proces PL_1 będzie miało znacznik czasowy $t_1 = 10$. Czas obsługi zdarzenia Δt_1 jest równy 3. Do pustego komunikatu $(1, null)$ PL_1 można więc dołączyć informację o czasie wystąpienia pierwszego zdarzenia z listy, tj. $(PL_1, 10)$. Jedna kopia tego komunikatu dociera do PL_2, który dodaje informację $(PL_2, 15)$ i zwraca tak wzbogacony komunikat do PL_1 oraz dodatkowo przesyła go do PL_3. Proces PL_3 rozszerza otrzymany komunikat o $(PL_3, 17)$ i przesyła go do PL_1 i PL_2. Tym sposobem proces PL_1 otrzymuje informację o sytuacji w całej sieci i może natychmiast przystąpić do obsługi pierwszego zdarzenia z listy. Zaproponowane podejście zmniejsza liczbę dodatkowych komunikatów, ale, niestety, nadal znaczna ich liczba musi być generowana.

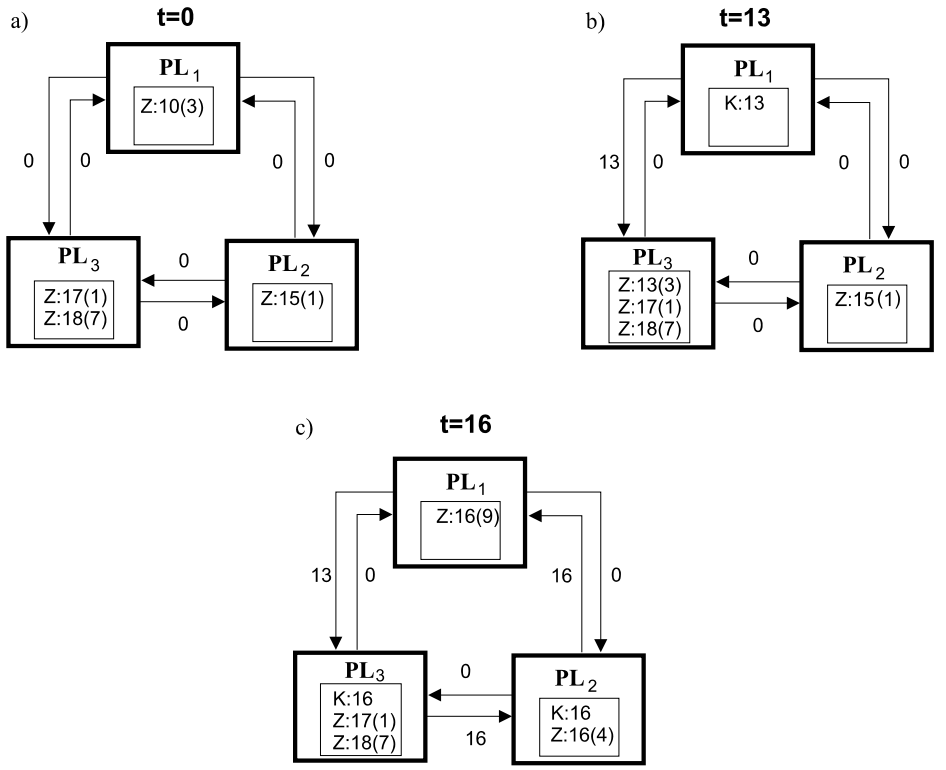
10.5.1.2. Schemat wykorzystujący okna

Inny algorytm został opisany w pracach [59, 60, 72]. Polega on na wyznaczaniu na osi czasu okna (przedziału czasu), w którym można bezpiecznie przetwarzać równolegle zdarzenia. Załóżmy, że każdy z procesów wykonał obliczenia do chwili t, w której dokonano globalnej synchronizacji. Zadaniem każdego procesu jest oszacowanie chwili t_i, w której wyśle on następny komunikat (przy założeniu, że nie otrzyma w tym czasie żadnych dodatkowych

komunikatów). Następnie, globalnie wybierany jest najmniejszy z czasów wyznaczonych przez wszystkie procesy ($t^*(t) = \min_i t_i$). W ten sposób definiowane jest okno $[t, t^*(t))$ i wszystkie procesy mogą równolegle symulować zdarzenia, których znaczniki czasowe przyjmują wartości z tego okna.

Przykład 10.4

Przyjrzyjmy się implementacji tego algorytmu na przykładzie przedstawionym na rys. 10.5. Zakładamy, że wszystkie procesy zostały wstępnie zsyn-



Rys. 10.5. Symulacja z zastosowaniem schematu wykorzystującego okna: a) inicjacja, wyznaczone okno $[0, 13)$; b) po obsłudze pierwszego okna, wyznaczone okno $[13, 16)$; c) po obsłudze drugiego okna, wyznaczone okno $[16, 18)$

chronizowane w chwili $t = 0$. Na rysunku przedstawione są listy zdarzeń procesów. W nawiasach umieszczono przewidywane czasy związane z obsługą zdarzeń. Tak więc, w wyniku obsługi przez proces PL_1 zdarzenia z chwili 10 nastąpi przesłanie komunikatu z wynikami w chwili 13. Każdy z procesów, uwzględniając czasy wystąpienia zdarzeń i czasy ich obsługi, oblicza czas, w którym wyśle następny komunikat $t'_i(0) = t_i + \Delta t_i$, tj. $t'_1(0) = 13$, $t'_2(0) = 16$, $t'_3(0) = 18$. Na tej podstawie jest wyznaczane okno czasowe: $[0, 13)$. Znacznik tylko jednego zdarzenia (Z:10) zawiera się w wyznaczonym

oknie czasowym i tylko to zdarzenie będzie obsłużone. Załóżmy, że wyniki wykonanych obliczeń zostaną przesłane do procesu PL_3. Biorąc pod uwagę przewidywany czas obsługi, proces PL_1 wysyła do procesu PL_3 komunikat informujący o przewidywanym w chwili $t = 13$ komunikacie z nowymi danymi i przystępuje do obsługi zdarzenia (Z:10). Na rysunku 10.5 b prezentowana jest sytuacja tuż przed wyznaczeniem nowego okna. Proces PL_1 zakończył obliczenia i na swojej liście zdarzeń umieścił zdarzenie związane z wysłaniem komunikatu zawierającego wyniki wykonanych obliczeń (K:13). Proces PL_3 umieścił na swojej liście zdarzeń zdarzenie związane z przewidywanym odebraniem od procesu PL_1 komunikatu (K:13). W chwili $t = 13$ procesy przystępują do wyznaczenia następnego okna, tj., $[13, 16)$. Procesy PL_1, PL_2 i PL_3 mogą przystąpić do obsługi zdarzeń z tego okna itd., aż do chwili, w której listy zdarzeń procesów będą puste. □

Omawiany w przykładzie 10.4 mechanizm zakłada, że jesteśmy w stanie przewidzieć *a priori* czas potrzebny na obsługę zdarzeń. Okna wyznaczane są globalnie, dla całego systemu. Podstawowym ograniczeniem tej metody jest to, że zdarzenia, występujące w różnych oknach, są przetwarzane sekwencyjnie, a równolegle mogą być realizowane jedynie zdarzenia z tego samego okna.

Inne podejście wykorzystujące okna, prezentowane w pracach [2, 3], zakłada wyznaczanie lokalnego okna dla każdego procesu ($t_i^* = \min_{j \in J} t_i^j$, j – procesy przesyłające dane do procesu i). W tym przypadku, podczas wyznaczania okna procesu PL_i, uwzględnia się listy zdarzeń tylko tych procesów, które wysyłają komunikaty bezpośrednio do PL_i. Wyznaczane okna dla różnych procesów będą więc różne. W danym oknie zdarzenia są przetwarzane sekwencyjnie, zdarzenia realizowane w różnych oknach są przetwarzane równolegle. Efektywność tego mechanizmu zależy od odpowiedniego rozmieszczenia okien czasowych w procesorach. Liczba tych okien i ich długość zależy od konkretnej aplikacji.

10.5.2. Techniki optymistyczne

W metodach optymistycznych dopuszcza się możliwość wystąpienia błędu przyczynowo-skutkowego i podaje sposoby na usunięcie skutków, spowodowanych wystąpieniem takiego błędu.

10.5.2.1. Mechanizm z zawijaniem czasu

Klasycznym reprezentantem technik optymistycznych jest mechanizm z zawijaniem czasu (ang. *rollback mechanism* lub *Time Warp Mechanism*). Zasada działania tego mechanizmu jest następująca. Przyjmuje się założenie,

że rozważany proces obliczeniowy PL_i nie otrzyma komunikatu od innego procesu z czasem symulacji mniejszym lub równym lokalnemu czasowi procesu LVT_i. Tak więc proces PL_i przystępuje do przetwarzania kolejnych zdarzeń z listy. Obliczenia są prowadzone do momentu otrzymania komunikatu z czasem symulacji mniejszym lub równym lokalnemu czasowi LVT_i. Proces PL_i jest wówczas zmuszany do cofnięcia swojego lokalnego zegara do czasu zapisanego w komunikacie i wycofania się z obliczeń wykonanych po tym czasie. Ponieważ proces mógł wcześniej wysłać do powiązanych z nim procesów własne komunikaty z większym znacznikiem czasu, procesy, które te komunikaty otrzymały, muszą być o tym fakcie poinformowane. Proces PL_i wysyła „antykomunikaty" $(t, j, i, m, *)$ unieważniające wszystkie wiadomości, których znaczniki czasowe były większe od chwili wystąpienia błędu.

Tak więc każdy proces PL_i realizuje następujący algorytm:

symulacja asynchroniczna (algorytm Time Warp)

```
LVTi = 0
lista zdarzeń procesu PLi = <..., (tk, mk), ...>

while(Warunek stopu symulacji nie jest spełniony)
{
    Symulacja procesu PFi:

     - usunięcie i obsługa kolejnych zdarzeń
     - aktualizacja listy i zegara
     - wysłanie komunikatów do innych procesów

    if(Odebrano komunikat (t, j, i, m))
    {
        if(W buforze nie znajduje się antykomunikat (t, j, i, m, *))
        {
            - wstawienie komunikatu (t, j, i, m) do bufora
           if(t <= LVTi)
           {
               Cofnięcie symulacji:
                - LVTi = t,
                - usunięcie z bufora stanów procesu
                  i komunikatów z chwil t* > t,
                - wysłanie antykomunikatów (t*, i, k, m, *)
            }
        }
    }
}
```

Wyróżniamy dwa rodzaje unieważnień („antykomunikatów"):

- agresywne, wysyłane od razu, gdy żąda się cofnięcia czasu,
- leniwe, wysyłane tylko w tych sytuacjach, gdy nadejście spóźnionego

komunikatu wpłynęłoby na wynik obliczeń, a więc zmieniło nadane wiadomości.

Jeden antykomunikat może wyzwolić całą lawinę antykomunikatów, gdyż inne procesory mogą również anulować przesłane przez siebie wiadomości. Niebezpieczeństwo wystąpienia takiej sytuacji jest tym większe, im większe są różnice w szybkościach poszczególnych procesorów. Może być ono zmniejszone poprzez odpowiednią dekompozycję zadania i alokację procesów obliczeniowych.

Należy zwrócić uwagę na jeszcze jeden, istotny mankament mechanizmu z zawijaniem czasu – duże wymagania odnośnie zasobów komputera. Cofnięcie obliczeń do dowolnej chwili symulowanego czasu wymaga przechowywania w pamięci każdego procesora następujących danych:

- stanu procesu obliczeniowego ze wszystkich chwil czasowych $t \leqslant LVT_i$,
- wszystkich odebranych komunikatów z danymi (które nie były anulowane),
- wszystkich odebranych antykomunikatów do komunikatów, które jeszcze nie zostały odebrane (zakładamy, że komunikaty mogą być dostarczane w innej kolejności niż były wysłane),
- wszystkich wysłanych komunikatów.

Proponuje się różne metody efektywnego zarządzania pamięcią komputera. Są to zarówno rozwiązania sprzętowe, jak i programistyczne [60, 73]. Jednym z podejść programistycznych jest periodyczne sprawdzanie i zapisywanie stanu. Podejściem alternatywnym jest zapamiętywanie na bieżąco tylko części wektora stanu – tej, która uległa zmianie w wyniku realizacji zdarzenia. Jest to tzw. podejście przyrostowe (ang. *state increment*) [60]. Analiza porównawcza obu technik: periodycznej i przyrostowej znajduje się w pracy [64].

Zbadajmy teraz działanie mechanizmu z zawijaniem czasu na przykładzie 10.4. Zakładając, że nie zostaną przesłane komunikaty z wcześniejszymi znacznikami czasowymi, każdy z trzech procesów przystępuje do obsługi pierwszego zdarzenia ze swojej listy. Komunikat przesłany od procesu PL_1 do PL_3, po przetworzeniu przez PL_1 zdarzenia z chwili 10, będzie miał znacznik czasowy 13. W związku z tym, że zegar procesu PL_3 po obsłudze zdarzenia z chwili 17 wskazuje 18, obliczenia i zegar T_3 zostaną cofnięte i będą wysłane odpowiednie antykomunikaty.

Proponowane są różne modyfikacje mechanizmu z zawijaniem czasu. Ich celem jest zmniejszenie liczby antykomunikatów – np. przesyłanie antykomunikatów tylko do tych procesów, w przypadku których spodziewamy się, że mogą one wpłynąć na przebieg obliczeń.

10.5.3. Techniki hybrydowe

Podejścia hybrydowe są połączeniem dwóch poprzednich metod [2, 60]:

Konserwatywno-optymistyczne

W tym przypadku przyjmuje się, że rozważany proces obliczeniowy przetwarza dane wejściowe, ale nie wysyła ich do innych, powiązanych z nim procesów, dopóki nie otrzyma specjalnego komunikatu zezwalającego na przesłanie danych. Mechanizm pozwala na przyspieszenie obliczeń i lepsze wykorzystanie mocy obliczeniowych w stosunku do podejścia konserwatywnego.

Optymistyczno-konserwatywne

Metoda ta działa tak jak metoda optymistyczna, ale tylko w pewnym okresie czasu (oknie). Między oknami następuje synchronizacja. Procesy obliczeniowe, których zegary lokalne osiągnęły czas równy długości okna, są wstrzymywane do momentu, aż wszystkie pozostałe procesy osiągną tę chwilę. Wtedy wszystkie procesy przechodzą do następnego okna. Cofanie czasu występuje tylko w obrębie okna (nie można cofnąć obliczeń do innego okna). W ten sposób zmniejszają się wymagania odnośnie zasobów komputera – mniejsza liczba danych musi być przechowywana w pamięci.

Przełączanie metod

Zakłada się możliwość zmiany stosowanej metody – przełączenie z podejścia konserwatywnego na optymistyczne lub odwrotnie.

Zdjęcia (migawki)

Metoda polega na zapamiętywaniu wartości wyjść procesów co pewien okres czasu (tzw. zdjęcia wyjść procesów obliczeniowych). W przypadku, gdy przez pewien okres czasu stany wyjść nie ulegają zmianie, zakłada się, że nastąpiło zakleszczenie obliczeń. Należy w takiej sytuacji odpowiednio zareagować.

Wydaje się, że podejścia hybrydowe, polegające na odpowiednim połączeniu technik optymistycznych i konserwatywnych, zaczynają obecnie dominować. Stale pojawiają się nowe pomysły, jak łączyć obie metody.

Na zakończenie rozważań dotyczących symulacji rozproszonej należy podkreślić, że implementacja symulatora działającego w sposób asynchroniczny jest znacznie trudniejsza niż symulatora synchronicznego. Ponieważ symulator działa asynchronicznie, kłopotliwe jest jego testowanie. Z powodu skomplikowanego kodu programów może się nawet okazać, że gotowy produkt jest wolniejszy niż jego synchroniczny odpowiednik. Wydaje się, że stosowanie symulacji asynchronicznej jest opłacalne jedynie w przypadku bar-

dzo rozbudowanych systemów z wieloma jednostkami (w tym jednostkami działającymi niezależnie od siebie), o różnej złożoności obliczeniowej.

Nie omawiamy w niniejszej monografii języków i środowisk oprogramowania do symulacji rozproszonej. Takie narzędzia są tworzone w wielu ośrodkach akademickich oraz firmach zajmujących się oprogramowaniem. Autorzy monografii są również współtwórcami środowiska oprogramowania do realizacji asynchronicznej symulacji równoległej zorientowanej na zdarzenia. Jest to system CSA&S-PV (*Complex Systems Analysis & Simulation – Parallel Version*), [62, 82]. Program łącznie z pełną dokumentacją jest udostępniony na stronie `http://www.ia.pw.edu.pl/~karbowsk/orr`.

W chwili obecnej wydaje się, iż standardem dla symulacji rozproszonej stanie się HLA (ang. *High Level Architecture*), opracowany przez Departament Obrony Stanów Zjednoczonych. Informacje dotyczące HLA oraz opis aplikacji realizowanych w tym standardzie można znaleźć m.in. w pracach [70, 81].

Bibliografia

[1] Arabas J., *Wykłady z algorytmów ewolucyjnych*, WNT, Warszawa 2001.

[2] Ayani R., Distributed Discrete Event Simulation, *Proceedings of System Modeling Control Conference*, Zakopane 1995.

[3] Ayani R., Rajaei H., Event Scheduling in Window Based Parallel Simulation Scheme, *Proceedings of the 4th IEEE Symposium on Parallel and Distributed Computing*, 1992.

[4] Bazaraa M.S., Sherali H.D., Shetty C.M., *Nonlinear Programming, Theory and Algorithms*, Wiley & Sons, Inc, 1993.

[5] Ben-Ari M., *Podstawy programowania współbieżnego i rozproszonego*, WNT, Warszawa 1996.

[6] Bertsekas D.P. i Tsitsiklis J.N., *Parallel and Distributed Computation: Numerical Methods*, Prentice Hall 1989.

[7] Bertsekas D.P., Tsitsiklis J.N., Some Aspects of Parallel and Distributed Iterative Algorithms - A Survey, *Automatica*, Vol. 27, No 1, pp. 3–21, 1990.

[8] Bertsekas D.P., *Nonlinear Programming*, Athena Scientific, Belmont, Massachusetts 1995.

[9] Bertsekas D.P., Gallager R., *Data Networks*, Prentice-Hall Int., Inc, 1992.

[10] Bielecki J., *Java 3 RMI: podstawy programowania rozproszonego*, Wydawnictwo Helion, Gliwice 1999.

[11] Brown Ch., *UNIX Distributed Programming*, Prentice Hall Int., 1994.

[12] Bryant R.E., *Simulation of Packet Communication Architecture Computer Systems*, MIT LCS – TR–188, Massachusetts Institute of Technology, 1977.

[13] Cellier F.E., *Continuous System Modeling*, Springer-Verlag 1991.

[14] Chandy K.M., Misra J., *Distributed Simulation: A Case Study in Design and Verification of Distributed Programs*, IEEE Transactions on Software Engineering, Vol. SE-5, No 5, pp. 440–452, 1979.

[15] Cichocki A., Unbehauen R., *Neural Networks for Optimization and Signal Processing*, John Wiley & Sons, Chichester 1993.

[16] Comer D.E., *Sieci komputerowe TCP/IP*, WNT, Warszawa 1998.

[17] Comer D.E., *Sieci komputerowe i intersieci*, WNT, Warszawa 1999.

[18] Conforti D., Musmanno R., Convergence and Numerical Results for a Parallel Asynchronous Quasi-Newton Method, *Journal of Optimization Theory and Applications*, Vol. 84, No 2, pp. 293–310, 1995.

[19] Coulouris G., Dollimore J., Kindberg T., *Systemy rozproszone. Podstawy i projektowanie*, WNT, Warszawa 1999.

[20] Culler D.E., Singh J.P. i Gupta A., *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufmann Publishers Inc., San Francisco 1998.

[21] Cybenko G., Approximation by Superpositions of a Single Function, *Mathematics of Control, Signals and Systems*, Vol. 2, pp. 303–314, 1989.

[22] Davis R., *Windows NT Network Programming*, Addison-Wesley 1994.

[23] Dekkers A., Aarts E., Global Optimization and Simulated Annealing, *Mathematical Programming*, No 50, pp. 367–393, 1991.
[24] Dongarra J.J., Meuer H.W. i Strohmaier E., TOP500 Supercomputer Sites (Raport publikowany 2 razy w roku), `http://www.top500.org`
[25] Findeisen W., Bailey F.N., Brdyś M., Malinowski K., Tatjewski P., Woźniak A., *Control and Coordination in Hierarchical Systems*, John Wiley & Sons, 1980.
[26] Findeisen W., *Struktury sterowania dla złożonych systemów*, Oficyna Wydawnicza PW, Warszawa 1997.
[27] Flynn M.J., Some Computer Organizations and Their Effectiveness, *IEEE Transactions on Computers*, Vol. C-21, No 9, pp. 948–960, 1972.
[28] Freeman T.L., Phillips C., *Parallel Numerical Algorithms*, Prentice Hall, UK, 1992.
[29] Gabassi M., Dupouy B., *Przetwarzanie rozproszone w systemie UNIX*, Lupus, 1995.
[30] Gelernter D., Generative Communication in Linda, *ACM Transactions on Programming Languages and Systems*, Vol. 7, No 1, pp. 80–112, 1985.
[31] Geist G.A., Kohl J.A., Papatopoulos P.M., PVM and MPI: a Comparison of Features, *Calculateuris Paralleles*, Vol. 8, No 2, 1996, `http://www.epm.ornl.gov/pvm/pvm_home.html`
[32] Goldberg D.E., *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley, Reading, 1989.
[33] Golub G., Ortega J.M., *Scientific Computing: An Introduction with Parallel Computing*, Academic Press, Inc., San Diego, 1993.
[34] Gropp W., Lusk, E., Skjellum A., *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, MIT Press, 1994.
[35] Hall M., Towfiq M., Geoff A., Treadwell D., Sanders, H., *Windows Sockets, An Open Interface for Network Programming under Microsoft Windows, Version 1.1*, `ftp://sunsite.unc.edu/pub/micro/pc-stuff/ms-windows/winsock/winsock-1.1/`, 1993.
[36] Heller M., *Advanced Win32 Programming*, John Wiley, 1993.
[37] Hornik K., Stinchcombe M., White H., Multilayer Feedforward Networks are Universal Approximators, *Neural Networks*, Vol. 2, pp. 359–366, 1989.
[38] Horst R., Tuy H., *Global Optimization Deterministic Approaches*, Springer-Verlag, 1989.
[39] Horst R., Pardalos P.M. (eds.), *Handbook of Global Optimization*, Kluwer Academic Publishers, 1995.
[40] Jefferson D.R., *Virtual time*, ACM Transactions on Programming Languages and Systems, 7(3), pp. 404–425, 1985.
[41] Jermolew Ju. M., *Metody stochasticzeskogo programirowania*, Nauka, Moskwa 1976.
[42] Karbowski A., Niewiadomska-Szynkiewicz E., *Distributed Global Optimization in Complex Control Systems; Case Study Results, Proceedings of the IEEE International Symposium on Industrial Electronics ISIE'96*, Vol. 1, pp. 272–276, Warszawa 1996.
[43] Kheir N.A., *Systems Modeling and Computer Simulation*, Marcel Dekker, Inc, 1996.
[44] Kirkpatrick S., Gelatt C.D., Vecchi M.P., Optimization by Simulated Annealing, *Science 220*, pp. 671–680, 1983.
[45] Kobyliński R., Karbowski A., *Środowisko WDM (Windows Distributed Machine) do tworzenia aplikacji równoległych i rozproszonych na platformie MS Windows*, Raport IAiIS PW, nr 99–57, 1999.
[46] Kobyliński R., Karbowski A., *Windows Distributed Machine – the First Step to Distributed Simulation World, Proc. ASTC 2000 – Advanced Simulation Technologies Conference*, SCS, Waszyngton 2000.
[47] Kozielski S., Szczerbiński Z., *Komputery równoległe*, WNT, Warszawa 1994.
[48] Loveman J., *Programowanie w Windows NT*, PLJ, Warszawa 1993.

[49] Marshall B., *Win32 System Services*, Prentice Hall, 1994.
[50] Meewella C.C., Mayne D.Q., *An Algorithm for Global Optimization of Lipschitz Continuous Functions, Journal of Optimization Theory and Applications*, Vol. 57, No 2, pp. 307–332, 1988.
[51] Meewella C.C., Mayne D.Q., *Efficient Domain Partitioning Algorithms for Global Optimization of Rational and Lipschitz Continuous Functions,* Journal of Optimization Theory and Applications, Vol. 61, No 2, pp. 247–270, 1989.
[52] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*, `http://www.mpi-forum.org/docs/mpi-11.ps`, 1995.
[53] Message Passing Interface Forum, *MPI-2:Extensions to the Message-Passing Interface*, `http://www.mpi-forum.org/docs/mpi-20.ps`, 1997.
[54] Michalewicz Z., *Algorytmy genetyczne + struktury danych = programy ewolucyjne*, WNT, 1996.
[55] Microsoft Corporation, *Introduction to the Win32 Application Programming Interface*, `http://www.microsoft.com/win32dev/apiext/intro32.htm`, 1994.
[56] Misra J., Distributed Discrete-Event Simulation, *Computing Surveys*, Vol. 18, No 1, 1986.
[57] Mowbray T.J., Zahavi R., *The Essential CORBA*, Wiley & Sons, Inc, 1995. **12**, pp. 45–68, 1973.
[58] Nelder J.A., Mead R., A Simplex Method for Function Minimization, *Computer Journal*, Vol. 7, pp. 308–313, 1965.
[59] Nicol D.M., *The Cost of Conservative Synchronization in Parallel Discrete-Event Simulation,* ACM, 40, pp. 304–333, 1993.
[60] Nicol D.M., Fujimoto R., *Parallel Simulation Today,* Annals of Operations Research Vol. 53, pp. 249–285, 1994.
[61] Nicol D.M., Reynolds P.F, *Optimal Dynamic Remaping of data Parallel Computations,* IEEE Trans. Comp., C-39, pp. 206–219, 1990.
[62] Niewiadomska-Szynkiewicz E., Warchoł M., Żmuda M., *Software Environment for Distributed Simulation of Complex Systems*, Preprints IFAC/IFORS/IMACS Symposium „Large Scale Systems: Theory and Applications", Patras, Greece, 1998.
[63] Pierwozwanski A.A., *Matematiczeskije modeli w uprawlenii proizwodstwom*, Nauka, Moskwa 1975.
[64] Palaniswamy A.C., Wilsey P.A., *An Analytical Comparison of Periodic Checkpointing and Incremental State Saving*, Proc. of the 7th Workshop on Parallel and distributed Simulation, pp. 127–134, 1993.
[65] Poliak B.T., Cypkin J.Z., *Psiewdogradientnyje ałgoritmy adaptacii i obuczenija,* Awtomatika i Tielemiechanika, Vol. 12, pp. 45–68, 1973.
[66] Press, W.H., Teukolsky, S.A., Vettering, T.W., Flannery, B.P., *Numerical Recipes in C*, Cambridge University Press, 1992.
[67] Price W.L., *Global Optimization Algorithms for a CAD Workstation,* Journal of Optimization Theory and Applications, Vol. 55, No 1, pp. 133–146, 1987.
[68] Reiher P.L., Jefferson D., *Dynamic Load Management in the Time Warp Operating System,* Trans. Soc. Comp., Simul., No 7, pp. 91–100, 1990.
[69] Rochkind M.J., *Programowanie w systemie UNIX dla zaawansowanych*, WNT, Warszawa 1993.
[70] SCS, *Simulation*, Special Issue „High Level Architecture", Vol. 73, No 5, 1999.
[71] Shir M., Otto S.W., Huss-Leolerman S., Walker D.W., Dongaria J., *MPI: The Complete Reference*, MIT Press, 1994.
[72] Sokol L.M., Briscoe D.P., Wieland, A.P., *MTW: A Strategy for Scheduling Discrete Simulation Events for Concurrent Execution,* Proc. SCS Multiconference on Distributed Simulation, Vol. 19, pp. 34–42, 1988.

[73] Soliman H.M., *On the Selection of the State Saving Strategy in Time Warp Parallel Simulation,* Transactions of the Society for Computer Simulation, Vol. 16, No 1, pp. 32–36, 1999.
[74] Törn A., Žilinskas A., *Global Optimization*, Springer Verlag, 1989.
[75] Tanenbaum A.S., *Rozproszone systemy operacyjne*, PWN, Warszawa 1997.
[76] Stachurski A., Wierzbicki A., *Podstawy optymalizacji*, Oficyna Wydawnicza PW, Warszawa 1999.
[77] Stevens W.R., *Programowanie zastosowań sieciowych w systemie UNIX*, WNT, Warszawa 1998.
[78] van der Steen A.J., *Overview of recent supercomputers*, NCF/Dept. of Computational Physics, Utrecht University, `http://www.phys.uu.nl/~steen`, 2000.
[79] Weiss Z., Gruźlewski T., *Programowanie współbieżne i rozproszone w przykładach i zadaniach*, WNT, Warszawa 1993
[80] WinSock Group, *Windows Sockets 2 Application Programming Interface*, `ftp://download.intel.com/ial/winsock2/`, 1996.
[81] Zeigler B.P., Praehofer H., Kim T.G., *Theory of Modeling and Simulation*, Academic Press, 2000.
[82] Żmuda M., Niewiadomska-Szynkiewicz E., *Parallel Simulation of Complex Systems, Software Environment and Applications,* Proc. 13th European Simulation Multiconference, SCS, Warszawa 1999.

Skorowidz

E

F

G

H

Q

R

S

Ś

T

U

W