Weryfikacja oprogramowania, korzystającego z MPI

Krzysztof Nozderko
kn201076@students.mimuw.edu.pl

23 maja 2005

Wprowadzenie

Równoległe obliczenia

- olbiczenia naukowców są często bardzo kosztowne obliczeniowo
- oprogramowanie naukowe często korzysta ze współbieżności
- obliczenia są dzielone między różne komputery
- komunikacja między współbieżnymi procesami poprzez interfejs MPI (Message Passing Interface)

Przykładowy program naukowy

Diffusion2d

- program wylicza funkcję $u : X \times Y \rightarrow Z$
- dziedzina funkcji u jest dyskretna, dwuwymiarowa
- funkcja u ewoluuje - wartości funkcji zmieniają się w kolejnych krokach
- zmiany te są opisane przez jakieś równania
- obliczone wartości funkcji są okresowo zapisywane

Message-Passing Interface

MPI - co to jest?

- standard, definujący składnię i semantykę dla biblioteki funkcji, wykorzystywanych do komunikacji między współbieżnymi procesami
- początki w latach 90-tych
- pierwsze biblioteki dla programów w C i Fortranie
- obecnie dostępne różne open-source'owe implementacje wysokiej jakości
- rzeczywisty standard dla równoległego oprogamowania naukowego

MPI - podstawy

- jest wiele autonomicznych programów, każdy wykonuje swój własny kod
- komunikacja między nimi poprzez wywołanie funkcji MPI
- standard zawiera 140 funkcji, my omówimy kilka najprostszych wykorzystywanych przez Diffusion2d

Pojęcia

- komunikator (*communicator*) - opisuje zbiór procesów, które mogą się ze sobą komunikować
  - jeśli komunikator zawiera n procesów
  - numeruje je sobie od 0 do n - 1
  - MPI_COMM_WORLD - predefinowany komunikator zawierający wszystkie procesy
- rank (*rank*) - identyfikator procesu w obrębie komunikatora

Wysyłanie i odbieranie

MPI_SEND(buf, count, datatype, dest, tag, comm)
MPI_RECV(buf, count, datatype, source, tag, comm, status)

buf	tablica z danymi do wysłania / gdzie otrzymane dane będą skopiowane
count	ile danych chcę wysłać / co najwyżej ile danych mogę otrzymać
datatype	typ danych które wysyłam / dostaję
dest	rank procesu do którego wysyłam
source	rank procesu od którego odbieram
tag	by można było rozróżniać wiadomości (id wiadomości)
comm	komunikator
status	parametr wyjściowy, zawiera informacje o otrzymanej wiadomości

W przypadku odbierania wiadomości source i tag mogą być odpowiednio MPI_ANY_SOURCE i MPI_ANY_TAG

Komunikacja

- MPI dostarcza różnych funkcji do przesyłania komunikatów synchronicznie i asynchronicznie, w różnych trybach
- komunikaty wysyłane przez jeden proces do drugiego procesu odbierane są w prawidłowej kolejności
- komunikaty wysłane przez różne procesy do tego samego odbiorcy mogą być odebrane w dowolnej kolejności

Jednoczesne wysłanie i odebranie wiadomości

MPI_SENDRECV(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf, recvcount, recvtype, source, recvtag, comm, status)
(proces rozdziela się na 2 niezależne wątki)

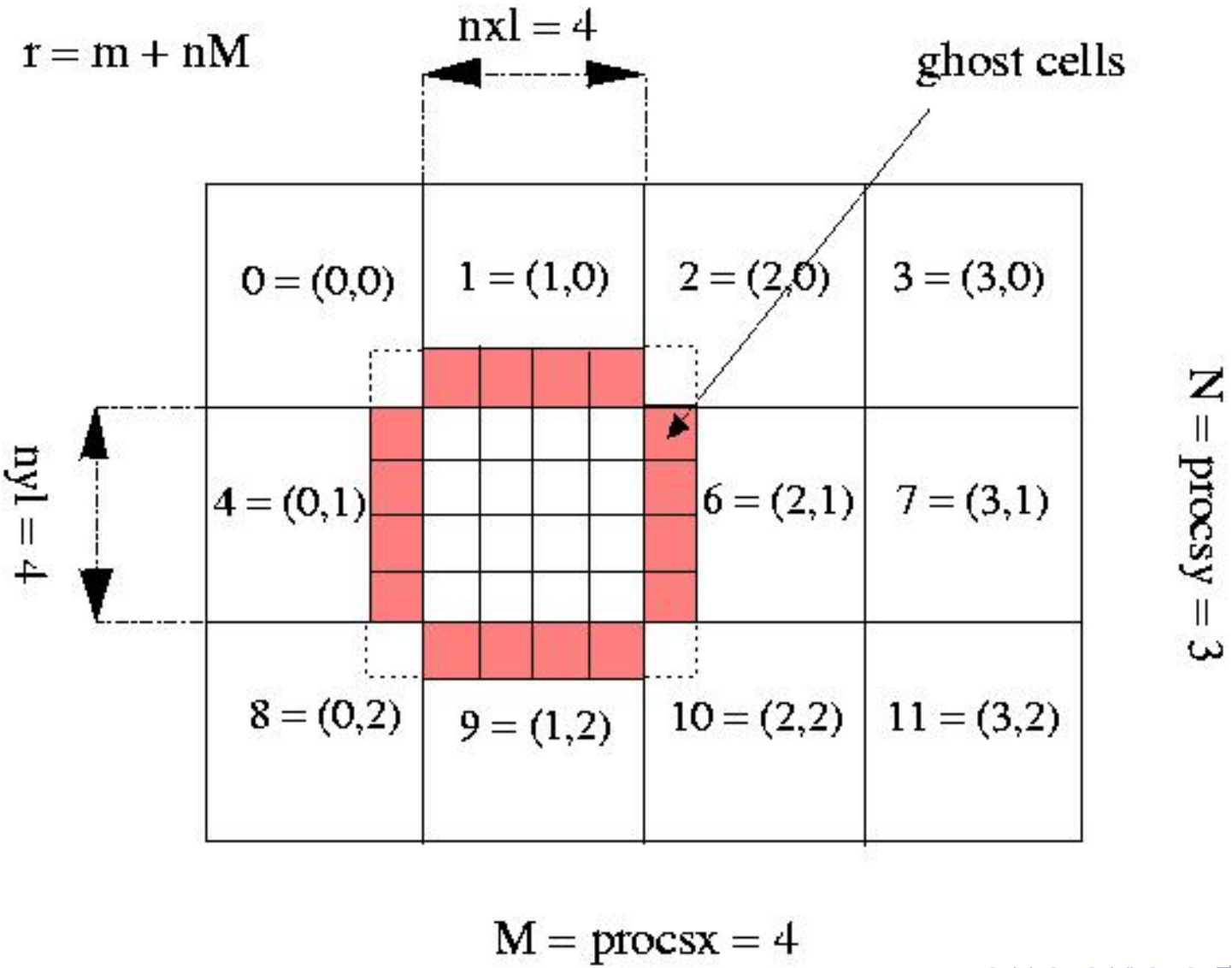
Synchronizacja wszystkich procesów z komunikatora

MPI_BARRIER(comm)

Diffusion2d - sieć procesów

- procesy zorganizowane są w prostokątną, cykliczną sieć M$\times$N
- dziedzina funkcji jest podzielona na kawałki (rozmiaru nxl$\times$nyl)
- każdy proces ma jeden swój kawałek dziedziny, za którą jest odpowiedzialny
- wartość funkcji zależy od poprzednich wartości sąsiadujących z nim pól
- każdy proces posiada ghost-cells – kopię (wziętą od sąsiadów) wartości funkcji *u* dla argumentów przyległych do jego kawałka dziedziny;
- z pewną częstotliwością wyniki obliczeń muszą być zasejwowane globalnie wiersz po wierszu

Geometria sieci



Diffusion2d – deklaracja zmiennych

```
/* wymiary sieci,
   czestotliwosc zapisu, ilosc iteracji */
int nprocsx, nprocsy, nxl, nyl, nprint, nsteps;

/* id procesu, jego rzutowanie na wspolrzedne */
int myproc, mycoord0, mycoord1;

/* identyfikatory sasiadow */
int upperNabe, lowerNabe, leftNabe, rightNabe;

int[,] u; /* funkcja u */

/* bufory do komunikacji*/
int[] send_buf, recv_buf, buf;
```

Diffusion2d – main() [1/3]

```
void main() {
  int iter = 0;
  //read nprocsx, nprocsy, nxl, nyl, nprint, nsteps
  MPI_Init();
  myproc = MPI_Comm_Rank();
  mycoord0 = myproc % nprocsx;
  mycoord1 = myproc / nprocsx;
```

Diffusion2d – main() [2/3]

```
upperNabe = mycoord0 + nprocsx*((mycoord1 + 1)%nprocsy);
lowerNabe = mycoord0 + nprocsx*
                       ((mycoord1 + nprocsy - 1)%nprocsy);
leftNabe = (mycoord0 + nprocsx - 1)%nprocsx +
            nprocsx*mycoord1;
rightNabe = (mycoord0 + 1)%nprocsx + nprocsx*mycoord1;

send_buf = new int[nxl];
recv_buf = new int[nxl];
buf = new int[nxl];

u = new int[nxl+2, nyl+2];
```

Diffusion2d – main() [3/3]

```
  setInitialValues();
  exchangeGhostCells();
  MPI_Barrier();

  for (iter = 1; iter <= nsteps; ++iter) {
    update(u);
    exchangeGhostCells();
    if ((iter % nprint) == 0) grid_write();
  }

  MPI_Barrier();
  MPI_Finalize();

} /* koniec main-a*/
```

Diffusion2d – exchangeGhostCells()

```
void exchangeGhostCells() {
 for(int i = 1;i <= nxl; ++i) send_buf[i-1] = u[i,1];
 MPI_Sendrecv(send_buf,lowerNabe, recv_buf,upperNabe);
 for(int i = 1;i <= nxl; ++i)u[i,nyl+1] = recv_buf[i-1];
 for(int i = 1; i <= nxl; ++i) send_buf[i-1] = u[i,nyl];
 MPI_Sendrecv(send_buf, upperNabe, recv_buf, lowerNabe);
 for(int i = 1;i <= nxl; ++i) u[i,0] = recv_buf[i-1];
 for(int j = 1;j <= nyl; ++j) send_buf[j-1] = u[1,j];
 MPI_Sendrecv(send_buf, leftNabe, recv_buf, rightNabe);
 for(int j = 1; j <= nyl; ++j)u[nxl+1,j]=recv_buf[j-1];
 for(int j = 1; j <= nyl; ++j) send_buf[j-1] = u[nxl,j];
 MPI_Sendrecv(send_buf, rightNabe, recv_buf, leftNabe);
 for(int j = 1; j <= nyl; ++j) u[0,j] = recv_buf[j-1];
}
```

Diffusion2d – grid_write()

```
void grid_write() {
  if (myproc != 0) {
   for (int n = 0; n < nprocsy; ++n)
    if (mycoord1 == n)
     for (int j = 1; j <= nyl; ++j)
      for (m = 0; m < nprocsx; ++m)
       if (mycoord0 == m) MPI_Send(u[1..nxl,j],0);
  } else {
   for (n = 0; n < nprocsy; ++n)
    for (int j = 1; j <= nyl; ++j)
     for (int m = 0; m < nprocsx; ++m) {
      int from_proc = m + nprocsx*n;
      if (from_proc != 0) MPI_Recv(buf, from_proc);
      else for(int i = 0; i < nxl; ++i) buf[i]=u[i+1,j];
      disk_write(buf);
     }
  }
  MPI_Barrier();
```

Własności

- DeadlockFree - program nigdy się nie zablokuje
- GlobalLockstep - proces może zacząć n-ty krok, dopiero gdy wszystkie pozostałe procesy obliczą już wartości funkcji w (n - 1) szym kroku
- LocalLockstep - proces może zacząć n-ty krok, dopiero gdy jego wszyscy sąsiedzi obliczą już wartość funkcji w (n - 1) szym kroku

Dla poprawności programu wystarczy by zachodziły DeadlockFree i LocalLockstep.

Pytania

Q1: czy można usunąć z kodu instrukcje MPI_Barier ?

Q2: czy zapisy na dysk są niezależne od przeplotu instrukcji oraz od sposobu buforowania komunikatów przez implementację MPI?

Przesyłanie komunikatów

Modelowanie komunikacji

- użyjemy chan_i_j dla wysyłania wiadomości od procesu i-tego do j-tego
- ograniczymy wielkość kanału przez chan_size

Uzasadnienie

- program nie korzysta ze wszystkich możliwości MPI takich jak np. MPI_ANY_SOURCE i MPI_ANY_TAG
- MPI nie narzuca ograniczenia na wielkość buforów komunikacyjnych, ale w pewnych przypadkach możemy uzasadnić konkretny wybór chan_size

Modelowanie komunikacji

Modele komunikacji

- prosty — gdy chan_size = 0
- złożony — gdy chan_size > 0

Prosty model komunikacyjny:

```
/* chan_size = 0 */
inline MPI_Send(schan, msg) { schan!msg }
inline MPI_Recv(rchan, rbuf) { rchan?rbuf }
```

Modelowanie komunikacji asynchronicznej

- w SPIN-ie kanały przechowują wiadomości w kolejce FIFO
- standard MPI pozwala swym implentacjom w dowolnej chwili zsynchronizować komunikujące się strony

Rozwiązanie

- `MPI_Recv` pozostawiamy więc bez zmian
- w `MPI_Send` dodajemy instrukcję, która może zablokować proces, aż kanał będzie pusty

```
inline MPI_Send(schan, msg) { /* chan_size > 0 */
  schan!msg; if :: 1 -> empty(schan) :: 1 fi }
```

Modelowanie jednoczesnego wysyłania i odbierania

chan_size > 0

```
inline MPI_Sendrecv(schan, msg, rchan, rbuf) {
  if :: schan!msg -> rchan?rbuf
     :: rchan?rbuf -> schan!msg fi;
  if :: 1 -> empty(schan) :: 1 fi }
```

chan_size = 0

- trzeba dodać obsługę przypadku wysyłania do siebie samego

```
inline MPI_Sendrecv(schan, msg, rchan, rbuf) {
  if :: (schan == rchan) -> rbuf = msg :: else ->
    if :: schan!msg -> rchan?rbuf
       :: rchan?rbuf -> schan!msg fi
  fi }
```

Modelowanie synchronizacji

MPI_BARRIER

- wprowadzamy dodatkowy proces i dodatkowy kanal synchroniczny
- zwykłe proces, by wejść wysyła kanałem 0, by wyjść wysyła 1
- dodatkowy proces zbiera od zwykłych uczestników zera, a potem jedynki

```
chan barrier_chan = [0] of {bit};
inline MPI_Barrier() { barrier_chan!0; barrier_chan!1 }
active prototype Barrier() {
end_b: do :: barrier_chan?0; ...; barrier_chan?0;
             barrier_chan?1; ...; barrier_chan?1 od }
```

Etykieta end_b - gdy pozostałe procesy się zakończą, SPIN nie uzna, że barrier się zablokował.

Modelowanie Diffusion2d

- nie będziemy modelować danych obliczanych przez program
- przesłanie wiadomości modelujemy jako wysłanie odpowiednim kanałem bitu 1
- będzie nas interesować tylko ile komunikatów zalega w konkretnym kanale

Modelowanie własności

- SPIN ma wbudowaną funkcjonalność sprawdzania deadlock-u
- pozostaje jeszcze wymodelować własności lockstep

Lockstep

far(i, j)

- predykat prawdziwy, gdy proces i chce wykonać update(u), a proces i-ty wyprzedza j-ty o więcej niż jeden krok
- dodajemy etykietę `Calculate` przed wywyłaniem update(u)
- wprowadzamy zmienne globalne `iter_i`, `iter_j`
- dodajemy

```
#define far_i_j
        (proc_i@Calculate && (iter_i - iter_j > 1))
```

Lockstep

Lockstep(i, j)

- żądanie by far(i, j) było zawsze fałszywe
- dodajemy never claim: `<>far_i_j`

$$GlobalLockstep = \bigwedge_{i \neq j} Lockstep(i,j)$$

$$LocalLockstep = \bigwedge_{i \neq j - sasiedzi} Lockstep(i,j)$$

Inne podejście do Lockstep

calc(i, n)

- predykat prawdziwy, gdy proces i chce wykonać update(u), a wrtością zmiennej `iter_i` jest n

```
#define calc_i_n (proc_i@Calculate && (iter_i == n))
```

Lockstep(i, j; n) - calc(i, n) jest poprzedzone przez calc(j, n - 1)

- never claim: `(!calc(j, n - 1)) U calc(i, n)`

$$Lockstep(i,j) = \bigwedge_{n} Lockstep(i,j;n)$$

Używamy SPIN-a

Opcje SPIN-a:

- `-DSAFETY` wszystkie własności dotyczą bezpieczeństwa
- `-DNOFAIR` uczciwość nie jest potrzebna
- `-DCOLLAPSE` dla lepszej kompresji

Przyjmujemy domyślnie

- `nxl = nyl = 1`
- `nprint = nsteps = 2`

Wyniki weryfikacji

DeadlockFree

- zachodzi
- szybko może zabraknąć pamięci - już przy konfiguracji 4×4, nawet gdy `chan_size = 0` (przy dostępnych 3GB)
- bariery 4×3, `chan_size = 0` (na procesorze 2.2GHz)
  - z barierami - 3.8×10^6 stanów, 153MB, 3 minuty
  - bez barier - 1.5×10^6 stanów, 60MB, troszkę ponad minutę
- prosta/złożona komunkacja dla 4×2, bez barier
  - `chan_size = 1` - 2.8×10^7 stanów
  - `chan_size = 0` - 9.6×10^4 stanów

Wyniki weryfikacji

LocalLockstep

- zachodzi
- nie zależy od modelu komunikacji

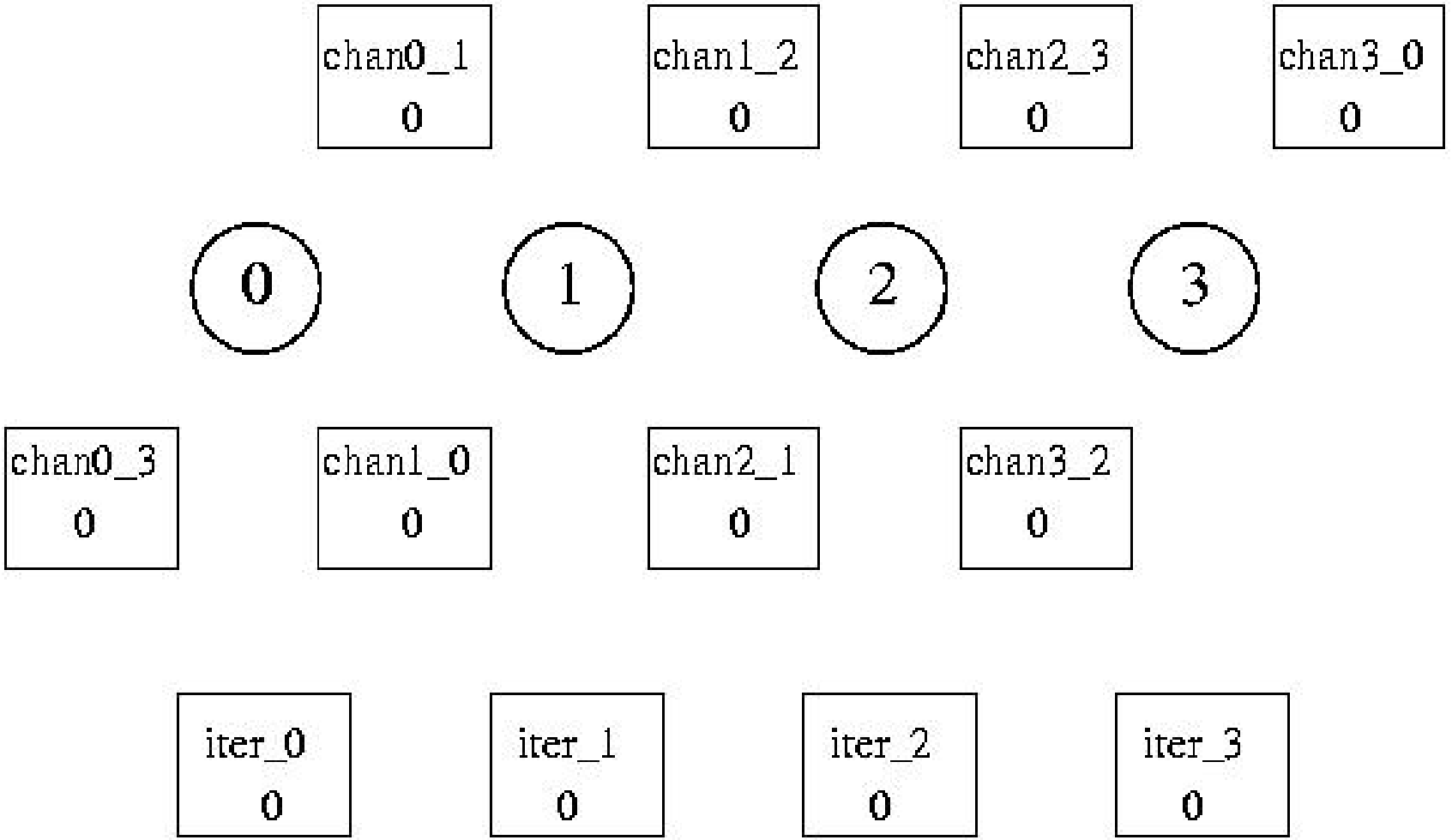
Wyniki weryfikacji

GlobalLockstep nie zachodzi!

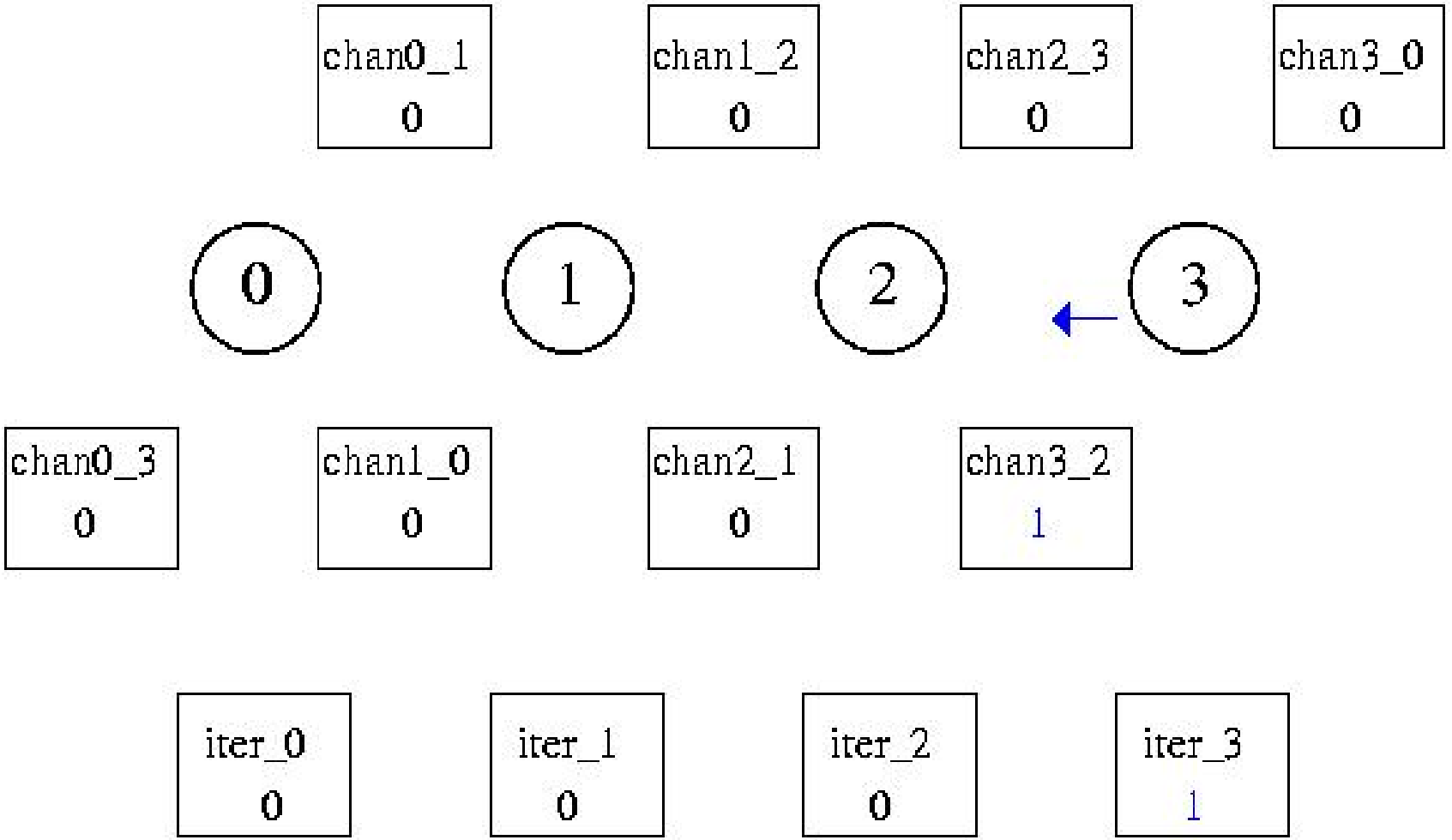
- $\texttt{nprocsx} \geq 4$ lub $\texttt{nprocsy} \geq 4$
- $\texttt{chan\_size} \geq 1$
- $\texttt{nsteps} \geq 2$

Przy takich warunkach, pewien proces może rozpocząć obliczanie u^2 podczas, gdy niektóre nie obliczyły jeszcze u^1.

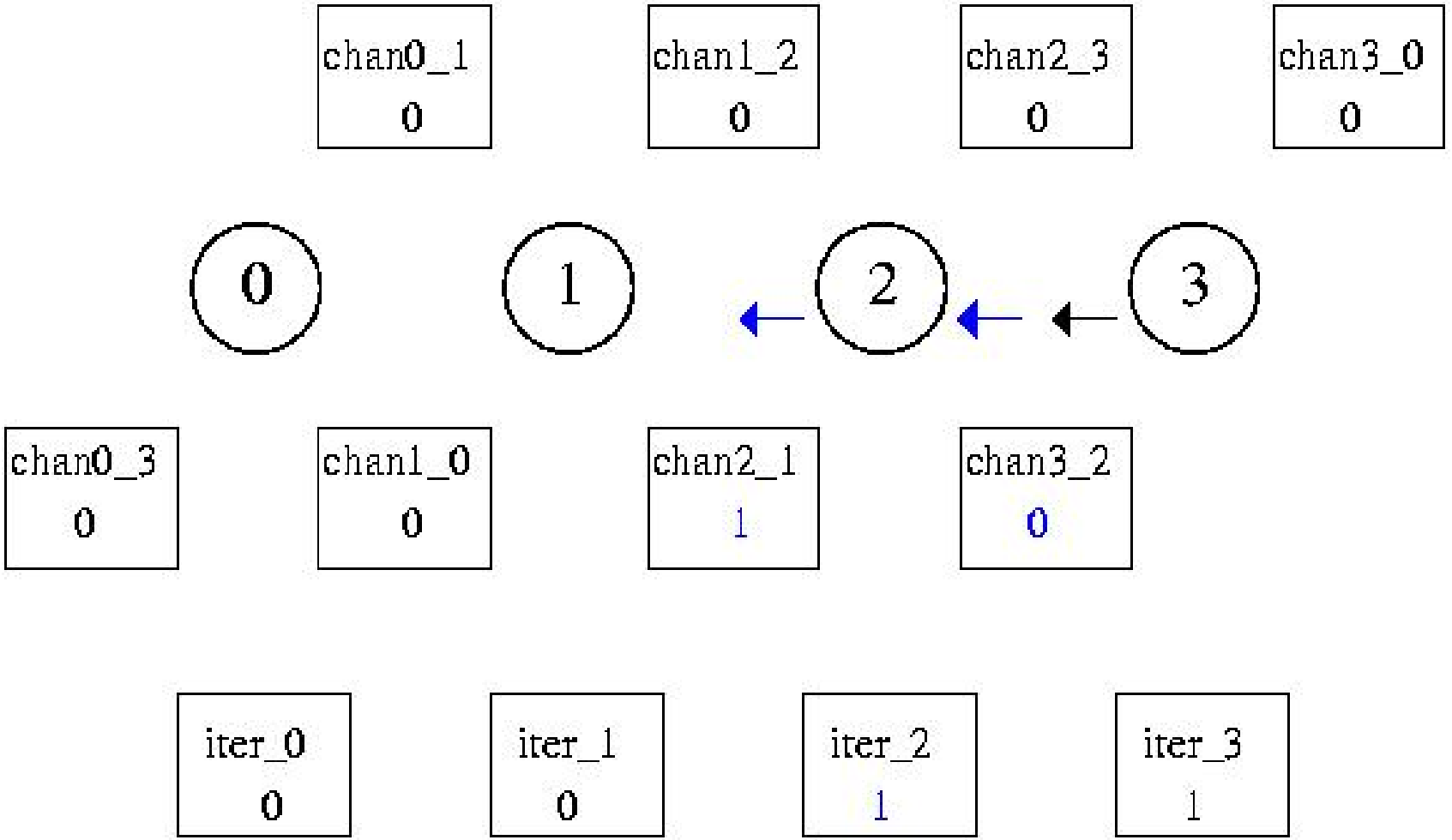
Kontrprzykład dla GlobalLockstep, dla 4×1



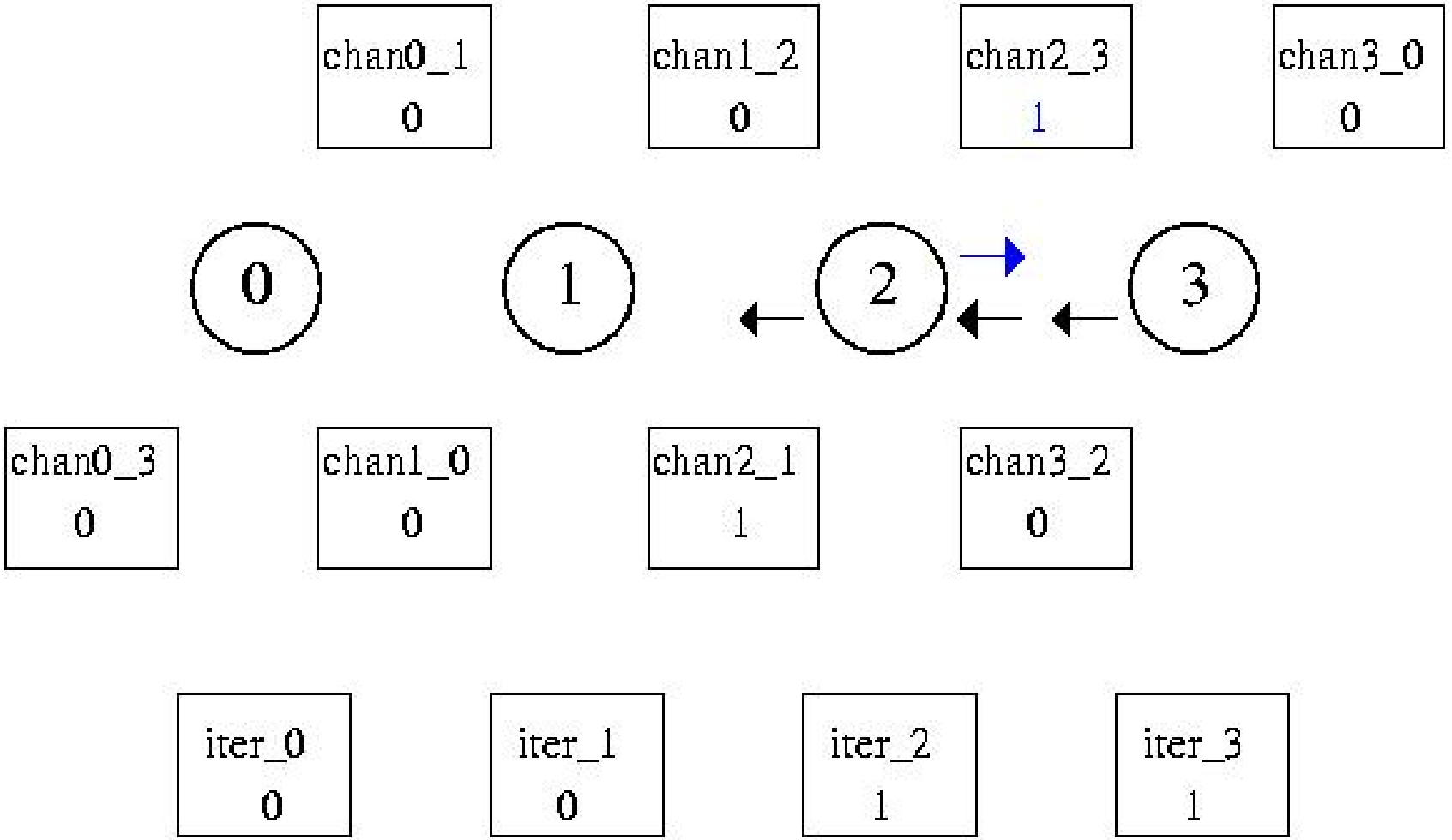
Kontrprzykład dla GlobalLockstep, dla 4×1



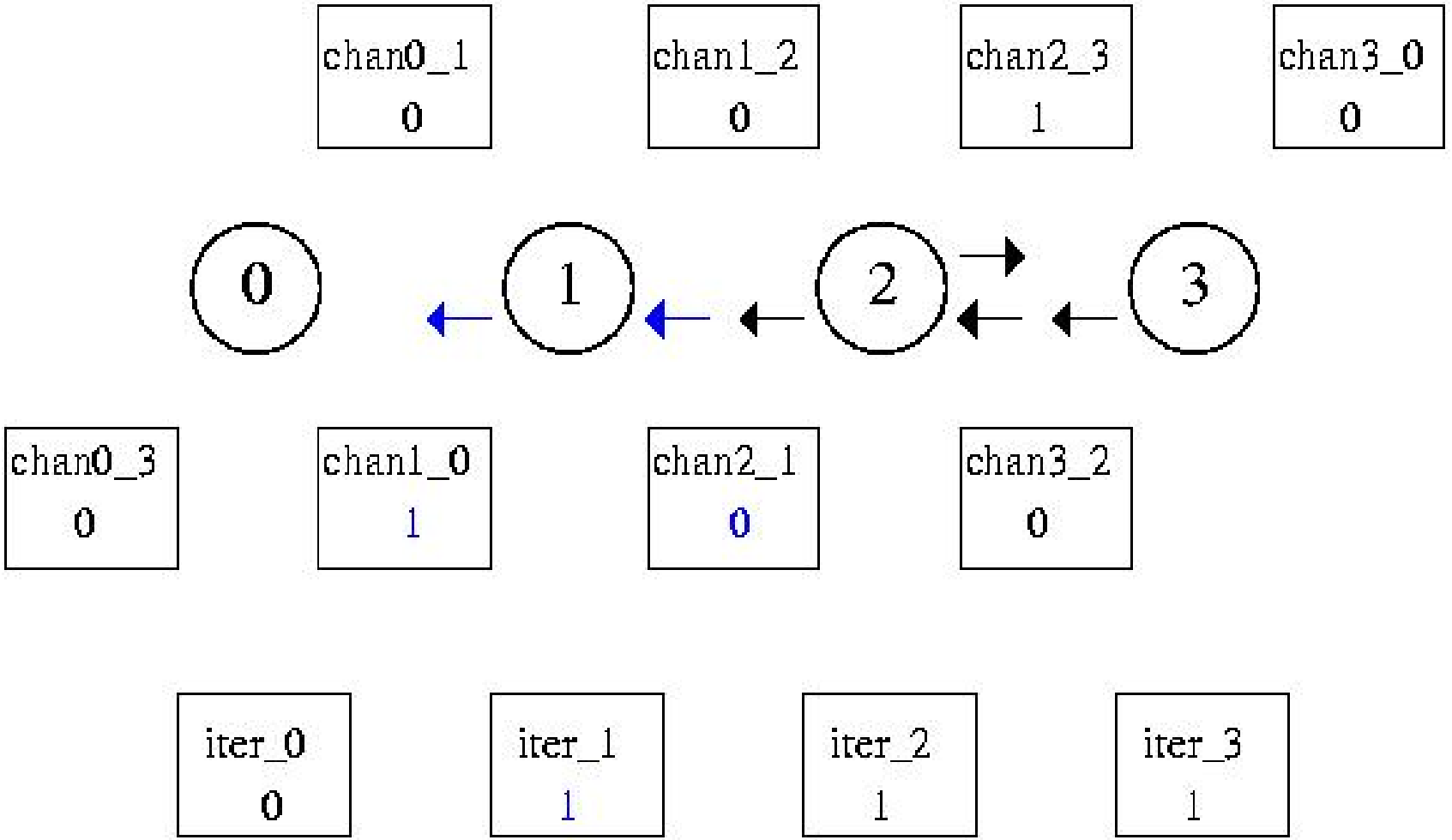
Kontrprzykład dla GlobalLockstep, dla 4×1



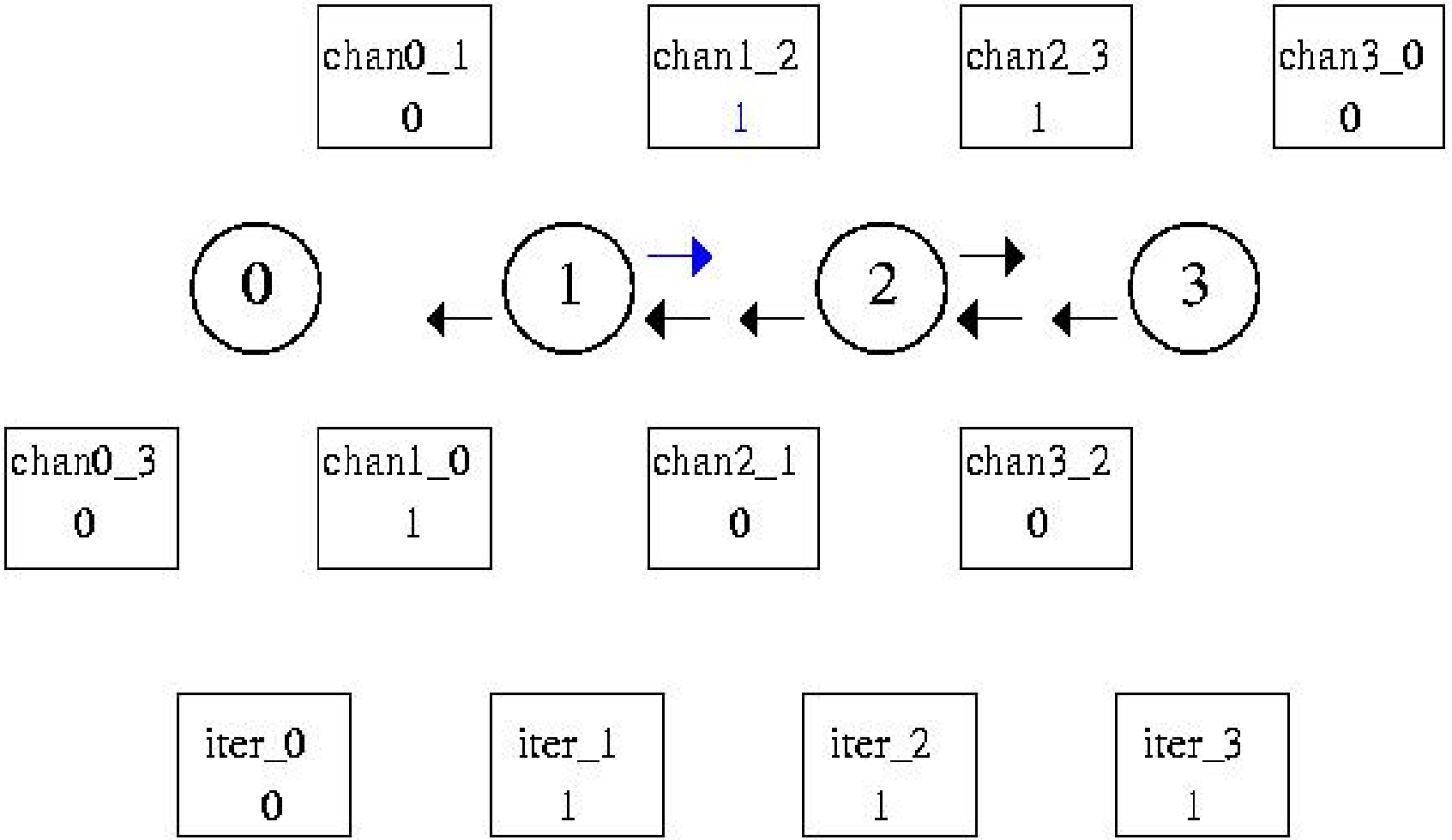
Kontrprzykład dla GlobalLockstep, dla 4×1



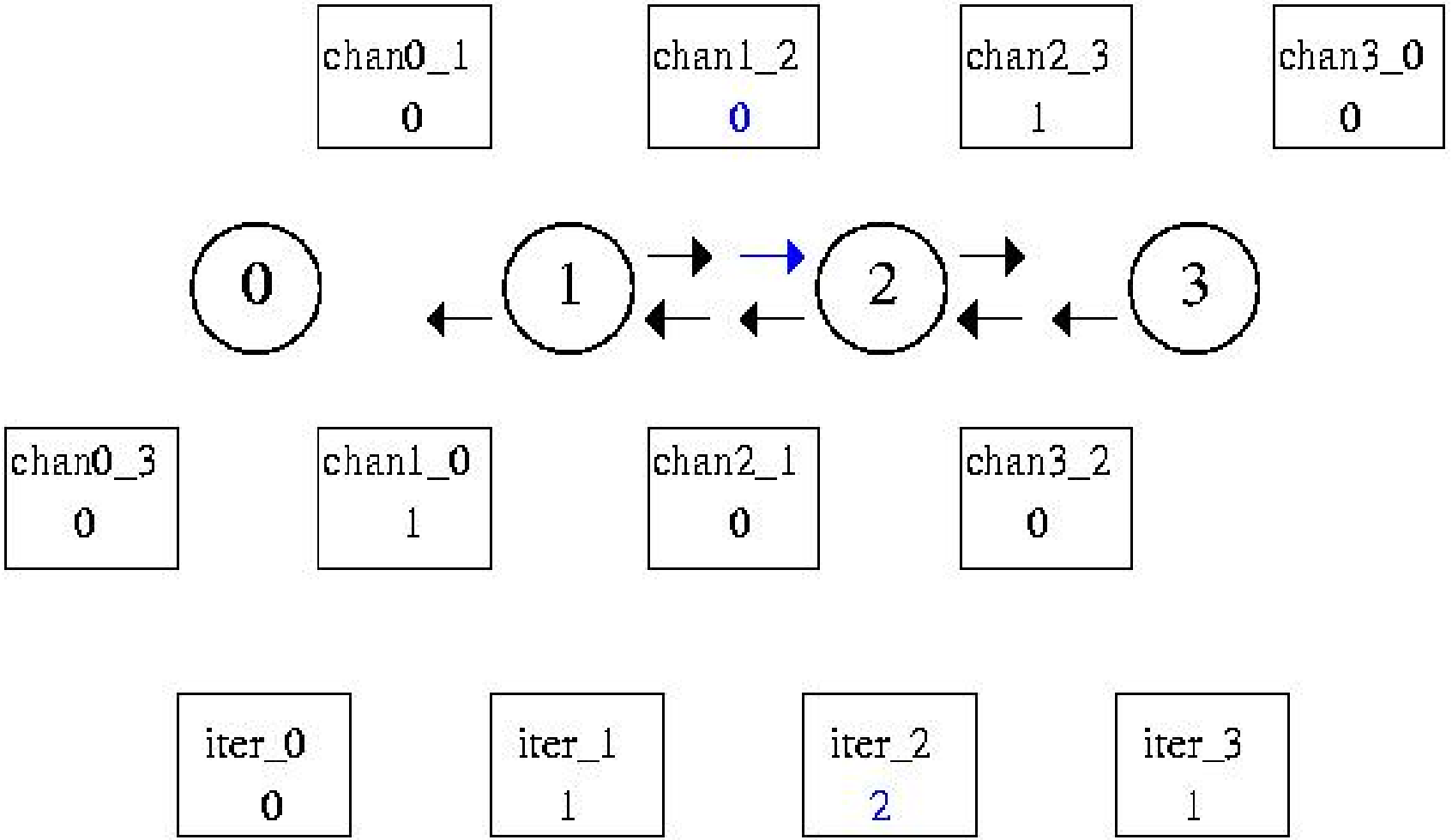
Kontrprzykład dla GlobalLockstep, dla 4×1



Kontrprzykład dla GlobalLockstep, dla 4×1



Kontrprzykład dla GlobalLockstep, dla 4×1



Wnioski z rozważań teoretycznych

- wystarczy więc sprawdzać deadlock-free dla `chan_size = 0`
- możemy więc usunąć bariery - nie wpływa to na bycie deadlock-free
- zapisy na dysk są poprawne

Literatura

- Stephen F. Siegel, George S. Avrunin

  Verification of MPI-Based Software for Scientific Computation

  http://laser.cs.umass.edu/∼siegel/projects
- Message-Passing Interface Standard 2.0
  http://www.mpi-forum.org/docs