

Припрема за лабораторијске вежбе из предмета Системска програмска подршка у реалном времену I

- 2018-2019/Вежба 5 -

Тема вежбе: Тutorials – Cilk примери

Садржај

1. Cilk синтакса и кључне речи: `cilk_spawn`, `cilk_sync`, `cilk_for`
2. Трка до података, редукујући хиперобјекти

Cilk plus

- Intel Cilk Plus је проширење језика C и C++
- Заснива се на језику Cilk који је развијен на MIT-у, и језику Cilk++ који је развила корпорација Cilk Arts
- Користи ефикасан распоређивач задатака са преузимањем задатака (енг. work stealing scheduler)
- Нуди хиперобјекте као механизам за решавање трке до података

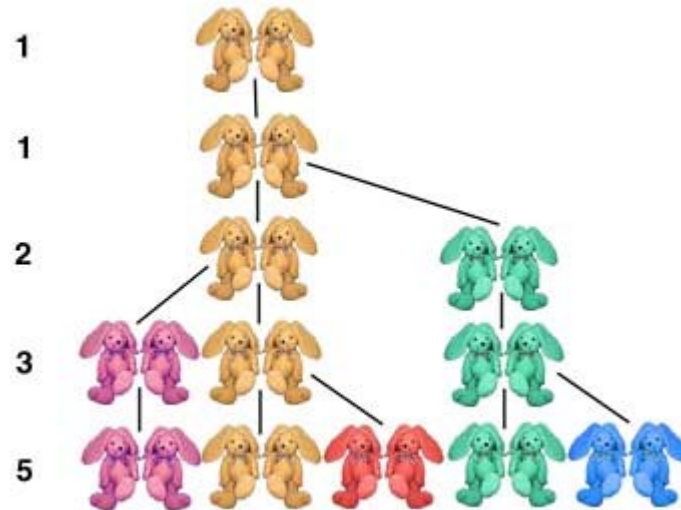
Пример 1: Фибоначијеви бројеви

- Фибоначијеви бројеви представљају низ бројева (0, 1, 1, 2, 3, 5, 8, 13, 21, 34...), у којем се сваки број добија као сума претходна два броја.

$$F_0 = 0,$$

$$F_1 = 1,$$

$$F_n = F_{n-1} + F_{n-2}, \text{ за } n > 1.$$



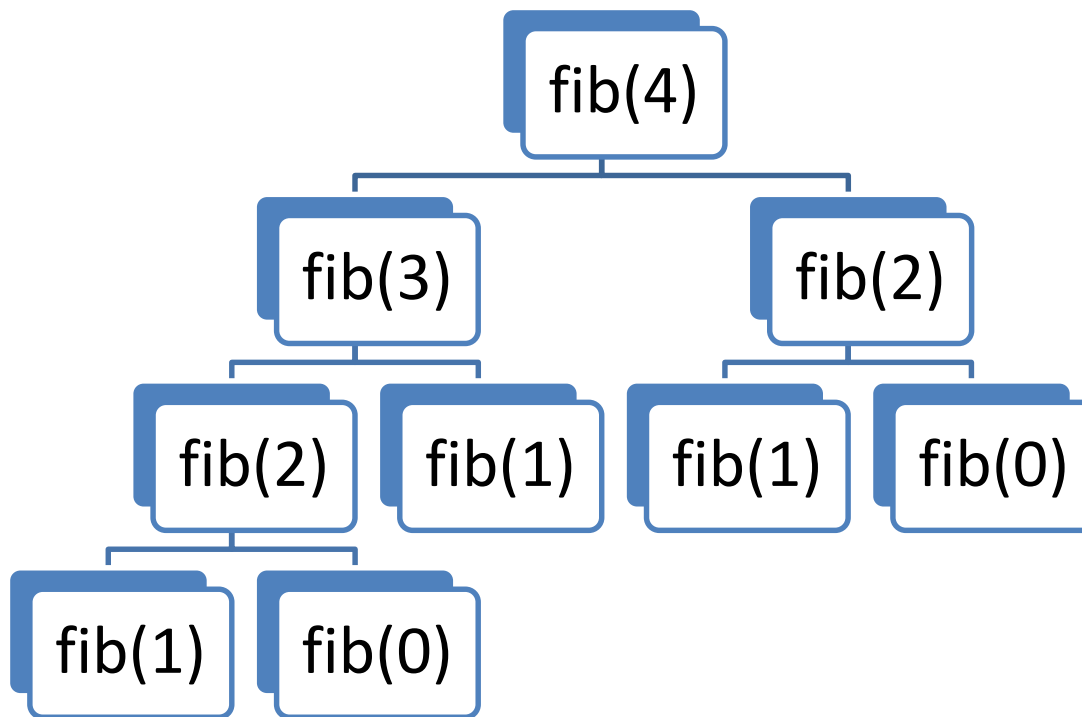
Програм за рачунање Фибоначијевих бројева

```
#include <stdio.h>
#include <stdlib.h>

int fib(int n)
{
    if (n < 2) return n;
    int x = fib(n - 1);
    int y = fib(n - 2);
    return x + y;
}

int main(int argc, char* argv[])
{
    int n = atoi(argv[1]);
    int result = fib(n);
    printf("Fibonacci of %d is %d.\n", n, result);
    return 0;
}
```

Извршавање програма за рачунање Фибоначијевих бројева



Кључ паралелизације:

Израчунавање `fib(n-1)` и `fib(n-2)` може да се извршава истовремено, без међусобног ометања (интерференције).

```
int fib(int n)
{
    if (n < 2) return n;
    int x = fib(n - 1);
    int y = fib(n - 2);

    return x + y;
}
```

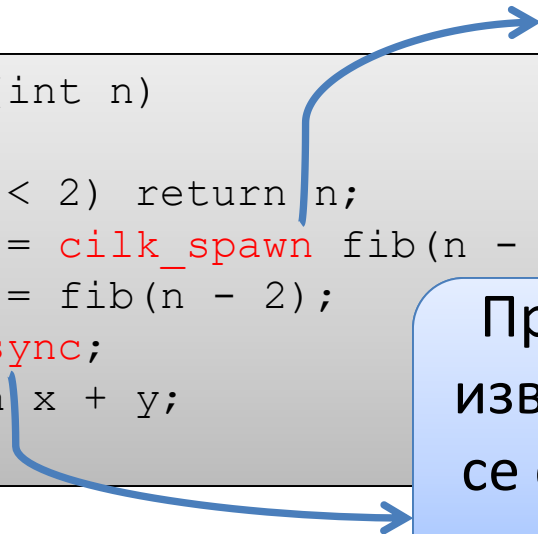
Серијска Фибоначи функција

```
int fib(int n)
{
    if (n < 2) return n;
    int x = fib(n - 1);
    int y = fib(n - 2);

    return x + y;
}
```


Паралелизам у Cilk Plus-у; кључне речи `cilk_spawn` и `cilk_sync`

```
int fib(int n)
{
    if (n < 2) return n;
    int x = cilk_spawn fib(n - 1);
    int y = fib(n - 2);
    cilk_sync;
    return x + y;
}
```



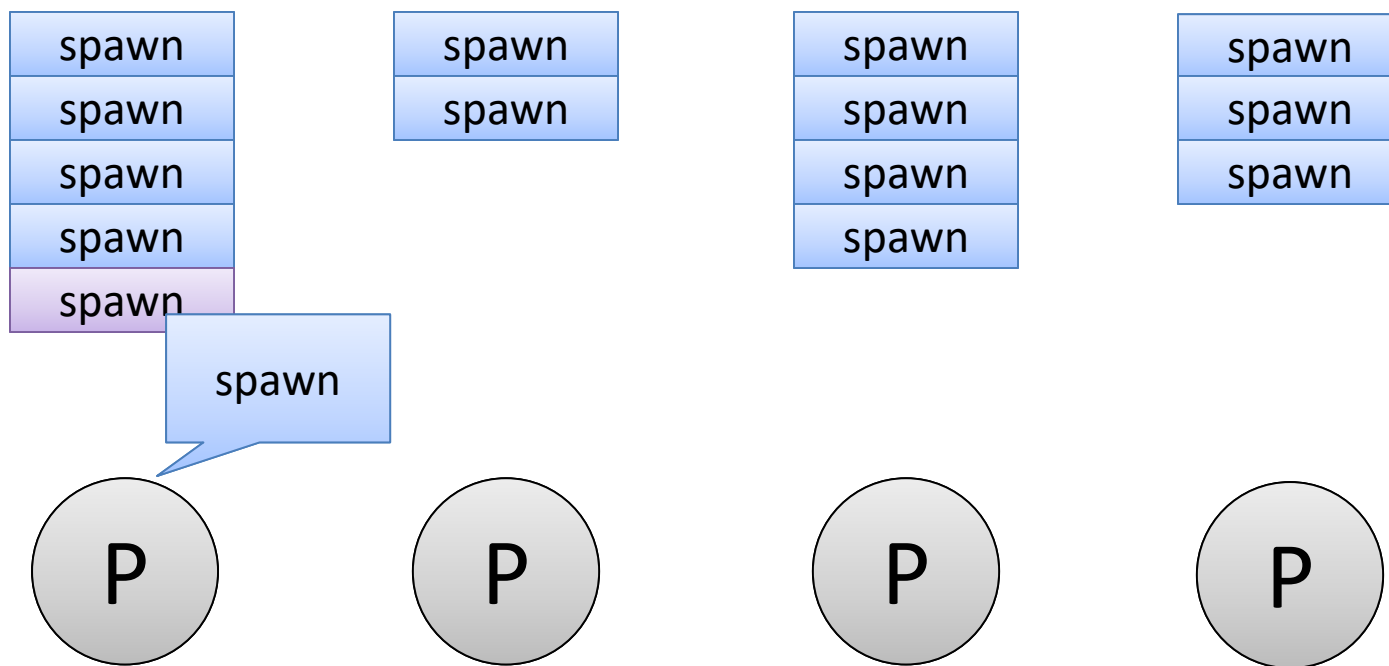
Функција **потомак** може да се извршава у паралели са **претком** – функцијом која је позива.

Предак не може да настави са извршавањем иза ове тачке док се сви измрешћени потомци не изврше.

Cilk кључне речи не **наређују** паралелно извршавање, него **га** омогућавају.

Распоређивач задатака са преузимањем задатака

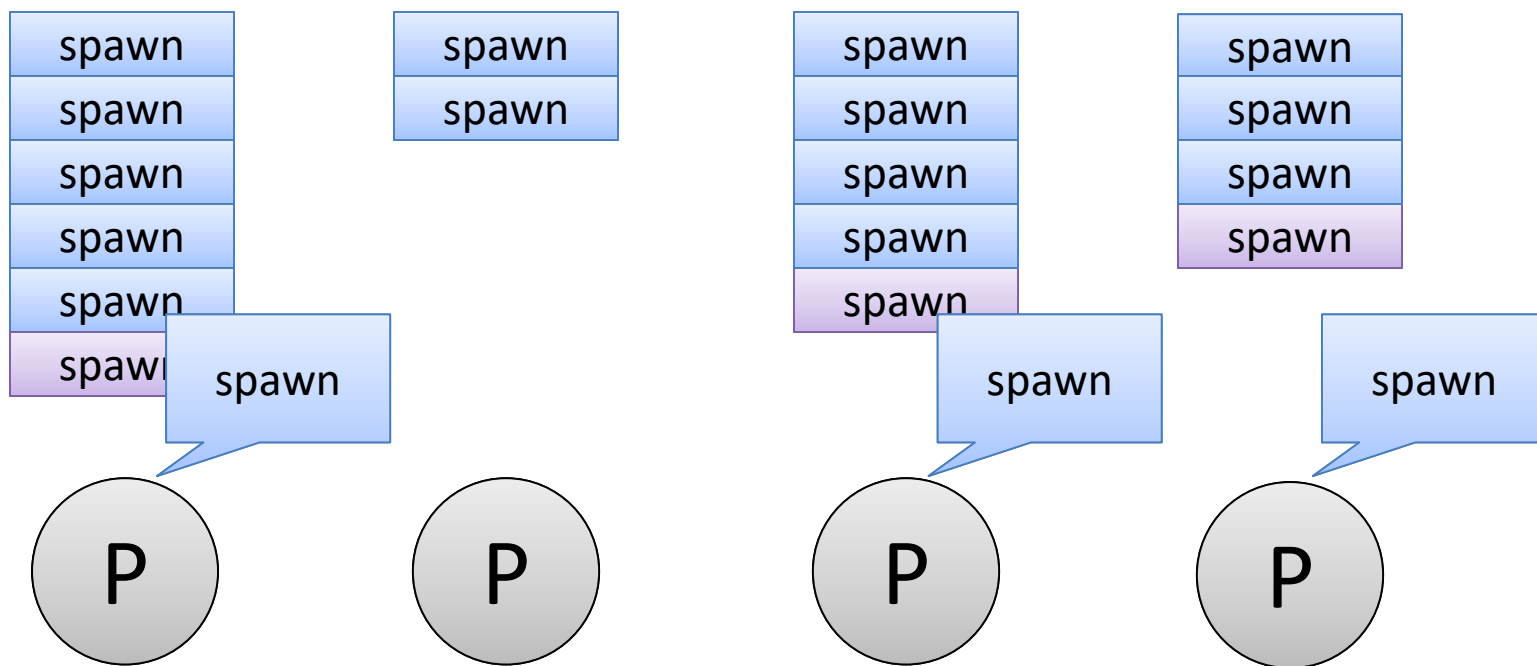
Сваки процесор поседује ред чекања за
измрешћене задатке:



Када сваки од процесора има посла, цена мрешћења
је приближно једнака цени позива функције.

Распоређивач задатака са преузимањем задатака

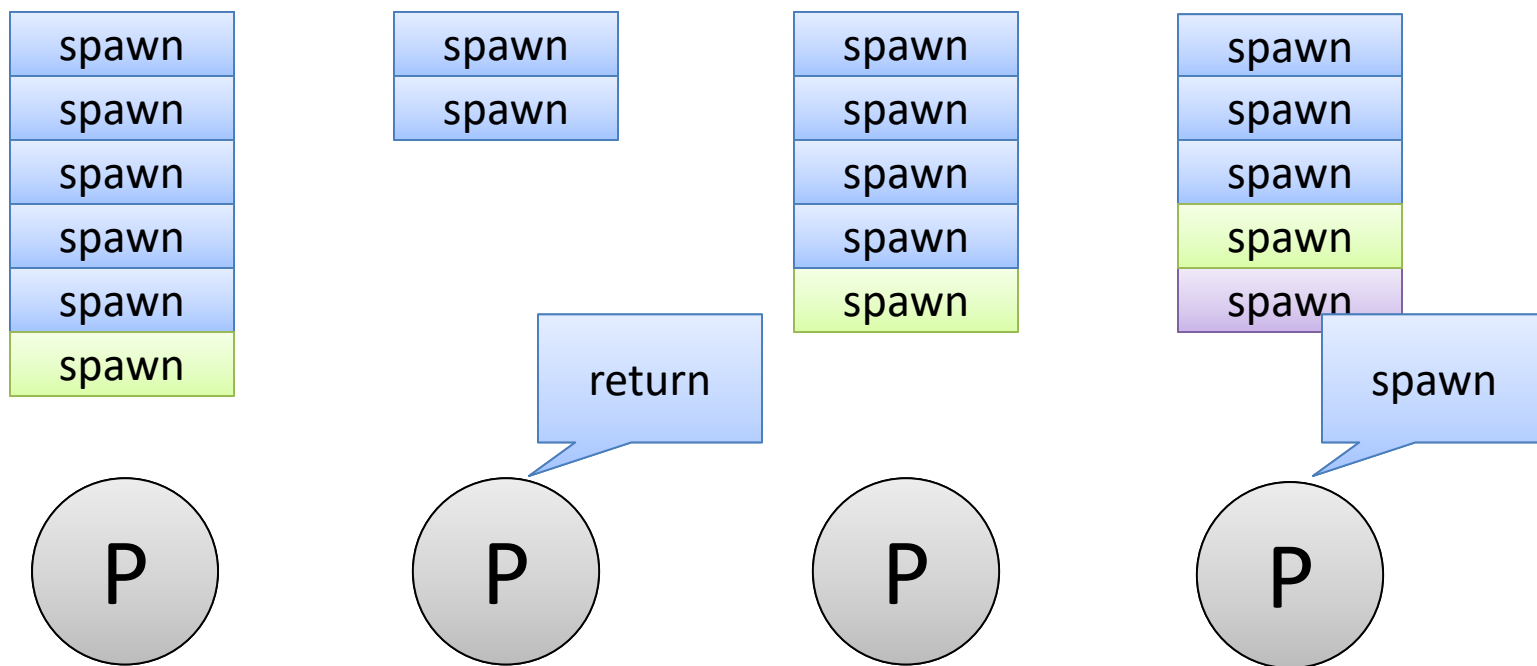
Сваки процесор поседује ред чекања за
измрешћене задатке:



Када сваки од процесора има посла, цена мрешћења
је приближно једнака цени позива функције.

Распоређивач задатака са преузимањем задатака

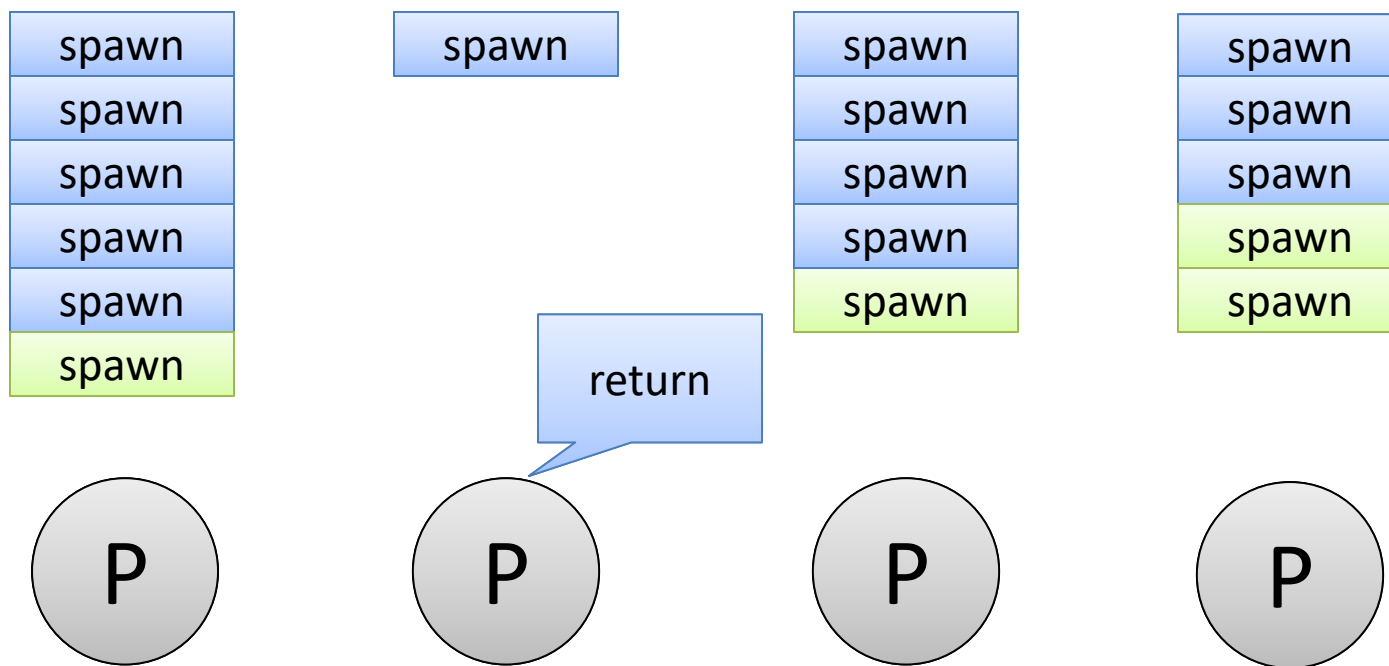
Сваки процесор поседује ред чекања за
измрешћене задатке:



Када сваки од процесора има посла, цена мрешћења
је приближно једнака цени позива функције.

Распоређивач задатака са преузимањем задатака

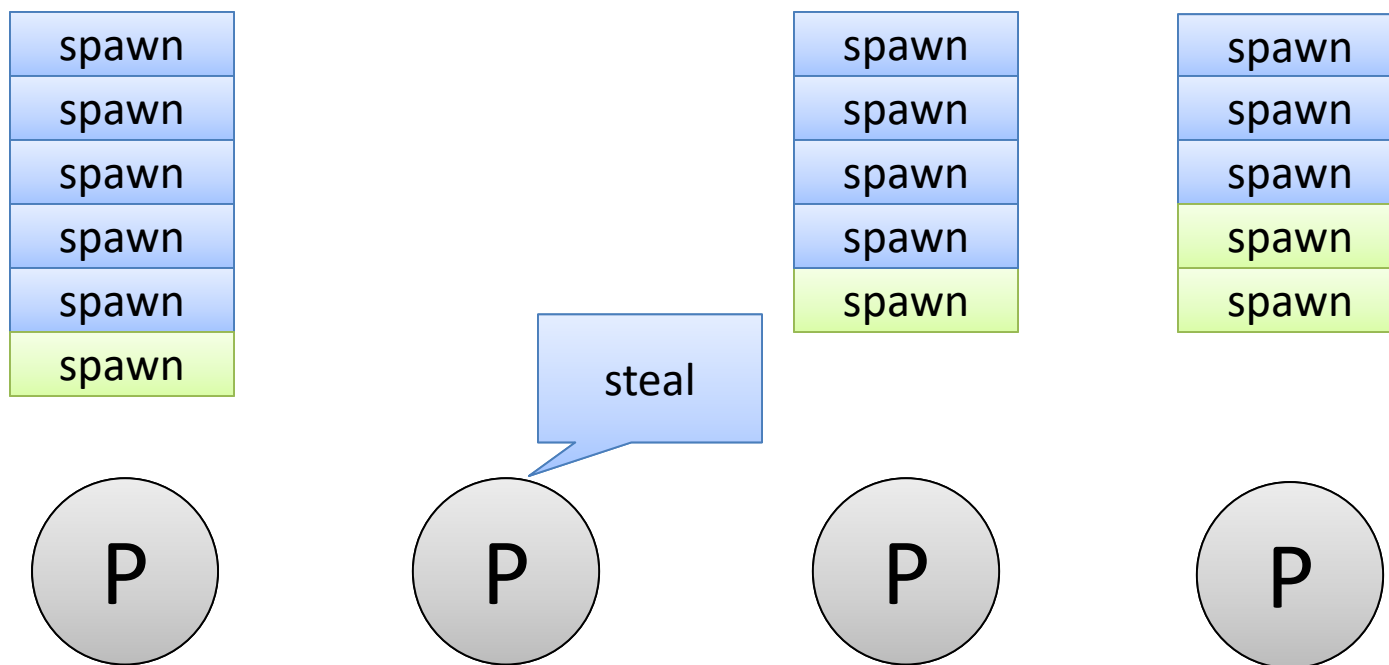
Сваки процесор поседује ред чекања за измрешћене задатке:



Када сваки од процесора има посла, цена мрешћења је приближно једнака цени позива функције.

Распоређивач задатака са преузимањем задатака

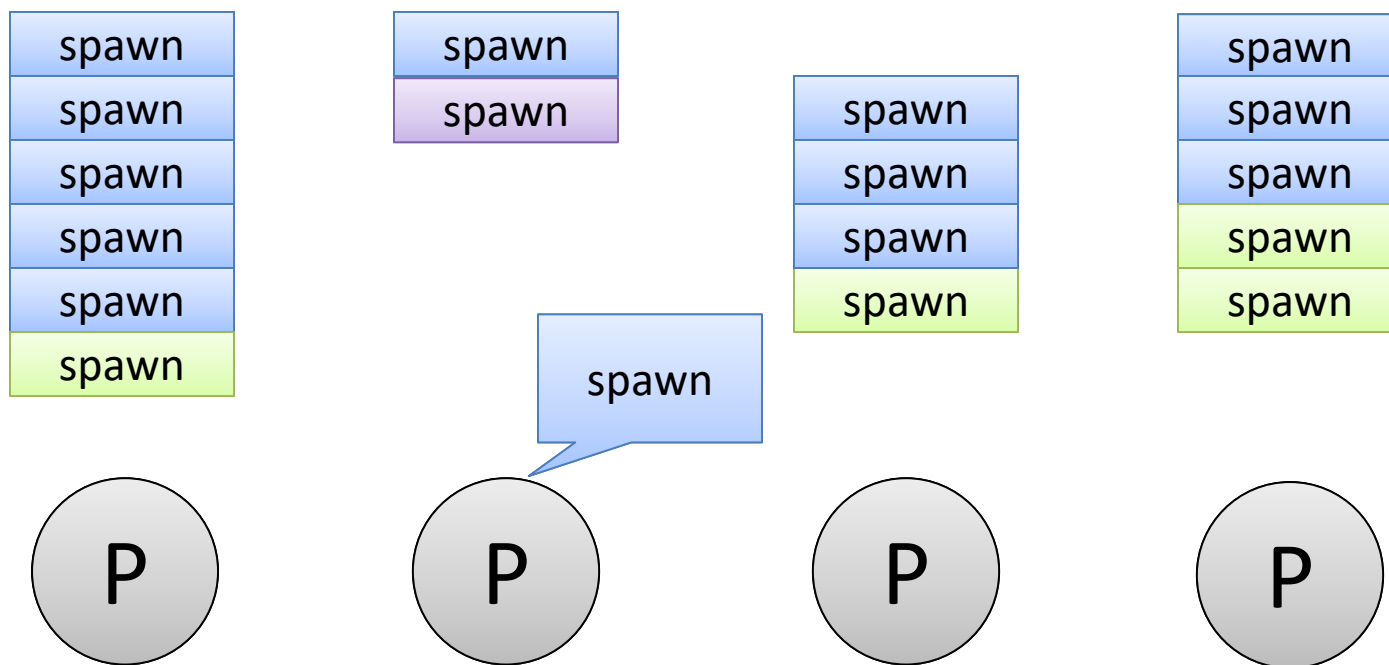
Сваки процесор поседује ред чекања за
измрешћене задатке:



Када процесор нема посла, он преузима задатке од
другог процесора.

Распоређивач задатака са преузимањем задатака

Сваки процесор поседује ред чекања за
измрешћене задатке:



Када је ниво паралелизма довољан, преузимање
задатака је ретко и убрзање је приближно линеарно.

Паралелизација петљи

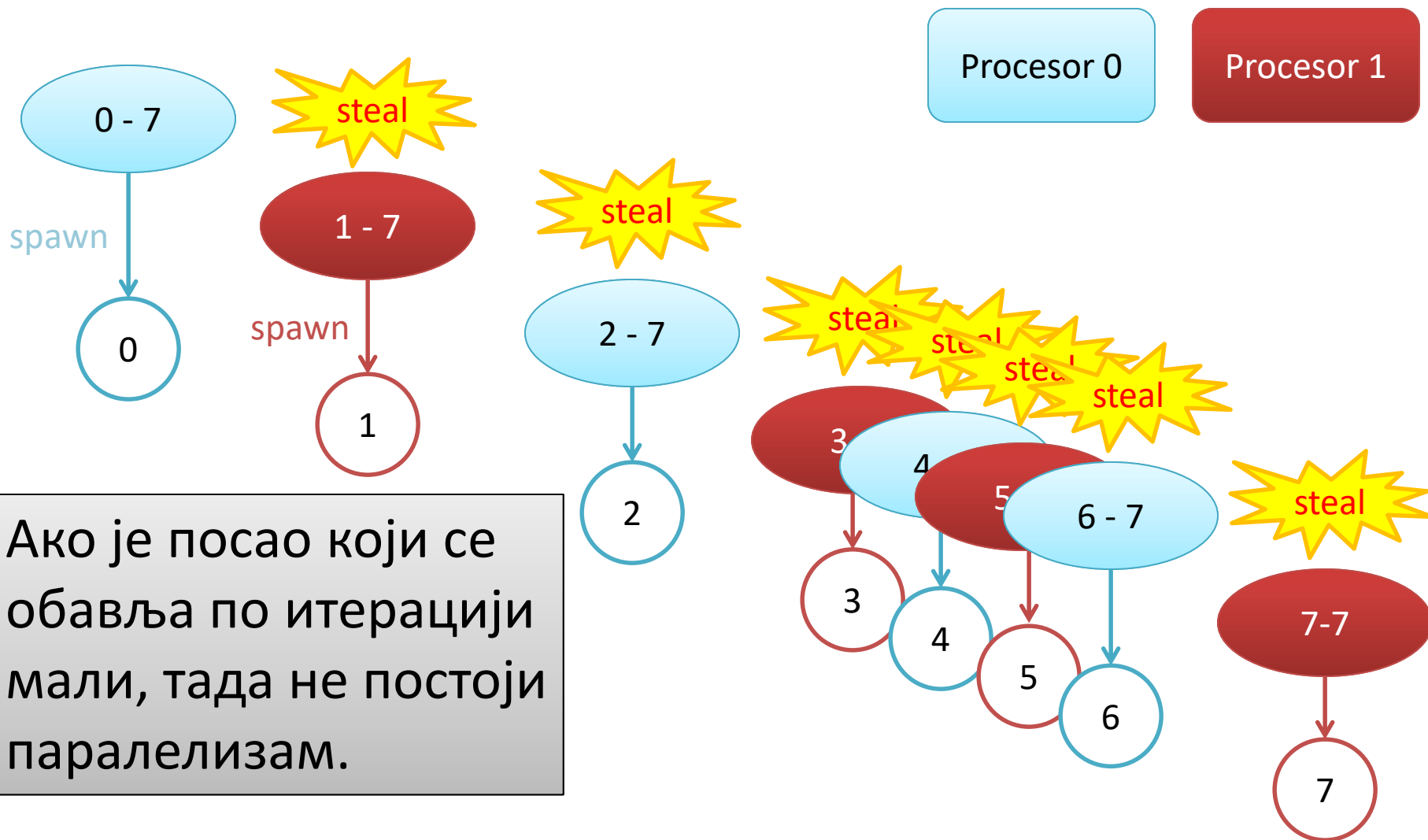
```
for (int i = 0; i < 8; i++)  
{  
    do_work(i);  
}
```



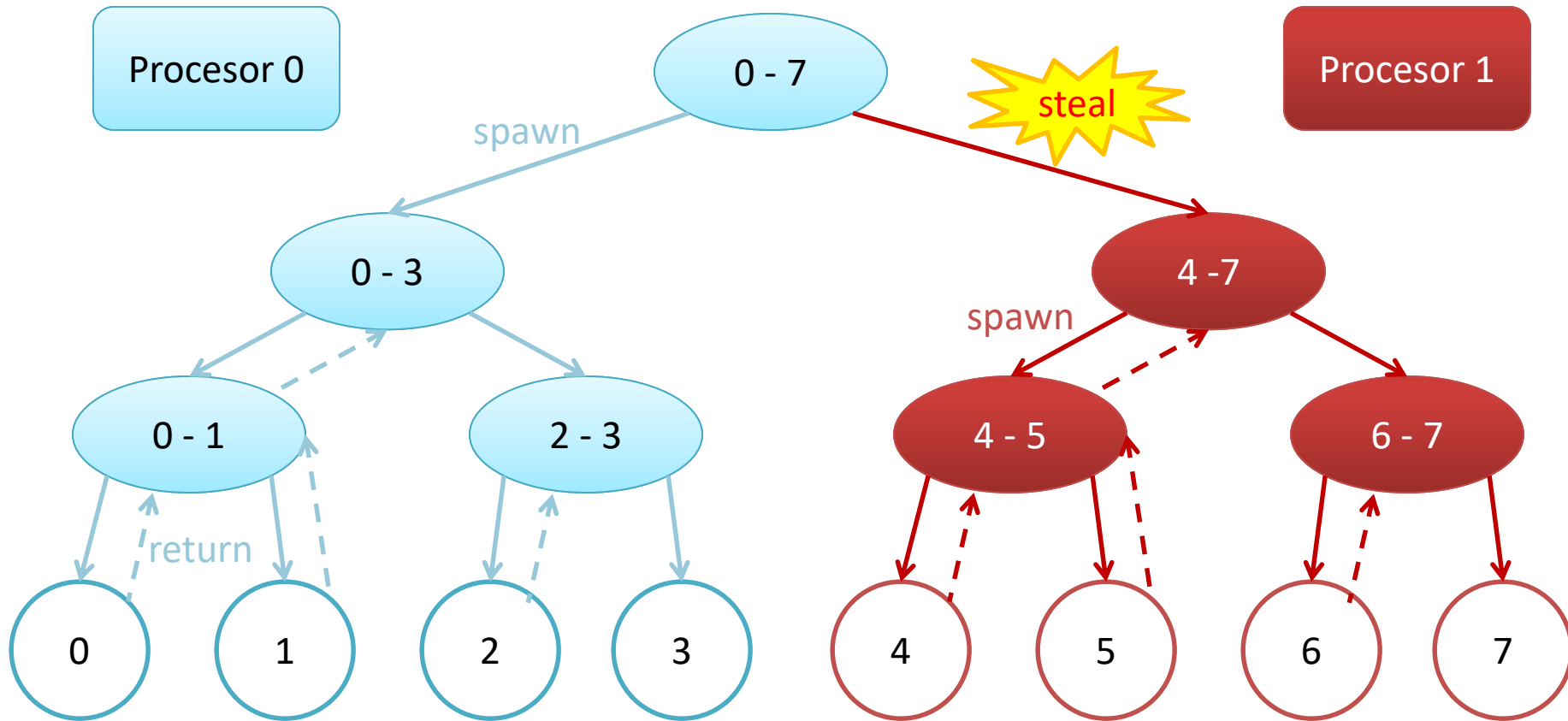
```
for (int i = 0; i < 8; i++)  
{  
    cilk_spawn do_work(i);  
}  
cilk_sync;
```

- Много додатног посла: мрешћење је јефтино, **преузимање задатака** је скупо.
- Низак ниво паралелизма: у сваком тренутку постоји само један произвођач паралелног посла.

Серијска *for* петља и *spawn*: небалансирано



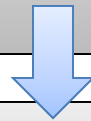
Алтернатива: подели и завладај



Подели и завладај резултује са мање преузимања задатака и вишим паралелизмом.

Кључна реч: `cilk_for`

```
for (int i = 0; i < 8; i++)  
{  
    do_work(i);  
}
```



```
cilk_for (int i = 0; i < 8; i++)  
{  
    do_work(i);  
}
```

cilk_for примењује подели и завладај над итерацијама.

Пример 2: Quicksort - секвенцијални

```
void quicksort (int arr[], int low, int high)
{
    int i = low; int j = high; int y = 0;

    int z = arr[(low + high) / 2]; // compare value

    do
    {
        while (arr[i] < z) i++; // find element above
        while (arr[j] > z) j--; // find element below

        if (i <= j) swap(arr, i, j); // swap two elements
    } while (i <= j);

    // recurse
    if (low < j) quicksort(arr, low, j);
    if (i < high) quicksort(arr, i, high);
}
```

Пример 2: Quicksort - паралелни

```
void quicksort (int arr[], int low, int high)
{
    int i = low; int j = high; int y = 0;

    int z = arr[(low + high) / 2]; // compare value

    do
    {
        while (arr[i] < z) i++; // find element above
        while (arr[j] > z) j--; // find element below

        if (i <= j) swap(arr, i, j); // swap two elements
    } while (i <= j);

    // recurse
    if (low < j) cilk_spawn quicksort(arr, low, j);
    if (i < high) quicksort(arr, i, high);
    cilk_sync;
}
```

Садржај

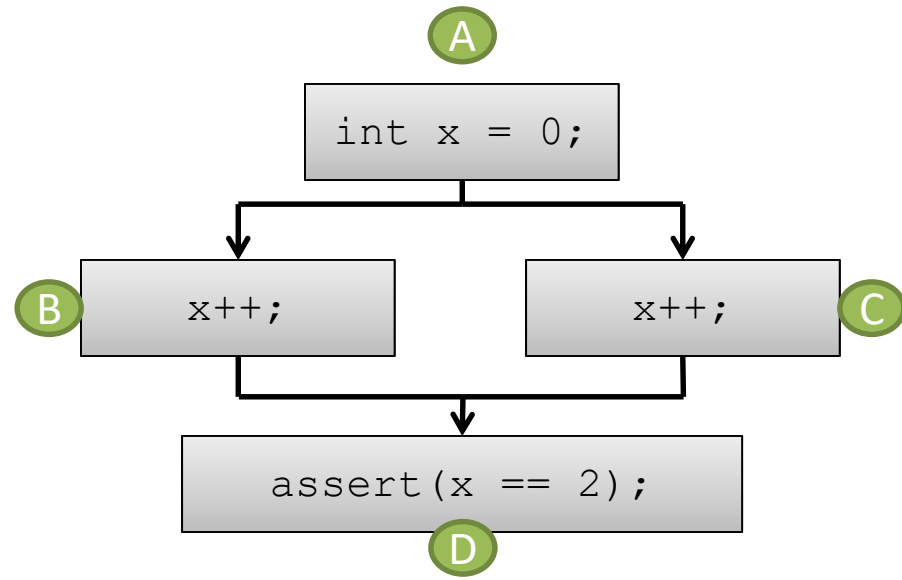
1. Cilk синтакса и кључне речи: `cilk_spawn`, `cilk_sync`, `cilk_for`
2. Трка до података, редукујући хиперобјекти

Трка до података

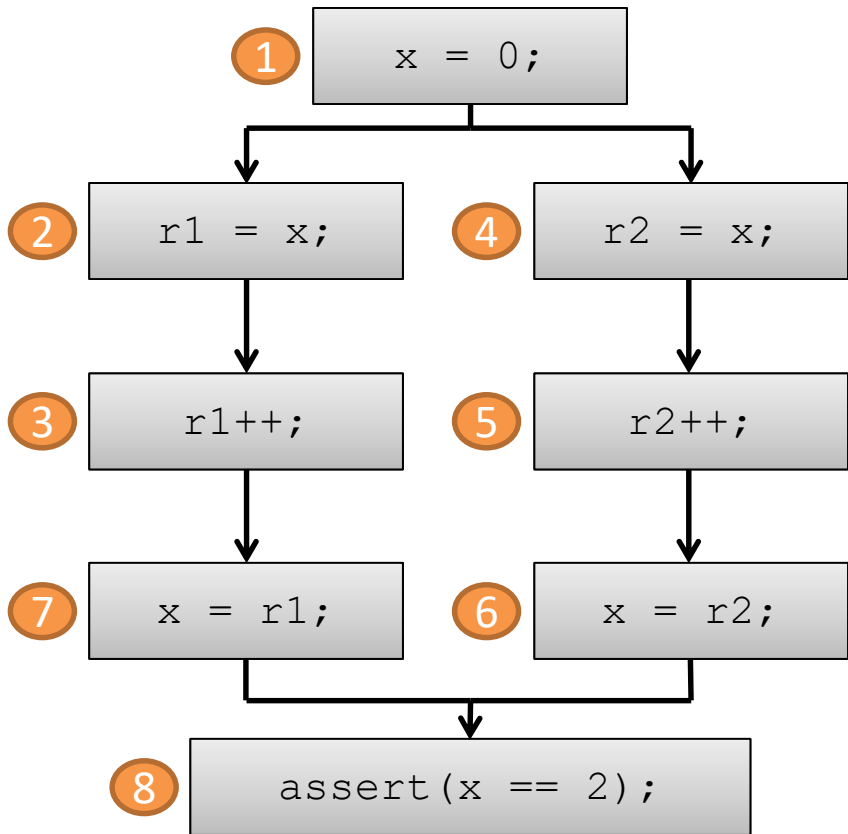
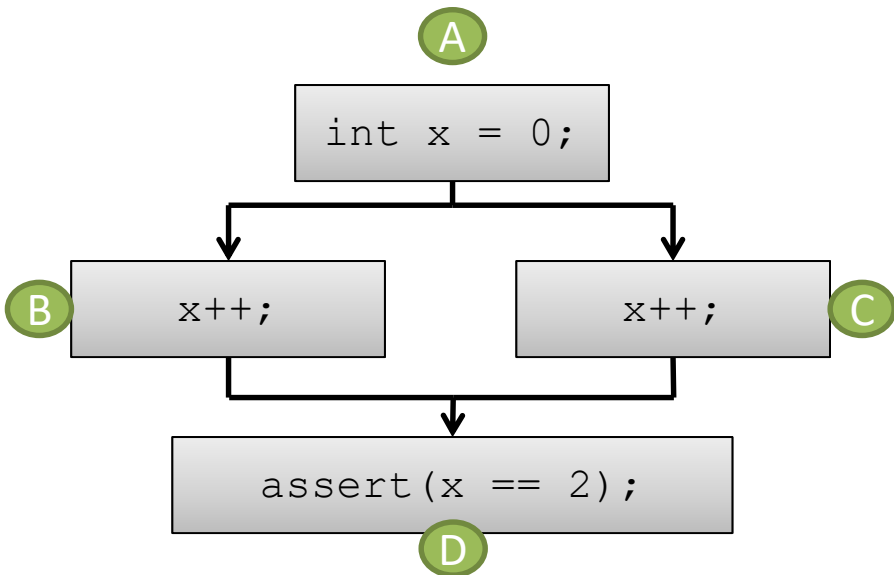
До трке до података долази када две логички паралелне инструкције приступају истој меморијској локацији и бар један од та два приступа је ради **писања**. Тада је вредност те меморијске локације неодређена. Глобалне променљиве су чест узрок трке до података.

Пример:

```
A int x = 0;  
  cilk_for (int i = 0; i < 2; i++)  
  {  
    B C x++;  
  }  
D assert(x == 2);
```



Ближи поглед



1
x

1
r1

1
r2

Избегавање трке до података

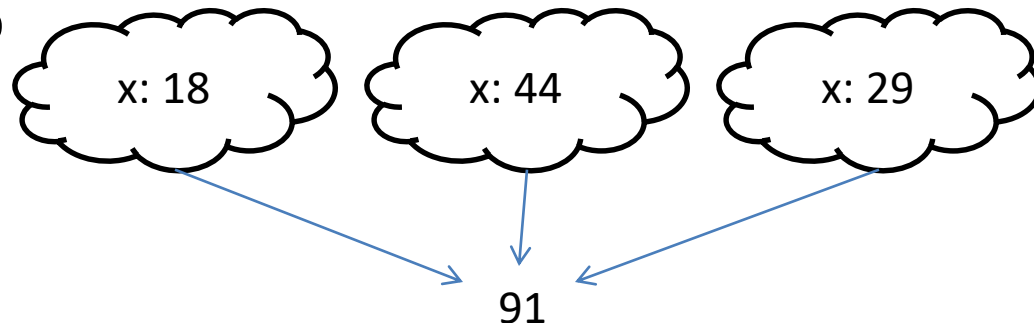
- Итерације петље `cilk_for` треба да буду независне.
- Између наредбе `cilk_spawn` и одговарајуће наредбе `cilk_sync`, програмски код измрешћеног потомка и код претка треба да буду независни.
(Аргументе функције која се мрести рачуна предак, пре него што дође до мрешћења.)
- Величина машинске речи је битна.
Обратити пажњу на трку до података упакованих у структуре:
освежавање `x.a` и `x.b` у паралели може да изазове трку, у зависности од оптимизације преводиоца. (Безбедно на `x86` и `x86_64`.)

```
struct  
{  
    char a;  
    char b;  
} x;
```

Редукујући хиперобјекти

- Променљива x може да се декларише као **редуктор** над **асоцијативном** операцијом попут сабирања, множења, логичког И, спајања листи итд.
- **Линије извршења** (енг. *strands*) могу да мењају вредност променљиве x , зато што хиперобјекти омогућавају сигуран приступ дељеним објектима дајући свакој паралелној линији посебну инстанцу – **поглед** (енг. *view*).
- Cilk Plus окружење координише *погледе* и комбинује их када је то пригодно.
- Када остане само једна инстанца објекта x , њена вредност је стабилна и може да се преузме.

Пример: сумирајући редуктор



Типови хиперобјеката

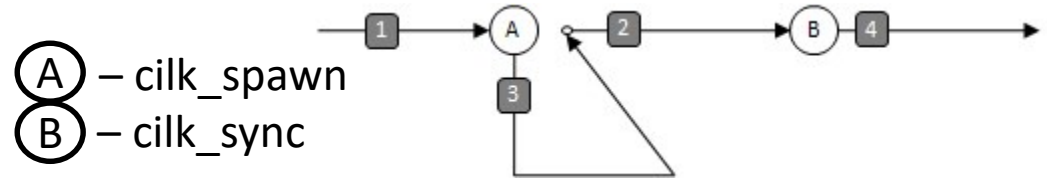
- Максимални (минимални) елемент у скупу и његов индекс: *reducer_max(min)* , *reducer_max_index(min)*.
- Сумирање: *reducer_opadd*
- Логичке и операције на нивоу бита: *reducer_opand*, *reducer_opor*, *reducer_opxor*
- Паралелни излазни ток: *reducer_ostream*
- Спајање стрингова: *reducer_basic_string*, *reducer_string* за тип *char*
- Спајање листи, додаванјем на крај и почетак, респективно: *reducer_list_append*, *reducer_list_prepend*

Вредност редуктора се преузима позивом функције *get_value()*.

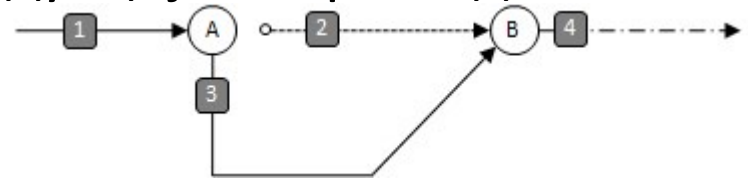
Препоручује се коришћење редуктора у језику C++ - редуктори у C-у су слабо документовани и стога тежи за употребу; такође, у програмском језику C не постоји преклапање оператора.

Како раде редуктори?

- Постоје два начина извршења наредбе **cilk_spawn** – са преузимањем или без преузимања задатака.
- Ако не дође до преузимања, редуктор се понаша као обична променљива.



- Ако дође до преузимања, линији извршавања која наставља иза наредбе **cilk_spawn** се додељује *поглед* на хиперобјекат, а потомак добија редуктор пошто се он извршава први после мрешћења. Приликом наредбе **cilk_sync**, вредност *погледа* се спаја са вредношћу редуктора помоћу операције редукције, а претходно створени *поглед* се уништава.



Пример 3: Налажење индекса најмањег елемента у низу

```
#include <cilk/cilk.h>
#include <cilk/reducer_min.h>

template <typename T>
size_t IndexOfMin(T array[], size_t n)
{
    cilk::reducer_min_index<size_t, T> r;

    cilk_for (int i = 0; i < n; i++)
    {
        r.calc_min(i, array[i]);
    }

    return r.get_index();
}
```

Пример 4: Секвенцијално сумирање

```
#include <iostream>

unsigned int compute(unsigned int i)
{
    return i; // return a value computed
              // from i
}

int main()
{
    unsigned int n = 1000000;
    unsigned int total = 0;

    // Compute the sum of integers 1..n
    for(unsigned int i = 1; i <= n; ++i)
    {
        total += compute(i);
    }
}
```

```
// the sum of the first n integers
// should be  $n * (n + 1) / 2$ 
unsigned int correct = (n * (n + 1)) / 2;

if (total == correct)
    std::cout << "Total (" << total
               << ") is correct" << std::endl;
else
    std::cout << "Total (" << total
               << ") is WRONG, should be "
               << correct << std::endl;

return 0;
}
```

Пример 4: Сумирање у паралели

```
#include <iostream>
#include <cilk/cilk.h>
#include <cilk/reducer_opadd.h>

unsigned int compute(unsigned int i)
{
    return i; // return a value computed
              // from i
}

int main()
{
    unsigned int n = 1000000;
    cilk::reducer_opadd<unsigned int>
total;

    // Compute the sum of integers 1..n
    cilk_for (unsigned int i = 1; i <= n;
++i)
    {
        total += compute(i);
    }
```

```
// the sum of the first n integers
// should be  $n * (n + 1) / 2$ 
unsigned int correct = (n * (n + 1)) / 2;

if (total.get_value() == correct)
    std::cout << "Total (" << total.get_value()
               << ") is correct" << std::endl;
else
    std::cout << "Total (" << total.get_value()
               << ") is WRONG, should be "
               << correct << std::endl;

return 0;
}
```

Редуктори у језику C++ су објекти, и као такви се не могу копирати директно позивом функције `memcpy()`. Потребно је користити конструктор копије.

Кориснички дефинисани редуктори

- Када ни један од расположивих редуктора не задовољава потребе програмера, он може да напише свој сопствени редуктор.
- Компоненте редуктора:
 - Конструктор и деструктор, који морају да буду јавни. Конструктор треба да постави *поглед* редуктора на **неутрални елемент**.
 - Класа ***monoid*** – мора да наследи `cilk::monoid_base<View>` и да садржи јавну статичку функцију `static void reduce (View *left, View *right)`.
 - Хиперобјекат који обезбеђује *погледе* за линије извршавања – приватна променљива декларисана као `cilk::reducer<Monoid> imp_`
 - Остатак редуктора, који обезбеђује рутине за приступ и промену података. По конвенцији, потребно је увести функцију `get_member()` која враћа вредност редуктора.

Пример 5: сопствени сумирајући редуктор

- Конструктор поставља *поглед* редуктора на неутрални елемент за сабирање (0).
- Функција `reduce()` додаје вредност десне инстанце вредности леве инстанце класе редуктора.
- Реализоване су операције `+=` и `++`. Остале операције се могу реализовати по потреби.
- Функција `get_value()` враћа резултат читавог низа операција сабирања. Иако је исправно позвати је у било ком тренутку, њене међувредности обично неће бити корисне пре него што се сви *погледи* хиперобјекта не редукују.