ПРОГРАМИРАНЕ С ОБЩА ПАМЕТ

OpenMp

OpenMP

- Стандарт за програмиране с обща памет
- Представлява API (Application Programming Interface) за паралелно програмиране на мултипроцесори
- Включва множество директиви за компилатора и библиотека от поддържащи функции
- Работи в комбинация със стандартен Fortran, C, C++

- Използва се за програмиране на мултипроцесори
- Използва се за програмиране на мултикомпютърни платформи с мултипроцесорни възли
- (напр. IBM RS/6000 с 512 възела, всеки от които съдържа до 16 процесора)
- Dell High Performance Computing Cluster – 64 възела, всеки възел съдържа 2 процесора Pentium III
- Fujitsu AP3000 – 1024 възела, всеки възел съдържа 2 процесора UltraSPARC

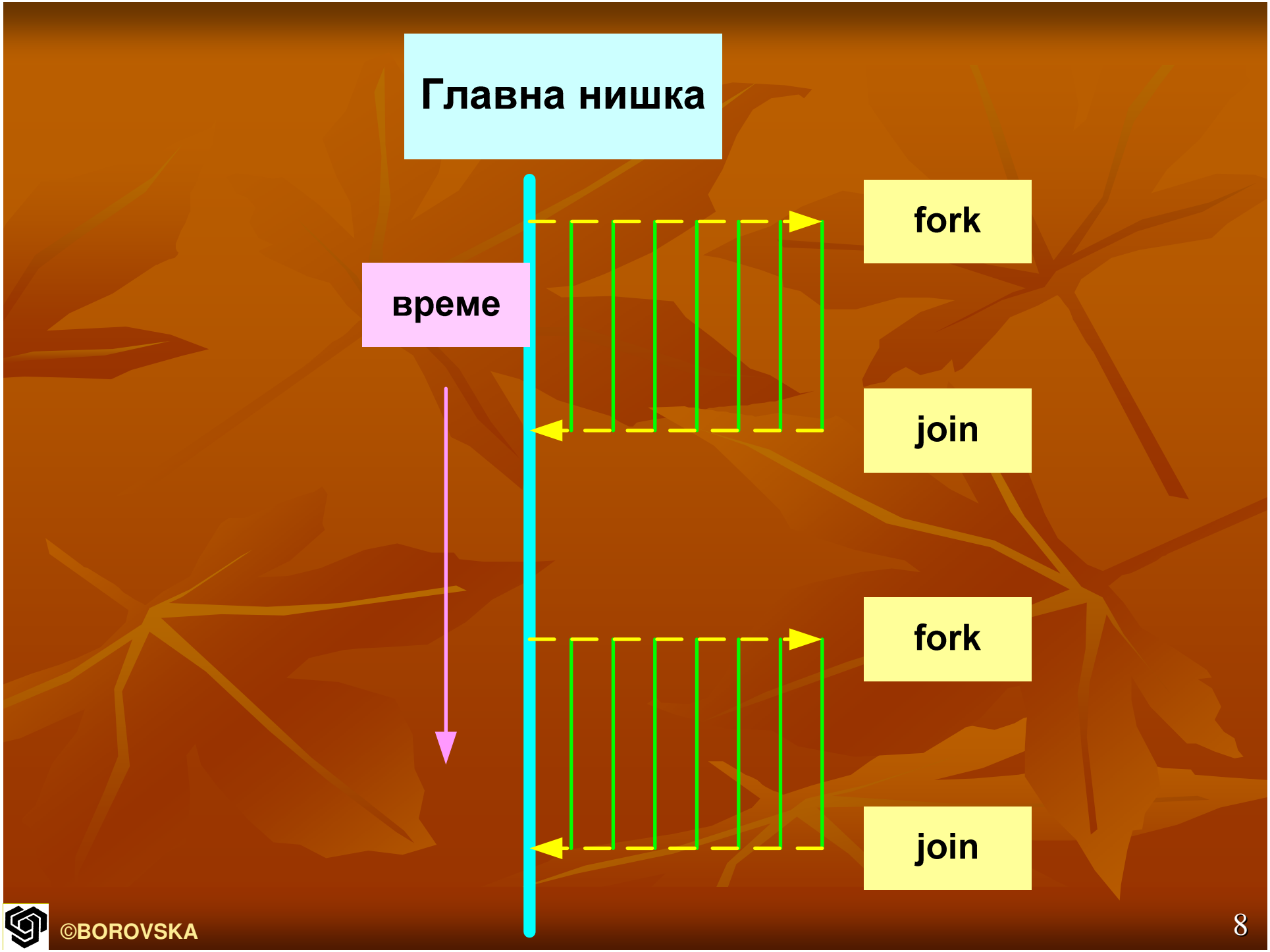
МОДЕЛ С ОБЩА ПАМЕТ

- *Представлява абстракция на централизирания мултипроцесор SMP*
- *Hardware се разглежда като множество процесори, всеки от които има еднакъв достъп до общата памет*
- *Процесорите си взаимодействат и се синхронизират чрез общата памет*

- ***Паралелизми fork/join*** *– стандартна концепция за паралелизъм в паралелната програма за обща памет*
- *При стартиране на програмата само една нишка, нар. master thread, е активна*
- *Главната нишка изпълнява последователните участъци на програмата*
- *Главната нишка разпростира допълнителни нишки -* ***fork***
- *В паралелните секции главната нишка и останалите нишки се изпълняват едновременно*

- ***В края на паралелния код разпространените нишки се терминират или суспендират и управлението се връща на главната нишка – join***
  - ***При MPI модела в общия случай всички процеси остават активни през времето на изпълнението на програмата***
  - ***При модела с обща памет броят на активните нишки се променя динамично в хода на изпълнение на програмата***
  - ***<u>Последователната програма може да се разглежда като специален случай на паралелна програма с обща памет- има само един fork/join</u>***

- Моделът с обща памет поддържа *инкрементална паралелизация* – варира от един fork/join около единствен цикъл до множество паралелни сегменти от код – главно предимство пред модела с обмен на съобщения
  - При модела с обмен на съобщения трансформацията на последователната програма в паралелна не е инкрементална
- Осигурява възможност за профилиране на паралелното изпълнение на програмата



Паралелни цикли for

- Потенциален паралелизъм в програмите на C представляват циклите for
  - При OpenMP лесно се определя кога итерациите на цикъла могат да бъдат изпълнени паралелно
    - Напр. за Sieve of Eratosthenes:

for (i=first; i<size; i+=prime) marked[i]=1;

Няма зависимости между итерациите

Дават се указания на компилатора за паралелизация на цикъла

Прагма parallel for

- Pragma – директива за компилатора в C или C++
- Съкращение от “pragmatic information”
- Начин да се предаде информация на компилатора
- Компилаторът може да игнорира тази информация
- Помага на компилатора да организира програмата

- Аналогично на другите редове, осигуряващи информация за предпроцесора, прагмата започва с #
- Синтаксис на прагма в C или C++

pragma omp <rest of pragma>

- Най-простата форма на прагмата parallel for

```
#pragma omp parallel for
    for (i = first; i < size; i += prime) marked[i] = 1;
```

- ***Управляващата клауза на цикъла не трябва да поддържа принудително излизане от цикъла – операторите break, return, exit, goto към етикети извън цикъла не са разрешени, операторът continue се допуска***

- ***Трябва да се определи броят на итерациите***
- ***Управляващата клауза на цикъла for трябва да има канонична форма – идентификаторите start, end, inc могат да бъдат изрази***

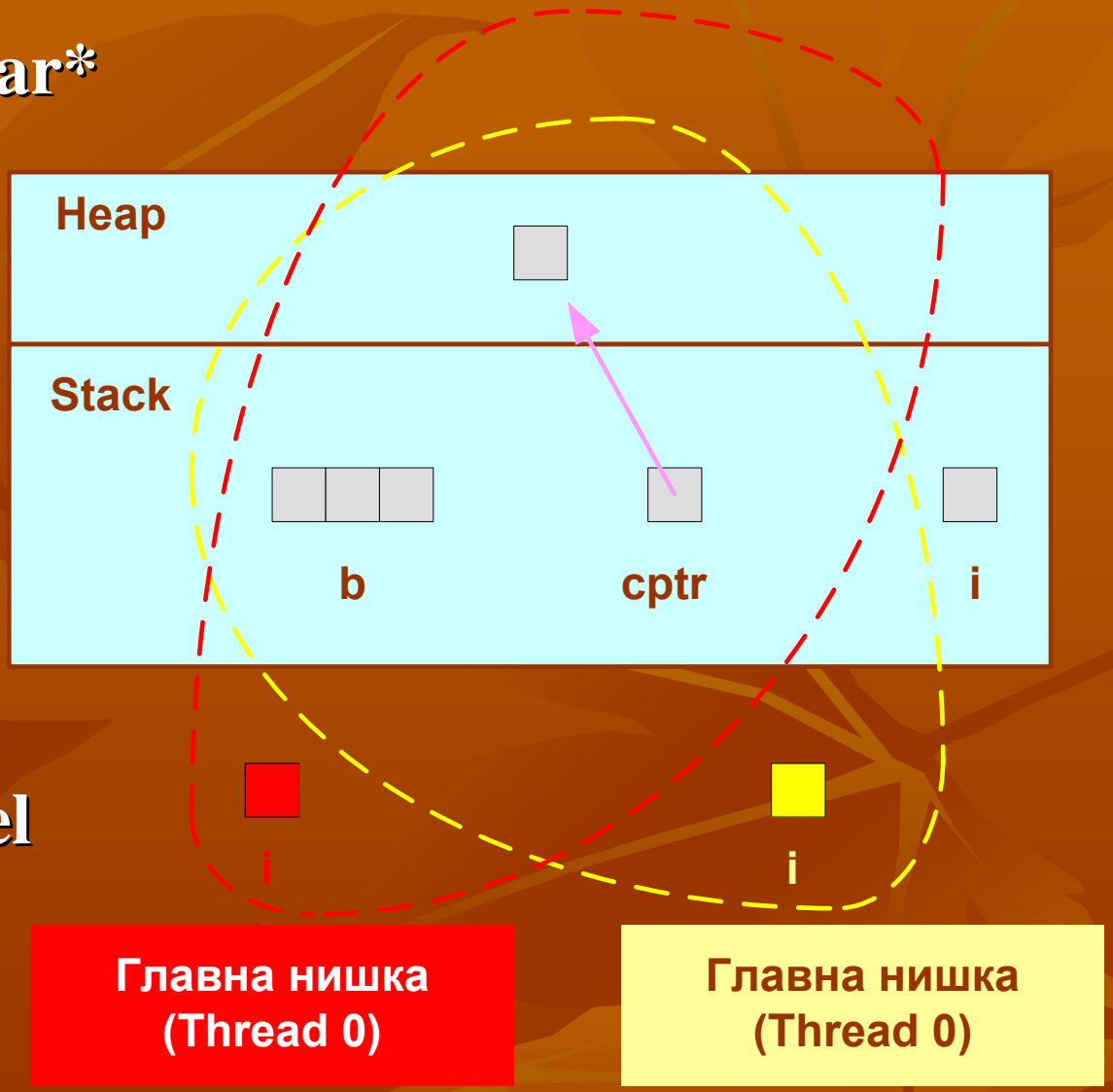
```
for (index = start; index { <, <=, >=, > } end; { index++, ++index, index--, --index, index += inc, index -= inc, index = index + inc, index = inc + index, index = index - inc } )
```

- По време на изпълнението на програмата главната нишка разпростира нови нишки за да покрие изпълнението на итерациите на цикъла
- Всяка нишка има свой изпълнителен контекст: адресното пространство на променливите, до които нишката може да осъществява достъп
- Изпълнителният контекст съдържа статични променливи, динамично разпределени структури, и променливи от run-time стека
- Изпълнителният контекст съдържа собствен допълнителен run-time стек за рамките на функциите, които извиква

- Общата променлива има един и същ адрес в изпълнителните контексти на всички нишки
- private променлива може да има различни адреси в изпълнителните контексти на отделните нишки
- Нишките могат да осъществяват достъп само до собствените private променливи, но не могат да осъществяват достъп до private променливи на други нишки
- При ***parallel for pragma*** променливите са общи by default, с изкл. на индекса на цикъла, който е private променлива

```
int main (int argc, char*
    argv[])
{
    int b[3] ;
    char* cptr ;
    int i;

    cptr = malloc (1) ;
#pragma omp parallel
    for
  for (i=0; i<3; i++)
    b[i]=i ;
```



- Стойността на променливата на средата OMP_NUM_THREADS осигурява default брой на нишките за секциите паралелен код
- Друга стратегия е броят на нишките да се зададе като равен на броя на процесорите в мултипроцесора
- Функцията omp_get_num_procs връща броя на физическите процесори, които могат да се използват от паралелната програма
- Цялото число, върнато от функцията може да бъде по-малко от броя на физическите процесори в зависимост от планиращата стратегия

- Функцията omp_set_num_threads използва стойността на параметъра за да зададе броя на активните нишки за паралелните секции на кода
- Function header:

void omp_set_num_threads (int i)

- Функцията може да се извиква от множество точки на програмата – възможност за адаптиране на нивото на паралелизъм към големината на гранулата или други характеристики на кода

- Задаването на броя на нишките да бъде = броя на наличните процесори е чрез:

```
int t ;
…
t = omp_get_num_procs();
Omp_set_num_threads (t) ;
```

- ***Размерът на гранулата*** се определя от броя на изчисленията, реализирани между стъпки на комуникация или синхронизация
- В общия случай, увеличаването на размера на гранулата подобрява производителността на паралелната програма

Проблем – как да се паралелизират вложени цикли?

- Лесно е да се даде директива на компилатора да паралелизира цикъл индексиран с i
- By default всички променливи са общи с изключение на индекс i на цикъла
- Това улеснява комуникацията между нишките, но може да създаде проблеми
- Паралелна реализация на итерациите на цикъл i – всяка нишка ще обработи n стойности на j за всяка итерация; съществува опасност нишките да не изпълнят n итерации → j също трябва да бъде private

Private Clause

- Clause е допълнителна компонента (опция) на pragma
- Тя дава директива на компилатора да направи една или повече променливи private
- Синтаксис: private (<списък променливи>)
- Директивата указва на компилатора да разпредели private копие на променливата за всяка нишка, изпълняваща блока от код, предхождан от pragma
- При влизане в паралелната конструкция и при напускането ѝ стойността на private променлива е неопределена by default

firstprivate Clause

- Използва се, ако искаме private променлива да наследи стойността на обща променлива
- Синтаксис firstprivate (<списък променливи>)
- Дава директива на компилатора да създаде private променливи с начални стойности = стойностите на променливите, управлявани от главната нишка, при влизане в цикъла
- Стойностите на променливите във firstprivate се инициализират веднъж за нишка, не по веднъж за итерация

lastprivate Clause

- ***Последната последователна итерация на цикъла*** е последната изпълнена итерация
- Дава директива на компилатора да генерира код в края на цикъла parallel for, който копира обратно в главната нишка стойността на private променливата от нишката, изпълнила последната последователна итерация на цикъла
- Pragma parallel for може да съдържа firstprivate & lastprivate clauses

КРИТИЧНИ СЕКЦИИ

- *пример: част от програма на C, изчисляваща стойността на π с помощта на числено интегриране по правилото на правоъгълника*

```
double area, pi, x;
int i, n;
…
area = 0.0;
for (i = 0; i < n ; i++)  {
  x = (i+0.5) / n;
  area += 4.0/ (1.0 + x*x);
}
pi = area / n;
```

- *Итерациите на този цикъл не са независими*
- *Всяка итерация на цикъла чете и обновява стойността на area*
- *При паралелизация на цикъла получаваме:*

```
double area, pi, x;
int i, n;
…
area = 0.0;
#pragma omp parallel for private(x)
for (i = 0; i < n; i++)    {
    x = (i+0.5) /n;
    area += 4.0/ (1.0 + x*x); /* Race Condition! */
}
pi = area / n;
```

- *Отговорът няма да е верен*
- *Изпълнението на оператора за присвояване не е атомична (atomic), неделима (indivisible) операция*
- *Получава се условие за състезание (race condition) – изчислителният процес се характеризира с недетерминистично поведение, когато множество нишки осъществяват достъп до обща променлива*
- *Примерна реализация: нишки A и B изчисляват паралелно различните итерации на цикъла*

- *Нишката A чете текущата стойност на area и изчислява сумата*

  area + 4.0/(1.0 + x*x)

- *Преди нишката A да успее да запише резултата в area, нишката B прочита стойността на area*
- *Нишката B изчислява сумата и записва резултата в area*
- *В случая стойността на area не е коректна*
- *Операторът за присвояване, който чете и обновява area трябва да се постави в критична секция – част от кода, която в даден момент може да се изпълнява само от една нишка*

critical Pragma

- *Критична секция в OpenMP се обособява, като се поставя прагмата critical преди блока от кода на C*

#pragma omp critical

- *Дава директива на компилатора да осигури взаимно изключване на нишките, изпълняващи едновременно указания блок от код*
- *Итерациите на цикъла for се разделят между нишките, само 1 нишка обновява area → малко ускорение (закон на Амдал)*

Клауза за редукция

Синтаксис:

reduction (<op>:<variable>)

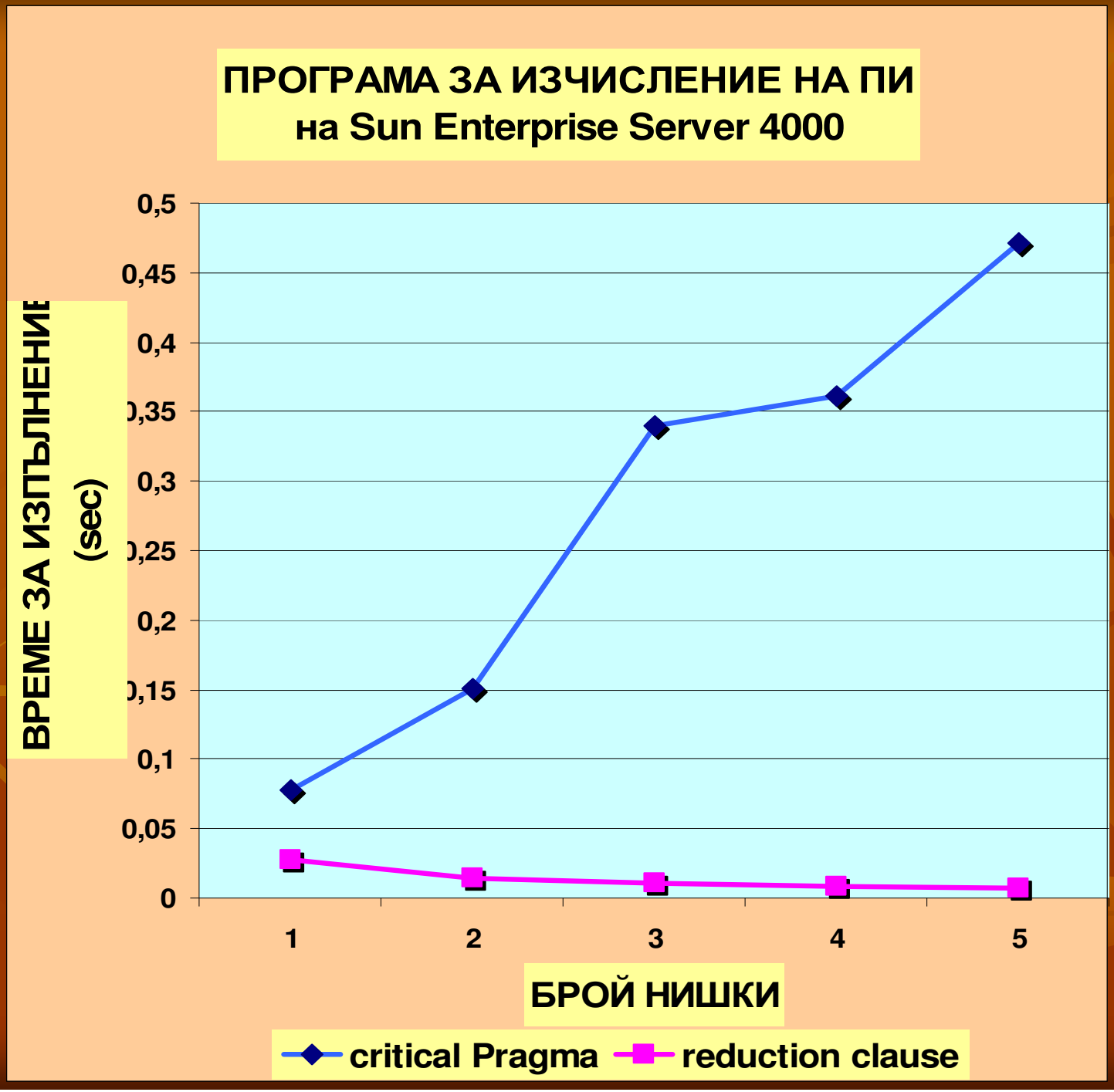
където: <op> е операция за редукция, а <variable> е общата променлива, съдържаща резултата от редукцията

Оператори за редукция в OpenMP за C и C++

Оператор	Значение	Разрешени типове	Начална стойност
+	Сума	float, int	0
*	Умножение	float, int	1
&	Побитова and	int	Всички битове са 1
|	Побитова or	int	0
^	Побитова eor	int	0
&&	Логическа and	int	1
||	Логическа or	int	0

Имплементация на кода за π при която критичната секция е заменена с клауза за редукция

```
double area, pi, x;
int i, n;
…
area = 0.0;
#pragma omp parallel for private(x) reduction (+:area)
for (i = 0; i < n; i++)    {
     x = (i+0.5)/n;
}
pi = area / n;
```


ПРОГРАМА ЗА ИЗЧИСЛЕНИЕ НА ПИ
на Sun Enterprise Server 4000
ВРЕМЕ ЗА ИЗПЪЛНЕНИЕ
(sec)
0,5
0,45
0,4
0,35
0,3
0,25
0,2
0,15
0,1
0,05
0
1
2
3
4
5
БРОЙ НИШКИ
critical Pragma
reduction clause

Ускоряване на изпълнението на паралелните цикли

1. *Инвертиране на циклите*
2. *Условно изпълнение на циклите*
3. *Планиране на изпълнението на циклите*

Инвертиране на циклите

- Разглеждаме следния код:

```
for (i = 1; i < m; i++)
    for (j = 0; j < n; j++)
        a [i] [j] = 2 * a [i-1] [j];
```

- Съществуват зависимости между редовете
- При колоните не съществуват зависимости ⇒ могат да бъдат обработвани паралелно
- ⇒ цикълът с индекс i не може да бъде изпълнен паралелно
- ⇒ цикълът с индекс j може да бъде изпълнен паралелно

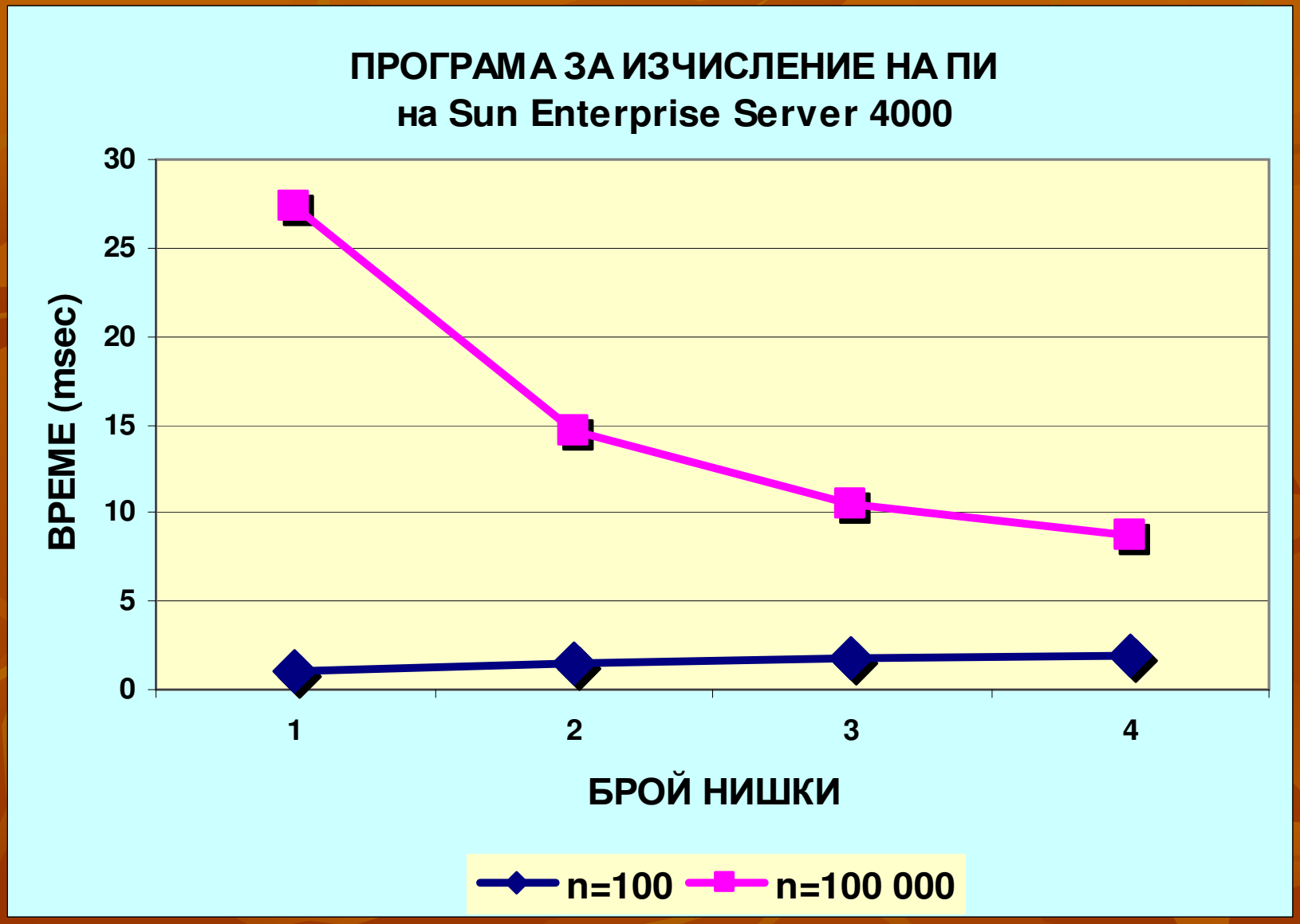
```
#pragma parallel for private(i)
for (j = 0; j < n; j++)
    for (i = 1; i < m, i++)
        a [i] [j] = 2 * a [i-1] [j] ;
```

- Необходим е само един fork/join (за външния цикъл)
- Зависимостите по данни не са променени
- Трансформацията на кода оказва влияние върху скоростта на попадение в кеша
- Нишката обработва данните по колони, а матрицата е разположена по редове в паметта
- Sup зависи от размера на матрицата, броя на активните нишки и архитектурата на системата

Условно изпълнение на цикли

- *При малък брой итерации на цикъла допълнителните разходи за разпростирането и терминирането на нишките (fork/join) може да доведе до забавяне на паралелното изпълнение спрямо последователното изпълнение*

```
  area = 0.0;
#pragma omp parallel for private(x) reduction (+:area)
   for (i=0; i<n; i++)   {
      x = (i + 0.5) / n;
      area += 4.0 / ( 1.0 + x * x) ;
   }
   pi = area / n ;
```


ПРОГРАМА ЗА ИЗЧИСЛЕНИЕ НА ПИ
на Sun Enterprise Server 4000
ВРЕМЕ (msec)
30
25
20
15
10
5
0
1
2
3
4
БРОЙ НИШКИ
n=100
n=100 000

if clause

- Дава директива на компилатора да вмъкне код, който в run-time определя дали цикълът ще се изпълни паралелно или последователно
- Синтаксис:

if (<scalar expression>)

- Ако стойността на скаларния израз е истина, цикълът се изпълнява паралелно, в противен случай – последователно

```
#pragma omp parallel for private(x) reduction (+:area) if (n > 5000)
    for (i=0; i<n,i++){
      …
```

Планиране на циклите

- При някои цикли времето за изпълнение на отделните итерации варира в широки граници
- Напр., двоен вложен цикъл за инициализиране на горна триъгълна матрица

```
for (i = 0; i < n, i++)
    for (j = i; j < n; j++)
        a [i] [j] = alpha_omega (i,j) ;
```

- Няма зависимост по данни между итерациите ⇒ паралелизираме външния цикъл- дисбаланс

schedule clause

- Дава възможност да укажем по какъв начин итерациите на цикъла се разпределят между нишките
- При ***static schedule*** – итерациите се разпределят между нишките преди изпълнението - дисбаланс на товара
- При ***dynamic schedule*** – само някои от итерациите се разпределят в началото, нишките, завършили обработката, могат да бъдат активирани отново – големи допълнителни разходи, добър баланс

schedule clause

- Определен брой итерации с последователни номера, наречени ***chunks***, се разпределят за изпълнение от всяка от нишките
- Увеличаването на размера на ***chunks*** намалява допълнителните разходи и увеличава скоростта на попаденията в кеша
- Намаляването на размера на ***chunks*** води до възможност за по-добро балансиране на изчислителния товар
- Синтаксис:

  schedule (type>[,<chunk>])

  размерът на chunks е опция

schedule clause

- ***schedule (static)***: статично разпределение – n/t последователни итерации за нишка
- ***schedule (static,C)***: всеки chunk съдържа C последователни итерации
- ***schedule (dynamic)***: динамично разпределение на итерациите между нишките
- ***schedule (dynamic,C)***: динамично разпределение на C итерации за нишка
- ***schedule (guided,C)***: динамично евристично разпределение – размерът на chunk намалява експоненциално от C

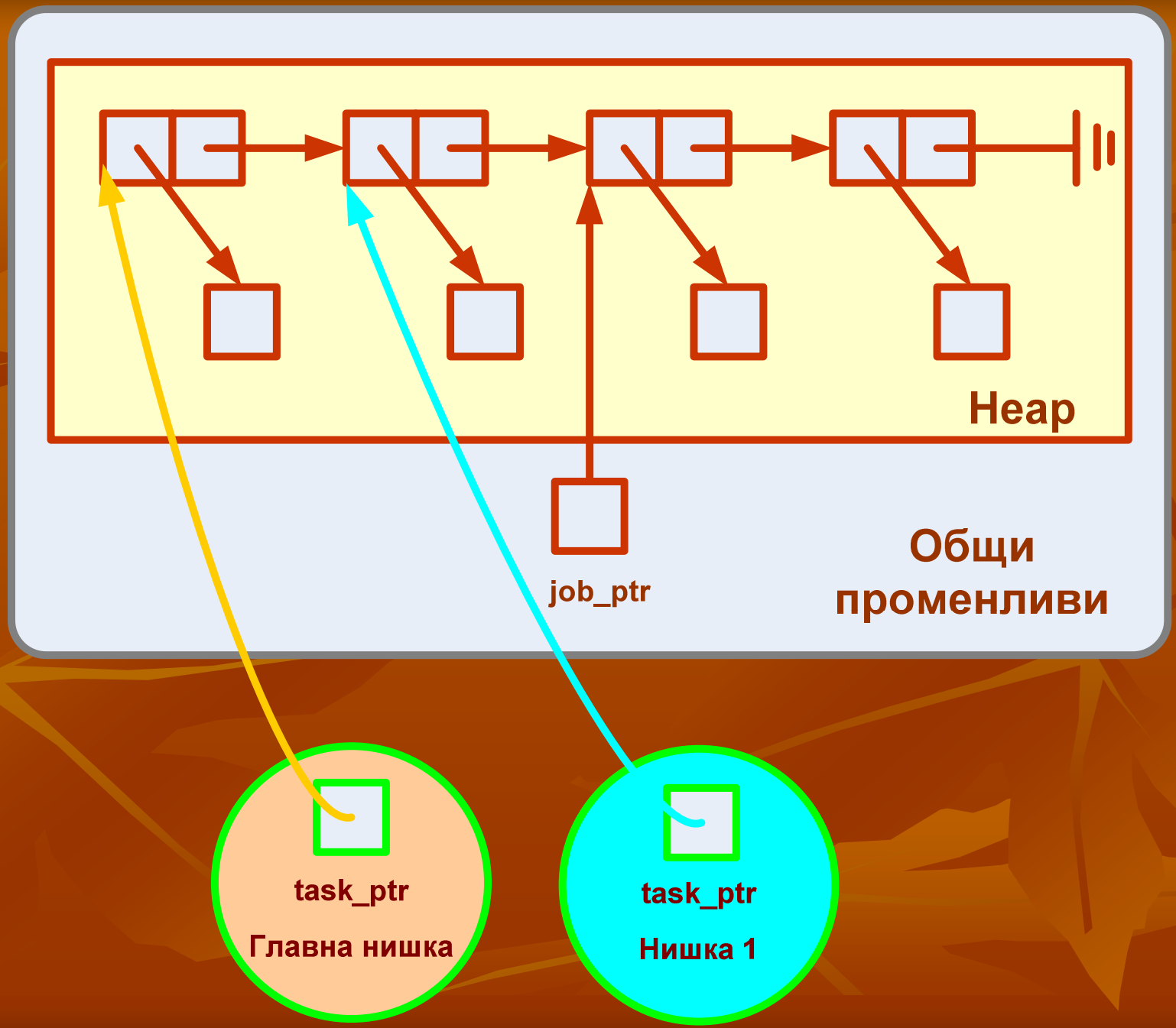
schedule clause

- ***schedule (guided)***: динамично евристично разпределение – минималният размер на chunk е 1
- ***schedule (runtime)***: видът на планиране се изпълнява в хода на изпълнение на програмата с помощта на променливата на средата OMP_SCHEDULE
  - Пример:

```
#pragma omp parallel for private(j) schedule (static,1)
for (i = 0; i < n, i++)
    for (j = 0; j < n, j++)
        a [i] [j] = alpha_omega (i , j) ;
```

Общи паралелизми по данни

- *Други паралелизми извън циклите for*
- *Напр, алгоритъм, обработващ свързан списък от задачи (при класифицирането на доку-менти) – отпада необходимостта от процес worker (MPI) т.к. всяка нишка има достъп до общия списък*
- *Всяка нишка взема следващата задача от списъка и я изпълнява – до изчерпването на задачите в списъка*
- *Две нишки не трябва да вземат една и съща задача – функцията* get_next_task - *атомична*


Heap
job_ptr
Общи променливи
task_ptr
Главна нишка
task_ptr
Нишка 1

```
int main (int argc, char argv [])
{
    struct job_struct  job_ptr;
    struct task_struct  task_ptr;
    …
    task_ptr = get_next_ task (&job_ptr);
    while (task_ptr != NULL)    {
        complete_task (task_ptr) ;
        task_ptr = get_next_ task (&job_ptr);
    }
    …
}
```

```
char get_next_task (struct job_struct job_ptr)    {
      struct task_struct answer ;
      if (job_ptr == NULL) answer = NULL;
      else    {
          answer = (job_ptr) ->task;
          job_ptr = (job_ptr)->next;
       }
       return answer;
}
```

Parallel Pragma

- *Предхожда блок от код, който трябва да бъде изпълнен от всички нишки*
- **Синтаксис:**

#pragma omp parallel

```
int main (int argc, char argv [])
{
    struct job_struct  job_ptr;
    struct task_struct  task_ptr;
#pragma omp parallel private (task_ptr)
      {
          task_ptr = get_next_task (&job_ptr) ;
          while (task_ptr != NULL)    {
              complete_task (task_ptr) ;
              task_ptr = get_next_ task (&job_ptr);
          }
      }
}
```

Функцията get_next_task - атомична

```
char get_next_task (struct job_struct job_ptr)    {
    struct task_struct answer;
#pragma omp critical
    {
        if (job_ptr == NULL) answer = NULL;
        else    {
            answer = (job_ptr)->task;
            job_ptr = (job_ptr)->next;
        }
    }
    return answer;
}
```

Функция omp_get_thread_num

- Пример – изчисляване на π по метода Монте Карло
- Ако искаме да ускорим изчислението, създаваме множество нишки, които трябва да използват различни потоци от случайни числа ⇒ трябва да можем да различаваме нишките
- В OpenMP всяка нишка има уникален идентификатор (номер)
- Използваме функцията (header)

int omp_get_thread_num (void)

- При t активни нишки – номера от 0 до t-1
- Главната нишка винаги има ID 0

Функция omp_get_num_threads

- *За да разделим итерациите между нишките, трябва да знаем броя на активните нишки*
- *Функция* **omp_get_num_threads (header)**

int omp_get_num_threads (void)

връща броя на активните нишки в текущия участък на паралелната програма

- *Броя на нишките и ID им номера се използват за разпределянето на итерациите между нишките*
- *Всяка нишка акумулира стойностите в private променлива*
- *Всяка нишка прибавя частичния си резултат към count в рамките на критична секция*

For Pragma

- Полезна при паралелизацията на цикли for
- Разглеждаме простия двоен вложен цикъл:

```
for (i = 0; i < m; i++)    {
    low = a[i] ;
    high = b[i] ;
    if (low > high)    {
        printf (“Exiting during iteration %d\n”, i) ;
        break ;
    }
    for   (j = low; j < high; j++)
        c[j] = (c[j] – a[i]) / b[i] ;
}
```

- Не можем да изпълним паралелно итерациите на външния цикъл, защото съдържат **break**
- Ако поставим parallel pragma непос-редствено преди цикъла с индекс i, ще имаме един fork/join
- Всяка нишка изпълнява целия код в блока (default)
- Нишките ще си разпределят итерациите на вътрешния цикъл
- Използваме pragma for

#pragma omp for

```
#pragma omp parallel private( i, j)
for (i = 0; i < m; i++)    {
    low = a[i] ;
    high = b[i] ;
    if (low > high)    {
        printf (“Exiting during iteration %d\n”, i) ;
        break ;
    }
#pragma omp for
    for (j = low; j < high; j++)
        c[j] = (c[j] – a[i]) / b[i] ;
}
```

Single Pragma

- Указва на компилатора, че само една нишка ще изпълни кода, който предхожда
- Синтаксис:

```
#pragma omp single
```

```
#pragma omp parallel private( i, j)
for (i = 0; i < m; i++)    {
    low = a[i] ;
    high = b[i] ;
    if (low > high)    {
#pragma omp single
        printf (“Exiting during iteration %d\n”, i) ;
        break ;
    }
#pragma omp for
    for (j = low; j < high; j++)
        c[j] = (c[j] – a[i]) / b[i] ;
}
```

nowait Clause

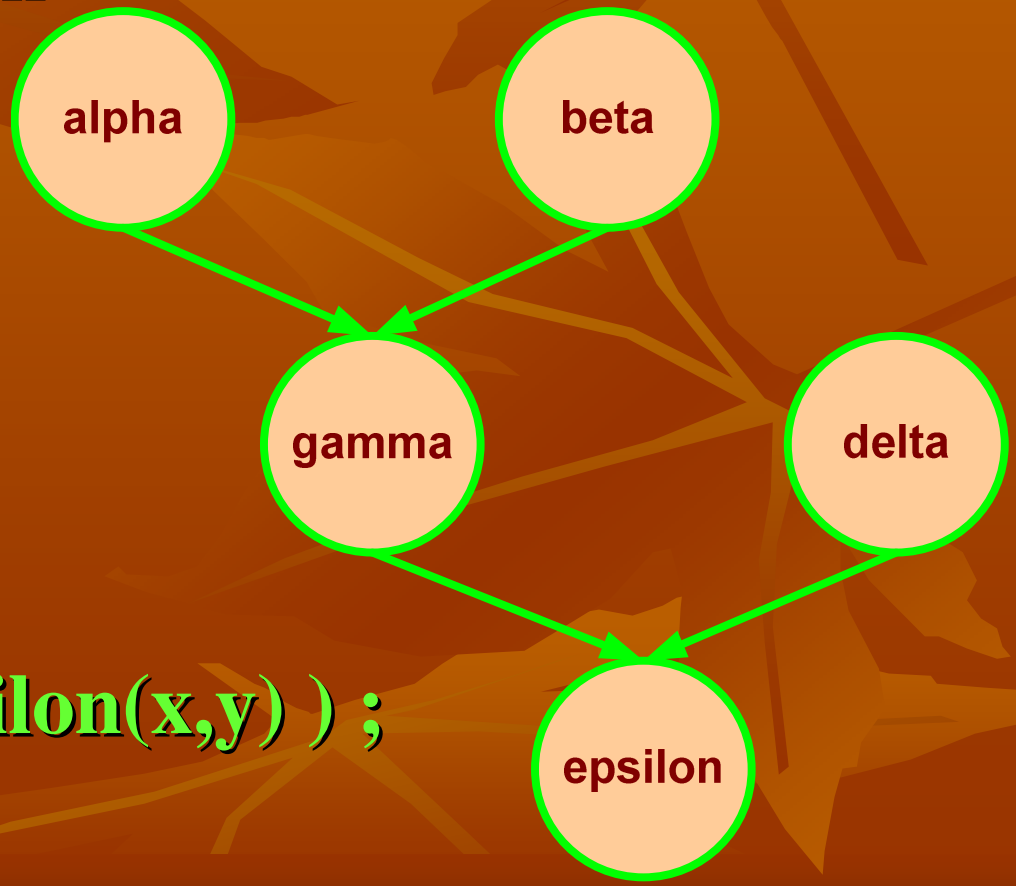
- Компилаторът поставя бариерна синхронизация в края на всеки оператор parallel for
- Клаузата nowait указва на компилатора да пропусне бариерната синхронизация в края на цикъла parallel for, индексиран с j
- Ако направим променливите low и high да бъдат private, не е необходима бариерна синхронизация в края на цикъла parallel for, индексиран с j

```
#pragma omp parallel private( i, j, low, high)
for (i = 0; i < m; i++)    {
    low = a[i] ;
    high = b[i] ;
    if (low > high)    {
#pragma omp single
        printf (“Exiting during iteration %d\n”, i) ;
        break ;
    }
#pragma omp for nowait
    for (j = low; j < high; j++)
        c[j] = (c[j] – a[i]) / b[i] ;
}
```

Функционален паралелизъм

- OpenMP осигурява възможност да определяме кода, който ще бъде изпълняван от отделните нишки
- Например:

```
v = alpha () ;
w = beta () ;
x = gamma (v, w) ;
y = delta () ;
printf (“%6.2f\n”, epsilon(x,y) ) ;
```



parallel sections Pragma

- Предхожда блок от *k* блока код, които ще бъдат изпълнявани паралелно от *k* нишки
- Синтаксис:

#pragma omp parallel sections

section Pragma

- Предхожда всеки блок от код, включен в блока, предхождащ прагмата parallel sections
- За примера – alpha, beta и delta могат да бъдат изпълнени едновременно
- Пренареждаме операторите за присвояване за да осигурим паралелното изпълнение на alpha, beta и delta

```
#pragma omp parallel sections
   {
#pragma omp section                    /*optional*/
      v = alpha ( ) ;
#pragma omp section
      w = beta ( ) ;
#pragma omp section
      y = delta ( ) ;
   }
   x = gamma (v, w) ;
   printf (“%6.2f\n”, epsilon(x,y) ) ;
```

sections Pragma

- Алтернативен начин за използване на функционалния паралелизъм
- Можем да изпълним паралелно alpha и beta, след което да изпълним паралелно gamma и delta
- Две паралелни секции, изпълнявани една след друга ⇒ необходими са две нишки
- Синтаксис:

```
#pragma omp sections
```

- ~ parallel sections pragma

```
#pragma omp parallel
  {
  #pragma omp sections
    {
    #pragma omp section            /*optional*/
      v = alpha ( ) ;
    #pragma omp section
      w = beta ( ) ;
    }
  #pragma omp sections
    {
    #pragma omp section            /*optional*/
    x = gamma (v, w) ;
    #pragma omp section
    y = delta ( ) ;
    }
  }
printf (“%6.2f\n”, epsilon(x,y) ) ;
```

Прагма	Разрешени клаузи
critical	няма
	firstprivate, lastprivate, nowait, private, reduction, schedule
parallel	firstprivate, if, lastprivate, private, reduction
parallel for	firstprivate, if, lastprivate, private, reduction, schedule
parallel sections	firstprivate, if, lastprivate, private, reduction
sections	firstprivate, lastprivate, nowait, private, reduction
single	firstprivate, nowait, private

Комбиниране на MPI и OpenMP

- Повечето комерсиални мултикомпютри представляват колекция от централизирани мултипроцесори
- Много комерсиални клъстери имат дву- или четири процесорни възли
- Желателно е да се трансформира MPI програма в програма с комбинирано използване на MPI и OpenMP за изпълнение на клъстер от мултипроцесори
- Един MPI процес се изпълнява от всеки мултипроцесор

- Вътре в паралелните секции на кода MPI процесите разпростират нишки за изпълнение от процесорите на мултипроцесора
- Нишките взаимодействат чрез общи променливи
- В много случаи хибридните програми с използване на MPI и OpenMP се изпълняват по-бързо от MPI програмите, защото имат по- малки разходи за комуникация

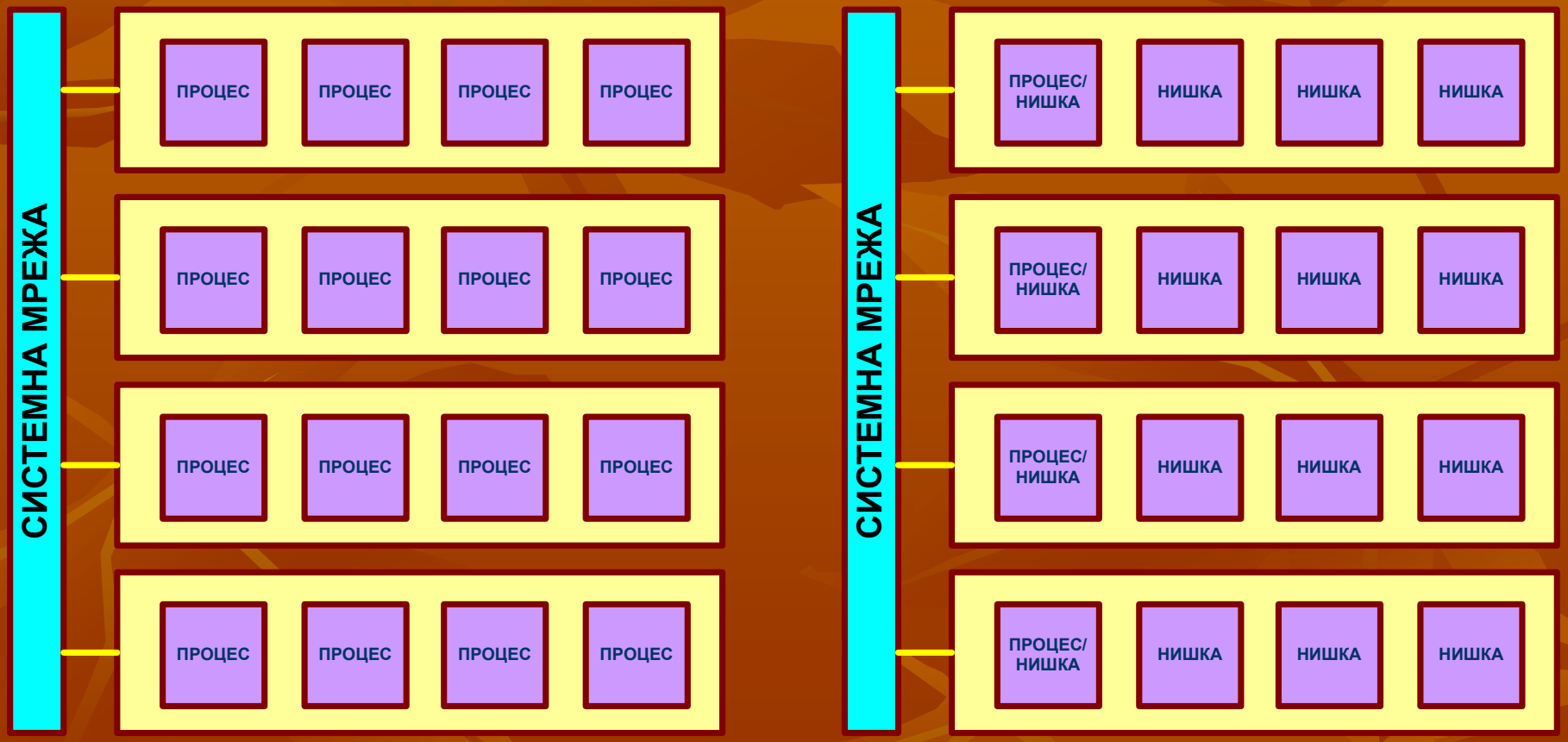
Програмиране на клъстер

- Клъстер от *m* мултипроцесора
- Всеки мултипроцесор има *k* процесора
- За да натовари всички процесори, MPI програмата трябва да създаде *m***k* процеса
- При стъпките на комуникация, всичките *m***k* процеса са активни
- Комбинираната програма трябва да създаде само *m* процеса
- В паралелните секции на кода работният товар се разпределя между *k* нишки на всеки мултипроцесор – всички процесори работят

Програмиране на клъстер

- При комуникация само *m* процеса са активни
- Обуславя по-ниски разходи на комуникация ⇒ по-голямо ускорение
- Комбинираната програма дава възможност да се паралелизират част от изчисленията чрез “по-леките” нишки вместо с “тежки” процеси
- Процесите, очакващи съобщения, могат да изпълняват нишки → по-голямо ускорение

Програмиране на клъстер



К Р А Й