THE SEIVE OF ERATOSTHENES

ПРЕБРОЯВАНЕ НА ПРОСТИТЕ ЧИСЛА

ПАРАЛЕЛНА РЕАЛИЗАЦИЯ С MPI

©BOROVSKA

Източници на паралелизъм

- Domain decomposition
- Маркиране на сложните числа
  - Немаркираните са прости
  - Една задача отговаря за множество елементи на масива

	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60

	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60

©BOROVSKA

	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60

	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60

©BOROVSKA

BLOCK DATA DECOMPOSITION

- Масивът се разделя на *p* непрекъснати блока с приблизително еднакъв размер
- Разпределянето на блоковете данни между процесите трябва да осигурява добър баланс на изчислителния товар
- **Всеки процес обработва** $\lceil n/p \rceil$

или $\lfloor n/p \rfloor$ **елемента**

Макроси за декомпозиция по блокове

```
#define BLOCK_LOW (id,p,n) ( (id) * (n) / (p) )
#define BLOCK_HIGH (id,p,n) (BLOCK_LOW
((id)+1,p,n) – 1)
#define BLOCK_SIZE (id,p,n) (BLOCK_LOW
((id)+1) – BLOCK_LOW (id) )
#define BLOCK_OWNER (index,p,n)   ( ( (p) *
( (index) + 1) – 1) / (n) )
```

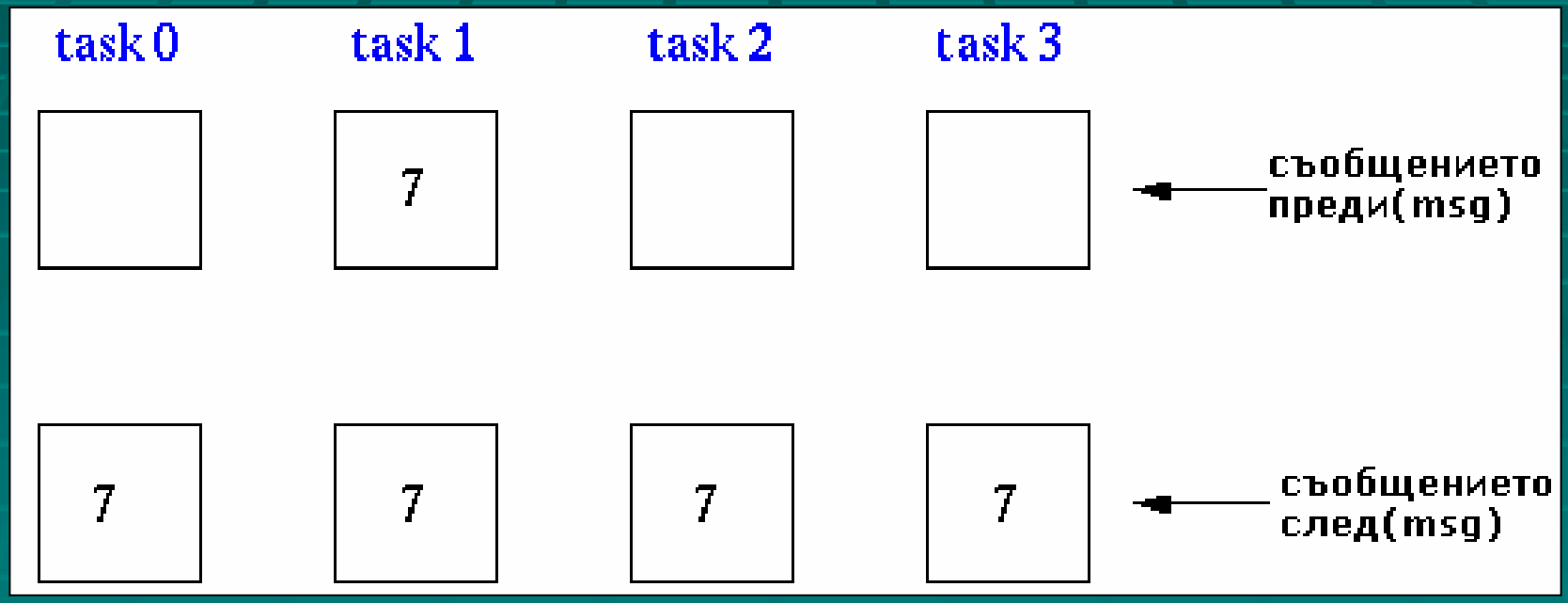
- За даден процес с ранг ID, и брой на елементите n, macro BLOCK_LOW → min index, обработван от процеса
- macro BLOCK_HIGH стойността на max index, обработван от процеса
- macro BLOCK_SIZE определя броя на елементите, обработвани от процес ID
- macro BLOCK_OWNER определя ранга на процеса, отговорен за обработката на елемента
- Utility macros - reference

Локални и глобални индекси

ПАРАЛЕЛНА ПРОГРАМА

- Всеки процес създава своята част от общия списък, съдържащ съответния брой булеви стойности
- Всеки процес трябва да знае стойността на k, за да може да маркира числата, кратни на k
- Всеки процес трябва да маркира всички числа, кратни на k в неговия блок между k^2 и n

- Процес 0 трябва да определи следващата стойност на k и да я изпрати на останалите процеси
- Стойността на k трябва да се копира в локалните инстанции на k в останалите процеси
- ***Broadcasting – функция за глобална комуникация***


task 0
task 1
task 2
task 3
7
съобщението
преди(msg)
7
7
7
7
съобщението
след(msg)

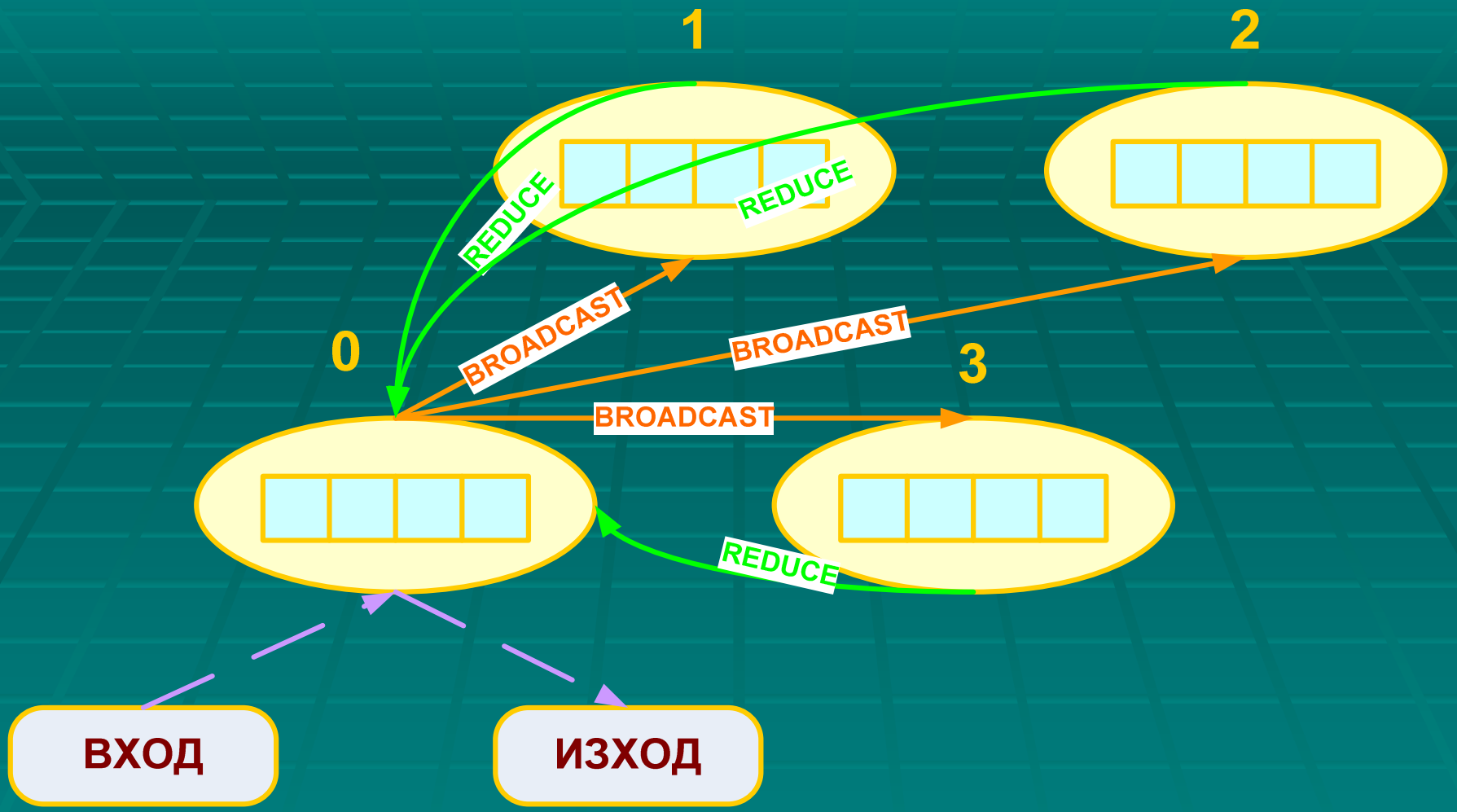
Функция MPI_Bcast

- ✓ Един процес изпраща едни и същи данни към всички останали процеси в рамките на комуникатора

```
Int MPI_Bcast  (
   void              *buffer,   /*адрес на 1ВИ елемент*/
   int               count,     /*брой елементи*/
   MPI_Datatype datatype,  /*тип на елементите*/
   int               root,      /*ID на процеса-източник*/
   MPI_Comm  Comm)    /*Комуникатор - групата*/
```

MPI_Bcast (&k, 1, MPI_INT, 0, MPI_COMM_WORLD)

- След изпълнението на broadcast, всеки процес разполага с актуалната стойност на k
- Всеки процес преброява простите числа в подмасива си
- Всички процеси изпращат сумата на простите си числа към процес 0 като извикват функцията MPI_Reduce



ГРАФ НА ЗАДАЧИТЕ

©BOROVSKA

ДОКУМЕНТИРАНЕ НА ПАРАЛЕЛНАТА ПРОГРАМА

- header file MyMPI.h съдържа макроси и прототипи на функции за разработваните утилити
- Дефинираме макро, което изчислява минималната от две стойности

```
#define MIN(a,b)     ( (a)<(b) ? (a):(b) )
```

- Потребителят трябва да зададе горната граница на пресяването като аргумент на командна линия; ако липсва – терминираме изпълнението

- Особено важно е всеки процес да извика MPI_Finalize () преди exit
- Ако съществува аргумент на командната линия, то стрингът се конвертита в int

```
If ( argc != 2 )   {
   if (!id)  printf  ( “Command line: %s <m>\n”, argv[0] );
   MPI_Finalize();
   exit  (1);
}
n = atoi (argv[1]);
```

- Програмата ще намери всички прости числа от 2 до n, при което търсим всички прости числа от n-1 int

- Всеки процес разполага с непрекъснат блок от масива за съхраняване на marks
- С помощта на макросите се определят долната и горната граници за процеса и общия брой на числата за пресяване

```
low_value = 2 + BLOCK_LOW ( id,p,n-1 ) ;
high_value = 2 + BLOCK_HIGH ( id,p,n-1 ) ;
size = BLOCK_SIZE ( id,p,n-1 ) ;
```

- Алгоритъмът работи само ако най-голямата стойност в масива на процес 0 е по-голяма от горната граница на пресяване

- Код за проверка на условието – ако не е изпълнено – изпълнението на програмата се прекратява

```
Proc0_size = (n-1) / p;

If  ( (2 + proc0_size) < (int) sqrt( (double)  n))  {
    if  ( !id ) printf  (“Too many processes\n”);
    MPI_Finalize ();
    exit   (1) ;
}
```

- Разпределяме дяловете от масива на процесите (allocate)
- Най-малката единица в паметта, която може да бъде индексирана в C е 1 байт
- Декларираме типа на масива **char**
- Ако разпределението на паметта се провали, изпълнението на програмата се прекратява

```
marked = (char *)  malloc  (size) ;
if (marked == NULL)   {
printf  (“Cannot allocate enough memory\n”) ;
MPI_Finalize () ;
exit  (1) ;
}
```

©BOROVSKA

- Елементите на списъка не са маркирани

```
For  (i = 0;  i < size; i++)  marked [i] = 0;
```

- Започваме с пресяване на числата, степени на 2
- Integer prime е стойността на текущото просто число, за което се извършва пресяването
- Integer index е неговия индекс в масива на процеса 0
- Само процес 0 използва тази променлива

```
if ( !id )  index = 0;
prime = 2;
```

- Имплементираме **repeat…until** в C като цикъл **do…while**
- Всеки процес трябва да маркира в неговия дял от списъка всички числа, кратни на простото число между простото число на квадрат и n
- За тази цел трябва да определим индекса на първото число, което трябва да се маркира
- Ако $prime^2$ е < от най-малката стойност в масива, разликата между тях определя индекса на първия маркиран елемент

©BOROVSKA

- В противен случай, разделяме low_value на prime → ако разликата е 0, low_value е кратно на prime и от него започва маркирането
- Ако разликата е ≠ 0, индексът се определя от първият елемент, кратен на prime

```
if  (prime * prime  >  low_value)
   first = prime * prime – low_value)
Else   {
   if  ( ! (low_value % prime) )  first = 0;
   else  first = prime  -  (low_value % prime) ;
}
```

- Пресяването фактически се осъществява от следния цикъл for
- Всеки процес маркира кратните числа на текущото просто число от първия индекс до края на масива

```
if  ( !id )   {
   while  (marked [++index]) ;
   prime = index + 2;
}
```

- Процес 0 разпръсква стойността на следващото просто число към останалите процеси

```
MPI_Bcast  (&prime, 1, MPI_INT, 0, MPI_COMM_WORLD) ;
```

©BOROVSKA

- Процесите продължават да пресяват дотогава, докато квадрата на текущото просто число е < или = на горната граница

} while (prime * prime <= n);

- Всеки процес преброява простите числа в неговия дял от масива

```
count = 0 ;
for  (i = 0; i < size;  i++)
   if  (!marked [i] )  count++;
```

- Общата сума на процесите се съхранява в глобална променлива global_count в процес 0

MPI_Reduce (&count, &global_count, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD) ;

©BOROVSKA

- Спираме таймера; elapsed_time съдържа броя на секундите за изпълнението на алгоритъма без да се отчите MPI startup

```
elapsed_time  += MPI_Wtime () ;
```

- Процес 0 отпечатва броя на простите числа и изминалото време

```
if  ( !id )  {
   printf (“%d primes are less than or equal to
   %d\n”, global_count, n);
   printf (“Total elapsed time:  %10.6f\n,
   elapsed_time) ;
}
```

- Извикване на MPI_Finalize

ПОДОБРЕНИЯ

1. ***ИЗТРИВАНЕ НА ЧЕТНИТЕ ЧИСЛА***

- Единственото просто четно число е 2 – излишно е да се ползват половината от булевите стойности
- Обработват се само нечетните числа – намалява се необходимата памет наполовина и се ускоряват изчисленията 2 пъти
- Изискванията за комуникация са същите

2. ***ЕЛИМИНИРАНЕ НА BROADCAST***

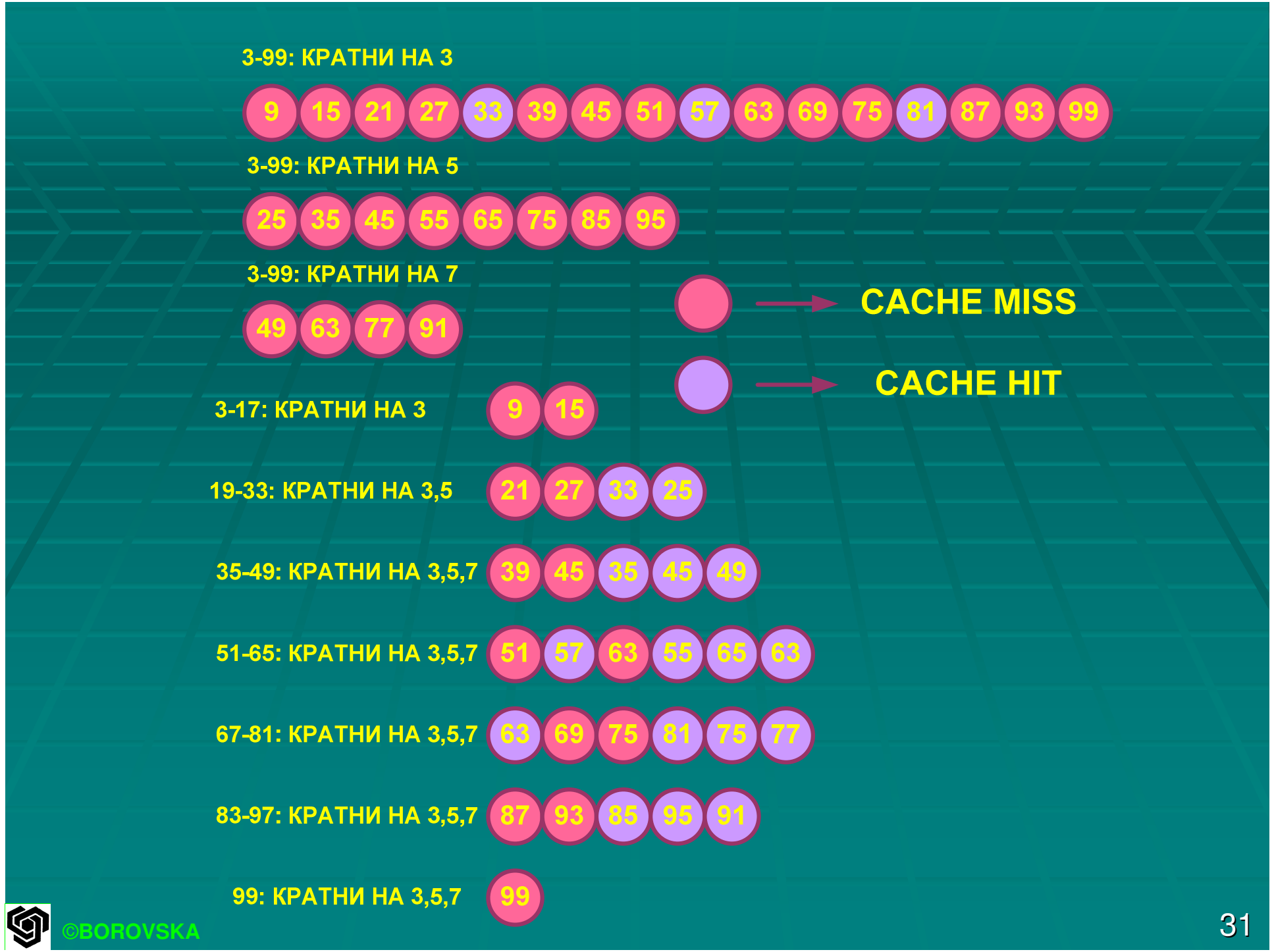
- При изпълнението на програмата процес 0 многократно определя новата стойност на текущото кратно число и я изпраща към останалите процеси
- Всеки процес може сам да си определя новата стойност на текущото кратно число
- Преди намиране на простите числа от 3 до n, всеки процес използва последователен алгоритъм за простите числа от 3 до sqrt(n) – индивидуално копие

3. РЕОРГАНИЗИРАНЕ НА ЦИКЛИТЕ

- Всеки процес маркира силно разпръснати елементи на голям масив – ниска скорост на попадения в кеша *(cache hit rate)*
- Итерации на външен цикъл – пресява простите числа между 3 и sqrt(n)
- Итерации на вътрешен цикъл – пресява простите числа между 3 и n
- Разменяме местата на двата цикъла и подобряваме скоростта на попадения в кеша

- Промяната на реда, в който се маркират съставните числа, могат значително да повишат скоростта на попадения в кеша
- Пример: търсим простите числа между 3 и 99
- Нека кеша има 4 линии, като всяка линия съдържа 4 байта
- Една линия съдържа байтове, представящи 3,5,7 и 9
- Следващата линия съдържа байтове, представящи 11, 13, 15 и 17

- Пресяват се всички съставни числа за текущото просто число, преди да се разгледа следващото просто число
- Тъмните кръгове представят липсите в кеша
- Когато алгоритъмът отново се обръща към малките числа, те липсват в кеша
- Пресяване на няколко прости числа от съдържанието на кеша – подобрява се скоростта на попаденията в кеша


3-99: КРАТНИ НА 3
9 15 21 27 33 39 45 51 57 63 69 75 81 87 93 99
3-99: КРАТНИ НА 5
25 35 45 55 65 75 85 95
3-99: КРАТНИ НА 7
49 63 77 91
CACHE MISS
CACHE HIT
3-17: КРАТНИ НА 3
9 15
19-33: КРАТНИ НА 3,5
21 27 33 25
35-49: КРАТНИ НА 3,5,7
39 45 35 45 49
51-65: КРАТНИ НА 3,5,7
51 57 63 55 65 63
67-81: КРАТНИ НА 3,5,7
63 69 75 81 75 77
83-97: КРАТНИ НА 3,5,7
87 93 85 95 91
99: КРАТНИ НА 3,5,7
99

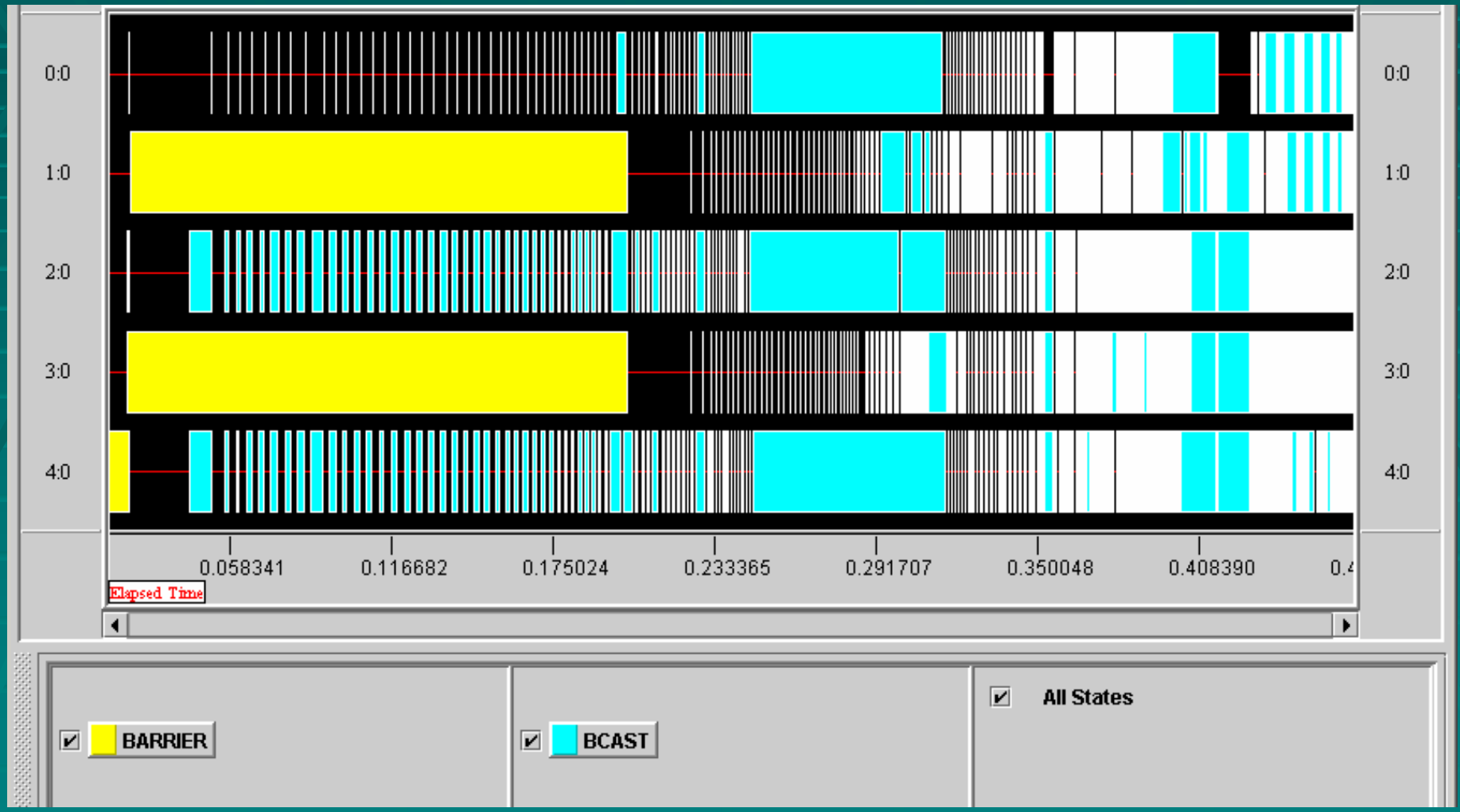
©BOROVSKA


Оригинална паралелна програма
Паралелна програма без запис на четните числа
Паралелна програма без broadcast
Паралелна програма с разместване на циклите за подобряване на скоростта на попадение в кеша
ВРЕМЕ (SEC)
25.6
22.4
19.2
16.0
12.8
9.6
6.4
3.2
1
2
3
4
5
6
7
8

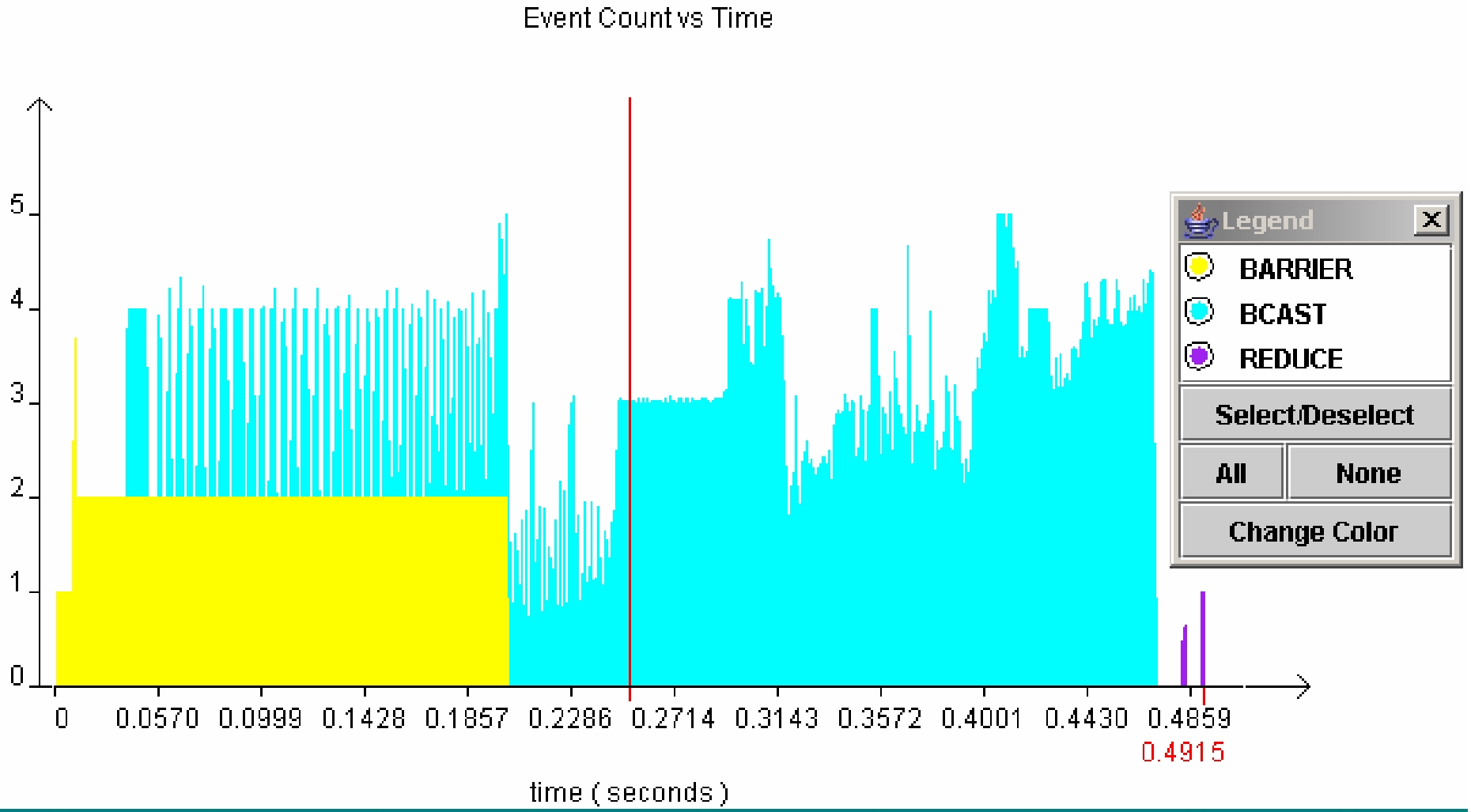
©BOROVSKA

- Основаваме се на последователния алгоритъм
- За идентифициране на паралелизма използваме domain decomposition methodology
- Използваме блоково разпределение на елементите на масива между процесите
- Процес 0 разпръсква текущото просто число (функция MPI_Bcast)
- Програмата постига добра производителност при масив от 100 млн. числа при изпълнение на commodity cluster

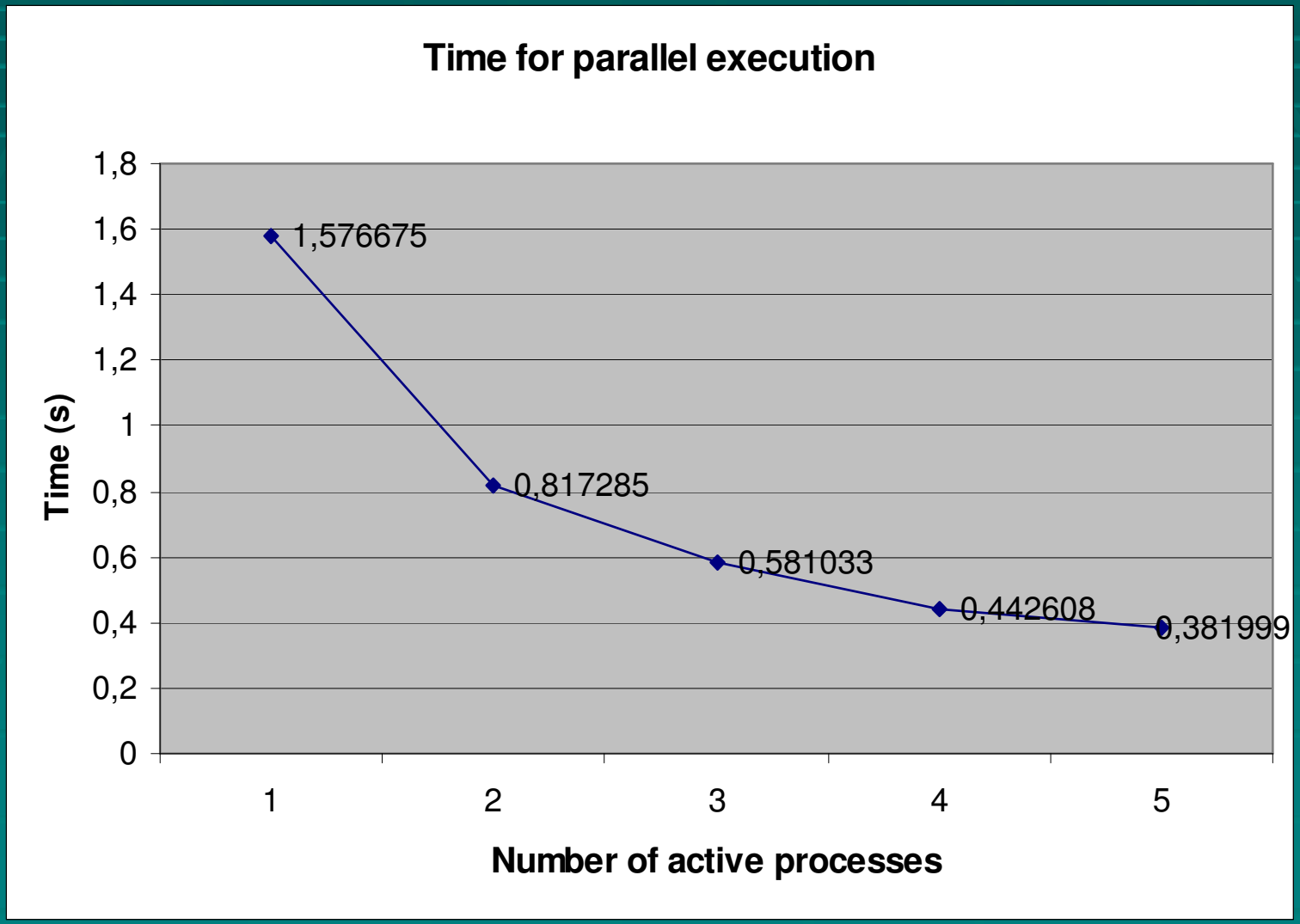
Профилиране на паралелизмите



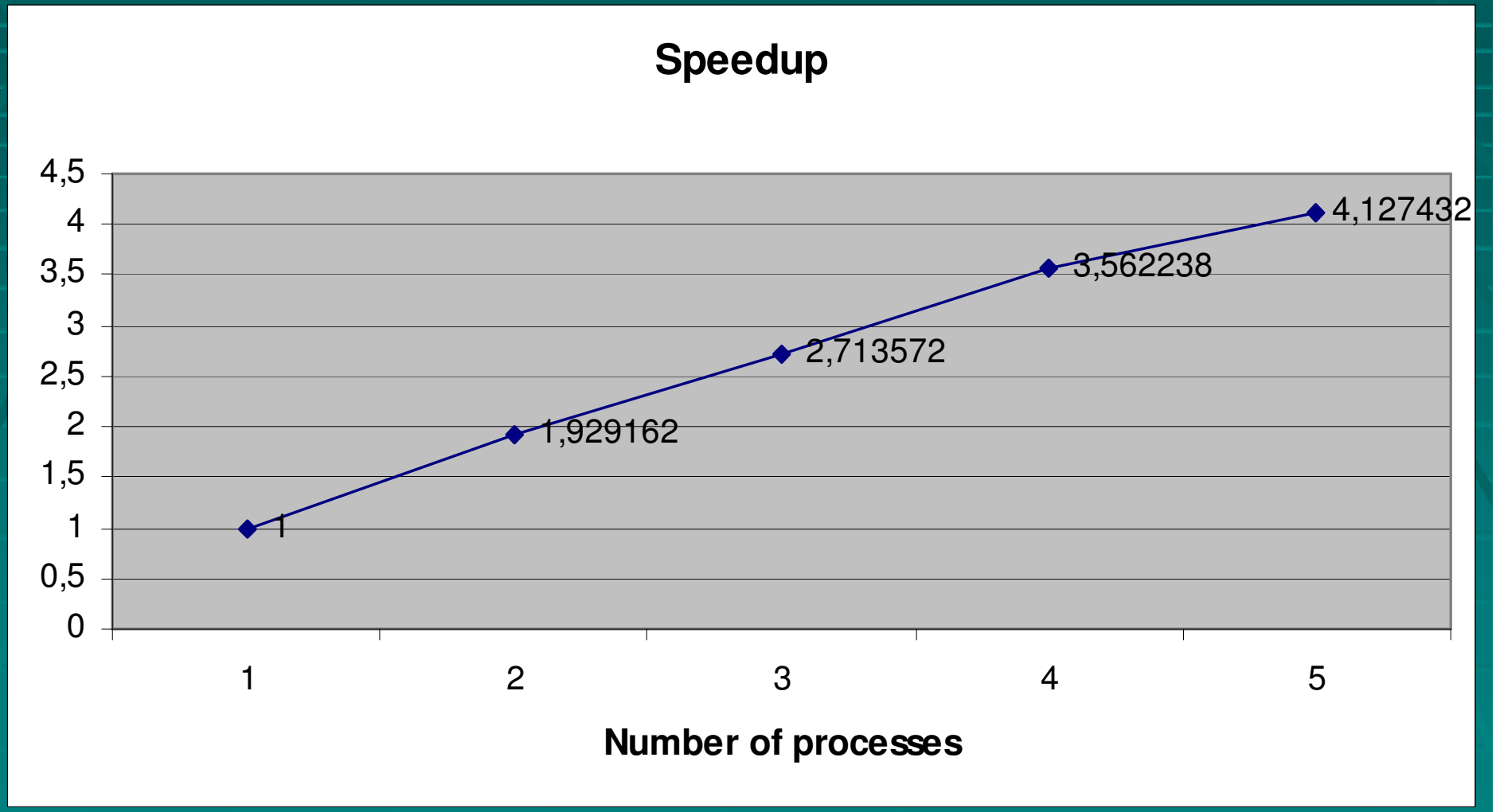
Профилиране на паралелизмите



Оценка на производителността



Оценка на производителността



КРАЙ