Паралелно Програмиране

Дизайн на паралелни програми

доц. д-р Александър Пенев

Първи Стъпки

Първи стъпки

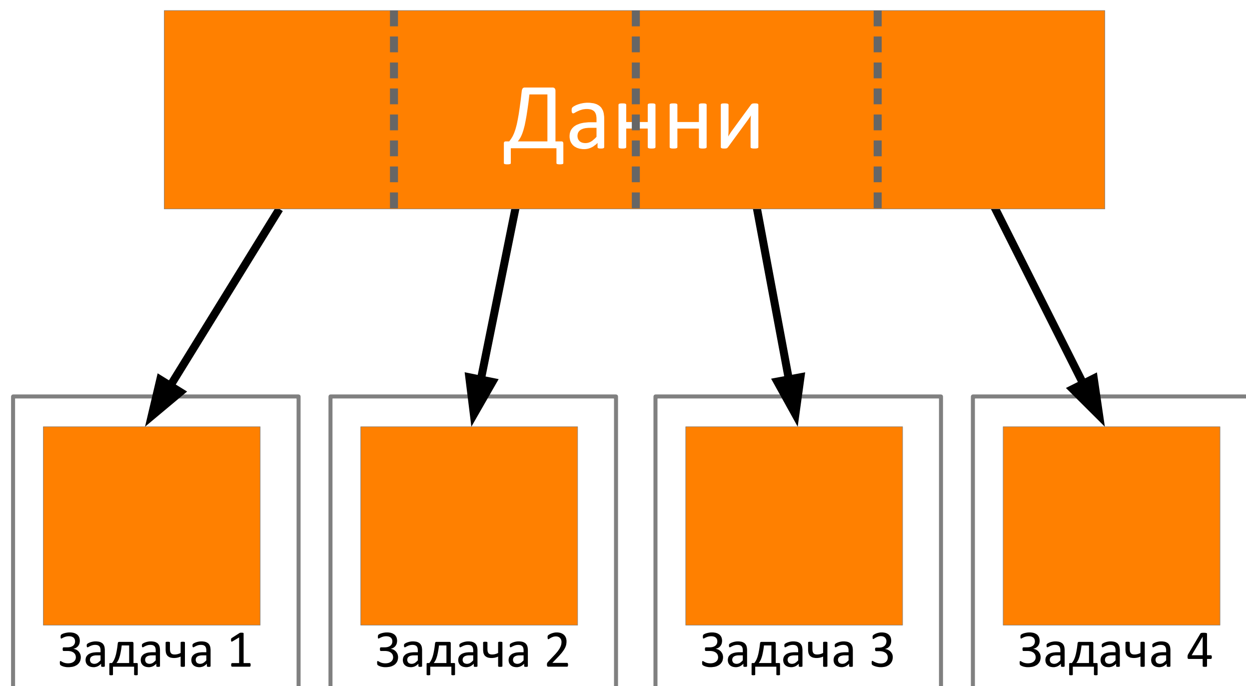
- **Разбиране на проблема**
  Подробно запознаване с проблема, който ще се решава „паралелно”. Ако имаме вече написана „серийна“ (не паралелна) програма, която ще преработваме тябва добре да разберем и кода;
- **Преди да инвестираме време в преработка**
  Добре е да направим груба оценка на това, какво може да се постигне в най-добрия случай т.е. дали проблема може да е паралелен като ray tracer или е по-скоро като „Фибоначи“?
- **Определяне на т.нар. „горещи точки“ (hotspots) на програмата**
  Това са местата на програмата където се извършва най-много „работа“. В повечето случаи това са няколко места в програмата. Може да се използват profiler-и и анализ на производителността за да ги открием, след което да се фокусираме в разпаралеляването точно на тях;
- **Определяне на т.нар. „тесни места“ (bottlenecks) в програмата**
  Има ли области, които са непропорционално бавни или предизвикват спиране или отлагане на паралелна работа? Например В/И обикновено е нещото, което забавя програмата. Възможно е да се преструктурира програмата или да се използва различен алгоритъм за намаляване или премахване на ненужните бавни области;
- **Идентифицирайте пречките пред паралелизма**
  Един общ клас пречки са данновите зависимости. Това беше показано от примера с пресмятането на числата на Фибоначи;

Първи стъпки

- Проучете други алгоритми, ако е възможно
  Това може да е най-голямо значение при проектирането на паралелно приложение;
- Възползвайте се от оптимизиран софтуер за трети страни
  Съществуват много добре оптимизирани и силно паралелни библиотеки – математически библиотеки, налични от водещи доставчици (IBM ESSL, MKL на Intel, AMCL на AMD и др.);
  Фреймурци, библиотеки с паралелни структури от данни и алгоритми и др.;

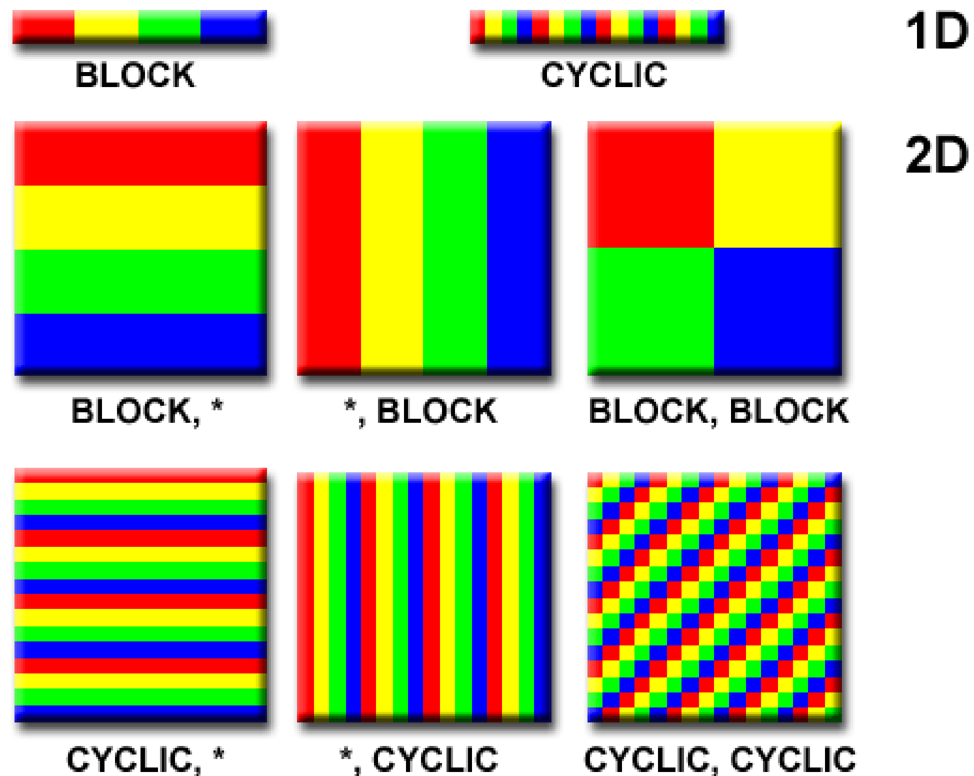
Как ще се Декомпозират Данните

Разбиване на данните при даннов паралелизъм



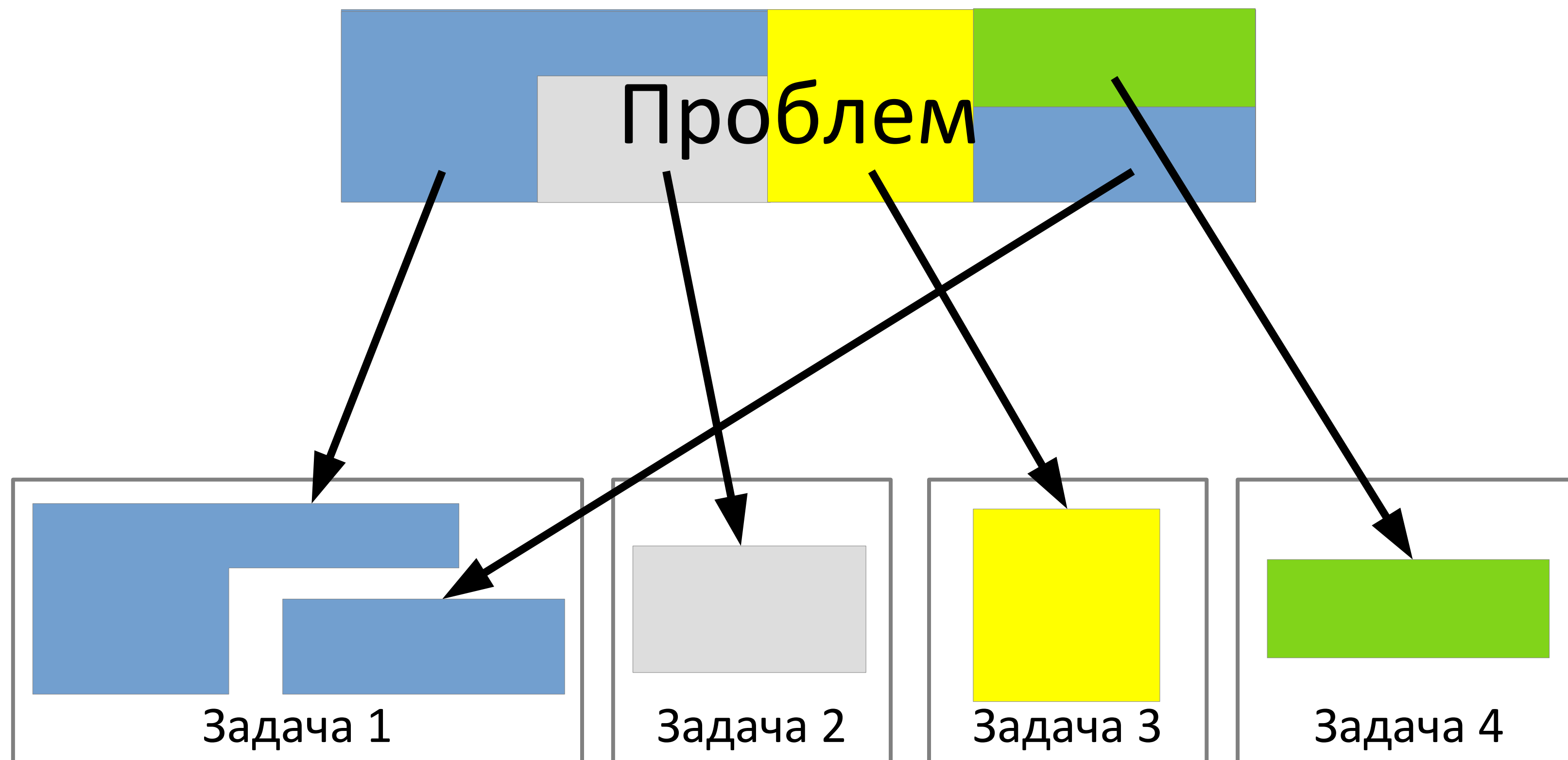
Видове декомпозиция на данни

- 1D, 2D, 3D, ...;
- Равномерно и Неравномерно;
- Статично и Динамично;
- Блоково или Циклично;



Как ще се Декомпозира Алгоритъма

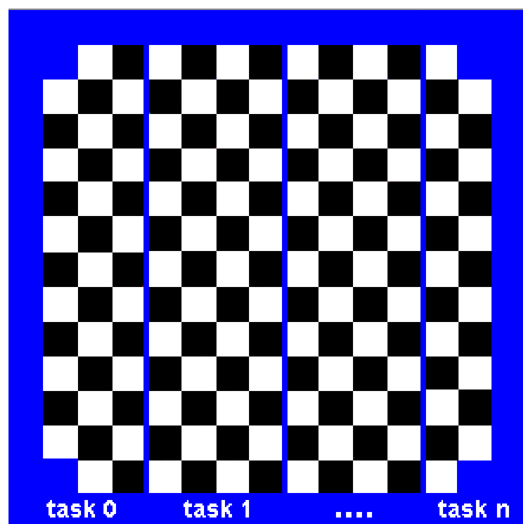
Декомпозиране на алгоритъма

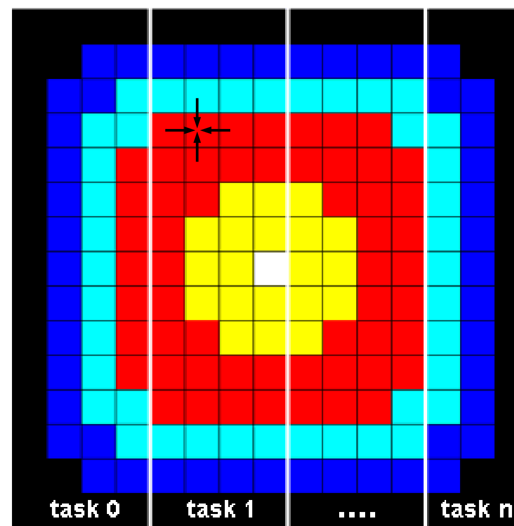


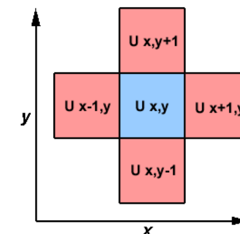
Видове Комуникация Между Подзадачите

Комуникация между подзадачите

- Без комуникация – например да обърнем цвета от черен в бял и от бял в черен за всяка точка
  Това не е свързано с днннова зависимост между различните части на изображението и това позволява да разбием данните на и да обработим парчетата независимо едно от друго;
- С комуникация – например да направим Gausian Blur на дадено изображение
  При разбиване на изображението, може да обработим всяко парче паралелно с другите, но при някои точки (граничните) има зависимост от данните на съседните парчета;







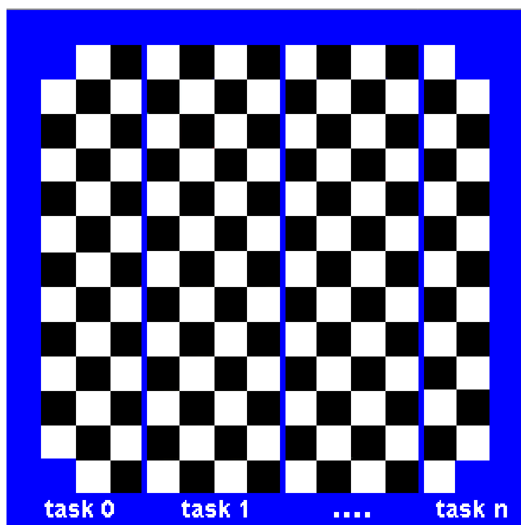
Комуникация между подзадачите

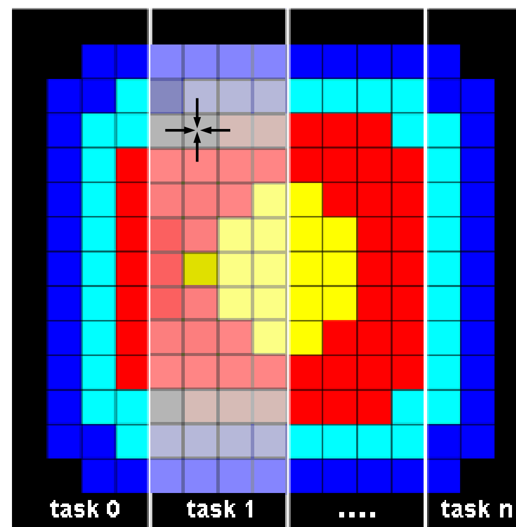
- Без комуникация – например да обърнем цвета от черен в бял и от бял в черен за всяка точка

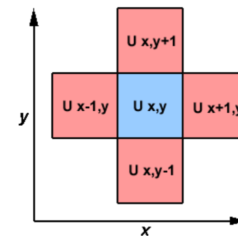
  Това не е свързано с днннова зависимост между различните части на изображението и това позволява да разбием данните на и да обработим парчетата независимо едно от друго;

- С комуникация – например да направим Gausian Blur на дадено изображение

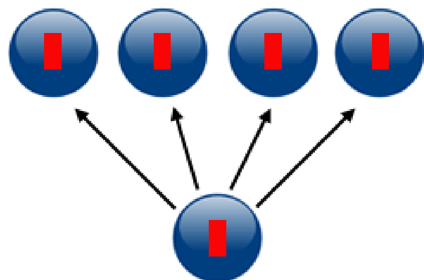
  При разбиване на изображението, може да обработим всяко парче паралелно с другите, но при някои точки (граничните) има зависимост от данните на съседните парчета;



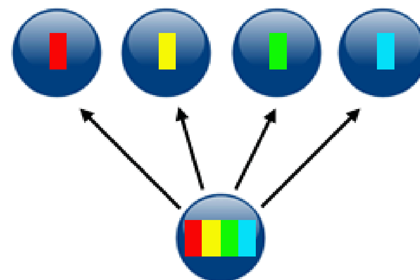




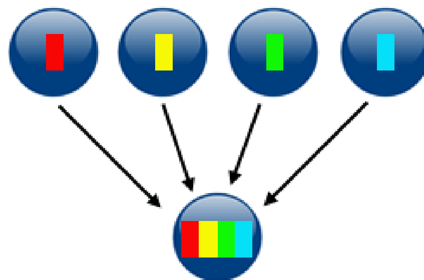
Комуникация между подзадачите



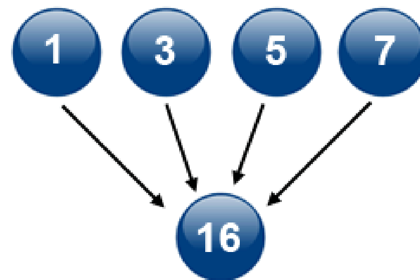
Разпръскване
(Broadcast)



Разпределяне
(Scatter)



Събиране
(Gather)



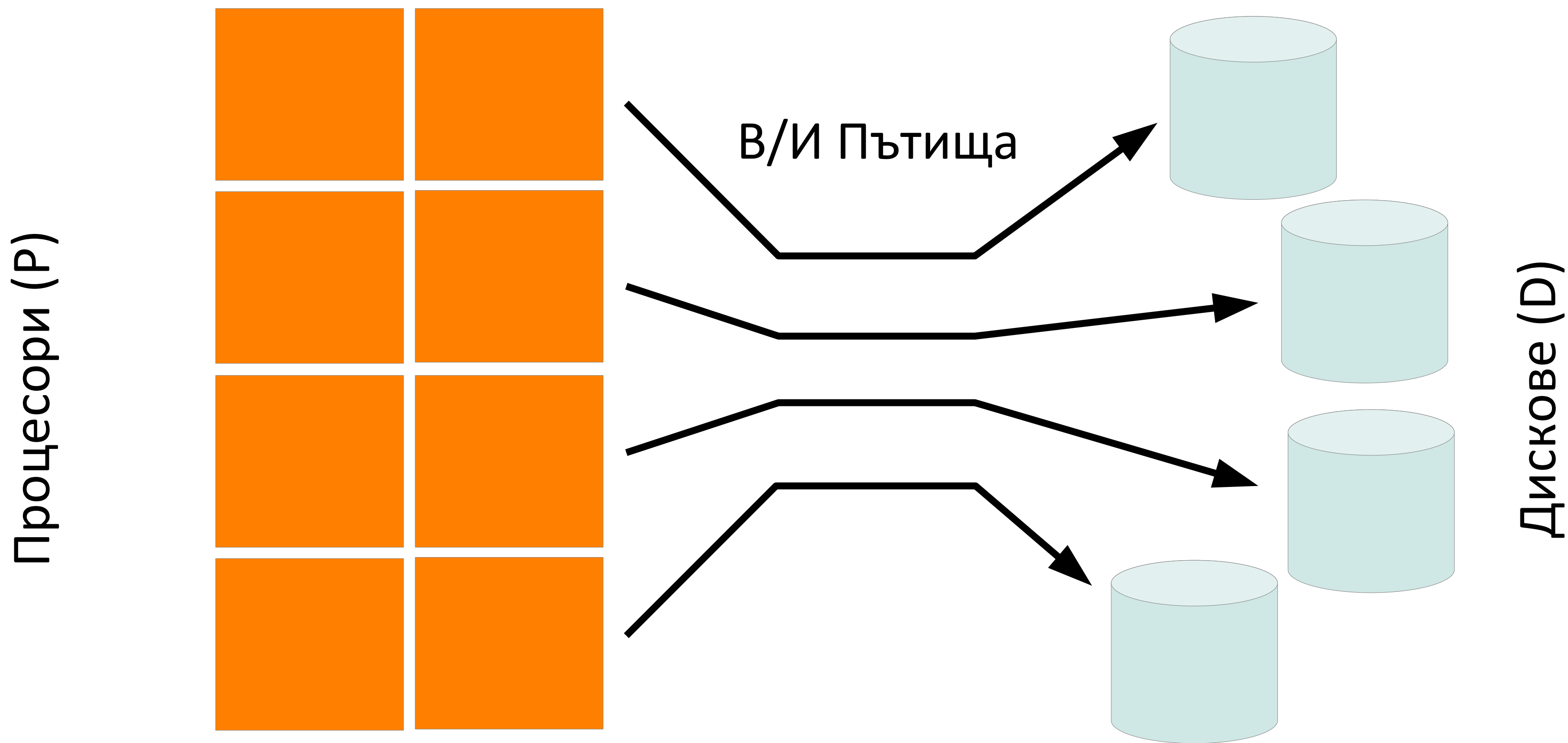
Редукция
(Reduction)

Комуникация между подзадачите

- **Разпръскване (Broadcast)**
  При този тип комуникация между задачите (обикновено една) задача предава едни и същи данни/команди към всички подзадачи, които те в последствие трябва да обработят/изпълнят;
- **Разпределяне (Scatter)**
  При този тип комуникация между задачите (обикновено една) задача разпределя данните/командите на части и предава всяка част на различна подзадача;
- **Събиране (Gather)**
  Обработените данни се връщат от всички подзадачи на главната (обикновено на тази, която е разпределила преди това работата) и от тях се формира общия резултат. Това обикновено става без съществени допълнителни обработки;
- **Редукция (Reduction)**
  Резултатите от обработените данни се връщат от всички подзадачи на главната (обикновено на тази, която е разпределила преди това работата) и от тях се формира общия резултат чрез т.нар. Редукция – сравнително прост или по-сложен алгоритъм (най-често) обобщаващ крайния резултат на базата на под резултатите. Например, ако търсим максимум в голям масив от данни, те може да се разпределят на части и всяка подзадача да търси максимума в своята част от данните, след това максимумите се редуцират в главната задача, като максимум от тези максимуми. Много често се формират дървета на редукция т.е. редукция на повече нива;
- **Други**

Вход / Изход

Вход / Изход



Вход / Изход

- Проблеми:
  - В/И операциите обикновено са пречещи на паралелизма;
  - Операциите за В/И изискват по-голямо количество време от операциите с оперативната памет;
  - Паралелните В/И системи може да не са добре реализирани или или да не са налични за всички платформи;
  - Операциите по запис могат да доведат до презаписване на файловете, когато всички задачи имат достъп до едно и също файлово пространство;
  - Операциите за четене могат да бъдат повлияни от способността на файловия сървър да обработва няколко заявки за четене едновременно;
  - В/И, които минава през мрежата (NFS, не-локални), могат да причинят сериозни проблеми;

Вход / Изход

- Налични са паралелни файлови системи. Например:
  - GPFS: Паралелна файлова система (IBM). Сега се нарича IBM Spectrum Scale;
  - Luster: за Linux клъстери (Intel);
  - HDFS: Разпределена файлова система Hadoop (Apache);
  - PanFS: Файлова система Panasas ActiveScale за Linux клъстери (Panasas, Inc.);
  - и др.
- Спецификацията на паралелния В/И интерфейс за MPI е налична като част от MPI-2;

Вход / Изход

- Препоръки:
  - Намалете общия В/И максимално;
  - Ако имате достъп до паралелна файлова система, използвайте я;
  - Писането на големи парчета данни е по-ефективно от писането на малки;
  - Няколко по-големи файлове е по-добре от много по-малки файлове;
  - Ограничете В/И в една или няколко серийни части на програмата. След това използвайте паралелни комуникации за разпределение на данните към паралелните подзадачи. Това е валидно и за записа;
  - Ограничете В/И в една или няколко подзадачи;

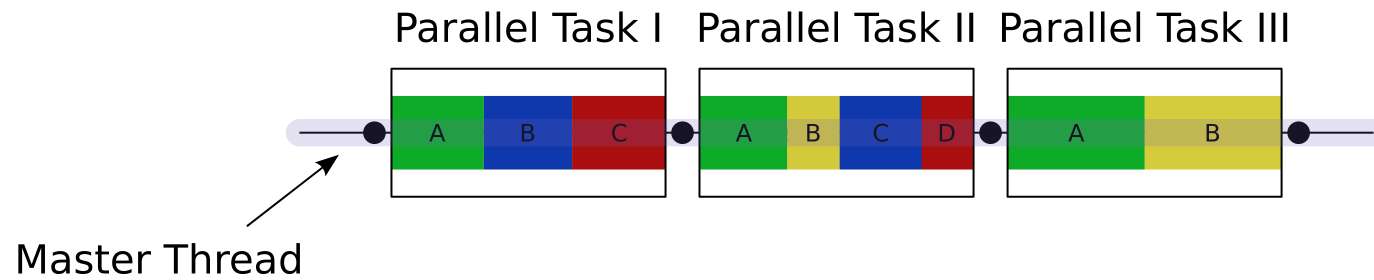
Добри Практики

Fork-Join

Fork-Join – пример

```
mergesort(A, lo, hi):
  if lo < hi:
    mid = ⌊lo + (hi - lo) / 2⌋
    fork mergesort(A, lo, mid) // process in parallel
    mergesort(A, mid, hi)      // main task handles second
    join
  merge(A, lo, mid, hi)
```

Fork-Join



Parallel Task I

Parallel Task II

Parallel Task III

Master Thread

A

B

C

A

B

C

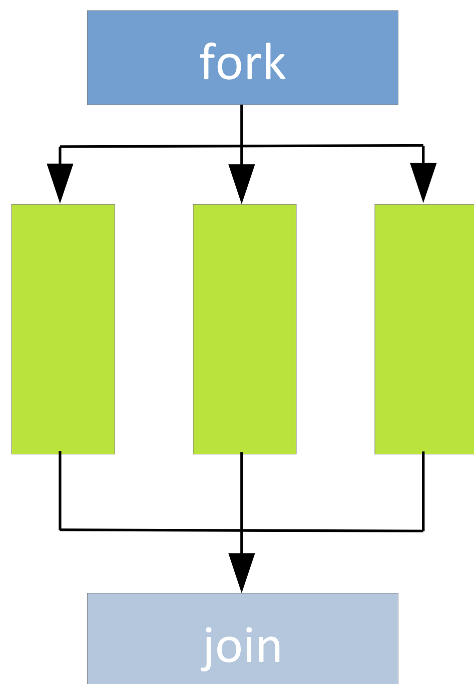
D

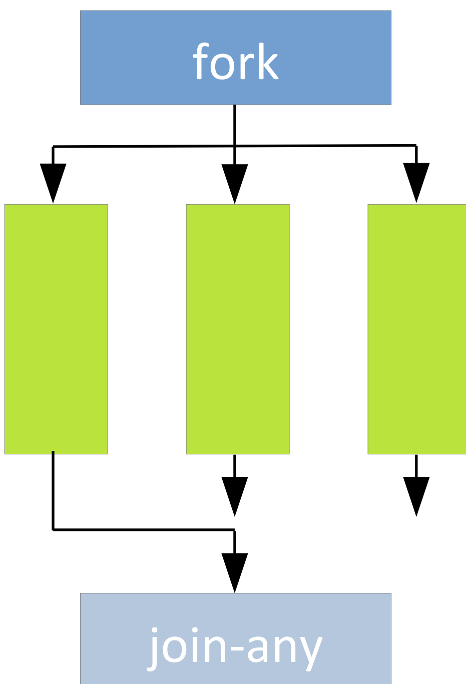
A

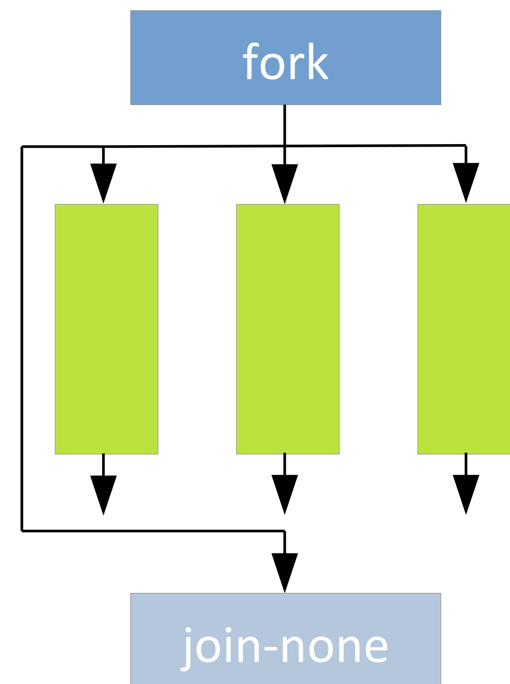
B

Fork-Join

- fork-join;
- fork-join-any;
- fork-join-none;





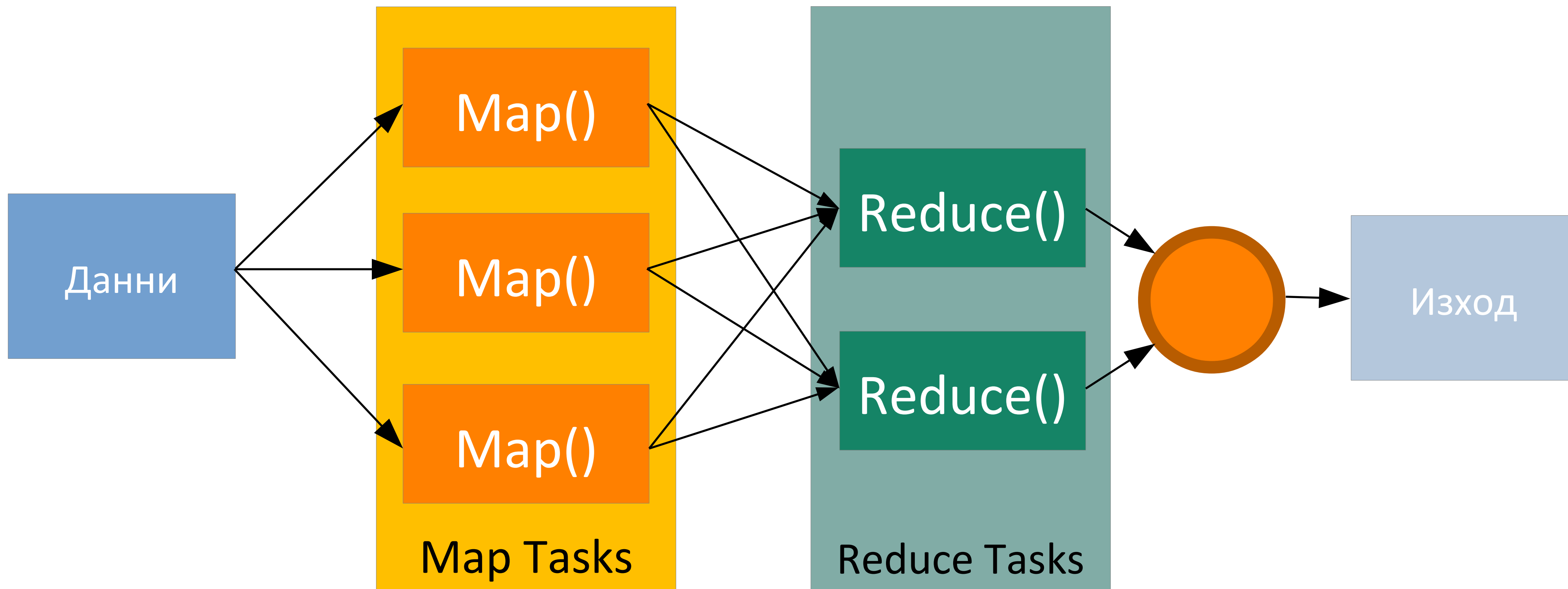


Fork-Join

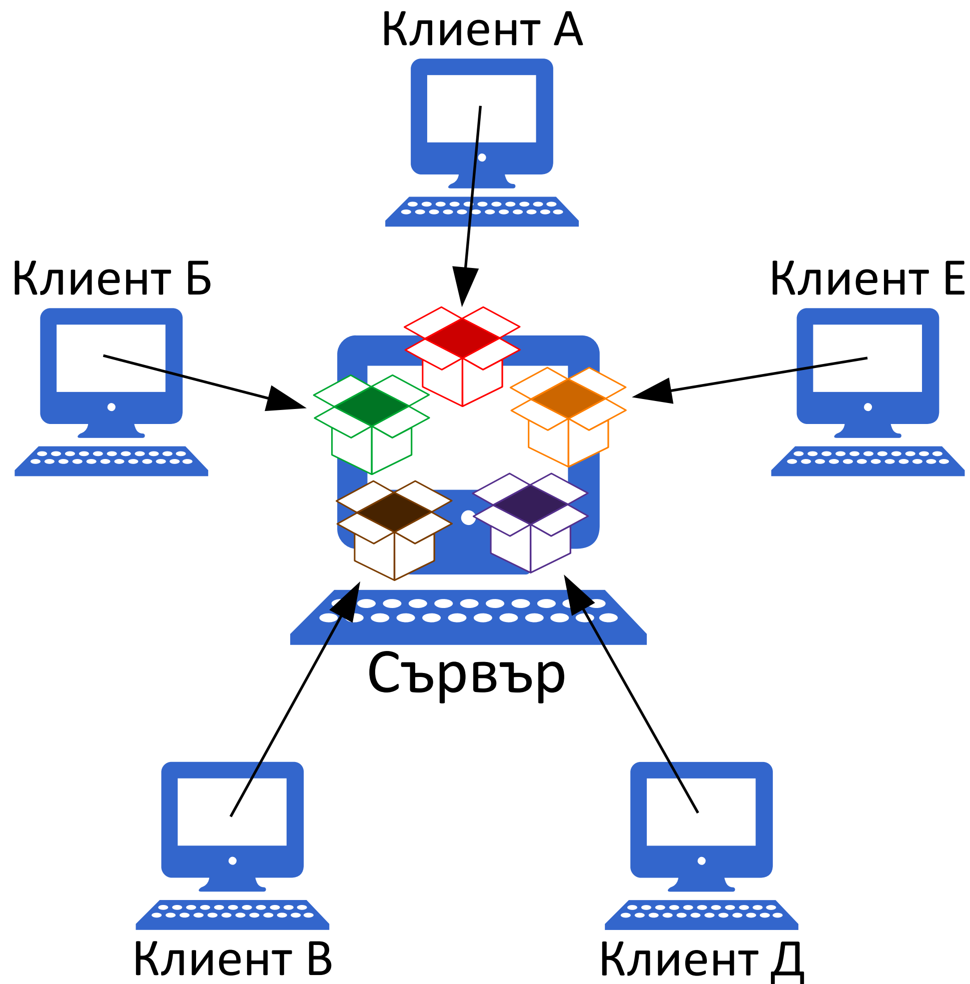
- OpenMP;
- Java concurrency framework;
- Task Parallel Library for .NET;
- Intel's Threading Building Blocks (TBB);
- Cilk;
- И др.

MapReduce

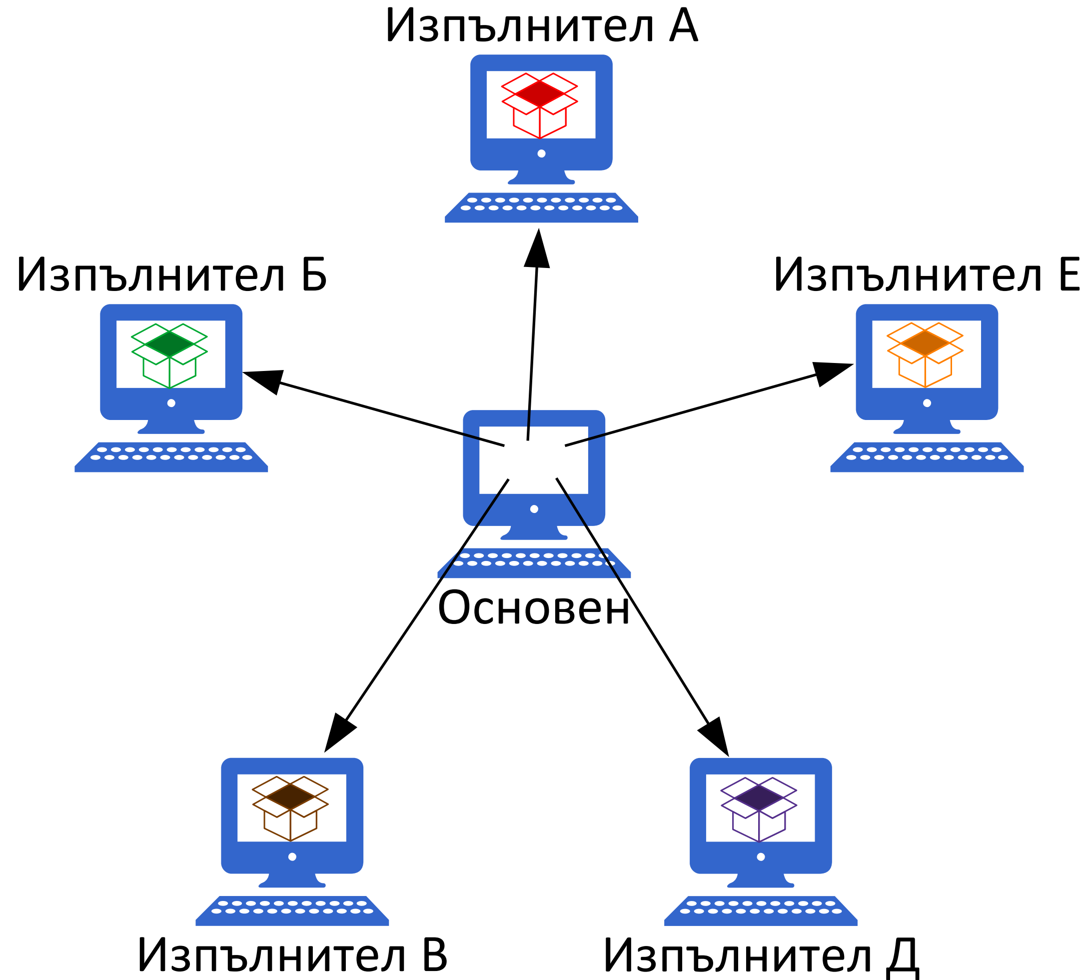
MapReduce



Класическа схема на обработка на данни



MapReduce подход



Пример – броене на думи (традиционен подход)

Входен файл

Deer Bear River
Car Car River
Deer Car Bear
...

MapReduce броене

Вход

Splitting (разделяне)	Mapping (мапинг)	Shuffling (пренареждане)	Reduce (редукция)
K1, V1	List(K2, V2)	K2, List(V2)	K3, V3

Deer Bear River
Car Car River
Deer Car Bear...

Deer Bear River → Deer, 1 / Bear, 1 / River, 1

Car Car River → Car, 1 / Car, 1 / River, 1

Deer Car Bear → Deer, 1 / Car, 1 / Bear, 1

Bear, (1,1) → Bear, 2

Car, (1,1,1) → Car, 3

Deer, (1,1) → Deer, 2

River, (1,1) → River, 2

List(K3, V3)

Bear, 2
Car, 3
Deer, 2
River, 2

Краен резултат

Входен файл

Key	Value
0	Deer Bear River
121	Car Car River
226	Deer Car Bear
…	…

MapReduce – пример

```
public static class Map extends
               Mapper<LongWritable,Text,Text,IntWritable> {
  public void map(LongWritable key, Text value,
                  Context context)
              throws IOException,InterruptedException {
    String line = value.toString();
    StringTokenizer tokenizer = new StringTokenizer(line);
    while (tokenizer.hasMoreTokens()) {
      value.set(tokenizer.nextToken());
      context.write(value, new IntWritable(1));
    }
  }
}
```

MapReduce – пример

```
public static class Reduce extends
                Reducer<Text,IntWritable,Text,IntWritable> {
  public void reduce(Text key, Iterable<IntWritable> values,
                          Context context)
                  throws IOException,InterruptedException {
    int sum=0;
    for(IntWritable x: values) {
      sum += x.get();
    }
    context.write(key, new IntWritable(sum));
  }
}
```

MapReduce – пример

```
public static void main(String[] args) throws Exception {
  Configuration conf = new Configuration();
  Job job = new Job(conf, "My Word Count Program");
  job.setJarByClass(WordCount.class);
  job.setMapperClass(Map.class);
  job.setReducerClass(Reduce.class);
  job.setOutputKeyClass(Text.class);
  job.setOutputValueClass(IntWritable.class);
  job.setInputFormatClass(TextInputFormat.class);
  job.setOutputFormatClass(TextOutputFormat.class);
  Path outputPath = new Path(args[1]);
  // Configuring the I/O path from the job's filesystem
  FileInputFormat.addInputPath(job, new Path(args[0]));
  FileOutputFormat.setOutputPath(job, new Path(args[1]));
}
```

Hotspots и Bottlenecks

Hotspots

- Това са местата в програмата, които отнемат най-много ресурси (най-често това е ресурса време);
- Инструментите за анализ на производителността на софтуер помагат за лесното им откриване;
- В тях трябва да се насочат основите усилия за оптимизация на програмата, както и при възможност да векторизираме и/или разпаралим тези части на алгоритъма;
- На следващите слайдове е показан част от анализа на проста програма за умножение на матрици. Използван е софтуерът за анализ Intel VTune Amplifier;

Hotspots

/home/student1/intel/amplxe/projects/matrix - Intel VTune Amplifier XE 2011

File View Help

r003lh

Lightweight Hotspots - Hotspots

Intel VTune Amplifier XE 2011

Analysis Target | Analysis Type | Collection Log | Summary | Bottom-up | Module Timeline

Elapsed Time: 108.382s

CPU Time:	107.877s
Instructions Retired:	8,600,000,000
CPI Rate:	33.442

The CPI may be too high. This could be caused by issues such as memory stalls, instruction starvation, branch misprediction or long latency instructions. Explore the other hardware-related metrics to identify...

Paused Time:	0s

Top Hotspots

This section lists the most active functions in your application. Optimizing these hotspot functions typically results in improving overall application performance.

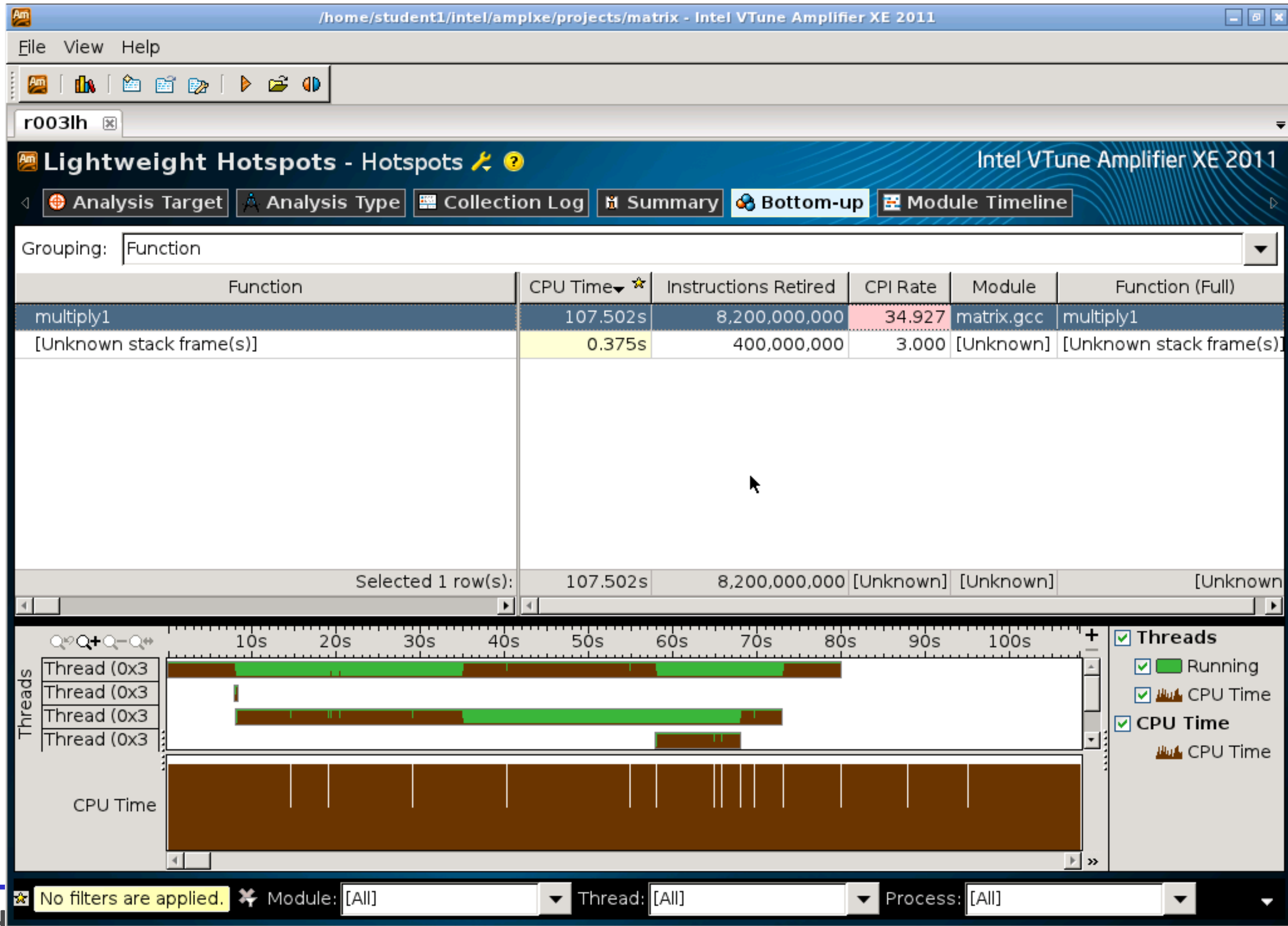
Function	CPU Time
multiply1	107.502s
[Unknown stack frame(s)]	0.375s

Collection and Platform Info

This section provides information about this collection, including result set size and collection platform data.

Command Line:	/opt/intel/vtune_amplifier_xe/samples/en/C++/matrix/linux/matrix.gcc
Frequency:	2.666 GHz
Logical CPU Count:	8
User Name:	student1
Operating System:	Linux
Computer Name:	performance-tunning
Result Size:	1 MB

Hotspots



Hotspots

/home/student1/intel/amplxe/projects/matrix - Intel VTune Amplifier XE 2011

File View Help

r003lh

Lightweight Hotspots - Hotspots Intel VTune Amplifier XE 2011

Analysis Target | Analysis Type | Collection Log | Summary | Bottom-up | Module Timeline | multiply.c

Source | Assembly

Line	Source	CPU Time	Instructions Retired
30			
31	`void multiply1(int msize, int tidx, int numt, TYPE a[][NUM], TYPE b[][NUM], TYPE`		
32	`{`		
33	`    int i,j,k;`		
34			
35	`// Naive implementation`		
36	`    for(i=tidx; i<msize; i=i+numt) {`		
37	`        for(j=0; j<msize; j++) {`		
38	`            for(k=0; k<msize; k++) {`	0.525s	600,000,000
39	`                    c[i][j] = c[i][j] + a[i][k] * b[k][j];`	106.977s	7,600,000,000
40	`            }`		
41	`        }`		
42	`    }`		
43	`}`		
44	`void multiply2(int msize, int tidx, int numt, TYPE a[][NUM], TYPE b[][NUM], TYPE`		
45	`{`		
46	`    int i,j,k;`		
47			
48	`// Loop interchange`		
	Selected 1 row(s):	106.977s	

No filters are applied. Module: [All] Thread: [All] Process: [All]

Hotspots

/home/student1/intel/amplxe/projects/matrix - Intel VTune Amplifier XE 2011

File View Help

r003lh

Lightweight Hotspots - Hotspots

Intel VTune Amplifier XE 2011

Analysis Target | Analysis Type | Collection Log | Summary | Bottom-up | Module Timeline | multiply.c

Source | Assembly

Line	Source	CPU Time	Instructions Ret
30			
31	void multiply1(int msiz		
32	{		
33	int i,j,k;		
34			
35	// Naive implementation		
36	for(i=tidx; i<msize		
37	for(j=0; j<msiz		
38	for(k=0; k<	0.525s	600,000
39	c[i][j] = c[i][j] + a[i][k] * b[k][j];		
40	}		
41	}		
42	}		
43	}		
44	void multiply2(int msiz		
45	{		
46	int i,j,k;		
47			
48	// Loop interchange		
	Selected 1 row(s):	106.977s	

Address	Line	Assembly	CPU Time	Ins
0x401297	31	xor %eax, %eax		
0x401299	31	nopl %eax, (%rax)		
		Block 6:		
0x4012a0	39	movsdq (%rsi,%rax,1),	0.600s	
0x4012a5	39	add $0x8, %rax		
0x4012a9	39	mulsdq (%r9), %xmm0		
0x4012ae	39	add $0x4000, %r9	105.401s	
0x4012b5	38	cmp %rcx, %rax	0.225s	
0x4012b8	39	addsdq (%r11,%r10,8),		
0x4012be	39	movsdq %xmm0, (%r11,%r	0.975s	
0x4012c4	38	jnz 0x4012a0 <Block 6>	0.300s	
		Block 7:		
0x4012c6	38	add $0x1, %r10		
0x4012ca	37	cmp %rbx, %r10		
0x4012cd	37	jnz 0x401290 <Block 5>		
		Block 8:		
0x4012cf	31	add %edx, %ebp		
0x4012d1	31	add %r12, %r11		
0x4012d4	36	cmp %ebp, %edi		
		Highlighted 6 row(s):	106.977s	

No filters are applied. Module: [All] Thread: [All] Process: [All]

Bottlenecks

- Понякога в реализацията на даден паралелен алгоритъм се допускат (логически) грешки;
- Съществуват множество инстрименти за анализ на работата и производителността на програмите, включително такива, които ни дават възможност да открием проблеми при паралелни програми;
- На следващият слайд е показан анализа на една примерна програма (Ray tracer-a Tachyon), в която има допусната грешка (използвана е критична секция с прекалено голяма общност на заключване – заключва се целия буфер на резулатата при запис в него);

Locks and Waits   Locks and Waits viewpoint (change)   Intel VTune Amplifier XE

Collection Log | Analysis Target | Analysis Type | Summary | Bottom-up | Caller/Callee | Top-down Tree | Tasks and Frames

Grouping: Sync Object / Function / Call Stack

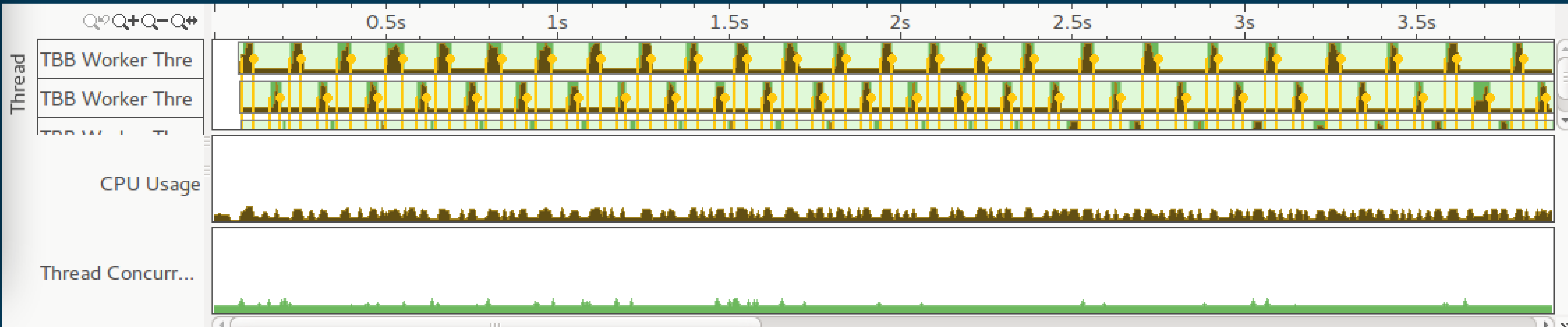
Sync Object / Function / Call Stack	Wait Time by Thread Concurrency (Idle, Poor, Ok, Ideal, Over)	Wai.. Cou.	Spin Time	Mod.	Object Type	Object Creation Module and Function
Mutex 0xed87ef56	65.717s	506	0s		Mutex	tachyon_analyze_locks!thread_trace
Futex 0xcb656872	0.074s	7	0s		%Futex	libtbb.so.2!tbb::internal::rml::private_worker::run
Stream 0x2b9e548	0.000s	2	0s		Stream	libc-2.19.so!__fprintf
Stream dat/balls.dat	0.000s	1	0s		Stream	tachyon_analyze_locks!readmodel
Stream /proc/self/m	0.000s	1	0s		Stream	libpthread-2.19.so!pthread_getattr_np
[Unknown]	0s	0	0.010s			[Unknown]![Unknown]
Selected 1 row(s):	65.717s	506	0s			tachyon_analyze_locks!thread_trace

Wait Time

Viewing 1 of 2 selected

95.1% (62.516s of 65.71

tachyon_analyze...e_locks.cpp
tachyon_analyze_...allel_for.h
libtbb.so.2![TB...scheduler.h:4
libtbb.so.2!tbb:...54 - task.cpp
tachyon_analyze...19 - task.h
tachyon_analyze_...arallel_for
tachyon_analyze_...allel_for.h
tachyon_analyze...e_locks.cpp
tachyon_analyze..._locks.cpp:
tachyon_analyze...ce_rest.cpp



Bottlenecks

- На графиката ясно се вижда, че отделните процеси се изчкват един друг и на практика в даден момент работи само едно ядро на многоядрения процесор;
- Вижда се и къде е изразходвано най-много време за “чакане” и от къде в програмата идва проблема;
- На практика буфера се е общ ресурс и е добре да се заключва при ползване, но алгоритъма е така реализиран, че две нишки никога не пишат в един и същи пиксел в него, така че в случая заключването е напълно излишно. Дори и да има запис в един и същ пиксел то е добре да заклюзваме само него, а не целия буфер;
- След премаване на излишното заключване полуаваме:

Locks and Waits   Locks and Waits viewpoint (change)   Intel VTune Amplifier XE

Collection Log | Analysis Target | Analysis Type | Summary | Bottom-up | Caller/Callee | Top-down Tree | Tasks and Frames

Grouping: Sync Object / Function / Call Stack

Sync Object / Function / Call Stack	Wait Time by Thread Concurrency (Idle, Poor, Ok, Ideal, Over)	Wai.. Cou.	Spin Time	Mod.	Object Type	Object Creation Module and Fur
Futex 0xcb656872	180.100ms	7	0ms		%Futex	libtbb.so.2!tbb::internal::rml::private_
Stream 0x2b9e548a	0.053ms	2	0ms		Stream	libc-2.19.so!__fprintf
Stream dat/balls.dat 0x8cfdb	0.013ms	1	0ms		Stream	tachyon_analyze_locks!readmodel
Stream /proc/self/maps 0xfa4	0.008ms	1	0ms		Stream	libpthread-2.19.so!pthread_getattr_n
[Unknown]	0ms	0	27.996ms			[Unknown]![Unknown]
Selected 1 row(s):	180.100ms	7	0ms			libtbb.so.2!tbb::internal::rml::private_

Wait Time

Viewing 1 of 1 selected

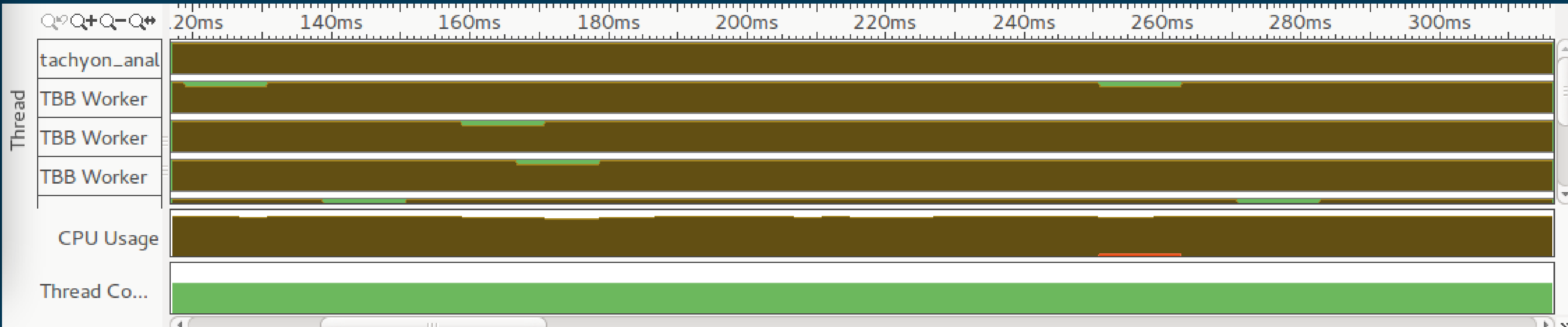
100.0% (0.180s of 0.180

libtbb.so.2!tbb::i...te_server.cp

libtbb.so.2![TBB ...e_server.cp

libpthread-2.19.s...ead_create

libc-2.19.so!__c...6c - clone.S



Неблокиращи Алгоритми и Структури Данни

Блокиране

- Традиционният подход за програмирането с много нишки е използването на примитиви за синхронизиране на достъпа до споделени ресурси;
- Примитиви за синхронизация като мутекси, семафори и критични секции са всички механизми, чрез които програмистът може да гарантира, че определени секции от код не се изпълняват едновременно, ако това би "повредило" споделените структури на паметта;
- Ако една нишка се опита да влезе в критична секция, в която се намира друга нишка, това води до блокиране на опитващата се да влезе в критичната секция, докато другата не я напусне;

Избягване на блокирането

- В някои случаи това може да се избегне;
- Използват се атомарни Read-Modify-Write операции;

Четене-Копиране-Обновяване и др.

- Zero-Copy – Подход, който се използва не само при паралелните приложения, свързан с избягрането от копиране на памет (особено когато става въпрос за големи обеми данни). При него един процес предава данните и собственоста върху тях (“по указател”) на друг без необходимост от заделяне на нови буфери и трансфер на данните от единия буфер в другия;
- RCU – механизъм, чрез който може да се избегне необходимостта от заключване при паралелна работа и достъп до общи структури от данни. Използва се в ядрото на Linux и други ОС, във базите от данни под формата на версионизиране на записисте и др;

Четене-Копиране-Обновяване и др.

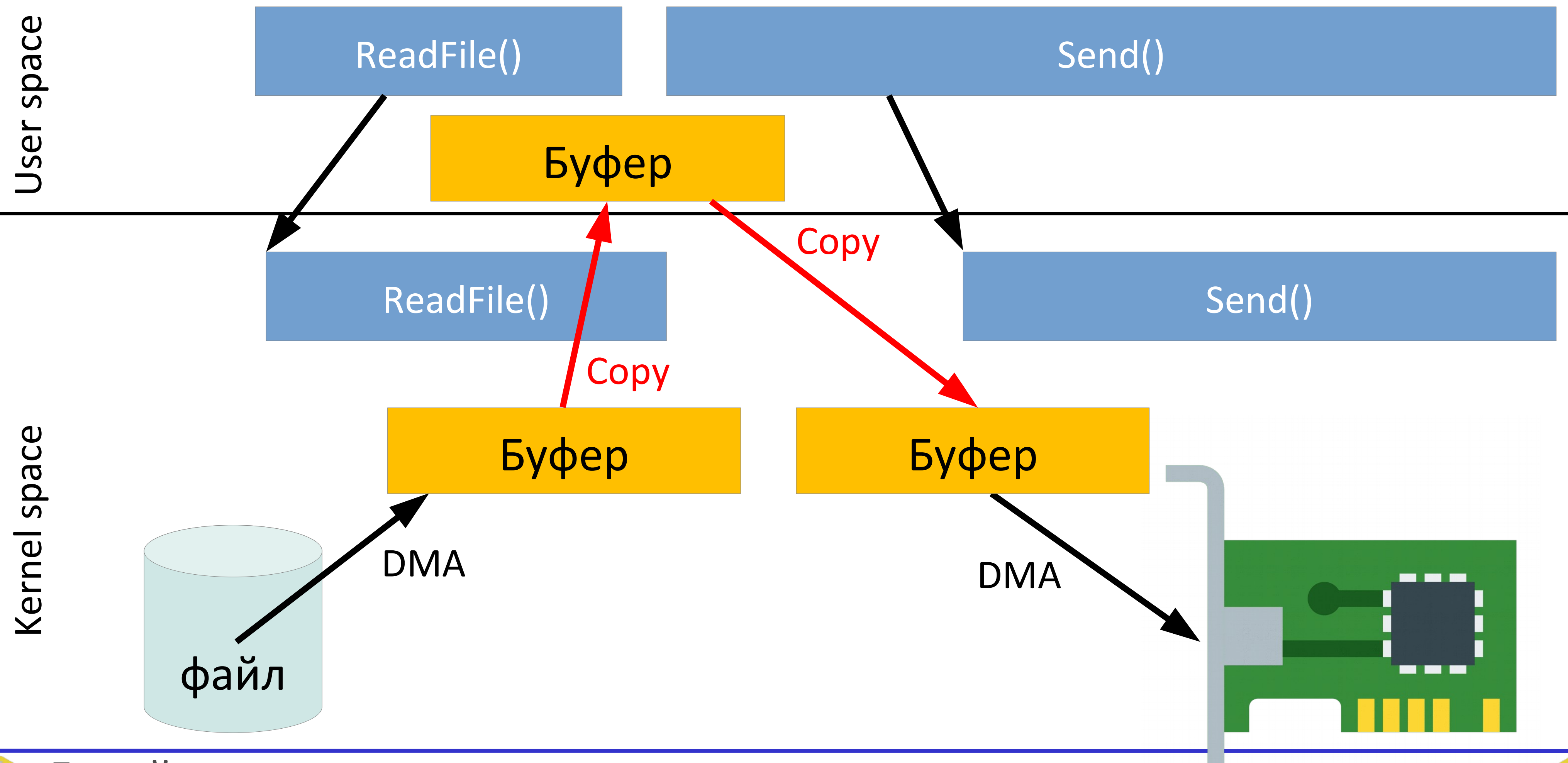
- Транзакционна памет – в съвременните процесори започва да се включват хардуерно ускорена поддръжка на тази концепция. Това позволява работата на паралелните програми, транзакциите в базите данни и др. да се ускорят многократно;
- Неблокиращи алгоритми и структури от данни;

Zero-Copy

Zero-Copy

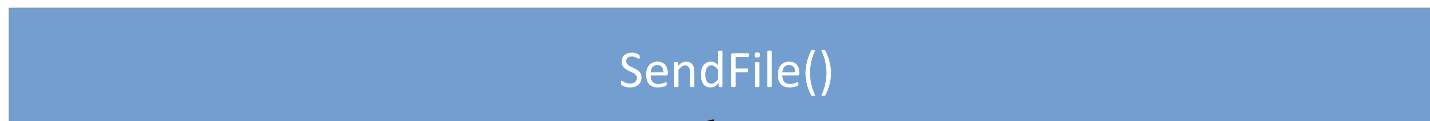
- Това са компютърни операции, при които процесорът не изпълнява задачата да копира данни от една област памет в друга;
- Това често се използва за спестяване на цикли на процесора и честотна лента на паметта (например, при предаване на файл по мрежа, той се чете и изпрща с използването на един буфер, а не се копира от ОС в буфер на прогрмата, след което този буфер да се копира в ОС за да бъде предаван по мрежата – традиционното предаване на файл по мрежата изисква поне две копирания на данните, както и поне две превключвания на контекста между kernel и user пространството);
- Съвременните HSA архитектури правят възможно Zero-Copy между CPU и GPU;

Send File – класически подход

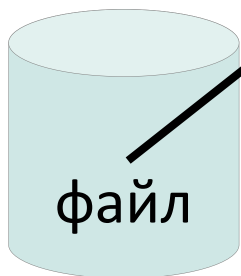


SendFile

User space

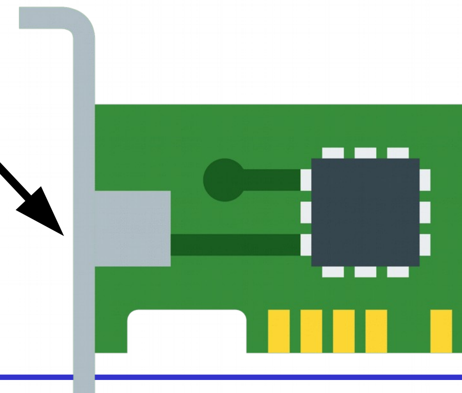


Kernel space

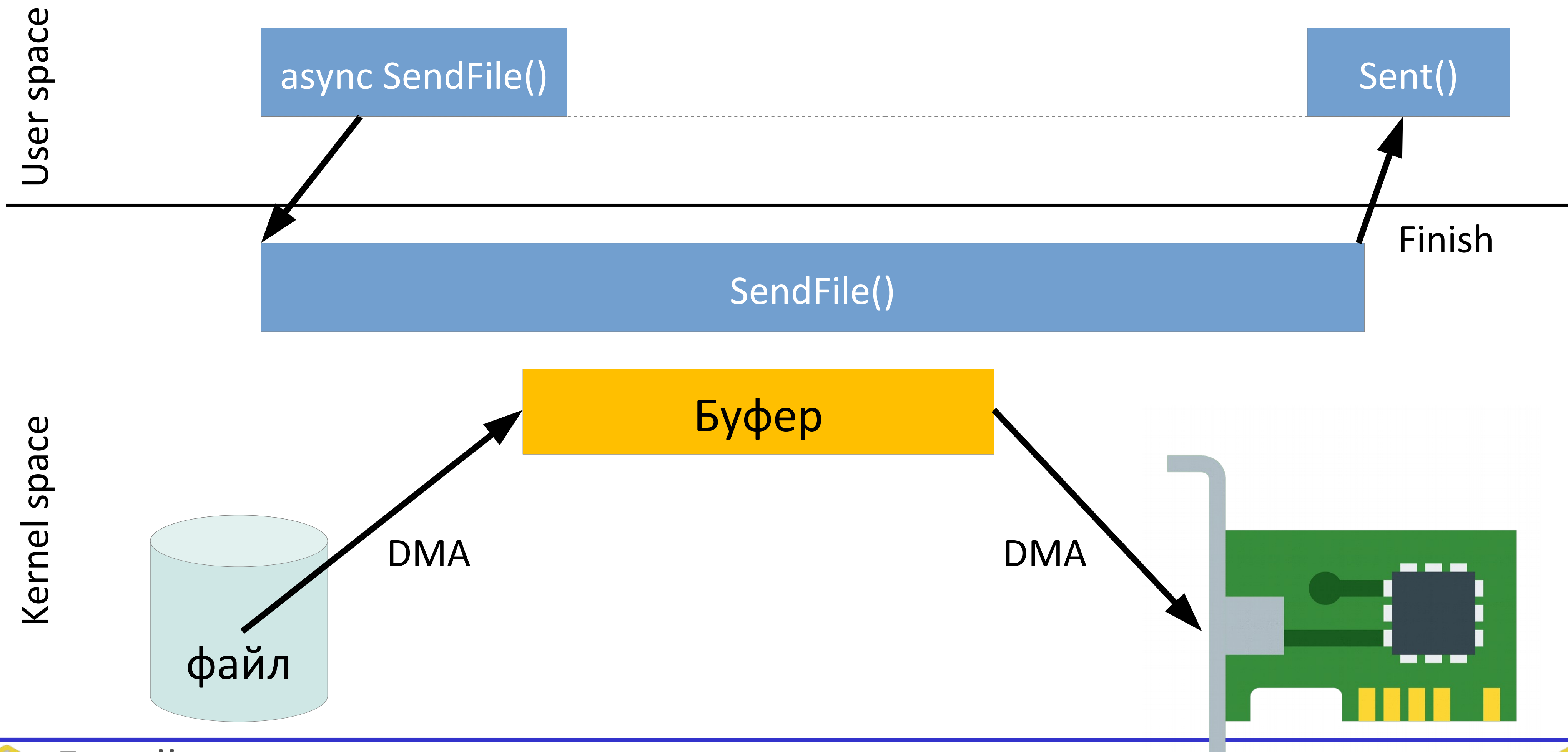


DMA

DMA



SendFile – async



Read-Copy-Update (RCU)

Copy-On-Write (COW)

и др.

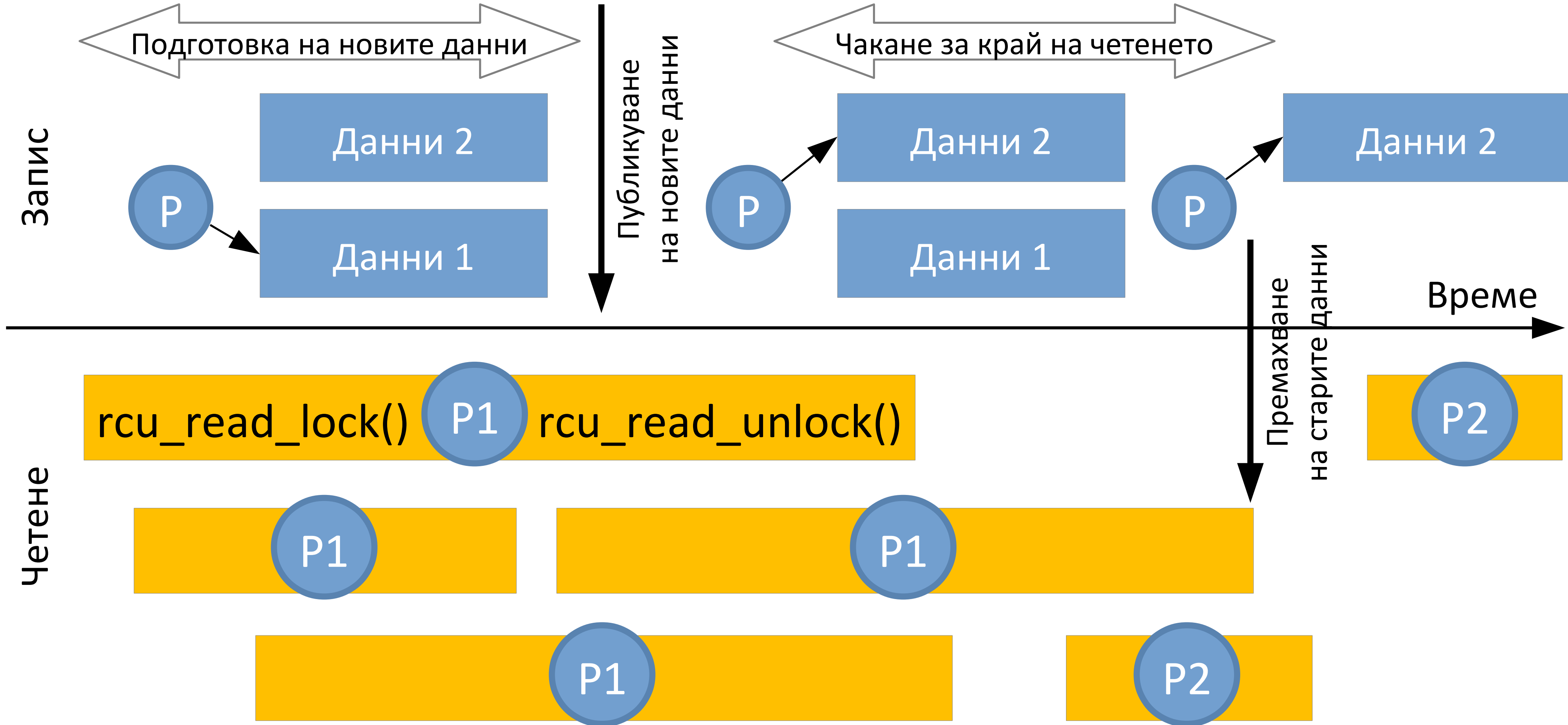
RCU

- Read-copy-update (RCU) е механизъм за синхронизация, основан на взаимно изключване, но RCU не прилага взаимно изключване в конвенционалния смисъл: RCU четците могат и работят едновременно с RCU актуализации. Вариантът на взаимно изключване на RCU е в пространството, като четците на RCU имат достъп до стари версии на данните, които се актуализират едновременно, а не във времето, както е при конвенционалните механизми за контрол на паралелността;
- Използва се, когато изпълнението на четенията е от решаващо значение. Този подход е компромис между памет и време, позволяващ бързи операции с цената на повече заета памет;

RCU

- Read-copy-update позволява на множество нишки ефективно да се четат от споделена памет чрез отлагане на актуализациите за предварително съществуващите четения за по-късно време, като същевременно новите читатели ще прочетат актуализираните данни;
- Това кара всички читатели да продължат работата си така, сякаш няма синхронизация, следователно четенията ще бъдат бързи, но също така актуализациите ще бъдат затруднени;

RCU протокол



Транзакционна памет

(Transactional Memory)

Транзакционна памет

- Има два вида ТМ – софтуерна (STM) и хардуерна (ТМ подпомогната от специални инструкции в процесора);
- STM често е много оптимистичен подход: една нишка извършва модификации на споделената памет, без да се съобразява какво могат да правят другите нишки, записвайки всяко четене и запис в лог;
- Вместо да се натовари записващия с отговорността да се увери, че записа не се отразява неблагоприятно върху други операции в ход, отговорността е на “читателя”, който след приключване на цяла транзакция проверява дали други нишки не са направили едновременно промени в паметта (и затова прочетеното вече не е актуално);

Транзакционна памет

- Тази последна операция, при която промените на транзакция се валидират и ако валидирането е успешно, тогава промените стават постоянни се нарича “commit”;
- Една транзакция може също да се прекрати по всяко време, отменяйки всички нейни досегашни промени;
- Ако транзакция не може да бъде завършена поради промени, тя обикновено се прекъсва и се изпълнява от самото начало, докато успее;