Спецкурс кафедры «Вычислительной математки»

Параллельные алгоритмы вычислительной алгебры

Александр Калинкин

Сергей Гололобов

Часть 3: Распараллеливание на компьютерах с общей памятью

- Средства программирования для компьютеров с общей памятью (OpenMP, TBB, Cilk, OpenCL, OpenACC)
- Понятие потока в вычислениях на компьютерах с общей памятью
- Особенности параллельных программ для компьютеров с общей памятью
- Представление об управляющих конструкциях OpenMP: Shared, Private, FirstPrivate, LastPrivate
- Примеры простейших эффективных и неэффективных алгоритмов
- Синхронизация параллельных вычислений

Средства программирования для компьютеров с общей памятью

Компьютер с общей памятью (shared memory)

Особенность: автоматический обмен информацией через общую память (все ядра могут прочитать любой кусок памяти)

ПОМНИМ: реальная структура процессора отличается от этой модели, поскольку она не учитывает иерархию памяти и скорость каналов, по которым передаются данные и команды

Средства программирования для компьютеров с общей памятью

Основное средство программирования: OpenMP (система директив препроцессора, которые сообщают компилятору, какие куски кода можно параллелить). Текущая версия 4.5.

Дополнительные средства программирования:

Pthreads (POSIX) – команды низкого уровня, работают на Линукс-подобных машинах

Winthreads – аналогичные команды для Windows

TBB – C++ библиотека для параллелизации высокого уровня

Cilk – C-подобные команды

Альтернативные подходы к параллелизации: более удобные в использовании и более высокоуровневые

Цель большинства из них – упростить процесс параллельного программирования на машинах с общей

Понятие потока

Поток (нить, thread) – блок команд и данных для исполнения на одном из исполняющих модулей\ядер (могут быть виртуальными)

Потоки деляться на физические (привязанные к ядрам процессора), виртуальные (привязанные к операционной системе) и программные (порождённые программой)

Реально мы программируем программные потоки, но иногда очень хочется программировать реальные потоки для повышения производительности

Понятие потока

Нужно знать, что существуют механизмы, которые позволяют пытаться установить связь между реальными, виртуальными и программными потоками (affinity) – либо через функции (GLibc), либо через переменные окружения, понятные программе (собранной компилятором Intel®, например)

Linux: sched_setaffinity(), sched_getaffinity()

Windows: SetThreadAffinityMask(), SetProcessAffinityMask()

“Универсальный” от Intel® (работает с соответствующими процессорами и компиляторами):
KMP_AFFINITY=“verbose,proclist=[3,2,1,0]”

Привяка потоков важна для производительности вычислительных программ – иначе программные потоки могут прыгать по разным ядрам процессора и требовать огромных перекачек данных между ядрами!

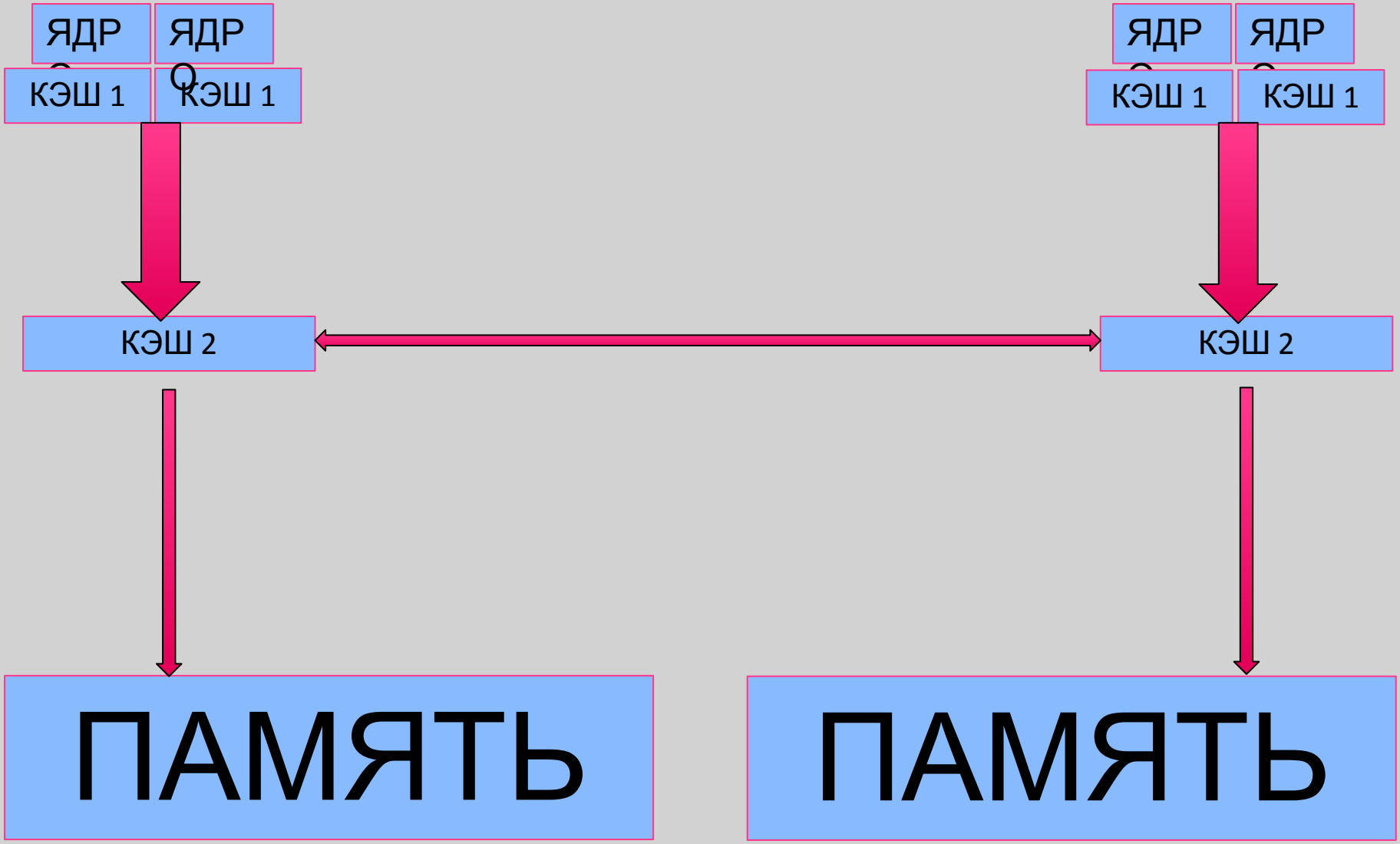
Уровни параллелизма

4 уровня

1. Физический (ядра процессора)
2. Виртуальный физический (гиперсрединг, hyperthreading)
3. Системный
4. Программный

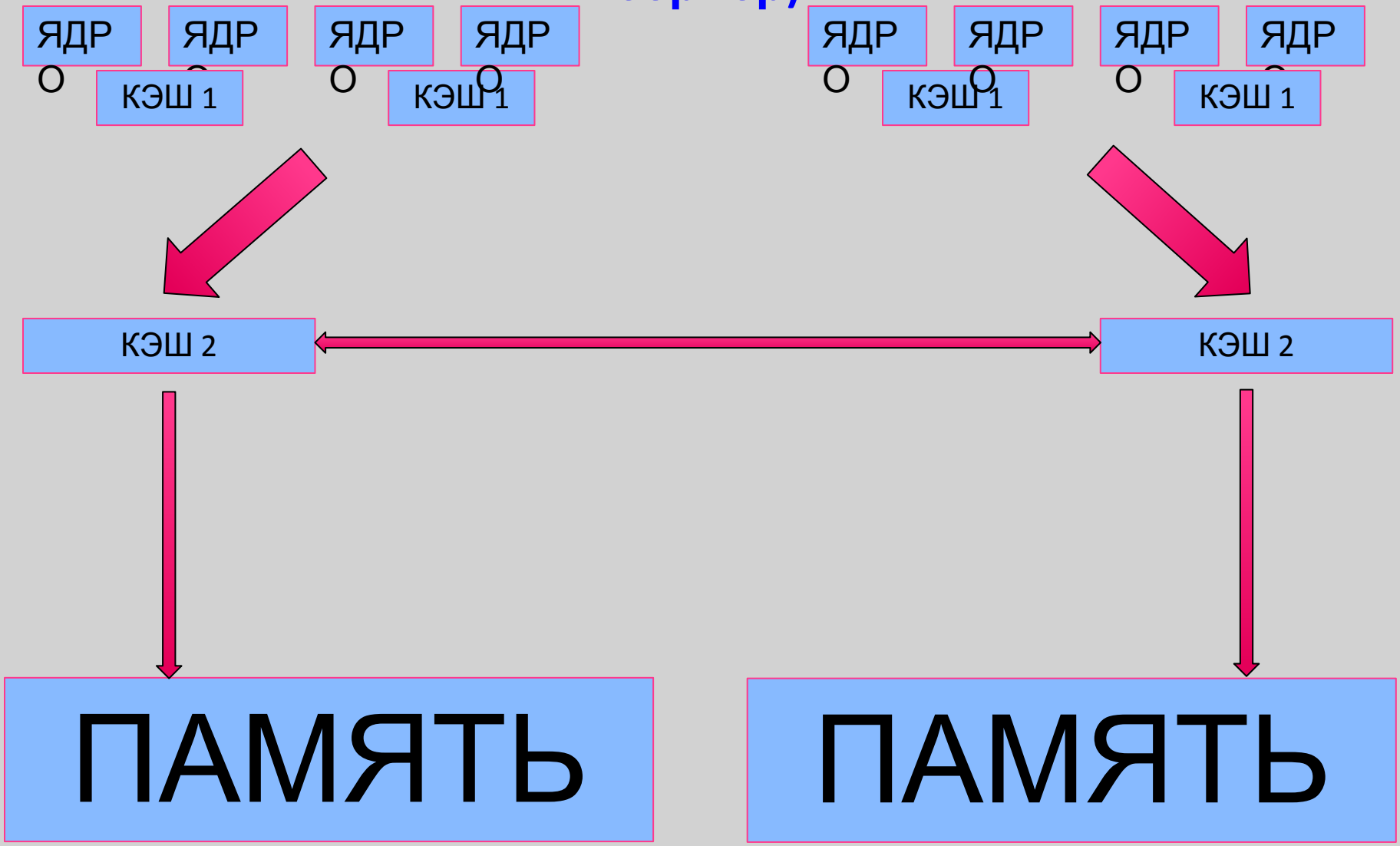
Инженер-программист всегда
программирует
только программный уровень

Физический уровень для 2-процессорного сервера



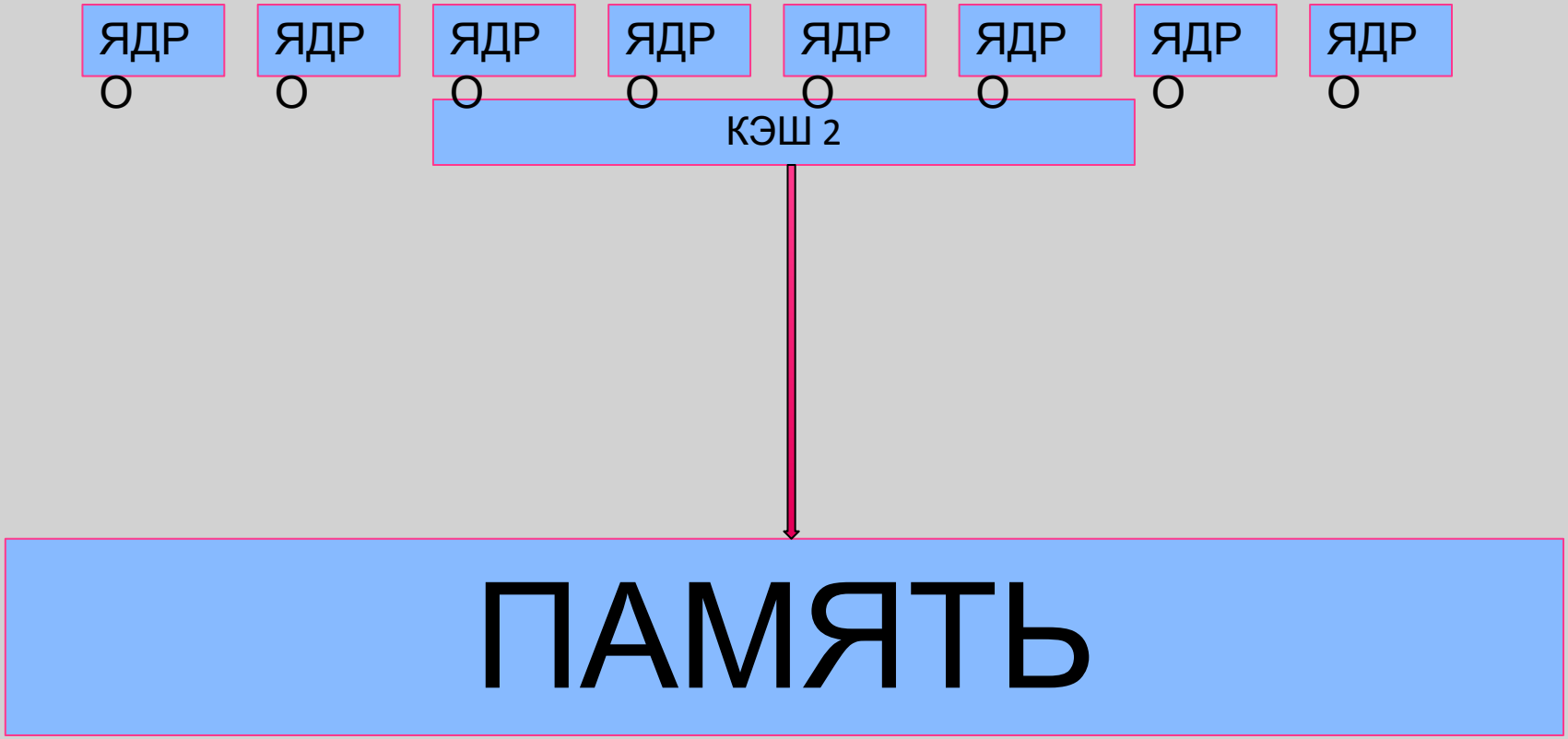
Упрощённая модель 2-процессорного сервера

Виртуальный физический (hyperthreading) уровень (тот же сервер)



Гиперсрединг обычно включён по умолчанию – выключайте, если можете для высокопроизводительных вычислений

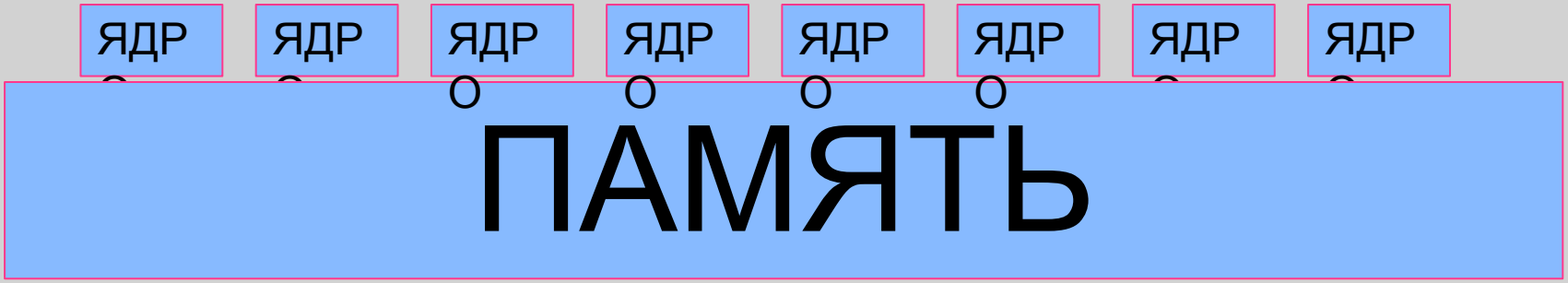
Уровень операционной системы (тот же сервер)



Операционная система рассматривает все ядра как одинаковые, о последствиях позже

Операционная система имеет ограниченную видимость архитектуры

Уровень программы (тот же сервер)



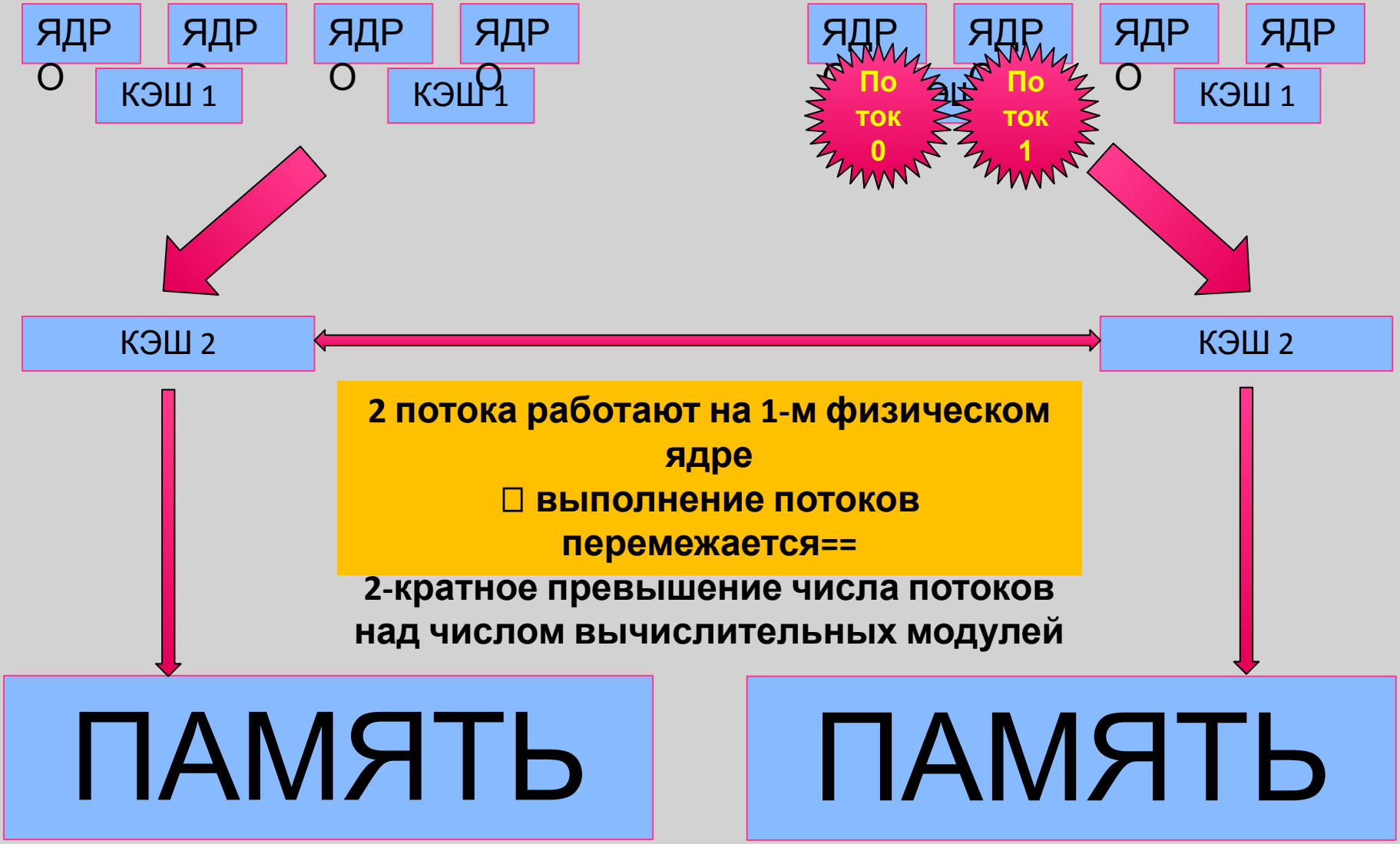
Наивная реализация алгоритма приводит к

- Неустойчивым замерам производительности
  - ($1^{-ый}$ прогон - 5.45с, $2^{-ой}$ прогон 6.34с)
- Плохая масштабируемость при увеличении числа потоков
  - (4 потока - 5.23с, 8 потоков - 6.12с)

OpenMP не очень помогает адресовать проблему ясного видения архитектуры

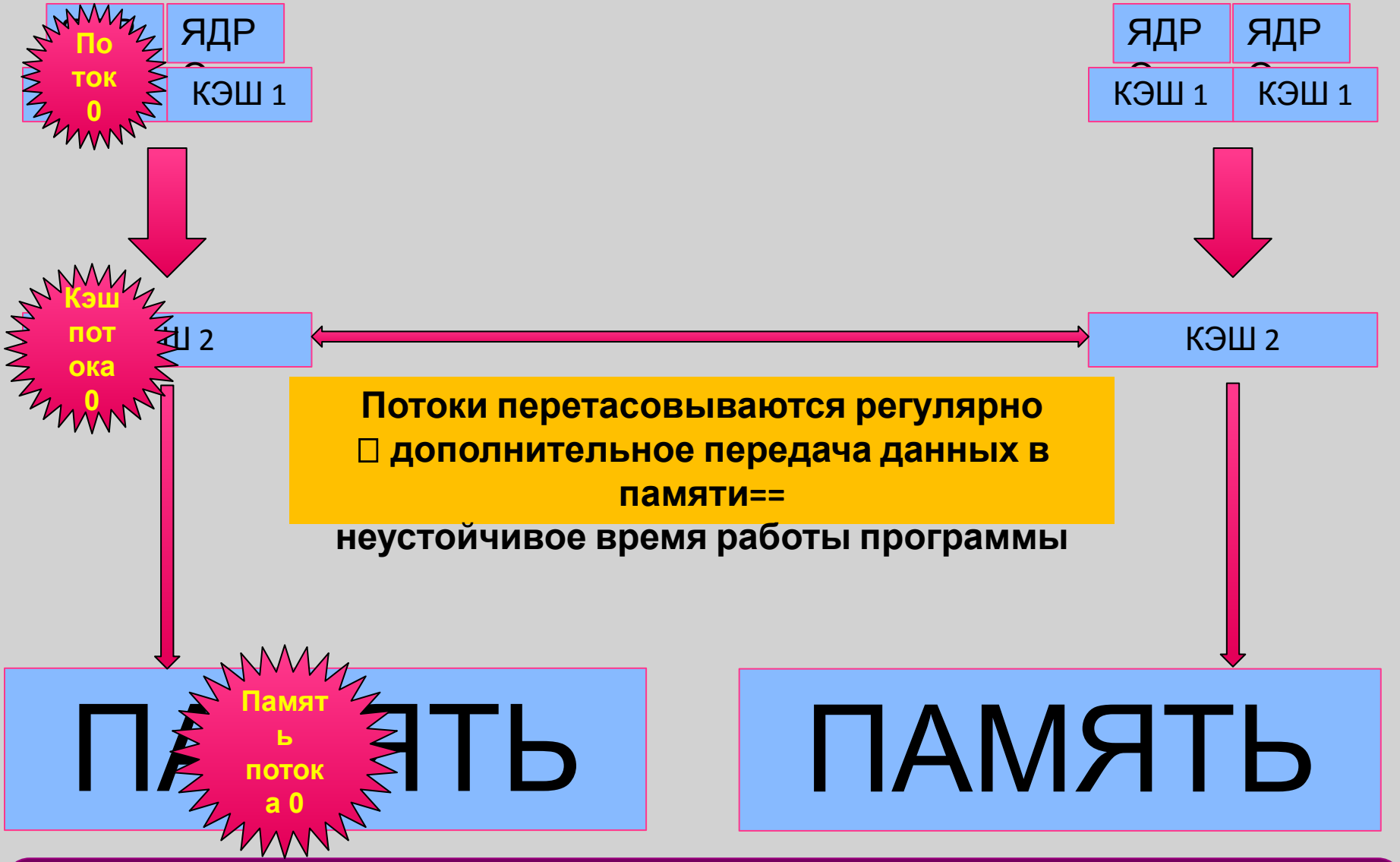
Программа практически не видит архитектуры

Влияние гиперсредингa



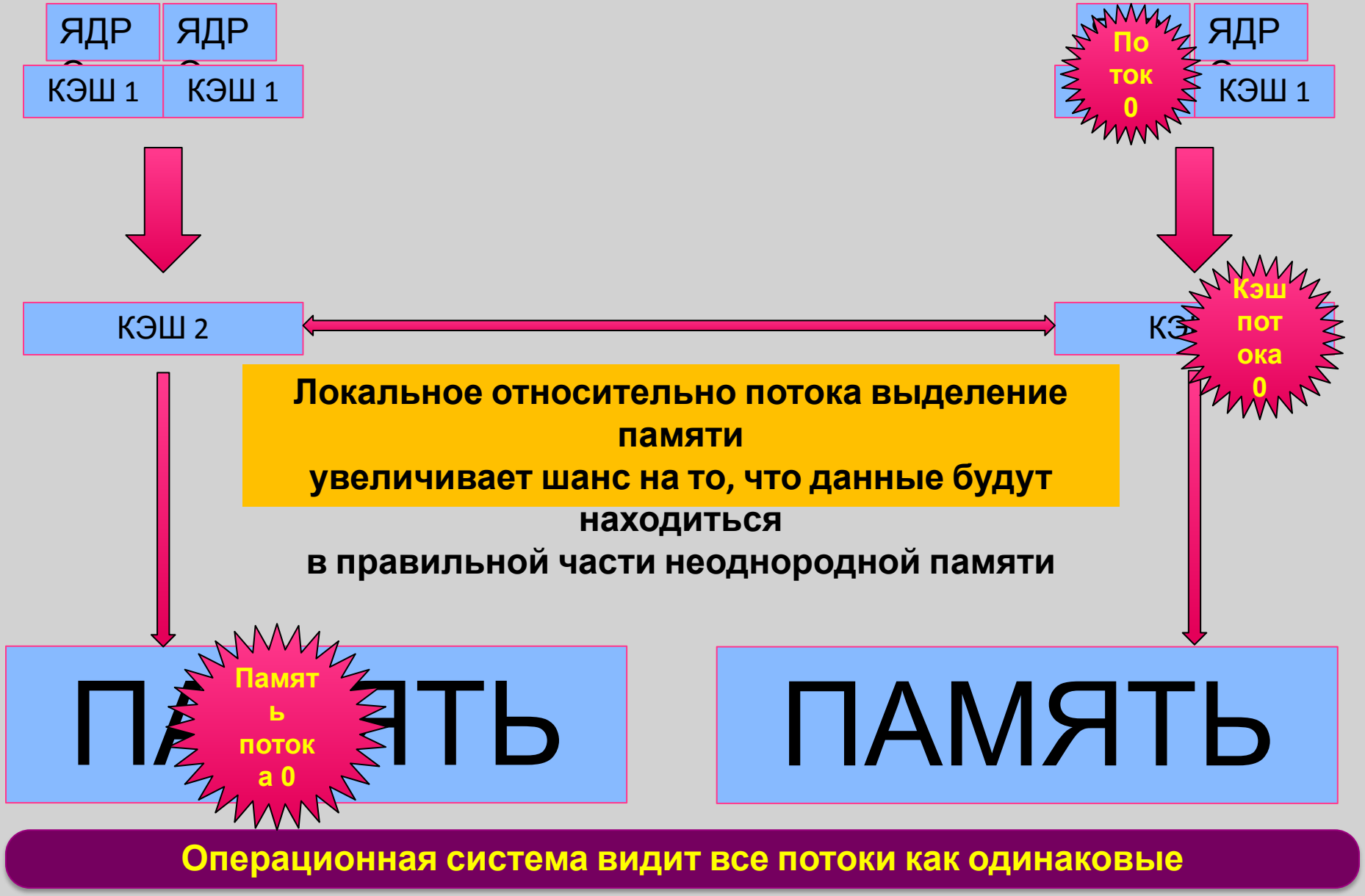
Выключайте гиперсредингв БИОСе по возможности

Влияние операционной системы (перетасовка)

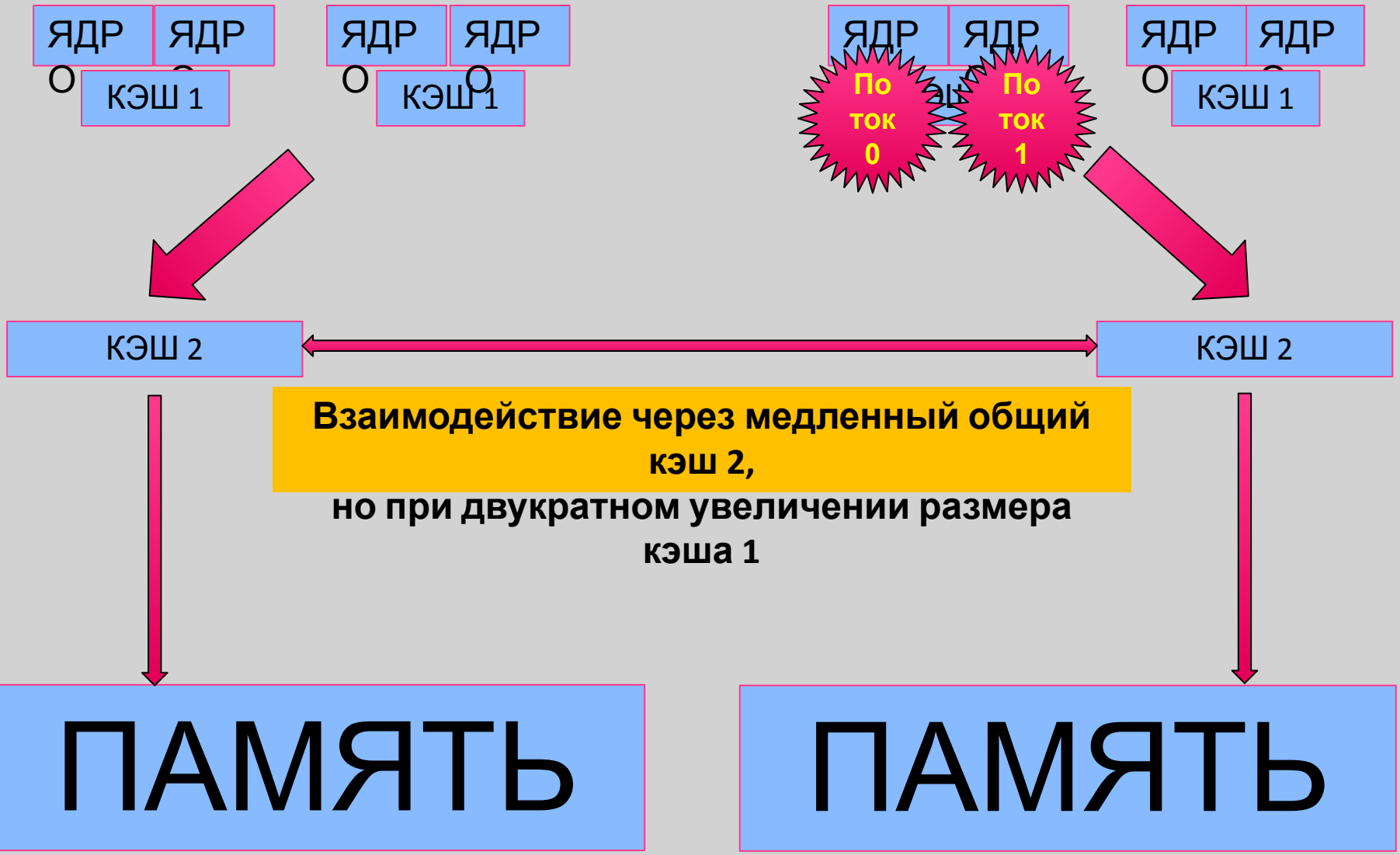


OS treats all HW threads as equal

Влияние неоднородной (NUMA) памяти

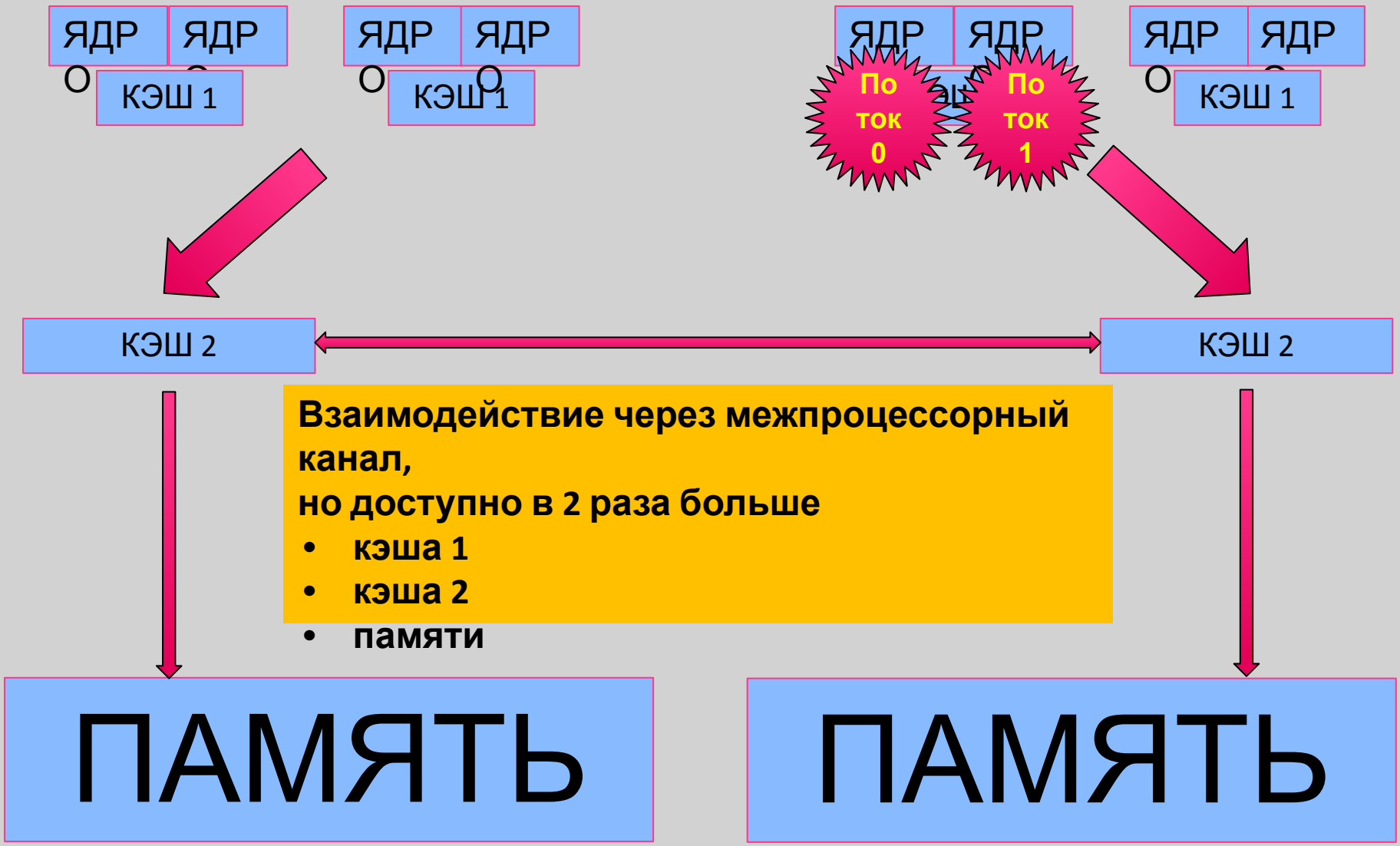


Влияние распределения потоков по ядрам 1



Правильное распределение увеличивает производительность программы

Влияние распределения потоков по ядрам 2



Правильное распределение увеличивает производительность программы

Инструменты Intel для решения проблем

Компиляторы Intel умеют устанавливать привязку разных видов потоков друг к другу

Linux*\OS X*

export KMP_AFFINITY=…

Windows*

set KMP_AFFINITY=…

KMP_AFFINITY=verbose (информация о том, как привязаны потоки)

KMP_AFFINITY не работает для процессоров, произведённых не компанией Intel, но есть аналогичные инструменты

KMP_AFFINITY главный инструмент для привязки потоков друг к другу

Работа с установками affinity

KMP_AFFINITY=verbose

Выдача

OMP: Info #179: KMP_AFFINITY: 2 packages x 8 cores/pkg x 2 threads/core (16 total cores)

Гиперсрединг присутствует

Гиперсрединг включён

OMP: Info #147: KMP_AFFINITY: Internal thread 0 bound to OS proc set
{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31}
OMP: Info #147: KMP_AFFINITY: Internal thread 1 bound to OS proc set
{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31}

Перетасовка потоков включена

KMP_AFFINITY=verbose,granularity=fine,compact,1,0

Выдача

OMP: Info #206: KMP_AFFINITY: OS proc to physical thread map:
OMP: Info #171: KMP_AFFINITY: OS proc 0 maps to package 0 core 0 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 16 maps to package 0 core 0 thread 1
OMP: Info #171: KMP_AFFINITY: OS proc 1 maps to package 0 core 1 thread 0
OMP: Info #171: KMP_AFFINITY: OS proc 17 maps to package 0 core 1 thread 1
OMP: Info #147: KMP_AFFINITY: Internal thread 0 bound to OS proc set {0}
OMP: Info #147: KMP_AFFINITY: Internal thread 1 bound to OS proc set {1}

Affinity для производительности

Перетасовка выключена

KMP_AFFINITY главный инструмент для привязки потоков друг к другу

Рекомендации

Если вы видите

1. Неустойчивое время работы программы
2. Плохую масштабируемость с N на 2N или с N на N+1 поток
3. Плохую производительность на >1 и <MAX потоках
4. Хорошую производительность на одной машине, плохую – на другой
5. Хорошую масштабируемость на одной машине, плохую – на другой

обратите внимание на установки affinity!

Особенности параллельных программ

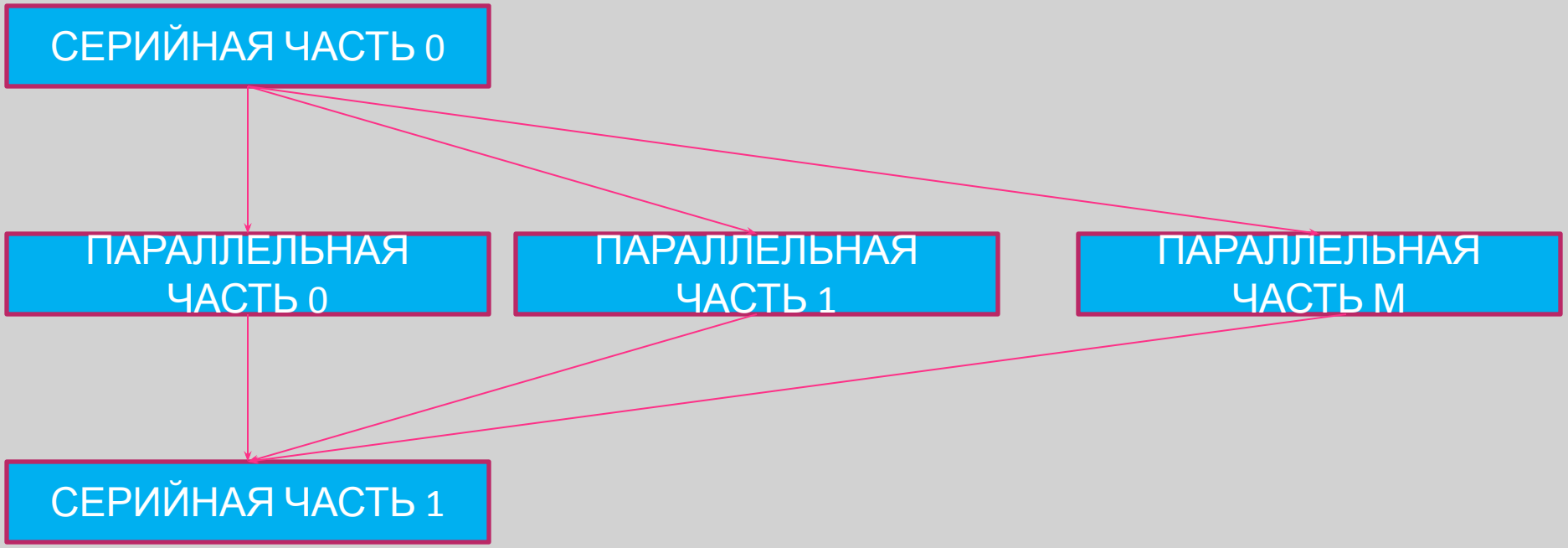
Основа – обычная последовательная программа

Дополнена директивами препроцессора, которые сообщают компилятору информацию о том, какого рода параллелизм возможен в данном месте программы

Преимущество и недостаток: легко превращает последовательную программу в параллельную (1 код для всех потоков)

Представление об управляющих конструкциях OpenMP

Принцип работы: интерпретация куска программы как программы для многих потоков



Как при этом выглядит программа? Сейчас узнаем...

Представление об управляющих конструкциях OpenMP

- **Директивы компилятора:**
  #pragma omp *NAME [clause [clause]…] (Си)*
  <!,*,c>$OMP NAME [clause [clause]…] (Фортран)
- **Пример**
  #pragma omp parallel for num_threads(4)
  c$OMP PARALLEL DO NUM_THREADS(4)
- **Хэдер с прототипами функций и типами:**
  #include <omp.h>

Ключи компиляторов для использования в параллельных программах с OpenMP

- **-fopenmp (gcc)**
- **-mp (pgi)**
- **/Qopenmp (intel)**

Инфо: www.openmp.org (английский), последняя версия 4.5

Представление об управляющих конструкциях OpenMP

- **Полезные функции**

void omp_set_num_threads(int num_threads); (Си)
subroutine omp_set_num_threads(*num_threads) (Фортран)*
integer *num_threads*
int omp_get_num_threads(void);
integer function omp_get_num_threads()
int omp_get_max_threads(void);
integer function omp_get_max_threads()
int omp_get_thread_num(void);
integer function omp_get_thread_num()
int omp_in_parallel(void);
logical function omp_in_parallel()
void omp_[set,get]_[nested,dynamic](int var);
subroutine omp_[set,get]_[nested,dynamic] (var)
logical var

Представление об управляющих конструкциях OpenMP

- **Полезные функции для продвинутого параллелизма**

```
void omp_init_lock(omp_lock_t *lock);
void omp_init_nest_lock(omp_nest_lock_t *lock);
subroutine omp_init_lock(svar)
integer (kind=omp_lock_kind) svar
subroutine omp_init_nest_lock(nvar)
integer (kind=omp_nest_lock_kind) nvar
void omp_destroy_lock(omp_lock_t *lock);
subroutine omp_destroy_lock(svar)
integer (kind=omp_lock_kind) svar
void omp_[set,unset,test]_lock(omp_lock_t *lock);
subroutine omp_[set,unset,test]_lock(svar)
integer (kind=omp_lock_kind) svar
```

Представление об управляющих конструкциях OpenMP

- **Пример OpenMP программы**

```
#pragma omp parallel for private(i) firstprivate(N) shared(a,b) lastprivate(j)
 for (i=0; i<N; i++,j++)
 {
   sum+=a[i]*b[i];
 }
```

Примеры простейших эффективных и неэффективных алгоритмов

- **Скалярное произведение (Версия 0)**

```
sum=0.0;
 for (int i=0; i<N; i++)
 {
   sum+=a[i]*b[i];
 }
```

Вариант	Результат	Время 1(dbg)	Время 2(opt)
Серийный	-6.710886e+007	0.361609	0.22866

Примеры простейших эффективных и неэффективных алгоритмов

- Скалярное произведение (Версия 1)

```
sum=0.0;
#pragma omp parallel for
  for (int i=0; i<N; i++)
  {
    sum+=a[i]*b[i];
  }
```

Вариант	Результат	Время 1(dbg)	Время 2(opt)
Серийный	-6.710886e+007	0.361609	0.22866
Простой параллельный	-3.997392e+007	0.506171	0.231347

Примеры простейших эффективных и неэффективных алгоритмов

- Скалярное произведение (Версия 2)

```
sum=0.0;
#pragma omp parallel for
  for (int i=0; i<N; i++)
  {
#pragma omp atomic
    sum+=a[i]*b[i];
  }
```

Вариант	Результат	Время 1(dbg)	Время 2(opt)
Серийный	-6.710886e+007	0.361609	0.22866
Простой параллельный	-3.997392e+007	0.506171	0.231347
Правильный параллельный	-6.710886e+007	4.02804	0.228379

Примеры простейших эффективных и неэффективных алгоритмов

- Скалярное произведение (Версия 3)

```
sum=0.0;
#pragma omp parallel for reduction(+:sum)
  for (int i=0; i<N; i++)
  {
    sum+=a[i]*b[i];
  }
```

Вариант	Результат	Время 1(dbg)	Время 2(opt)
Серийный	-6.710886e+007	0.361609	0.22866
Простой параллельный	-3.997392e+007	0.506171	0.231347
Правильный параллельный	-6.710886e+007	4.02804	0.228379
Быстрый параллельный	-6.710886e+007	0.277683	0.230722

Примеры простейших эффективных и неэффективных алгоритмов

- Скалярное произведение (Версия 4)

```
sum=0.0;
#pragma omp parallel for reduction(+:sum) num_threads(4)
  for (int i=0; i<N; i++)
  {
    sum+=a[i]*b[i];
  }
```

Вариант	Результат	Время 1(dbg)	Время 2(opt)
Серийный	-6.710886e+007	0.361609	0.22866
Простой параллельный	-3.997392e+007	0.506171	0.231347
Правильный параллельный	-6.710886e+007	4.02804	0.228379
Быстрый параллельный	-6.710886e+007	0.277683	0.230722
Слишком параллельный	-6.710886e+007	0.339858	0.229528

Примеры простейших эффективных и неэффективных алгоритмов

- **Скалярное произведение (Версия 5)**

```
sum=0.0;
#pragma omp parallel for reduction(+:sum) shared(a,b)
  for (int i=0; i<N; i++)
  {
    sum+=a[i]*b[i];
  }
```

Вариант	Результат	Время 1(dbg)	Время 2(opt)
Серийный	-6.710886e+007	0.361609	0.22866
Простой параллельный	-3.997392e+007	0.506171	0.231347
Правильный параллельный	-6.710886e+007	4.02804	0.228379
Быстрый параллельный	-6.710886e+007	0.277683	0.230722
Слишком параллельный	-6.710886e+007	0.339858	0.229528
Оформленный параллельный	-6.710886e+007	0.29047	0.229149

Примеры простейших эффективных и неэффективных алгоритмов

- **Скалярное произведение (Версия 6)**

```
sum=0.0;
#pragma omp parallel for reduction(+:sum) shared(a,b) schedule(dynamic)
 for (int i=0; i<N; i++)
 {
   sum+=a[i]*b[i];
 }
```

Вариант	Результат	Время 1(dbg)	Время 2(opt)
Серийный	-6.710886e+007	0.361609	0.22866
Простой параллельный	-3.997392e+007	0.506171	0.231347
Правильный параллельный	-6.710886e+007	4.02804	0.228379
Быстрый параллельный	-6.710886e+007	0.277683	0.230722
Слишком параллельный	-6.710886e+007	0.339858	0.229528
Оформленный параллельный	-6.710886e+007	0.29047	0.229149
Динамический параллельный	-6.710886e+007	3.11317	0.230229

Примеры простейших эффективных и неэффективных алгоритмов

- **Скалярное произведение (Версия 7)**

```
 sum=0.0;
 int M=32;
#pragma omp parallel for re
 for (int i=0; i<N/M; i++)
 {
  for (int j=i*M; j<(i+1)*M; j
  {
   sum+=a[j]*b[j];
  }
 }
```

Вариант	Результат	Время 1(dbg)	Время 2(opt)
Серийный	-6.710886e+007	0.361609	0.22866
Простой параллельный	-3.997392e+007	0.506171	0.231347
Правильный параллельный	-6.710886e+007	4.02804	0.228379
Быстрый параллельный	-6.710886e+007	0.277683	0.230722
Слишком параллельный	-6.710886e+007	0.339858	0.229528
Оформленный параллельный	-6.710886e+007	0.29047	0.229149
Динамический параллельный	-6.710886e+007	3.11317	0.230229
Блочный параллельный	-6.710886e+007	0.27316	0.227583

Примеры простейших эффективных и неэффективных алгоритмов

- **Скалярное произведение (Версия 8)**

```
#pragma omp parallel sections
 {
#pragma omp section
   {
    //printf("I'm thread No. %i\n",
    for (int i=0; i<N/2; i++)
    {
     sum1+=a[i]*b[i];
    }
   }
#pragma omp section
   {
    //printf("I'm thread No. %i\n",
    for (int i=N/2; i<N; i++)
    {
     sum2+=a[i]*b[i];
    }
   }
 }
```

Вариант	Результат	Время 1(dbg)	Время 2(opt)
Серийный	-6.710886e+007	0.361609	0.22866
Простой параллельный	-3.997392e+007	0.506171	0.231347
Правильный параллельный	-6.710886e+007	4.02804	0.228379
Быстрый параллельный	-6.710886e+007	0.277683	0.230722
Слишком параллельный	-6.710886e+007	0.339858	0.229528
Оформленный параллельный	-6.710886e+007	0.29047	0.229149
Динамический параллельный	-6.710886e+007	3.11317	0.230229
Блочный параллельный	-6.710886e+007	0.27316	0.227583
Многозадачный параллельный	-6.710886e+007	0.396685	0.235459

Примеры простейших эффективных и неэффективных алгоритмов

- **Итог:**

Вариант	Результат	Время 1(dbg)	Время 2(opt)
Серийный	-6.710886e+007	0.361609	0.22866
Простой параллельный	-3.997392e+007	0.506171	0.231347
Правильный параллельный	-6.710886e+007	4.02804	0.228379
Быстрый параллельный	-6.710886e+007	0.277683	0.230722
Слишком параллельный	-6.710886e+007	0.339858	0.229528
Оформленный параллельный	-6.710886e+007	0.29047	0.229149
Динамический параллельный	-6.710886e+007	3.11317	0.230229
Блочный параллельный	-6.710886e+007	0.27316	0.227583
Многозадачный параллельный	-6.710886e+007	0.396685	0.235459

Увы, но на [illegible] но отличаться от приведённых в таблице выше...

Синхронизация параллельных вычислений

- Простейшая конструкция синхронизации: barrier

```
#pragma omp barrier
!$omp barrier
```

- Исполнение параллельного кода присотанавливается до тех пор, пока все потоки не дойдут до данного места в программе
  - Польза: Гарантирует, что все потоки продолжат вычисления не раньше, чем закончат предыдущий кусок работы
  - Проблема: скорость работы программы определяется самым медленным из потоков

Синхронизация параллельных вычислений

```
loop:
#pragma omp parallel for shared(a,b) reduction(+:sum)
 for (int i=begin; i<end; i++)
 {
  sum+=a[i]*b[i];
 }
```

2 раза посчитал sum, но параллельно!

sum1=loop(a,b,0,N/2);
sum2=loop(a,b,N/2,N);
sum=sum1+sum2;
	#pragma omp parallel private(sum)
	{
	sum=loop(a,b,0,N/2);
	#pragma omp barrier
	sum+=loop(a,b,N/2,N);
	}

Сменилось состояние ОС!

Вариант	Результат	Время 1(dbg)	Время 2(opt)
Блочный параллельный	-6.710886e+007	0.261467	0.233598
2-шаговый параллельный	-6.710886e+007	0.268518	0.258157
2-шаговый с барьером	-6.710886e+007	0.541211	0.249455

2 раза посчитал sum, но параллельно!

Синхронизация параллельных вычислений

- Конструкции Single & Master

#pragma omp single\master

!$omp single\master

- Исполнение данной части кода происходит одним из потоков и содержит барьер неявно (single)\главным потоком без барьера(master)
  - Польза: Гарантирует исполнение куска кода только одни потоком
  - Проблема: нужно быть внимательным!

Синхронизация параллельных вычислений

```
loop:
#pragma omp parallel for shared(a,b) reduction(+:sum)
  for (int i=begin; i<end; i++)
  {
    sum+=a[i]*b[i];
  }
```

```
sum1=loop(a,b,0,N/2);
sum2=loop(a,b,N/2,N);
sum=sum1+sum2;
```

```
#pragma omp parallel
  {
#pragma omp single
    sum=loop(a,b,0,N/2);
#pragma omp single
    sum+=loop(a,b,N/2,N);
  }
```

Вариант	Результат	Время 1(dbg)	Время 2(opt)
Блочный параллельный	-6.710886e+007	0.261467	0.233598
2-шаговый параллельный	-6.710886e+007	0.268518	0.258157
2-шаговый с барьером	-6.710886e+007	0.541211	0.249455
2-шаговый с single	-6.710886e+007	0.493632	0.239554

Компилятор поработал

Синхронизация параллельных вычислений

- Конструкция Critical

```
#pragma omp critical
!$omp critical
```

- Исполнение данной части кода происходит потоками по очереди
  - Польза: Гарантирует исполнение куска кода всеми потоками по очереди
  - Проблема: неявная последовательность в коде

Синхронизация параллельных вычислений

```
loop:
#pragma omp parallel for shared(a,b) reduction(+:sum)
  for (int i=begin; i<end; i++)
  {
    sum+=a[i]*b[i];
  }
```

```
sum1=loop(a,b,0,N/2);
sum2=loop(a,b,N/2,N);
sum=sum1+sum2;
```

```
#pragma omp parallel
  {
#pragma omp critical
    sum1=loop(a,b,0,N/2);
#pragma omp critical
    sum2=loop(a,b,N/2,N);
  }
sum=sum1+sum2;
```

Вариант	Результат	Время 1(dbg)	Время 2(opt)
Блочный параллельный	-6.710886e+007	0.261467	0.233598
2-шаговый параллельный	-6.710886e+007	0.268518	0.258157
2-шаговый с барьером	-6.710886e+007	0.541211	0.249455
2-шаговый с single	-6.710886e+007	0.493632	0.239554
2-шаговый с critical	-6.710886e+007	0.946724	0.242806

Снова компилятор

Синхронизация параллельных вычислений

- Конструкция Flush

#pragma omp flush

!$omp flush

- Делает видимой всем часть памяти (переменные), которые принадлежат данному потоку
  - Польза: Позволяет организовать обмен информацией между потоками
  - Проблема: требует внимательности!

Flush неявно стоит, например, при входе и выходе из parallel for

Немного об OpenMP 4.*

- OpenMP 4.0 – это ответ на вызов альтернативного стандарта OpenACC
  - Дополнения, связанные с использованием вспомогательных устройств типа Intel® Xeon Phi™ или GPGPU всевозможных производителей (device constructs)
  - Дополнительные возможности по реализации и отладке продвинутого параллелизма (SIMD, depend, proc_bind, user defined reduction, cancel, OMP_DISPLAY_ENV)
  - Расширенная поддержка Фортрана 2003 и C++

(Не забудьте посмотреть на OpenMP 4+!)

Резюме

Компьютер с общей памятью является простейшим вариантом параллельного компьютера

Компьютер с общей памятью исполняет потоки команд с данными (threads)

Наиболее устоявшийся подход к программированию для компьютеров с общей памятью – система директив и библиотек OpenMP, но конкуренты ему подрастают и очень активно

Отладка и настройка параллельных алгоритмов усложняется по сравнению с серийным кодом на порядок

Дополнительную сложность при оптимизации создаёт различная архитектура компьютеров с общей памятью

Оптимизация параллельной программы осложняется разным поведением программы в отладочном и в компиляторно (автоматически)-оптимизированном режиме

Задания на понимание

1. Напишите программу на Си, которая выполняет Версии 0-8 алгоритма для вычисления скалярного произведения на вашем многоядерном компьютере для векторов длины 67108864. Выпишите таблицу времён работы различных реализаций алгоритма в отладочной и оптимизированной вариации. Найдите лучшую и худшую реализацию алгоритма.
2. Выполните задание 1 на Фортране.
3. Напишите программу, вычисляющую умножение квадратной матрицы размера MxM на вектор-столбец размера M.
4. Распараллельте алгоритм из задания 3 различными способами (а лучше, не менее 5).
5. Определите наиболее и наименее эффективные реализации алгоритма из задания 4 на вашем конкретном компьютере. Проанализируйте, что могло послужить причиной для различной эффективности алгоритма в