Быстрые программы = архитектуро-ориентированные алгоритмы и данные

- Южный федеральный университет, мехмат,
  Борис Штейнберг & группа разработчиков ОРС
  - www.ops.rsu.ru

- Работа поддержана

ангстрем

группа компаний

http://ops.rsu.ru

Группа ОРС:
эксперименты по ускорению программ и разработка «Оптимизирующей распараллеливающей системы»

Со-основателю Яндекса Илье Сегаловичу посвящается

Проблема: ***производительность многих наших программ иногда на ДВА ПОРЯДОКА меньше заявленной производительности компьютера.***

- Оптимизирующие и распараллеливающие компиляторы не решают эту проблему.

Две вещи существенно ускоряют программы: параллелизм и использование памяти

- На двуядерном процессоре с SSE библиотечная программа перемножает матрицы быстрее простой студенческой программы **в 60 раз!**
- Ускорение:
- **в 8 раз** за счет параллельности (2 ядра по 4 операции векторизатора),
- **в 8 раз** – за счет оптимизации использования памяти.
- ***Как выжимать такое ускорение?***

Автоматическая векторизация и распараллеливание у компиляторов развиты плохо.

- *Мы все равно пользуемся компиляторами и любим их, но только настоящий друг может сказать, что «вся спина сзади белая»*

Результаты проверки возможностей автоматической векторизации циклов известными компиляторами

Компилятор	Версия	Релиз	№ примера 1	2	3	4	5	6	7	8
Clang/LLVM	3.3	17.06.2013	+	–	–	–	–	–	–	–
GCC	4.8.1	31.05.2013	+	+	–	–	–	–	+	–
Intel C++ Compiler	13.1.3	20.06.2013	+	+	+	–	–	–	+	–
MS Visual C++	11.0	15.08.2012	+	–	–	–	–	–	–	–

- Наличие (отсутствие) в различных компиляторах автоматических средств, выполняющих преобразования программ, способствующие векторизации.
- Анализ В. Брагилевского и О. Штейнберга

- **1) Тривиально векторизуемый цикл**
- For (int i = 0; i < SIZE; ++i)
- { A[i] = B[i] + C[i];}
- **2) Разрезание**
- For (int i = 0; i < SIZE; ++i)
- { A[i] = B[i] + C[i];
- D[i+2] = D[i+1] + D[i];}
- **3) Перестановка операторов и разрезание**
- for (i=0; i<SIZE; i=i+1)
- { A[i] = B[i] + C[i];
- C[i+1] = D[i] + E[i];}
- **4) Введение временных массивов**
- For (i = 0; I < SIZE; i++)
- { A[i] = B[i] + C[i];
- C[i] = A[i+1] + B[i];}
- **5) Растягивание скаляров**
- For (i = 0; I < SIZE; i++)
- { A[i] = B[i]+C;
- C = A[i+1]+B[i];}
- **6) Повышение размерности массива**
- For (i = 0; i < SIZE; i++)
- { A[i] = B[i] + C[j];
- C[j] = A[i+1] + B[i];}
- **7) Рекуррентный цикл**
- For (int i = 0; i < SIZE; ++i)
- { A[i] = A[i-4] + A[i-8];}
- **8) Рекуррентный цикл**
- For (int i = 0; i < SIZE; ++i)
- { A[i] = A[i-1] + A[i-2];}

Преобразования под векторизацию вручную

```
For (i = 0; I < SIZE; i++)                    //цикл 4
{  A[i] = B[i] + C[i];
C[i] = A[i+1] + B[i];}

For (i = 0; I < SIZE; i++)
{T[i] = A[i+1]                    // введение временного массива
 A[i] = B[i] + C[i];
C[i] = T[i] + B[i];}

For (i = 0; I < SIZE; i++) // результат разбиения цикла
{T[i] = A[i+1]};
For (i = 0; I < SIZE; i++)
{A[i] = B[i] + C[i]};
 For (i = 0; I < SIZE; i++)
{C[i] = T[i] + B[i];}
```

Невозможно автоматически распараллелить алгоритм Флойда (а вручную – можно!).

- FOR K=1 TO N DO
- FOR I=1 TO N DO
- FOR J=1 TO N DO
- IF A(I,J)>A(I,K)+A(K,J) THEN

- A(I,J)=A(I,K)+A(K,J)

- Информационные зависимости не позволяют распараллеливать цикл.
- Но зависимости реализуются при I=K или J=K. При любом из этих равенств логическое выражение условного оператора 0>A(I,J). Это выражение ложно, **если элементы матрицы положительны**, но тогда и **зависимостей нет**. Если программист знает, что его матрица неотрицательна, то **он может распараллеливать.**

Эволюция вычислений.

- Умножение чисел на современных процессорах выполняется примерно в 15 раз быстрее, чем сомножители читаются из памяти.
- А 50 лет назад **было наоборот!**
- Оперативная память не успевает подавать данные и для 1 ядра.

Memory Contention (cont #2)

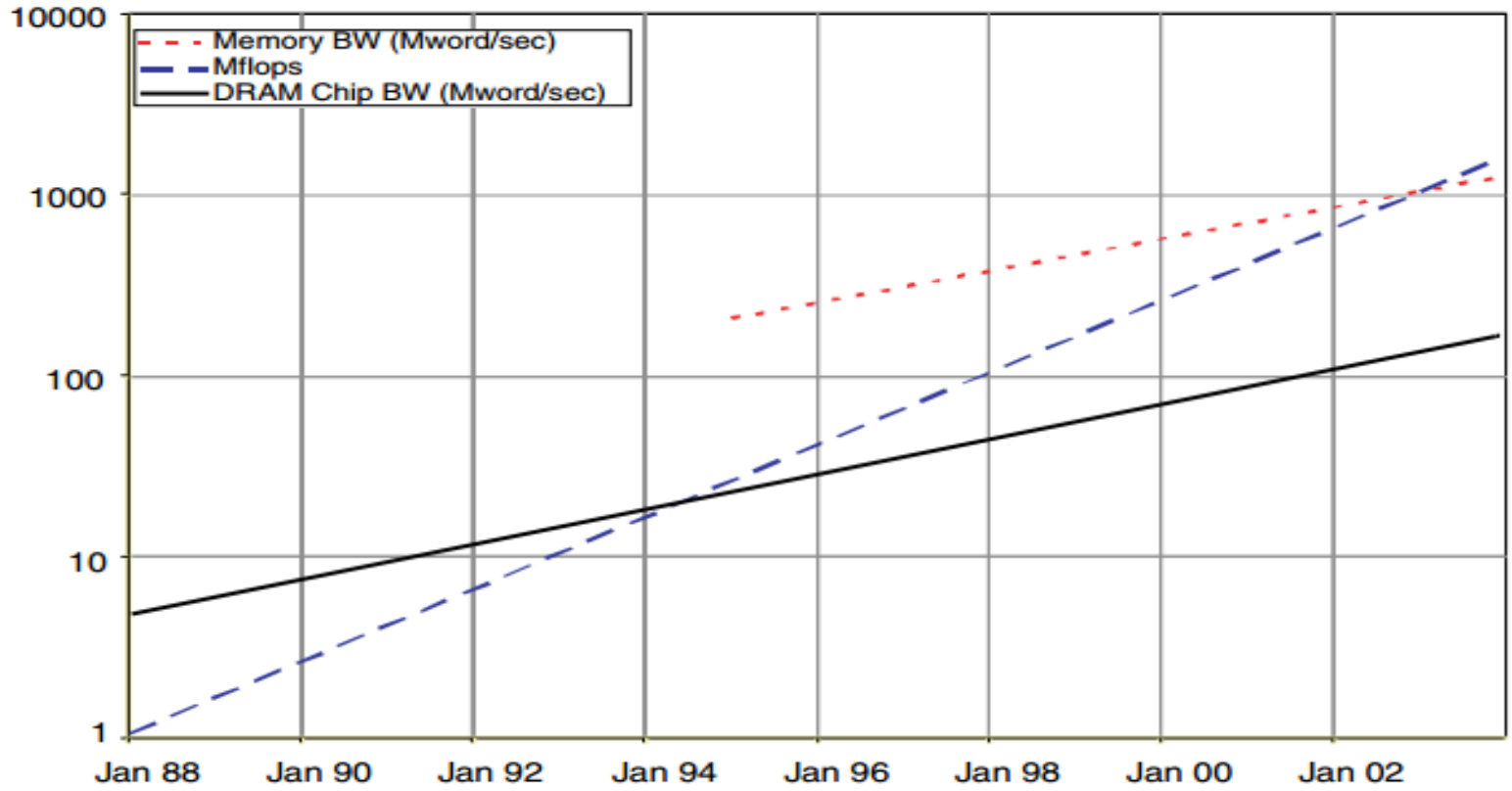


FIGURE 5.3 Arithmetic performance (Mflops), memory bandwidth, and DRAM chip bandwidth per calendar year.

Graham, S.L., Snir, M., and Patterson, C.A., 2005: Getting Up To Speed: The Future Of Supercomputing. National Academies Press, 289p.

Слайд презентации доклада И.Бермуса (Мельбурн, Австралия)

About requirements to the compilers, some examples of optimisation technique and efficient usage of the modern HPC systems

Новые архитектуры требуют новых оптимизированных программ

X(i+2) = X(i+2)+5

- заменять следующим кодом?

K = i+2

X(k) = X(k)+5

- Замена уменьшает количество операций, но вводит новую переменную k. Если для этой переменной нет свободного регистра, то надо обращаться к памяти и терять быстродействие.
- Такая замена давала ускорение на старых процессорах, использовалась в старых пакетах прикладных программ и однозначно давала ускорение. Сейчас замена может дать замедление.

Скалярное произведение векторов.

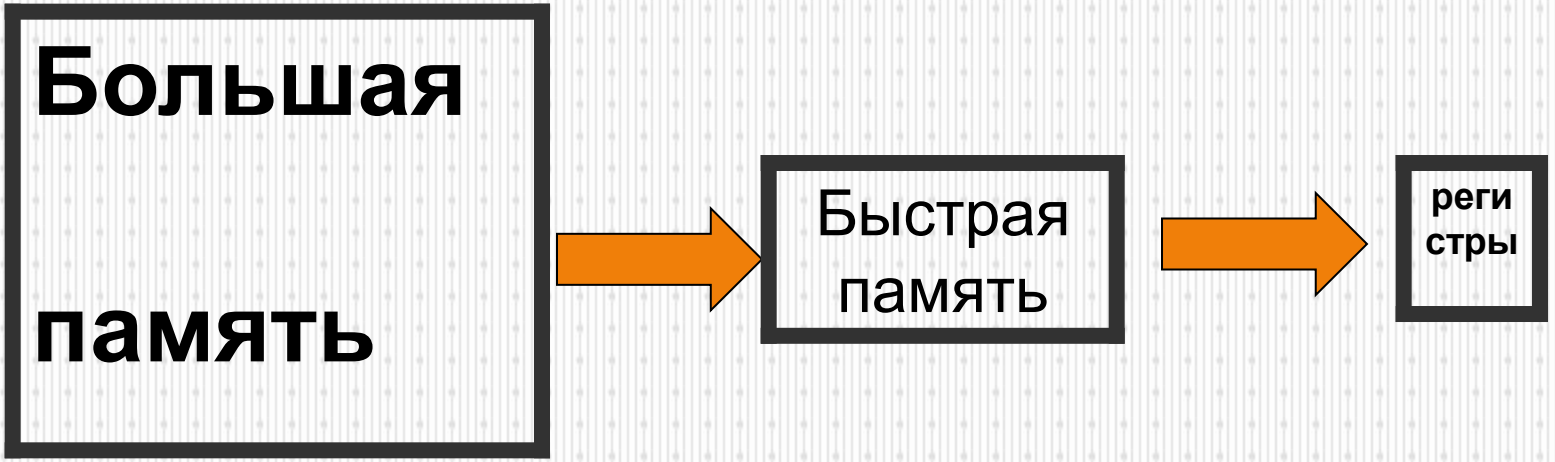
- $x = a_1*b_1 + a_2*b_2 + \ldots + a_n*b_n$
- Арифметических операций столько же, сколько обращений к памяти.
- Если время чтения данных в 15 раз дольше времени умножения, то
- ***Процессор загружен на 1/15 мощности***.

Распараллеливать скалярное произведение?

- $x = a_1*b_1 + a_2*b_2 + \ldots + a_n*b_n$

- Вычисление составляет всего ***1/15*** всего времени работы программы. Мы можем эту долю в разы уменьшить, но останется неизменным ***14/15*** времени на чтение данных
- Сначала готовить данные (оптимизировать обращения к памяти (если это возможно!), а затем – распараллеливать.

Простая иерархия памяти создает непростые проблемы для программиста



Перемножение матриц.

- do i = 1, n
- do j = 1, n
- **do k = 1, n**
- X(i,j) = X(i,j)+A(i,k)*B(k,j).

- Если в кэш помещаются все три матрицы, то нет проблем, процессор использует $3*n^2$ данных и выполняет n^3 умножений. Мало обращений к памяти и много операций!

- Если в кэш помещаются только строка матрицы A и столбец B, то процессор выполняет скалярные произведения и недогружен.

Блочный вид матрицы

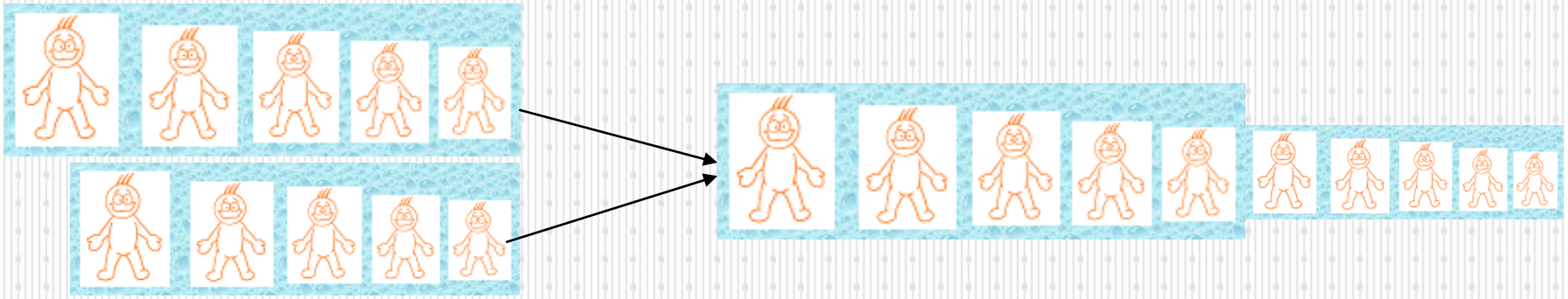
A_{11}	A_{12}	A_{13}	A_{14}
A_{21}	A_{22}	A_{23}	A_{24}
A_{31}	A_{32}	A_{33}	A34
A_{41}	A_{42}	A_{43}	A_{44}

Блочное перемножение матриц

- f = n/m
- do i1 = 1, f
- do j1 = 1, f
- do k1 = 1, f
- do i = 1, m
- do j = 1, m
- do k = 1, m
- XX(i1,j1)(i,j) =
- XX(i1,j1)(i,j) + AA(i1,k1)(i,k) * BB(k1,j1)(k,j)
-
- В быстрой памяти должны помещаться 3 блока: $3*m^2$ данных и m^3 операций

СЛИЯНИЕ ДЛЯ СОРТИРОВКИ

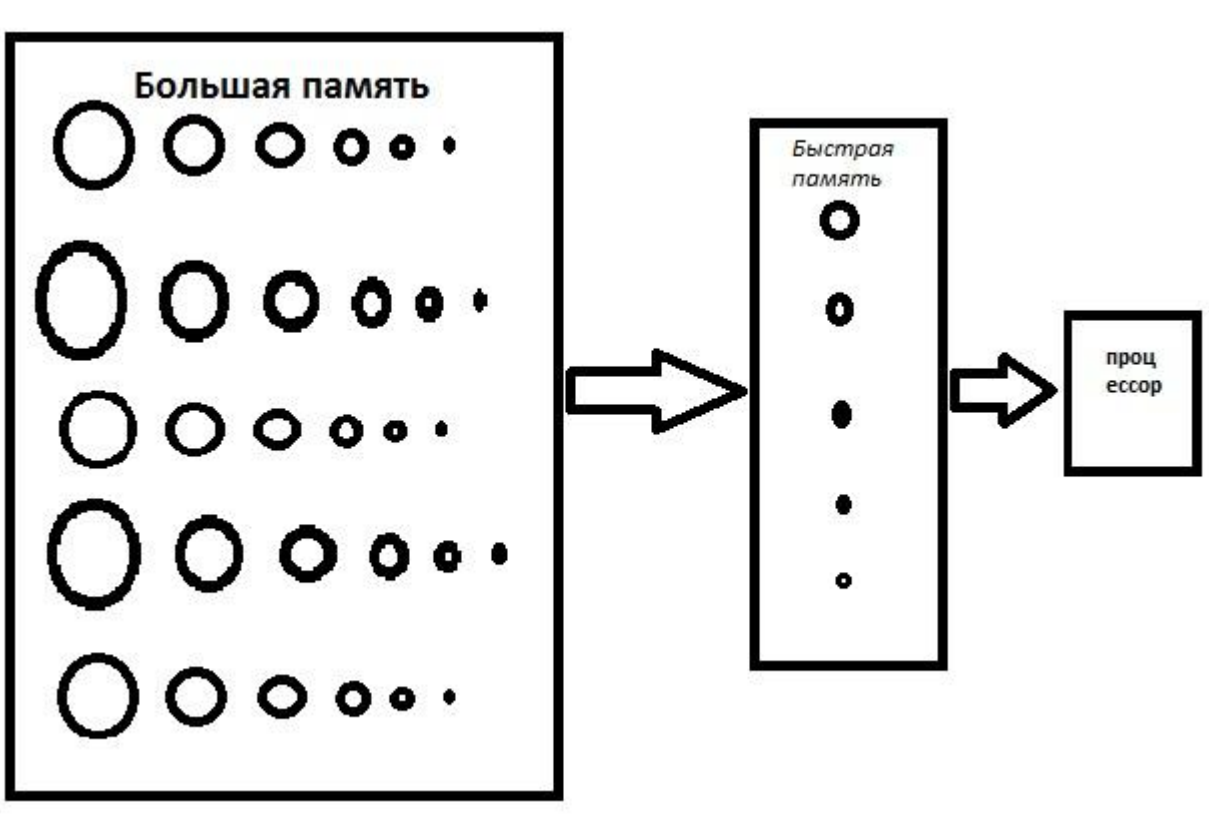
При слиянии 2 отсортированных массивов на каждое сравнение приходится одна запись в память.Процессор недогружен.



Задача сортировки массива, который не помещается в кэш.

Массив разбиваем на части, каждая из которых помещается в быстрой памяти и сортируем. Затем применяем множественное слияние.

МНОЖЕСТВЕННОЕ СЛИЯНИЕ



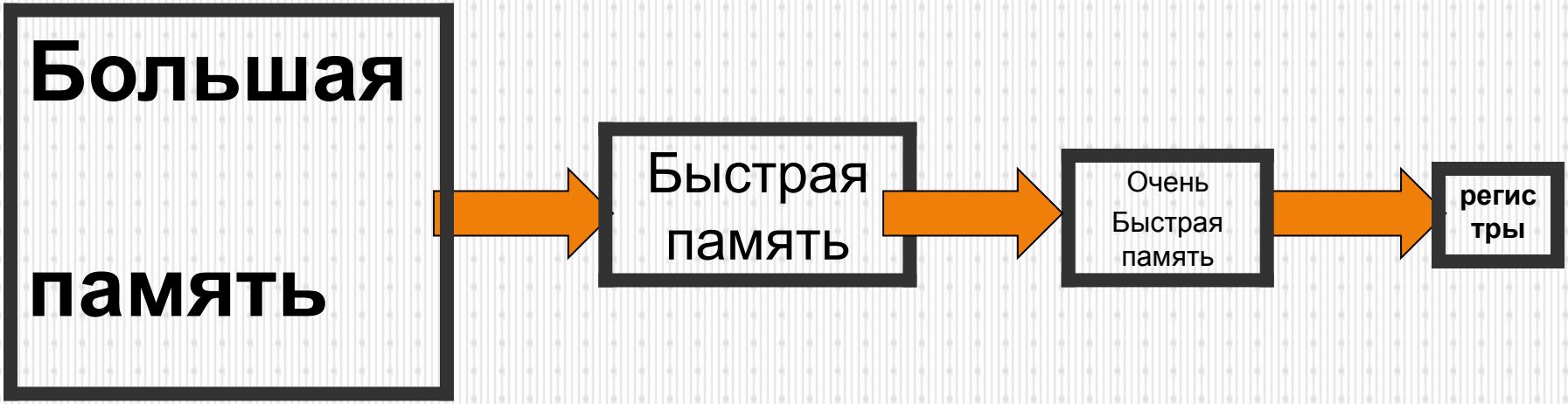
При множественном слиянии в быстрой памяти минимальные элементы каждого массива в специальной структуре (дерево). Чтений/записей большой памяти столько же, сколько и сортируемых элементов (а не в log(k), где k – количество массивов, как при попарном слиянии)

Программист должен сам разбивать задачу на подзадачи

При переходе к блочному перемножению матриц меняется порядок выполнения сложений при вычислении суммы (скалярного произведения). Это может привести к изменению погрешности округлений чисел с плавающей запятой и к переполнению.

иначе, это сделает компилятор, причем неоптимально

Модуль большой памяти
*
модуль быстрой памяти
*
модуль очень быстрой памяти



Несколько уровней блочности (подзадач)

- do i1 = 1, f
- do j1 = 1, f
- do k1 = 1, f
- do i = 1, m
- do j = 1, m
- do k = 1, m
- do ii = 1, d
- do jj = 1, d
- do kk = 1, d
- XX(i1,j1)(i,j)(ii,jj) = …
- Блоки могут быть прямоугольными (не квадратными).

Блочное распределение массивов - дополнение к блочному коду!

- Данные пересылаются в кэш память кэш-линейками, которые больше стандартных чисел.
- Элементы блоков должны быть в памяти рядом, чтобы совместно попасть в кэш.

- Заметим, что компиляторы используют стандарты при размещении матриц: ФОРТРАН размещает по столбцам, а Си и Паскаль – по строкам

Блочное размещение матрицы в оперативной памяти

$a_{11}\ a_{12}\ a_{13} \dots a_{1n}$ $a_{21}\ a_{22}\ a_{23} \dots a_{2n}$ $a_{31}\ a_{32}\ a_{33} \dots a_{3n}$
string 1 **string 2** **string 3**

$a_{11}\ a_{21}\ a_{31} \dots a_{n1}$ $a_{12}\ a_{22}\ a_{32} \dots a_{n2}$ $a_{13}\ a_{23}\ a_{33} \dots a_{n3}$
colomn 1 **colomn 2** **colomn 3**

$A_{11}\ A_{12}\ A_{13} \dots A_{1m}$ $A_{21}\ A_{22}\ A_{23} \dots A_{2n}$ $A_{31}\ A_{32}\ A_{33} \dots A_{3n}$
block 1 **block 2** **block 3**

A_{11}	A_{12}	A_{13}	A_{14}
A_{21}	A_{22}	A_{23}	A_{24}
A_{31}	A_{32}	A_{33}	A34
A_{41}	A_{42}	A_{43}	A_{44}

Стандарты сдерживают производительность?

- Существуют стандарты распределения данных в оперативной памяти для компиляторов. Например, компилятор языка ФОРТРАН размещает матрицы в оперативной памяти по столбцам, а Си – по строкам. Но при блочной обработке матриц желательно, чтобы элементы каждого блока (а не строки или столбца) оказывались в памяти рядом. В этом случае стандартное распределение массива может создавать кэш-промахи при обращении к нему.

Рекорды выходят за рамки стандартов

- - Рекорд быстродействия перемножения матриц Kazushige Goto, автора пакета GotoBLAS, превзойден в пакете PLASMA Джека Донгарры. Это удалось достичь за счет нестандартных блочных размещений данных при соблюдении принципа разбиения задач на подзадачи от быстрой памяти к большой.
- Еще более быстрая программа использует двойной блочный код и двойное блочное размещение массивов:
- http://ops.opsgroup.ru/downloads/dgemm.zip.

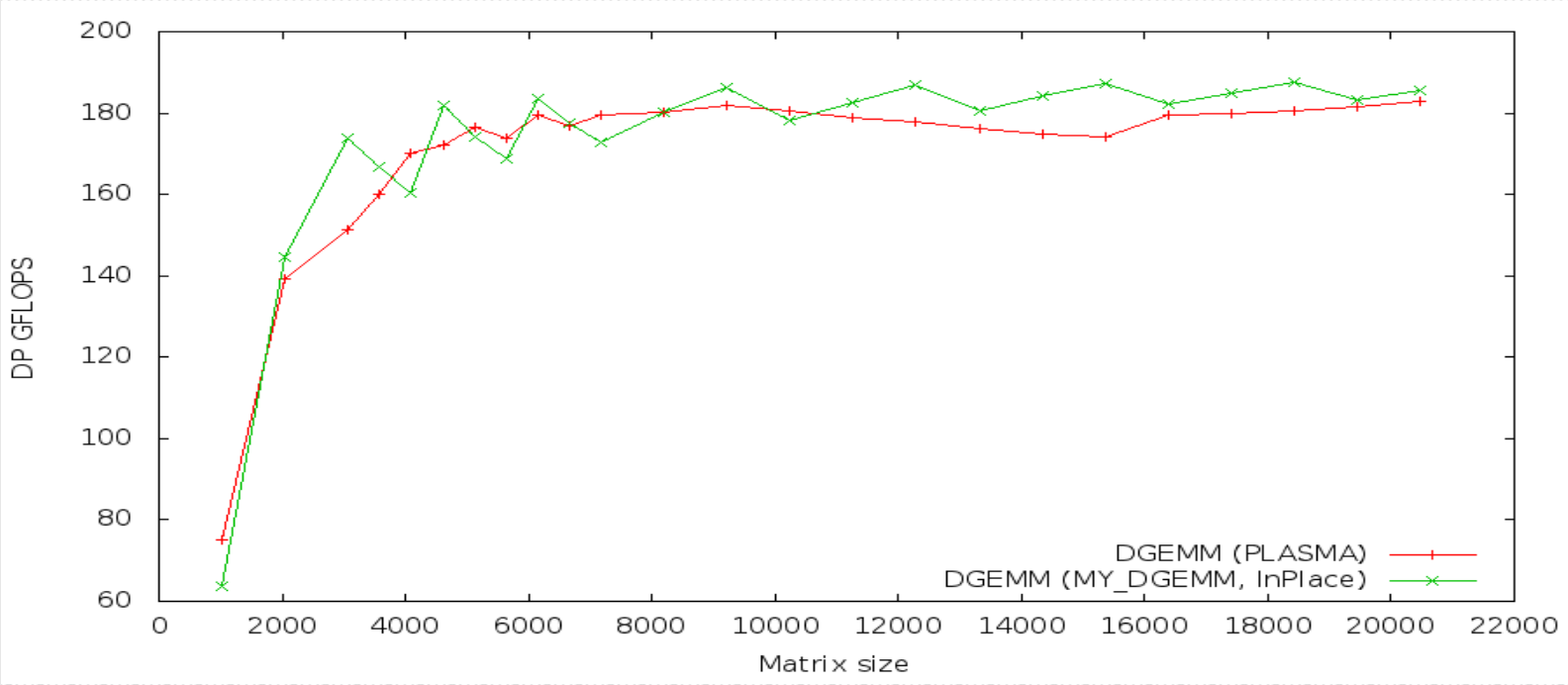
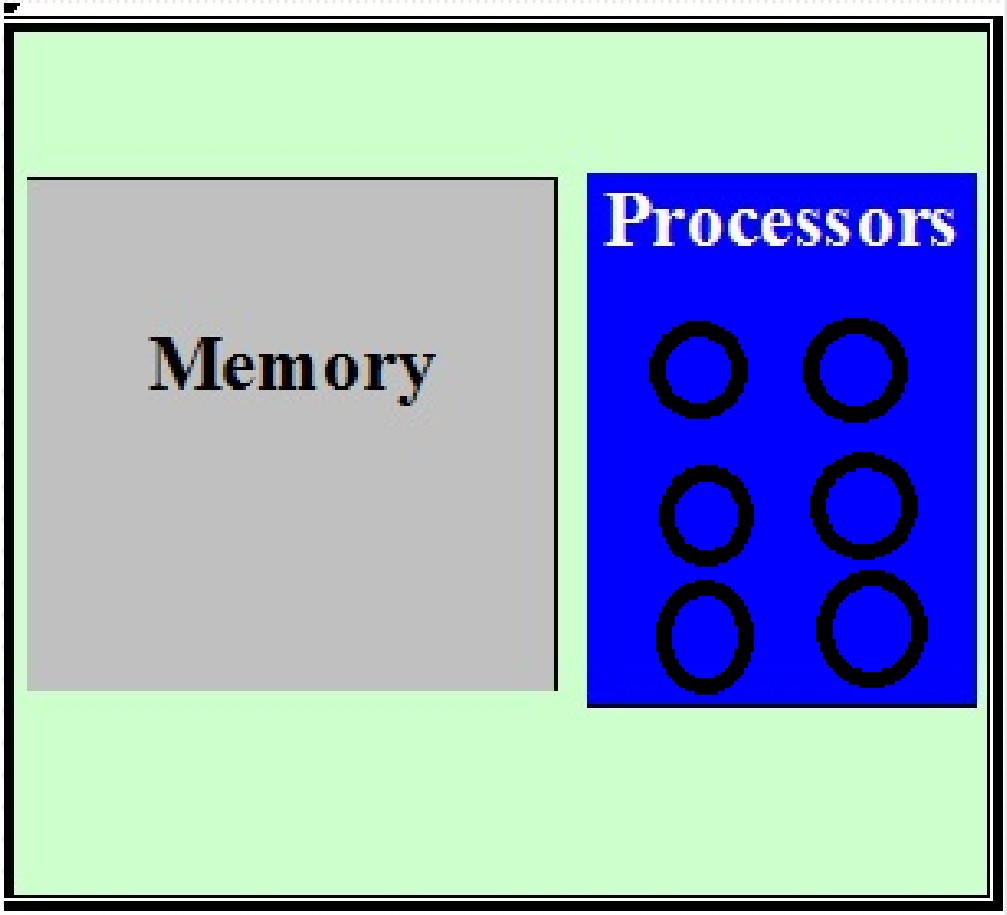


График сравнения производительности алгоритма OPS и пакета PLASMA

Оптимальный алгоритм перемножения матриц

- Оптимальный алгоритм перемножения матриц, по рекомендации японского программиста Goto, должен формироваться от регистров.
- Предположим, что в процессоре 8 ядер и каждое обрабатывает векторы с четырьмя координатами = 32 умножения одновременно. Тогда в регистры должны загружаться 4 числа одной матрицы (A) и 8 чисел другой матрицы (B): 4*8=32
- Исходя из объема самой быстрой L1 кэш-памяти формируются блоки матрицы A, имеющие по 4 строки, и блоки матрицы B, имеющие по 8 столбцов. Затем формируются блоки других уровней, исходя из объемов памяти других уровней.
- Видимо, стандартные процессоры подстроены под задачи LinPack, которые эквивалентны перемножению матриц.



Площадь кристалла делится на части под память и вычислители. Как разделить оптимально? – Для разных задач по разному

Под какие задачи делится площадь кристалла массовых процессоров?

- Видимо, стандартные процессоры подстроены под задачи LinPack, которые эквивалентны перемножению матриц.
- Если у Вас задача другого типа, чем перемножение матриц – то нужен другой процессор или ускоритель (GPU, ПЛИС,…).

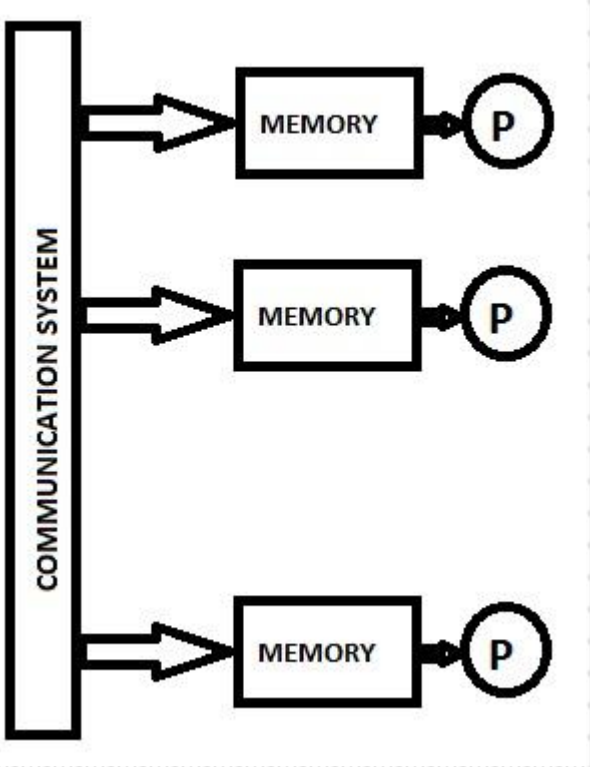
Оптимальное расположение полей в таблицах

aaaa	bbbb	cccc	dddd	eeee	ffff
aaaa	bbbb	cccc	dddd	eeee	ffff
aaaa	bbbb	cccc	dddd	eeee	ffff
aaaa	bbbb	cccc	dddd	eeee	ffff
aaaa	bbbb	cccc	dddd	eeee	ffff
aaaa	bbbb	cccc	dddd	eeee	ffff
aaaa	bbbb	cccc	dddd	eeee	ffff
aaaa	bbbb	cccc	dddd	eeee	ffff
aaaa	bbbb	cccc	dddd	eeee	ffff
aaaa	bbbb	cccc	dddd	eeee	ffff
aaaa	bbbb	cccc	dddd	eeee	ffff
aaaa	bbbb	cccc	dddd	eeee	ffff
aaaa	bbbb	cccc	dddd	eeee	ffff
aaaa	bbbb	cccc	dddd	eeee	ffff
aaaa	bbbb	cccc	dddd	eeee	ffff
aaaa	bbbb	cccc	dddd	eeee	ffff

aaaa	bbbb	eeee	cccc	dddd	ffff
aaaa	bbbb	eeee	cccc	dddd	ffff
aaaa	bbbb	eeee	cccc	dddd	ffff
aaaa	bbbb	eeee	cccc	dddd	ffff
aaaa	bbbb	eeee	cccc	dddd	ffff
aaaa	bbbb	eeee	cccc	dddd	ffff
aaaa	bbbb	eeee	cccc	dddd	ffff
aaaa	bbbb	eeee	cccc	dddd	ffff
aaaa	bbbb	eeee	cccc	dddd	ffff
aaaa	bbbb	eeee	cccc	dddd	ffff
aaaa	bbbb	eeee	cccc	dddd	ffff
aaaa	bbbb	eeee	cccc	dddd	ffff
aaaa	bbbb	eeee	cccc	dddd	ffff
aaaa	bbbb	eeee	cccc	dddd	ffff
aaaa	bbbb	eeee	cccc	dddd	ffff
aaaa	bbbb	eeee	cccc	dddd	ffff

- Рядом располагаются те поля, которые часто вместе встречаются в вызовах программы

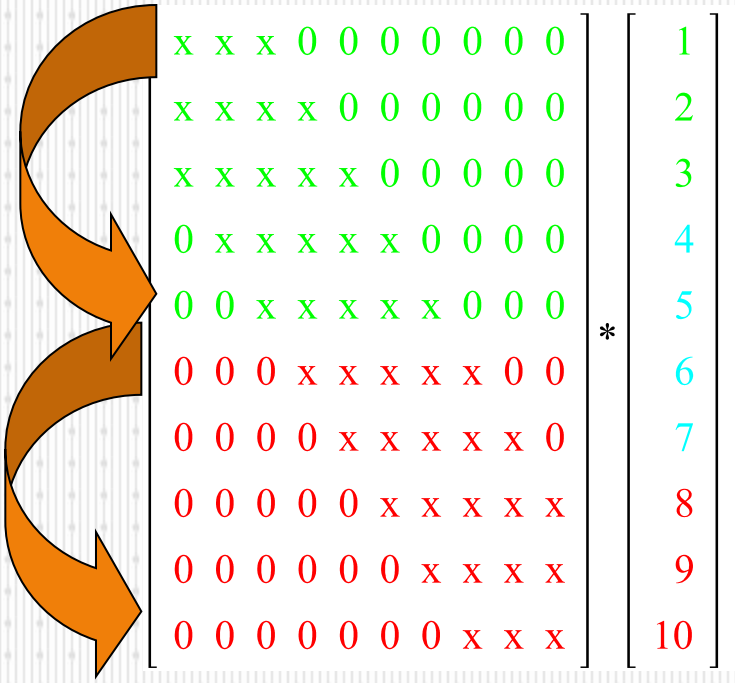
В распределенной памяти самая долгая операция – пересылка данных



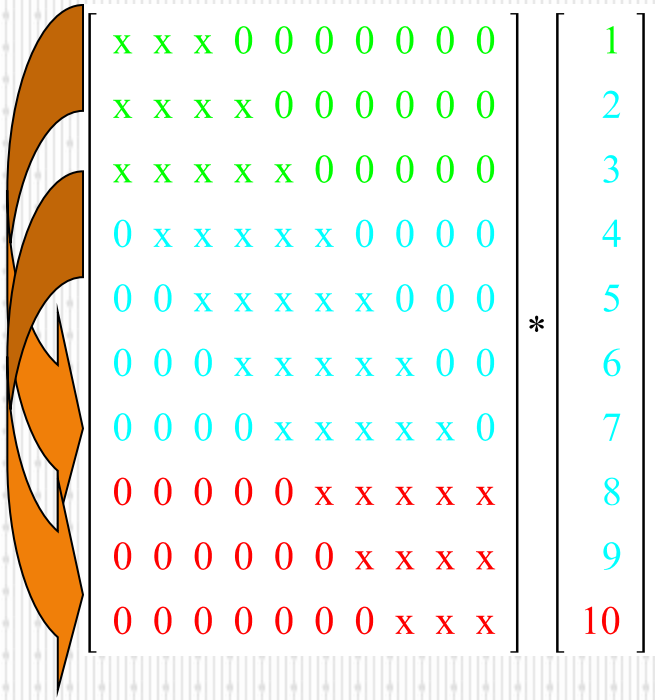
- Размещения с перекрытиями минимизируют количество межпроцессорных пересылок

Размещения с перекрытиями минимизируют количество межпроцессорных пересылок

- Для итерационного умножения ленточной матрицы на вектор (итераций: 10, размер вектора: 65536, ширина ленты: 513) размещение данных с перекрытием показывает выигрыш быстродействия на 22-24%.
- Для метода Якоби решения трехмерной задачи Дирихле для уравнения Лапласа (размерность сетки: 500*500*500, количество итераций - 2000) размещение данных с перекрытием показывает выигрыш быстродействия в два раза.



$$\begin{bmatrix} x & x & x & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ x & x & x & x & 0 & 0 & 0 & 0 & 0 & 0 \\ x & x & x & x & x & 0 & 0 & 0 & 0 & 0 \\ 0 & x & x & x & x & x & 0 & 0 & 0 & 0 \\ 0 & 0 & x & x & x & x & x & 0 & 0 & 0 \\ 0 & 0 & 0 & x & x & x & x & x & 0 & 0 \\ 0 & 0 & 0 & 0 & x & x & x & x & x & 0 \\ 0 & 0 & 0 & 0 & 0 & x & x & x & x & x \\ 0 & 0 & 0 & 0 & 0 & 0 & x & x & x & x \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & x & x & x \end{bmatrix} * \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \end{bmatrix}$$



$$\begin{bmatrix} x & x & x & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ x & x & x & x & 0 & 0 & 0 & 0 & 0 & 0 \\ x & x & x & x & x & 0 & 0 & 0 & 0 & 0 \\ 0 & x & x & x & x & x & 0 & 0 & 0 & 0 \\ 0 & 0 & x & x & x & x & x & 0 & 0 & 0 \\ 0 & 0 & 0 & x & x & x & x & x & 0 & 0 \\ 0 & 0 & 0 & 0 & x & x & x & x & x & 0 \\ 0 & 0 & 0 & 0 & 0 & x & x & x & x & x \\ 0 & 0 & 0 & 0 & 0 & 0 & x & x & x & x \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & x & x & x \end{bmatrix} * \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \end{bmatrix}$$

Размещение обычное и с перекрытием

LU-разложение матрицы с диагональным преобладанием (Л. Гервич)

Intel® Pentium® Processor E5300 (2M Cache, 2.60 GHz, 800 MHz FSB)

N = 1000

d	Блочный код	Блочный код + распределение массивов	Производительность
40	518	416	26%

Стандартный алгоритм	Производительность
1395	235%

Задача Дирихле. Итерационный блочный параллельный метод Якоби. (Л. Гервич)

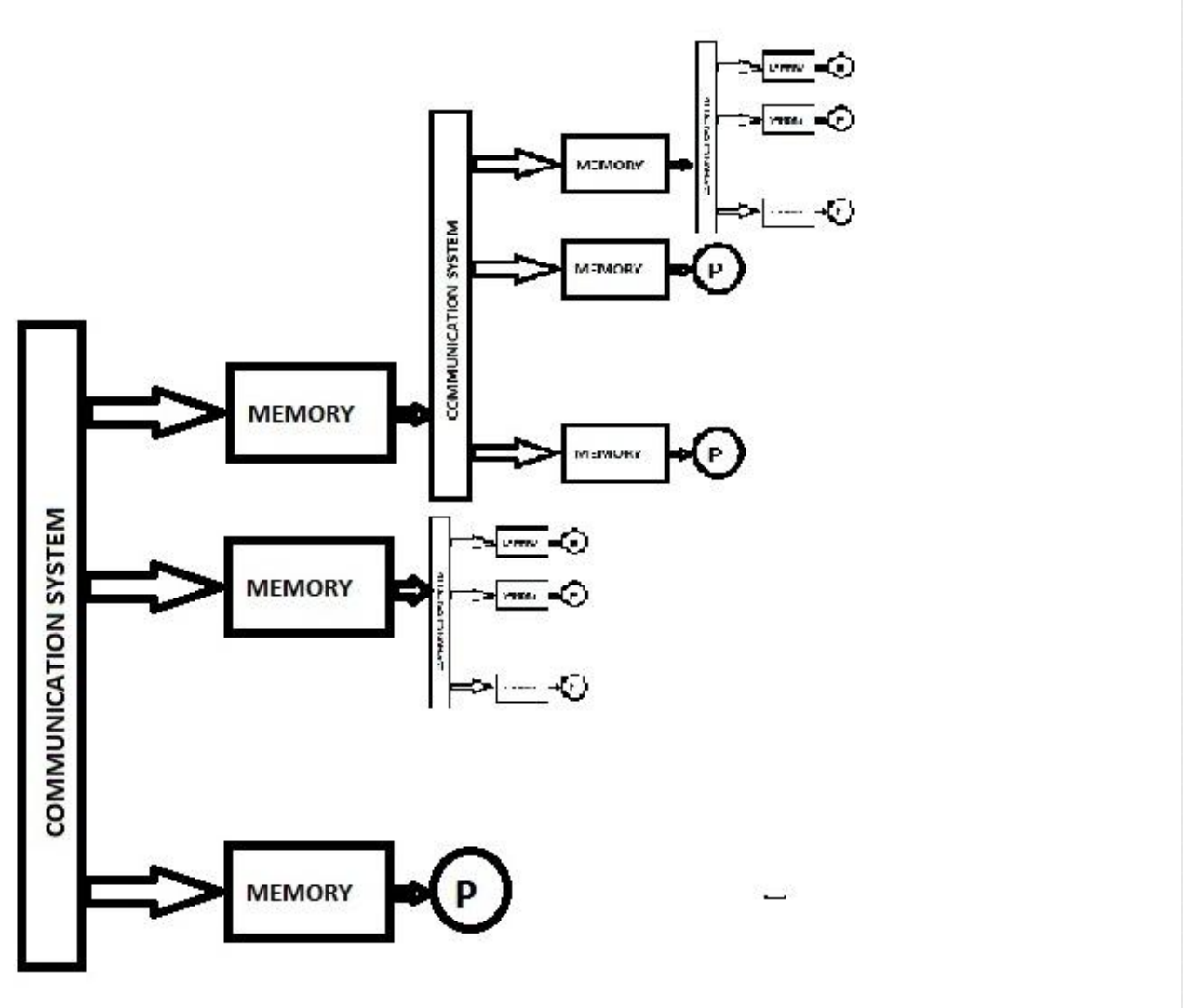
Intel® Pentium® Processor E5300 (2M Cache, 2.60 GHz, 800 MHz FSB)

N = 4000

d	Блочный код	Блочный код + распределение массивов	Производительность
40	164	133	23%

Стандартный алгоритм	Производительность
379	184%

Иерархии архитектуры – иерархия подзадач



Разбиение задачи на подзадачи

- - подзадачи должны формироваться так, чтобы в быстрой памяти малого объема помещались данные, с которыми выполняется много операций;
- - каждая подзадача должна быть максимально оптимизирована для своего уровня иерархии.
- - подзадач должно быть мало

Разбиение задачи на подзадачи: от регистров к большой памяти

- -Самые эффективные алгоритмы перемножения матриц используют методику *register packing* K. Goto, при выделение иерархии подзадач начинается от регистров к «Очень быстрой памяти», затем к «Быстрой памяти» и т.д.
- Эта методика может применяться и к другим задачам.
- Суть методики в том, что подзадача, которая попадает на вычислительное ядро с векторизатором должна быть максимально эффективна, выжимать максимальную производительность.
- Затем формируются подзадачи следующего уровня иерархии
- Если есть необходимость писать переносимую программу, то разбиение на подзадачи следует начинать с того уровня иерархии, который доступен высокоуровневому программированию.

Выравнивание последовательностей. Десятикратное ускорение (Ж. Абу-Халил)

Длина строк	Время , с				
	Алгоритм Хиршберга (последовательный)	Алгоритм Нидлмана-Вунша (последовательный)	Алгоритм с оптим. использованием памяти (параллельный)	Блочный алгоритм с оптим. использованием памяти (параллельный)	Блочный алгоритм (параллельный)
27532517	0.39	0.17	0.29	0.16	0.12
837674 88	3.34	1.30	1.23	0.75	0.50
1675214976	12.6	4.99	3.38	2.2	1.45
134016 119808	661.9	недостаточно памяти	135.3	88.9	47.7
268032 239616	2015.7	недостаточно памяти	639	362.7	171.1

Цена вопроса

- - параллельный код, использующий размещения данных с перекрытиями, имеет объем вдвое больше, чем «обычный» параллельный код;
- - объем кода программы рекордного по быстродействию перемножению матриц в 100 раз больше обычной последовательной программы.

- Цена достижения быстродействия: большие затраты времени, высокая квалификация разработчиков.

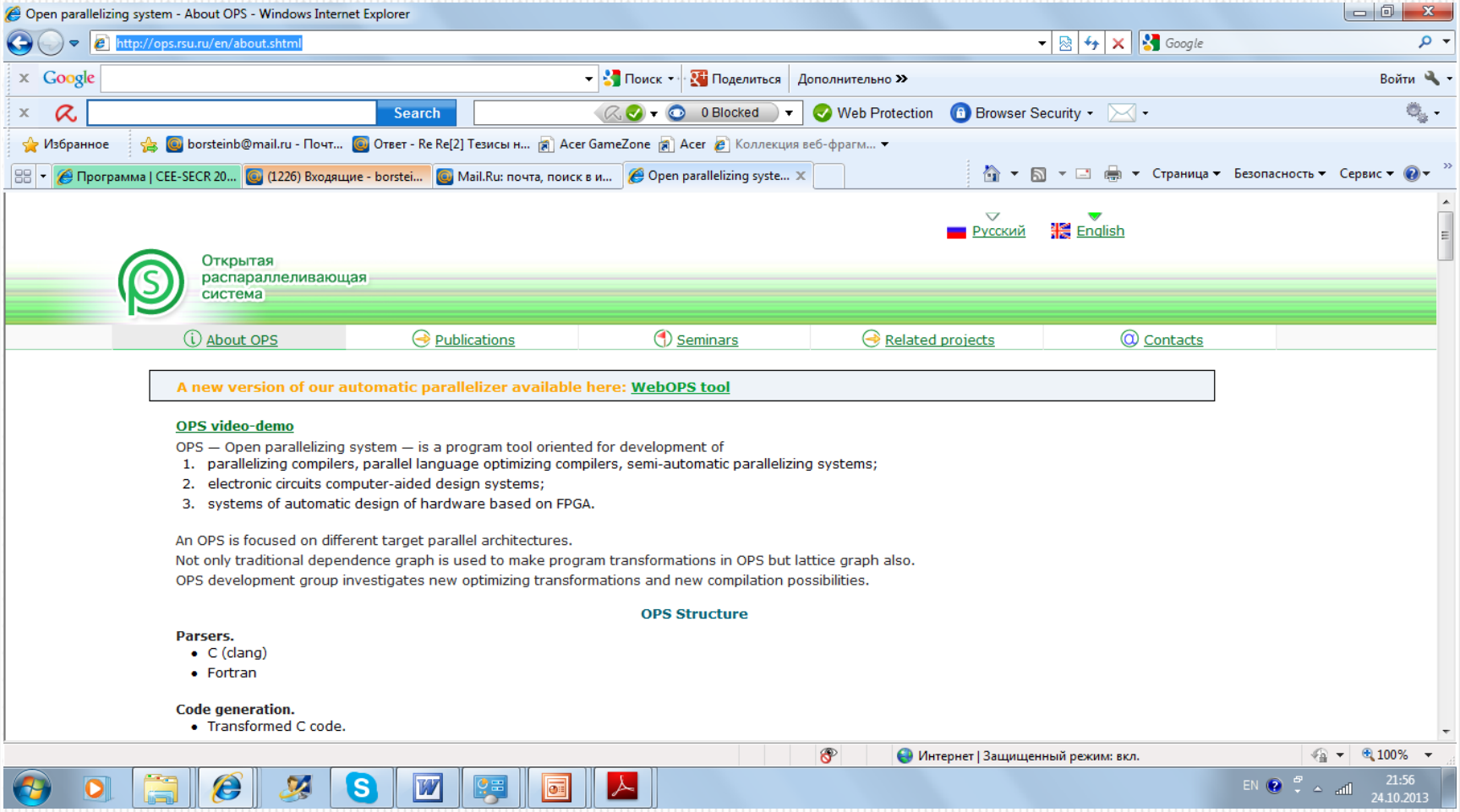
Некоторые черты эффективного ПО

- Соответствие описания иерархии подзадач иерархии архитектуры железа
- Структуры и распределение данных ориентированы на минимизацию обращений программы к памяти
- Использование новых инструментов автоматизации разработки быстрых программ.

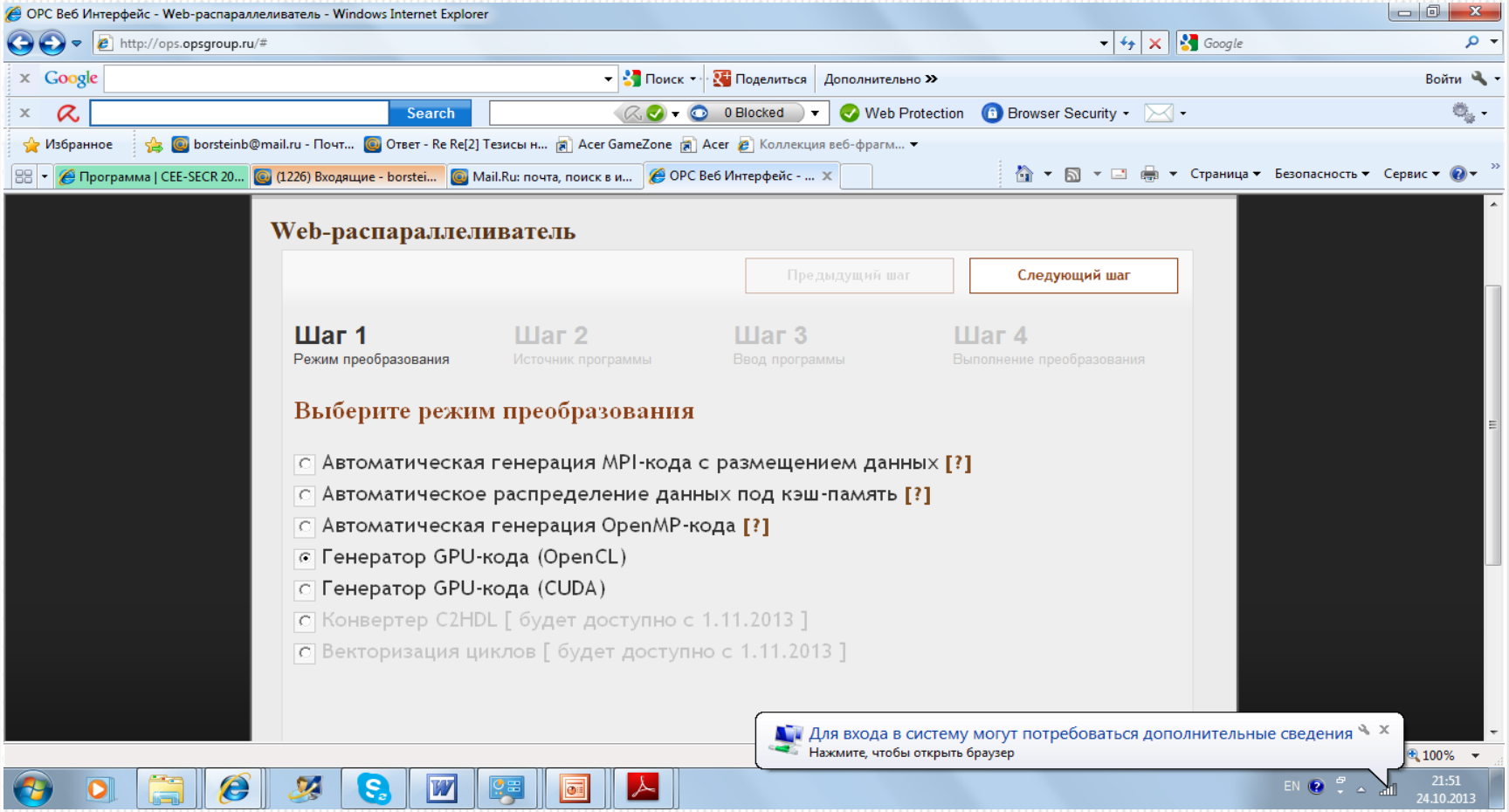
Черты перспективных инструментов автоматической оптимизации программ

- Диалоговый режим компиляции для определения информационных зависимостей.
- Использование решетчатых графов для одновременного перехода к блочному коду и распараллеливанию.

- Группа OPC работает над этим!



Оптимизирующая распараллеливающая система

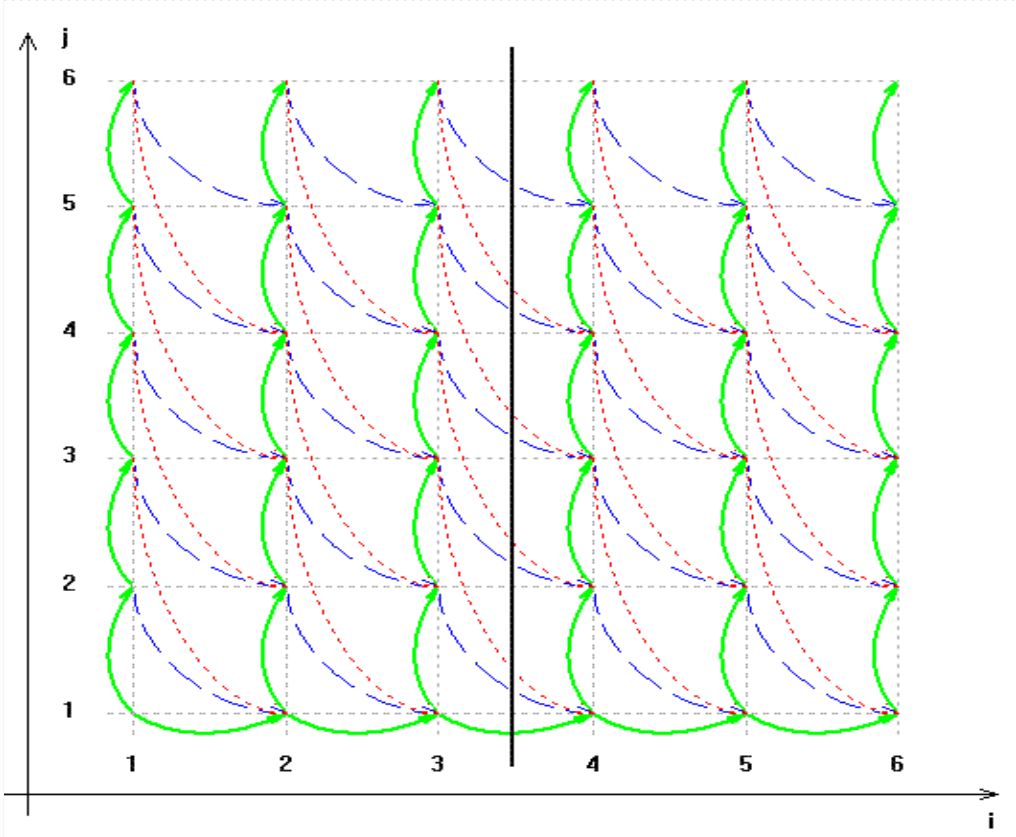


Веб-автораспараллеливатель. Меню

Веб-автораспараллеливатель. Результат распараллеливания

Решетчатый граф построен с помощью ОРС.

```
DO 99 i = 1, N
DO 99 j = 1, M
99      X(i+j) = A(i,j)*X(i+j-1)+B(i,j)
```

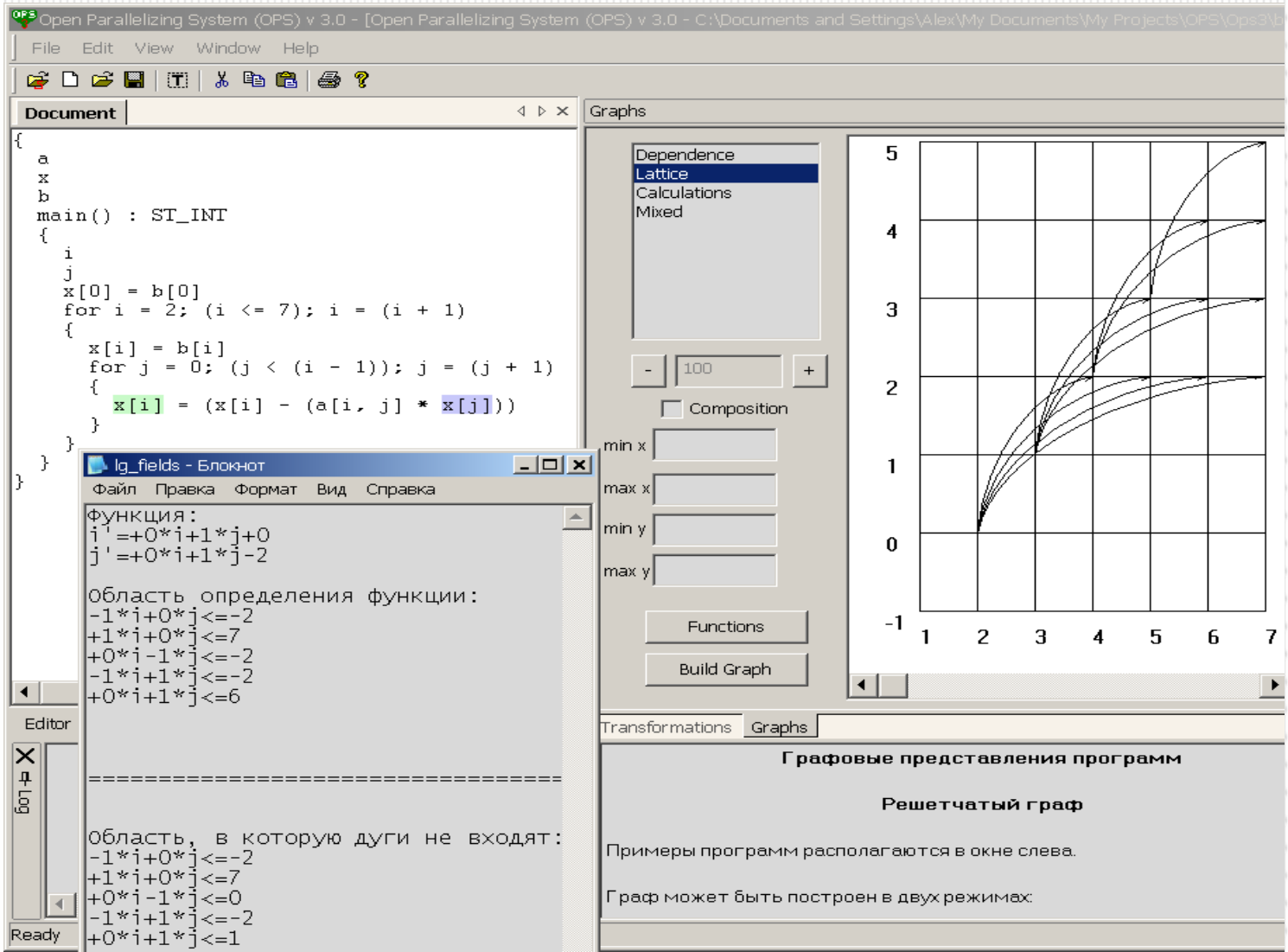


- При разбиении программы на два потока барьеры нужны на каждой итерации цикла со счетчиком j.

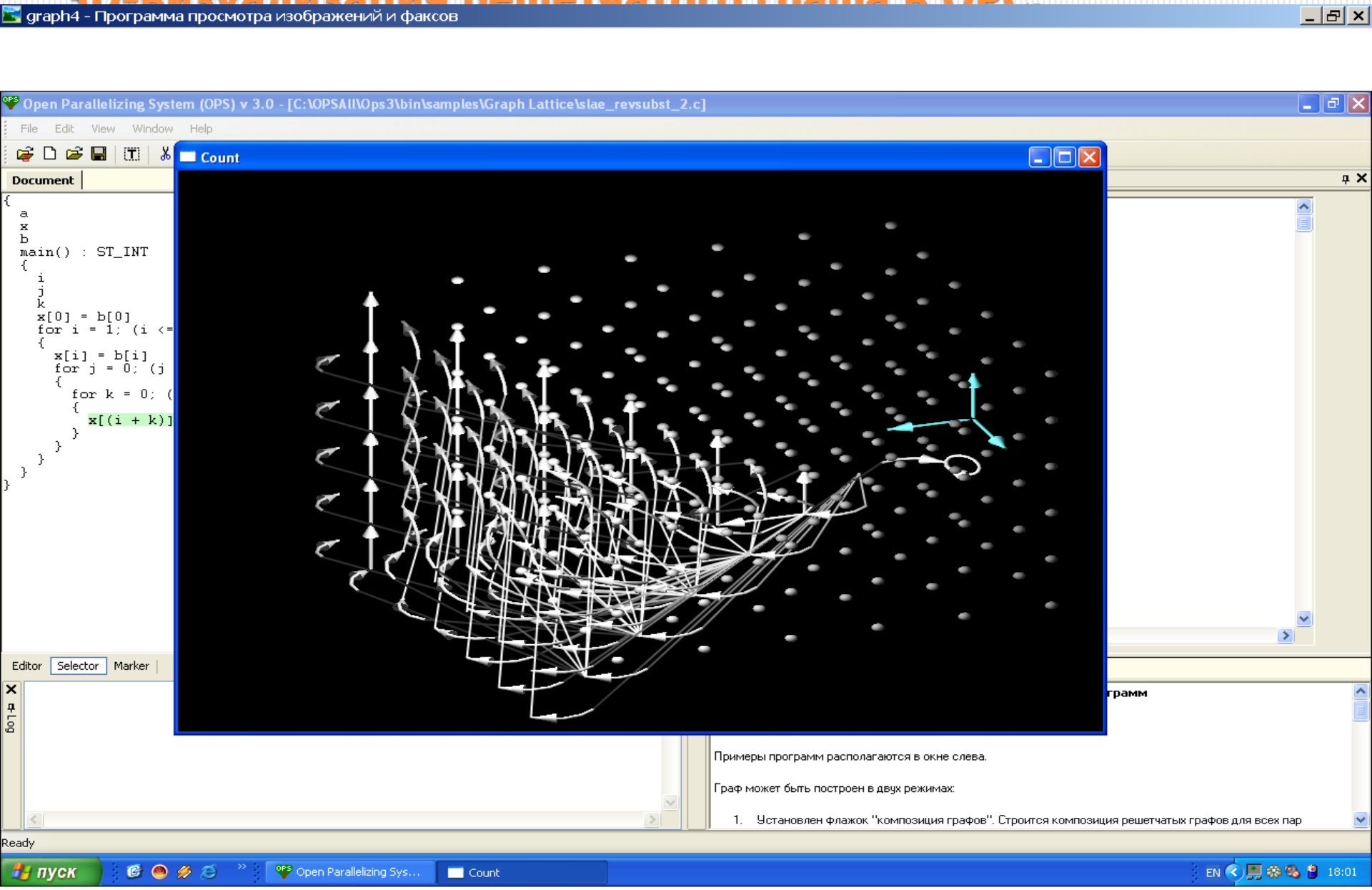
История решетчатых графов

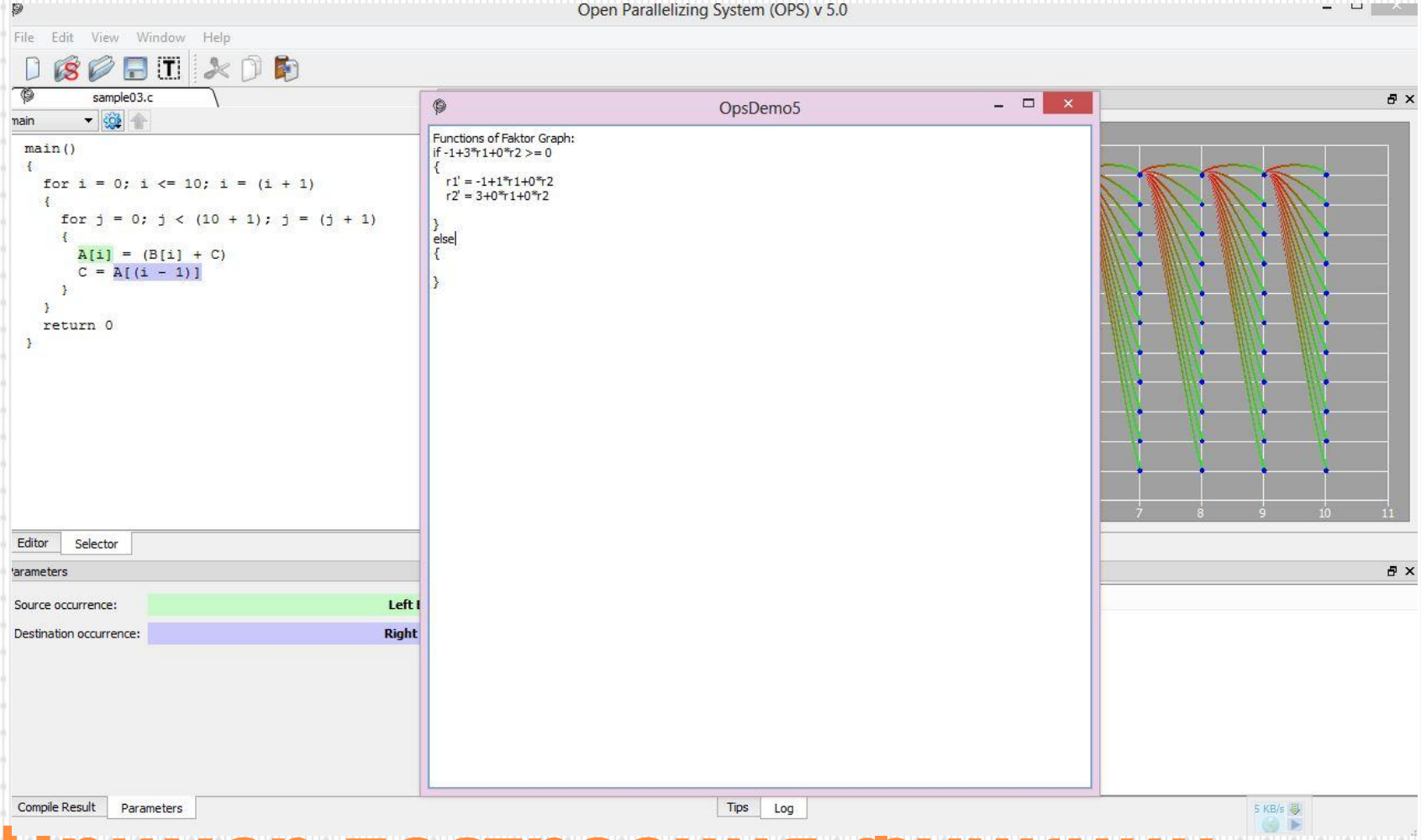
- Решетчатые графы использовались концептуально, для иллюстрации идей в методе гиперплоскостей Л. Лампорта, в методе параллелепипедов В. Вальковского и многих у других исследователей.
- Традиционные формы хранения графа в виде матрицы смежностей или в виде списка дуг не эффективны: прочтение такого графа может потребовать больше времени, чем последовательное исполнение программы, по которой этот граф построен.
- В конце 80-х годов был сделан прорыв в этом направлении: В.В. Воеводин и P. Feautrier научились хранить этот граф в виде функций, и разработали алгоритмы построения этих функций. У P. Feautrier функция строится для программ из линейного класса и содержит в своем определении не очень удобные для исследования условные операторы. У В.В. Воеводина функция является кусочно-линейной и определена на подмножестве практически значимых программ линейного класса.

Визуализация элементарного решетчатого графа для выделенной пары вхождений переменной в OPC. В специальном окне представлены функции, описывающие решетчатый граф.



3D-визуализация решетчатого графа в OPS





Пример построения функции фактор графа в OPC

- Пока придет новое поколение инструментов-компиляторов, об оптимизации следует беспокоиться самому.

Оптимизация: память и параллельность

Со-основателю Яндекса Илье Сегаловичу посвящается

- Ему было только 48 и нестало этим летом.
- Успешный программист любящий детей веселый человек

Спасибо за внимание!

http://ops.rsu.ru