# Элементы программирования экзафлопсных систем

Л. Гервич, Б. Штейнберг, М. Юрушкин

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

- Мы можем написать неоптимальную программу для настольного компьютера и запустить ее на сутки.
- Не оптимизировать программу для кластера – расточительно – дорого стоит машинное время.
- Дл экзафлопсного суперкомпьютера ПО должно быть рекордным по быстродействию.

# Компьютерам рекордной производительности – адекватные программы!

- Пользователю все равно за счет чего достигается быстродействие: за счет электроники или за счет оптимизации программы. Для суперкомпьютера стоимостью 1 миллиард 1% железа стоит 10 млн. – это окупает затраты на оптимизацию ПО
- *если знать, как оптимизировать!*

## *Две особенности вычислительных архитектур существенно влияют на оптимизацию программ: параллелизм и иерархия 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?***

# 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.
- 99 ядер 100-ядерного процессора будут простаивать!

# Memory Contention (cont #2)

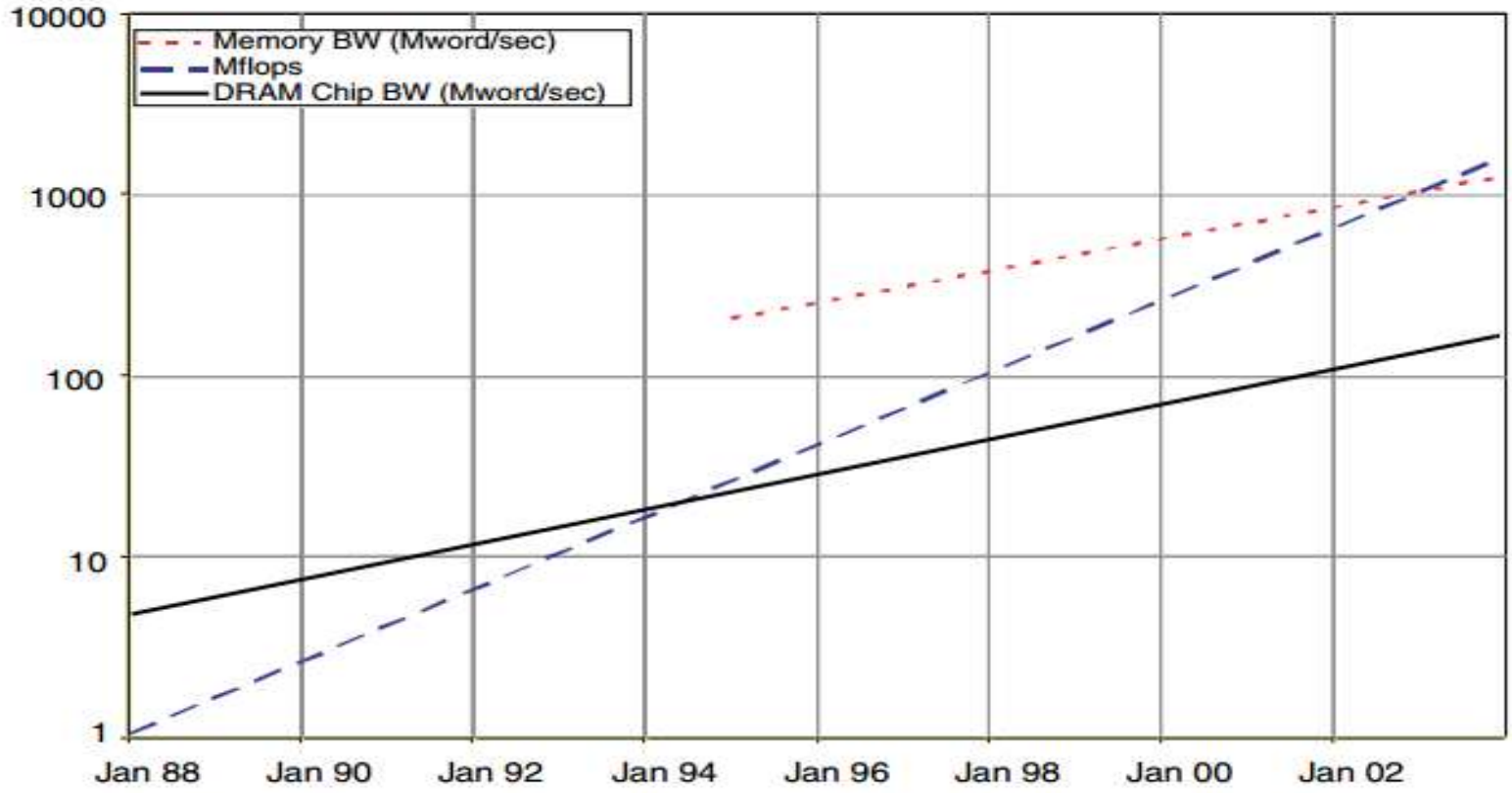


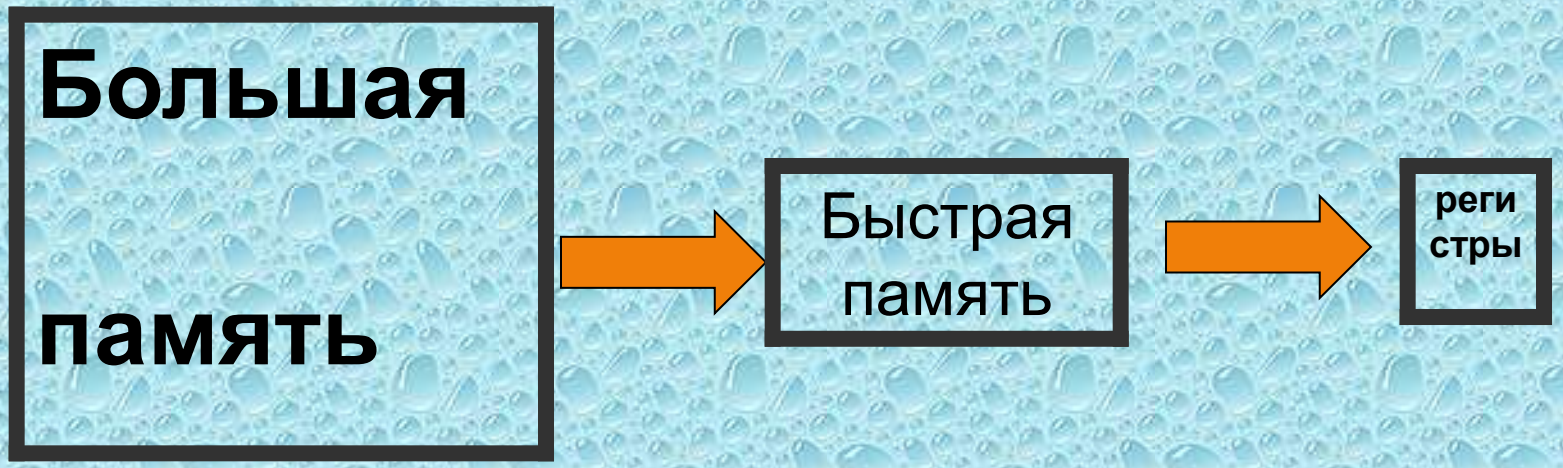

FIGURE 5.3 Arithmetic performance (Mflops), memory bandwidth, and DRAM chip bandwidth per calendar year.

**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**

# Большая, но медленная память
# *
# быстрая, но маленькая память
# (кэш или локальная)



# 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***.
- *Быстрая память не нужна.*

# 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 читает $3*n^2$ 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.

***Рассмотрим методы создания ПО рекордной производительности на примере задачи перемножения матриц***

# Представление матрицы в блочном виде

- 

| | | | |
|---|---|---|---|
| $A_{11}$ | $A_{12}$ | $A_{13}$ | $A_{14}$ |
| $A_{21}$ | $A_{22}$ | $A_{23}$ | $A_{24}$ |
| $A_{31}$ | $A_{32}$ | $A_{33}$ | 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)
-
- Блоки подбираются так, чтобы 3 блока поместились в быстрой памяти

# 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).

# Программист должен сам разбивать задачу на подзадачи

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

- Два внутренних цикла умножают k-ый СТОЛБЕЦ A(i,k) матрицы A на k-ую СТРОКУ B(k,j) матрицы B. Результирующая n*n матрица прибавляется к матрице X. СТОЛБЕЦ и СТРОКА – это 2*$n$ данных, а операций с ними - $n^2$ ! Мало обращений к памяти и много операций!
- Здесь нет обычного  software pipeline, поскольку одно данное является аргументом многих операций.

# 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.
- Программа одного из соавторов отстает от пиковой на 5 %, но использует нестандартное размещение матриц. Такое размещение можно получить нестандартным компилятором или специальными прагмами
- Видимо, стандартные процессоры подстроены под задачи LinPack, которые эквивалентны перемножению матриц.

# 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.

# Стандарты сдерживают производительность?

- Стандарты компиляторов.
- в языке ФОРТРАН матрицы размещаются в оперативной памяти по столбцам,
- в Си – по строкам.
- Но при блочной обработке матриц желательно, чтобы элементы каждого блока (а не строки или столбца) оказывались в памяти рядом.
- 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 (одного из соавторов данного доклада) использует двойной блочный код и двойное блочное размещение массивов:
- http://ops.opsgroup.ru/downloads/dgemm.zip.

# Разбиение задачи на подзадачи

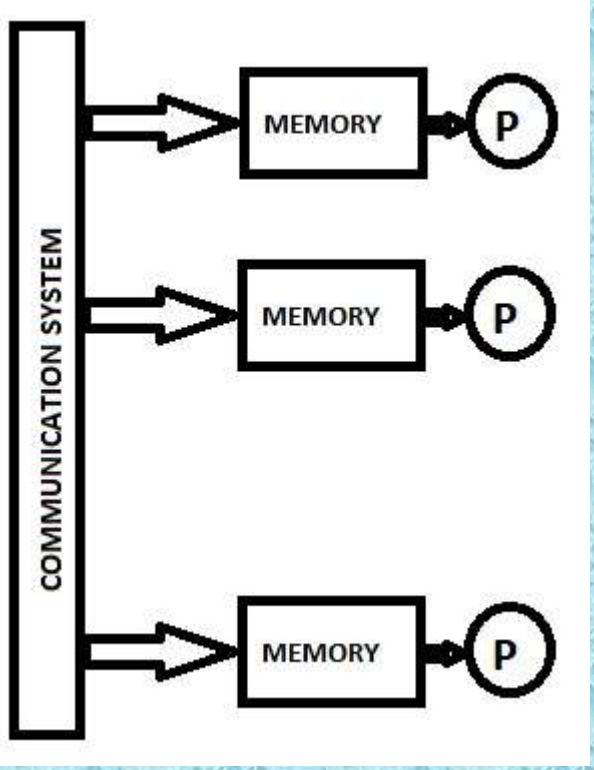
- - подзадачи должны формироваться так, чтобы в быстрой памяти малого объема помещались данные, с которыми выполняется много операций;
- - каждая подзадача должна быть максимально оптимизирована для своего уровня иерархии.
- - подзадач должно быть мало

Ближайшее время теория алгоритмов должна развиваться в направлении разбиения задач на подзадачи.

# 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.

# 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.

# 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)

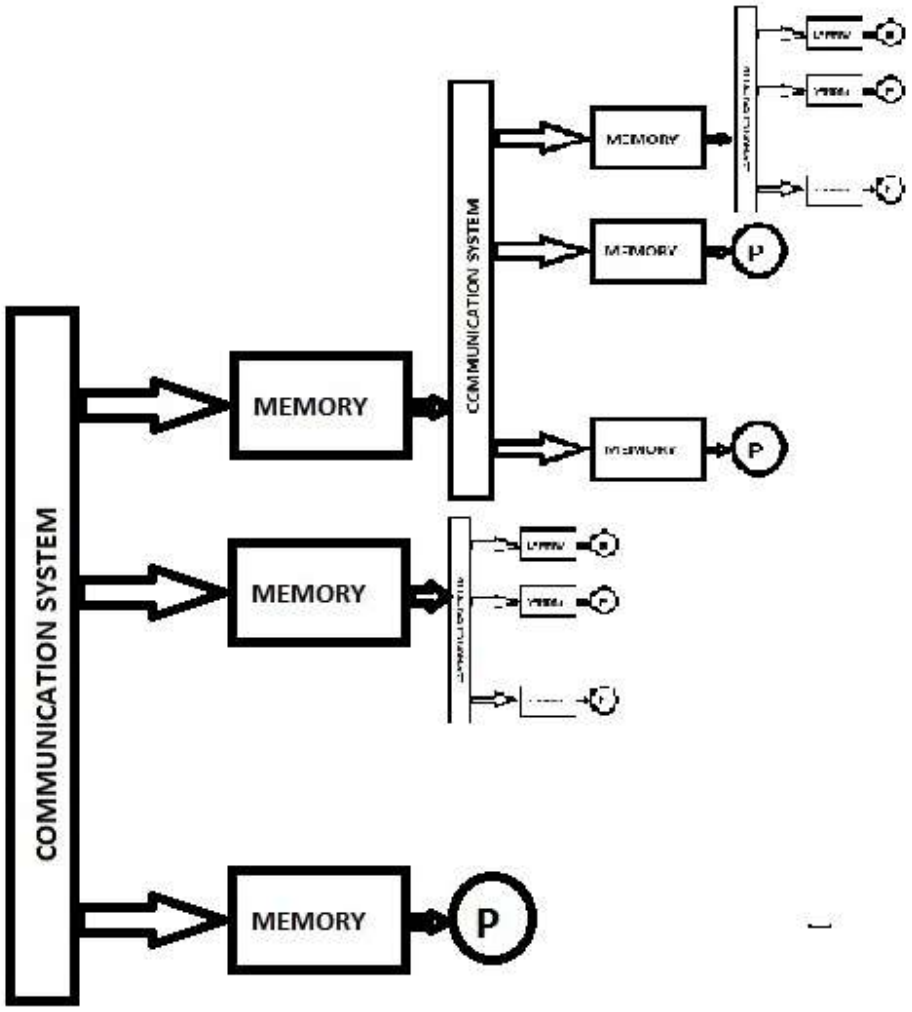
Intel® Pentium® Processor E5300 (2M Cache, 2.60 GHz, 800 MHz FSB)

N = 4000

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

| Стандартный алгоритм | Производительность |
|---|---|
| 379 | 184% |

# Иерархии архитектуры – иерархия подзадач



# 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 |
| 16752 14976 | 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 |

# Некоторые черты эффективного ПО экзафлопсных суперкомпьютеров

- 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.

- **Цена вопроса :**
***большие затраты времени,***
***высокая квалификация разработчиков.***

- - 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.

# Спасибо за внимание!

http://ops.rsu.ru