Мастер-класс по моделям и инструментам программирования для HPC

3 декабря 2009

Эксперт:
Сергей Немнюгин (СПбГУ)

Программные инструменты Intel – фактический стандарт инструментов в области HPC.

Модели программирования в HPC

С.А.Немнюгин

Д.Пузырёв, П.Кавригин, С.Толушкин

Санкт-Петербургский государственный университет

Кафедра вычислительной физики

snemnyugin@mail.ru

День HPC-Intel, Челябинск, 2009

Все грани высокопроизводительных вычислений

Производительность – одна из метрик эффективности программного обеспечения.

Другие – *масштабируемость* и т.д.

Количественный критерий производительности – число «полезных» операций, выполняемых в единицу времени или время, затрачиваемое на выполнение фиксированного объема «полезной» работы.

«Полезные» операции должны составлять большую часть операций, выполняемых вычислительной программой. Это – цель оптимизации.

Все грани высокопроизводительных вычислений

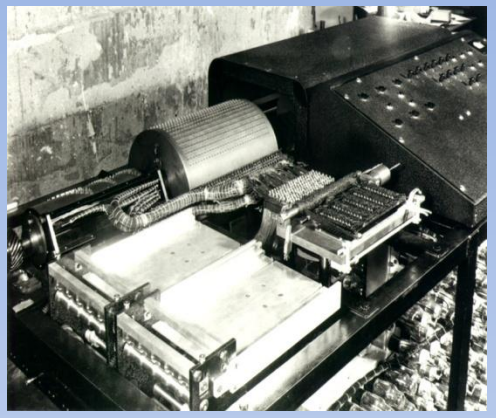
8 граней оптимизации:

- Выбор вычислительной системы
- Выбор модели
- Использование эффективных вычислительных алгоритмов
- Использование приёмов написания оптимального кода
- Использование оптимизированных библиотек
- Оптимизация при компиляции
- Оптимизация готовой программы на основе анализа её выполнения
- Параллельное программирование

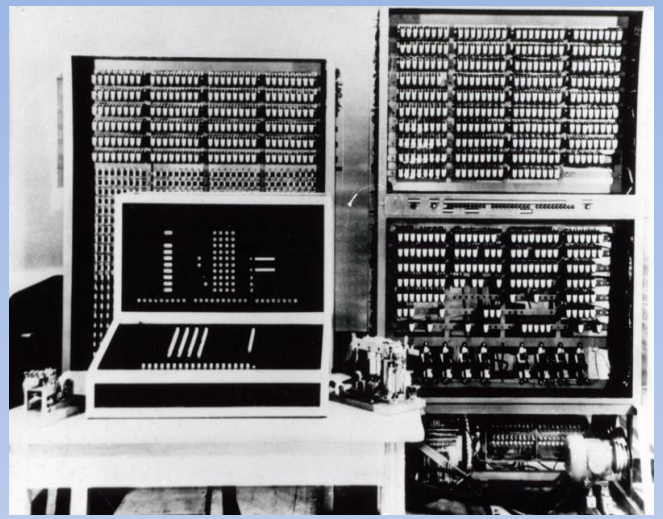
Выбор вычислительной системы

Эволюция высокопроизводительных вычислительных систем

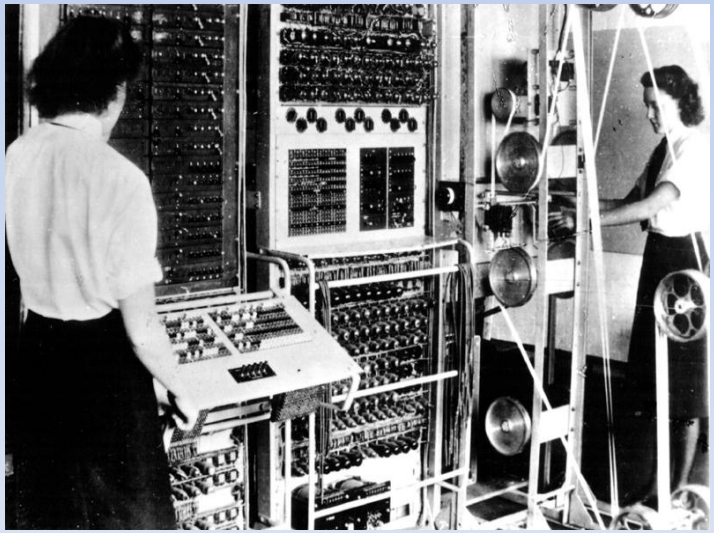
- Счеты.
- Логарифмическая линейка.
- Арифмометр (Томас Кольмар).
- Вычислительная клавишная машина



1939 год. Вычислительная машина Атанасова-Берри



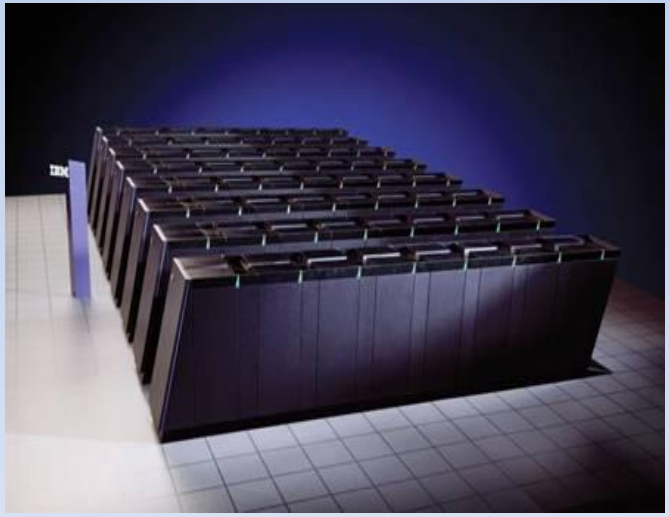
1941 год. Вычислительная машина Конрада Цузе



1944 год. Colossus

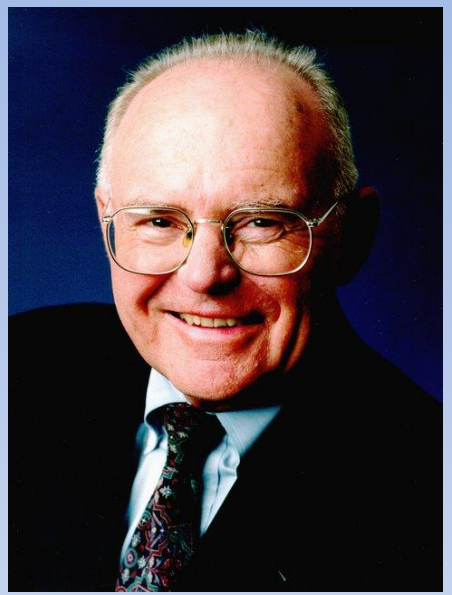
1946 год. ENIAC

1964 год. CDC 6600 (Сеймур Крэй)

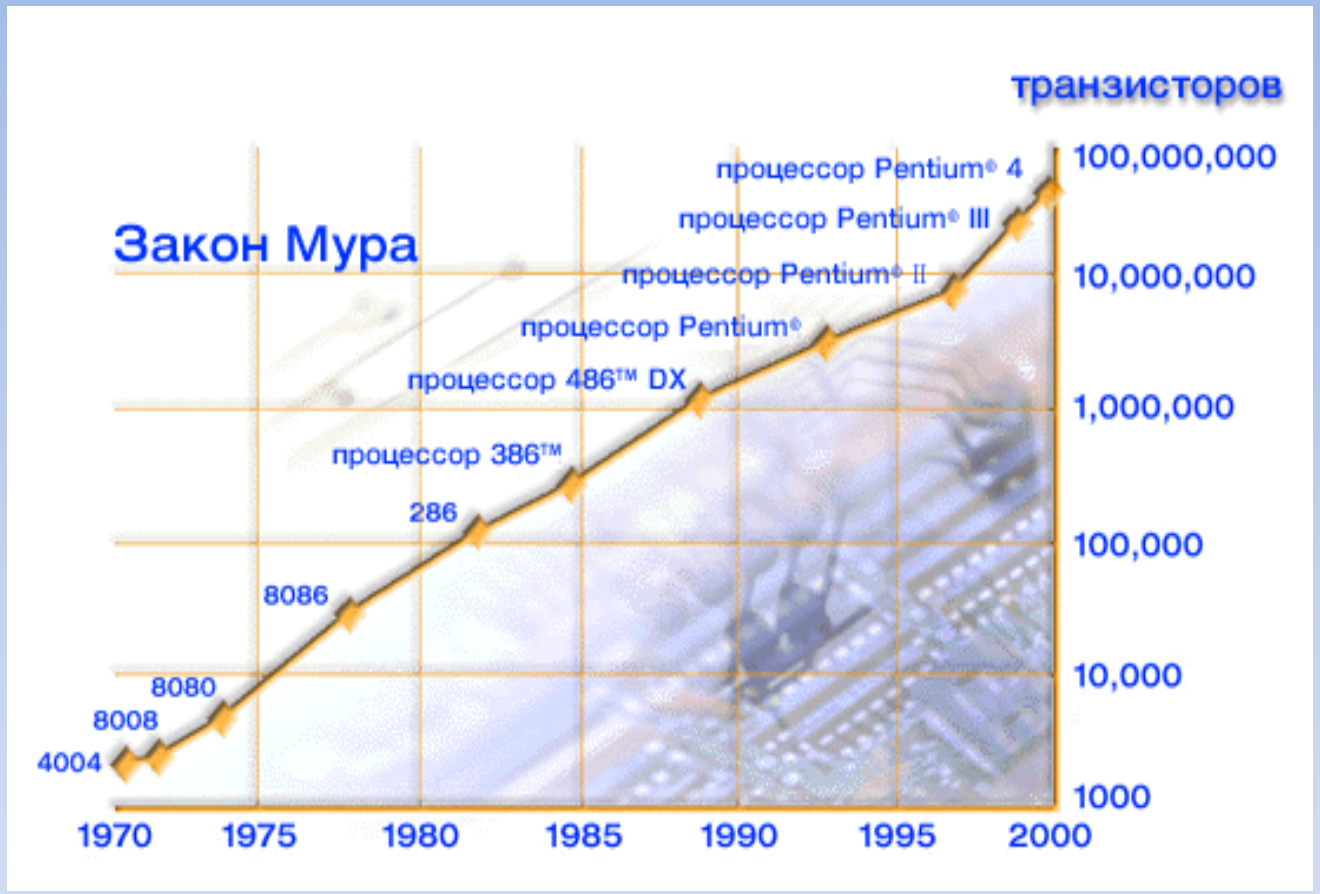


Blue Gene

Roadrunner (Лос-Аламосская Национальная Лаборатория, 1.1 петафлопс)



Гордон Мур


транзисторов
Закон Мура
процессор Pentium® 4
процессор Pentium® III
процессор Pentium® II
процессор Pentium®
процессор 486™ DX
процессор 386™
286
8086
8080
8008
4004
100,000,000
10,000,000
1,000,000
100,000
10,000
1000
1970
1975
1980
1985
1990
1995
2000

Модель

Моделирование системы взаимодействующих частиц методом молекулярной динамики

В вычислительных науках процесс создания высокопроизводительного приложения может начинаться с формулировки модели, то есть задолго до написания, отладки и верификации программного кода.

Метод молекулярной динамики является одним из основных методов моделирования динамики систем взаимодействующих между собой частиц.

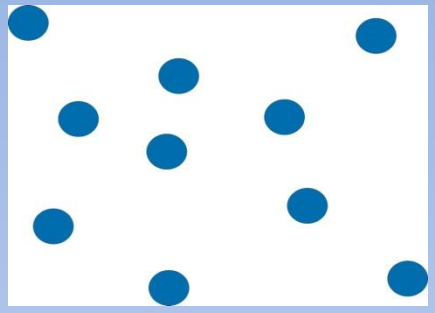
Применяется в молекулярной физике, астрофизике и других науках.

Динамика системы описывается уравнениями классической механики.

Стандартные предположения модели:

1. классическая динамика;

2. парные взаимодействия.



Рассматривается система N частиц, динамика которых описывается вторым законом Ньютона:

$$\mathbf{F}_i = m_i \mathbf{a}_i$$

Здесь $\mathbf{F}_i$ - равнодействующая всех сил, действующих на i-ю частицу, m_i - ее масса и $\mathbf{a}_i$ - ускорение, с которым частица двигается под действием силы. Это обыкновенное дифференциальное уравнение второго порядка, которое можно записать в виде:

$$\mathbf{F}_i = m_i \frac{d^2 \mathbf{r}_i}{dt^2}$$

где $\mathbf{r}_i$ - радиус-вектор i-й частицы.

Равнодействующая сил, является суммой парных взаимодействий i-й частицы со всеми остальными частицами:

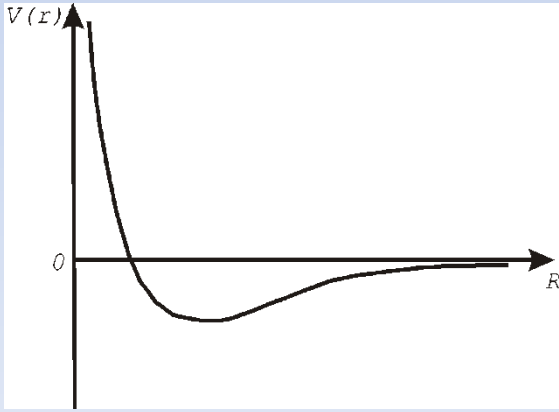
$$\mathbf{F}_i = \sum_{j=1}^{i-1} \mathbf{F}(\mathbf{r}_i, \mathbf{r}_j) + \sum_{j=i+1}^{N} \mathbf{F}(\mathbf{r}_i, \mathbf{r}_j)$$

Сила взаимодействия связана с потенциалом взаимодействия. В расчетах молекулярных систем часто используют потенциал Леннарда-Джонса:

$$V(r) = 4\varepsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right]$$

Потенциал Леннарда-Джонса хорошо описывает взаимодействие между атомами аргона.

$$\mathbf{F}(\mathbf{r}_i, \mathbf{r}_j) = \frac{24\ \ \mathbf{r}_i - \mathbf{r}_j}{\left|\mathbf{r}_i - \mathbf{r}_j\right|^2}\left[\frac{2}{\left|\mathbf{r}_i - \mathbf{r}_j\right|^{12}} - \frac{1}{\left|\mathbf{r}_i - \mathbf{r}_j\right|^{6}}\right]$$



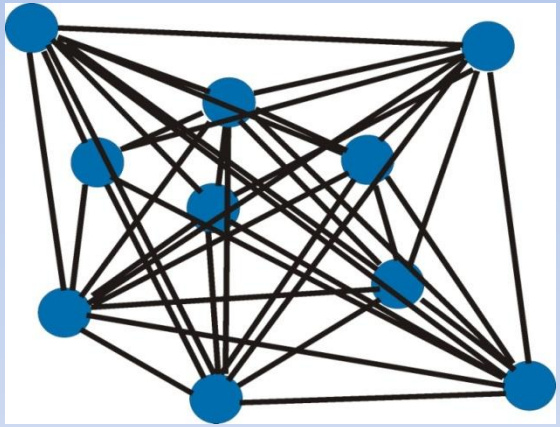
Моделирование (алгоритм Верле)

Дискретизация по времени:

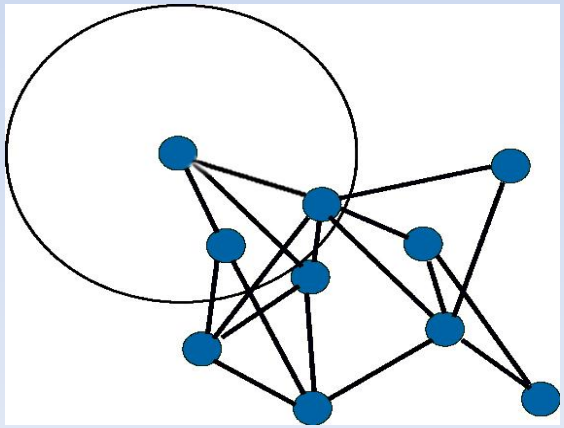
$$\mathbf{r}_i^{k+1} = 2\mathbf{r}_i^k - \mathbf{r}_i^{k-1} + \mathbf{F}(\mathbf{r}_i^k)\frac{\Delta t^2}{m_i} \qquad \mathbf{r}_i^k = \mathbf{r}_i(t^k)$$

Наиболее трудоёмкая операция – вычисление сил.

Трудоемкость вычисления сил ~n^2



Введение радиуса обрезания приводит к уменьшению трудоемкости. Это – модификация модели.



Оптимизация, не требующая модификации модели

Вычисление сил парных взаимодействий (система из 5 частиц). На чем можно сэкономить?

$$\mathbf{F}_1 = \mathbf{F}_{12} + \mathbf{F}_{13} + \mathbf{F}_{14} + \mathbf{F}_{15}$$
$$\mathbf{F}_2 = \mathbf{F}_{21} + \mathbf{F}_{23} + \mathbf{F}_{24} + \mathbf{F}_{25}$$
$$\mathbf{F}_3 = \mathbf{F}_{31} + \mathbf{F}_{32} + \mathbf{F}_{34} + \mathbf{F}_{35}$$
$$\mathbf{F}_4 = \mathbf{F}_{41} + \mathbf{F}_{42} + \mathbf{F}_{43} + \mathbf{F}_{45}$$
$$\mathbf{F}_5 = \mathbf{F}_{51} + \mathbf{F}_{52} + \mathbf{F}_{53} + \mathbf{F}_{54}$$

3-й закон Ньютона

$$\mathbf{F}_{ij} = -\mathbf{F}_{ji}$$

Любую парную силу достаточно вычислить только один раз!

Количество вычислений парных сил можно уменьшить в два раза!

Алгоритмы

Выбор подходящего алгоритма важен для производительности приложения. Можно потратить много времени на оптимизацию плохого алгоритма не добившись успеха

Эффективность алгоритма определяется количеством операций, требуемых для его выполнения

Пузырьковая сортировка: $O(n^2)$
Быстрая сортировка: $O(n \log n)$

	256 элементов	**1000 элементов**	**10000 элементов**
Пузырьковая сортировка	65 000	1 000 000	100 000 000
Быстрая сортировка	2 048	9 965	133 000

Быстрая сортировка:

- из массива выбирается некоторый опорный элемент a[i],
- запускается процедура разделения массива, которая перемещает все ключи, меньшие, либо равные a[i], влево от него, а все ключи, большие, либо равные a[i] - вправо,
- теперь массив состоит из двух подмножеств, причем левое меньше, либо равно правому,
- для обоих подмассивов: если в подмассиве более двух элементов, рекурсивно запускаем для него ту же процедуру.

В итоге получится полностью отсортированная последовательность.

Матричное умножение

- *Прямой метод*. Трудоемкость $O(n^3)$
- *Метод Штрассена*. Трудоемкость $O(n^{2.81})$
- *Метод Копперсмита-Винограда*. Трудоемкость $O(n^{2.38})$

Прямой метод

$$C = AB \qquad c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

Трудоемкость $O(n^3)$

Метод Штрассена

Решаемая задача – вычисление произведения матриц
A × B при больших A и B. Обе матрицы – квадратные, размера n × n, а n – степень двойки. Тогда можно воспользоваться рекурсивным алгоритмом.
Введем обозначения:

$$A \times B = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & g \\ f & h \end{pmatrix} = \begin{pmatrix} ae + bf & ag + bh \\ ce + df & cg + dh \end{pmatrix} = \begin{pmatrix} r & s \\ t & u \end{pmatrix}$$

i	A_i	B_i	$P_i = A_i \times B_i$
1	a	$g - h$	$ag - ah$
2	$a + b$	h	$ah + bh$
3	$c + d$	e	$ce + de$
4	d	$f - e$	$df - de$
5	$a + d$	$e + h$	$ae + ah + de + dh$
6	$b - d$	$f + h$	$bf + bh - df - dh$
7	$a - c$	$e + g$	$ae + ag - ce - cg$

$P_1 = A_1 \times B_1;$ $P_2 = A_2 \times B_2;$ $P_3 = A_3 \times B_3;$ $P_4 = A_4 \times B_4;$
$P_5 = A_5 \times B_5;$ $P_6 = A_6 \times B_6;$ $P_7 = A_7 \times B_7;$ $P_8 = A_8 \times B_8;$
$r = P_1 + P_2;$ $s = P_3 + P_4;$ $t = P_5 + P_6;$ $u = P_7 + P_8;$

Всего 8 матричных умножений и 4 матричных сложения.

$r = P4 - P2 + P5 + P6$

$s = P1 + P2$

$t = P3 + P4$

$u = P1 + P5 - P3 - P7$

В этом алгоритме потребовалось 7 матричных умножений и 18 сложений

Алгоритм Штрассена позволяет уменьшить асимптотическую оценку трудоемкости до $T(n) = n^{\log 7}$.

Алгоритм Копперсмита-Винограда

Трудоемкость $O(n^{2.38})$

Базовая операция – вычисление скалярного произведения вектора-строки (первый матричный сомножитель) и вектора-столбца (второй матричный сомножитель).

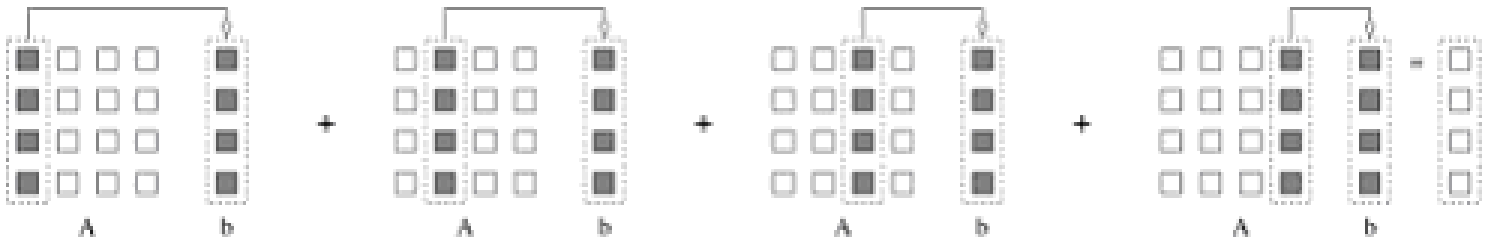
V = (v1, v2, v3, v4) W = (w1, w2, w3, w4)
V • W = v1w1 + v2w2 + v3w3 + v4w4

Это равенство допускает следующее представление:

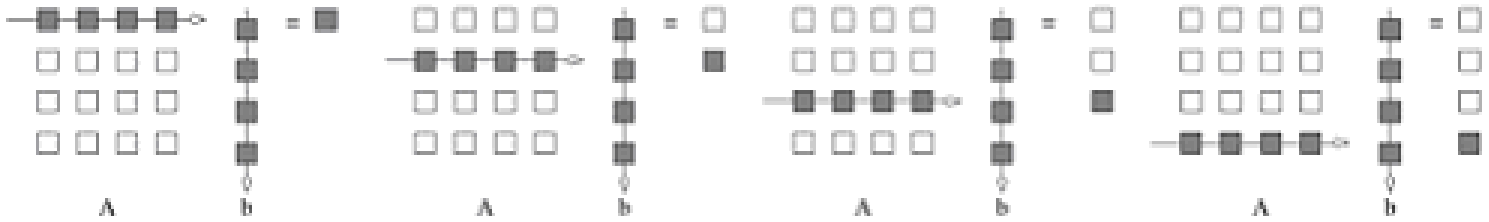
V • W = (v1 + w2)(v2 + w1) + (v3 + w4)(v4 + w3) - v1v2 - v3v4 - w1w2 - w3w4

Компоненты второй части формулы могут быть вычислены заранее, это позволяет уменьшить число операций.

Кодирование



(a)



(b)

Библиотеки

Использование оптимизированных библиотек

Пример

Матричные вычисления:

- BLAS (Basic Linear Algebra Subroutines) Levels 1-3
- LAPACK (Linear Algebra Package), ScaLAPACK
- Visual Numerics Inc. ® IMSL (International Mathematics and Statistics Library)
- Intel ® MKL (Math Kernel Library)
- ……

BLAS

- Уровень 1. Основные операции с векторами.
- Уровень 2. Основные матрично-векторные операции.
- Уровень 3. Основные матрично-матричные операции.

Библиотека оптимизируется с учетом архитектуры

Intel ® MKL

Состав библиотеки:

- BLAS (3 уровня + расширение – уровень 1 для разреженных векторов)
- LAPACK – вычислительная алгебра, в том числе решение спектральных задач
- DFT (дискретное преобразование Фурье) – в том числе многомерное. Многопоточная реализация
- Vector Mathematical Library – математические функции
- Vector Statistical Library – набор векторизованных генераторов случайных чисел

Оптимизирована для архитектуры Intel ®

Intel ® Integrated Performance Primitives

Библиотека готовых компонентов для разработки мультимедийных приложений для вычислительных платформ Intel.

Включает модули для обработки сигналов и выполнения векторных и матричных операций, функции сжатия и распаковки речи и статических/динамических изображений, средства шифрования и обработки аудиоданных и текстовых строк.

Intel IPP обеспечивает прозрачное использование расширенных возможностей процессоров Intel, таких, как технология MMX, наборы команд Streaming SIMD Extensions и Streaming SIMD Extensions 2. Библиотека Intel IPP оптимизирована для работы с самыми разными микропроцессорами компании Intel, включая Intel Pentium 4, Intel Itanium 2, Intel Xeon, а также процессорами для карманных устройств с архитектурой XScale.

Библиотека Intel IPP поддерживает 32- и 64-битные операционные системы Windows и Linux, включая встраиваемые версии, такие как Windows Mobile.

LAPACK

Библиотеки подпрограмм LAPACK и ScaLAPACK поддерживают работу с:

- заполненными прямоугольными или квадратными матрицами;
- ленточными матрицами (узкими);
- заполненными матрицами, хранящимися на внешнем носителе.

Компиляция

Использование возможностей автоматической оптимизации компилятора может дать значительный выигрыш в производительности

Компиляция

- icl [ключи] файлы

- /Fe (Windows), -o (Linux):
  имя исполняемого файла

- /Zi (Windows), -g (Linux):
  отладочная информация

Автоматическая оптимизация

/Od (Windows), -O0 (Linux)

- Отключение оптимизации
- Подразумевается в режиме Debug в MS Visual Studio

/O1 (Windows), -O1 (Linux)

- Глобальная оптимизация
- Не увеличивает размер кода

/O2 (Windows), -O2 (Linux)

- Увеличивает размер кода
- Оптимизация времени выполнения
- Опция по умолчанию
- Подстановки в коде
- Развертывание циклов
- Векторизация

/O3 (Windows), -O3 (Linux)

- /O2 + более агрессивные методы
- Подстановка кода в ветвлениях
- Оптимизация под размер кэша
- Предвыборка и предсказания ветвления
- Возможна большая эффективность в приложениях, включающих обработку больших массивов

Примеры

Matrices.cpp

- icl /FeMatricesOd /Od Matrices.cpp
- icl /FeMatricesO1 /O1 Matrices.cpp
- icl /FeMatricesO2 /O2 Matrices.cpp
- icl /FeMatricesO3 /O3 Matrices.cpp

Сравним результаты

Intel Core 2 Duo T3700 2.00 GHz, 2 Gb RAM
Среднее значение времени выполнения:

- MatricesOd . . . 21.760 s
- MatricesO1 . . . 14.047 s
- MatricesO2 . . . 9.630 s
- MatricesO3 . . . 9.568 s

Оптимизация под архитектуру

/arch:*name* (Windows)

- Оптимизация под архитектуру
- IA32, SSE, SSE2, SSE3
- Работает на любых процессорах, поддерживающих выбранную архитектуру

/Qx (Windows), -x (Linux)

- Оптимизация под архитектуру Intel
- QxHost
- QxAVX
- QxSSE2, QxSSE3, QxSSE3_ATOM
- QxSSE4.1, QxSSE4.2
- QxSSSE3

/Qax (Windows), -ax (Linux)

- Оптимизация под архитектуру Intel
- Для многоядерных систем
- Конкретные наборы инструкций задаются, как для /Qx

Пример

Primes.cpp

- icl /FePOd /Od Primes.cpp
- icl /FePx /QaxSSSE3 Primes.cpp

Сравним результаты

Intel Core 2 Duo T3700 2.00 GHz, 2 Gb RAM

Среднее значение времени выполнения:

- POd . . . . . 29.807 s
- Px . . . . . 4.515 s

Автоматическое распараллеливание

/Qparallel (Windows), -parallel (Linux)

- Автоматическое распараллеливание
- Для многоядерных архитектур
- Работа с циклами

Пример

FermatsCubes.cpp

- icl /FeFCOd /Od FermatsCubes.cpp
- icl /FeFCPar /Qparallel FermatsCubes.cpp

Сравним результаты

Intel Core 2 Duo T3700 2.00 GHz, 2 Gb RAM

Среднее значение времени выполнения:

- FCOd . . . . . 25.948 s
- FCPar . . . . . 3.776 s

Оптимизация с профилированием

/Qprof-gen (Windows),
-prof-gen (Linux)

/Qprof-use (Windows),
-prof-use (Linux)

- Инструментовка
- Сбор информации
- Компиляция с учетом проанализированных данных

Пример

Branches.cpp

- icl /FeBOd /Od Branches.cpp
- icl /FeBPrg /Qprof-gen Branches.cpp
- BPrg.exe
- icl /FeBProf /Qprof-use Branches.cpp

Сравним результаты

Intel Core 2 Duo T3700 2.00 GHz, 2 Gb RAM

Среднее значение времени выполнения:

- BOd . . . . . 13.171 s
- BProf . . . . . 0.0 s

Межпроцедурная оптимизация

/Qip (Windows),
-ip (Linux)

- Анализ вызываемых функций приложения
- Встраивание
- Подстановки

Пример

SquareRoots.cpp

- icl /FeSqOd /Od SquareRoots.cpp
- icl /FeSqip /Qip SquareRoots.cpp

Сравним результаты

Intel Core 2 Duo T3700 2.00 GHz, 2 Gb RAM

Среднее значение времени выполнения:

- SqOd . . . . . 35.692 s
- Sqip . . . . . 9.588 s

Дополнительная информация

/fast Windows),
-fast (Linux)

Межпроцедурная оптимизация

- /O3
- /QxHost
- -static (Linux)
- -no-prec-div (Linux),
  /Qprec-div- (Windows)

Дополнительные опции

- /Qunroll:n (Win), -unroll*n* (Linux)
- /Qvec[-] (Win), -[no-]vec (Linux)
- /Qopt-prefetch[:n][-] (Win),
  -[no]-opt-prefetch[=n] (Linux)
- /Qopenmp (Win), -openmp (Linux)
- /fp:*name* (Win),
  -fp-model *name* (Linux)

Пример с изменением поведения программы

error.f90

Оптимизация готовой программы

Что такое оптимизация на основе анализа выполнения?

Анализ и переработка исходного кода программы на основе информации, полученной при выполнении этой программы на конкретном оборудовании

Когда нужно пользоваться оптимизацией на основе анализа выполнения?

- Для улучшения производительности программы
- Для выявления неэффективного/неиспользующегося кода
- При переработке приложений
- Для контроля за разработкой больших программных продуктов
- Для определения необходимого оборудования

Что предлагает Intel®

- Intel® VTune Performance Analyzer (Windows, Linux):
  профессиональный инструмент динамического анализа и профилирования приложений
- Intel® Parallel Amplifier (Windows):
  часть Intel Parallel Studio, сфокусирован на профилировании параллельных приложений

Intel® VTune Performance Analyzer

- Интерфейсы: командная строка

  +

  Windows - отдельный GUI и интеграция в Visual Studio
  Linux - интеграция в Eclipse
- Инструменты профилирования по времени/ по событиям
- Информация о загрузке системы
- Профилирование потоков (threads)
- Дополнительные способы профилирования

Сбор статистики:

- по времени (TBS);
- по заданному количеству событий (например, по заданному количеству ошибочно предсказанных ветвлений, EBS);
- привязка к исходному коду или дизассемблирование.

Tuning Assistant

- комментарии по проблемам оптимальности кода;
- подсказки по модификации кода.

А также:

- «мастера» для конфигурирования коллекторов;
- упаковка и перенос проектов на другую машину.

Для анализа приложения необходимы:

- Исполняемый (бинарный) файл, подготовленный соответствующим образом
- Исходный файл (не обязательно)
- Символьная информация
- Информация о номерах строк

На что обратить внимание при анализе в первую очередь:

- ❑ Cache misses
- ❑ Branch misprediction
- ❑ FP (вычисления с плавающей точкой)
- ❑ Resource related stalls
- ❑ 64k aliasing (P4)

В Help for VTune performance analyzer имеется информация о событиях

Основы динамического анализа приложений: уровни анализа

- Уровень системы

- Уровень приложения

- Уровень архитектуры процессора

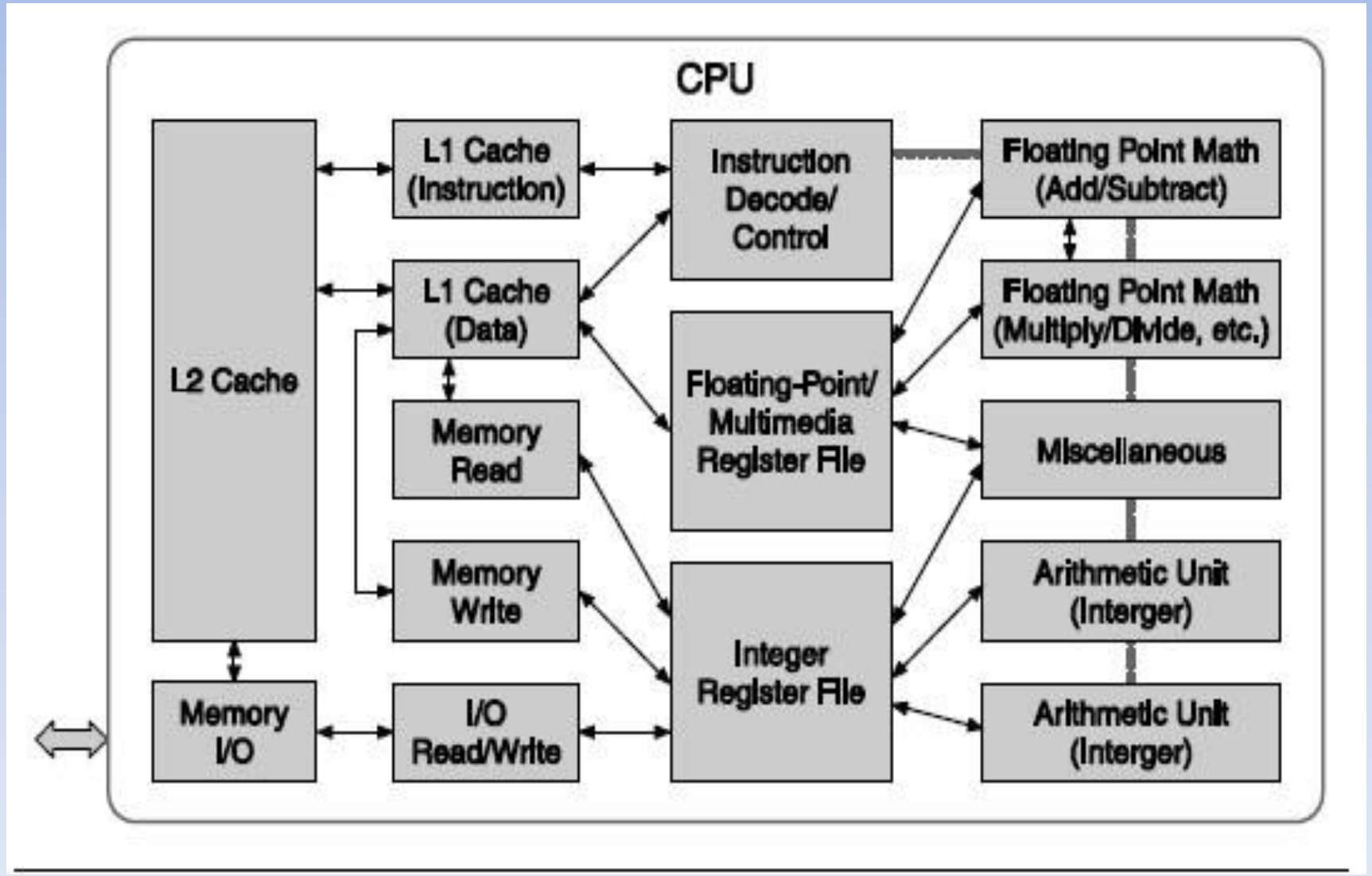
Основы динамического анализа приложений: системный уровень

- Оптимизация на основе анализа использования приложением различных ресурсов системы
- Наиболее эффективна для приложений с интенсивным вводом/выводом данных
- Иногда пропускается в случае процессорно-интенсивных задач
- Инструменты для системного анализа обычно уже уже есть в операционной системе

Основы динамического анализа приложений: уровень приложения

- Цель - улучшение алгоритмов, использования многопоточности, включение оптимизированного кода
- Часто эффективна без дополнительно погружения на уровень архитектуры процессора
- Анализ на уровне приложения наиболее связан с анализом исходного кода программы

Основы динамического анализа приложений: уровень архитектуры процессора



Основы динамического анализа приложений: уровень архитектуры процессора

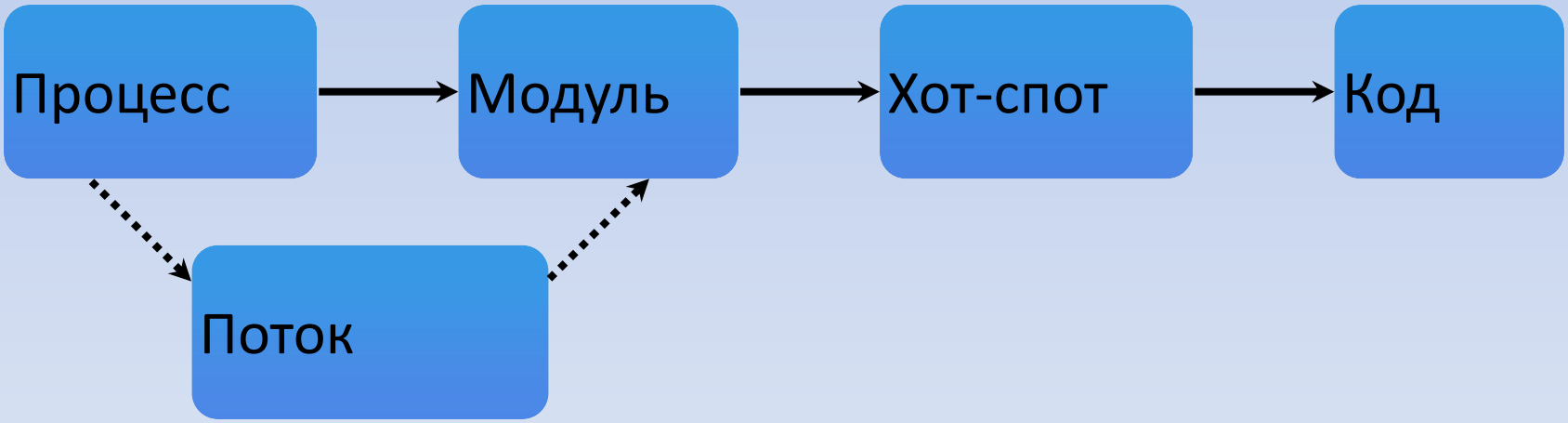
- Оптимизация на основе анализа выполнения программы на процессоре конкретной архитектуры
- Наиболее важна для процессорно-интенсивных задач
- VTune позволяет оптимизировать программу для любой современной архитектуры Intel

Использование VTune: Проекты и активности

- **Проект**:
  файл .vpj, содержит различные активности для профилируемых приложений
- **Активность**:
  собирает различные типы данных при выполнении приложения

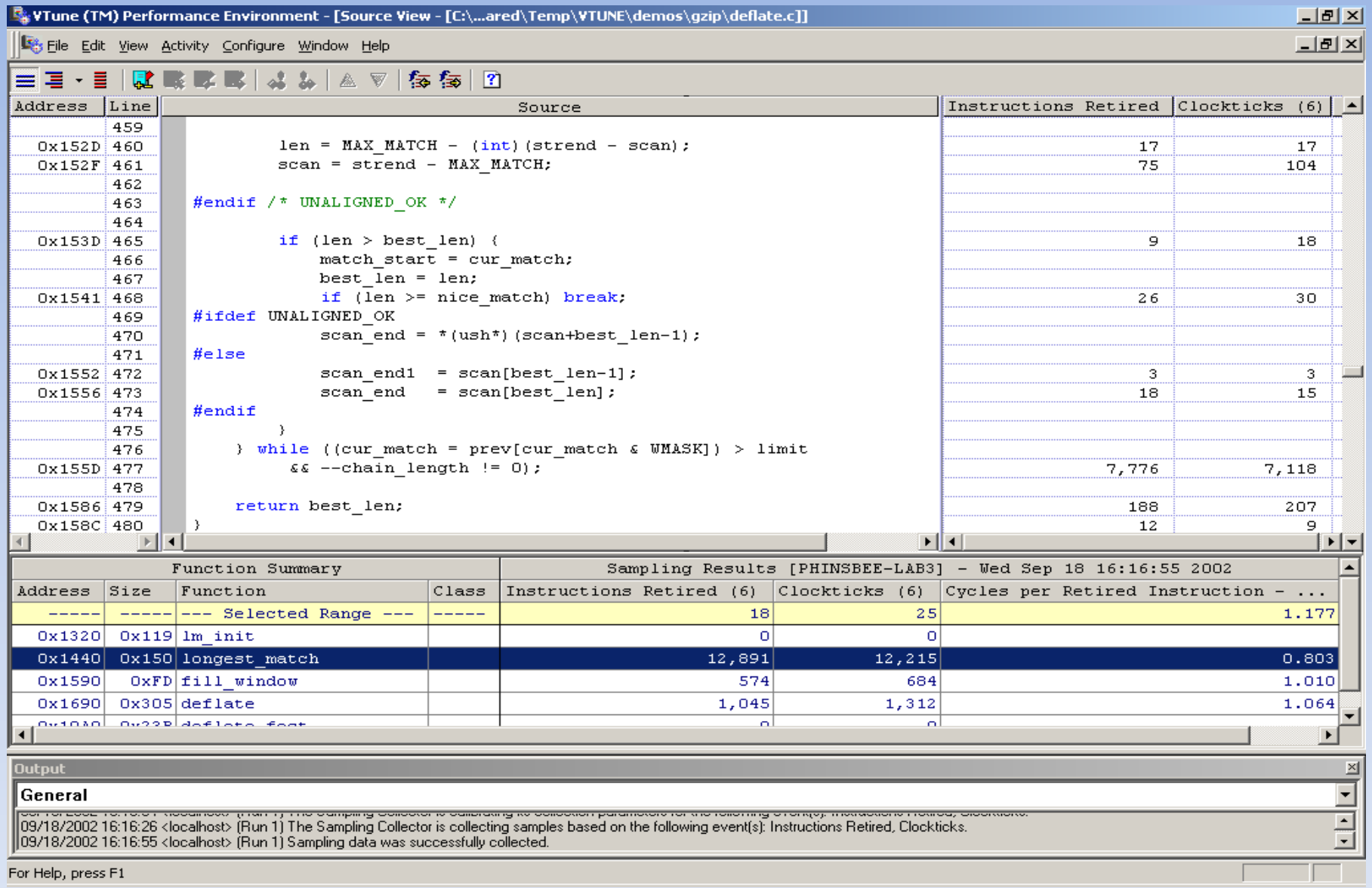
Использование VTune: “Drilling down”

Постепенное углубление в структуру программы



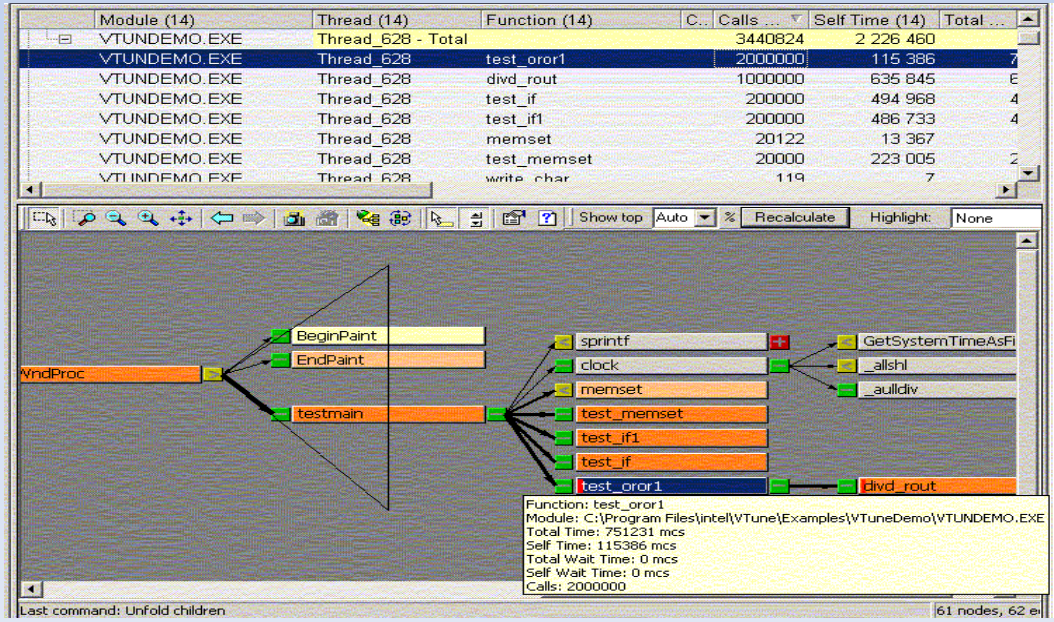
Использование VTune: Sampling

- Основной (первый) тип анализа
- Поиск хот-спотов
  - без изменения исходного кода
  - без инструментовки
- Минимальное вмешательство в ход выполнения программы
- Сэмплирование всей системы


VTune (TM) Performance Environment - [Source View - [C:\...ared\Temp\VTUNE\demos\gzip\deflate.c]]
File Edit View Activity Configure Window Help
Address Line Source Instructions Retired Clockticks (6)
0x152D 460 len = MAX_MATCH - (int)(strend - scan); 17 17
0x152F 461 scan = strend - MAX_MATCH; 75 104
0x153D 465 if (len > best_len) { 9 18
0x1541 468 if (len >= nice_match) break; 26 30
0x1552 472 scan_end1 = scan[best_len-1]; 3 3
0x1556 473 scan_end = scan[best_len]; 18 15
0x155D 477 } while ((cur_match = prev[cur_match & WMASK]) > limit 7,776 7,118
0x1586 479 return best_len; 188 207
0x158C 480 } 12 9
Function Summary
Sampling Results [PHINSBEE-LAB3] - Wed Sep 18 16:16:55 2002
Address Size Function Class Instructions Retired (6) Clockticks (6) Cycles per Retired Instruction - ...
----- ----- --- Selected Range --- ----- 18 25 1.177
0x1320 0x119 lm_init 0 0
0x1440 0x150 longest_match 12,891 12,215 0.803
0x1590 0xFD fill_window 574 684 1.010
0x1690 0x305 deflate 1,045 1,312 1.064
Output
General
09/18/2002 16:16:26 <localhost> (Run 1) The Sampling Collector is collecting samples based on the following event(s): Instructions Retired, Clockticks.
09/18/2002 16:16:55 <localhost> (Run 1) Sampling data was successfully collected.
For Help, press F1

Использование VTune: Call Graph

- Построение графика вызова функций
- Поиск критического пути программы
  - автоматическое внедрение в год небольших участков (инструментовка)
  - некоторое замедление исполнения программы
- Не работает для защищенных частей системы



Call Graph
Особенности инструментовки

- Добавление опции /fixed:NO в опциях линкера
- Можно управлять уровнем инструментовки для каждого модуля и функции
- Можно управлять размером буфера

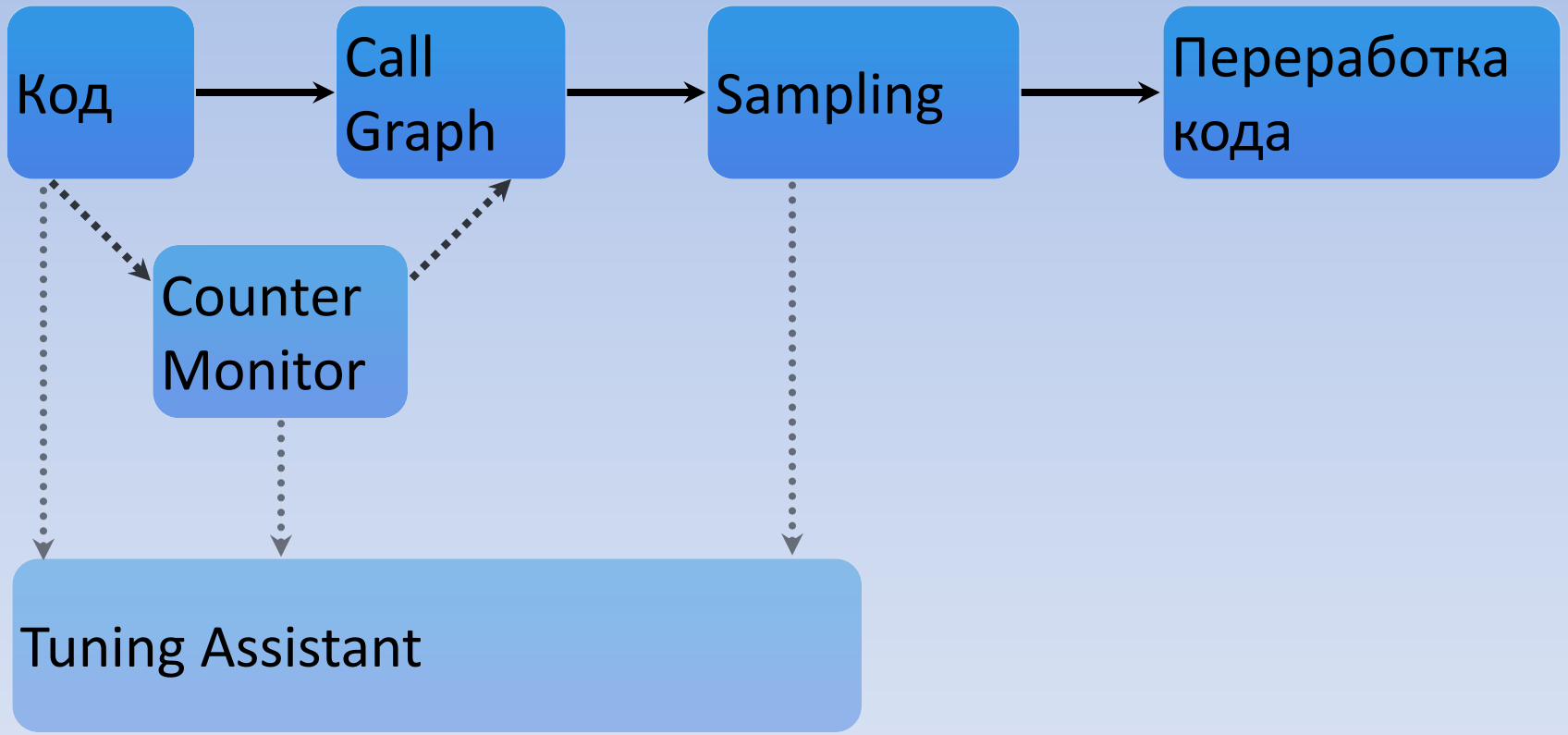
Использование VTune: Counter Monitor

- Анализ использования приложением ресурсов системы
  - отображение в реальном времени информации с различных счетчиков (OS, приложений, процессора)
  - не связан напрямую с исходным кодом программы
- Особенно важен в случае графических/сетевых приложении

Использование VTune: Tuning Assistant

- Автоматический анализ и формирование отчета, основанного на:
  - статическом анализе исходного кода статическом анализе дисассемблированного кода
  - данных, полученных из сэмплинга
  - данных счетчиков использования системных ресурсов
- Эффективен для небольших участков кода в режиме сэмплинга

Схема оптимизации с помощью VTune



Parallel Studio

Advisor, Composer, Inspector, Amplifier

Microsoft Windows, интеграция в Visual Studio

Advisor

Выявление «кандидатов» на распараллеливание

Composer

- Intel® C++ Compiler, Intel® Threading Building
- Blocks, Intel® Integrated Performance Primitives,
- and Intel® Parallel Debugger Extension.

Inspector

Выявление ошибок многопоточности

Amplifier

Анализ производительности, оптимизация параллельных приложений

Параллельное программирование

Многопоточная программа

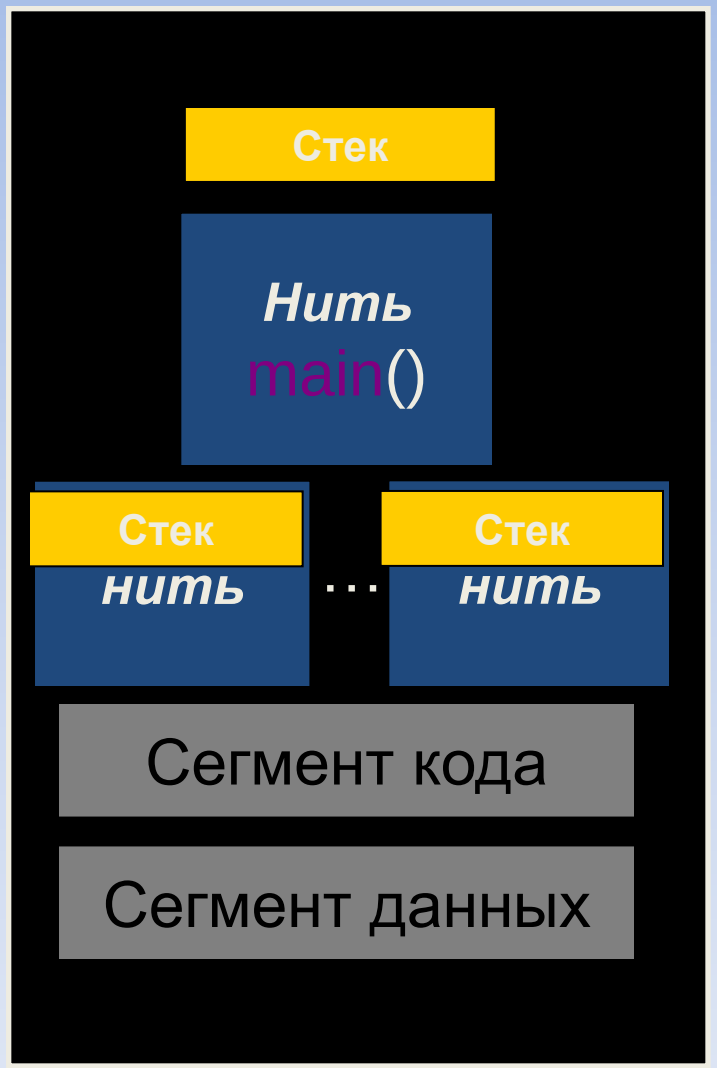
Поток (нить) представляет собой последовательный поток управления (последовательность команд) в рамках одной программы.

При создании процесса порождается главный поток, выполняющий инициализацию процесса. Он же начинает выполнение команд

Поток и процесс соотносятся следующим образом

- процесс имеет главный поток, инициализирующий выполнение команд процесса;
- любой поток может порождать в рамках одного процесса другие потоки;
- каждый поток имеет собственный стек;
- потоки, соответствующие одному процессу, имеют общие сегменты кода и данных.

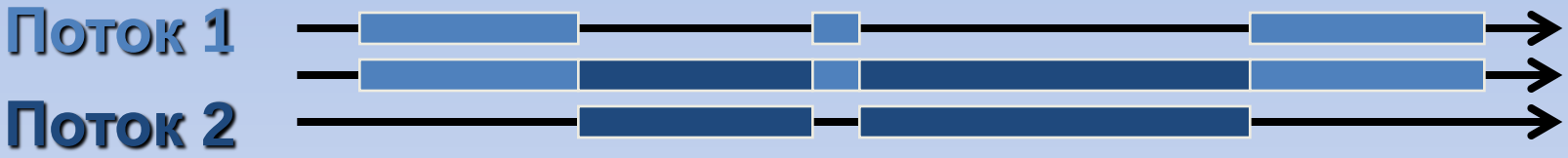
Многопоточная программа



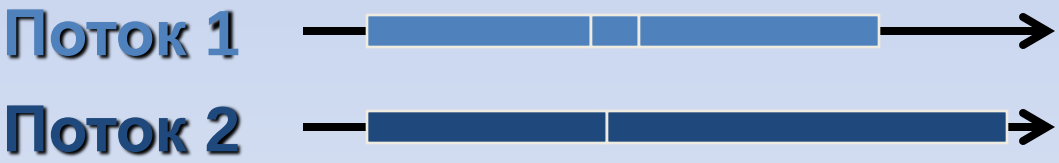
Многопоточная программа

Конкуренция за ресурсы и параллельное исполнение

Конкуренция за ресурсы (видимый параллелизм)



Реальный параллелизм



Для реализации реального параллелизма требуется соответствующая архитектура – многоядерная или многопроцессорная с общей памятью

Многопоточная программа

При разработке многопоточных приложений возникают следующие проблемы:

- ❑ гонки за данными;
- ❑ блокировки;
- ❑ несбалансированность загрузки.

Гонки за данными (data races)

Гонки за данными являются следствием зависимостей, когда несколько потоков модифицируют содержимое одной и той же области памяти. Наличие гонок за данными не всегда является очевидным. Они могут приводить к конфликтам двух типов:

1) конфликт «чтение-запись»;
2) конфликт «запись-запись».

Многопоточная программа

Два способа борьбы с гонками за данными:

- использование преимущественно локальных по отношению к потоку, а не разделяемых переменных;
- управление доступом к разделяемым переменным с помощью различных средств синхронизации (они могут быть реализованы с помощью семафоров, событий, критических секций, взаимных блокировок - мьютексов).

Гонки за данными могут быть скрыты синтаксисом языка программирования

Примеры конструкций языка программирования, в которых могут возникать гонки за данными

Поток 1	Поток 2	Причина возникновения гонок за данными
`X += 1`	`X += 2`	Компилятор заменяет операцию += раздельными операциями чтения и записи X
`A[i] += 1`	`A[j] += 2`	Возможно совпадение значений индексов i и j
`*p += 1`	`*q += 2`	Указатели p и q могут ссылаться на один адрес
`Func(1)`	`Func(2)`	Func может суммировать значение аргумента и значение внутренней разделяемой переменной
`Add [abc], 1`	`add [abc], 2`	На уровне команд модификация [abc] заменяется раздельными операциями чтения и записи

Многопоточная программа

Блокировки

Блокировка (тупик) возникает, если поток ожидает выполнение условия, которое не может быть выполнено. Обычно возникновение тупиковой ситуации является следствием конкуренции потоков за ресурс, который удерживается одним из них.

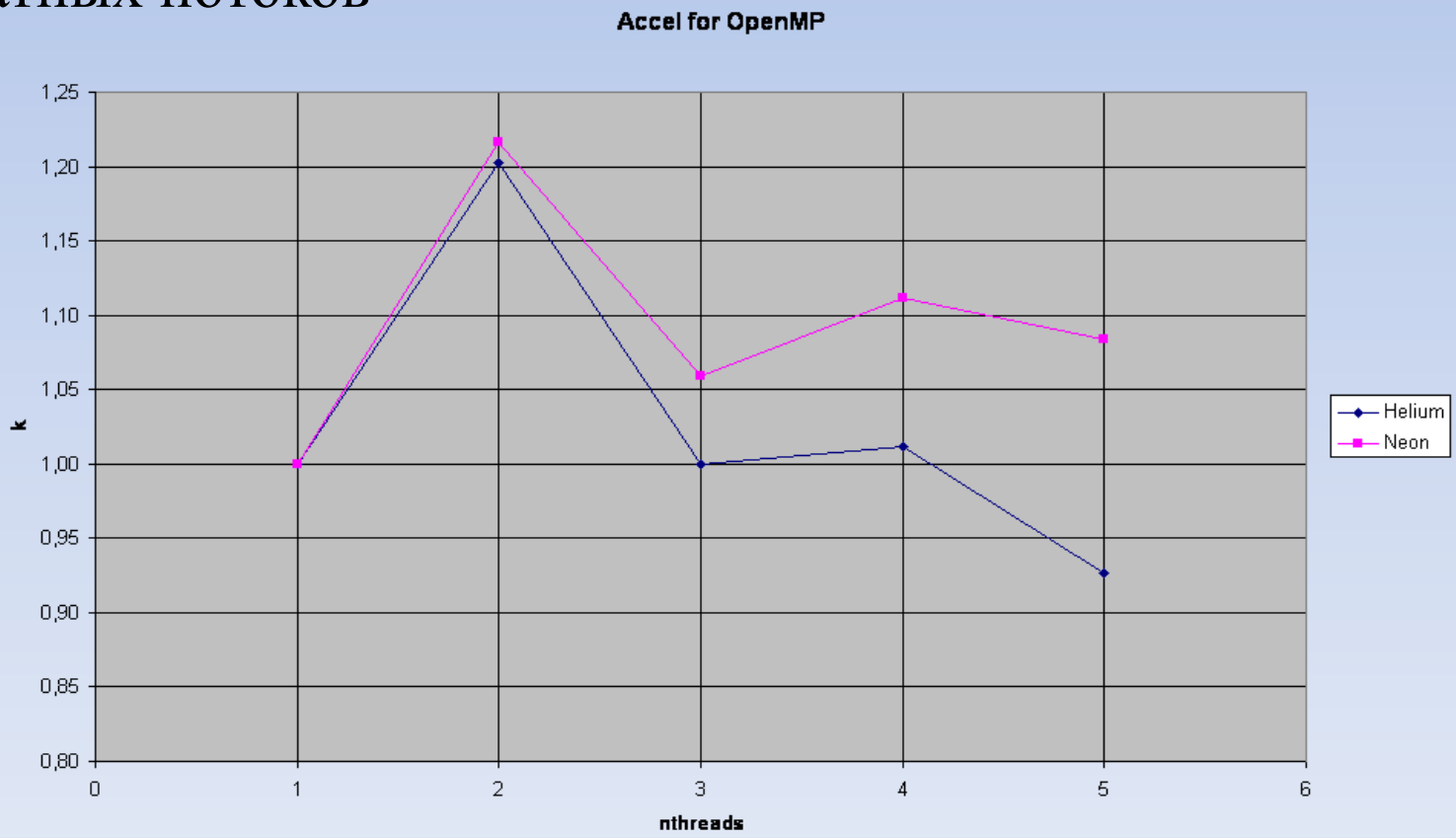
Условия возникновения тупика

- доступ к ресурсу эксклюзивен (возможен только одним потоком);
- поток может удерживать ресурс, запрашивая другой;
- ни один из конкурирующих потоков не может освободить запрашиваемый ресурс.

Многопоточная программа

Масштабируемость

Число программных потоков должно совпадать с числом аппаратных потоков



Многопоточная программа

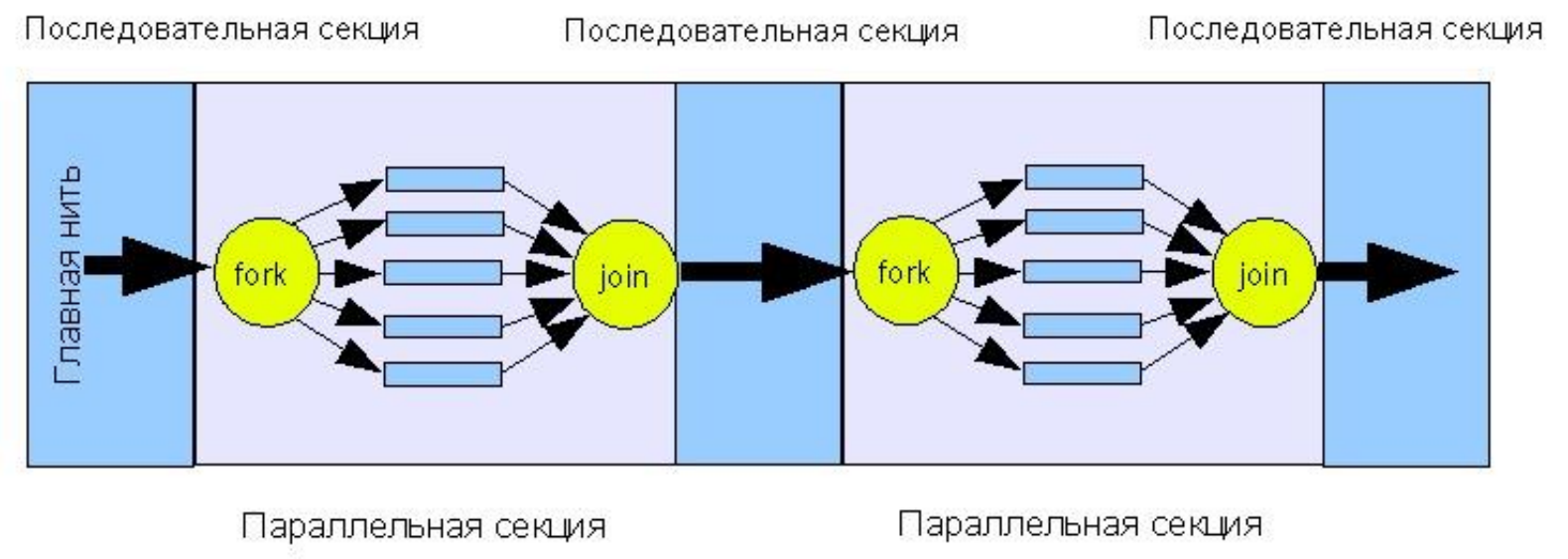
Реализации

POSIX Threads
Windows API низкоуровневые инструменты

OpenMP высокоуровневые инструменты

OpenMP

OpenMP – модель программы



OpenMP – модель программы

- программа состоит из последовательных и параллельных секций;
- в начальный момент времени порождается основная нить, выполняющая последовательные секции программы;
- при входе в параллельную секцию программы выполняется операция fork, порождающая набор нитей;
- каждая нить имеет свой уникальный числовой идентификатор (0 для мастер-нити). Все параллельные нити исполняют один код;
- при выходе из параллельной секции выполняется операция join, завершающая выполнение всех нитей кроме главной.

За что мы любим OpenMP

- масштабируемость;
- возможность пошагового распараллеливания;
- переносимость;
- высокоуровневое программирование;
- поддержка языков Fortran и C/C++;
- поддержка модели параллелизма данных

OpenMP – компоненты

1) *Директивы компилятора* - используются для создания потоков, распределения работы между потоками и их синхронизации. Директивы включаются в исходный текст программы.

2) *Подпрограммы библиотеки времени выполнения* - используются для установки и определения атрибутов потоков. Вызовы этих подпрограмм включаются в исходный текст программы.

3) *Переменные окружения* - используются для управления поведением параллельной программы.

OpenMP – привязки

Привязка к языку C

В программах на языке C прагмы, имена функций и переменных окружения OpenMP начинаются с omp, omp_ или OMP_. Формат директивы:

#pragma omp директива [оператор_1[, оператор_2, :]]

В OpenMP-программе используется заголовочный файл omp.h.

OpenMP – привязки

Привязка к языку Fortran

В программах на языке Fortran директивы компилятора, имена подпрограмм и переменных окружения начинаются с OMP или OMP_. Формат директивы компилятора:

{!|C|*}$OMP директива [оператор_1[, оператор_2, :]]

Директива начинается в первой (фиксированный формат записи текста языка Fortran 77) или произвольной (свободный формат) позиции строки. Допускается продолжение директивы в следующей строке, в этом случае действует стандартное в данной версии языка правило для обозначения строки продолжения (непробельный символ в шестой позиции для
фиксированного формата записи и амперсанд для свободного формата).

OpenMP – примеры

```
program omp_example
integer i, k, N
real*4 sum, h, x
print *, "Please, type in N:"
read *, N
h = 1.0 / N
sum = 0.0
!$OMP PARALLEL DO SCHEDULE(STATIC) REDUCTION(+:sum)
do i = 1, N
x = i * h
sum = sum + 1.e0 * h / (1.e0 + x**2)
enddo
print *, 4.0 * sum
end
```

OpenMP – примеры

```
#include "omp.h"
#include <stdio.h>
double f(double x) {
return 4.0 / (1 + x * x); }
main () {
const long N = 100000;
long i;
double h, sum, x;
sum = 0;
h = 1.0 / N;
#pragma omp parallel shared(h) {
#pragma omp for private(x) reduction(+:sum)
for (i = 0; i < N; i++) {
x = h * (i + 0.5);
sum = sum + f(x); } }
printf("PI = %f\n", sum / N); }
```

OpenMP

Накладные расходы

parallel	1.5 мкс (Intel ® Xeon 3Ггц) l
barrier	1.0
schedule(static)	1.l0
schedule(guided)	6.0
schedule(dynamic)	50.0
ordered	0.5
single	1.0
reduction	2.5

OpenMP

Накладные расходы

schedule(dynamic)	50.0 мкс
schedule(dynamic, 16)	5.0 мкс

Спецификация MPI. История создания. Версии

Большой интерес к системам передачи сообщений возник в 1980-е годы. Его следствием стало появление достаточно большого количества реализаций, создававшихся разными коллективами разработчиков и предназначенных для разных вычислительных систем.

Примеры:
p4, PICL, PVM, LAM, TCGMSG, Express и другие

Возникла необходимость координировать и стандартизировать этот процесс, поэтому в рамках конференции Supercomputing’92 состоялось совещание, на котором было принято решение о разработке стандарта программного интерфейса обмена сообщениями.

Процесс стандартизации был поддержан индустрией:
Convex, Cray, IBM, Intel, nCUBE, NEC, Thinking Machines и др.

Индустрия была заинтересована в средстве разработки эффективных программ для высокопроизводительных вычислительных систем. В жертву производительности были принесены устойчивость и синхронизация процессов.

Сайт, на котором можно найти официальные документы MPI:
http://www.mpi-forum.org

Версия MPI-1 вышла в 1994 году

Версия MPI-2 вышла в 1998 году, первая реализация появилась в 2002 году.

Версия MPI-2.1 вышла в начале сентября 2008 года

Спецификация MPI-1

Содержит описание стандарта программного интерфейса обмена сообщениями. Спецификация учитывает опыт предшествующих разработок и ориентирована на большую часть аппаратных платформ. Несмотря на то, что MPI рассчитано на использование с языками C/C++ и Fortran, семантика в значительной степени не зависит от языка.

В MPI-1 описываются интерфейсы процедур двухточечного и коллективного обмена, сбора информации, организации обменов в группах процессов, синхронизации процессов, виртуальные топологии, привязки к языкам программирования и т. д.

Спецификация MPI-2

Является дальнейшим развитием MPI. Новое в MPI-2:

- возможность создания новых процессов во время выполнения MPI-программы (в MPI-1 количество процессов фиксировано, крах одного приводит к краху всей программы);
- новые разновидности двухточечных обменов (односторонние обмены);
- новые возможности коллективных обменов;
- поддержка внешних интерфейсов;
- операции параллельного ввода-вывода с файлами.

Реализации MPI

Реализации MPI

MPI CHameleon (MPICH) является свободно распространяемой “opensource” реализацией MPI. Этот пакет доступен в исходных кодах, поэтому допускает гибкую настройку.
Поддерживается работа в различных версиях ОС UNIX, Mac OS и в последних версиях Microsoft Windows. В последнем случае имеется возможность установки из бинарных файлов.

MPICH соответствует спецификации MPI-2.
Поддерживаются различные коммуникационные среды (в т.ч. 10 Gigabit Ethernet, InfiniBand, Myrinet, Quadrics).
Пока не поддерживаются системы, гетерогенные по форматам хранения данных. Имеется версия с поддержкой пакета Globus.

Официальный сайт MPICH
http://www-unix.mcs.anl.gov

Реализации MPI

LAM (Local Area Multicomputer) MPI – “opensource” реализация MPI, соответствующая спецификации MPI-1 и, в значительной мере, спецификации MPI-2.
LAM (http://www.lam-mpi.org/) поддерживает гетерогенные конфигурации, поддерживает пакет Globus и удовлетворяет IMPI (Interoperable MPI). Поддерживаются различные коммуникационные системы (в т.ч. Myrinet).
IMPI – попытка создания стандарта, обеспечивающего интероперабельность различных реализаций MPI (http://impi.nist.gov/). В настоящее время IMPI поддерживается такими реализациями, как:

- **LAM/MPI**
- **MPI/Pro**
- **Hewlett-Packard MPI (от версии 1.7)**
- **GridMPI**

Имеется инструмент проверки на соответствие IMPI.
Разработчики утверждают, что LAM может работать на метакластерных системах.

Реализации MPI

Intel ® MPI входит в состав Intel® Cluster Toolkit.
Это коммерческая реализация MPI, оптимизированная для архитектуры Intel.

Сайт в Интернете:
http://www.intel.com

Реализации MPI

Microsoft MPI входит в состав Compute Cluster Pack SDK.

Ориентирован на работу в среде ОС Microsoft Windows и доступен, в том числе, по лицензии MSDN Academic Alliance.
Входит в состав Microsoft HPC Server 2008.
Основан на MPICH2, но включает дополнительные средства управления заданиями.

Поддерживается спецификация MPI-2.

Реализации MPI

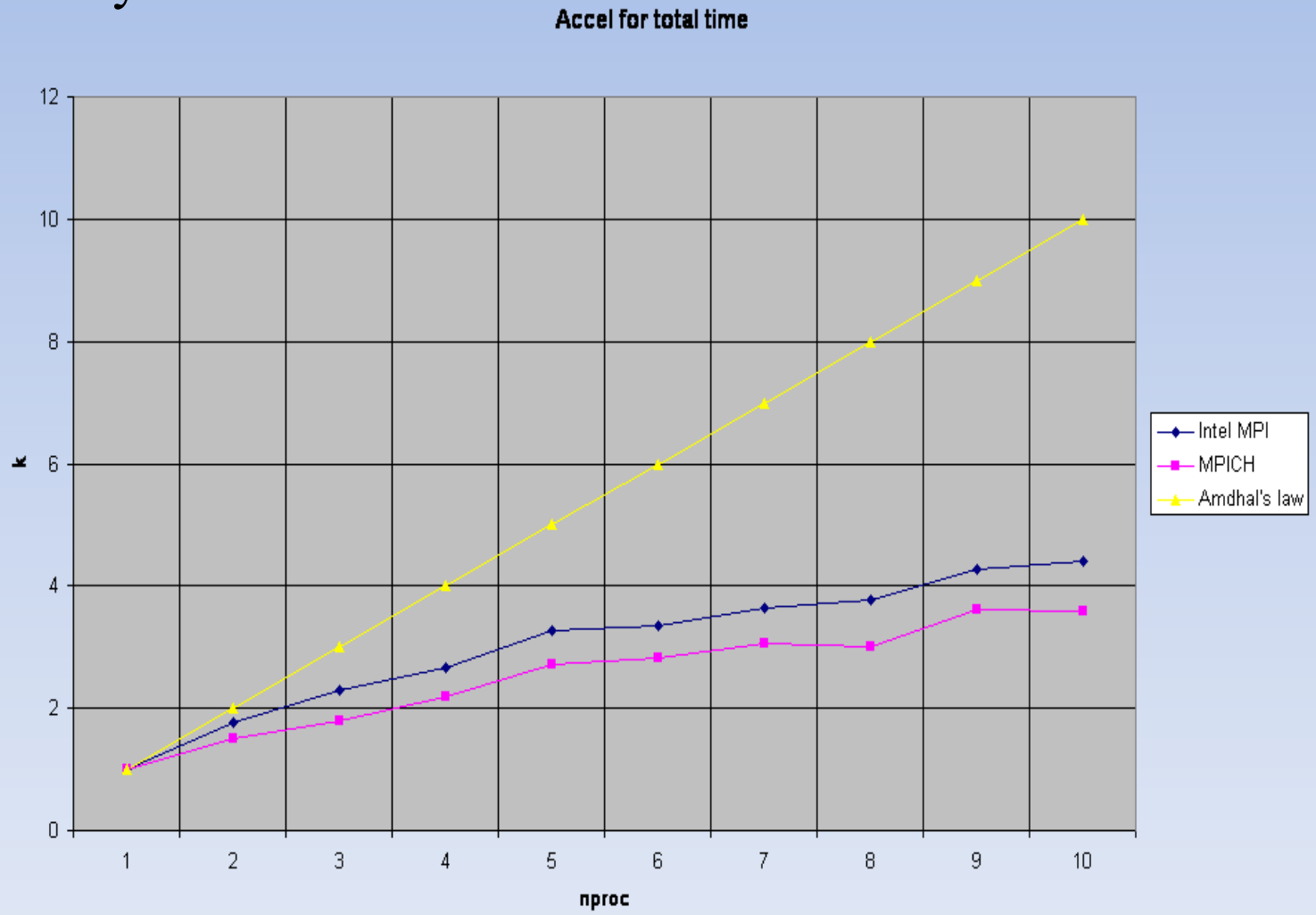
OpenMPI – “opensource” реализация MPI-2, разрабатываемая консорциумом представителей академических, научных и индустриальных кругов.

Полное соответствие спецификации MPI-2.
Поддержка различных ОС.
Поддержка различных коммуникационных сред.

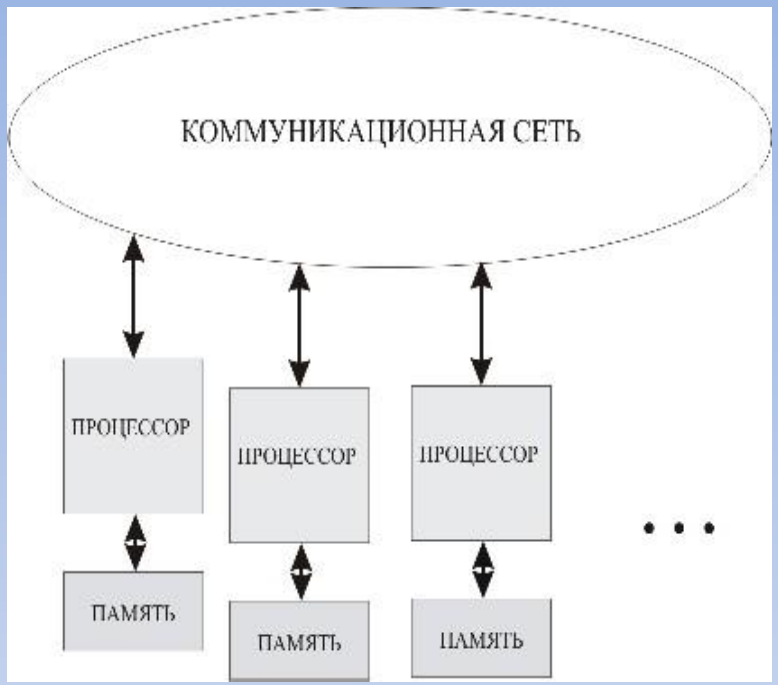
Сайт в Интернете:
http://www.open-mpi.org

Масштабируемость MPI-программ. Сравнение двух реализаций

Зависимость ускорения параллельной версии кода ACE от количества узлов:

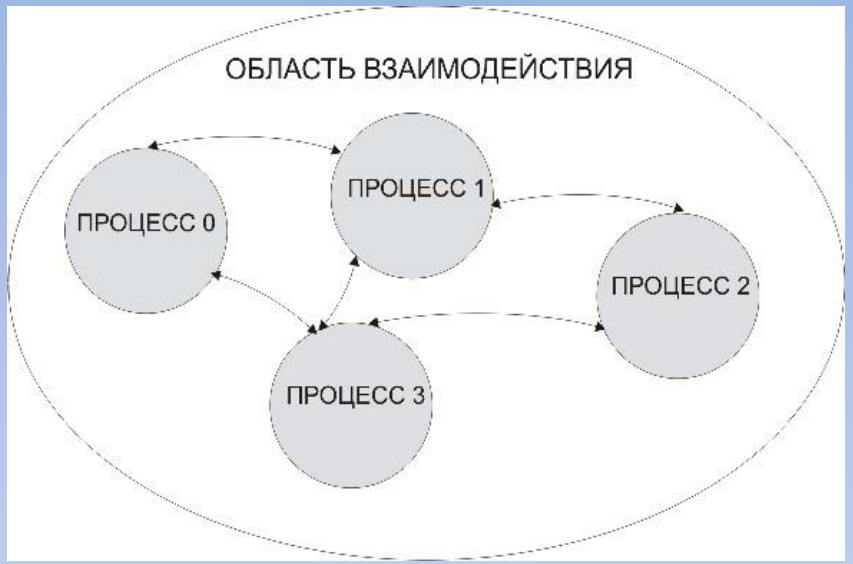


Основные понятия, терминология



Двухточечные обмены используются для организации локальных и неструктурированных коммуникаций. При выполнении глобальных операций используются коллективные обмены.

Асинхронные коммуникации реализуются с помощью запросов о получении сообщений.



Область взаимодействия (область связи) определяет группу процессов. Все процессы, принадлежащие одной области взаимодействия, могут обмениваться сообщениями. Описывает это множество процессов специальная информационная структура, которая называется *коммуникатором*. Этот механизм скрывает от программиста внутренние коммуникационные структуры.

Коммуникаторы, создаваемые по умолчанию:

- **MPI_COMM_WORLD** - содержит все процессы данного приложения;
- **MPI_COMM_SELF** - коммуникатор, содержащий только вызывающий процесс;
- **MPI_COMM_NULL** - пустой коммуникатор.

Сообщение содержит пересылаемые данные и служебную информацию:

- идентификатор процесса-отправителя сообщения (*ранг* процесса);
- адрес, по которому размещаются пересылаемые данные процесса-отправителя;
- тип пересылаемых данных;
- количество данных (размер буфера сообщения для того, чтобы принять сообщение, процесс должен отвести для него достаточный объем оперативной памяти);
- идентификатор процесса, который должен получить сообщение;
- адрес, по которому должны быть размещены данные процессом-получателем;
- идентификатор коммуникатора, описывающего область взаимодействия, внутри которой происходит обмен.

Ранг источника дает возможность различать сообщения, приходящие от разных процессов

Тег - это задаваемое пользователем целое число от 0 до 32767, идентификатор сообщения.

Теги могут использоваться для соблюдения определенного порядка приема сообщений.

Данные, содержащиеся в сообщении, в общем случае организованы в массив элементов, каждый из которых имеет один и тот же тип.

Кроме пересылки данных система передачи сообщений поддерживает пересылку информации о состоянии процессов коммуникации. Это может быть, например, уведомление о том, что прием данных, отправленных другим процессом, завершен.

Перед использованием процедур передачи сообщений программа должна "подключиться" к системе обмена сообщениями.

Подключение выполняется с помощью соответствующего вызова процедуры из библиотеки. В одних реализациях модели допускается только одно подключение, а в других несколько подключений к системе.

При передаче сообщение передается в буфер отправки, а затем процессу-получателю сообщения.

Прием сообщения начинается с подготовки буфера достаточного размера. В этот буфер записываются принимаемые данные.

Данные могут быть закодированы, в этом случае при их считывании выполняется декодирование. Кодирование данных (преобразование в некоторый универсальный формат) требуется для того, чтобы обеспечить обмен сообщениями между системами с разным форматом представления данных.

Операция отправки или приема сообщения считается завершенной, если программа может вновь использовать такие ресурсы, как буферы сообщений.

Компиляция и запуск программ в MPI-2

MPI-2

Демон mpd

В реализациях MPI-2 ключевую роль играет демон mpd. Он управляет выполнением процессов MPI-программы на данном вычислительном узле. Демоны запускаются от имени конкретных пользователей ОС и не оказывают влияния друг на друга.

Запуску параллельной программы предшествует запуск демона mpd на всех узлах вычислительной системы. Демоны взаимодействуют друг с другом.

В программных реализациях MPI имеются средства управления работой демонов mpd.

MPI-2

Конфигурационный файл .mpd.conf

Пример файла **.mpd.conf**

MPD_SECRETWORD=kalosha

MPD_PORT_RANGE=1003:10003

Файл mpd.hosts

Пример файла **.mpd.conf**

pd00 ifhn=195.168.0.69

pd01 ifhn=192.168.0.74

pd02 ifhn=192.168.0.75

pd03 ifhn=192.168.0.76

pd04 ifhn=192.168.0.77

pd05

pd06

pd07

MPI-2

Запуск демонов

mpdboot –n *число_демонов* [ключи**]**

Запуск параллельной программы

mpiexec [ключи**] –n *число имя_исполняемого файла***

Завершение работы всех демонов

mpdallexit

Привязки к языкам программирования

Привязка к языку C

При использовании MPI в программах на языке C в именах функций используется префикс MPI_, а первая буква имени набирается в верхнем регистре.

Имена подпрограмм имеют вид Класс_действие_подмножество или Класс_действие. Аналогичное правило действует и для подпрограмм MPI для языка Fortran. В C++ подпрограмма является методом для определенного класса, имя имеет в этом случае вид MPI::Класс::действие_подмножество.

Для некоторых действий введены стандартные наименования:

- **Create** — создание нового объекта;
- **Get** — получение информации об объекте;
- **Set** — установка параметров объекта;
- **Delete** — удаление информации;
- **Is** — запрос о том, имеет ли объект указанное свойство.

Привязка к языку C

В MPI принята своя система обозначения типов данных, которая соответствует типам данных в языках C и Fortran (соответствие неполное).

Тип данных MPI	Тип данных C
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	Нет соответствия
MPI_PACKED	Нет соответствия

Привязка к языку Fortran

Все имена подпрограмм и констант MPI начинаются с MPI_.

Коды завершения передаются через дополнительный параметр целого типа (находится на последнем месте в списке параметров подпрограммы).

Код успешного завершения — MPI_SUCCESS.

Константы и другие объекты MPI описываются в файле **mpi.h**, который обязательно включается в MPI-программу с помощью оператора include. Этот оператор находится в начале программы.

Переменная status является массивом стандартного целого типа. Его размер и индексы задаются именованными константами:

```
integer status(MPI_STATUS_SIZE)
…
if(status(MPI_TAG).EQ.tag1) then
…
```

Привязка к языку Fortran

Тип данных MPI	Тип данных FORTRAN
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_DOUBLE_COMPLEX	DOUBLE COMPLEX
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER
MPI_BYTE	Нет соответствия
MPI_PACKED	Нет соответствия
Типы, которые имеются не во всех реализациях MPI	
MPI_INTEGER1	INTEGER*1
MPI_INTEGER2	INTEGER*2
MPI_INTEGER4	INTEGER*4
MPI_REAL4	REAL*4
MPI_REAL8	REAL*8

Структура MPI-программы

Структура MPI-программы

При написании программы MPI следует соблюдать определенные правила, иначе она окажется неработоспособной.

В начале программы, сразу после ее заголовка, необходимо подключить соответствующий заголовочный файл. В программе на языке C это **mpi.h**:

```
#include "mpi.h"
```

а в программе на языке Fortran — **mpif.h**:

```
include "mpif.h"
```

В этих файлах содержатся описания констант и переменных библиотеки MPI.

Структура MPI-программы

Первым вызовом библиотечной процедуры MPI в программе должен быть вызов подпрограммы инициализации MPI_Init, перед ним может располагаться только вызов MPI_Initialized, с помощью которого определяют, инициализирована ли система MPI. Вызов процедуры инициализации выполняется только один раз. В языке Fortran у процедуры инициализации единственный аргумент — код ошибки:

```
integer IERR
…
call MPI_Init(ierr)
```

Структура MPI-программы

В C параметры функции инициализации получают адреса аргументов главной программы, задаваемых при ее запуске:

```
MPI_Init(&argc, &argv);
```

Загрузчик **mpiexec** в конец командной строки запуска MPI-программы добавляет служебные параметры, необходимые MPI_Init. В программах на языке Fortran аргументы командной строки не используются.

Процедура инициализации создает коммуникатор со стандартным именем **MPI_COMM_WORLD**. Это имя указывается во всех последующих вызовах процедур MPI.

Структура MPI-программы

После выполнения всех обменов сообщениями в программе должен располагаться вызов процедуры

MPI_Finalize(ierr)

В результате этого вызова удаляются структуры данных MPI и выполняются другие необходимые действия.

Программист должен позаботиться о том, чтобы к моменту вызова процедуры MPI_Finalize были завершены все пересылки данных.

После выполнения данного вызова другие вызовы процедур MPI, включая MPI_Init, недопустимы.

Исключение составляет подпрограмма MPI_Initialized, которая возвращает значение "истина", если процесс вызывал MPI_Init. Данный вызов может находиться в любом месте программы.

Структура MPI-программы

Организация программы по схеме master-slave

```
program parallel
…
if (процесс = мастер) then
 master
else
 slave
endif
…
end
```

Простейшая MPI-программа

simple_MPI.cpp

```
#include <mpi.h>
#include <stdio.h>
int main(int argc, char *argv[]) {
 int ProcNum, ProcRank, tmp;
 MPI_Status status;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &ProcNum);
    MPI_Comm_rank(MPI_COMM_WORLD, &ProcRank);
    printf("Hello world from process %i \n", ProcRank);
    MPI_Finalize();
 return 0;
}
```

simple_MPI.f90

```
program main_mpi
 include 'mpif.h'
 integer myid, numprocs, ierr
 call MPI_Init(ierr)
 call MPI_Comm_rank(MPI_COMM_WORLD, myid, ierr)
 call MPI_Comm_size(MPI_COMM_WORLD, numprocs, ierr)
 print *, "process ", myid, " of ", numprocs
 call MPI_Finalize(ierr)
end
```

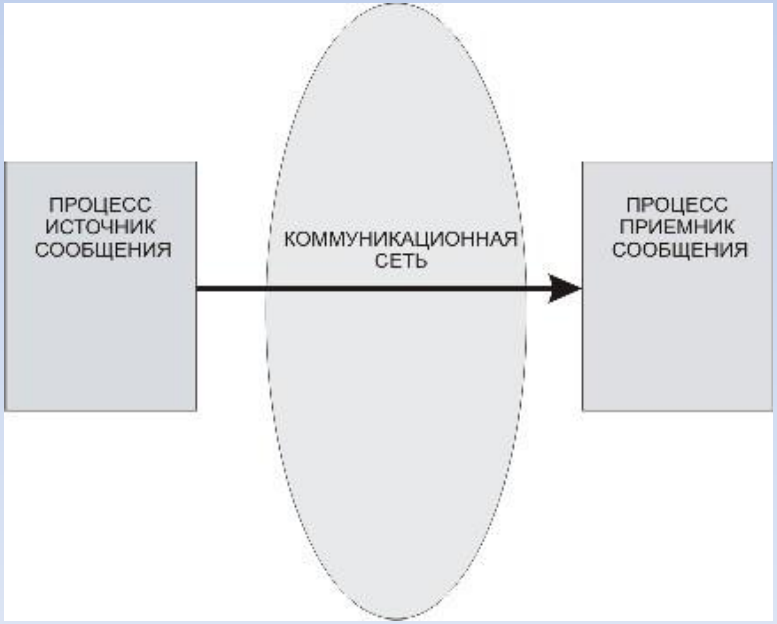
```
[nemnugin@pd00 ~]$ mpdboot -n 3
[nemnugin@pd00 ~]$ mpicxx simple_MPI.cpp
[nemnugin@pd00 ~]$ mpiexec -n 3 ./a.out
Hello world from process 0
Hello world from process 1
Hello world from process 2
[nemnugin@pd00 ~]$
```

Двухточечные обмены

Двухточечные обмены

Двухточечный (point-to-point, p2p) обмен

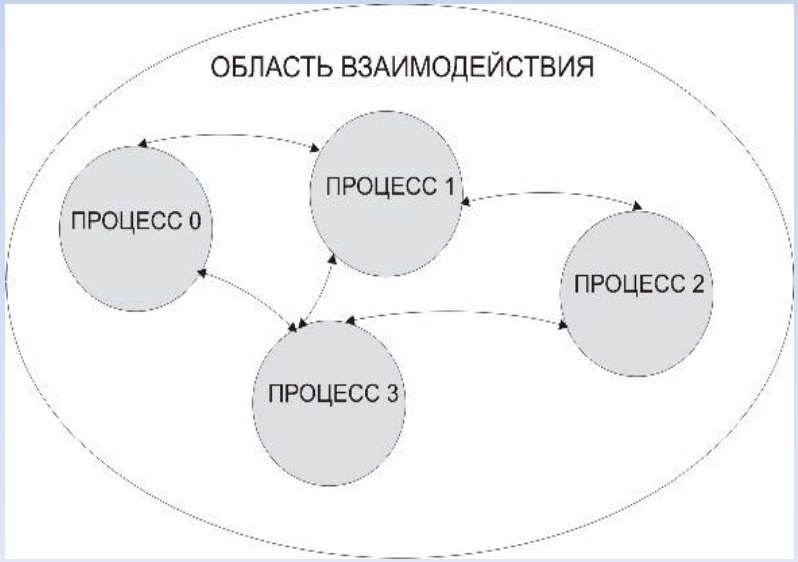
В двухточечном обмене участвуют только два процесса, процесс-отправитель и процесс-получатель (источник сообщения и адресат).
Двухточечные обмены используются для организации локальных и неструктурированных коммуникаций.



Разновидности двухточечного обмена

- *блокирующие* прием/передача, которые приостанавливают выполнение процесса на время приема или передачи сообщения;
- *неблокирующие* прием/передача, при которых выполнение процесса продолжается в фоновом режиме, а программа в нужный момент может запросить подтверждение завершения приема сообщения;
- *синхронный* обмен, который сопровождается уведомлением об окончании приема сообщения;
- *асинхронный* обмен, который таким уведомлением не сопровождается.

Двухточечный обмен возможен только между процессами, принадлежащими одной области взаимодействия (одному коммуникатору).

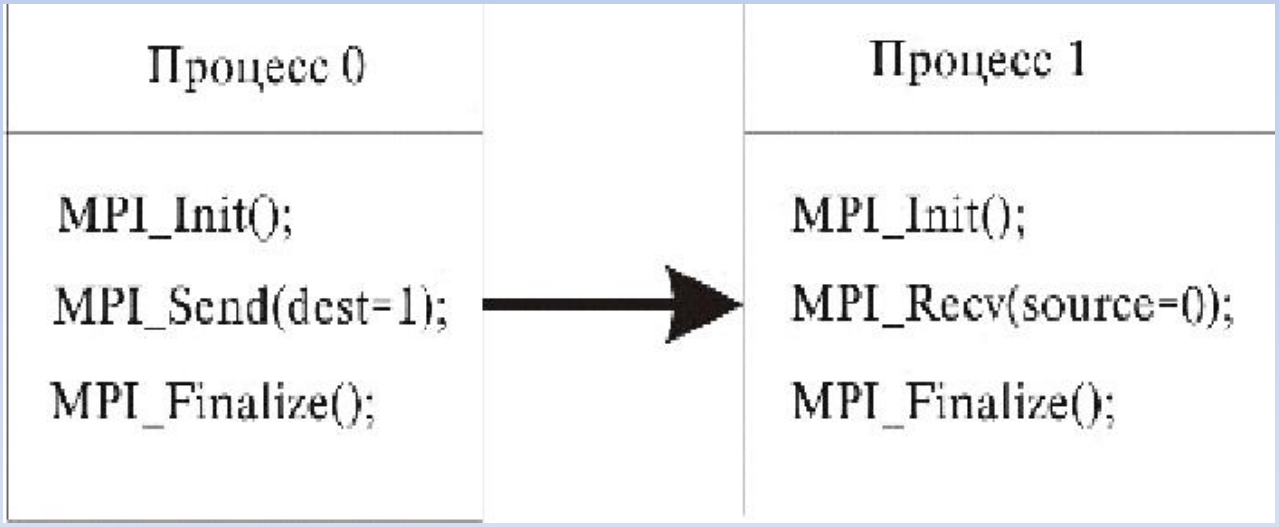


Правильно организованный двухточечный обмен сообщениями должен исключать возможность блокировки или некорректной работы параллельной MPI-программы.

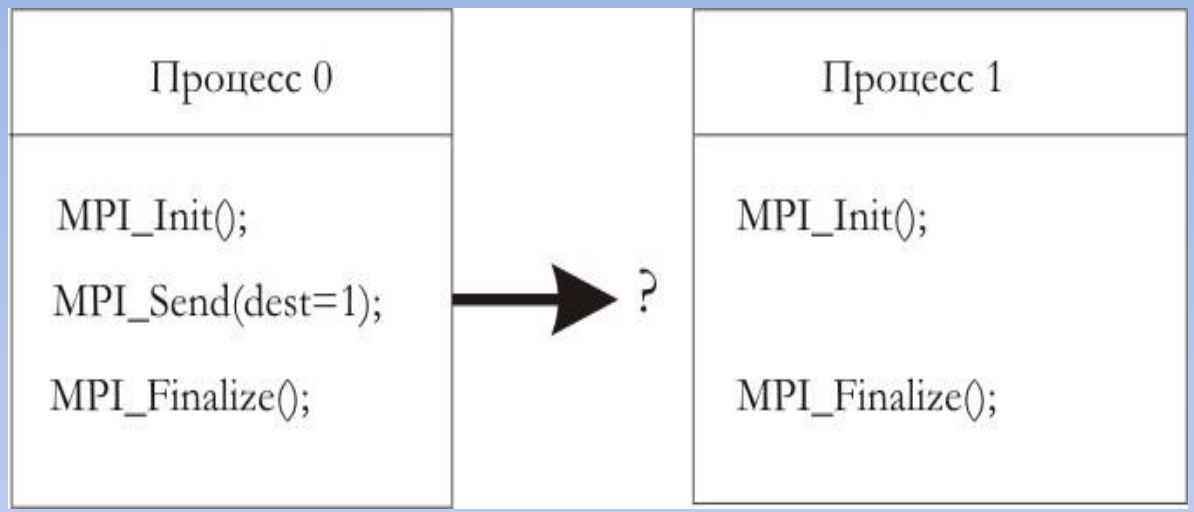
Примеры ошибок в организации двухточечных обменов:

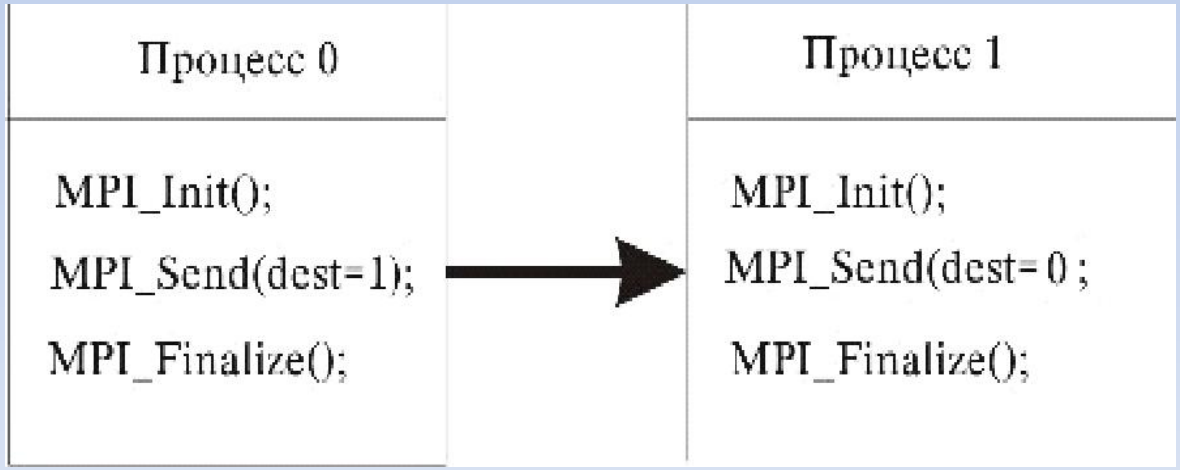
- выполняется передача сообщения, но не выполняется его прием;
- процесс-источник и процесс-получатель одновременно пытаются выполнить блокирующие передачу или прием сообщения.

Правильно



Неправильно





Стандартный блокирующий двухточечный обмен

Стандартная блокирующая передача

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype,int
dest, int tag, MPI_Comm comm)

MPI_Send(buf, count, datatype, dest, tag, comm,  ierr)
```

- `buf` - адрес первого элемента в буфере передачи;
- `count` - количество элементов в буфере передачи (допускается `count` = 0);
- `datatype` - тип MPI каждого пересылаемого элемента;
- `dest` - ранг процесса-получателя сообщения (целое число от 0 до n – 1, где n число процессов в области взаимодействия);
- `tag` - тег сообщения;
- `comm` - коммуникатор;
- `ierr` - код завершения.

При стандартной блокирующей передаче после завершения вызова (после возврата из функции/процедуры передачи) можно использовать любые переменные, использовавшиеся в списке параметров. Такое использование не повлияет на корректность обмена.
Дальнейшая «судьба» сообщения зависит от реализации MPI. Сообщение может быть сразу передано процессу-получателю или может быть скопировано в буфер передачи.

Завершение вызова не гарантирует доставки сообщения по назначению. Такая гарантия предоставляется при использовании других разновидностей двухточечного обмена (см. далее материал этой лекции).

Стандартный блокирующий прием

int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)

MPI_Recv(buf, count, datatype, dest, tag, comm, status, ierr)

- buf - адрес первого элемента в буфере приёма;
- count - количество элементов в буфере приёма;
- datatype - тип MPI каждого пересылаемого элемента;
- source - ранг процесса-отправителя сообщения (целое число от 0 до n – 1, где n число процессов в области взаимодействия);
- tag - тег сообщения;
- comm - коммуникатор;
- status - статус обмена;
- ierr - код завершения.

Подпрограмма MPI_Recv может принимать сообщения, отправленные в любом режиме.

Прием может выполняться от произвольного процесса, а в операции передачи должен быть указан вполне определенный адрес. Приемник может использовать «джокеры» для источника и для тега. Процесс может отправить сообщение и самому себе, но следует учитывать, что использование в этом случае блокирующих операций может привести к «тупику».

Пример использования операции блокирующего двухточечного обмена

```
#include <mpi.h>
#include <stdio.h>
int main (int argc, char *argv[]) {
	int ProcNum, ProcRank, tmp;
	MPI_Status status;
	MPI_Init ( &argc, &argv );
	MPI_Comm_size (MPI_COMM_WORLD, &ProcNum);
	MPI_Comm_rank (MPI_COMM_WORLD, &ProcRank);
	if(ProcRank == 0){
		printf("Hello world from process %i \n", ProcRank);
		for(int i = 1; i < ProcNum; i++){

	MPI_Recv(&tmp,1,MPI_INT,MPI_ANY_SOURCE,0,MPI_COMM_WORLD,
&status);
			printf("Hello world from process %i \n", tmp);
		}
	}else{
		MPI_Send(&ProcRank,1,MPI_INT,0,0,MPI_COMM_WORLD);
	}
	MPI_Finalize();
	return 0;
}
```

Пример программы, попадающей «в тупик»

```
program main_mpi
include 'mpif.h'
integer rank, tag, cnt, ierr, status(MPI_STATUS_SIZE)
real sndbuf, rcvbuf
tag = 0
sndbuf = 3.14159
cnt = 1
call MPI_Init(ierr)
call MPI_Comm_rank(MPI_COMM_WORLD, rank, ierr)
if (rank.eq.0) then
 call MPI_Recv(rcvbuf, cnt, MPI_REAL, 1, tag, MPI_COMM_WORLD, status, ierr)
 call MPI_Send(sndbuf, cnt, MPI_REAL, 1, tag, MPI_COMM_WORLD, ierr)
else
 call MPI_Recv(rcvbuf, cnt, MPI_REAL, 0, tag, MPI_COMM_WORLD, status, ierr)
 call MPI_Send(sndbuf, cnt, MPI_REAL, 0, tag, MPI_COMM_WORLD, ierr)
end if
call MPI_Finalize(ierr)
stop
end
```

Отладка и профилирование параллельных MPI-программ

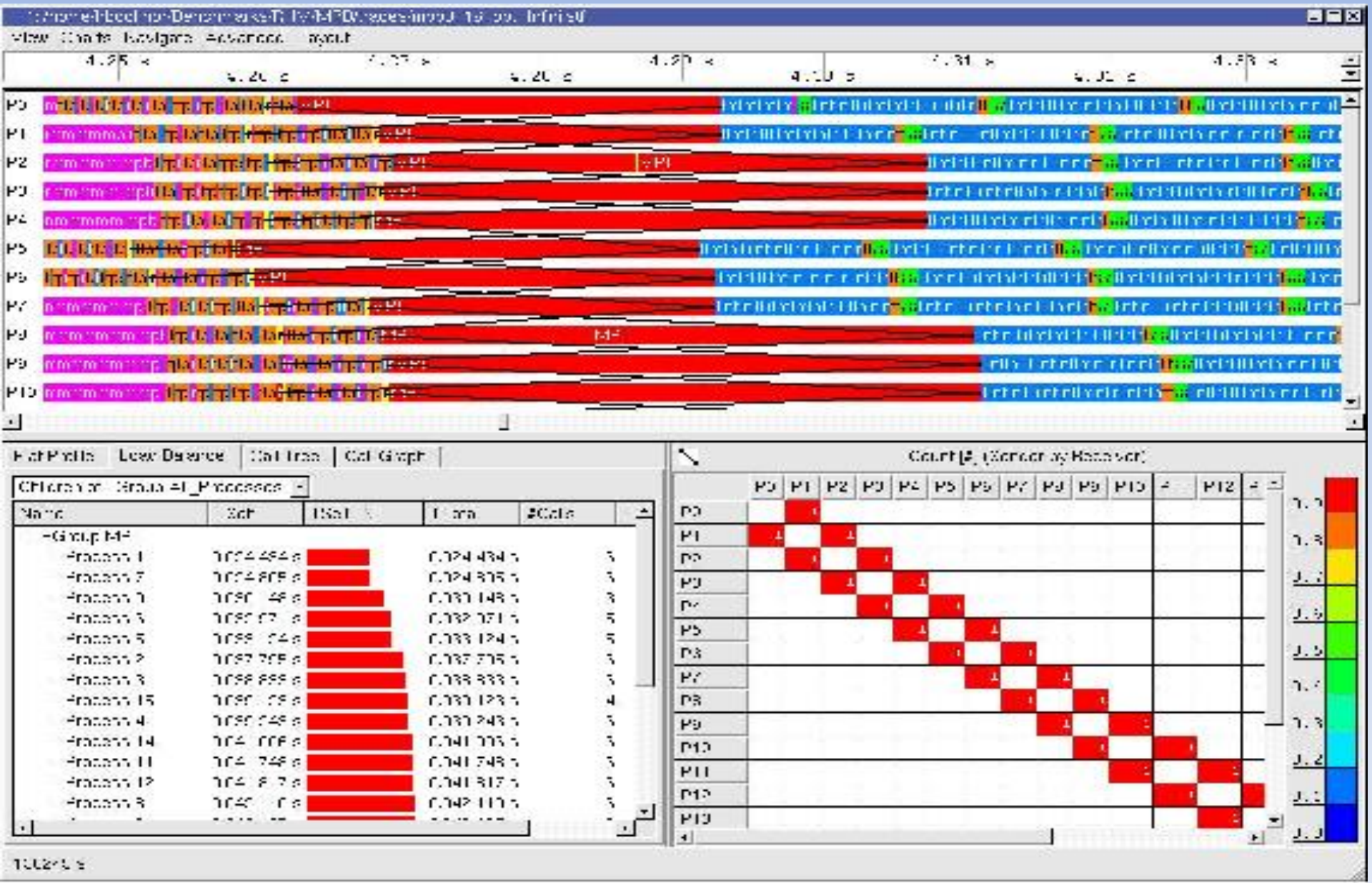
Отладка параллельных MPI-программ без использования специальных программных инструментов сложна и малоэффективна.

Существуют разные инструменты, среди них **jumpshot** – собственное средство отладки MPI.

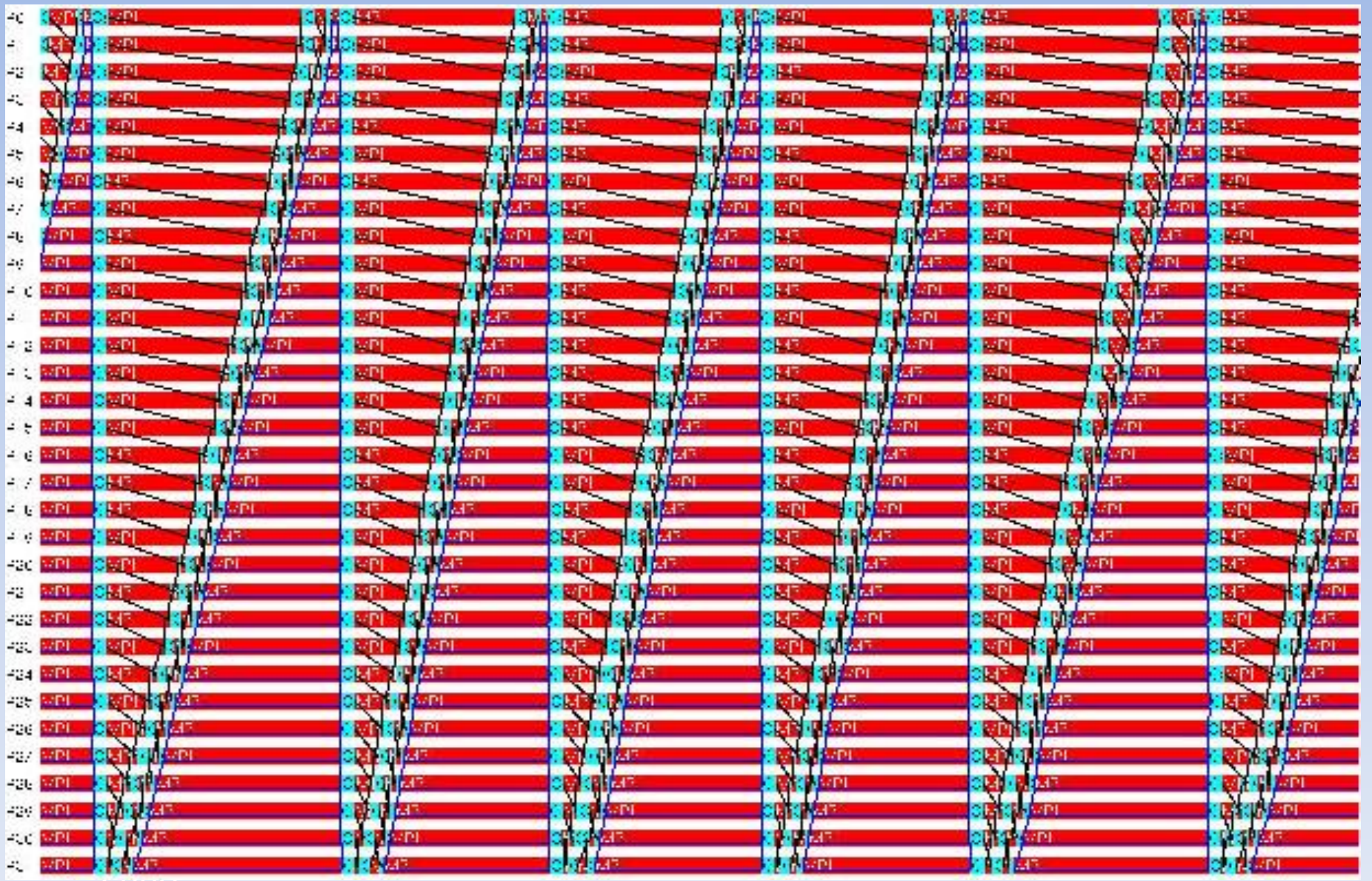
Intel ® Trace Analyzer and Collector – это инструмент анализа, для которого характерно следующее:

- ❑ анализ выполняется на основе статистики, собранной во время выполнения программы;
- ❑ «инструментовка» исполняемого файла почти не влияет на производительность программы;
- ❑ анализ выполняется и для обменов сообщениями;
- ❑ поддерживается OpenMP и гибридная модель параллельного программирования MPI+OpenMP;
- ❑ поддерживается многопоточность Java.

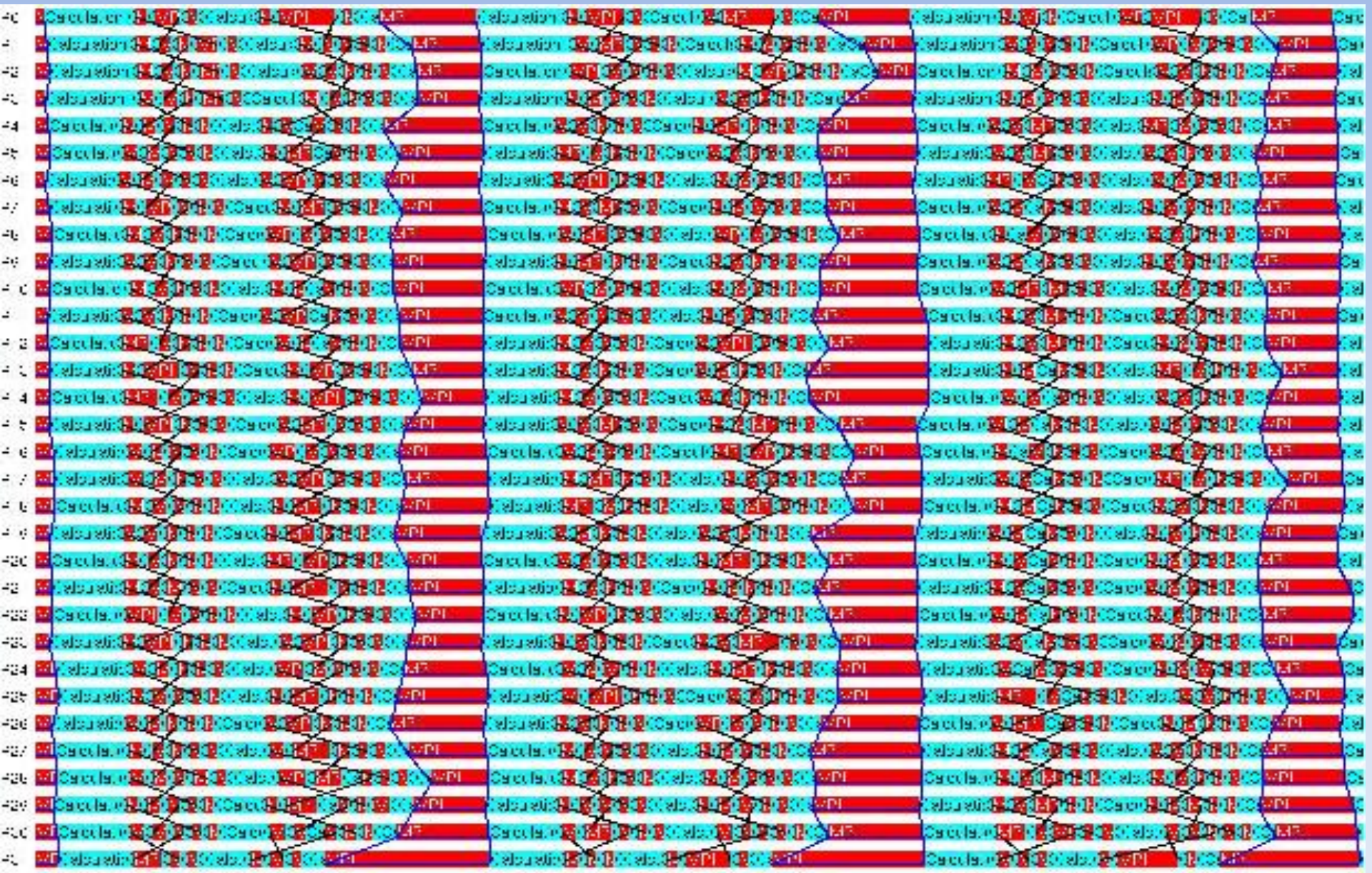
Интерфейс программы



Так выглядит «плохая» программа (блокирующие обмены):



Так выглядит «хорошая» программа (неблокирующие обмены):



Пример оптимизации промышленного кода

Выявляем «горячие пятна» (Intel® Vtune Performance Analyzer)

1060	subroutine aXpY(a, x, y, z)		1
1061	use numbers		
1062	implicit none		
1063			
1064	real x(0:Nx1,0:Ny1,0:Nz1),		
1065	& y(0:Nx1,0:Ny1,0:Nz1),		
1066	& z(0:Nx1,0:Ny1,0:Nz1)		
1067			
1068	real*8 a		
1069	integer i, j, k		
1070			
1071	!$omp parallel do	55881	74089
1072	!$omp& private(i, j, k)		
1073	do k=1,Nz	4	1
1074	do j=1,Ny	2	
1075	do i=1,Nx	56	31
1076	z(i,j,k) = a * x(i,j,k) + y(i,j,k)	87685	78394
1077	enddo	9906	12502
1078	enddo	323	268
1079	enddo		
1080	!$omp end parallel do		
1081	end	1	
1082	****** End of aXpY ******		
1083			
1084			
1085	****** Subroutine amult11 ******		
1086	* Perform multiplication {vec g} == [A] {vec phi} *		
1087	* *		
1088	* *		

Table 2. Most time-consuming functions

Name	Part of the total execution time
SOLVM13	34%
AMULT11	15%
AXPY	12%
DOTPRD11CELL	8%
DOTPRD11	6%
LUSOLVE	4%
ASMULT	4%

Находим решение проблемы (Intel® MKL)

```
1036        subroutine aXpY( a, x, y, z )
1037         use numbers
1038    *     implicit none
1039
1040         real    x(0:Nx1,0:Ny1,0:Nz1),
1041        &        y(0:Nx1,0:Ny1,0:Nz1),
1042        &        z(0:Nx1,0:Ny1,0:Nz1)
1043
1044         real*8  a
1045    *     integer i, j, k
1046
1047         integer dim
1048
1049         dim = Nx1 * Ny1* Nz1
1050
1051         z = daxpy(dim, a, x, 1, y, 1)
1052
1053    *     do k=1,Nz
1054    *       do j=1,Ny
1055    *         do i=1,Nx
1056    *           z(i,j,k) = a * x(i,j,k) +
          y(i,j,k)
1057    *         enddo
1058    *       enddo
```

```
1059    *     enddo
1060        end
1061        ********************************
          End of aXpY
          ********************************
```

Table 4. After aXpY optimization - most time-consuming functions:

	Before MKL	After aXpY MKL modified
SOLVM13	34%	60%
AMULT11	15%	31%
AXPY	12%	0.5%
DOTPRD11CELL	8%	8%
LUSOLVE	4%	
ASMULT	4%	