

## Лабораторная работа № 4

### Программирование многопроцессорных систем и использование векторных команд

Цель работы:

1. Познакомиться с программированием многопроцессорных систем с общей памятью на примере ПЭВМ с многоядерными процессорами.
2. Познакомиться с программированием векторных процессоров на примере векторных команд SSE.
3. Познакомиться с некоторыми возможностями современных оптимизирующих компиляторов и математических библиотек.
4. Познакомиться с программированием кластеров с использованием интерфейса MPI

#### 1. Введение

Данная работа посвящена предварительному знакомству с многопроцессорными системами и векторными процессорами на примере современных многоядерных процессоров фирмы Intel и с некоторыми технологиями их программирования. Материал работы в значительной степени основан на материалах курса «Введение в методы параллельного программирования» ННГУ (авт. В.П. Тергель, А.А.Лабутина и др.) / 1 /

#### 2. Классификация и некоторые основные понятия многопроцессорных ВС.

Согласно **классификации Флинна** ВС по взаимодействию потоков команд и данных делятся на 4 группы :

- SISD (ОКОД – одиночный поток команд и данных)
- SIMD (ОКМД – одиночный поток команд и много потоков данных)
- MISD (МКОД – много потоков команд и один поток данных)
- MIMD (МКМД – много потоков команд и данных)

В данной работе нас в основном интересуют **SIMD** системы (векторные процессоры) и **MIMD** (основной класс многопроцессорных систем).

Согласно другим классификациям, в основном касающимся MIMD, можно выделить т.н. системы с **общей памятью (разделяемой)**, которые обычно относят к **мультипроцессорам**, и системы с **передачей сообщений (мультикомпьютеры или кластеры)**. В системах с общей памятью процессоры могут иметь однородный доступ к памяти (**UMA**) и даже – симметричный доступ (все процессоры равноправные и вся система симметрична и однородна), в последнем случае системы называют **SMP**, либо – неоднородный доступ (**NUMA** - время доступа зависит от адреса).

Другой классификационный признак – используемый системами **тип параллелизма** (или – тип параллелизма, на который они рассчитаны). Это параллелизм независимых ветвей, естественный (векторный или матричный) параллелизм, параллелизм смежных операций и др. К естественному параллелизму можно отнести также задачи с параллелизмом по данным, в том числе – независимые по данным. В последнем случае обмен между независимыми ветвями может быть минимальным. Кроме того, выделяют **мелкозернистый** параллелизм и **крупнозернистый** параллелизм, в зависимости от масштаба и количества параллельных ветвей.

Многопроцессорные системы могут использовать разный тип параллелизма задачи, но, как правило, не параллелизм смежных операций – его используют

суперскалярные процессоры и процессоры с длинным командным словом. При этом для задач с интенсивным обменом по понятным причинам предпочтительнее сильносвязанные SMP системы. С другой стороны, естественный параллелизм и мелкозернистые задачи при сравнительно низкой интенсивности обмена могут максимально использоваться в кластерах.

### 3. Краткий обзор используемых в работе технологий

#### 3.1 OpenMP

Одним из наиболее популярных средств программирования компьютеров с общей памятью, базирующихся на традиционных языках программирования и использовании специальных комментариев, в настоящее время является технология **OpenMP**. За основу берется последовательная программа, а для создания ее параллельной версии *пользователю предоставляется набор директив, процедур и переменных окружения*. Стандарт OpenMP разработан для языков Фортран, С и С++.

Разработкой стандарта занимается организация OpenMP Architecture Review Board, в которую вошли представители крупнейших компаний - разработчиков SMP-архитектур и программного обеспечения.

Эффективность подхода OpenMP заключается в том, что разделяемые для параллельных процессов данные располагаются в общей памяти и для организации взаимодействия не требуется операций передачи сообщений.

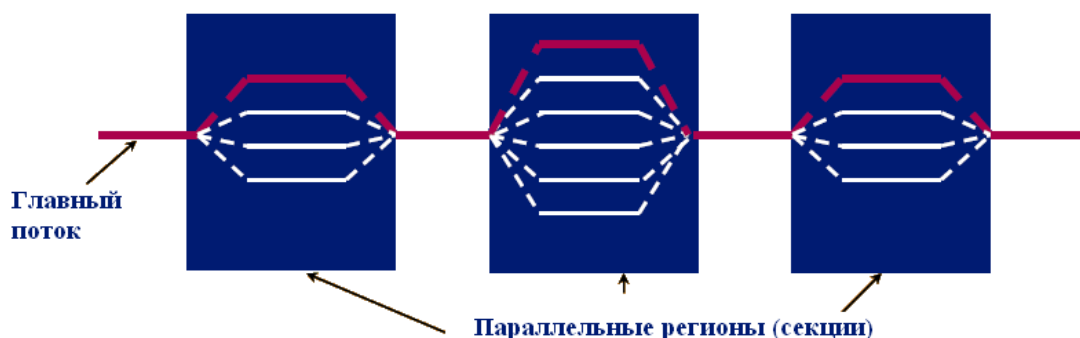


Рис.1 Процесс исполнения OpenMP программы

Весь текст программы разбит на последовательные и параллельные области (см. рис.1). В начальный момент времени порождается нить-мастер или "основная" нить, которая начинает выполнение программы со стартовой точки. Технология OpenMP опирается на понятие общей памяти, поэтому она, в значительной степени, ориентирована на SMP-компьютеры. На подобных архитектурах возможна эффективная поддержка нитей, исполняющихся на различных процессорах, что позволяет избежать значительных накладных расходов на поддержку классических UNIX-процессов.

Основная нить и только она исполняет все последовательные области программы. При входе в параллельную область порождаются дополнительные нити. После порождения каждая нить получает свой уникальный номер, причем нить-мастер всегда имеет номер 0. Все нити исполняют один и тот же код, соответствующий параллельной области. При выходе из параллельной области основная нить дожидается завершения остальных нитей, и дальнейшее выполнение программы продолжает только она.

OpenMP - ориентирована на распараллеливание циклов, поэтому многие разработчики называют подобные технологии *for* – ориентированными.

### Конструкции OpenMP

OpenMP прост в использовании и включает лишь два базовых типа конструкций: директивы `pragma` и функции исполняющей среды OpenMP. Директивы `pragma`, как правило, указывают компилятору реализовать параллельное выполнение блоков кода. Все эти директивы начинаются с `#pragma omp`. Как и любые другие директивы `pragma`, они игнорируются компилятором, не поддерживающим конкретную технологию — в данном случае OpenMP.

Функции OpenMP служат в основном для изменения и получения параметров среды. Кроме того, OpenMP включает API-функции для поддержки некоторых типов синхронизации. Чтобы задействовать эти функции библиотеки OpenMP периода выполнения (исполняющей среды), в программу нужно включить заголовочный файл `omp.h`. Если вы используете в приложении только OpenMP-директивы `pragma`, включать этот файл не требуется.

Для реализации параллельного выполнения блоков приложения нужно просто добавить в код директивы `pragma` и, если нужно, воспользоваться функциями библиотеки OpenMP периода выполнения. Директивы `pragma` имеют следующий формат:

**`#pragma omp <директива> [раздел [ [,] раздел]...]`**

OpenMP поддерживает директивы `parallel`, `for`, `parallel for`, `section`, `sections`, `single`, `master`, `critical`, `flush`, `ordered` и `atomic`, которые определяют или механизмы разделения работы или конструкции синхронизации.

Раздел (clause) — это необязательный модификатор директивы, влияющий на ее поведение. Списки разделов, поддерживаемые каждой директивой, различаются, а пять директив (`master`, `critical`, `flush`, `ordered` и `atomic`) вообще не поддерживают разделы.

### 3.2 Векторные команды SSE

Streaming SIMD Extensions SSE – это векторные команды с плавающей запятой, выполняемые процессором в специальном блоке. Это развитие системы команд MMX (MultiMedia eXtensions – мультимедийные расширения). MMX предлагает работу с целочисленными векторами с количеством элементов от 1 до 8. При этом используются 64-разрядные регистры MMX, физически размещаемые в регистрах сопроцессора с плавающей запятой. Подробнее с MMX можно познакомиться в предыдущей версии лабораторной работы №4 и с помощью мультимедийного руководства фирмы Intel ([aae.vistcom.ru](http://aae.vistcom.ru)).

В системе команд SSE (а также 2, 3 и 4) используются 128-битные специальные регистры XMM и отдельные операционные устройства. Допускается как обработка с плавающей, так и с фиксированной запятой.

Для работы с SSE необходимо использовать либо команды ассемблера, либо – т.н. **intrinsic** – “интринсики” – специальные мини-функции на языке высокого уровня (в данной работе - C++), позволяющие напрямую работать с регистрами SSE и выполнять SSE операции. За одну операцию SSE параллельно выполняются либо 4 операции с плавающей запятой одинарной точности (`float`), либо – 2 операции двойной точности (`double`).

### 3.3 MPI

MPI – Message Passing Interface – Интерфейс передачи сообщений. Представляет собой API для организации передачи сообщений между процессами, запущенными как на одном, так и на разных вычислительных узлах кластера. (coming soon :) )

#### 4. Используемые программные средства и их настройка

В работе используются Visual Studio 2005 (либо Express + Platform SDK и Professional), компилятор Intel C++ Compiler 9.1 (или более поздний), библиотека MPICH 2.

При создании проекта в Visual C++ создайте пустой (empty) проект типа Win32 Console Application и добавьте прилагаемый пример кода. Затем отключите в свойствах проекта использование Precompiled Headers (**Not using precompiled headers**). Свойства проекта удобнее всего вызвать из окна **Solution Explorer** (View – Solution Explorer). Используйте версию **release** проекта. При запуске программ запускайте их из командной строки или файлового менеджера, а не из оболочки.

Для использования компилятора Intel C++ в среде Visual Studio 2005 Standard/Professional/... (не в Express) проект Visual Studio необходимо конвертировать для использования компилятора Intel (по умолчанию в проектах Visual Studio используется компилятор Microsoft). Убедитесь, что на вашем компьютере установлен компилятор Intel. Далее щелкните правой кнопкой мыши на названии проекта в окне Solution Explorer и в появившемся контекстном меню выберите пункт **“Convert to use Intel(R) C++ Project System”** (см. рис. 2).

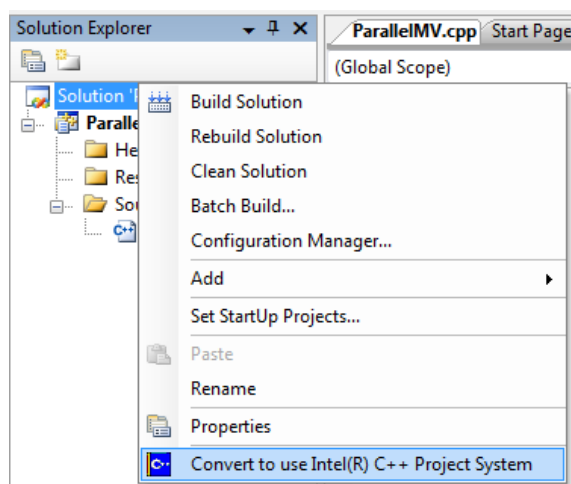


Рис. 2

В результате появится предупреждение о том, что формат проекта изменится на формат, подходящий для компилятора. На это предупреждение надо ответить согласием, то есть нажать **“Yes”**.

Если все действия были проделаны правильно, то после их выполнения окно проекта Solution Explorer должно иметь вид, представленный на рисунке 3 (Обратите внимание на пиктограмму C++ на синем фоне – она указывает на использование компилятора Intel)

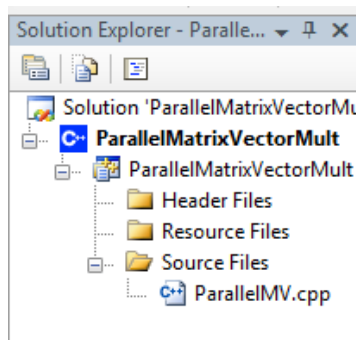


Рис. 3

По поводу настройки программных средств – обращайтесь к преподавателю и читайте мануалы :) (to be continued ... :) )

## 5. Решение тестовой задачи разными способами

### 5.1 Описание тестовой задачи

В качестве примера рассмотрим задачу **умножения матрицы на вектор**. Это типовая задача, решаемая во многих библиотеках программ и часто являющаяся базовой для решения матричных задач более высокого уровня.

Псевдокод, описывающий эту задачу, может быть таким :

Исходные данные:	$A[n][m]$ – матрица размерности $n \times m$ .
	$b[m]$ – вектор, состоящий из $m$ элементов.
Результат:	$c[n]$ – вектор из $n$ элементов.

```
// Последовательный алгоритм умножения матрицы на вектор
for (i = 0; i < n; i++) {
    c[i] = 0;
    for (j = 0; j < m; j++) {
        c[i] += A[i][j]*b[j]
    }
}
```

### 5.2 Последовательный код.

Используемый в данной работе код приводится в / 1 /. Он включает основную функцию `main()`, служебные функции для расчета времени с помощью WinAPI, функции создания, инициализации и уничтожения матрицы и векторов, и, наконец, собственно функцию расчета произведения, аналогичную представленному выше псевдокоду :

```

void SerialResultCalculation (float* pMatrix, float* pVector,
                             float* pResult, int Size) {
    int i, j; // Переменные цикла
    for (i=0; i<Size; i++)
    {
        for (j=0; j<Size; j++)
            pResult[i] += pMatrix[i*Size+j]*pVector[j];
    }
}

```

Эта функция отличается от приведенного выше псевдокода во-первых, именами, во-вторых - особенностью работы с массивами в C++, а в-третьих тем, что здесь выполняется обработка только квадратных матриц.

Заслуживают внимания также функции точного определения времени с помощью WinAPI :

```

// Функция преобразует тип LARGE_INTEGER в double
double LiToDouble (LARGE_INTEGER x) {
    double result =
        ((double)x.HighPart) * 4.294967296E9 + (double)((x).LowPart);
    return result;
}

// Функция возвращает текущее время в секундах в виде double
double GetTime() {
    LARGE_INTEGER lpFrequency, lpPerfomanceCount;
    QueryPerformanceFrequency (&lpFrequency);
    QueryPerformanceCounter (&lpPerfomanceCount);
    return LiToDouble(lpPerfomanceCount)/LiToDouble(lpFrequency);
}

```

После компиляции и прогона программы на ПК с процессором Celeron M 1.5 получаем для размерности 10000 время задержки, равное 1.27 с. Следует отметить, что для создания программ использовался компилятор Intel C++ Compiler 9.1. (В данном случае оптимизация была отключена.)

### 5.3 Распараллеливание с помощью OpenMP

Распараллелить выполнение приведенной выше функции с помощью прагм OpenMP несложно. Достаточно добавить одну (!) прагму OMP и указать опцию компилятора **/Qopenmp** (Свойства проекта/C++/Language). Вид этой прагмы следующий :

**#pragma omp parallel for private (j)**

Собственно **omp** указывает на прагму OpenMP, **parallel** – на то, что это прагма распараллеливания, **for** – что происходит распараллеливание цикла **for** (а именно – того, который следует сразу за прагмой), а предложение **private(j)** указывает, что переменная **j** должна быть уникальна (локальна) в каждой нити (поточе).

```

void ParallelResultCalculation (float* pMatrix, float* pVector, float* pResult, int Size)
{
    int i, j; // Loop variables
    #pragma omp parallel for private (j)
    for (i=0; i<Size; i++)
    {
        for (j=0; j<Size; j++)
            pResult[i] += pMatrix[i*Size+j]*pVector[j];
    }
}

```

Для устранения ошибок при запуске программы во-первых, включите в файл ссылку на заголовочный файл <omp.h> (хотя согласно документации это не требуется), а во-вторых, возможно, потребуется найти соответствующую динамическую библиотеку и поместить ее в каталог release, где находится исполняемый файл вашей программы. При поиске dll -файла обратите внимание, что Вы выбрали файл для нужной платформы (x86 32 бита – на это указывает имя каталога, в котором размещается библиотека).

#### 5.4 Векторизация с помощью SSE/SSE2

Операции с матрицами хорошо векторизуются, следовательно, мы можем использовать SIMD-расширения процессоров, а именно – SSE (MMX не позволяет работать с плавающей запятой). Но для использования SSE инструкций нам необходимо явно указывать их, поэтому код основной расчетной функции придется переделать.

Все арифметические SSE инструкции оперируют над векторами, длина которых зависит от типа данных: для float n=4, для double n=2. Будем оперировать типом float, поскольку для него прирост производительности заметнее :). Заметим, кстати, что и приведенные ранее примеры используют тип float.

Будем использовать по-прежнему ленточную схему распараллеливания, а вернее – просто оригинальный алгоритм умножения. Отличие в данном случае состоит в том, что мы просто уменьшим количество шагов при вычислении каждого элемента во внутреннем цикле в 4 раза – по числу элементов, параллельно обрабатываемых в SSE-инструкции.

На каждом шаге теперь мы будем брать по 4 элемента строки матрицы, по 4 элемента вектора и перемножать их одной командой SSE, а затем – складывать одной командой сразу 4 пары элементов. Все, что останется сделать в конце цикла – это сложить 4 частичных суммы явно. Для этого в SSE3 есть специальная команда «горизонтального» сложения, а вот в SSE2, к сожалению, ее нет, поэтому в приведенном ниже примере приходится явно перегружать вектор из 4 компонент в массив, а потом – складывать элементы массива.

```

void SerialResultCalculationSSE(float* pMatrix, float* pVector,
                                float* pResult, int Size)
{
    __m128 row0;
    __m128 row1;
    __m128 row2;
    __m128 res4;

    float* res_4;
    float res=0;

    res_4 = (float *)_aligned_malloc(4*sizeof(float), 16);

```

```

int i,j,k;  // Переменные цикла

for (i=0; i < Size; i++)
{
    res4=_mm_setzero_ps();

    for (j=0; j < Size; j=j+4)
    {
        row1 = _mm_load_ps(&pVector[j]); //загрузка вектора
        row0 = _mm_load_ps(&pMatrix[Size*(i)+j] ); //загрузка строки матрицы
        row2 = _mm_mul_ps(row0, row1); //перемножение
        res4 = _mm_add_ps(row2, res4); // накопление 4-х сумм
    }

    _mm_store_ps(&res_4[0], res4); // переброс в массив

    pResult[i] = res_4[0] + res_4[1] + res_4[2] + res_4[3]; // окончательное
    // формирование элемента
}
}

```

## 5.5 Использование компиляторной оптимизации

Современные компиляторы языков C++ и Fortran (а также и других, в частности, Java, языков платформы .NET) позволяют оптимизировать код как на уровне алгоритма (разворачивание циклов, автоматическое распараллеливание и векторизация), так и на уровне учета особенностей конкретного процессора (поддержка векторных команд, количество потоков команд и ядер, объем и организация кэш-памяти).

В данной работе мы изучаем два компилятора – Microsoft Visual C++ 8.0 из пакета Visual Studio 2005 и Intel C++ Compiler 9.1 (и старше). Оба компилятора допускают оптимизацию с ключами **/O1**, **/O2**, **/O3**. Опция **/Od** отключает оптимизацию. Оба компилятора учитывают особенности процессоров Intel, наличие векторных команд SSE/SSE2/SSE3, допускают автоматическое распараллеливание и векторизацию. Вполне естественно, что компилятор от Intel делает это несколько лучше.

Для компилятора Intel нужно использовать специфическую для него оптимизацию **/Ofast** и оптимизацию для Core 2 Duo. (в разделе Intel Specific свойств проекта/C++/Optimization).

## 5.6 Распараллеливание с помощью MPI на кластере

(coming soon :)

## 5.7 MKL

В библиотеке Intel **MKL** (Math Kernel Library) реализованы в том числе функции для быстрого решения СЛАУ. Естественно, что эта библиотека оптимизирована для процессоров Intel и позволяет получать значения производительности, близкие к теоретической для этих процессоров.

(coming soon :)

## 6. Задания

### Порядок выполнения работы следующий :

6.1 Рассмотреть примеры программ, выполнить их компиляцию и сборку, а затем – исследовать время вычислений (по 3 прогона) для разной размерности задачи. Заполнить таблицы для двух компиляторов – VC++ и Intel C++ (ускорение меряется относительно последовательной программы, скомпилированной VC++) без оптимизации и с оптимизацией для размерностей от 1000 до 10000 с шагом 1000:

Раз- мер- ность N	После- дова- тель- ная про- грам- ма	С исполь- зова- нием OpenMP	Уско- рение	SSE	Уско- рение	После- дова- тель- ная с опти- миза- цией	Уско- рение	SSE с опти- миза- цией	Уско- рение	MPI	Уско- рение
1000											

Сделайте выводы о полученных результатах.

Приведите теоретическую оценку производительности при решении этой задачи.

6.2 Определите производительность компьютера по тесту linpack 9.1 из библиотеки Intel MKL

Тест linpack представляет собой стандартный тест производительности, в котором выполняется решение системы линейных алгебраических уравнений (СЛАУ) большой размерности. Его оптимизированный вариант, использующий библиотеку MKL, включен в состав этой библиотеки и доступен для скачивания с сайта Intel (?).

Откройте папку benchmark/linpack. Там содержатся командные файлы runme\_xeon32.bat и runme\_xeon64.bat. Их нужно запустить, предварительно настроив количество параллельных потоков OMP\_NUM\_THREADS и содержимое файлов с параметрами linput\_xeon32 и linput\_xeon64. В них указывается количество запускаемых тестов и размерность матрицы. Укажите размерности 1000 и 5000 :

```
2                # number of tests
1000 5000        # problem sizes
1000 5000        # leading dimensions
4 1              # times (trials) to run a test
4 1              # alignment values (in KBytes)
```

Расчитайте максимальную размерность, при которой матрица будет помещаться в память вашей машины (не забудьте оставить место для самой ОС :).

Также приведите теоретическую оценку производительности при решении этой задачи и пиковое значение производительности для процессора вашей машины.

6.3 Усовершенствуйте программу, использующую SSE – включите инструкции SSE3.

6.4 Выполните индивидуальные задания :

1. Скалярное произведение векторов (в цикле)
2. Сумма всех элементов матрицы

3. Среднее арифметическое строк (столбцов) матрицы
4. Умножение матрицы на множитель
5. Сложение матриц
6. Сумма квадратов элементов строк матриц (еще – см. задания по старому варианту 4 работы)

## **7. Вопросы**

**(coming soon)**

## **8. Дополнительная литература и ссылки**

1. Гергель В.П. Многопроцессорные вычислительные системы и параллельное программирование - Материалы учебного курса ННГУ. - [www.software.unn.ac.ru](http://www.software.unn.ac.ru)