Реферат

на тему

«Библиотека OpenMP»

Преподаватель: Кулябов Дмитрий Сергеевич

Выполнил: Кремер Илья

Группа: НК-401

Москва, РУДН – 2011

Оглавление

Распараллеливание программ 2

Основные способы распараллеливания 3

Визуализация распараллеливания 4

OMP – «распараллеливающие» конструкции 4

Основные инструменты 5

Демонстрация работы 6

Способы использования 8

  Распараллеливание циклов 8

  Эффективность 10

  Дополнения к директиве for 11

  Нахождение минимума/максимума 12

  Параллельное вычисление рекуррентных последовательностей 13

Заключение 17

Литература и др. источники 17

Распараллеливание программ

Существуют две основные причины, вызывающие необходимость распараллеливать программы:

- Выполнение программой двух и более одновременных действий, задач.

Например, прослушивание нажатий на клавиши клавиатуры или щелчков мыши по каким-то элементам в окне выполняется в отдельном потоке. Иначе реализация графического интерфейса не представляется возможной. Кроме того, большинство программ очень часто выполняют много различных задач одновременно. Например, когда мы открываем в браузере несколько вкладок сразу, то каждая из них загружается и прорисовывается вместе с другими.

- Ускорение работы программ на современных компьютерах (с 2005 на ПК, с 2001 на серверах) и эффективное использование суперкомпьютеров (с 1960-х).

С суперкомпьютерами всё просто – идея оных заключается в объединении нескольких вычислительных машин в одно целое, а значит необходимо разбить выполнение задачи на несколько частей для того, чтобы она выполнялась быстрее – разные части на разных узлах.

С современными обычными компьютерами дело обстоит несколько интересней – развитие процессоров для ПК, ноутбуков, планшетов и других устройств, использующихся в повседневной жизни, а также для машин, работающих на серверах с какого-то момента, пошло по пути установки двух и более процессоров в один процессорный элемент. Такие процессоры известны всем по названию «многоядерные». Так, в 2001 году в продажу поступил первый двухъядерный процессор IBM Power4, предназначенный для серверов. В 2002 вышли Intel Xeon и Intel Pentium 4, использующие технологию Hyper-Threading – виртуальная двухпроцессорность на одном кристалле, которую можно использовать эффективно только распараллелив приложение.

Весной 2005 года Intel выпустила первый в мире двухъядерный процессор с x86 – Pentium D. А в недалёком 2006 все ознакомились со знаменитой линейкой, название которой уже в те дни говорило само за себя – Intel Core 2. Почти в то же самое время вышел первый аналогичный AMD – Athlon 64 X2. В настоящее время двух и более ядерные процессоры имеют

большую популярность, чем одноядерные, а совсем недавно (2011) начали появляться смартфоны с двухъядерными процессорами на борту.

Сразу следует ещё раз обратить внимание на то, что появление таких процессоров вызывает необходимость распараллеливать программы. Можно подумать, что одна и та же программа должна работать на одноядерном процессоре медленнее, чем на двухъядерном, но на деле это не так, даже если сами ядра примерно равны по мощности. Обязательно необходимо, чтобы программа знала и говорила процессору, что именно можно выполнять одновременно.

Хотя конечно, если запустить две и более «тяжёлых» программы одновременно, то при всех прочих равных условиях, двухъядерный процессор окажется впереди, т.к. здесь уже ОС займётся тем, что скажет, на каком ядре какую запускать и в итоге оба ядра будут работать на свою полную мощность, тем самым заметно выигрывая у одноядерного.

Основные способы распараллеливания

- Встроенные в язык средства создания новых потоков (threads). Например, это функция `fork()` в C, класс `Thread` в Java.

Конечно такие инструменты в основном связаны с первой причиной, вызывающей необходимость распараллеливания (см. пред. раздел).

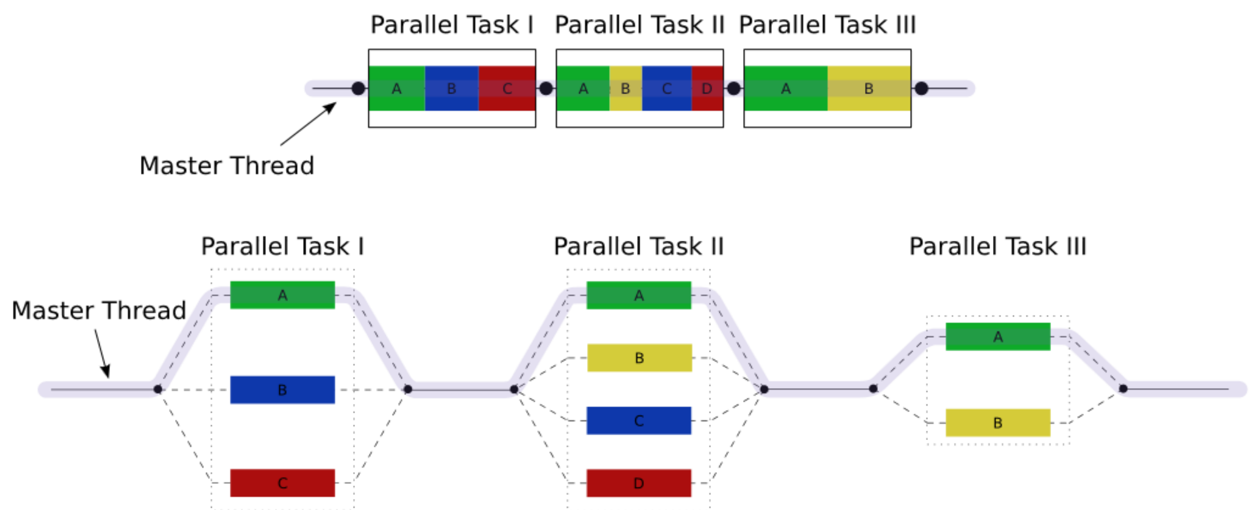
- Дополнительные инструменты, обеспечивающие возможность передачи сообщений между процессами, например MPI.

Этот вариант в основном используется в суперкомпьютерах.

В общих чертах смысл следующий: одна программа запускается несколько раз, каждая копия на разном узле (ядре процессора в случае с ПК). Эти запуски производятся специальной программой, которая входит в инструментарий для распараллеливания; они организованы таким образом, что каждый процесс (запущенная копия) знает свой номер (а номера идут последовательно с нуля) и может послать или принять сообщение на любой другой номер, т.е. любому другому процессу или от процесса. Так процессы делают свои куски общего задания и обмениваются промежуточными значениями и результатами, чтобы в каком-то процессе потом собрать конечный ответ.

- Внедрение специальных «распараллеливающих» конструкций в уже написанную программу, например, технология OpenMP, являющаяся темой этого реферата.

Визуализация распараллеливания



Серая нить на рисунке – это как бы выполнялась программа, не будь она распараллелена. В этом случае её распараллеливают три раза. Первый раз на 3 нити (процесса), второй раз – на 4, и третий – на 2. Цветные кусочки – это части выполняемого задания на время распараллеливания.

OMP – «распараллеливающие» конструкции

OMP – очень распространённый инструмент для распараллеливания программ, в том числе в прикладной сфере, т.е. в программах, имеющихся у многих из нас на ПК. Он имеет ряд достоинств:

- Сначала пишем, потом распараллеливаем.

Это не всегда так, но по отношению ко второму типу инструментов (напр. MPI) это 100% всегда не так. Поскольку все пересылки делаются вручную, возникает необходимость заранее продумать ход её выполнения (как процесс, выдающий ответ будет его собирать?, будет ли программа работать на любом количестве процессов или только на определённом или определённых?).

- Очень маленький, следовательно, простой для изучения набор директив.

Маленький и простой – понятия относительные, но дело в том, что по крайней мере существует несколько очень синтаксически простых и интуитивно понятных директив (будут рассмотрены в реферате), которые подходят для огромного количества случаев.

- Поддерживается многими компиляторами.

Вообще OMP существует для C/C++ и Fotran. Выбор разработчиков пал на эти разные языке не просто так – C/C++ являются языками разработки большинства современных ресурсоёмких прикладных программ, а Fortran – популярен в научной сфере и в промышленных вычислениях, т.е. активно используется на суперкомпьютерах наряду с C и C++. OpenMP включён и в GCC (gcc/gfotran) и в среды Microsoft (различные Visual Studio) и в компиляторы от Intel.

- Не портит код, если не поддерживается.

Это утверждение, как и первое тоже не 100% истинно, но некоторые примеры будут продемонстрированы в реферате. Смысл его очень прост – многие из конструкций, о которых идёт речь в заголовке предназначены для компилятора и никак не сказываются на конечном бинарном файле, если компилятор их не понял.

Кроме того всегда остаётся возможность управления компиляцией с помощью проверки:

```
#ifdef _OPENMP
    // код, использующий функции OMP
#else
    // обойтись без OMP
#endif
```

Основные инструменты

Управление параллелилизмом ведётся с помощью директив (в C/C++ это `#pragma omp` *`директива`*, в Fotran – `!$omp` *`директива`*, нескольких функций и некоторых переменных окружения.

Основные директивы:

- `parallel` – задаёт начало параллельной области
- `do/for` – автоматическое распараллеливание циклов (`do` – для Fotran, `for` – для C).

- barrier, master, sections – управление потоками: синхронизация, отдельные указания
- Различные дополнения к директивам.

Основные функции:

- omp_get_num_threads() – узнать, сколько потоков в текущей параллельной области
- omp_get_thread_num() – узнать номер текущей нити (все нити нумеруются с 0).
- Функции управления свойствами параллельных областей: omp_set_nested(true|false), omp_set_dynamic(true|false) и др.

Переменные окружения:

- OMP_DYNAMIC – значение говорит о том, разрешено ли программе менять количество процессов динамически.
- OMP_NUM_THREADS – количество процессов в параллельной области по умолчанию, и пр.

Демонстрация работы

Следующая простая программка поможет легко понять, как работает код, использующий OMP (компиляция этого кода и всех приложенных к реферату программ делается след. образом: **gcc -fopenmp code.c** или аналогично с помощью gortran для Fotran):

```
 1 #include <stdio.h>
 2 #include <omp.h>
 3
 4 int main() {
 5     int pid;
 6     printf("До распараллеливания... \n");
 7 #pragma omp parallel num_threads(3), private(pid)
 8   {
 9     pid = omp_get_thread_num();
10     printf("Hello, World! (pid: %d)\n", pid);
11   }
12 }
```

Программа выведет 4 сообщения, первое – «До распараллеливания...», затем три сообщения «... (pid 0)», «... (pid 1)» и «... (pid 2)», порядок которых непредсказуем. Разберёмся в выполнении этого кода.

6 строчка выполняется, как ей и следует, т.к. она остаётся нетронутой инструментами OMP. Далее идёт директива, которая указывает, что всё, что находится в фигурных скобках, принадлежащих ей (код между 8 и 11 строчками) должно выполнится в трёх процессах (`num_threads(3)`), так, что переменная `pid` в каждом процессе своя (а не одна на все процессы) – `private(pid)`, о том, что означает последнее – чуть позже.

В итоге каждый процесс узнает свой номер и сохранит его в переменной `pid`, после чего выведет сообщение.

Очень важно, что при выводе необходимо использовать буферные функции, иначе сообщения с разных процессов могут перебивать друг друга (из середины одного начнёт идти другое). Функция `printf()` и конструкция `print` в Fotran являются буферными.

Посмотрим, как ещё можно задать количество процессов (кроме, чем как дополнением директивы `parallel`, директивой `num_threads()`):

- По запросу пользователя, с помощью чтения из файла и т.д.:

```
    int numthreads;
    printf("Количество процессов: ");
    scanf ("%d", &numthreads);
    if (numthreads > 128)
        numthreads = 128;
    omp_set_num_threads(numthreads);
#pragma omp parallel //опускаем num_threads(n)
```

В случае некорректного, но допустимого по типу ввода (неположительное число), например, `omp_set_num_threads(-10)` установит 1 процесс, так что выражение:

```
if (numthreads < 1)
    numthreads = 1;
```

было бы лишним в этом фрагменте кода.

- С помощью соответствующей переменной среды:

```
$ env OMP_NUM_THREADS=3 ./a.out          при запуске
$ export OMP_NUM_THREADS=3               заранее
```

По умолчанию количество потоков равно количеству логических ядер процессора (а с технологией Hyper-Threading, которой обладают все современные процессоры это количество ядер умноженное на два), что

делает распараллеливание в автоматическом режиме ещё более простым, оставляя только директиву `parallel`. Однако тогда программа загрузит процессор на 100%, что не всегда удобно для пользователя – всё остальное начнёт подтормаживать, т.к. все ядра окажутся занятыми на свой максимум. Чтобы программист мог без лишних проблем оптимизировать загрузку, существует функция, которая возвращает количество логических ядер: `omp_get_num_procs()`. С помощью неё можно гибко настраивать загрузку, например так:

```
numthreads = (omp_get_num_procs()/3) + 1;
//для 2 физ. ядер – 2, для 4 – 3, для 8 – 6, и т.д.
```

Либо вообще составить таблицу соответствия количества ядер и количества процессов до, например, 64.

Способы использования

Посмотрим, как можно использовать распараллеливание с помощью OpenMP на практике.

Из предыдущего примера было видно, что нумерация процессов (нитей) идёт с нуля и до `(numthreads - 1)`, следовательно, самый простой способ распределить выполнение задачи между процессами в общем случае, это использовать такую схему:

```
#pragma omp parallel private(pid)
  {
    pid = omp_get_thread_num();
    if (pid == 0) {
        //часть задачи 1
    } else if (pid == 1) {
        //часть задачи 2
    } //...
  }
```

Распараллеливание циклов

Однако на практике чаще всего возникает необходимость распараллеливать циклы, с тем, чтобы каждый процесс выполнял какую-то часть итераций. Упомянутая директива `for` (и её полный Fortran-аналог `do`) проделывают такую работу автоматически. Рассмотрим программу, которая перемножает две матрицы:

```
1 #include <stdio.h>
2 #include <omp.h>
```

```
3
4  #define N 4096
5
6  int main() {
7      double a[N][N], b[N][N], c[N][N];
8      int i, j, k;
9      for (i = 0; i < N; i++)
10         for (j = 0; j < N; j++) //инициализация матриц:
11             a[i][j] = b[i][j] = i + j;
12 #pragma omp parallel for shared(a, b, c) private(i, j, k)
13     for (i = 0; i < N; i++)
14         for (j = 0; j < N; j++) {
15             c[i][j] = 0;
16             for (k = 0; k < N; k++)
17                 c[i][j] += a[i][k] * a[k][j];
18         }
19 }
```

В этом коде итерации внешнего цикла перемножения матриц (по `i`), автоматически распределятся между процессами (количество процессов – по умолчанию).

На этом же примере разберём уже фигурировавшую директиву `private` и её противоположность – `shared`. Первая заставляет перечисленные в её скобках переменные копироваться в каждый процесс, а вторая указывает на то, что в её списке переменные являются общими для всех, т.е. все процессы работают с одной и той же копией переменных. Эти директивы (есть ещё различные их вариации) называются директивами управления данными. Почему в приведённом коде индексные переменные `i`, `j`, `k` не могут быть общими для всех? Потому что в таком случае есть возможность т.н. «гонки данных» (data race), т.е. конфликта, когда два разных процесса одновременно меняют или читают значение переменной, причём каждый по-своему; либо когда один процесс ссылается на неактуальное («испорченное» другим процессом) для него значение переменной.

Допустим, 0-ой процесс взял свою первую итерацию и изменил `i` на `0`. Затем он обнулил `c[0][0]`, а затем взялся за цикл по `k`. Но через мгновение 1-му процессу достаётся, допустим, 5-ая итерация, и он ставит `i` значение `5`. Следовательно, свои первые 5 итераций (0, 1, 2, 3, 4) 0-ой процесс уже может выполнить с не соответствующей для него `i = 5`, т.к. такое значение она чуть позже приняла в другом процессе. Фактически, поведение этого куска кода с `shared(i, j, k)` для 2-ух и более процессов не предсказуемо и скорее всего неверно. Очень важно, что даже в

случае, когда один процесс меняет значение переменной, а другой читает – уже возникает гонка данных и как следствие некорректный результат или малая эффективность работы кода. В случаях, когда все процессы только читают значения, гонка данных может отразиться только на скорости выполнения кода.

По умолчанию, все переменные по вхождению в параллельную область становятся `shared`, а счётчики циклов и переменные, порождённые внутри параллельной области автоматически делаются `private`. Однако никогда не будет лишним внимательно проследить за распараллеливаемым алгоритмом и самим принять эти решения.

Важно также помнить, что при использовании `private` не определено начальное значение указанных в нём переменных, т.е. переменные как бы объявляются заново. Чтобы в параллельной области использовать то значение, которое было до входа в ней существует специальная директива `firstprivate`. Кроме того, иногда дополнительное удобство привносит директива `lastprivate`: каждая из переменных, заключённая в её скобки на выходе станет такой, какой она была на последнем витке цикла в распараллеленной области.

Эффективность

Этот пример с перемножением матриц[1], отлично показывает как краткость (для работы OMP добавлена только одна строчка), так и эффективность этого инструмента.

Замерим время выполнения перемножения с помощью включённой в OpenMP функции `omp_get_wtime()` (тип `double`, возвращает время в секундах, пройденное с некоторого момента):

```
double t1 = omp_get_wtime();
/* перемножение (строчки 12 - 18) */
printf("Прошло %lf секунд\n", omp_get_wtime() - t1);
```

Программа, запущенная на домашнем компьютере (Intel i7 860) даёт результаты, пропорциональные тем, что указаны в книге Антонова (там на узле суперкомпьютера СКИФ МГУ «ЧЕБЫШЁВ» при 1 процессе выполнение занимает 165 секунд, а при 8 – 40):

Нитей	Время, сек	Загрузка CPU, %

[1] А.С. Антонов, «Параллельное программирование с использованием технологии OpenMP».

1	2500	13
2	900	25
4	600	50
8	500	100

Использование класса `Thread` в Java даёт вполне сопоставимые результаты (от 2900 до 600 секунд), однако сложность написания такого алгоритма будет несравненно выше.

Дополнения к директиве for

Часто бывает так, что итерации цикла не равны по сложности выполнения, тогда в то время, как один процесс быстро расправится с выданными ему итерациями, другой может ещё долго работать со своими. Для устранения этой проблемы директива `for` имеет расширение `schedule`; общий вид записи:

```
#pragma omp for schedule(type, chunk)
```

Рассмотрим самые ключевые моменты использования. Основные два типа (`type`) – это `static` и `dynamic`, `chunk` – означает «порция».

Первый тип говорит о том, что итерации распределяются равномерно, второй – в зависимости от скорости работы процесса.

Например, у нас есть 3 процесса и 20 итераций, распределение описано как `schedule(static, 7)`. Тогда 0-ой процесс получит первые 7 итераций, 1-ый – с 8-ой по 14-ую, и 3-ий оставшиеся 6. `dynamic` – распределение по принципу «кто раньше освободится, тот получит следующую порцию». Для предыдущего расклада разницы никакой не будет, но если итераций будет 22 или более, то такая раздача может ускорить выполнение цикла.

Если второй аргумент не задан, напр., `schedule(dynamic)`, то он автоматически становится равным единице.

Существует ещё тип распределения `guided` – это `dynamic`, в котором сначала `chunk` берётся большим, чем указан во втором аргументе, а потом постепенно уменьшается до заданной в `chunk` величины или до ещё немного меньшего значения; а также типы `auto` и `runtime`. Первый указывает на то, что тип распределения выбирается компилятором или

системой, а второй – во время выполнения исходя из установленного значения переменной среды OMP_SCHEDULE. При использовании двух последних типов указывать chunk не следует.

Нахождение минимума/максимума

Как говорилось вначале, OMP характерен тезис «сначала пишем, потом распараллеливаем». В этом параграфе мы рассмотрим случаи, когда распараллеливание «в лоб», как это было сделано с перемножением матриц невозможно.

Напомним ход решения такой распространённой и тривиальной задачи, как нахождение минимального или максимального элемента. Сначала полагается, что искомый элемент – самый первый. Затем перебираются все остальные элементы и если рассматриваемый элемент «бьёт рекорд», то минимум или максимум принимает его значение. На первый взгляд можно просто распараллелить цикл перебора элементов, но сделать это так, как делалось выше нельзя, поскольку минимальный элемент не может быть объявлен ни как private, ни как shared. Если он будет private, то каждый процесс найдёт свой минимум. Директива lastprivate не спасёт программиста в такой ситуации, в любом случае каждый процесс найдёт минимум среди выданных ему элементов. Если минимум будет объявлен как shared, то неизбежно начнётся гонка данных, что приведёт к неверному и непредсказуемому ответу.

Выход из такой ситуации достаточно прост. Заводится столько переменных под минимум, сколько всего процессов. Наиболее универсальный и простой способ – создать массив, вида:

```
mins[num_threads];
```

При поиске минимума каждый процесс будет работать со своим элементов shared массива mins, а после выхода из параллельной области останется только сравнить значения mins[i]. Ниже приведён пример программы, которая ищет минимум из целочисленного массива при любом (в том числе 1) числе процессов.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4 #include <omp.h>
5
6 int main() {
```

```
 7      int th_num, num_threads;
 8      int *mins, minimum;
 9      const int N = 100;
10      int numbers[N];
11
12      srand(time(NULL));
13      for (int i = 0; i < N; i++) {
14          numbers[i] = rand() % 100;
15          printf("%d\t", numbers[i]);
16          if ((i + 1) % 10 == 0) printf("\n");
17      }
18
19      printf("\nКоличество нитей: ");
20      scanf ("%d", &num_threads);
21      omp_set_num_threads(num_threads);
22      mins = (int *) malloc(sizeof(int) * num_threads);
23
24  #pragma omp parallel private(th_num) shared(mins, numbers)
25      {
26          th_num = omp_get_thread_num();
27          mins[th_num] = numbers[0];
28  #pragma omp for schedule(static, 1)
29          for (int i = 1; i < N; i++) {
30              if (numbers[i] < mins[th_num])
31                  mins[th_num] = numbers[i];
32          }
33          printf("Процесс #%d нашёл минимум: %d\n",
34                  th_num, mins[th_num]);
35      } // конец параллельной области
36      minimum = mins[0];
37      for (int i = 1; i < num_threads; i++)
38          if (mins[i] < minimum)
39              minimum = mins[i];
40      printf("Ответ: %d\n", minimum);
41  }
```

Параллельное вычисление рекуррентных последовательностей

Бóльшую сложность представляет из себя другой распространённый тип задач – вычисление последовательности элементов, в которой каждый следующий элемент зависит от предыдущего. Например, вычисление факториала без распараллеливания выглядит очень просто:

```
fact = 1;
for (i = 2; i < n; i++)
    fact = fact * i;
```

Поставить над этим `for` директиву распараллеливания никак нельзя, ведь тогда какому-то процессу обязательно придётся вычислять итерацию, предшественница которой ещё не была вычислена.

Решение проблемы очень простое. Пусть один процесс вычисляет произведения чётных членов, а другой – нечётных (для двух процессов). По выходу из параллельной области останется перемножить промежуточные результаты. Для 3 и более нитей просто увеличим шаг.

Задача становится ещё более лёгкой с использованием директивы `reduction`, которая производит выбранную операцию над всеми копиями какой-то одной переменной сразу перед выходом из параллельной области с тем, чтобы в ней же оказался результат действия. Для простоты понимания приведём пример программы, которая считает факториал числа:

```
 1 #include <stdio.h>
 2 #include <omp.h>
 3
 4 int main() {
 5     int num_threads;
 6     int factorial, n;
 7
 8     printf("Введите n: ");
 9     scanf ("%d", &n);
10     printf("Количество нитей: ");
11     scanf ("%d", &num_threads);
12
13     omp_set_num_threads(num_threads);
14
15     factorial = 1;
16 #pragma omp parallel
17 #pragma omp for reduction(* : factorial)
18     for (int i = 2; i <= n; i++)
19         factorial *= i;
20     printf("%d\n", factorial);
21 }
```

Отметим сразу, что для того, чтобы распараллелить эту программу необходимо добавить в обычный линейный код всего две строчки. Однако без `reduction` алгоритм был бы несколько более сложным и стал бы похожим на предыдущую задачу. Каждый процесс работал бы с своим элементом массива, а затем, после выхода из параллельной области, маленький цикл перемножал бы все получившиеся элементы.

Однако попробуем рассмотреть более затруднительные случаи; в качестве примера решим задачу со следующим условием:

Задан массив *A[N]*, необходимо вычислить *S[N]* по формуле:

$$S[i] = \frac{\prod_{j=1}^{i} a[j]}{\sum_{j=1}^{i} a[j]}$$

Вычислять массив S прямо по формуле чересчур затратно и неправильно – с каждым следующим элементом *S[i]* будет всё больше и больше вычислений, которые совершенно не оправданны, т.к. числители и знаменатели, образующие *S[i]* рекуррентно связаны. Линейное программное решение выглядит следующим образом[2]:

```
Sn = A(1); Sd = A(1)
S(1) = 1
do i = 2, n
    Sn = Sn * A(i)
    Sd = Sd + A(i)
    S(i) = Sn / Sd
enddo
```

Очевидно, что вычисления суммы и произведения не зависят друг от друга, так что сделаем параллельный вариант следующим образом:

```
    nums(1) = A(1); denoms(1) = A(1)
    call omp_set_num_threads(2)
!$omp parallel private(pid) shared(nums, denoms, A)
    pid = omp_get_thread_num()
    if (pid == 0) then
        do i = 2, n
            nums(i) = nums(i - 1) * A(i)
        enddo
    else
        do i = 2, n
            denoms(i) = denoms(i - 1) + A(i)
        enddo
    endif
!$omp end parallel

!$omp parallel num_threads(8) shared(denoms, nums, S, n)
!$omp do
    do i = 1, n
        S(i) = nums(i)/denoms(i)
    enddo
!$omp end parallel
```

Нам пришлось завести массивы `nums(n)` и `denoms(n)`, что увеличивает потребление памяти вдвое по сравнению с предыдущим вариантом, но такая программа покажет 50-ти процентный прирост производительности по сравнению с предыдущей. И всё же она использует

[2] Выберем язык Fortran для решения этой задачи.

только 2 процесса в первой части вычислений. Попробуем переделать её таким образом, чтобы задействовать больше ресурсов процессора.

В зависимости от желаемого количества процессов код и алгоритм будут выглядеть по-разному. В качестве примера возьмём 4 процесса. Сначала мы будем вычислять двойки произведений ($tn_1 = A_1 * A_2$, $tn_2 = A_2 * A_3$, ...) и, аналогично, двойки сумм (td).

Откроем параллельную область на 4 процесса со следующими свойствами:

```
!$omp parallel private(pid) shared(A, td, tn, nums, denoms)
```

И сразу же вычислим двойки:

```
!$omp do schedule(dynamic, 100)
    do i = 1, n - 1
        tn(i) = A(i) * A(i + 1)
        td(i) = A(i) + A(i + 1)
    enddo
```

После этого используем их, чтобы вычислить числители и знаменатели через одного в разных нитях:

```
    pid = omp_get_thread_num()
    if (pid == 0) then
        nums(1) = A(1)
        do i = 3, n, 2
            nums(i) = nums(i - 2) * tn(i - 1)
        enddo
    else if (pid == 1) then
        denoms(1) = A(1)
        do i = 3, n, 2
            denoms(i) = denoms(i - 2) + td(i - 1)
        enddo
    else if (pid == 2) then
        nums(2) = tn(1)
        do i = 4, n, 2
            nums(i) = nums(i - 2) * tn(i - 1)
        enddo
    else if (pid == 3) then
        denoms(2) = td(1)
        do i = 4, n, 2
            denoms(i) = denoms(i - 2) + td(i - 1)
        enddo
    endif
```

Но поскольку код выше использует массивы tn и td, вычисление которых происходило до этого в этой же параллельной области, то нам

необходимо синхронизировать все процессы к началу его выполнения, иначе часть необходимых `tn/td` может оказаться не вычисленной к началу этих четырёх циклов. Вместо того, чтобы закрывать область и открывать такую же новую, будем использовать директиву синхронизации (вставим эту строчку сразу после вычисления `tn` и `td`):

```
!$omp barrier
```

Освободившиеся процессы, дойдя до этой строки не пойдут дальше, а остановятся и будут ждать до тех пор, пока все процессы не дойдут до «барьера». Теперь осталось только вычислить `S`. Сделаем это точно так же, как и в предыдущем варианте, закрыв эту область и открыв новую с максимальным для используемой машины количеством процессов.

Такой способ прибавляет ещё 30% производительности по сравнению с предыдущим. Таким образом получаем двукратное ускорение относительно нераспараллеленного варианта.

Заключение

В заключение можно сказать[3], что рассмотренный выше пример является лишь одним из огромного множества случаев трудно решаемых математических задач и алгоритмов, которые требуется распараллелить для эффективного использования суперкомпьютеров и почти всех современных ПК.

В промышленных проектах OMP иногда используется вместе с другими технологиями в одной программе – часто с MPI. Так, связь между разными узлами может осуществляться через передачу сообщений (MPI), а распараллеливание на самом узле реализовываться с помощью OpenMP.

Литература и др. источники

1. А.С. Антонов, «Параллельное программирование с использованием технологии OpenMP», – издательство Московского Университета, 2009.
2. http://parallel.ru – Информационно-аналитический центр.
3. http://en.wikipedia.org/ – иллюстрация распараллеливания.
4. http://www.ferra.ru/ – Сергей Озеров, Алекс Карабуто, «Двухъядерные процессоры Intel и AMD: теория, часть 1», 16.06.2005.

[3] К реферату прилагается несколько примеров и презентация.

5. http://openmp.ru/ – dmitry, «Обзор способов параллельного программирования», 10.03.2008.