Государственное образовательное учереждение
высшего профессионального образования
Уфимский государственный авиационный университет

кафедра математики

Отчёт по лабораторной работе № 1

«Параллельное вычисление суммы числового ряда»

Вариант 6

Параллельное программирование

Выполнил: Ибрагимов Р. Р.,
студент группы ПМИ-34
Проверил: Юлдашев А. В.,
Преподаватель кафедры ВВТиС

Уфа, УГАТУ, 19 ноября 2013 г.
LaTeX

Содержание

1 Цель работы

Для многопроцессорных и многоядерных вычислительных систем с общей и распределённой памятью на примере задачи параллельного вычисления суммы числового ряда научиться программно реализовывать простейшие параллельные вычислительные алгоритмы и проводить анализ их эффективности.

2 Задание на лабораторную работу

1. Написать последовательную версию программы вычисления суммы ряда на языке Си. Произвести анализ времени её выполнения при использовании компиляторов различных производителей и различных ключей оптимизации под операционными системами Windows и Linux.

2. Выполнитть три расчёта с различными значениями N, таким образом, чтобы время работы порграммы с использованием компилятора intel составляло примерно 30, 60 и 90 секунд. Полученные результаты занести в таблицу

3. Создать параллельную версию написанной программы путём автоматического распараллеливания средствами компилятора корпорации intel. Результаты замеров времени работы занести в таблицу.

4. Выполнить распараллеливание последовательной программы путём включения в её текст директив интерфейса OpenMP.

5. Вычислить ускорение и эффективность распаралеливания

6. Написать параллельную программу с использованием базовых функций MPI.

2.1 Задание варианта

$$\sum_{n=1}^{N} \frac{\sin(\frac{1}{n^2})}{(5n-1)^2}$$

3 Выполнение работы

3.1 Написание и отладка программы

Сперва напишем последовательную программу, скомпилируем её различными компиляторами, а затем будем её ускорять и распараллеливать как стандартными средствами компиляторов, так и путём переписывания для использования различных технологий параллелизации. Все исходные коды приведены далее.

Все программы проверяются на одной и той же машине с CPU Intel Pentium B980 2 Cores 2.4 GHz, под управлением операционной системы Ubuntu 13.10 x64 Desktop.

Версии компиляторов и т.д.

```
gcc (Ubuntu/Linaro 4.8.1-10ubuntu8) 4.8.1
icc (ICC) 14.0.0 20130728
```

3.1.1 Команды запуска

GNU Compiler Collection (GCC)

Последовательная версия:

```
$ gcc -std="c99¨ -Wall -pedantic -o main onethread/main.c -lm && ./main N
```

Примечание. Здесь и далее N – количество элементов ряда, которые необходимо просуммировать.

Intel C++ Compiler (ICC)

```
$ icc -std="c99¨ -Wall -pedantic -o main onethread/main.c -lm && ./main N
```

3.1.2 Использование ключей компилятора

Используем ключи компиляторов, с целью выяснить, как различные опции и настройки влияют на производительность кода.

Для всех компиляторов будут проверены ключи -O1, -O2, -O3 выполняющие оптимизации размера исполняемого файла, оптимизации вызова функций (стандартные оптимизации) и интенсивные оптимизации циклов (развёртку) соответственно.

При этом для компилятора ICC будет также проверен ключ -fast, являющийся комбинацией ключей -O3 -ipo -static -xHOST -no-prec-div, при этом флаг -xHOST означает оптимизацию для того процессора, на котором запущен компилятор.

3.2 Вычислительный кластер УГАТУ

Coming soon... :)

4 Исходный код

4.1 Последовательная версия / встроенная параллелизация

Листинг 1. (onethread/main.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

double f(long long int n){
 return sin(1/(n*n))/((5 * n - 1)*(5*n - 1));
}

double sum(long long int N){
 double SUM = 0;
 #pragma loop count min(16)
 for(long long int n = 0; n < N; ++n){
  SUM += f(n+1);
 }
 return SUM;
}

int main(int argc, char *argv[]){
 long long int N = argc > 1 ? atol(argv[1]) : 1000000;

 printf("sum %lld = ", N);
 clock_t time = clock();
 double SUM;
 SUM = sum(N);
 printf("%lf\n", SUM);
 time = clock() - time;
 printf("evaluation time: %lf\n", (double)time / CLOCKS_PER_SEC);
 return 0;
}
```

4.2 OpenMP

Листинг 2. (openMP/main.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

double f(long long int n){
 return sin(1/pow(n, 2))/pow(5 * n - 1, 2);
}

double sum(long long int N){
 double SUM = 0;
 #pragma omp parallel for
 for(long long int n = 0; n < N; ++n){
  SUM += f(n+1);
 }
 return SUM;
}

int main(int argc, char *argv[]){
 long long int N = argc > 1 ? atol(argv[1]) : 10000;

 printf("sum %lld = ", N);
 clock_t time = clock();
 double SUM;
 SUM = sum(N);
 printf("%lf\n", SUM);
 time = clock() - time;
 printf("evaluation time: %lf\n", (double)time/CLOCKS_PER_SEC);
 return 0;
}
```

4.3 MPI

Листинг 3. (mpi/main.c)

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>

double f(long long int index){ // расчёт значения члена ряда с номером index
 return sin(1 / pow(index, 2)) / pow(5 * index - 1, 2);
}

double sum(long long int start, long long int end){ // выполняет поиск частичных сумм начиная с индекса start и до end
 double s = 0;
 for(long int i = start; i < end; ++i){
  s += f(i);
 }
 return s;
}

int main(int argc, char *argv[]){
 long long int N; // размерность ряда
 double s; // частичная сумма ряда
 int comm_size, comm_rank; // количество процессоров и номер процессора
 double global_start_time, global_end_time, global_work_time; // время работы

 N = argc > 1 ? atoll(argv[1]) : 10000; // если параметр N нельзя получить из коммандной строки, используем значение по умолчанию

 MPI_Init(&argc, &argv); // распараллеливание начинается отсюда
 MPI_Comm_rank(MPI_COMM_WORLD, &comm_rank); // каждый процесс узнает о количистве процессов всего
 MPI_Comm_size(MPI_COMM_WORLD, &comm_size); // каждый процесс узнаёт о собственном номере

 if(comm_rank == 0) global_start_time = MPI_Wtime(); // главный процесс запоминает время начала расчёта

 int partials = N / comm_size;
 long long int m = N % comm_size;
 long long int start = comm_rank * partials + 1;
 long long int end = (comm_rank + 1) * partials + 1;
 if(m > 0){
  if(comm_rank < m){
   start += comm_rank;
   end += comm_rank + 1;
```

```
40    }else{
41      start += m;
42      end += m;
43    }
44   }
45
46   double start_time, end_time; // локальные времена для времени расчёта
       частичных сумм
47   start_time = MPI_Wtime(); // запоминаем время начала расчёта
48   s = sum(start, end); // расчёт частичной суммы
49   end_time = MPI_Wtime(); // запоминаем время конца расчёта
50   double work_time = end_time - start_time; // получаем время конца расчёта
51   MPI_Barrier(MPI_COMM_WORLD); // ждём все процессы
52
53   if(comm_rank != 0){ // если процесс не главный
54    MPI_Send(&s, 1, MPI_DOUBLE, 0, comm_size * comm_rank,
      MPI_COMM_WORLD); // посылаем главному процессу
       результат
55   }
56
57   if(comm_rank == 0){ // если процесс главный
58    double partial_sum;
59    double global_sum = s; // глобальная сумма пока это частичная сумма,
       посчитанная главным процессом
60    MPI_Status status;
61    for(int i = 1; i < comm_size; ++i){ // принимаем все частичные суммы от
       других процессов
62     MPI_Recv(&partial_sum, 1, MPI_DOUBLE, i, comm_size * i,
      MPI_COMM_WORLD, &status);
63     global_sum += partial_sum; // и доавляем к глобальной сумме
64    }
65    global_end_time = MPI_Wtime(); // глобальное время конца расчёта
66    global_work_time = global_end_time - global_start_time;
67   }
68
69   fprintf(stderr, "proc %d of %d (%lf sec):\tsum from %lld to
       %lld:\t%1.8lf\n", comm_rank, comm_size, work_time, start, end-1, s);
       // вывод своего результата каждым
       процессом
70
71   MPI_Barrier(MPI_COMM_WORLD); // ждём все процессы
72   if(comm_rank == 0){
73    fprintf(stderr, "global work time: %lf s; global sum of %lld elements
       = %lf\n", global_work_time, N, s); // вывод результата и времени работы
       всей программы
74   }
75   MPI_Finalize(); // конец параллелизации
76   return 0; // конец программы
77  }
```

4.4 Анализ результатов

4.4.1 Последовательная программа

Компилятор	Ключи	N	Время работы, сек	Результат
gcc	нет	1400000000	143.140000	0.052592
gcc	нет	2400000000	245.260000	0.052592
gcc	нет	3400000000	347.570000	0.052592
gcc	-O0	1400000000	143.110000	0.052592
gcc	-O0	2400000000	245.270000	0.052592
gcc	-O0	3400000000	347.650000	0.052592
gcc	-O1	1400000000	126.310000	0.052592
gcc	-O1	2400000000	216.570000	0.052592
gcc	-O1	3400000000	306.820000	0.052592
gcc	-O2	1400000000	126.040000	0.052592
gcc	-O2	2400000000	216.200000	0.052592
gcc	-O2	3400000000	306.180000	0.052592
gcc	-O3	1400000000	126.220000	0.052592
gcc	-O3	2400000000	217.200000	0.052592
gcc	-O3	3400000000	308.320000	0.052592
icc	нет	1400000000	35.290000	0.052592
icc	нет	2400000000	60.450000	0.052592
icc	нет	3400000000	85.650000	0.052592
icc	-O0	1400000000	39.920000	0.052592
icc	-O0	2400000000	68.270000	0.052592
icc	-O0	3400000000	96.500000	0.052592
icc	-O1	1400000000	31.820000	0.052592
icc	-O1	2400000000	54.500000	0.052592
icc	-O1	3400000000	77.230000	0.052592
icc	-O2	1400000000	35.290000	0.052592
icc	-O2	2400000000	60.800000	0.052592
icc	-O2	3400000000	85.690000	0.052592
icc	-O3	1400000000	35.440000	0.052592
icc	-O3	2400000000	60.490000	0.052592
icc	-O3	3400000000	85.650000	0.052592
icc	-fast	1400000000	34.710000	0.052592
icc	-fast	2400000000	59.490000	0.052592
icc	-fast	3400000000	84.300000	0.052592

Тут мы видим, что даже в самом быстром случае и с меньшим количеством элементов программа, скомпилированная компилятором GCC всё-равно медленнее самой медленной программы с большим количеством элементов, скомпилированной компилятором ICC.

Также видно, что GCC по умолчанию не оптимизирует программу, а ICC как минимум выставляет флаг -O2.

В моём случае самой быстрой оказалась программа, собранная ICC с флагом -O1, выдающим минимальный размер исполняемого файла.

При этом использование ключа -fast не дало существенного ускорения кода, но сузило размеры семейства машин, на которых можно запустить расчёт с помощью этой программы.

5 Выводы

~~Компилятор от Intel шикарен~~

Из данной лабораторной работы я сделал вывод, что компиляция программы под конкретное железо, на котором она будет работать, играет существенную роль в скорости её выполнения, в данном случае, код, сгенерированный компилятором производителя моего процессора (intel) выполнялся на порядок быстрее кода, сгенерироавнного GCC для всех процессоров. Хотя дальнейшие оптимизации «под конкретную систему» уже не дают существенного прироста скорости выполнения.

Также, увидел, что параллелизация выполнения независимого кода позволяет существенно ускорить выполнение программы и, при этом, более-менее равномерно загрузить все системные вычислительные ресурсы, большая часть икоторых бы простаивала в случае выполнения последовательной программы.

В рамках этой работы я познакомился с способами ускорения работы программ, используя различные оптимизации, которые выполняет компилятор, встроенными средствами параллелизации, и другими технологиями распараллеливания кода, такими как OpenMP и MPI.

отчёт подготовлен в LaTeX
Ибрагимов Ринат, УГАТУ ОНФ ПМИ-34, Уфа, 19 ноября 2013 г.