УДК 004.021

М.В. Гончарова, А.В. Касицкий, П.И. Бидюк

АНАЛИЗ ПРОИЗВОДИТЕЛЬНОСТИ ЭВМ ПРИ МНОГОПОТОЧНОЙ РЕАЛИЗАЦИИ ВЫЧИСЛЕНИЙ

Аннотация. Исследована задача повышения производительности приложений для параллельных ЭВМ путем многопоточной реализации вычислений. Для классических вычислительных задач построены и программно реализованы последовательный и параллельный алгоритмы, выполнен их сравнительный анализ и тестирование на различных архитектурах ЭВМ.

Ключевые слова: параллельные ЭВМ, производительность компьютеров, многопоточные приложения.

ВВЕДЕНИЕ

Ключевой характеристикой работы компьютеров принято считать его производительность. Наиболее перспективными направлениями увеличения производительности является: увеличение скорости работы электрических схем и внедрение параллелизма обработки данных. Последнее стало особо актуальным с распространением многоядерных компьютеров, но рост аппаратного параллелизма архитектуры должен сопровождаться совершенствованием программного обеспечения, иначе никакого выигрыша в производительности получено не будет. Важно не только создавать новые многопоточные приложения, но и оптимизировать под многоядерные процессоры старое программное обеспечение.

ПОСТАНОВКА ЗАДАЧИ

На основании принципов разработки многопоточных приложений необходимо построить и программно реализовать последовательные и параллельные алгоритмы решения таких классических, широко используемых задач, как вычисление частных сумм гармонического ряда и сортировка массива. Также следует проанализировать эффективность разработанных приложений, протестировав их на различных архитектурах вычислительных систем.

ВЫЧИСЛЕНИЕ ЧАСТНЫХ СУММ ГАРМОНИЧЕСКОГО РЯДА

В качестве простого примера рассмотрим задачу вычисления частных сумм гармонического ряда:

$$S = \sum_{k=1}^{N} \frac{1}{k}. \quad (1)$$

Последовательный алгоритм вычисления частичной суммы гармонического ряда из N элементов может быть реализован на языке C++ следующим образом:

```
double Sum1(int N)
   {//One thread solution
   double res=0;
   for(int i=1;i<=N;i++)
      //Main summing cycle
      res+=1./i;//Add-up
   return res;}
```

Очевидно, что можно реализовать и параллельный алгоритм суммирования. Разобьем сумму (1) на две: $S = S_1 + S_2$, где $S_1 = \sum_{k=1}^{N/2} \frac{1}{k}$, $S_2 = \sum_{k=N/2+1}^{N} \frac{1}{k}$. Эти суммы независимы и могут вычисляться параллельно в отдельных потоках (threads). Реализация этого алгоритма при помощи стандартных средств ОС Windows:

```
struct Tparam{
   //Structure with all parametrs we need to pass
   //to our thread function
   int L,R;
   //Left and Right limits of summing range
   double res;
   //The result thred function will return
   Tparam(int L,int R):L(L),R(R),res(0){}
   //Simple constructor
   };

static unsigned long __stdcall ThreadFunc(void * params){
   //Thread function
   Tparam p=*((Tparam*)params);
   //Convert parameters to the "readable" view
   for(int i=p.L;i<=p.R;i++)
      //Main summing cycle
      p.res+=1./i;//Add-up
   ((Tparam*)params)->res=p.res;
   //Return the sum value
   return 0;
   }

double Sum2(int N){
   //Two thred solution
   HANDLE h[2];
   //Two handles for two threads
```

 ISSN 1562-9945

```
Tparam
a1(1,N/2),
//First thread parametr
//[1 .. N/2] range
a2(N/2+1,N);
//Second thread parametr
//[N/2+1 .. N] range
h[0]=CreateThread(NULL,0,ThreadFunc,(void*)&a1,0,NULL);
//Sum [1 .. N/2]
h[1]=CreateThread(NULL,0,ThreadFunc,(void*)&a2,0,NULL);
//Sum [N/2+1 .. N]
WaitForMultipleObjects(2,h,true,INFINITE);
//Wait for both threads to finish
return a1.res+a2.res;
//Sum[1 .. N] = Sum[1 .. N/2] + Sum[N/2+1 .. N]
}
```

Сравним скорость выполнения параллельного и последовательного алгоритма путем вызова обеих функций Sum1 и Sum2 с одинаковым параметром (например, пусть N=1000000000); при этом измеряется время, необходимое для вычислений. Для этого определим класс __timer, в котором будет использоваться функция clock(), прототип которой содержится в заголовочном файле <ctime>, она возвращает процессорное время.

```
class __timer{
   //Just timer class. Used for time mesurment
   time_t t0,t1;
   //Variebles to store start and stop times
   public:
   void Start(){
      //Set start time
      t0=clock();
      }
   void Stop(){
      //Set stop time
      t1=clock();
      }
   double Time(){
      //Calculate the difference in seconds, and return it
      return 1.*(t1-t0)/CLK_TCK;
      }
};
```

Таким образом, ускорение параллельного алгоритма, по сравнению с последовательным, можно провести на основе анализа результатов работы функции HarmonicSum. В ней выполняется суммирование при помощи обоих алгоритмов, при этом замеряется время, необходимое для выполнения каждого из них.

```
void HarmonicSum(int N=1000000000){
   double ans;
   cout<<"HarmonicSum"<<endl;
   //One Thread
   cout<<"One Thread Algorithm:"<<endl;
   Timer.Start();
   ans=Sum1(N);
   cout<<"Result = "<<ans<<endl;
   Timer.Stop();
   cout<<"Time was "<<Timer.Time()<<endl;
   //Two Threads
   cout<<"Two Threads Algorithm:"<<endl;
   Timer.Start();
   ans=Sum2(N);
   cout<<"Result = "<<ans<<endl;
   Timer.Stop();
   cout<<"Time was "<<Timer.Time()<<endl;
   cout<<endl;
   }
```

СОРТИРОВКА МАССИВА

Для решения многих прикладных задач важной является проблема сортировки данных. На языке C++ функцией sort(first, last) Стандартной библиотеки шаблонов (STL) реализован алгоритм сортировки IntroSort. Эта функция сортирует диапазон [first...last) в порядке неубывания. Не будем вдаваться в суть алгоритма сортировки, для данного исследования это не принципиально.

Допустим, что дан массив A, который состоит из N произвольных целых чисел. Необходимо отсортировать его по неубыванию. Последовательный вариант сортировки без труда выполняется упомянутой выше функцией sort: sort(A,A+N).

Попробуем ускорить процедуру сортировки при использовании нескольких ядер. Для этого разделим исходный неупорядоченный массив на две части так, чтобы максимальный элемент первой части не превышал минимального элемента второй части (по аналогии с алгоритмом сортировки Хоара). Отсортировав каждую часть независимо, получим искомый массив.

Оптимальное быстродействие метода может быть достигнуто при разбиении массива на блоки одинакового размера. Для этого воспользуемся стандартным алгоритмом библиотеки STL nth_element(first, nth, last). Время, требуемое для работы этого алгоритма, линейно зависит от размера массива, поэтому его использование не должно привести к большим затратам времени.

 ISSN 1562-9945

После операции nth_element элемент диапазона [first...last) в позиции, определенной параметром nth, является элементом, который был бы в этой позиции, если бы диапазон был полностью отсортирован. Также для любого итератора i в диапазоне [first...nth) и любого итератора j в диапазоне [nth...last) выполняется необходимое нам условие: *i<=*j. Соответственно, выполнив nth_element(A,A+N/2,A+N), получим требуемое разбиение массива. Сортировка двух частей массива выполняется так, как и для последовательного алгоритма, с помощью функции sort.

Реализация на языке C++:

```
struct Tparam{
   //Structure with all parametrs we need to
   //pass to our thread function
   int *L,*R;
   //Left and Right limits of the sorting range
   Tparam(int *L,int *R):L(L),R(R){}
   //Simple constructor
   };

void Fill(int*A,int N,int seed){
   //Fill the array with "random" numbers
   srand(seed);
   //We use the same seed for both algorithms
   for(int i=0;i<N;i++)
      A[i]=rand();
   }

static unsigned long __stdcall ThreadFunc(void * params){
   Tparam p=*((Tparam*)params);
   //Convert parameters to the "readable" view
   sort( p.L , p.R );
   //Sort specified range with standart sort() function
   return 0;
   }

void sort2(int* A,int *ApN){
   int N=int(ApN-A);
   //N is the nuumber of elements to sort
   //We use standart method to divide the array
   nth_element(A,A+N/2,A+N);
   HANDLE h[2];
   //Two handles for two threads
   Tparam a1(A+0,A+N/2),
   //First thread parametr - [0 .. N/2) range
   a2(A+N/2,A+N);
   //Second thread parametr - [N/2 ..  N) range
   h[0]=CreateThread(NULL,0,ThreadFunc,(void*)&a1,0,NULL);
   //Sort [0 .. N/2)
   h[1]=CreateThread(NULL,0,ThreadFunc,(void*)&a2,0,NULL);
```

```
   //Sort [N/2 ..  N)
   WaitForMultipleObjects(2,h,true,INFINITE);
   //Wait for both threads to finish
   }

int* A;
//The array we'll sort

void Sorting(int N=100000000){
   A=new int[N];
   //Allocate the memory
   int seed=clock();
   //Define common seed
   cout<<"Sorting"<<endl;
   //One Thread
   cout<<"One Thread Algorithm:"<<endl;
   Fill(A,N,seed);
   Timer.Start();
   sort(A,A+N);
   Timer.Stop();
   cout<<"Time was "<<Timer.Time()<<endl;
   //Two Threads
   cout<<"Two Threads Algorithm:"<<endl;
   Fill(A,N,seed);
   Timer.Start();
   sort2(A,A+N);
   Timer.Stop();
   cout<<"Time was "<<Timer.Time()<<endl;
   cout<<endl;
   }
```

ТЕСТИРОВАНИЕ

Тестирующая программа написана на языке C++, откомпилирована в Microsoft Visual Studio 2005 в режиме Release. Тесты выполнены на компьютерах под управлением ОС Windows XP с .Net Framework 2.0. Все компьютеры были оборудованы как минимум 512Мб оперативной памяти (максимальное использование памяти программой значительно меньше). Результаты тестирования приведены в таблицах 1 и 2.

Положение компьютеров в таблице не означает, что какой-то процессор лучше, а какой-то хуже. Таблицы отображают лишь то, как процессоры показали себя на конкретных задачах.

Для тестирования программы произведен подсчет суммы первых 1000000000 членов гармонического ряда. В колонке 1 Thread отображено время работы однопоточной реализации алгоритма, в колонке 2 Threads — двухпоточной. Результаты упорядочены по времени работы двухпоточной реализации.

 ISSN 1562-9945

Таблица 1

Вычисление частичных сумм гармонического ряда

CPU_Name	CPU Freq	L2 Cash	Cores#	Mem_Freq	1 Thread	2 Threads
Intel® Core™ 2 Duo E8400	3,00GHz	1x 6MB	2	1000MHz	6,72	3,41
Intel® Pentium® DC E5200	2,50Ghz	1x 2MB	2	800 MHz	8,25	4,17
AMD Turion™ 64x2	1,86GHz	1x 512kB	2	532MHz	9,88	5,03
Intel® Pentium® D	2,80GHz	2x 2MB	2	800MHz	15,03	7,55
Intel® Pentium® D	2,67GHz	2x 1MB	2	800MHz	16,59	8,08
Intel® Pentium® DC T2390	1,86GHz	1x 1MB	2	532MHz	16,75	8,38
AMD Athlon™ XP 3000+	2,17GHz	1x 512kB	1	400MHz	8,42	8,47
AMD Athlon™ 64 3200+	2,01GHz	1x 512kB	1	400MHz	8,70	8,56
AMD Sempron™ 3000+	1,80GHz	1x 512kB	1	400MHz	10,03	9,94
Intel® Core™ 2 Duo T5200	1,60GHz	1x 2MB	2	532MHz	20,28	10,16
Intel® Celeron® D	3,06GHz	1x 512kB	1	532MHz	13,50	13,44
Intel® Pentium® 4	3,20GHz	1x 2MB	1(2HT)	664MHz	13,20	14,59
Intel® Celeron® M 440	1,86GHz	1x 1MB	1	532MHz	17,61	18,02
Intel® Celeron® M	1,70GHz	1x 1MB	1	400MHz	18,67	19,13
Intel® Celeron®	2,66GHz	1x 256kB	1	266MHz	20,02	27,88

Для всех двуядерных процессоров время работы двухпоточной реализации оказалось практически в два раза меньшим, чем однопоточной. Для одноядерных процессоров время двухпоточной реализации близко ко времени однопоточной. При этом, двухпоточная слегка выигрывает на более новых процессорах и проигрывает на старых (чем старше процессор, тем больше проигрыш). Это связано с тем, что производители все больше оптимизируют свои процессоры для выполнения многопотоковых приложений.

Таблица 2

Сортировка массива

CPU_Name	CPU Freq	L2 Cash	Cores#	Mem_Freq	1 Thread	2 Threads
Intel® Core™ 2 Duo E8400	3,00GHz	1x 6MB	2	1000MHz	7,16	4,47
Intel® Pentium® DC E5200	2,50Ghz	1x 2MB	2	800 MHz	8,84	5,45
Intel® Pentium® DC T2390	1,86GHz	1x 1MB	2	532MHz	11,39	7,64
Intel® Pentium® D	2,80GHz	2x 2MB	2	800MHz	13,63	8,19
AMD Turion™ 64x2	1,86GHz	1x 512kB	2	532MHz	14,08	8,31
Intel® Core™ 2 Duo T5200	1,60GHz	1x 2MB	2	532MHz	13,92	8,91
Intel® Pentium® 4	3,20GHz	1x 2MB	1(2HT)	664MHz	12,38	8,97
Intel® Pentium® D	2,67GHz	2x 1MB	2	800MHz	15,78	10,42
Intel® Celeron® M 440	1,86GHz	1x 1MB	1	532MHz	12,06	12,23
AMD Athlon™ 64 3200+	2,01GHz	1x 512kB	1	400MHz	12,09	12,52
Intel® Celeron® D	3,06GHz	1x 512kB	1	532MHz	12,72	12,92
Intel® Celeron® M	1,70GHz	1x 1MB	1	400MHz	13,25	13,75
AMD Athlon™ XP 3000+	2,17GHz	1x 512kB	1	400MHz	16,89	16,75
Intel® Celeron®	2,66GHz	1x 256kB	1	266MHz	51,19	49,91
AMD Sempron™ 3000+	1,80GHz	1x 512kB	1	400MHz	81,44	83,13

Для тестирования быстродействия вычислений также произведена сортировка массива из 100000000 32-битных целых чисел, сгене-

рированных случайным образом. В колонке 1 Thread отображено время работы однопоточной реализации алгоритма, в колонке 2 Threads — двухпоточной. Результаты упорядочены по времени работы двухпоточной реализации. Для всех двуядерных процессоров время работы двухпоточной реализации примерно в 1,6 раза меньше чем однопоточной. Коэффициент 2 не достигается, потому что процедура nth_element() выполняется однопоточно и выполняется одновременное обращении к памяти. Необходимо обратить внимание на то, что одноядерный Pentium 4 с технологией HT показал ускорение в 1,38 раза, несмотря на то, что он имеет только одно физическое ядро. Для одноядерных процессоров (без HT) время двухпоточной реализации близко ко времени однопоточной (небольшие отклонения вызваны в первую очередь тем, что используются разные алгоритмы).

ВЫВОДЫ

Разработаны многопоточные приложения для реализации алгоритма вычисления гармонической суммы ряда и сортировки массива. Созданные программы протестированы на различных архитектурах ЭВМ. На основе собранной информации сделаны выводы о характеристиках исследуемых компьютеров и реальному ускорению производительности вычислений:

Большая частота не означает большую производительность, как считалось еще несколько лет назад. Это можно заметить, сравнив результаты, полученные для Intel® Core™ 2 Duo E8400 3,00GHz и Intel® Celeron® D 3,06GHz (табл. 1, 2).

Черезвычайно важными являются размер и принцип организации кэш-памяти.

Процессоры AMD быстрей работают с арифметикой, чем Intel того же класса. Intel в свою очередь лучше использует механизм кеширования.

Intel® Pentium® 4 с технологией Hyper-Threading характеризуется ускорением для двухпоточных приложений, несмотря на то, что он имеет только одно физическое ядро.

Важную роль играет быстродействие памяти. По этой причине Intel® Celeron® с памятью 266MHz показал значительно худшие результаты по сравнению со своими конкурентами.

На двуядерных процессорах, распараллеливание арифметических операций дает выигрыш почти в два раза. В то же время, распарал-

 ISSN 1562-9945

лелив «активные» обращения к памяти, получим ускорение в среднем в полтора раза, даже при двухканальной архитектуре памяти.

Прирост производительности в целом достаточно велик; при этом, чем новее процессор, тем этот прирост больше.

ЛИТЕРАТУРА

1. Джиллеспи М. Масштабирование программных архитектур для многоядерных вычислительных систем будущего. // доступно по адресу http://softwarecommunity-rus.intel.com/articles/rus/1276.htm.
2. Чабуксвар Р. Максимизация энергосбережения на мобильных платформах // доступно по адресу http://softwarecommunity-rus.intel.com/articles/rus/2702.htm.
3. Мэттсон Т. Основы многопоточного программирования // доступно по адресу http://softwarecommunity-rus.intel.com/articles/rus/1201.htm.