Изучаем инфраструктуру OpenMP на примере компилятора GCC

Арпан Сен
технический директор
Synapti Computer Aided Design Pvt Ltd

08.07.2013

Инфраструктура Open Multiprocessing (OpenMP) представляет собой чрезвычайно мощную спецификацию, позволяющую использовать преимущества многопроцессорных систем в приложениях C, C++ и Fortran. В этой статье я расскажу о том, как использовать возможности OpenMP в коде C++, и покажу на примерах, как начать работать с OpenMP.

Инфраструктура OpenMP позволяет эффективно реализовывать технологии параллельного программирования на C, C++ и Fortran. Компилятор GNU Compiler Collection (GCC) версии 4.2 поддерживает спецификацию OpenMP 2.5, а GCC версии 4.4 – самую последнюю спецификацию OpenMP 3. Другие компиляторы, включая Microsoft® Visual Studio, также поддерживают OpenMP. Эта статья научит вас использовать прагмы компилятора OpenMP; также она содержит информацию о некоторых API-интерфейсах OpenMP и раскрывает некоторые приемы параллельных вычислений с использованием OpenMP. Во всех примерах этой статьи используется компилятор GCC 4.2.

Начало работы

Огромным достоинством OpenMP является отсутствие необходимости в дополнительных действиях, за исключением стандартной инсталляции компилятора GCC. Компиляция OpenMP-приложений должна выполняться с опцией `-fopenmp`.

Создаем первое OpenMP-приложение

Начнем с написания простого приложения **Hello, World!**, содержащего дополнительную прагму. Код этого приложения представлен в листинге 1.

Листинг 1. Программа "Hello World", написанная с использованием OpenMP

```
#include <iostream>
int main()
{
  #pragma omp parallel
  {
    std::cout << "Hello World!\n";
  }
}
```

После компиляции и запуска этого кода с помощью g++ вы увидите на экране надпись **Hello, World!**. Теперь перекомпилируем код с опцией `-fopenmp`. Результат работы программы представлен в листинге 2.

Листинг 2. Компиляция и запуск кода с использованием опции `-fopenmp`

```
tintin$ g++ test1.cpp -fopenmp
tintin$ ./a.out
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
```

Что же произошло? Когда вы используете опцию компилятора `-fopenmp`, в дело вступает директива `#pragma omp parallel`. В процессе компиляции внутренние механизмы GCC создают столько параллельных потоков, сколько могут выполняться при условии оптимальной загрузки системы (в зависимости от конфигураций аппаратного обеспечения и операционной системы), при этом в каждом создаваемом потоке выполняется код, заключенный в блоке после прагмы. Такое поведение называется *неявным распараллеливанием*, а ядро OpenMP состоит из набора мощных прагм, избавляющих вас от написания множества типовых фрагментов кода (для интереса вы можете сравнить приведенный кода с реализацией этой же задачи с помощью POSIX-потоков [pthreads]). Я использую компьютер с процессором Intel® Core i7 с 4 физическими ядрами по 2 логических ядра в каждом, что объясняет результаты, полученные в листинге 2 (8 потоков = 8 логических ядер).

Далее рассмотрим прагмы parallel более подробно.

Возможности OpenMP parallel

Количеством потоков можно легко управлять с помощью прагмы с аргументом `num_threads`. Ниже представлен код из листинга 1 с заданным количеством потоков (5 потоков):

Листинг 3. Управление количеством потоков с помощью `num_threads`

```
#include <iostream>
int main()
{
  #pragma omp parallel num_threads(5)
  {
    std::cout << "Hello World!\n";
  }
}
```

Вместо аргумента `num_threads` можно воспользоваться альтернативным способом для задания количества потоков выполнения кода. Здесь мы подошли к рассмотрению первого API-интерфейса OpenMP под названием `omp_set_num_threads`. Эта функция определяется

в файле заголовка omp.h. Для выполнения кода из листинга 4 не требуется использовать какие-либо дополнительные библиотеки, – просто используйте опцию `-fopenmp`.

Листинг 4. Точное управление потоками с помощью `omp_set_num_threads`

```
#include <omp.h>
#include <iostream>
int main()
{
  omp_set_num_threads(5);
  #pragma omp parallel
  {
    std::cout << "Hello World!\n";
  }
}
```

Наконец, для управления работой OpenMP можно использовать внешние переменные среды. Можно исправить код из листинга 2 и просто напечатать фразу **Hello World!** шесть раз, задав для переменной `OMP_NUM_THREADS` значение 6, как показано в листинге 5.

Листинг 5. Переменные среды для настройки OpenMP

```
tintin$ export OMP_NUM_THREADS=6
tintin$ ./a.out
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
```

Мы рассмотрели три стороны OpenMP: прагмы компилятора, API-интерфейсы времени выполнения и переменные среды. Что произойдет, если использовать переменные среды вместе с API-интерфейсами времени выполнения? API-интерфейсы имеют более высокий приоритет.

Практический пример

В OpenMP используются технологии неявного распараллеливания, и для передачи инструкций компилятору можно использовать прагмы, явные функции и переменные среды. Давайте рассмотрим пример, наглядно демонстрирующий пользу от применения OpenMP. Посмотрите на код, представленный в листинге 6.

Листинг 6. Последовательная обработка в цикле `for`

```
int main( )
{
int a[1000000], b[1000000];
// ... код инициализации массивов a и b;
int c[1000000];
for (int i = 0; i < 1000000; ++i)
  c[i] = a[i] * b[i] + a[i-1] * b[i+1];
// ... выполняем некоторые действия с массивом c
 }
```

Очевидно, что цикл `for` можно распараллелить и обрабатывать сразу несколькими ядрами процессора, поскольку вычисление значения любого элемента `c[k]` никак не зависит от остальных элементов массива `c`. В листинге 7 показано, как можно это сделать с помощью OpenMP.

Листинг 7. Параллельная обработка в цикле `for` с помощью прагмы `parallel for`

```
int main( )
{
int a[1000000], b[1000000];
// ... код для инициализации массивов a и b;
int c[1000000];
#pragma omp parallel for
for (int i = 0; i < 1000000; ++i)
  c[i] = a[i] * b[i] + a[i-1] * b[i+1];
// ... выполняем некоторые действия с массивом c
 }
```

Прагма `parallel for` помогает распределить рабочую нагрузку цикла `for` между несколькими потоками, каждый из которых может обрабатываться отдельным ядром процессора; таким образом общее время вычислений существенно снижается. Это подтверждается в листинге 8.

Листинг 8. Пример с использованием API-функции omp_get_wtime

```
#include <omp.h>
#include <math.h>
#include <time.h>
#include <iostream>

int main(int argc, char *argv[]) {
    int i, nthreads;
    clock_t clock_timer;
    double wall_timer;
    double c[1000000];
    for (nthreads = 1; nthreads <=8; ++nthreads) {
        clock_timer = clock();
        wall_timer = omp_get_wtime();
        #pragma omp parallel for private(i) num_threads(nthreads)
        for (i = 0; i < 1000000; i++)
          c[i] = sqrt(i * 4 + i * 2 + i);
        std::cout << "threads: " << nthreads <<  " time on clock(): " <<
            (double) (clock() - clock_timer) / CLOCKS_PER_SEC
           << " time on wall: " <<  omp_get_wtime() - wall_timer << "\n";
    }
}
```

В листинге 8 мы измеряем время выполнения внутреннего цикла `for`, увеличивая при этом количество потоков. API-функция `omp_get_wtime` возвращает затраченное фактическое время (в секундах), прошедшее с начала заданной точки отсчета. Таким образом, значение `omp_get_wtime() - wall_timer` возвращает фактическое время выполнения цикла `for`. Системный вызов `clock()` используется для оценки времени, затраченного центральным процессором на выполнение всей программы, т. е. прежде чем получить итоговый результат, мы суммируем все эти временные интервалы с учетом потоков. На моем компьютере с процессором Intel Core i7 я получил результаты, приведенные в листинге 9.

Листинг 9. Статистика выполнения внутреннего цикла `for`

```
threads: 1 time on clock(): 0.015229 time on wall: 0.0152249
threads: 2 time on clock(): 0.014221 time on wall: 0.00618792
threads: 3 time on clock(): 0.014541 time on wall: 0.00444412
threads: 4 time on clock(): 0.014666 time on wall: 0.00440478
threads: 5 time on clock(): 0.01594 time on wall: 0.00359988
threads: 6 time on clock(): 0.015069 time on wall: 0.00303698
threads: 7 time on clock(): 0.016365 time on wall: 0.00258303
threads: 8 time on clock(): 0.01678 time on wall: 0.00237703
```

Хотя время центрального процессора (time on clock) во всех случаях получилось примерно одинаковым (как и должно быть, не принимая во внимание дополнительное время, затраченное на создание потоков и контекстного переключателя), интересующее нас фактическое время (time on wall) постоянно уменьшалось при увеличении количества потоков, которые предположительно выполнялись параллельно отдельными процессорными ядрами. Итак, сделаем последнее замечание относительно синтаксиса прагмы: `#pragma parallel for private(i)` означает, что переменная цикла `i` рассматривается как локальная память потока; каждый поток содержит свою копию этой переменной. Локальная переменная потока не инициализируется.

Критические участки кода в OpenMP

Разумеется, вы понимаете, что нельзя полностью доверить OpenMP автоматически обрабатывать критические участки кода, не так ли? Конечно, вам не придется явным образом создавать взаимные исключения (мьютексы), тем не менее критические участки необходимо указывать. Синтаксис приводится в следующем примере:

```
#pragma omp critical (optional section name)
{
// 2 потока не могут выполнять этот блок кода одновременно
}
```

Код, следующий после директивы `pragma omp critical`, может выполняться в заданный момент времени только в одном потоке. Кроме того, имя раздела `optional section name` является глобальным идентификатором, и критические участки с одинаковым идентификатором не могут обрабатываться одновременно двумя потоками. Рассмотрим код в листинге 10.

Листинг 10. Несколько критических участков с одинаковыми именами

```
#pragma omp critical (section1)
{
myhashtable.insert("key1", "value1");
}
// ... здесь содержится какой-то другой код
#pragma omp critical (section1)
{
myhashtable.insert("key2", "value2");
}
```

Посмотрев на код в этом листинге, можно предположить, что две операции вставки данных в хэш-таблицу никогда не будут выполняться одновременно, поскольку имена

критических участков совпадают. Это немного отличается от того, к чему вы привыкли, обрабатывая критические участки в pthreads, для которых характерно использование большого количества блокировок (что может приводить к лишним сложностям).

Блокировки и мьютексы в OpenMP

Интересно, что OpenMP содержит свои версии мьютексов (в конце концов, OpenMP – это не только прагмы). Итак, познакомьтесь с типом `omp_lock_t`, определенным в заголовочном файле omp.h. Обычные мьютексные операции в стиле pthread-потоков содержат значение true, даже если имена API-функций одинаковы. Вот пять API-функций, о которых необходимо знать:

- **`omp_init_lock`**: эта API-функция должна использоваться первой при обращении к типу `omp_lock_t` и предназначена для инициализации. Необходимо отметить, что сразу после инициализации блокировка будет находиться в исходном (unset) состоянии.
- **`omp_destroy_lock`**: уничтожает блокировку. В момент вызова этой API-функции блокировка должна находиться в исходном состоянии; это означает, что нельзя вызвать функцию `omp_set_lock`, а затем уничтожить блокировку.
- **`omp_set_lock`**: устанавливает `omp_lock_t`, т. е. активирует мьютекс. Если поток не может установить блокировку, он продолжает ожидать до тех пор, пока такая возможность не появится.
- **`omp_test_lock`**: пытается установить блокировку, если она доступна; возвращает `1` в случае успеха и `0` в случае неудачи. Это функция является *неблокирующей*, т. е. она не заставляет поток ожидать установления блокировки.
- **`omp_unset_lock`**: сбрасывает блокировку в исходное состояние.

В листинге 11 содержится простая реализация устаревшей однопоточной очереди, усовершенствованной для обработки нескольких потоков с использованием блокировок OpenMP. Обратите внимание на то, что этот пример не является лучшим универсальным вариантом для использования и приведен просто для демонстрации возможностей.

Листинг 11. Усовершенствование однопоточной очереди с помощью OpenMP

```
#include <openmp.h>
#include "myqueue.h"

class omp_q : public myqueue<int> {
public:
   typedef myqueue<int> base;
   omp_q( ) {
      omp_init_lock(&lock);
   }
   ~omp_q() {
       omp_destroy_lock(&lock);
   }
   bool push(const int& value) {
      omp_set_lock(&lock);
      bool result = this->base::push(value);
      omp_unset_lock(&lock);
      return result;
   }
```

```
   bool trypush(const int& value)
   {
       bool result = omp_test_lock(&lock);
       if (result) {
          result = result && this->base::push(value);
          omp_unset_lock(&lock);
       }
       return result;
   }
   // likewise for pop
private:
   omp_lock_t lock;
};
```

Вложенные блокировки

Другими типами блокировок в OpenMP являются различные варианты блокировки `omp_nest_lock_t`. Они похожи на `omp_lock_t`, но имеют дополнительное преимущество: такая блокировка может быть включена удерживающим ее потоком несколько раз. Каждый раз, когда вложенная блокировка активируется удерживающим ее потоком с помощью `omp_set_nest_lock`, увеличивается внутренний счетчик блокировок. Блокировка освобождается удерживающим потоком тогда, когда один или несколько вызовов `omp_unset_nest_lock` снижают значение внутреннего счетчика блокировок до `0`. Для работы с `omp_nest_lock_t` используются следующие API-функции:

- **`omp_init_nest_lock(omp_nest_lock_t* )`**: устанавливает внутренний счетчик вложенности в `0`.
- **`omp_destroy_nest_lock(omp_nest_lock_t* )`**: уничтожает блокировку. Вызов этой API-функции для блокировки со значением счетчика, отличного от нуля, приводит к непредсказуемым результатам.
- **`omp_set_nest_lock(omp_nest_lock_t* )`**: аналогична функции `omp_set_lock` за исключением того, что удерживающий поток может вызывать ее несколько раз.
- **`omp_test_nest_lock(omp_nest_lock_t* )`**: является неблокирующей версией API-функции `omp_set_nest_lock`.
- **`omp_unset_nest_lock(omp_nest_lock_t* )`**: освобождает блокировку при достижении внутренним счетчиком блокировок значения `0`. В остальных случаях каждый вызов этой API-функции уменьшает значение счетчика.

Детальный контроль выполнения заданий

Мы уже видели, что блок кода, который следует за директивой `pragma omp parallel`, обрабатывается параллельно всеми потоками. Код внутри этих блоков можно также разделить на категории, которые будут выполняться в указанных потоках. Рассмотрим код в листинге 12.

Листинг 12. Использование прагмы parallel sections

```
int main( )
{
  #pragma omp parallel
  {
```

```
    cout << "Это выполняется во всех потоках\n";
    #pragma omp sections
    {
      #pragma omp section
      {
        cout << "Это выполняется параллельно\n";
      }
      #pragma omp section
      {
        cout << "Последовательный оператор 1\n";
        cout << "Это всегда выполняется после оператора 1\n";
      }
      #pragma omp section
      {
        cout << "Это тоже выполняется параллельно\n";
      }
    }
  }
}
```

Код, предшествующий директиве `pragma omp sections` и следующий сразу за директивой `pragma omp parallel`, обрабатывается параллельно всеми потоками. При помощи директивы `pragma omp sections` код, следующий за ней, разбивается на отдельные подсекции. Каждый блок `pragma omp section` может выполняться отдельным потоком. Тем не менее отдельные операторы внутри блока section всегда выполняются последовательно. В листинге 13 показаны результаты выполнения кода из листинга 12.

Листинг 13. Результаты выполнения кода из листинга 12

```
tintin$ ./a.out
Это выполняется во всех потоках
Это выполняется во всех потоках
Это выполняется во всех потоках
Это выполняется во всех потоках
Это выполняется во всех потоках
Это выполняется во всех потоках
Это выполняется во всех потоках
Это выполняется во всех потоках
Это выполняется параллельно
Последовательный оператор 1
Это тоже выполняется параллельно
Это всегда выполняется после оператора 1
```

В листинге 13 мы снова видим 8 изначально созданных потоков. Для обработки блока `pragma omp sections` достаточно трех потоков из восьми. Внутри второй секции мы указали порядок, в котором выполняются операторы вывода текста. В этом и заключается смысл использования прагмы `sections`. При необходимости можно указывать порядок выполнения блоков кода.

Директивы firstprivate и lastprivate в связке с параллельными циклами

В этой статье я уже показывал, как объявить локальную память потока с помощью директивы `private`. Но как инициализировать локальные переменные потока? Может быть, синхронизировать их со значением переменной главного потока перед дальнейшими действиями? В таких случаях нам пригодится директива `firstprivate`.

Директива `firstprivate`

Используя директиву `firstprivate(переменная)`, можно инициализировать переменную в потоке, присвоив ей любое значение, которое она имела в главном потоке. Рассмотрим код из листинга 14.

Листинг 14. Использование локальной переменной потока, которая не синхронизирована с главным потоком

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

int main()
{
  int idx = 100;
  #pragma omp parallel private(idx)
  {
    printf("В потоке %d idx = %d\n", omp_get_thread_num(), idx);
  }
}
```

Вот что я получил в результате (ваши результаты могут быть другими).

```
В потоке 1 idx = 1
В потоке 5 idx = 1
В потоке 6 idx = 1
В потоке 0 idx = 0
В потоке 4 idx = 1
В потоке 7 idx = 1
В потоке 2 idx = 1
В потоке 3 idx = 1
```

В листинге 15 содержится код с использованием директивы `firstprivate`. Как и ожидалось, вывод результатов показывает, что переменная `idx` имеет значение `100` во всех потоках.

Листинг 15. Использование директивы `firstprivate` для инициализации локальных переменных потоков

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

int main()
{
  int idx = 100;
  #pragma omp parallel firstprivate(idx)
  {
    printf("В потоке %d idx = %d\n", omp_get_thread_num(), idx);
  }
}
```

Также обратите внимание на то, что для доступа к идентификатору потока был использован метод `omp_get_thread_num()`. Этот идентификатор отличается от идентификатора, выводимого командой `top` операционной системы Linux®, и эта схема является лишь способом, позволяющим OpenMP отслеживать количество потоков. Если вы планируете использовать директиву `firstprivate` в коде `C++`, то обратите внимание на другую ее особенность: переменная, используемая директивой `firstprivate`, является копирующим

конструктором для инициализации самой себя из переменной главного потока, поэтому если копирующий конструктор является приватным для вашего класса, это может привести к неприятным последствиям. Теперь перейдем к рассмотрению директивы `lastprivate`, которая во многом является обратной стороной монеты.

Директива `lastprivate`

Вместо того чтобы синхронизировать локальную переменную потока с данными главного потока, мы будем синхронизировать переменную главного потока с данными, которые будут получены в результате выполнения последнего цикла. В листинге 16 выполняется параллельный цикл `for`.

Листинг 16. Параллельный цикл `for` без синхронизации данных с главным потоком

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

int main()
{
  int idx = 100;
  int main_var = 2120;

  #pragma omp parallel for private(idx)
  for (idx = 0; idx < 12; ++idx)
  {
    main_var = idx * idx;
    printf("В потоке %d idx = %d main_var = %d\n",
      omp_get_thread_num(), idx, main_var);
  }
  printf("Возврат в главный поток со значением переменной main_var = %d\n", main_var);
}
```

На моем компьютере с 8 ядрами OpenMP создает для блока `parallel for` шесть потоков. Каждый поток, в свою очередь, насчитывает по две итерации в цикле. Итоговое значение переменной `main_var` зависит от потока, который выполнился последним и, следовательно, от значения переменной `idx` в этом потоке. Другими словами, значение переменной `main_var` не зависит от последнего значения переменной `idx`, но зависит от значения, которое содержала переменная `idx` того потока, который был выполнен в последнюю очередь. Этот пример продемонстрирован в листинге 17.

Листинг 17. Зависимость значения переменной main_var от последнего выполненного потока

```
В потоке 4 idx = 8 main_var = 64
В потоке 2 idx = 4 main_var = 16
В потоке 5 idx = 10 main_var = 100
В потоке 3 idx = 6 main_var = 36
В потоке 0 idx = 0 main_var = 0
В потоке 1 idx = 2 main_var = 4
В потоке 4 idx = 9 main_var = 81
В потоке 2 idx = 5 main_var = 25
В потоке 5 idx = 11 main_var = 121
В потоке 3 idx = 7 main_var = 49
В потоке 0 idx = 1 main_var = 1
В потоке 1 idx = 3 main_var = 9
Возврат в главный поток со значением переменной main_var = 9
```

Запустите код из листинга 17 несколько раз, чтобы убедиться в том, что значение переменной `main_var` в главном потоке всегда зависит от значения переменной `idx` в последнем выполненном потоке. А что делать, если необходимо синхронизировать значение переменной главного потока с итоговым значением переменной `idx` в цикле? Здесь на помощь приходит директива `lastprivate`, работа которой продемонстрирована в листинге 18. Как и в предыдущем случае, запустите код из листинга 18 несколько раз, и вы увидите, что итоговое значение переменной `main_var` в главном потоке равно `121` (т .е. значению переменной `idx` в последней итерации цикла).

Листинг 18. Синхронизация с помощью директивы `lastprivate`

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

int main()
{
  int idx = 100;
  int main_var = 2120;

  #pragma omp parallel for private(idx) lastprivate(main_var)
  for (idx = 0; idx < 12; ++idx)
  {
    main_var = idx * idx;
    printf("В потоке %d idx = %d main_var = %d\n",
      omp_get_thread_num(), idx, main_var);
  }
  printf("Возврат в главный поток со значением переменной main_var = %d\n", main_var);
}
```

В листинге 19 представлены результаты выполнения кода из листинга 18.

Листинг 19. Результаты выполнения кода из листинга 18 (обратите внимание на то, что значение `main_var always` всегда равно 121 в главном потоке)

```
В потоке 3 idx = 6 main_var = 36
В потоке 2 idx = 4 main_var = 16
В потоке 1 idx = 2 main_var = 4
В потоке 4 idx = 8 main_var = 64
В потоке 5 idx = 10 main_var = 100
В потоке 3 idx = 7 main_var = 49
В потоке 0 idx = 0 main_var = 0
В потоке 2 idx = 5 main_var = 25
В потоке 1 idx = 3 main_var = 9
В потоке 4 idx = 9 main_var = 81
В потоке 5 idx = 11 main_var = 121
В потоке 0 idx = 1 main_var = 1
Возврат в главный поток со значением переменной main_var = 121
```

И последнее замечание: для поддержки оператора `lastprivate` объектом `C++` требуется, чтобы соответствующий класс содержал доступный публичный метод `operator=`.

Сортировка слиянием в OpenMP

Рассмотрим реальный пример, в котором OpenMP сокращает время выполнения задачи. Возьмем не слишком оптимизированную версию процедуры `merge sort`, достаточную для

демонстрации преимуществ от использования OpenMP. Этот пример приведен в листинге 20.

Листинг 20. Сортировка слиянием в OpenMP

```
#include <omp.h>
#include <vector>
#include <iostream>
using namespace std;

vector<long> merge(const vector<long>& left, const vector<long>& right)
{
    vector<long> result;
    unsigned left_it = 0, right_it = 0;

    while(left_it < left.size() && right_it < right.size())
    {
        if(left[left_it] < right[right_it])
        {
            result.push_back(left[left_it]);
            left_it++;
        }
        else
        {
            result.push_back(right[right_it]);
            right_it++;
        }
    }

    // Занесение оставшихся данных из обоих векторов в результирующий
    while(left_it < left.size())
    {
        result.push_back(left[left_it]);
        left_it++;
    }

    while(right_it < right.size())
    {
        result.push_back(right[right_it]);
        right_it++;
    }

    return result;
}

vector<long> mergesort(vector<long>& vec, int threads)
{
    // Условие завершения: список полностью отсортирован,
    // если он содержит только один элемент.
    if(vec.size() == 1)
    {
        return vec;
    }

    // Определяем местоположение среднего элемента в векторе
    std::vector<long>::iterator middle = vec.begin() + (vec.size() / 2);

    vector<long> left(vec.begin(), middle);
    vector<long> right(middle, vec.end());

    // Выполнение сортировки слиянием над двумя меньшими векторами

    if (threads > 1)
    {
      #pragma omp parallel sections
      {
```

```
        #pragma omp section
        {
          left = mergesort(left, threads/2);
        }
        #pragma omp section
        {
          right = mergesort(right, threads - threads/2);
        }
      }
    }
    else
    {
      left = mergesort(left, 1);
      right = mergesort(right, 1);
    }

    return merge(left, right);
}

int main()
{
  vector<long> v(1000000);
  for (long i=0; i<1000000; ++i)
    v[i] = (i * i) % 1000000;
  v = mergesort(v, 1);
  for (long i=0; i<1000000; ++i)
    cout << v[i] << "\n";
}
```

При использовании 8 потоков для выполнения процедуры `merge sort` время выполнения сократилось с 3,7 (при использовании одного потока) до 2,1 секунды. Здесь необходимо лишь соблюдать осторожность с количеством потоков. Я начал с 8 потоков, но реальная отдача от их использования может варьироваться в зависимости от конфигурации системы. Если количество потоков не ограничено явным образом, то могут быть созданы сотни, если не тысячи потоков, что с очень высокой вероятностью приведет к снижению производительности системы. Кроме того, при работе с кодом `merge sort` полезно использовать прагму `sections`, которая была рассмотрена ранее.

Заключение

На этом я заканчиваю статью. Мы в достаточной мере рассмотрели прагмы parallel и способы создания потоков, убедились в том, что OpenMP уменьшает время выполнения задач и позволяет выполнять синхронизацию и гибкий контроль, а также рассмотрели практический пример с использованием `merge sort`. Конечно, вам еще предстоит изучить очень многое, и лучше всего поможет в этом Web-сайт проекта OpenMP. Всю дополнительную информацию вы можете найти в разделе Ресурсы.

Об авторе

Арпан Сен

Арпан Сен (Arpan Sen) – ведущий инженер, работающий над разработкой программного обеспечения в области автоматизации электронного проектирования. На протяжении нескольких лет он работал над некоторыми функциями UNIX, в том числе Solaris, SunOS, HP-UX и IRIX, а также Linux и Microsoft Windows. Он проявляет живой интерес к методикам оптимизации производительности программного обеспечения, теории графов и параллельным вычислениям. Арпан является аспирантов в области программных систем.