**通过 GCC 学习 OpenMP 框架**

Open Multiprocessing (OpenMP) 框架是一种功能极为强大的规范，可以帮助您利用 C、C++ 和 Fortran 应用程序中的多个核心带来的好处。本文将介绍如何在 C++ 代码中使用 OpenMP 特性，并提供了几个示例帮助您开始学习 OpenMP。

0https://dw1.s81c.com/developerworks/i/v17/dw-cmts-arrow.png [评论](https://www.ibm.com/developerworks/cn/aix/library/au-aix-openmp-framework/#icomments)

[Arpan Sen](https://www.ibm.com/developerworks/cn/aix/library/au-aix-openmp-framework/#authorN1001F), 独立作家

2012 年 10 月 29 日

* [expand](https://www.ibm.com/developerworks/cn/aix/library/au-aix-openmp-framework/#toggle)**内容**



在 IBM Bluemix 云平台上开发并部署您的下一个应用。

[**开始您的试用**](https://developer.ibm.com/sso/bmregistration?lang=zh_CN&ca=dwchina-_-bluemix-_-au-aix-openmp-framework-_-sidebar)

OpenMP 框架是使用 C、C++ 和 Fortran 进行并发编程的一种强大方法。GNU Compiler Collection (GCC) V4.2 支持 OpenMP 2.5 标准，而 GCC 4.4 支持最新的 OpenMP 3 标准。包括 Microsoft® Visual Studio 在内的其他编译器也支持 OpenMP。在本文中，您可以学习使用 OpenMP 编译指示 (pragma)，寻找对 OpenMP 提供的一些应用程序编程接口 (API) 的支持，并使用一些并行算法对 OpenMP 进行测试。本文将使用 GCC 4.2 作为首选编译器。

**入门**

OpenMP 的一大特点就是您只需完成标准的 GCC 安装即可。支持 OpenMP 的程序必须使用 -fopenmp 选项进行编译。

**您的第一个 OpenMP 程序**

让我们先从一个 **Hello, World!** 打印应用程序开始，它包括一个额外的编译指示。[清单 1](https://www.ibm.com/developerworks/cn/aix/library/au-aix-openmp-framework/#list1) 显示了相关代码。

**清单 1. 使用了 OpenMP 的 Hello World 程序**

#include <iostream>

int main()

{

#pragma omp parallel

{

std::cout << "Hello World!\n";

}

}

在使用 g++ 编译和运行清单 1 中的代码时，控制台中应该会显示一个 **Hello, World!**。现在，使用 -fopenmp 选项重新编译代码。[清单 2](https://www.ibm.com/developerworks/cn/aix/library/au-aix-openmp-framework/#list2) 显示了输出。

同时，运行时需指定环境变量export OMP\_NUM\_THREADS=2

若不指定，可能还会只有一个线程

**清单 2. 使用 -fopenmp 命令编译并运行代码**

tintin$ g++ test1.cpp -fopenmp

tintin$ ./a.out

Hello World!

Hello World!

Hello World!

Hello World!

Hello World!

Hello World!

Hello World!

Hello World!

发生了什么？#pragma omp parallel 仅在您指定了 -fopenmp 编译器选项后才会发挥作用。在编译期间，GCC 会根据硬件和操作系统配置在运行时生成代码，创建尽可能多的线程。每个线程的起始例程为代码块中位于指令之后的代码。这种行为是 *隐式的并行化*，而 OpenMP 本质上由一组功能强大的编译指示组成，帮您省去了编写大量样本文件的工作。（为了进行比较，您需要了解使用 Portable Operating System Interface (POSIX) 线程 [pthreads] 实现您刚才的程序将会怎样）。我使用的计算机运行的是 Intel® Core i7 处理器，有四个物理核心，每个核心有两个逻辑核心，因此清单 2 的输出看上去非常合理（8 个线程 = 8 个逻辑核心）。

接下来，让我们详细了解并行编译指示。

[**回页首**](https://www.ibm.com/developerworks/cn/aix/library/au-aix-openmp-framework/#ibm-pcon)

**使用 OpenMP 实现并行处理的乐趣**

使用编译命令的 num\_threads 参数控制线程的数量非常简单。下面显示了 [清单 1](https://www.ibm.com/developerworks/cn/aix/library/au-aix-openmp-framework/#list1) 中的代码，可用线程的数量被指定为 5（如 [清单 3](https://www.ibm.com/developerworks/cn/aix/library/au-aix-openmp-framework/#list3) 所示）。

**清单 3. 使用 num\_threads 控制线程的数量**

#include <iostream>

int main()

{

#pragma omp parallel num\_threads(5)

{

std::cout << "Hello World!\n";

}

}

这里没有使用 num\_threads 方法，而是使用另一种方法来修改运行代码的线程的数量。这还会将我们带到您将要使用的第一个 OpenMP API：omp\_set\_num\_threads。在 omp.h 头文件中定义该函数。不需要链接到额外的库就可以获得 [清单 4](https://www.ibm.com/developerworks/cn/aix/library/au-aix-openmp-framework/#list4) 中的代码，只需使用 -fopenmp。

**清单 4. 使用 omp\_set\_num\_threads 对线程的创建进行调优**

#include <omp.h>

#include <iostream>

int main()

{

omp\_set\_num\_threads(5);

#pragma omp parallel

{

std::cout << "Hello World!\n";

}

}

最后，OpenMP 还使用了外部环境变量控制其行为。您可以调整 [清单 2](https://www.ibm.com/developerworks/cn/aix/library/au-aix-openmp-framework/#list2) 中的代码，通过将 OMP\_NUM\_THREADS 变量设置为 6，就可以将 **Hello World!** 输出六次。[清单 5](https://www.ibm.com/developerworks/cn/aix/library/au-aix-openmp-framework/#list5) 显示了代码的执行。

**清单 5. 使用环境变量来调整 OpenMP 行为**

tintin$ export OMP\_NUM\_THREADS=6

tintin$ ./a.out

Hello World!

Hello World!

Hello World!

Hello World!

Hello World!

Hello World!

目前，我们已经讨论了 OpenMP 的三个方面：编译指示、运行时 API 和环境变量。同时使用环境变量和运行时 API 会出现什么情况？运行时 API 将获得更高的优先权。

**一个实际示例**

OpenMP 使用隐式并行化技术，您可以使用编译指示、显式函数和环境变量来指导编译器的行为。让我们看一个 OpenMP 可以真正派上用场的示例。请考虑 [清单 6](https://www.ibm.com/developerworks/cn/aix/library/au-aix-openmp-framework/#list6) 中的代码。

**清单 6. 在 for 循环中执行顺序处理**

int main( )

{

int a[1000000], b[1000000];

// ... some initialization code for populating arrays a and b;

int c[1000000];

for (int i = 0; i < 1000000; ++i)

c[i] = a[i] \* b[i] + a[i-1] \* b[i+1];

// ... now do some processing with array c

}

显然，您可以将 for 循环分为几个部分，在不同的核心中运行它们，任何 c[k] 与 c 数组中的其他元素都是独立的。[清单 7](https://www.ibm.com/developerworks/cn/aix/library/au-aix-openmp-framework/#list7) 显示了 OpenMP 如何帮助您实现这一点。

**清单 7. 在 for 循环中使用 parallel for 指令进行并行处理**

int main( )

{

int a[1000000], b[1000000];

// ... some initialization code for populating arrays a and 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];

// ... now do some processing with array c

}

parallel for 编译指示可以帮助您将 for 循环工作负载划分到多个线程中，每个线程都可以在不同的核心上运行，这显著减少了总的计算时间。[清单 8](https://www.ibm.com/developerworks/cn/aix/library/au-aix-openmp-framework/#list8) 演示了这一点。

**清单 8. 理解 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";

}

}

在 [清单 8](https://www.ibm.com/developerworks/cn/aix/library/au-aix-openmp-framework/#list8) 中，可以通过不断增加线程的数量来计算运行内部 for 循环的时间。omp\_get\_wtime API 从一些任意的但是一致的点返回已用去的时间，以秒为单位。因此，omp\_get\_wtime() - wall\_timer 将返回观察到的所用时间并运行 for 循环。clock() 系统调用用于预估整个程序的处理器使用时间，也就是说，将各个特定于线程的处理器使用时间相加，然后报告最终的结果。在我的 Intel Core i7 计算机中，[清单 9](https://www.ibm.com/developerworks/cn/aix/library/au-aix-openmp-framework/#list9) 显示了最终的报告结果。

**清单 9. 运行内部 for 循环的统计数据**

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

尽管处理器时间在所有执行中都是相同的（应该是相同的，除去创建线程和上下文切换的时间），但是所用时间将随着线程数量的增加而逐渐减小，这意味着数据在核心中实现了并行处理。关于指令语法，最后一点需要注意的是：#pragma parallel for private(i) 意味着循环向量 i 将被作为一个线程本地存储进行处理，每个线程有一个该向量的副本。线程本地向量未进行初始化。

[**回页首**](https://www.ibm.com/developerworks/cn/aix/library/au-aix-openmp-framework/#ibm-pcon)

**OpenMP 的临界区段（critical section）**

您没有真正考虑过 OpenMP 是如何自动处理临界区段的，是吗？当然，您不需要显式创建一个互斥现象 (mutex)，但是您仍然需要指定临界区段。下面显示了相关语法：

#pragma omp critical (optional section name)

{

// no 2 threads can execute this code block concurrently

}

pragma omp critical 之后的代码只能由一个线程在给定时间运行。同样，optional section name 是一个全局标识符，在同一时间，两个线程不能使用相同的全局标识符名称运行临界区段。请考虑 [清单 10](https://www.ibm.com/developerworks/cn/aix/library/au-aix-openmp-framework/#list10) 中的代码。

**清单 10. 多个具有相同名称的临界区段**

#pragma omp critical (section1)

{

myhashtable.insert("key1", "value1");

}

// ... other code follows

#pragma omp critical (section1)

{

myhashtable.insert("key2", "value2");

}

在这一代码的基础上，您可以作出一个很安全的假设：永远不会出现两个散列表同时插入的情况，因为临界区段名是相同的。这与您在使用 pthread 时处理临界区段的方式有着明显的不同，后者的特点就是使用（或滥用）锁。

**使用 OpenMP 实现锁和互斥**

有趣的是，OpenMP 提供了自己的互斥（因此，它并不是全部关于编译指示）：omp\_lock\_t，它被定义为 omp.h 头文件的一部分。常见的 pthread 形式的语法操作也适用，甚至 API 名称也是类似的。您需要了解以下 5 个 API：

* **omp\_init\_lock**：此 API 必须是第一个访问 omp\_lock\_t 的 API，并且要使用它来完成初始化。注意，在完成初始化之后，锁被认为处于未设置状态。
* **omp\_destroy\_lock**：此 API 会破坏锁。在调用该 API 时，锁必须处于未设置状态，这意味着您无法调用 omp\_set\_lock 并随后发出调用来破坏这个锁。
* **omp\_set\_lock**：此 API 设置 omp\_lock\_t，也就是说，将会获得互斥。如果一个线程无法设置锁，那么它将继续等待，直到能够执行锁操作。
* **omp\_test\_lock**：此 API 将在锁可用时尝试执行锁操作，并在获得成功后返回 1，否则返回 0。这是一个*非阻塞 API*， 也就是说，该函数不需要线程等待就可以设置锁。
* **omp\_unset\_lock**：此 API 将会释放锁。

[清单 11](https://www.ibm.com/developerworks/cn/aix/library/au-aix-openmp-framework/#list11) 显示了一个遗留的单线程队列，它被扩展为可使用 OpenMP 锁实现多线程处理。请注意，这一行为并不适合所有场景，这里主要用它来进行快速演示。

**清单 11. 使用 OpenMP 扩展一个单线程队列**

#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;

};

**嵌套锁**

OpenMP 提供的其他类型的锁为 omp\_nest\_lock\_t 锁的变体。它们与 omp\_lock\_t 类似，但是有一个额外的优势：已经持有锁的线程可以多次锁定这些锁。每当持有锁的线程使用 omp\_set\_nest\_lock 重新获得嵌套锁时，内部计数器将会加一。当一个或多个对 omp\_unset\_nest\_lock 的调用最终将这个内部锁计数器重置为 0 时，就会释放该锁。下面显示了用于 omp\_nest\_lock\_t 的 API：

* **omp\_init\_nest\_lock(omp\_nest\_lock\_t\* )**：此 API 将内部嵌套计数初始化为 0。
* **omp\_destroy\_nest\_lock(omp\_nest\_lock\_t\* )**：此 API 将破坏锁。使用非零内部嵌套计数对某个锁调用此 API 将会导致出现未定义的行为。
* **omp\_set\_nest\_lock(omp\_nest\_lock\_t\* )**：此 API 类似于 omp\_set\_lock，不同之处是线程可以在已持有锁的情况下多次调用这个函数。
* **omp\_test\_nest\_lock(omp\_nest\_lock\_t\* )**：此 API 是 omp\_set\_nest\_lock 的非阻塞版本。
* **omp\_unset\_nest\_lock(omp\_nest\_lock\_t\* )**：此 API 将在内部计数器为 0 时释放锁。否则，计数器将在每次调用该方法时递减。

[**回页首**](https://www.ibm.com/developerworks/cn/aix/library/au-aix-openmp-framework/#ibm-pcon)

**对任务执行的细粒度控制**

我们已经了解了所有线程以并行的方式运行 pragma omp parallel 之后的代码块。您可以对代码块中的代码进一步分类，然后由选定的线程执行它。请考虑 [清单 12](https://www.ibm.com/developerworks/cn/aix/library/au-aix-openmp-framework/#list12) 中的代码。

**清单 12. 学习使用 parallel sections 编译指示**

int main( )

{

#pragma omp parallel

{

cout << "All threads run this\n";

#pragma omp sections

{

#pragma omp section

{

cout << "This executes in parallel\n";

}

#pragma omp section

{

cout << "Sequential statement 1\n";

cout << "This always executes after statement 1\n";

}

#pragma omp section

{

cout << "This also executes in parallel\n";

}

}

}

}

pragma omp sections 和 pragma omp parallel 之间的代码将由所有线程并行运行。pragma omp sections 之后的代码块通过 pragma omp section 进一步被分为各个子区段。每个 pragma omp section 块将由一个单独的线程执行。但是，区段块中的各个指令始终按顺序运行。[清单 13](https://www.ibm.com/developerworks/cn/aix/library/au-aix-openmp-framework/#list13) 显示了清单 12 中代码的输出。

**清单 13. 运行清单 12 中代码所产生的输出**

tintin$ ./a.out

All threads run this

All threads run this

All threads run this

All threads run this

All threads run this

All threads run this

All threads run this

All threads run this

This executes in parallel

Sequential statement 1

This also executes in parallel

This always executes after statement 1

在清单 13 中，您将再次一开始就创建 8 个线程。对于这 8 个线程，只需使用三个线程执行 pragma omp sections 代码块中的工作。在第二个区段中，您指定了输出语句的运行顺序。这就是使用 sections 编译指示的全部意义。如果需要的话，您可以指定代码块的顺序。

[**回页首**](https://www.ibm.com/developerworks/cn/aix/library/au-aix-openmp-framework/#ibm-pcon)

**理解与并行循环一起使用的 firstprivate 和 lastprivate 指令**

在前文中，我们了解了如何使用 private 声明线程本地存储。那么，您应当如何初始化线程本地变量呢？在运行之前使用主线程中的变量的值同步本地变量？此时，firstprivate 指令就可以发挥作用了。

**firstprivate 指令**

使用 firstprivate(variable)，您可以将线程中的变量初始化为它在主线程中的任意值。请参考 [清单 14](https://www.ibm.com/developerworks/cn/aix/library/au-aix-openmp-framework/#list14) 中的代码。

**清单 14. 使用与主线程不同步的线程本地变量**

#include <stdio.h>

#include <omp.h>

int main()

{

int idx = 100;

#pragma omp parallel private(idx)

{

printf("In thread %d idx = %d\n", omp\_get\_thread\_num(), idx);

}

}

下面是我得到的输出。您的结果可能有所不同。

In thread 1 idx = 1

In thread 5 idx = 1

In thread 6 idx = 1

In thread 0 idx = 0

In thread 4 idx = 1

In thread 7 idx = 1

In thread 2 idx = 1

In thread 3 idx = 1

[清单 15](https://www.ibm.com/developerworks/cn/aix/library/au-aix-openmp-framework/#list15) 显示了带有 firstprivate 指令的代码。和期望的一样，输出在所有线程中将 idx 初始化为 100。

**清单 15. 使用 firstprivate 指令初始化线程本地变量**

#include <stdio.h>

#include <omp.h>

int main()

{

int idx = 100;

#pragma omp parallel firstprivate(idx)

{

printf("In thread %d idx = %d\n", omp\_get\_thread\_num(), idx);

}

}

还要注意的是，您使用了 omp\_get\_thread\_num( ) 方法访问线程的 ID。这与 Linux®top 命令显示的线程 ID 不同，并且这只是 OpenMP 用于跟踪线程数量的一种方式。如果您准备将 firstprivate 用于您的 C++ 代码，那么还要注意，firstprivate 指令使用的变量是一个副本构造函数，用于从主线程的变量初始化自身，因此对您的类使用一个私有的副本构造函数肯定会产生不好的结果。现在让我们了解一下lastprivate 指令，该指令在很多方面与 firstprivate 正好相反。

**lastprivate 指令**

与使用主线程的数据初始化线程本地变量不同，您现在将使用最后一次循环计数生成的数据同步主线程的数据。[清单 16](https://www.ibm.com/developerworks/cn/aix/library/au-aix-openmp-framework/#list16) 中的代码运行了并行的 for 循环。

**清单 16. 使用并行的 for 循环，没有与主线程进行同步**

#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("In thread %d idx = %d main\_var = %d\n",

omp\_get\_thread\_num(), idx, main\_var);

}

printf("Back in main thread with main\_var = %d\n", main\_var);

}

在我的拥有 8 个核心的开发计算机上，OpenMP 为 parallel for 块创建了 6 个线程。每个线程执行两次循环迭代。main\_var 的最终值取决于最后一个运行的线程，即该线程中的 idx 的值。换言之，main\_var 的值并不取决于 idx 的最后一个值，而是任意一个最后运行的线程中的 idx 的值。[清单 17](https://www.ibm.com/developerworks/cn/aix/library/au-aix-openmp-framework/#list17) 的代码解释了这一点。

**清单 17. main\_var 的值取决于最后一次线程运行**

In thread 4 idx = 8 main\_var = 64

In thread 2 idx = 4 main\_var = 16

In thread 5 idx = 10 main\_var = 100

In thread 3 idx = 6 main\_var = 36

In thread 0 idx = 0 main\_var = 0

In thread 1 idx = 2 main\_var = 4

In thread 4 idx = 9 main\_var = 81

In thread 2 idx = 5 main\_var = 25

In thread 5 idx = 11 main\_var = 121

In thread 3 idx = 7 main\_var = 49

In thread 0 idx = 1 main\_var = 1

In thread 1 idx = 3 main\_var = 9

Back in main thread with main\_var = 9

多次运行清单 17 中的代码，确定主线程中的 main\_var 的值始终依赖于最后运行的线程中的 idx 的值。那么如果您希望同步主线程的值与循环中 idx 的最终值，该怎样做呢？这就是 lastprivate 指令发挥其作用的地方，如 [清单 18](https://www.ibm.com/developerworks/cn/aix/library/au-aix-openmp-framework/#list18) 所示。与清单 17 中的代码相似，多次运行清单 18 中的代码会发现主线程中的 main\_var 的最终值为 121（idx 为最终的循环计数器值）。

**清单 18. 使用 lastprivate 指令实现同步**

#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("In thread %d idx = %d main\_var = %d\n",

omp\_get\_thread\_num(), idx, main\_var);

}

printf("Back in main thread with main\_var = %d\n", main\_var);

}

[清单 19](https://www.ibm.com/developerworks/cn/aix/library/au-aix-openmp-framework/#list19) 显示了清单 18 的输出。

**清单 19. 清单 18 的代码输出（请注意，主线程中 main\_var always 值的等于 121）**

In thread 3 idx = 6 main\_var = 36

In thread 2 idx = 4 main\_var = 16

In thread 1 idx = 2 main\_var = 4

In thread 4 idx = 8 main\_var = 64

In thread 5 idx = 10 main\_var = 100

In thread 3 idx = 7 main\_var = 49

In thread 0 idx = 0 main\_var = 0

In thread 2 idx = 5 main\_var = 25

In thread 1 idx = 3 main\_var = 9

In thread 4 idx = 9 main\_var = 81

In thread 5 idx = 11 main\_var = 121

In thread 0 idx = 1 main\_var = 1

Back in main thread with main\_var = 121

最后一个注意事项：要支持对 C++ 对象使用 lastprivate 操作符，则需要相应的类具有公开可用的 operator= 方法。

[**回页首**](https://www.ibm.com/developerworks/cn/aix/library/au-aix-openmp-framework/#ibm-pcon)

**使用 OpenMP 实现 merge sort**

让我们看一个 OpenMP 将会帮助您节省运行时间的真实示例。这并不是一个对 merge sort 进行大量优化后的版本，但是足以显示在代码中使用 OpenMP 的好处。[清单 20](https://www.ibm.com/developerworks/cn/aix/library/au-aix-openmp-framework/#list20) 显示了示例的代码。

**清单 20. 使用 OpenMP 实现 merge sort**

#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++;

}

}

// Push the remaining data from both vectors onto the resultant

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)

{

// Termination condition: List is completely sorted if it

// only contains a single element.

if(vec.size() == 1)

{

return vec;

}

// Determine the location of the middle element in the vector

std::vector<long>::iterator middle = vec.begin() + (vec.size() / 2);

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

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

// Perform a merge sort on the two smaller vectors

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";

}

使用 8 个线程运行 merge sort 使运行时的执行时间变为 2.1 秒，而使用一个线程时，该时间为 3.7 秒。此处您惟一需要注意的是线程的数量。我使用了 8 个线程：具体的数量取决于系统的配置。但是，如果不明确指定的话，那么有可能会创建成千上百个线程，并且系统性能很可能会下降。前面讨论的 sections 编译指示可以很好地优化 merge sort 代码。