Parallel Programming with OpenMP

HIGH SPEED

OpenMP brings the power of multiprocessing to your C, C++, and Fortran programs. **BY WOLFGANG DAUTERMANN**

If you bought a new computer recently, or if you are wading through advertising material because you plan to buy a computer soon, you will be familiar with terms such as "Dual Core" and "Quad Core." A whole new crop of consumer computers includes two- or even four-core CPUs, taking the humble PC into what used to be the domain of high-end servers and workstations. But just because you have a multiprocessor system doesn't mean all the processors are working hard.

In reality, often only one processor is busy. Figure 1 shows the *top* program output for Xaos, a fractal calculation program. The program seems to be using 100 percent of the CPU. But appearances can be deceptive: The computer's actual load is just 60 percent.

Pressing the *1* button lists the CPUs separately. In this mode (Figure 2), you can easily see the load on the individual cores: One CPU is working hard (90 percent load), while the other is twiddling its thumbs (0.3 percent load).

Linux introduced support for multiple processor systems many moons ago, and the distributors now install the multiple CPU–capable **SMP** kernel by default. Linux, therefore, has what it takes to leverage the power of multiple cores. But what about the software?

A program running on the system must be aware of the multiple processor architecture in order to realize the performance benefits. OpenMP is an API specification for "… multi-threaded, shared memory parallelization" [1]. The OpenMP specification defines a set of

GLOSSARY

SMP: Symmetric multi-processor system. All of the machine's CPUs can access the shared main memory – in contrast to cluster systems, in which separate machines exchange data over the wire. OpenMP is suitable for parallel programming on SMP systems.

Thread: One popular definition of thread is a "lightweight process." A Unix process has a separate memory area and various resources are assigned to it – such as environmental variables, network connections, or device access. A thread shares memory and certain other resources with other threads in a process. This reduces the management overhead compared with processes, and facilitates switching between threads. Pressing Shift+H in the *top* tool enables and disables the thread display.

compiler directives, run-time library routines, and environment variables for supporting multi-processor environments.

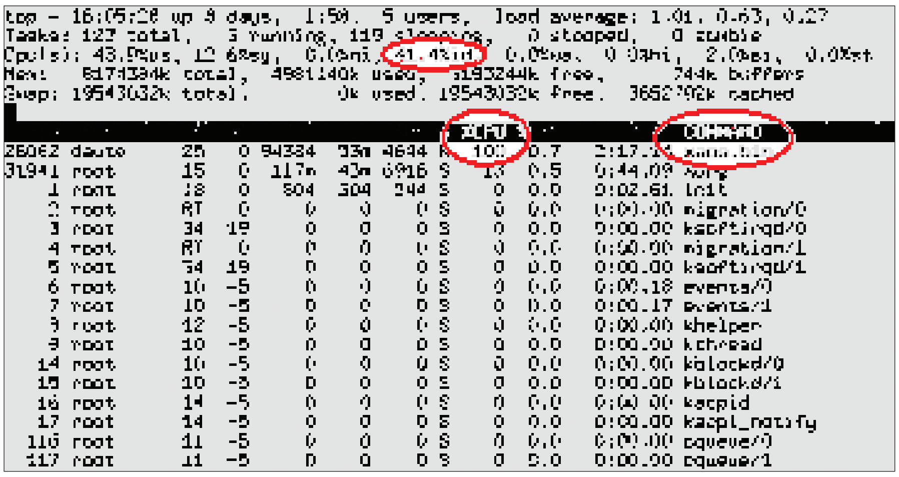
C/C++, and Fortran programmers can use OpenMP to create new multi-processor-ready programs and to convert existing programs to run efficiently in multi-processor environments.

Multi-Tracking

A computer will work its way sequentially – that is, one instruction after another – through programs written in C/C++ or some other programming language. Of course, this technique will only keep one processor core busy. Parallelization lets you make more efficient use of a multi-processor system.

The OpenMP programming interface, which has been under constant development by various hardware and compiler manufacturers since 1997, provides a very simple and portable option for parallelizing programs written in C/C++ and Fortran.

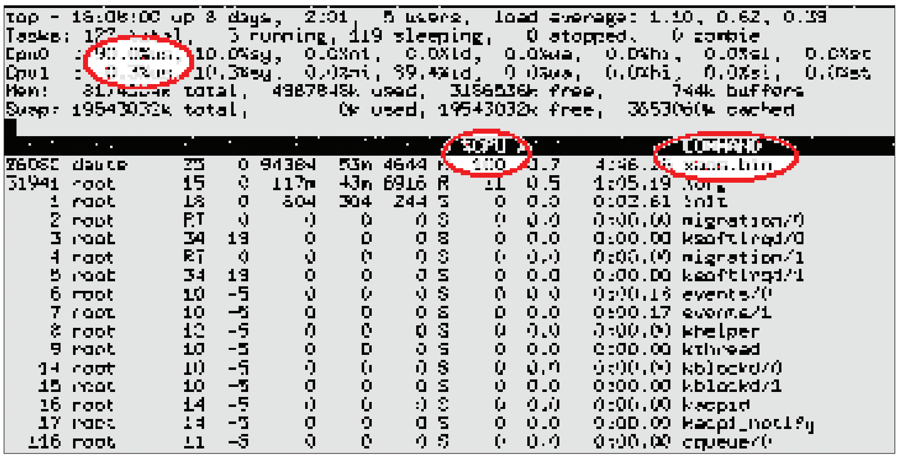
OpenMP can boost the performance of a program significantly, but only if the CPU really has to work hard – and if the task lends itself to parallelization. Such is often the case when working with computationally intensive programs.



```
top - 16:05:28 up 8 days,  1:50,  5 users,  load average: 1.01, 0.63, 0.27
Tasks: 122 total,   3 running, 119 sleeping,   0 stopped,   0 zombie
Cpu(s): 43.9%us, 12.6%sy,  0.0%ni, 41.4%id,  0.0%wa,  0.0%hi,  2.0%si,  0.0%st
Mem:   8174384k total,  4981140k used,  3193244k free,      744k buffers
Swap: 19543032k total,        0k used, 19543032k free,  3652792k cached

                                                   %CPU                  COMMAND
26062 dauto      25   0 94384  93m 4644 R  100  0.7   2:17.14 [illegible]
31941 root       15   0  117m  43m 6916 S   13  0.5   9:44.09 Xorg
    1 root       18   0   804  304  244 S    0  0.0   0:02.61 init
    2 root       RT   0     0    0    0 S    0  0.0   0:00.00 migration/0
    3 root       34  19     0    0    0 S    0  0.0   0:00.00 ksoftirqd/0
    4 root       RT   0     0    0    0 S    0  0.0   0:00.00 migration/1
    5 root       34  19     0    0    0 S    0  0.0   0:00.00 ksoftirqd/1
    6 root       10  -5     0    0    0 S    0  0.0   0:00.18 events/0
    7 root       10  -5     0    0    0 S    0  0.0   0:00.17 events/1
    8 root       12  -5     0    0    0 S    0  0.0   0:00.00 khelper
    9 root       10  -5     0    0    0 S    0  0.0   0:00.00 kthread
   14 root       10  -5     0    0    0 S    0  0.0   0:00.00 kblockd/0
   15 root       10  -5     0    0    0 S    0  0.0   0:00.00 kblockd/1
   16 root       14  -5     0    0    0 S    0  0.0   0:00.00 kacpid
   17 root       14  -5     0    0    0 S    0  0.0   0:00.00 kacpi_notify
  116 root       11  -5     0    0    0 S    0  0.0   0:00.00 cqueue/0
  117 root       11  -5     0    0    0 S    0  0.0   0:00.00 cqueue/1
```

Figure 1: By default, top shows the total load for all CPUs ...



```
top - 16:08:00 up 8 days,  2:01,  5 users,  load average: 1.10, 0.62, 0.39
Tasks: 122 total,   3 running, 119 sleeping,   0 stopped,   0 zombie
Cpu0  : 90.0%us, 10.0%sy,  0.0%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu1  :  0.3%us, 10.3%sy,  0.0%ni, 89.4%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   8174384k total,  4987848k used,  3186536k free,      744k buffers
Swap: 19543032k total,        0k used, 19543032k free,  3653060k cached

                                                   %CPU                  COMMAND
26062 dauto      25   0 94384  53m 4644 R  100  0.7   4:46.15 [illegible].bin
31941 root       15   0  117m  43m 6916 R   11  0.5   1:05.19 Xorg
    1 root       18   0   804  304  244 S    0  0.0   0:02.61 init
    2 root       RT   0     0    0    0 S    0  0.0   0:00.00 migration/0
    3 root       34  19     0    0    0 S    0  0.0   0:00.00 ksoftirqd/0
    4 root       RT   0     0    0    0 S    0  0.0   0:00.00 migration/1
    5 root       34  19     0    0    0 S    0  0.0   0:00.00 ksoftirqd/1
    6 root       10  -5     0    0    0 S    0  0.0   0:00.16 events/0
    7 root       10  -5     0    0    0 S    0  0.0   0:00.17 events/1
    8 root       12  -5     0    0    0 S    0  0.0   0:00.00 khelper
    9 root       10  -5     0    0    0 S    0  0.0   0:00.00 kthread
   14 root       10  -5     0    0    0 S    0  0.0   0:00.00 kblockd/0
   15 root       10  -5     0    0    0 S    0  0.0   0:00.00 kblockd/1
   16 root       14  -5     0    0    0 S    0  0.0   0:00.00 kacpid
   17 root       14  -5     0    0    0 S    0  0.0   0:00.00 kacpi_notify
  116 root       11  -5     0    0    0 S    0  0.0   0:00.00 cqueue/0
```

Figure 2: ... but on request it will give you the load for the individual processor cores.

One, Two, Many

The OpenMP API supplies programmers with a simple option for effectively parallelizing their existing serial programs through the specification of a couple of additional compiler directives, which would look something like the following code snippet:

```
#pragma omp ⮐
name_of_directive [clauses]
```

Compilers that don't support OpenMP, such as older versions of GCC before version 4.2, will just ignore the compiler directives, meaning that the source code can still be complied as serial code:

```
$ gcc -Wall test.c
test.c: In function 'main':
test.c:12: warning: ignoring ⮐
#pragma omp parallel
```

OpenMP-specific code can also be compiled conditionally, with the *#ifdef* directive: OpenMP defines the *_OPENMP* macro for this purpose.

An OpenMP program launches normally as a serial program with one **thread**. One instruction arrives after another. The first OpenMP statement I will introduce creates multiple threads:

Listing 1: Parallel Sections and Loops

Variant 1: Parallel Sections

```
... /* one thread */
#pragma omp parallel /* many
threads */
{
#pragma omp sections
#pragma omp section
...  /* Program section A running
parallel to B and C */
#pragma omp section
...  /* Program section B running
parallel to A and C */
#pragma omp section
... /*  Program section C running
parallel to A and B */
}
... /* one thread */
```

Variant 2: Parallel Loops

```
... /* a thread */
#pragma omp parallel [clauses ...]
#pragma omp for [clauses ...]
for (i=0;i<N;i++) {
    a[i]= i*i; /* parallelized */
    }
... /* one thread */
```

Listing 2: reduction()

```
01 a = 0 ; b = 0 ;
02 #pragma omp parallel for
   private(i) shared(x, y, n)
   reduction(+:a, b)
03 for (i=0; i<n; i++) {
04     a = a + x[i] ;
05     b = b + y[i] ;
06     }
```

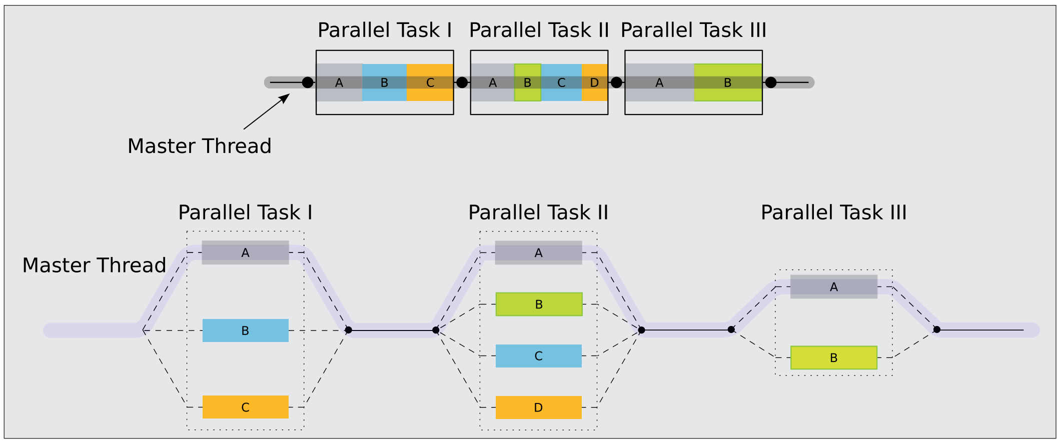


Figure 3: The OpenMP Fork-Join model.

```
... one Thread
#pragma omp parallel
{ ... many threads }
... one thread
```

Figure 3 shows how the program is distributed over multiple threads and then reunited to a single thread.

Divide and Conquer

Now you have created multiple threads, but at the moment, they are all doing the same thing. The idea is that the threads should each handle their share of the workload at the same time. The programming language C has two approaches to this problem. Fortran, a programming language that is popular in scientific research, has a third approach: "parallel work sharing."

The first variant, parallel sections, runs program sections (blocks of program code that are not interdependent) that support parallel execution, parallel to one another.

So that this can happen, *#pragma omp parallel* defines multiple threads. This means that you can run multiple, independent program blocks in individual threads with no restrictions on the number of parallel *sections* (Listing 1, Variant 1: Parallel Sections). Also, you can combine the two compiler directives, *parallel* and *sections*, to form a single directive, as in *#pragma omp parallel sections*.

The second variant, parallel *for()* loops, parallelizes for loops, which is especially useful in the case of computationally intensive mathematical programs (Listing 1, Variant 2: Parallel Loops).

Figure 4 shows how this works. Again you can combine *#pragma omp parallel* and *#pragma omp for* to *#pragma omp parallel for*.

Scope

In shared memory programming multiple CPUs can access the same variables. This makes the program more efficient and saves copying. In some cases, each thread needs its own copy of the variables – such as the loop variables in parallel *for()* loops.

Clauses specified in OpenMP directives (see the descriptions Table 1) define the properties of these variables. You can append clauses to the OpenMP *#pragma*, for example:

```
#pragma omp parallel for shared(x, y) private(z)
```

Errors in *shared()*/*private()* variable declarations are some of the most common causes of errors in parallelized programming.

Reduction

Now you now know how to create threads and distribute the workload over multiple threads. However, how can you get all the threads to work on a collated result – for example, to total the values in an array? *reduction()* (Listing 2) handles this.

The compiler creates a local copy of each variable in *reduction()* and initializes it independently of the operator (e.g., 0 for " + ", 1 for "*"). If, say, three

Table 1: Clauses

Clause	Meaning
shared(variable_list)	Only one version of the variable exists, and all parallel program sections access it. All threads have read and write access. If a thread changes a variable, this also affects the other threads. Default: All variables are *shared()* except the loop variables in *#pragma omp for*.
private(variable_list)	Each thread has a private, non initialized copy of the variable. Default: Only loop variables are private.
default(shared|private|none)	Defines the default behavior of the variables: *none* means that you must explicitly declare each variable as *shared()* or *private()*.
firstprivate(variable_list)	Just like *private()*; however, in this case, all copies are initialized with the value of the variable before the parallel loop/region.
lastprivate(variable_list)	The variable is assigned the value from the last thread to change the variable in sequential processing after the parallel loop/region has been completed.

Listing 3: Avoiding Race Conditions

```
#ifdef _OPENMP
#include <omp.h>
#endif
#include <stdio.h>
int main() {
    double a[1000000];
    int i;
    #pragma omp parallel for
    for (i=0; i<1000000; i++) a[i]=i;
    double sum = 0;
    #pragma omp parallel for shared (sum) private (i)
    for ( i=0; i < 1000000; i++) {
        #pragma omp critical (sum_total)
        sum = sum + a[i];
    }
    printf("sum=%lf\n",sum);
}
```

threads are each handling one third of the loop, the master thread adds up the subtotals at the end.

Who is Faster?

Debugging parallelized programs is an art form in its own right. It is particularly difficult to find errors that do not occur in serial programs and do not occur regularly in parallel processing. This category includes what are known as race conditions: different results on repeated runs of a program with multiple blocks that are executed parallel to one another, depending on which thread is fastest each time. The code in Listing 3 starts by filling an array in parallel and then goes on to calculate the sum of these values in parallel.

Without the OpenMP *#pragma omp critical (sum_total)* statement in line 13, the following race condition could occur:

- Thread 1 loads the current value of *sum* into a CPU register.
- Thread 2 loads the current value of *sum* into a CPU register.
- Thread 2 adds *a[i + 1]* to the value in the register.
- Thread 2 writes the value in the register back to the *sum* variable.
- Thread 1 adds *a[i]* to the value in the register.
- Thread 1 writes the value in the register to the *sum* variable.

Because thread 2 overtakes thread 1 here, thus winning the “race,” *a[i + 1]* would not be added correctly. Although thread 2 calculates the sum and stores it in the *sum* variable, thread 1 overwrites it with an incorrect value.

The *#pragma omp critical* statement makes sure that this does not happen. All threads execute the critical code, but only one at any time. The example in Listing 3 thus performs the addition correctly without parallel threads messing up the results. For elementary operations (e.g., *i++*) *#pragma omp atomic* will atomically execute a command. Write access to *shared()* variables also should be protected by a *#pragma omp critical* statement.

Listing 4: Unavoidable Barrier

```
#pragma omp parallel shared (A, B, C)
{
  Calculationfunction(A,B);
  printf("B was calculated from A\n");
#pragma omp barrier
  Calculationfunction(B,C);
  printf("C was calculated from B\n");
}
```

Listing 5: Hello, World

```
/* helloworld.c (OpenMP Version) */
#
#ifdef _OPENMP
#include <omp.h>
#endif
#include <stdio.h>
int main(void)
{
  int i;
#pragma omp parallel for
  for (i = 0; i < 4; ++i)
  {
    int id = omp_get_thread_num();
    printf("Hello, World from thread %d\n", id);
    if (id==0)
      printf("There are %d threads\n", omp_get_num_threads());
  }
  return 0;
}
```

Is Everyone There?

In some cases, it is necessary to synchronize all the threads.The *#pragma omp barrier* statement sets up a virtual hurdle: All the threads wait until the last one reaches the barrier before processing can continue. But think carefully before you introduce an artificial barrier – causing threads to suspend processing is going to affect the performance boost that parallelizing the program gave you. Threads that are waiting do not do any work. Listing 4 shows an example in which a barrier is unavoidable.

The *Calculationfunction()* line in this listing calculates the second argument with reference to the first one. The arguments in this case could be arrays, and the calculation function could be a complex mathematical matrix operation. Here, it is essential to use *#pragma omp barrier* – the failure to do so would mean some threads would start with the second round of calculations before the values for the calculation in *B* become available.

Some OpenMP constructs (such as *parallel*, *for*, *single*) include an implicit barrier that you can explicitly disable by adding a *nowait* clause, as in *#pragma omp for nowait*. Other synchronize mechanisms include:

- *# pragma omp master {Code}*: Code that is only executed once and only by the master thread.
- *# pragma omp single {Code}*: Code that is only executed once, but not necessarily by the master thread
- *# pragma omp flush (Variables)*: Cached variables written back to main memory ensures a consistent view of the memory.

These synchronization mechanisms will help keep your code running smoothly in multi-processor environments.

Library Functions

OpenMP has a couple of additional functions, which are listed in Table 2. If you want to use them, you need to include the *omp.h* header file in C/C++. To make sure the program will build with-

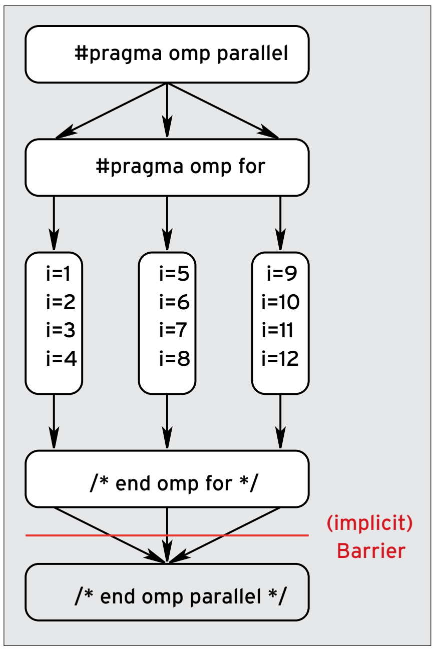


Figure 4: A Parallel for Loop.

out OpenMP, it would make sense to add the *#ifdef _OPENMP* line for conditional compilation.

```
#ifdef _OPENMP
#include <omp.h>
threads = omp_get_num_threads();
#else
threads = 1
#endif
```

Locking functions allow a thread to lock a resource, by reserving exclusive access

$$\frac{\pi}{4} = \frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots + \frac{(-1)^n}{2n+1} + \cdots$$

Figure 5: Formula for calculating pi by Gregor Leibniz.

(*omp_set_lock()*) to it. Other threads can then use a *omp_test_lock()* query to find out whether the resource is locked. This setup is useful if you want multiple threads to write data to a file, but want to restrict access to one thread at a time. When you use locking functions, be careful to avoid deadlocks.

A deadlock can occur if threads need resources but lock each other out. For example, if thread 1 successfully locks up resource A and is now waiting to use resource B, while thread 2 does exactly the opposite. Both threads wait forever.

Listing 7: Calculating Pi

```
/* pi-openmp.c (OpenMP version) */
#
#include <stdio.h>
#define STEPCOUNTER 1000000000
int main(int argc, char *argv[])
 {
 long i;
 double pi = 0;
 #pragma omp parallel for reduction(+: pi)
 for (i = 0; i < STEPCOUNTER; i++) {
   /* pi/4 = 1/1 - 1/3 + 1/5 - 1/7 + ...
        To avoid the need to continually change
        the sign (s=1; in each step s=s*-1),
        we add two elements at the same time. */
        pi += 1.0/(i*4.0 + 1.0);
        pi -= 1.0/(i*4.0 + 3.0);
 }
 pi = pi * 4.0;
 printf("Pi = %lf\n", pi);
 return 0;
 }
```

Environmental Variables

Some environmental variables control the run-time behavior of OpenMP programs; the most important is *OMP_NUM_THREADS*. It specifies how many threads can operate in a parallel regions, because too many threads will actually slow down processing. The *export OMP_NUM_THREADS = 1* tells a program to run with just one thread in bash – just like a normal serial program.

OpenMP Hands On

To use OpenMP in your own programs, you need a computer with more than one CPU, or a multi-core CPU and an OpenMP-capable compiler. GNU compilers later than version 4.2 support OpenMP. Also, the Sun compiler for Linux is free [2], and the Intel Compiler is free for non-commercial use [3].

Listing 5 shows an OpenMP version of the classic *Hello World* program. To enable OpenMP, set *-fopenmp* when launching GCC. Listing 8 shows the

Listing 6: Parallel Pi

```
$ gcc -Wall -fopenmp -o pi-openmp pi-openmp.c
$ export OMP_NUM_THREADS=1 ; time ./pi-openmp
Pi = 3.141593
real    0m31.435s
user    0m31.430s
sys     0m0.004s
$ export OMP_NUM_THREADS=2 ; time ./pi-openmp
Pi = 3.141593
real    0m15.792s
user    0m31.414s
sys     0m0.012s
```

Table 2: OpenMP Functions

Function	Explanation
int omp_get_num_threads()	Gets the number of threads.
int omp_get_thread_num()	Gets the current thread number.
void omp_set_num_threads(int)	Sets the number of threads to be used in future parallel regions.
Locking Functions	
void omp_init_lock(omp_lock_t)*	Initializes a lock.
void omp_set_lock(omp_lock_t)*	Waits and then sets a lock; blocks if the lock is not available.
int omp_test_lock(omp_lock_t)*	Waits and then sets a lock; does not block if the lock is not available.
void omp_unset_lock(omp_lock_t)*	Removes a lock.
void omp_destroy_lock(omp_lock_t)*	Destroys a lock.

Listing 8: Building Hello World

```
$ gcc -Wall -fopenmp helloworld.c
$ export OMP_NUM_THREADS=4
[...]
$ ./a.out
Hello World from thread 3
Hello World from thread 0
Hello World from thread 1
Hello World from thread 2
There are 4 threads
```

Listing 9: Notification

```
$ icc -openmp helloworld.c
helloworld.c(8): (col. 1) remark:
OpenMP DEFINED LOOP WAS PARALLELIZED.
```

commands for building the program along with the output.

If you are using the Sun compiler, the compiler option is *-xopenmp*. With the Intel compiler, the option is *-openmp*. The Intel compiler even notifies the programmer if something has been parallelized (Listing 9).

Benefits?

For an example of a performance boost with OpenMP, I'll look at a test that calculates pi [4] with the use of Gregory Leibniz's formula (Listing 7 and Figure 5). This method is by no means the most efficient for calculating pi; however, the goal here is not to be efficient but to get the CPUs to work hard.

Parallelizing the *for()* loop with OpenMP does optimize performance (Listing 6). The program runs twice as fast with two CPUs than with one, in that more or less the whole calculation can be parallelized.

If you monitor the program with the *top* tool, you will see that the two CPUs really are working hard and that the *pi-openmp* program really does use 200 percent CPU power.

This effect will not be quite as pronounced for some problems, in which case, you might need to resort to serial execution for a large proportion of the program. Of course, your two CPUs will not be a big help in such a case, and the performance boost will be less significant. Amdahl's Law [5] (see the "Amdahl's Law" box for an explanation) applies here. ■

Amdahl's Law

"Speedup" describes the factor by which a program can be accelerated with parallelization. In an ideal case, program execution with *N* processors would take just 1/*N* of the time required by a serial program. This ideal case is known as linear speedup. In the real world, linear speedup often is impossible to achieve because some parts of a program do not particularly lend themselves to parallelization.

Given a part of a program that supports parallelization, *P* (thus, 1 – *P* is the non-parallelizable part), and the number of processors available, *N*, the maximum speedup is calculated by the formula in Figure 6.

$$\frac{1}{(1-P)+\frac{P}{N}}$$

Figure 6: Amdahl's Law.

If the serial part of the program (1-P) is 1/4, the speedup cannot be greater than 4 – no matter how many processors you use.

INFO

[1] OpenMP homepage: *http://www.openmp.org*

[2] Sun compiler: *http://developers.sun.com/sunstudio/*

[3] Intel compiler: *http://www.intel.com/cd/software/products/asmo-na/eng/compilers/clin/*

[4] Calculating pi (Wikipedia): *http://en.wikipedia.org/wiki/Computing_Pi*

[5] Amdahl's law (Wikipedia): *http://en.wikipedia.org/wiki/Amdahl's_law*