JANUARY 2004 LINUX MAGAZINE Linux in Europe • User Mode Linux • PHP 5 Reflection Volume 6 / Issue 1

LINUX MAGAZINE

OPEN SOURCE. OPEN STANDARDS.

USER MODE LINUX: RUN LINUX ON LINUX

THE STATE OF THE EUROPEAN UNION

Linux is Hopping Across the Pond

PHP 5 REFLECTION

The Path to Self-Discovery

XSH

Navigate XML Documents Interactively

ALSO INSIDE:

- Samba 3's *net* Command
- Go Parallel with OpenMP
- Diagramming with DIA

JANUARY 2004

$5.95US $6.95CAN

01

0 09281 01105 7

WWW.LINUXMAGAZINE.COM

PROJECT OF THE MONTH: phpBB

Multi-Processing with OpenMP

By Forrest Hoffman

In this column's previous discussions of parallel programming, the focus has been on distributed memory parallelism, since most Linux clusters are best suited to this programming model. Nevertheless, today's clusters often contain two or four (or more) processors per node. While one could simply start multiple MPI processes on such nodes to use these processors, taking best advantage of the hardware requires a different approach. Processors within a node typically share all the memory within that node, and they can communicate much more quickly with each other than with processors on other nodes.

What's needed is a method that can be used to exploit a different level of parallelism, that which is available on symmetric multi-processor (SMP) machines. While a number of shared-memory-supporting packages exist — including SHMEM (shared memory used in conjunction with System V Inter-Process Communication), HPF (High-Performance FORTRAN), Pthreads (POSIX threads), as well as some vendor-specific compiler directives — one of the easiest to use is *OpenMP*.

Widely accepted by the high performance computing (HPC) community, OpenMP is a specification for a set of compiler directives, an applications programming interface (API), and a set of environment variables that can be used to specify shared memory parallelism in FORTRAN and C/C++ programs. The OpenMP specifications were developed by the OpenMP Architecture Review Board (ARB), a group of hardware and software vendors working in conjunction with government and academic researchers. To read the specifications or learn more about the OpenMP Architecture Review Board, see http://www.openmp.org.

Most commercial supercomputer vendors provide implementations of OpenMP optimized for their hardware, and most commercial compilers for Linux offer OpenMP support. For example, Intel's compilers are designed to deliver optimal performance for their new hyperthreading processors. But you don't need a commercial supercomputer or even a Linux cluster to use OpenMP. In fact, it may be used to take advantage of the multiple processors in a machine. Moreover, you may find OpenMP to be an easy way to exploit some inherent concurrency in your application even on a single processor computer.

OpenMP may be used to obtain most of the parallel performance you can expect from your code, or it may serve as a stepping stone to parallelizing an entire application with MPI. For large computational science problems running on clusters of SMP machines, OpenMP is often used in conjunction with MPI to provide two levels of simultaneous parallelism for optimal performance.

How OpenMP Works

OpenMP is based on a *fork and join* model of parallel execution in which a *master thread* (thread number 0) spawns a *team of threads* as needed in *parallel regions* of the code. At the end of a parallel region, the team of threads collapses back to a single thread (the master), and serial execution continues from that point. Any number of parallel regions may be contained in a single program, so it may fork and join many times throughout program execution, as shown in *Figure One*.

OpenMP is usually used to parallelize one or more time-consuming loops in an array-based program. The loop itera-

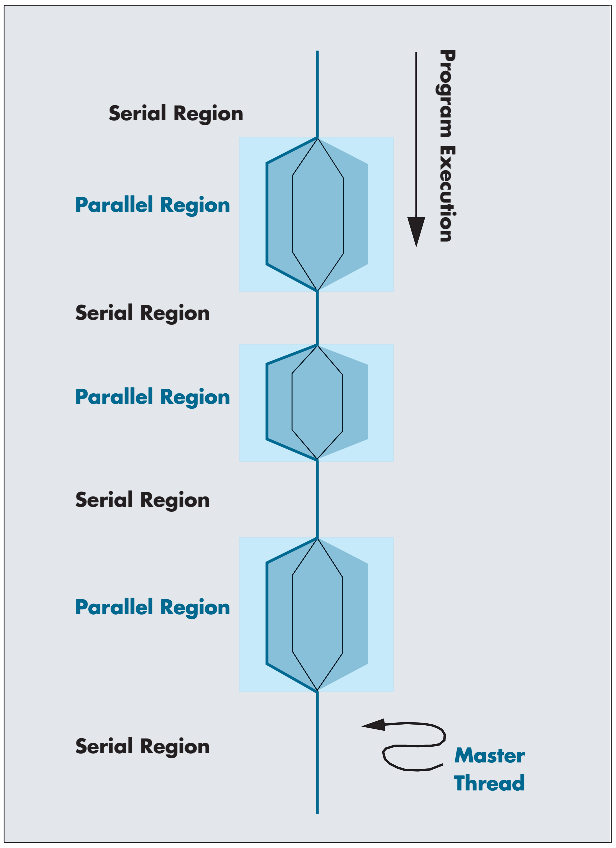

FIGURE ONE: OpenMP uses a fork and join model of execution in which a master thread "forks" a team of threads in parallel regions of code. After parallel execution of tasks, these threads "join" to the master thread, and the master executes serial regions alone.

LISTING ONE: *stat.c*, an example of using OpenMP

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#ifdef _OPENMP
#include <omp.h>
#endif /* _OPENMP */

#define ROWS   8
#define COLS   10000000

int num_threads;
double array[(ROWS*COLS)];

void fill_array()
{
  int i, j;

  srandom((unsigned int)0);
  for (j = 0; j < ROWS; j++)
    for (i = 0; i < COLS; i++)
      array[(j*COLS+i)] = (double)random() /
        RAND_MAX;

  return;
}

double sum_row(int j)
{
  int i, k;
  double row_sum = 0.0, row_mean, sum_squares = 0.0,
    row_sigma, junk = 0.0;

  printf("sum_row(%d) called", j);
#ifdef _OPENMP
  printf(" by thread #%d", omp_get_thread_num());
#endif /* _OPENMP */
  printf("\n");
  for (i = 0; i < COLS; i++)
    row_sum += array[(j*COLS+i)];
  row_mean = row_sum / (double)COLS;
  for (i = 0; i < COLS; i++)
    sum_squares += (array[(j*COLS+i)] - row_mean) *
      (array[(j*COLS+i)] - row_mean);
  row_sigma = sqrt(sum_squares / (double)COLS);
  /* these loops waste time to emulate a bunch of
      complex calculations */
  for (i = 0; i < 2; i++) {
    for (k = 0; k < ROWS * COLS; k++) {
      if (k%2) junk += array[k] * array[k];
      else junk -= array[k] * array[k];
      junk = sqrt(junk);
    }
  }

  printf("row %d: mean = %lf, standard deviation =
    %lf\n", j, row_mean, row_sigma);

  return row_sum;
}

double sumsq_row(int j, double mean)
{
  int i;
  double sum_squares = 0.0;

  printf("sumsq_row(%d, %lf) called", j, mean);
#ifdef _OPENMP
  printf(" by thread #%d", omp_get_thread_num());
#endif /* _OPENMP */
  printf("\n");
  for (i = 0; i < COLS; i++)
    sum_squares += (array[(j*COLS+i)] - mean) *
      (array[(j*COLS+i)] - mean);

  printf("row %d: sum of squares = %lf\n", j,
sum_squares);

  return sum_squares;
}

int main(int argc, char **argv)
{
  int j;
  double sum = 0.0, mean, sum_squares = 0.0, sigma;

#ifdef _OPENMP
  num_threads = omp_get_max_threads();
#else /* _OPENMP */
  num_threads = 1;
#endif /* _OPENMP */

  printf("Program started with ");
#ifdef _OPENMP
  printf("%d threads\n", num_threads);
#else /* _OPENMP */
  printf("threading disabled\n");
#endif /* _OPENMP */

  fill_array();

#pragma omp parallel for private(j) reduction(+:
sum)
  for (j = 0; j < ROWS; j++)
    sum += sum_row(j);
  mean = sum / (double)(ROWS * COLS);
#pragma omp parallel for private(j) reduction(+:
  sum_squares)
  for (j = 0; j < ROWS; j++)
    sum_squares += sumsq_row(j, mean);
  sigma = sqrt(sum_squares / (double)(ROWS * COLS));
  printf("Array sum = %lf, mean = %lf, standard
    deviation = %lf\n", sum, sum / (double)(ROWS *
      COLS), sigma);

  exit(0);
}
```

tions shouldn't depend on each other; that is, iterations should be order independent and must be able to be computed concurrently. A simple directive placed above such loops enable OpenMP threading with little or no additional code modifications. Since the constructs are directives, the same code can be compiled with compilers with no OpenMP. When run, the same answers should be obtained.

Structured blocks of code can be parallelized with OpenMP, but those blocks must have a single point of entry and a single point of exit. The only other branches allowed are `STOP` statements in FORTRAN and `exit()` in C/C++. There is an implied barrier at the end of a structured block that causes threads to wait until they have all completed execution of that block before proceeding. This behavior can be overridden with a `nowait` clause in the block's directive.

OUTPUT ONE: *stat.c* without OpenMP enabled

```
[forrest@node01]$ pgcc -O -o stat stat.c
[forrest@node01]$ time ./stat

Program started with threading disabled
sum_row(0) called
row 0: mean = 0.500034, standard deviation = 0.288651
sum_row(1) called
row 1: mean = 0.500045, standard deviation = 0.288658
sum_row(2) called
row 2: mean = 0.499887, standard deviation = 0.288665
sum_row(3) called
row 3: mean = 0.500147, standard deviation = 0.288600
sum_row(4) called
row 4: mean = 0.499920, standard deviation = 0.288638
sum_row(5) called
row 5: mean = 0.500086, standard deviation = 0.288568
sum_row(6) called
row 6: mean = 0.500124, standard deviation = 0.288674
sum_row(7) called
row 7: mean = 0.499992, standard deviation = 0.288713
sumsq_row(0, 0.500029) called
row 0: sum of squares = 833193.862532
sumsq_row(1, 0.500029) called
row 1: sum of squares = 833234.227900
sumsq_row(2, 0.500029) called
row 2: sum of squares = 833273.014429
sumsq_row(3, 0.500029) called
row 3: sum of squares = 832899.634968
sumsq_row(4, 0.500029) called
row 4: sum of squares = 833118.311465
sumsq_row(5, 0.500029) called
row 5: sum of squares = 832716.030687
sumsq_row(6, 0.500029) called
row 6: sum of squares = 833324.233857
sumsq_row(7, 0.500029) called
row 7: sum of squares = 833549.315257
Array sum = 40002341.975667, mean = 0.500029,
  standard deviation = 0.288646

real 5m22.651s
user 5m20.828s
sys 0m1.840s
```

OpenMP Syntax

OpenMP directives are based on the standard `#pragma` directives for C/C++, and on `C$OMP`, `!$OMP`, or `*$OMP` comments in FORTRAN. Compliant compilers usually require specification of a command line flag to enable interpretation of OpenMP directives. When enabled, the `_OPENMP` preprocessor macro is defined (as *yyyymm* signifying the year and month of the approved specification) when compiling.

Directives fall into three classes: parallel constructs, work-sharing constructs (within parallel constructs), or combined, parallel work-sharing constructs. The formats of the directives are…

```
#pragma omp directive-name [clause[ [,]
  clause]...]
```

… for C/C++, and…

```
!$OMP directive-name [clause[ [,]
  clause]...]
```

for FORTRAN-90. FORTRAN also has ending directives since it doesn't have a code-blocking structure. Most directives can have a number of clauses. Clauses control how variables and data are shared, how the work is partitioned, and how threads are scheduled.

OK, enough formalism. Let's jump into an example.

Listing One contains a fairly simple piece of C code, called *stat.c* that uses OpenMP. It calculates the mean and standard deviation of a large array of random numbers between 0 and 1. (We all know the answer; that's why it's a good test code.) The code may be compiled with or without OpenMP enabled, and when run generates the same result in either case.

The OpenMP API elements are "wrapped" with `#ifdef`s referencing the `_OPENMP` macro that's defined when the appropriate compiler command line flag is used to enable OpenMP. When enabled, the OpenMP header file, *omp.h*, is included and two calls are used somewhere in the code: `omp_get_max_threads()` and `omp_get_thread_num()`.

Starting in `main()`, `omp_get_max_threads()` is used to obtain the maximum number of threads that may be used during program execution. This number is stored in `num_threads`. If OpenMP is disabled, `num_threads` is set to one. This may be good practice in codes where some information about each thread is stored in an automatic array. Next, the program prints `Program started with` and either reports the number of threads or prints *threading disabled.*

Then `fill_array()` is called to stuff random numbers into a global array of size `ROWS * COLS`. The random number seed is set to a specific value (in this case `NULL`), so that the exact same result is obtained each time the program is run. In this way, you can check to be sure that the serial version and the parallel version give the same output.

Now the fun begins. The code loops over all rows and calls a routine (`sum_row()`) to sum the values in each row and accumulate the results in the variable `sum`. Clearly the sum of values in each row can be calculated simultaneously without side-effects. Therefore, you can place an OpenMP directive above the `for` loop to spawn multiple threads that can run `sum_row()` concurrently.

The `parallel for` construct is used along with the `private(`*`variable-list`*`)` and `reduction(`*`operator : variable-list`*`)` clauses. The `for` loop following such a directive must have *canonical shape*. That means it must look like: `for (`*`init; var oper b; incr`*`)` where *`init`* must be of the form *`integer var = initial value`*; *`var`* must be a variable of signed integer type; *`oper`* is one of `<`, `<=`, `>`, or `>=`; *`b`* is the bound for the loop; and *`incr`* must be one of a limited number of forms for incrementing or decrementing *var*. Moreover, it must be possible to calculate the loop length — the number of iterations — upon entry to the loop, and the loop can have no other means of exit (no `break`s allowed!), except by satisfying the *`var oper b`* expression or calling `exit()`.

The `for` loop in the example meets all of these criteria. However, some non-deterministic models may have loops that do not. If such loops in these other models need shared-memory parallelization, OpenMP is not the solution. Fortunately, a large class of models can be parallelized in deterministic fashion.

The `parallel for` directive uses a `private` clause that specifies that `j` is a private variable. That means that each thread should have its own copy of `j`, possibly with a unique on each thread. Since this is the loop control variable, each thread will have a different value for `j`. It turns out that loop control variables are private by default, so it's not necessary to explicitly specify the variable in a private clause.

The `reduction` clause tells the compiler that the value for the variable `sum` should be private on each thread (and initialized to zero), and that their values should be added up (and stored for the master thread) before leaving the loop. If this clause hadn't been used, all threads would have added the return values of `sum_row()` to the same shared `sum` variable. While this might work in some configurations, it could cause values to be lost when multiple threads contend for updating the same memory location.

After the loop is complete, the master thread computes the mean of the entire array by dividing the accumulated row sums by the number of array elements (`ROWS * COLS`). Next, a second parallel region — another `parallel for` — is entered. This loop computes the sum of the squares by row, and accumulates the results in `sum_squares`. Then, the master thread calculates and reports the standard deviation, `sigma`.

Before compiling and running the code, a couple of things to keep in mind. First, both sum_row() and sumsq_row() print out the number of the thread that's executing the routine (when compiled with OpenMP enabled). This allows you to know which thread is working on which row in the matrix.

OUTPUT TWO: Compiling and running *stat* with OpenMP using 2 threads.

```
[forrest@node01]$ pgcc -mp -O -o stat stat.c

[forrest@node01]$ time OMP_NUM_THREADS=2 ./stat

Program started with 2 threads
sum_row(0) called by thread #0
sum_row(4) called by thread #1
row 0: mean = 0.500034, standard deviation = 0.288651
sum_row(1) called by thread #0
row 4: mean = 0.499920, standard deviation = 0.288638
sum_row(5) called by thread #1
row 1: mean = 0.500045, standard deviation = 0.288658
sum_row(2) called by thread #0
row 5: mean = 0.500086, standard deviation = 0.288568
sum_row(6) called by thread #1
row 2: mean = 0.499887, standard deviation = 0.288665
sum_row(3) called by thread #0
row 6: mean = 0.500124, standard deviation = 0.288674
sum_row(7) called by thread #1
row 3: mean = 0.500147, standard deviation = 0.288600
row 7: mean = 0.499992, standard deviation = 0.288713
sumsq_row(0, 0.500029) called by thread #0
sumsq_row(4, 0.500029) called by thread #1
row 0: sum of squares = 833193.862532
sumsq_row(1, 0.500029) called by thread #0
row 4: sum of squares = 833118.311465
sumsq_row(5, 0.500029) called by thread #1
row 5: sum of squares = 832716.030687
sumsq_row(6, 0.500029) called by thread #1
row 1: sum of squares = 833234.227900
sumsq_row(2, 0.500029) called by thread #0
row 6: sum of squares = 833324.233857
sumsq_row(7, 0.500029) called by thread #1
row 2: sum of squares = 833273.014429
sumsq_row(3, 0.500029) called by thread #0
row 3: sum of squares = 832899.634968
row 7: sum of squares = 833549.315257
Array sum = 40002341.975667, mean = 0.500029,
  standard deviation = 0.288646

real 3m0.119s
user 5m55.715s
sys 0m3.402s
```

Second, after the row sum and row sum of squares are computed in `sum_row()`, another set of loops is executed to calculate a variable called `junk`. These loops simply waste lots of time performing meaningless computations so that the `sum_row()` routine will take lots of time. It turns out that, because of the additional overhead required to handle OpenMP threading, this code would run faster without OpenMP if not for these time-wasting loops.

The last point is an important one. OpenMP parallelism improves performance only if sufficient amounts of concurrent work needs to be performed. Even with the long rows in the array in this example code, the computations can be performed so fast that the standard deviation can be calculated more quickly by a single thread. A certain amount of overhead is required for OpenMP even if only a single thread is used, so be sure the time-to-solution of your problem is improved before adopting any parallel construct.

Now, The Numbers

First, let's compile and run the code *without* OpenMP. *Output One* contains the results. This example uses the Portland Group's (http://www.pgroup.com) C compiler, which has OpenMP support. The code was run on a dual processor 1.0 GHz Pentium III machine, running RedHat Linux 7.3.

As you can see, the *stat* code printed the mean and standard deviation of each row in the matrix as well as the sum, mean, and standard deviation of the entire matrix. No surprises here: the mean is 0.5 and the standard deviation is 0.289. Thanks to the time-wasting loops, it took 5 minutes, 22 seconds to complete the calculations.

Next, we compile and run *stat* with two OpenMP threads. To enable OpenMP support on the Portland Group compilers, use the *–mp* command line option. The number of threads may be specified either in a directive, by calling `omp_set_num_threads()` within the code, or by setting the `OMP_NUM_THREADS` environment variable. In *Output Two*, the `OMP_NUM_THREADS` environment variable has been set to 2 while running the program to establish the requested number of threads.

The program reports that it was started with 2 threads, and you can see that sums for rows 0, 1, 2, and 3 were performed by thread 0, while sums for rows 4, 5, 6, and 7 were calculated by thread 1 concurrently. The same goes for the calls to `sumsq_row()`. The exact same answers were obtained, but this time the computation took only three minutes to complete.

Because of the overhead associated with OpenMP threads, you see a 1.8X speedup instead of something close to a 2.0 times speedup. In fact, when the same code is run on the same machine with only a single OpenMP thread, it takes 5 minutes, 35 seconds, or 13 seconds longer than when compiled without OpenMP.

It's possible, and sometimes advantageous, to use more threads than the number of available processors, but usually only in powers of two. When run with four threads on the same machine, the *stat* program completes in 3 minutes, 21 seconds. Because all the work is divided up perfectly symmetrically in our code, all threads are always working, so no performance gain was seen. However, in more complex codes where asymmetries exist, additional threads may improve overall performance.

Next month, we'll delve deeper into OpenMP, and learn about other constructs for shared–memory parallelism. Until then, try OpenMP on some of the loops in your favorite model.

Forrest Hoffman is a computer modeling and simulation researcher at Oak Ridge National Laboratory. He can be reached at forrest@climate.ornl.gov.