FEBRUARY 2004 LINUX MAGAZINE Storage Tank • PostgreSQL 7.4 • Berkeley DB XML Volume 6 / Issue 2

LINUX MAGAZINE

OPEN SOURCE. OPEN STANDARDS.

STORAGE TANK: IBM'S SMART SAN

POSTGRESQL 7.4
The Database Administrator's Database

BERKELEY DB XML
Native Storage for XML Documents

THE Z SHELL
Teach Your Old Shell New Tricks

ALSO INSIDE:

- Building Class-less Classes in Perl
- Mastering the XFT Font System
- MySQL's Many MyISAM Tables

FEBRUARY 2004

$5.95US $6.95CAN 02

0 09281 01105 7

WWW.LINUXMAGAZINE.COM

PROJECT OF THE MONTH: Compiere

OpenMP Multi-Processing, Part 2

By Forrest Hoffman

This month, we continue our focus on shared-memory parallelism using OpenMP. As a quick review, remember that OpenMP consists of a set of compiler directives, a handful of library calls, and a set of environment variables that can be used to specify run-time parameters. Available for both FORTRAN and C/C++ languages, OpenMP can often be used to improve performance on symmetric multi-processor (SMP) machines or SMP nodes in a cluster by simply (and carefully) adding a few compiler directives to the code. Most commercial compilers for Linux provide support for OpenMP, as do compilers for commercial supercomputers.

Last month's column provided a brief introduction to the *fork and join* model of execution, where a team of threads is spawned (or forked) at the beginning of a concurrent section of code (called a *parallel region*) and subsequently killed (or joined) at the end of the parallel region. Frequently used to parallelize one or more time-consuming loops in array-based programs, OpenMP splits up loop iterations (which must be independent at some level) among a specified number of threads, allowing iterations to be processed at the same time instead of sequentially.

The OpenMP syntax of directives and clauses and the `parallel for` directive were discussed in the previous column. This month, we take a step back to look at more rudimentary directives, and focus on the library calls and environment variables that support OpenMP parallelism. The descriptions and code examples are based on the C/C++ OpenMP 2.0 specification.

Directives from Above

OpenMP directives sit just above the code block that they affect. They are case sensitive, and take the form `#pragma omp` *directive-name [clause[[,] clause]…].*

The most basic directive is the `parallel` construct. It defines a structured block of code as a parallel region. This parallel region will be executed simultaneously by multiple OpenMP threads. The `parallel` directive accepts the `if`, `private`, `firstprivate`, `default`, `shared`, `copyin`, `reduction`, and `num_threads` clauses.

When program execution encounters a parallel construct, a team of threads is created if no `if` clause is specified or when the `if (`*`scalar-expression`*`)` evaluates to a non-zero value. When a team of threads is spawned, the thread that encountered the parallel construct becomes the *master thread* (with a thread number of 0), and all threads in the team execute the code region in parallel. If the `if` expression evaluates to zero, the region is executed by only a single thread.

The `private`, `firstprivate`, `default`, `shared`, `copyin`, and `reduction` clauses affect the data environment of the parallel region. We'll see how to use these clauses later. The `num_threads` clause can be used to specify the number of threads that should execute the parallel region. This clause overrides the `omp_set_num_threads()` library function (if called), which overrides the `OMP_NUM_THREADS` environment variable (if set). The `num_threads` clause affects only the parallel region directly below the parallel construct of which it is a part.

The number of threads that execute a parallel region is also dependent upon whether or not *dynamic adjustment* of the number of threads is enabled. If dynamic adjustment is disabled, the requested number of threads will execute the parallel region. On the other hand, if dynamic adjustment is enabled, the requested number of threads is the maximum number that may execute the parallel region, because the run-time environment automatically adjusts the number of threads to make best use of system resources. The `omp_set_dynamic()` library function and the `OMP_DYNAMIC` environment variable may be used to enable or disable dynamic adjustment.

If a thread in a team encounters another parallel directive, it spawns a new team of threads and becomes the master thread of that new team. By default, this sort of nested parallel region is serialized (that is to say that it's executed by a new team of one thread, the one that made itself a master). This default behavior can be changed by calling `omp_set_nested()` or by setting the `OMP_NESTED` environment variable. However, the number of threads that execute a nested parallel region is not defined by the OpenMP standard. The compiler is free to serialize any nested parallel regions.

There is an implied barrier at the end of a parallel region. That means that threads wait at that point until all threads have completed the parallel region, and then only the master thread continues executing code outside the parallel region. Many OpenMP constructs have an implied synchronization point at the end of the affected code block. For some of these constructs, the `nowait` clause may be used to eliminate this barrier; however, the `nowait` clause is not valid for a `parallel` construct.

In addition to the `parallel` construct, OpenMP includes a variety of work-sharing constructs that are used to divide up work among threads in a parallel region of code. These directives include the `for`, `sections`, and `single` constructs. All three of these directives must appear within a parallel region. The `parallel for` construct introduced last month is merely a short-hand for specifying a parallel region containing only a loop.

The `for` and `parallel for` constructs divide loop iterations among threads according to the scheme specified in the `schedule` clause or the `OMP_SCHEDULE` environment variable. The `schedule` clause takes one of the following forms:

- `schedule(static [,chunk_size])` deals out blocks of iterations of size `chunk_size` to each thread.
- `schedule(dynamic [,chunk_size])` lets each thread grab `chunk_size` iterations off a queue until all are done.
- `schedule(guided [,chunk_size])` specifies that threads dynamically grab blocks of iterations. The size of blocks starts large and shrinks down to size `chunk_size` as the calculation proceeds.

FIGURE ONE: The complete list of OpenMP Library Functions from the C/C++ OpenMP 2.0 specification

`void omp_set_num_threads(int num_threads)` sets the default number of threads to use for parallel regions (an alternative to the `OMP_NUM_THREADS` environment variable and the `num_threads` clause).

`int omp_get_num_threads(void)` returns the number of threads currently executing the parallel region in which it is called.

`int omp_get_max_threads(void)` returns the largest number of threads that could be used in subsequent parallel regions.

`int omp_get_thread_num(void)` returns the thread number of the thread executing the function.

`int omp_get_num_procs(void)` returns the number of physical processors available to the program/process.

`int omp_in_parallel(void)` returns a non-zero value if called within a parallel region; otherwise it returns 0.

`void omp_set_dynamic(int dynamic_threads)` enables or disables dynamic adjustment of threads (an alternative to the `OMP_DYNAMIC` environment variable).

`int omp_get_dynamic(void)` returns a non-zero value if dynamic adjustment is enabled; otherwise it returns 0.

`void omp_set_nested(int nested)` enables or disables nested parallelism (an alternative to the `OMP_NESTED` environment variable).

`int omp_get_nested(void)` returns a non-zero value if nested parallelism is enabled; otherwise it returns 0.

`void omp_init_lock(omp_lock_t *lock)` initializes a lock.

`void omp_init_nest_lock(omp_lock_t *lock)` initializes a nestable lock.

`void omp_destroy_lock(omp_lock_t *lock)` ensures that a lock is uninitialized.

`void omp_destroy_nest_lock(omp_lock_t *lock)` ensures that a nestable lock is uninitialized.

`void omp_set_lock(omp_lock_t *lock)` blocks the thread until the specified lock is available and then sets the lock; the lock must have been previously initialized.

`void omp_set_nest_lock(omp_lock_t *lock)` blocks the thread until the specified nestable lock is available and then sets the nestable lock; the lock must have been previously initialized.

`void omp_unset_lock(omp_lock_t *lock)` releases ownership of a lock.

`void omp_unset_nest_lock(omp_lock_t *lock)` releases ownership of a nestable lock.

`int omp_test_lock(omp_lock_t *lock)` attempts to set a lock but does not block execution of the thread.

`int omp_test_nest_lock(omp_lock_t *lock)` attempts to set a nestable lock but does not block execution of the thread.

`double omp_get_wtime(void)` returns the number of wall clock seconds since some (arbitrary) time in the past.

`double omp_get_wtick(void)` returns the number of seconds between successive clock ticks.

- `schedule(runtime)` specifies that schedule and chunk size taken from `OMP_SCHEDULE` environment variable.

OpenMP Library Functions and Environment Variables

We've already seen the importance of many OpenMP library functions and environment variables. Most of the library functions are used for querying or managing the threading environment, while the environment variables are used for setting run-time parameters.

A complete list of library functions is contained in *Figure One*, and a complete list of environment variables is shown in *Figure Two*. In most cases, calls to library functions override parameters set in environment variables.

It's often desirable to know the number of threads that will be used in parallel regions during program execution. The `omp_get_max_threads()` routine can be used to determine the maximum number of threads that can be used. However, if dynamic adjustment is enabled or the `num_threads` clause for a directive is set to a number smaller than the requested number of threads for the program, a parallel region may not execute with the maximum number of threads. A call to `omp_get_num_threads()` made *within* the parallel region of interest returns the actual number of threads being used in that region. When called *outside* a parallel region, `omp_get_num_threads()` returns a value of one.

Let's Get Coding!

The *omptest1.c* program shown in *Listing One* uses a number of OpenMP library functions and uses a `parallel` construct containing a `for` construct to perform some simple calculations. This program will compile and run serially (with OpenMP turned off) or in parallel, and (importantly!) generates the same answer in either case.

FIGURE TWO: OpenMP Environment Variables

`OMP_SCHEDULE` is used to set the default scheduling type and optional chunk size for `for` and `parallel for` directives.

`OMP_NUM_THREADS` is used to set the default number of threads to use during execution, unless it's overridden by calls to `omp_set_num_threads()` or by `num_threads` clause on a `parallel` directive.

`OMP_DYNAMIC` can be set to `TRUE` or `FALSE` to enable or disable dynamic adjustment of threads, respectively.

`OMP_NESTED` can be set to `TRUE` or `FALSE` to enable or disable nested parallelism, respectively.

Remember from last month's column that the `_OPENMP` preprocessor macro is defined when code is compiled with OpenMP enabled. It's a good idea to take advantage of this useful macro by wrapping OpenMP-specific program elements, like the inclusion of the *omp.h* header file and calls to OpenMP library functions, inside `#ifdef`s. This ensures that the same code can be compiled for serial execution with OpenMP disabled.

Just inside `main()`, a call is made to `omp_get_num_procs()` to discover the number of available CPUs on the machine. Then this number is passed in a call to `omp_set_num_threads()` to set the number of OpenMP threads that should be used in parallel regions. Remember that requesting the number of threads this way overrides any value in the `OMP_NUM_THREADS` environment variable. Next, the maximum number of threads is obtained by calling `omp_get_max_threads()`. When compiled without OpenMP, the `nprocs` and `max_threads` variables are simply set to one.

In the next set of statements, the number of processors and the maximum number of threads are printed, along with the actual definition value of `_OPENMP`. According to the standard, this value should be set to the year and month of the approved OpenMP specification. Looking at *Figure Three*, we see that the value printed for the Portland Group compiler used is `199810`.

A call to `omp_in_parallel()` is made to test whether or not execution is within a parallel region. We know it is outside of a parallel region, but we're just testing the library functions here. As expected, the line `outside a parallel region` is printed in *Figure Three*.

Next, we encounter a `parallel` construct and a structured block of code within curly brackets. At this point, threads will be spawned that all execute the code block. Two integer variables, `tid` and `num_threads`, are declared inside the block of code — these variables are private to the threads, meaning each thread has its own copy of these variables. Then we test the return value from `omp_in_parallel()` again to make sure it knows we are now in a parallel region.

At this point, `omp_get_num_threads()` is called, and the value is stored in `num_threads`. Then `omp_get_thread_num()` is called, and its return value is stored in `tid`. When compiled without OpenMP, `num_threads` is set to one and `tid` is set to zero. All threads then print a `Hello World` message that includes their thread id (`tid`) and the number of threads (`num_threads`).

We should have as many `Hello World!` lines printed as there are threads executing the parallel region. According to *Figure Three*, `inside a parallel region` is printed twice (once for each thread), and "Hello World!" lines are printed by threads 0 and 1. As usual with parallel codes, output from different threads is not necessarily in order.

LISTING ONE: *omptest1.c,* an OpenMP application in C

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

int max_threads;

double func(int i, int j)
{
  int k;
  double val;

  /* simulate lots of work here */
  for (k = 0; k < 10000; k++)
    val += sqrt((k+1)*(k+1) /
              ((i*i) + (j*j) + 1));

  return sqrt((i*i) + (j*j));
}

int main(int argc, char **argv)
{
  int i, j, nprocs;
  double val, total;

#ifdef _OPENMP
  nprocs = omp_get_num_procs();
  omp_set_num_threads(nprocs);
  max_threads = omp_get_max_threads();
#else /* _OPENMP */
  nprocs = 1;
  max_threads = 1;
#endif /* _OPENMP */

  printf("Program started on %d processor node
    with ", nprocs);
#ifdef _OPENMP
  printf("a maximum of %d threads\n", max_threads);
  printf("Note: _OPENMP is defined as %d\n",
    _OPENMP);
#else /* _OPENMP */
  printf("threading disabled\n");
#endif /* _OPENMP */

#ifdef _OPENMP
  if (omp_in_parallel()) printf("inside a
    parallel region\n");
  else printf("outside a parallel region\n");
#endif /* _OPENMP */

#pragma omp parallel
  {
    int tid, num_threads;

#ifdef _OPENMP
    if (omp_in_parallel()) printf("inside a
      parallel region\n");
    else printf("outside a parallel region\n");
    num_threads = omp_get_num_threads();
    tid = omp_get_thread_num();
#else /* _OPENMP */
    num_threads = 1;
    tid = 0;
#endif /* _OPENMP */
    printf("Hello World! from thread %d of %d\n",
      tid, num_threads);

    /* loop inside a parallel region is executed by
      all threads */
    for (i = 0; i < 4; i++)
      printf("%d: loop 1 iteration %d\n", tid, i);

    /* loop inside a parallel region with a OpenMP
      for (workload
     * sharing) directive is split up among threads */
#pragma omp for reduction(+: total) private(j, val)
    for (i = 0; i < 10; i++) {
      printf("%d: loop 2 iteration %d\n", tid, i);
      for (j = 0; j < 1000; j++) {
        val = func(i, j);
        total += val;
      }
    }

    printf("Total = %lf\n", total);

    printf("Goodbye World! from thread %d of %d\n",
      tid, num_threads);
  }

  printf("Finished!\n");

  exit(0);
}
```

Next, a `for` loop is encountered. This loop, which simply prints out the loop iteration number with the thread number at the beginning of the line, is fully executed by each of the threads. The lines in *Figure Three* confirm this behavior. If loop iterations should instead be split *across* threads (so that each is executed only once), an OpenMP `for` construct must be used.

The next loop is such a loop, and it has a `for` directive above it. Here, the ten iterations are split among the available threads. As we can see in *Figure Three*, thread 0 (the master thread) processed iterations 0-4, while iterations 5-9 are processed by thread 1. All the rules and restrictions discussed in last month's column about the `parallel for` construct apply here as well.

The implied barrier at the end of the `i` loop causes threads to wait for all other threads to complete the calculation

before continuing outside the `for` code block. Doing otherwise (by specifying a `nowait` clause with the `for` directive) could cause an incorrect value of `total` to be printed in the next program statement by one or more threads.

Next, all threads print the accumulated total and `Goodbye World!`. That ends the `parallel` code block. Then `Finished!` is printed by only the master thread, and the program exits.

Figure Four contains similar results from compiling and running the same code with OpenMP disabled. As we can see, the same answers are obtained either way, but the OpenMP version ran almost twice as fast as the serial version when it used two threads.

The `reduction` and `private` clauses used in the `for` construct are critically important. Removing them will likely cause the program to generate a different answer when using multiple threads. Go ahead and try it.

This sort of data environment problem results in what's called a *race condition*. Since the variables `total`, `j`, and `val` are declared outside the parallel region, they are shared by default. By declaring `j` and `val` private, separate versions of these variables are created for each thread. Since we want `total` to have the accumulated sum from all threads, the `reduction` clause can be used to make that happen. Remember that the loop index, `i`, is automatically made private so it does not have to be declared as such.

These data environment issues can be tricky, so we'll discuss the OpenMP clauses used to manage them in next month's column. In addition, we'll also investigate the remaining OpenMP directives.

But you know enough now to be dangerous, so get started on your own code today!

Forrest Hoffman is a computer modeling and simulation researcher at Oak Ridge National Laboratory. He can be reached at forrest@climate.ornl.gov. You can download the code used in this column from http://www.linuxmagazine.com/downloads/2004-02/extreme.

FIGURE THREE: Parallel execution of *omptest1*

```
[forrest@node01 openmp]$ pgcc -mp -O -o
  omptest1 omptest1.c
[forrest@node01 openmp]$ time ./omptest1

Program started on 2 processor node with a
  maximum of 2 threads
Note: _OPENMP is defined as 199810
outside a parallel region
inside a parallel region
Hello World! from thread 0 of 2
0: loop 1 iteration 0
0: loop 1 iteration 1
0: loop 1 iteration 2
inside a parallel region
Hello World! from thread 1 of 2
1: loop 1 iteration 0
1: loop 1 iteration 1
1: loop 1 iteration 2
1: loop 1 iteration 3
1: loop 2 iteration 5
0: loop 1 iteration 3
0: loop 2 iteration 0
1: loop 2 iteration 6
0: loop 2 iteration 1
1: loop 2 iteration 7
0: loop 2 iteration 2
1: loop 2 iteration 8
0: loop 2 iteration 3
1: loop 2 iteration 9
0: loop 2 iteration 4
Total = 4995904.563410
Goodbye World! from thread 1 of 2
Total = 4995904.563410
Goodbye World! from thread 0 of 2
Finished!

real 0m3.622s
user 0m7.195s
sys 0m0.002s
```

FIGURE FOUR: Serial execution of *omptest1*

```
[forrest@node01 openmp]$ pgcc -O -o omptest1-
  serial omptest1.c
[forrest@node01 openmp]$ time ./omptest1-
  serial

Program started on 1 processor node with
  threading disabled
Hello World! from thread 0 of 1
0: loop 1 iteration 0
0: loop 1 iteration 1
0: loop 1 iteration 2
0: loop 1 iteration 3
0: loop 2 iteration 0
0: loop 2 iteration 1
0: loop 2 iteration 2
0: loop 2 iteration 3
0: loop 2 iteration 4
0: loop 2 iteration 5
0: loop 2 iteration 6
0: loop 2 iteration 7
0: loop 2 iteration 8
0: loop 2 iteration 9
Total = 4995904.563410
Goodbye World! from thread 0 of 1
Finished!

real  0m7.204s
user  0m7.193s
sys 0m0.004s
```