A Beginner's Primer to OpenMP

logicchild

19 Feb 2010 CPOL

An article written with the purpose to help any beginner to use OpenMP.

Introduction

There is a lot of technical documentation espousing concurrency and parallel computing. Code can run concurrently on one microprocessor, but unless there are multiple cores, or CPUs, the code will not run in parallel. In this quest for data parallelism, the for loop construct is often stressed because it is the only control flow structure that steps over every item in, say, an array of elements. This can, in most cases, define an iteration space, where the bounds are divided. These separate parts that sum to form the whole are executed simultaneously on separate cores that reside on the same physical microprocessor. This article is for the beginning student of OpenMP. Threading libraries like the Intel TBB make strong use of the C++ language, but there is a lot of research going on in order to make OpenMP work more effectively with C++. This might involve some sort of integration with the algorithms defined in the Standard Template Library. As of the writing of this article, the current version of OpenMP is version 3.0.

OpenMP is a means of achieving parallelism and therefore concurrency using the C and FORTRAN languages, despite the fact that there are a few C++ OpenMP source code examples out there on the Internet. The OpenMP APIs define certain things differently for both of those languages most of the time, but they also define things the same way for certain elements some of the time. In short, the idea is to find code blocks that can be parallelized by using the pragma compiler directive. Other looping constructs eventually branch out of the loop when a condition is met or a condition is a Boolean true. An OpenMP construct is defined to be a directive (pragma) plus a block of code. It cannot be just any block of code: it must be a structured block. That is, a block with one point of entry at the top and a single point of exit at the bottom. An OpenMP program, therefore, cannot branch into or out of a structured block of code. The student of OpenMP should refer to the home web site, http://www.OpenMP.org, continually when referencing tutorials that seek to explain how it works. The code examples provided were compiled on the command line using the *cl.exe* compiler and the */openmp* switch.

Definitions

One of the goals the designers had for OpenMP is for programs to execute and yield the same results whether they use one thread or many threads. The term for a program to yield the same results if it executed one thread or many is called sequentially equivalent. Incremental parallelism refers to a programming practice (which is not always possible) in which a sequential

program evolves into a parallel program. That is, the programmer starts working with a sequential program from the top down, block by block, and finds pieces of code that are better off executing in parallel. Thus, parallelism is added incrementally. Having said that, OpenMP is a collection of compiler directives, library routines, and environmental variables that specify shared-memory concurrency in FORTRAN, C, and (soon) C++ programs. Note that in the Windows OS, any memory that can be shared is shared. OpenMP directives demarcate code that can be executed in parallel (called parallel regions), and control how code is assigned to threads. The threads in OpenMP code operate under the fork-join model. When the main thread encounters a parallel region while executing an application, a team of threads is forked off, and these threads begin to execute the code within the parallel region. At the end of the parallel region, the threads within the team wait until all other threads in the team have finished before being joined. The main thread resumes serial execution with the statement following the parallel region. The implicit barrier at the end of all parallel regions preserves sequential consistency. More to the point, an executing OpenMP program starts a single thread. At points in the program where parallel execution is desired, the program forks additional threads to form a team of threads. The threads execute in parallel across a region of code called the parallel region. At the end of the parallel region, the threads wait until the full team arrives, and then they join back together. At that point, the original or master thread continues until the next parallel region (or end of the program).

All OpenMP pragmas have the same prefix of `#pragma omp`. This is followed by an OpenMP directive construct and one or more optional clauses to modify the construct. OpenMP is an explicitly parallel programming language. The compiler doesn't guess how to exploit concurrency. Any parallelism expressed in a program is there because the programmer directed the compiler to put it there. To create threads in OpenMP, the programmer designates blocks of code that are to run in parallel. This is done in C and C++ with the pragma used to define a parallel region within an application: use the parallel construct:

```
#pragma omp parallel
```

Now, we will take a few simple examples. When compiled, this code is meant to print a string to standard output console:

```
#include <stdio.h>
int main()
{
   printf("E is the second vowel\n");
}
```

Outputs "E is the second vowel".

Now, we add the compiler directive to define a parallel region in this simple program:

```
#include <stdio.h>
#include "omp.h"
int main()
{
#pragma omp parallel
  {
      printf("E is the second vowel\n");
  }
}
```

With a dual-core processor, this is the output:

```
E is the second vowel
E is the second vowel
```

Now, we include a local variable:

```
#include <stdio.h>
#include "omp.h"
int main()
{
    int i=5;
#pragma omp parallel
  {
    printf("E is equal to %d\n",i);
  }
```

```
}
```

OpenMP is a shared-memory programming model. A good rule that holds in most cases is that a variable allocated prior to the parallel region is shared between the threads. So the program prints:

```
E is equal to 5
E is equal to 5
```

The OpenMP specification includes a set of environmental variables and API functions to enable control over the program. One useful environmental variable is `OMP_NUM_THREADS`, which will set the number of threads to be used for the team in each parallel region. The corresponding API function to set the number of threads is `omp_num_threads()`. If a variable is declared inside a parallel region, it is said to be local or private to a thread. In C, a variable declaration can occur in any block. Such an example is shown below. Included is a call to a function called `omp_get_thread_num()`. This integer function is part of the OpenMP runtime library. It returns an integer unique to each thread that ranges from zero to the number of threads minus one.

```
#include <stdio.h>
#include <omp.h>
int main()
{
  int i= 256; // a shared variable
#pragma omp parallel
  {
    int x; // a variable local or private to each thread
    x = omp_get_thread_num();
    printf("x = %d, i = %d\n",x,i);
  }
}
```

Output

```
x = 0, i = 256
x = 1, i = 256

//note the value of x decrements,
//while the value of i remains the same
```

Synchronization

Synchronization is all about timing. Threads running within a process must sometimes access resources, because the container process has created a handle table where the threads can access resources by a handle identification number. A resource could be a Registry key, a TCP port, a file, or any other type of system resource. It is obviously important for those threads to access those resources in an orderly fashion. It is also obvious that two threads cannot execute simultaneously in the same CRITICAL_REGION. For example, if one thread writes some data to a message queue, and then another thread writes over that data, then we have data corruption. More to the point, we have a race condition: two threads race to execute at a single instance because they (think) appear to be scheduled that way. A race condition results in a serious system crash. So, how does OpenMP handle these issues?

OpenMP has synchronization constructs that ensure mutual exclusion to your critical regions. Use these when variables must remain shared by all threads, but updates must be performed on those variables in parallel regions. The critical construct acts like a lock around a critical region. Only one thread may execute within a protected critical region at a time. Other threads wishing to have access to the critical region must wait until no thread is executing the critical region. OpenMP also has an atomic construct to ensure that statements will be executed in an atomic, uninterruptible manner. There is a restriction on which types of statements you can use with the atomic construct, and you can only protect a single statement. The single and master constructs will control execution of statements within a parallel region so that only one thread will execute those statements (as opposed to allowing only one thread at a time). The former will use the first thread that encounters the construct, while the latter will allow only the master thread (the thread that executes outside of the parallel regions) to execute the protected code.

The OpenMP runtime library is then expressed in compiler directives, but there are certain language features that can only be handled by library functions. Here are a few of them:

- `omp_set_num_threads()` takes an integer argument and requests that the Operating System provide that number of threads in subsequent parallel regions.
- `omp_get_num_threads()` (integer function) returns the actual number of threads in the current team of threads.
- `omp_get_thread_num()` (integer function) returns the ID of a thread, where the ID ranges from 0 to the number of threads minus 1. The thread with the ID of 0 is the master thread.

And, here is code that uses some OpenMP API functions to extract information about the environment:

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

int main (int argc, char *argv[])
{
int nthreads, tid, procs, maxt, inpar, dynamic, nested;

/* Start parallel region */
#pragma omp parallel private(nthreads, tid)
  {

  /* Obtain thread number */
  tid = omp_get_thread_num();

  /* Only master thread does this */
  if (tid == 0)
    {
    printf("Thread %d getting environment info...\n", tid);

    /* Get environment information */
    procs = omp_get_num_procs();
    nthreads = omp_get_num_threads();
    maxt = omp_get_max_threads();
    inpar = omp_in_parallel();
    dynamic = omp_get_dynamic();
    nested = omp_get_nested();

    /* Print environment information */
    printf("Number of processors = %d\n", procs);
    printf("Number of threads = %d\n", nthreads);
    printf("Max threads = %d\n", maxt);
    printf("In parallel? = %d\n", inpar);
    printf("Dynamic threads enabled? = %d\n", dynamic);
    printf("Nested parallelism supported? = %d\n", nested);

    }

  }
}
```

Output

```
Thread 0 getting environment info...
Number of processors = 2
Number of threads = 2
Max threads = 2
In parallel? = 1
Dynamic threads enabled? = 0
Nested parallelism supported? = 0
```

Some More Core Concepts

In certain cases, a large number of independent operations are found in loops. Using the loop worksharing construct in OpenMP, you can split up these loop iterations and assign them to threads for concurrent execution. The parallel for construct will initiate a new parallel region around the single for loop following the pragma, and divide the loop iterations among the threads of the team. Upon completion of the assigned iterations, threads sit at the implicit barrier at the end of the parallel region, waiting to join with the other threads. It is possible to split up the combined parallel for construct into two pragmas: a parallel construct and the for construct, which must be lexically contained within a parallel region. Here is an example of the former:

```
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>
#define CHUNKSIZE   10
#define N       100

int main (int argc, char *argv[])
{
  int nthreads, tid, i, chunk;
  float a[N], b[N], c[N];

/* Some initializations */
for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;
chunk = CHUNKSIZE;

#pragma omp parallel shared(a,b,c,nthreads,chunk) private(i,tid)
  {
  tid = omp_get_thread_num();
  if (tid == 0)
    {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }
  printf("Thread %d starting...\n",tid);

  #pragma omp for schedule(dynamic,chunk)
  for (i=0; i < N; i++)
       c[i] = a[i] + b[i];
    printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
    }

  }  /* end of parallel section */

}
```

So the output would be:

```
Number of threads = 2
 Thread 1 starting...
 Thread 0 starting...
 Thread 0: c[10]= 20.000000
 Thread 1: c[0]= 0.000000
 Thread 0: c[11]= 22.000000
 Thread 1: c[1]= 2.000000
 Thread 0: c[12]= 24.000000
 Thread 1: c[2]= 4.000000
Thread 0: c[13]= 26.000000
Thread 1: c[3]= 6.000000
Thread 0: c[14]= 28.000000
Thread 1: c[4]= 8.000000
Thread 0: c[15]= 30.000000
```

```
Thread 1: c[5]= 10.000000
Thread 0: c[16]= 32.000000
Thread 1: c[6]= 12.000000
Thread 0: c[17]= 34.000000
Thread 1: c[7]= 14.000000
Thread 0: c[18]= 36.000000
Thread 1: c[8]= 16.000000
Thread 0: c[19]= 38.000000
Thread 1: c[9]= 18.000000
Thread 0: c[20]= 40.000000
Thread 1: c[30]= 60.000000
Thread 0: c[21]= 42.000000
Thread 1: c[31]= 62.000000
Thread 0: c[22]= 44.000000
Thread 1: c[32]= 64.000000
Thread 0: c[23]= 46.000000
Thread 1: c[33]= 66.000000

.   .   .   .   .    .    .

Thread 1: c[84]= 168.000000
Thread 1: c[85]= 170.000000
Thread 1: c[86]= 172.000000
Thread 1: c[87]= 174.000000
Thread 1: c[88]= 176.000000
Thread 1: c[89]= 178.000000
Thread 1: c[90]= 180.000000
Thread 1: c[91]= 182.000000
Thread 1: c[92]= 184.000000
Thread 1: c[93]= 186.000000
Thread 1: c[94]= 188.000000
Thread 1: c[95]= 190.000000
Thread 1: c[96]= 192.000000
Thread 1: c[97]= 194.000000
Thread 1: c[98]= 196.000000
Thread 1: c[99]= 198.000000
Thread 0: c[24]= 48.000000
Thread 0: c[25]= 50.000000
Thread 0: c[26]= 52.000000
Thread 0: c[27]= 54.000000
Thread 0: c[28]= 56.000000
Thread 0: c[29]= 58.000000
```

Clauses Used in the Data Environment

We begin by defining the terms we will use to describe the data environment in OpenMP. In a program, a variable is a container (or more concretely, a storage location in memory) bound to a name and holding a value. Variables can be read and written as the program runs (as opposed to constants that can only be read). In OpenMP, the variable that is bound to a given name depends on whether the name appears prior to a parallel region, inside a parallel region, or following a parallel region. When the variable is declared prior to a parallel region, it is by default shared, and the name is always bound to the same variable. OpenMP, however, includes clauses that can be added to parallel and to the worksharing constructs to control the data environment. These clauses affect the variable bound to a name. A private (list) clause directs the compiler to create, for each thread, a private (or local) variable for each name included in the list. The names in the private list must have been defined and bound to share variables prior to the parallel region. The initial values of these new private variables are undefined, so they must be explicitly initialized. Furthermore, after the parallel region, the value of a variable bound to a name appearing in a private clause for the region is undefined.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#define N       50
#define CHUNKSIZE   5
```

```
int main (int argc, char *argv[])
{
  int i, chunk, tid;
  float a[N], b[N], c[N];
  char first_time;

/* Some initializations */
for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;
chunk = CHUNKSIZE;
first_time = 'y';

#pragma omp parallel for     \
  shared(a,b,c,chunk)        \
  private(i,tid)             \
  schedule(static,chunk)     \
  firstprivate(first_time)

  for (i=0; i < N; i++)
  {
    if (first_time == 'y')
    {
      tid = omp_get_thread_num();
      first_time = 'n';
    }
    c[i] = a[i] + b[i];
    printf("tid= %d i= %d c[i]= %f\n", tid, i, c[i]);
  }
}
```

The output is as expected:

```
tid= 0 i= 0 c[i]= 0.000000
tid= 1 i= 5 c[i]= 10.000000
tid= 0 i= 1 c[i]= 2.000000
tid= 1 i= 6 c[i]= 12.000000
tid= 0 i= 2 c[i]= 4.000000
tid= 1 i= 7 c[i]= 14.000000
tid= 0 i= 3 c[i]= 6.000000
tid= 1 i= 8 c[i]= 16.000000
tid= 0 i= 4 c[i]= 8.000000
tid= 1 i= 9 c[i]= 18.000000

.  . .   .  .  and so on

tid= 1 i= 47 c[i]= 94.000000
tid= 0 i= 43 c[i]= 86.000000
tid= 1 i= 48 c[i]= 96.000000
tid= 0 i= 44 c[i]= 88.000000
tid= 1 i= 49 c[i]= 98.000000
```

License

Share

About the Author

logicchild
Software Developer Monroe Community
United States

No Biography provided

Comments and Discussions

You must Sign In to use this message board.

Spacing Relaxed Layout Normal Per page 25 Update

Permalink Layout: fixed | fluid

-- There are no messages in this forum --