PhUSE 2011

Paper CC02

Writing Parallel Processing Compatible Engines Using OpenMP

Aniruddha Deshmukh, Cytel Inc., Pune, India

ABSTRACT

Software for sophisticated statistical procedures usually involves compute-intensive algorithms and large amounts of data. Numerical methods, optimization and simulation requirements all involve massive amount of computing. One way to dramatically improve runtime performance is to exploit multiple CPUs now commonplace even on desktop computers.

OpenMP (Open Multi-Processing) is an open standard for portable and scalable parallel programming with shared memory and distributed shared memory multiprocessors. It provides an API for parallel programming in C/C++ and FORTRAN.
This paper describes some key approaches taken in OpenMP to support the development of parallel applications. This is done with the help of one real-life example deployed with Visual C++. In the example, the benefit of using OpenMP is illustrated. The paper also describes some common pitfalls to look out for and gives some guidelines to new programmers.

Basic familiarity with algorithm development and programming methods is assumed, along with some understanding of processes and threads.

INTRODUCTION

Many software applications in the Clinical Trials domain have at their core, software 'engines' that provide quantitative methods such as statistical analysis, simulation, scenario building and evaluation, business intelligence, and OLAP. Such applications tend to be heavily compute-intensive. They frequently involve a massive number of repetitious computations, or building and traversal of enormous data structures such as trees or networks, or both. This situation offers a perfect case for parallel computing – the technique of exploiting the availability of multiple CPUs on the target machine or network.

Programming the engines for parallel processing requires prior planning to facilitate parallelization by design, and requires supporting hardware and software tools to build in the ability to utilize multiple processors if available.

OpenMP is one of the standards for parallel computing defined by a technology consortium. It is implemented by various C/C++ compilers from vendors like Microsoft, GNU, IBM, Intel etc. We will, however, limit our discussion to the OpenMP implementation supported by Visual C++ on Windows, using Visual Studio 2010 as the development environment.

Although it is currently in version 3.1 (released in July 2011), Visual C++ supports the OpenMP standard 2.0, which we will use in this paper.

OpenMP is a vast topic and cannot be discussed completely in one paper. As such, we will restrict the scope of this paper to discussing one practical example and highlighting the key OpenMP features used in the example. We will end the paper with some guidelines for programmers.

WHY OPENMP?

Parallel computing offers the potential for high performance. Enhanced performance of computer applications is the only practical purpose of parallel processing. To get this benefit, support for parallel processing must be available in

both hardware and software. The recent advances in multi-core and multi-CPU hardware and their wide availability have ensured the necessary hardware support.

In the software, in order to implement parallelism and attain meaningful enhancement in performance, the developer has to invest in additional design and programming complexity. Sometimes, this complexity can become a deterrent to utilizing the full computing power available at the hardware level.

The goal of OpenMP is to provide a portable standard parallel computing API specifically for programming shared memory multiprocessors. It provides support for the three basic aspects of parallel programming: workload division, communicating between threads, and synchronization between threads.

OpenMP offers the following specific advantages:

- Ability to implement parallelism in segments – small segments of the application may be parallelized independently.
- Requires small to moderate increase in code size – increase in code size depends upon extent of changes required for parallel scalability
- Availability of rich application development and debugging tools
- Portability – OpenMP makes use of compiler directives and some simple library calls. The directives are automatically ignored if the compiler does not support OpenMP. The OpenMP API also addresses the portability issue of the library calls in non-OpenMP environments.

OPENMP API

The OpenMP API is a set of compiler directives and a small set of library routines used to express shared memory parallelism. The directives provide sufficient support to program parallel execution threads.The library routines can be used to get finer control over the execution of the threads. Currently, OpenMP is supported in three programming languages: C, C++ and FORTRAN.

The APIs fall into one of three categories: control structures for work division, data environment constructs for communicating between threads, and synchronization constructs for coordinating execution of multiple threads.

When OpenMP support is enabled, a macro called _OPENMP is automatically defined. The use of OpenMP library calls and environment variables can be turned on and off using this macro.

```
#ifdef _OPENMP
        // Statements using OpenMP library calls and environment variables
#endif
```

OpenMP pragmas or compiler directives have the following general syntax:

```
#pragma omp <directive> [<clauses>]
```

The pragmas direct the compiler to apply the directive to sections of code. The directives are either work sharing or synchronization constructs. The clauses are optional modifiers and affect the behavior of the directives.

Detailed discussion of all these directives and their clauses is beyond the scope of this paper. We will restrict our discussion to an example that illustrates the usage of some important directives and clauses and the benefit they offer in terms of improved performance.

EXAMPLE: PARALLELIZING A LARGE SCALE SIMULATION

Simulations of clinical trials, like the simulations of many other real life processes, involve modeling individual steps of that process, coding them in some programming language and repeating them over and over with the objective of studying the operating characteristics of that process in detail. Typically a bulk of computations of the individual simulations are independent of each other i.e. the execution of one simulation is not affected by that of another simulation. This independent nature of the simulations makes them the perfect candidate for parallelization.

The following diagram shows the standard simulation workflow employing parallelism to improve performance.

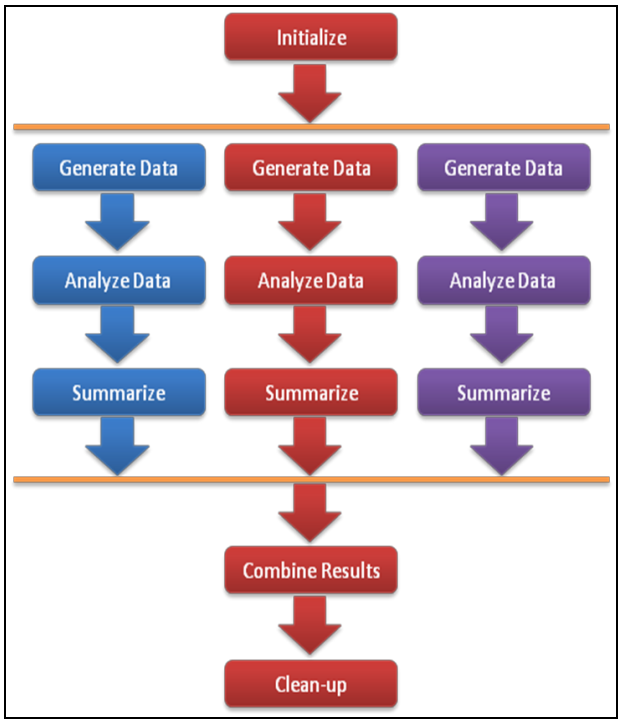

Figure 1: Simulation Workflow

We now present a code snippet that implements this workflow. This code is much simplified than the actual code. We have removed details that are not relevant to our discussion.
This function uses OpenMP to parallelize execution of multiple simulations. We have divided it in various blocks and marked the start and end of these blocks in order to make it easier to explain the logic.

```
// Function to perform specified number of simulations
// lSimCount – total number of simulations to be performed
// lSampleSize – total number of subjects in the simulated trial
void PerformSimulations(long lSimCount, long lSampleSize, char* sDataFile)
{
    // Local variables

    double* pdArrRndNum; // Pointer to array of random numbers generated in one simulation
    long lSeed; // Seed passed to the random number generator
    TRIAL_DATA* pTrialData; // Pointer to object storing data generated in one trial
    SIM_OUT* pSimOut; // Pointer to object storing summarized output

    // Perform any initial processing here like initializing lSeed etc.
    // Now performing the simulations one by one

    #pragma omp parallel for ordered private(pTrialData)
    for(long lSimInd = 0; lSimInd < lSimCount; lSimInd++)
    {
        // <START OF BLOCK 1>
        // Start of a new simulation
        // Perform any initial processing required for this simulation here.

        // Allocate memory to pTrialData.
        // ……..
        // <END OF BLOCK 1>
```

Iterations of this loop are executed by multiple threads in parallel.

```
            // Before generating the data required in this simulation, we need to generate
            // the random numbers.
            // <START OF BLOCK 2>
            #pragma omp ordered
            {
                  // Generating random numbers in the range (0, 1)
                  GenerateRandomNumbers(lSampleSize, pdArrRndNum, lSeed);
            }
            // <END OF BLOCK 2>

            // <START OF BLOCK 3>
            // Generate the subject data using the random numbers generated.
            GenerateTrialData(pTrialData, lSampleSize, pdArrRndNum);
            // <END OF BLOCK 3>

            // <START OF BLOCK 4>
            // Analyze and summarize data.
            #pragma omp critical (SUMMARIZE)
            {
                  AnalyzeAndSummarize(pTrialData, pSimOut);
            }
            // <END OF BLOCK 4>
      }
}

// Function for generating random numbers in the range (0, 1)
void GenerateRandomNumbers(long lCount, double* pdArrRndNum, long& lSeed)
{
      // Generating the random numbers one by one in the following loop
      for(long lNum = 0; lNum < lCount; lNum++)
      {
            // Some logic for generate one random number at a time
            // Store the generated number at pdArrRndNum[lNum].

            // lSeed gets modified in this iteration and the modified value is used in the
            // next iteration.
      }
      // Modified lSeed value is also returned to the caller.
}

// Function to generate trial data
void GenerateTrialData(TRIAL_DATA* pTrialData, long lSampleSize,
                        double* pdArrRndNum)
{
      // Some logic for generating subject by subject data
}

// Function to write generated data to the specified file
// It opens the file in append mode and closes after the data is written.
void AnalyzeAndSummarize(TRIAL_DATA* pTrialData, SIM_OUT* pSimOut)
{
      // Some logic for analyzing the data stored in pTrialData
      // and updating the summary output in pSimOut
}
```

Code in this block is executed sequentially by the threads.

Code in this block is executed by only one thread at a time.

Code Snippet # 1

We will now discuss the OpenMP features used in this code snippet.

THE PRAGMA *OMP PARALLEL FOR*

The use of the pragma *omp parallel for* parallelizes the *for* loop following the pragma. Depending upon the number of processors available, the master thread creates 0 or more child threads. The iterations of the loop are divided

among this team of the master and the child threads. The threads in the team execute in parallel. Thus there is no guarantee that the iterations will happen in an ordered manner as is the case when only one thread executes the entire loop.

There is an implied barrier at the end of the *for* loop. Every thread waits at this implied barrier after it has finished executing its iterations, until all other threads have finished their loops too. Only the master thread continues beyond the loop.

E.g. suppose we are running just 10 simulations and these simulations / loop iterations are divided among 4 threads (1 master and 3 child threads created by the master when the pragma *omp parallel for* is hit). The loop iterations may get divided among the threads in any fashion.

The following table displays one possible division:

Thread	Iterations Assigned
Thread 1 (Master)	5, 6, 7
Thread 2 (Child)	3, 4
Thread 3 (Child)	1, 2
Thread 4 (Child)	8, 9, 10

Table # 1: Possible Assignment of Loop Iterations to Threads

This kind of division creates no problem with the code in blocks numbered 1 and 3. But it would most likely cause problems with the code in blocks numbered 2 and 4 if the pragma *omp ordered* is not used with these blocks.

THE *PRIVATE* CLAUSE

The variable *pTrialData* is declared *private*. It is a pointer to the memory block that holds the data generated in one simulation.

pTrialData is declared outside the *for* loop. If we do not declare it *private*, it will be automatically shared. This will have unintended consequences as all threads in the team will access the same 4 byte memory location that stores the value of the pointer.

Each simulation will allocate memory required for storing the data to be generated and store the address of this memory block into the shared pointer variable, potentially overwriting the value stored there by another thread before.

As a result, when multiple threads try to access the memory blocks allocated by them for storing data through this pointer, they will potentially end up accessing the wrong memory location and this will most likely result in data corruption.

For this reason, it is essential that the variable *pTrialData* be declared *private*. In that case, each thread will get its own 4 byte memory location named *pTrialData* that will store the address of the memory block allocated by it and we will get the intended result.

SYNCHRONIZATION

Let us looks at the blocks numbered 2 and 4.

Block # 2 calls the function GenerateRandomNumbers() which generates the required random numbers. Generation of random numbers is necessarily a sequential task. To generate the random number sequence correctly, it is necessary that the modified value of the random number seed from one random number generation step is passed on to the next step and that this sequence is maintained throughout the various simulations.

To ensure this, the *ordered* directive is used around random number generation. It ensures that the code in the block is executed in sequential order by the threads in the team and not in the order in which the threads reach this block.

In block # 4, we are calling the function AnalyzeAndSummarize() which performs statistical analysis of the data stored in *pTrialData* and updates the variable *pSimOut* using the summarized results from each simulation.

The variable *pTrialData* is private to each thread. But the variable *pSimOut*, is shared among all threads and is subject to manipulation by different threads. If not done properly, this can lead to memory corruption and incorrect behavior at the run time. However, unlike the code from the block # 2, no particular order is required while summarizing the simulation results.

In such cases, using the pragma *omp critical* is useful. It starts a critical section that restricts execution of the associated structured block to one thread at a time. If one thread has already entered the critical section and is executing the code inside, other threads wait at the start of the critical section when they reach there and one of these threads gets an entry into the critical section when the previous thread is finished.

The following figure shows the manner in which the 4 threads from table # 1 above execute. In this figure, I1 through I10 indicate the 10 iterations of the loop and B1 through B4 indicate the 4 code blocks that are executed in each of these iterations.

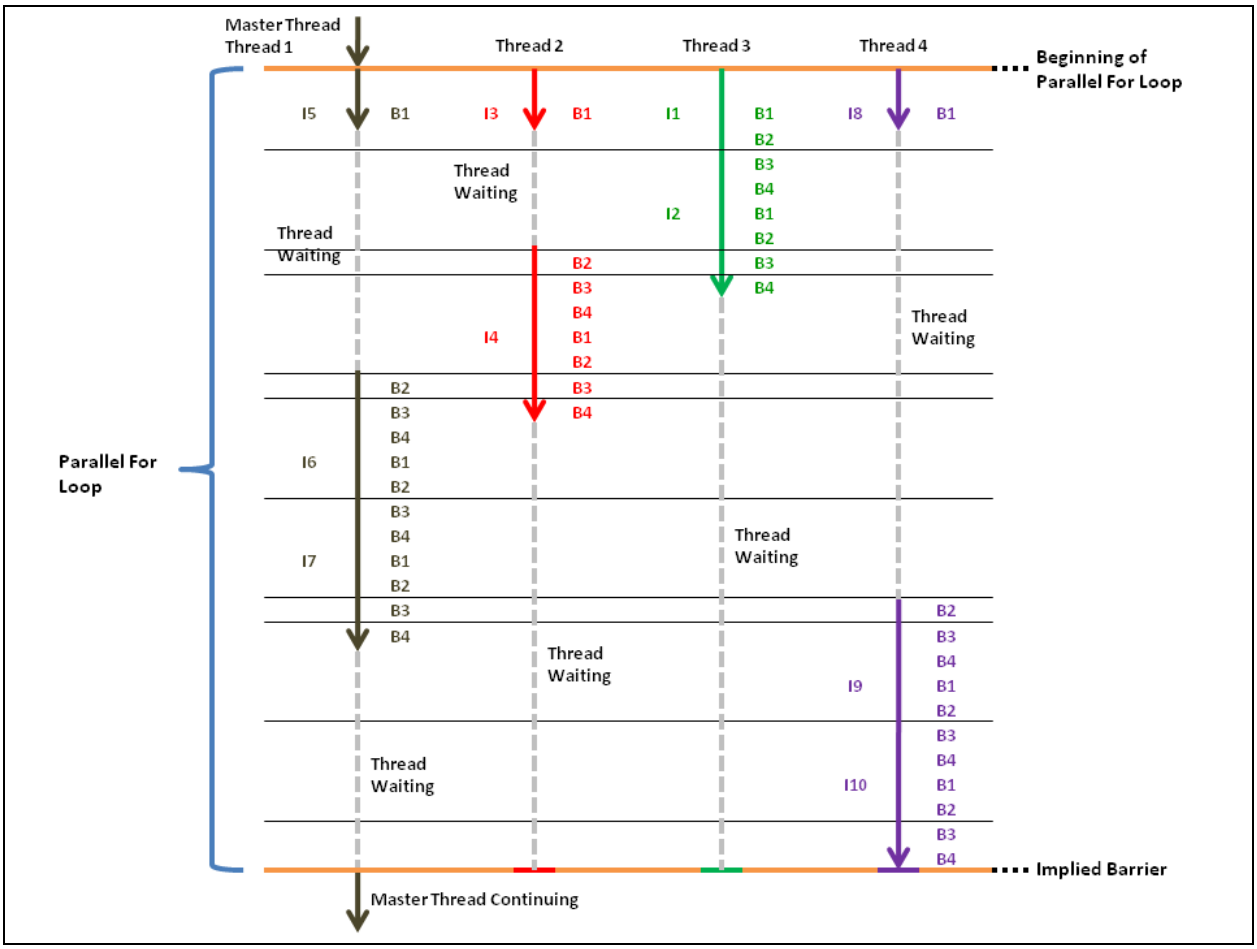

Figure # 1: Order of Loop Execution and States of Threads in Table # 1

RESULTS

In one of our products named SiZ™, we coded simulations of a trial involving generating data for a large number of subjects and then analyzing the data using a multiple comparison procedure (a well known statistical technique). We executed these simulations on a laptop with 3 GB RAM and a quad-core processor with a speed of 2.4 GHz.

We discovered performance improvement of the following order.

Test #	# of Simulations	Sample Size	Time Required		% Improvement with OpenMP
			OpenMP Enabled	OpenMP Disabled	
1	85000	2100	2 mins, 6 secs	6 mins, 40 secs	69%
2	100000	3854	6 mins, 38 secs	14 mins, 48 secs	55%

Table # 2: Performance Improvement with OpenMP

PROBLEMS WHERE PARALLEL COMPUTING COULD BE USEFUL

With parallel computing technologies like OpenMP becoming commonplace, it is now possible to solve problems that took very long to solve in the past. Following are some such problems:

- Large simulations (like those in weather forecasting or the one we discussed in this paper)
- Problems in linear algebra
- Graph traversal
- Branch and bound methods
- Dynamic programming
- Combinatorial techniques

SOME GUIDELINES TO THE PROGRAMMER

- In case of a parallel region, there can be no entry into the block other than from the beginning and no exit from the block other than at the end. Else the program is non-conforming and may crash or produce unspecified results.
- There should not be any interdependence between the threads created by a *parallel* directive.
- In case multiple clauses are used with a directive, their order of evaluation is unspecified. There should not be any dependence on the order of their evaluation.
- If exceptions are used, an exception thrown by one thread must be caught by the same thread. Otherwise the program may crash.
- In case of parallel loops, there must not be any loop carried dependencies.
- In case of nested *for* loops, the loop variable of the outermost loop is *private* by default whereas those of the inner loops are shared by default. The *private* clause can be used with these variables so that they are not shared.
- In case of nested loops, parallelizing the loop that performs the most work gives best results.
- Public members of classes are shared. Critical regions should be used while updating public members.
- In case of *parallel* sections, the sections must be independent of each other and access of shared data must be synchronized. Otherwise the results are undefined.
- OpenMP is poor in error handling. Programs written with OpenMP need intensive testing.
- It is difficult to identify problems arising because of non-conformance to the rules. These problems do not surface on single processor machines as OpenMP is disabled there. On multiprocessor machines, the problems surface and are hard to debug.
- OpenMP is not supported with Profile Guided Optimization by the compiler.
- OpenMP is ideal for work like large array / list processing, heavy compute intensive loops (like in case of some simulations) etc. If the workload is not adequate, performance may even degrade because of using OpenMP.

CONCLUDING REMARKS

OpenMP is a simple yet powerful technology for parallelizing applications. It provides ways to parallelize data processing loops as well as functional blocks of code. It can be integrated easily into existing applications and turned on or off simply by throwing a compiler switch. OpenMP is an easy way to tap into the processing power of multicore CPUs. However, one must read and understand the specs carefully and test the code thoroughly on single processor and multiprocessor systems.

REFERENCES

- Kang Su Gatlin, Pete Isensee, OpenMP and C++: Reap the Benefits of Multithreading without All the Work, *MSDN Article, URL: http://msdn.microsoft.com/en-us/magazine/cc163717.aspx*
- Rohit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, Ramesh Menon, Parallel Programming in OpenMP, *Morgan Kaufmann Publishers*
- OpenMP Website: http://www.openmp.org for the complete OpenMP specification

CONTACT INFORMATION

Your comments and questions are valued and encouraged. You can contact the author at:

Aniruddha Deshmukh, aniruddha.deshmukh@cytel.com, Phone: +91-20-66040170

Cytel Statistical Software & Services Pvt. Ltd., 8th Floor, Siddharth Towers, Off Karve Road, Kothrud
Pune 411029, INDIA