OpenMP Tasking Analysis for Programmers

Xavier Teruel(1)  Christopher Barton(2)  Alejandro Duran(1)  Xavier Martorell(1)
Eduard Ayguadé(1)  Priya Unnikrishnan(2)  Guansong Zhang(2)  Raul Silvera(2)

(1) Barcelona Supercomputing Center – Universitat Politècnica de Catalunya
C/ Jordi Girona, 31 - 08034 Barcelona (Spain)
{xteruel, aduran, xavim, eduard}@ac.upc.edu

(2) IBM Toronto Laboratory
8200 Warden Avenue, Markham, Ontario L6G 1C7 (Canada)
{kbarton, priyau, guansong, rauls}@ca.ibm.com

Abstract

As of 2008, the OpenMP 3.0 standard includes task support allowing programmers to exploit irregular parallelism. Although several compilers are providing support for this new feature there has not been extensive investigation into the real possibilities of this extension. Several papers have discussed the programming model itself while other papers have discussed design and implementation on different platforms. There are also papers demonstrating performance results using well known kernel applications.

This paper presents an analysis of the OpenMP tasking model possibilities, using the IBM XL compiler implementation. Using different parameters such as the number of tasks, task granularity and parallelism pattern, this paper explores how such parameters can affect the average performance and identifies the limits of the OpenMP tasking model.

1 Introduction

OpenMP [3] appeared in 1997 as a parallel programming model for shared-memory environments, jointly defined by a group of major computer hardware and software vendors. It is a simple and flexible interface for developing parallel applications through a directive based programming model. If these directives are ignored by the compiler, the sequential execution of the program is preserved. The simplicity and ease of use are reasons for its wide acceptance as a de facto standard for shared-memory parallel programming.

The most significant feature in the latest OpenMP specification (OpenMP 3.0 [7]) is the tasking model [4]. The tasking model supports unstructured parallelism by providing two new directives: task and taskwait. The task directive allows the user to identify independent work units, and the taskwait directive is a synchronization primitive to synchronize the execution of tasks.

Although several compilers are providing support for this new feature [1] there has not been extensive investigation into the real possibilities of this extension. Several papers have discussed the programming model itself [4, 5] while other papers have discussed design and implementation on different platforms [17, 18].
There are also papers demonstrating performance results using well known kernel applications [6, 11].

In this paper, we focus on the evaluation of the OpenMP tasking model using the IBM XL compiler suite. We use three kernel benchmarks and several synthetic benchmarks created with the purpose of testing task parallelism. Using different parameters such as the number of tasks, task granularity and parallelism pattern, this paper explores how such parameters can affect the average performance.

The rest of the paper is organized as follows: Section 2 describes the related work in this area. Section 3 presents an OpenMP tasking parallelism analysis. Section 4 summarizes some conclusions and finally, Section 5 discusses the future work.

2 Related Work

The EPCC microbenchmark suite [9] has been used in many publications to measure overheads of synchronization and loop scheduling in OpenMP. The synchronization benchmark measures the overhead in work-sharing and mutual exclusion directives. The loop scheduling benchmark compares and measures the overhead introduced by the scheduling policies (STATIC, DYNAMIC, and GUIDED) available with OpenMP. While there are several papers and discussions derived from such benchmarks, most of these studies focus on measuring the performance of a specific OpenMP infrastructure [14, 15] or include additional extensions [8] that complement the existing EPCC suite.

There are also several papers specifically related with the OpenMP tasking model. However, their approach is focused in evaluating the programming model expressivity [4, 5], discussing implementation details [17, 18], or proposing new techniques which improve user application performance [12, 10].

3 Tasking Analysis

This paper builds on work previously published in [18]. We want to explore how parameters in the task model impact performance. To achieve this, we focus on three kernel benchmarks, SparseLU, Multisort and Fibonacci as well as several synthetic benchmarks.

The SparseLU benchmark computes an LU matrix factorization over sparse matrices. Due to the sparseness of the matrix, work-sharing solutions have to deal with a lot of load imbalance. In our experiments, the matrix size is set at 5000x5000, submatrix sizes are set at 100x100 float elements and the complete matrix has an sparseness factor of 0.15 (15% of the matrix elements have a non-zero value).

Multisort is a variation of the mergesort algorithm. It uses a parallel divide and conquer mergesort [2] and a serial quicksort [16] when the array is small. In our experiments we use array size set to 128 MB.

Fibonacci computes the $n$th fibonacci number using a recursive parallelization. Although the benchmark is not representative of parallel computation it is a simple test case of a deep tree composed of very fine grain tasks which is interesting for our study purpose. We use this benchmark to investigate how limiting the task generation impacts performance.

We also use several synthetic kernels to measure the execution time for different data sharing attributes. We can specify the number of tasks (TASKS), task granularity (GRAIN), and delay between task creation (DELAY). We test these benchmarks using two different scenarios:

1. In a parallel/for scenario with multiple threads creating tasks (using #pragma omp for) and multiple threads executing them. These tests are labeled as “P/F” in following figures.

2. In a parallel/single scenario with one thread creating tasks (using #pragma omp single) and multiple threads executing them. These tests are labeled as “P/S” in following figures.

Figures 1 and 2 show the sequential code and the corresponding parallel code using tasks. The Data Sharing Attribute (DSA) on line 5 of the synthetic benchmark in Figure 2 can be one of (SHARED, PRIVATE and FIRSTPRIVATE).

All the experiments were performed on an IBM POWER5 machine with 64 SMT processors running at 1656 MHz, 512 GB memory and
executing AIX 5.3. Benchmarks were compiled using a development version of the IBM XL C/C++ Enterprise Edition for AIX with -O3 and -q64 flags for both the sequential and parallel versions. In addition, the parallel version is compiled with the -qomp=omp flag. At runtime, thread binding is enabled to bind every other processor using the environment variable XLSMPOPTS=startproc=0:stride=2 and idle threads are forced to busy-wait by setting XLSMPOPTS=spins=0:yields=0. The number of threads is controlled using the OMP_NUM_THREADS environment variable.

Figure 3 presents the speedup obtained for the SparseLU and Multisort benchmarks, using sequential execution time as the baseline. The sparseLU shows a linear speedup up to 44 threads before reaching saturation. This benchmark will be discussed in more details in following sections. On the other hand, the multisort kernel shows a quite poor speedup performance.

As we already comment in our previous work [18], the task generation pattern in the Multisort benchmark seems to be the likely cause for this poor performance. Figure 4 shows the task generation for the multisort kernel. Multisort uses a recursive algorithm for splitting the work among descendants tasks and merging results after the work is done. Each task also executes two task synchronization primitives (taskwait) which can also have an impact on the application performance.

In order to further analyze task parallelism and recursion we have included the Fibonacci benchmark which will be discussed in following sections.
3.1 Profiling benchmark execution

Figure 5 shows the execution of the SparseLU benchmark using the tprof profiling tool. As the number of processors increase we can see that the percentage of time spent in the runtime library and kernel execution time also increases. A closer analysis of the time spent in the runtime library reveals that more time is spent in locking routines executed by threads when they attempt to take work from the ready-queue.

On the other hand, the kernel execution time is almost completely consumed by the kernel function .check.lock (see Figure 6). This function is necessary for critical regions, as is the case for memory allocation. Due to the sparseness of the SparseLU benchmark, a block of the matrix is only allocated when it is needed, and in our case it is needed inside the parallel region (when tasks are being executed). The effect of this parallel allocation is the cause of the saturation phase we observe with more than 44 processors.

By modifying the benchmark to pre-allocate the whole matrix in the sequential part of the code we avoid the use of memory allocation functions inside the parallel region. Figure 7 shows a comparison of the two benchmark versions, demonstrating that the use of this type of functions inside a parallel region can reduce the average performance.

As the sparseness of the SparseLU benchmark is motivated by the reduction of memory consumption we think preallocating all the matrix blocks outside the parallel region is not a solution. Nevertheless, programmers must take into account the tradeoffs between memory consumption and execution time when developing parallel applications.
3.2 Spawning parallelism with tasks

There are three different factors that can impact the creation of parallelism in OpenMP Tasks:

- The scenario.
- The granularity.
- The number of tasks.

The scenario is the parallel region where the tasks are created. Although multiple scenarios can exist we will summarize them in two main schemes:

- Parallel/Single scenario.
- Parallel/For scenario.

In the first case a single thread is responsible for generating all of the tasks. In this case we have a situation with a single producer and multiple consumers (i.e., multiple threads are waiting to execute the tasks being generated). In the second case tasks are generated in parallel by all of the threads in the parallel region and all of the threads are also responsible to execute the generated tasks. This case is caused by the existence of a parallel loop and creates a situation with multiple producers and multiple consumers. Other models can exist including hybrid scenarios. The recursion model in task creation will be discussed in a following section.

Figure 8 shows the speedup obtained using the synthetic benchmark for task parallelism creation. We observe that for 16 to 36 processors the parallel single construct behaves slightly better than parallel for construct. In this scenario there is a single thread generating tasks and thus inserting tasks in the ready queue pool while other threads are picking up work from the same queue. In the parallel for scenario multiple threads are generating tasks at the same time which seems to cause some contention on the pool. For more than 36 processors the parallel for scenario performs much better than the parallel single. This is because just one thread is not able to generate enough work to keep the remaining threads busy. The amount of generated work depends on the number of tasks and task granularity.

Task granularity is the amount of work that a single task executes. In our synthetic benchmark we measure this parameter in milliseconds. Figure 9 shows how different task granularities effect the parallel single scenario.
Results show that as the task granularity increases the benchmark performance also increases. Thus, granularity is an important factor programmers must take into account when designing parallel codes.

The number of tasks combined with the task granularity determines the amount of work executed by the parallel region. The number of tasks can affect load balance. An insufficient number of tasks prevent an adequate distribution of work among threads, leading to starvation. On the other hand, increasing the number of tasks allows the scheduler to easily distribute the work among all threads. Figure 10 shows that by increasing the number of tasks the application performance improves slightly for 24 to 36 processors. Figure 10 also shows the parallel single and granularity problems (previously discussed) when the number of processors exceeds 40.

As the amount of work is determined by the program itself (by increasing the number of tasks we reduce the granularity and vice versa), programmers have to reach a trade-off between the number of tasks and the task granularity.

3.3 Data sharing attributes

Data sharing attributes also can impact the application execution time. In OpenMP, private and firstprivate variables become local variables for the new task, but in the case of the firstprivate the local variable must be initial-
ized. In our task implementation, firstprivate variables are initialized at task creation and also require an extra heap allocation [18]. Such factors have an impact on the execution time.

Figure 11 shows a comparison of the synthetic benchmark using the same parameters but a different data-sharing attribute. It compares the speedup between the synthetic benchmark using a private variable and a firstprivate variable. Both benchmarks present the granularity problem discussed in the previous section but in the firstprivate case we observe a performance degradation above 40 processors while in the private case the degradation occurs above 48 processors.

Using the same example but increasing granularity up to 4 msec we observe that firstprivate has a lower impact on the execution time. Figure 12 shows the results obtained in such circumstances and we can observe the results for private and firstprivate are practically the same.

Programmers also have to pay attention to OpenMP data-sharing attributes defaults. According with the OpenMP 3.0 specification:

"In a task construct, if no default clause is present, a variable that is determined to be shared in all enclosing constructs, up to and including the innermost enclosing parallel construct, is shared. [...] In a task construct, if no default clause is present, a variable whose data-sharing attribute is not determined by the rule above is firstprivate"

3.4 Recursion and granularity

Granularity can also be managed in recursive applications. The basic mechanism to increase granularity in such recursive codes is through cutting off generation of tasks by level. In Figure 13 we can see the Fibonacci algorithm annotated with OpenMP tasking directives. The algorithm has been modified in order to generate tasks up to a given depth. When the level reaches the cutoff threshold no more tasks are created under this branch of recursion. In Figure 14 we can see how tasks are generated when fixing cutoff at 2.

```
int Fibonacci(int n, int level)
{
    int a,b;
    if (n < 2) return n;
    if (level < cutoff){
        #pragma omp task
        a=Fibonacci(n-1, level+1)
        #pragma omp task
        b=Fibonacci(n-2, level+1)
        #pragma omp taskwait
    }else{
        a=Fibonacci(n-1, level+1)
        b=Fibonacci(n-2, level+1)
    }
    return (a+b);
}
```

Programmers also have to pay attention to OpenMP data-sharing attributes defaults.

```
result = Fibonacci(5,0);
```

![Figure 13: Cutting off tasks in Fibonacci.](image)

```
Figure 13: Cutting off tasks in Fibonacci.
```

```
result = Fibonacci(5,0);
```

![Figure 14: Fibonacci task creation (using cutoff=2).](image)

```
result = Fibonacci(5,0);
```

![Figure 14: Fibonacci task creation (using cutoff=2).](image)
Figure 15: Fibonacci speedup (based on different cutoff levels).

In this paper we have presented some insights for programming with the OpenMP programming model. We focused on the OpenMP tasking model which allows programmers to deal with irregular parallelism but. However, in order to achieve good performance, in our opinion, also needs to take into account some parallelization details. Table 1 summarizes DOS and DON’TS we have already presented in the previous section.

One of the first issues we have presented is to avoid critical functions in a parallel region. Specifically we have discussed the case of malloc, but other functions that require some mutual exclusion are can have similar effects and should be avoided. Profiling the code is a good method to detect these cases.

We have also observed that for a parallel scenario, multiple produces scale better than a single producer. This is due to the increased level of parallelism achieved by parallelizing the task creation in addition to task execution.

We also commented on the tradeoffs between task granularity and the number of tasks. In particular, we observed that tasks must have sufficient granularity and the granular-ity should increase as the number of consumer threads increases. This ensures that all threads are kept busy doing useful work. The number of tasks also have an effect on the load balance. Programmers have to trade-off between number of tasks and granularity in order to get a proper load balance and thus a good performance.

Programmers should avoid the use of firstprivate variables if they are not needed. Firstprivate variables require a copy to initialize the variable and, in most of the cases, an extra memory allocation.

And finally, we have seen how to manage granularity in recursive applications. It can be achieved by applying a cutoff based on the recursion level.
Table 1: Summary of Dos and Don’ts when programming with OpenMP Tasks.

<table>
<thead>
<tr>
<th>DOS</th>
<th>DON'TS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallelize task creation (if possible), as well as task execution.</td>
<td>In parallel regions, do not use services potentially containing locks or critical sections (e.g. memory allocation).</td>
</tr>
<tr>
<td>Use appropriate task granularity / number of tasks ratio based on the number of threads.</td>
<td>Do not use firstprivate variables when they are not required.</td>
</tr>
<tr>
<td>Control task granularity (and number of tasks) in recursive algorithms by cutting off based on the recursion level.</td>
<td>Do not try to exploit fine granularity tasks, as the overhead of management would grow too much.</td>
</tr>
</tbody>
</table>

5 Future Work

With respect the implementation of tasking, there are several improvements that we are considering. First, whether using local per-processor ready queues can benefit the execution by reducing the contention overhead. Also, we will study better the scalability of the applications when reaching 128 CPUs.

We will investigate the possibilities to incorporate precedences among tasks, in such a way that task graphs can be built at runtime and tasks scheduled accordingly when their data precedences from previous tasks get satisfied.

In the near future, our goal is to do a more detailed performance evaluation through a multiple compilers/architectures comparison and using a wider set of applications. Our goal is to continue with the performance evaluation of our tasking implementation with other benchmarks and applications. Also to get detailed measurements about the tasking behavior in non-numerical and/or irregular applications [13]. We also plan to compare different systems using such a common set of benchmarks, and determine if we can further improve our implementation.

Acknowledgments

This work has been supported by the IBM Center for Advanced Studies (CAS), Toronto, the BSC-IBM MareIncognito project, the European Commission in the context of the SARC project (contract no. 27648), the HiPEAC-2 Network of Excellence (FP7/ICT 217068), the Spanish Ministry of Education (contract no. TIN2007-60625), and by the Generalitat de Catalunya (2009-SGR-980).

About the Authors

Xavier Teruel received his B.Sc. (2003) and his M.Sc. (2006) in Computer Science at Universitat Politècnica de Catalunya (UPC). Currently he is a Ph.D. student at Computer Architecture Department from the same university. Xavier’s research interests include shared memory environments and parallel programming models. Since 2006 he is working as a researcher within the Parallel Programming models team in the Computer Sciences Department at Barcelona Supercomputing Center (BSC).

Dr. Christopher Barton works with the TPO Development group at the IBM Toronto Laboratory. He leads the SMP group, which focuses on compiler optimizations for automatic parallelization as well as explicitly parallel programs. He is also interested in high performance computing with a focus on languages and compilers for large scale machines. He received his Ph.D. in Computing Science from the University of Alberta (2009) and his M.Sc. also from the University of Alberta (2003).

Dr. Alejandro Duran received his Computer Engineering degree (2002) and Ph.D. (2008) from Universitat Politècnica de Catalunya (UPC). Currently he is a Senior Researcher at Barcelona Supercomputing Center (BSC). Previously, he was an assistant professor at UPC. Alex’s research interests include parallel environments, programming languages, compiler optimizations and operating systems.

Dr. Xavier Martorell received the M.Sc.
and Ph.D. degrees in computer science from the Universitat Politècnica de Catalunya (UPC) in 1991 and 1999, respectively. Since 1992 he has lectured on operating systems, parallel runtime systems and OS administration. He has been an associate professor in the Computer Architecture Department at UPC since 2001. His research interests cover the areas of operating systems, runtime systems, compilers and applications for high-performance multiprocessor systems. He spent one year working with the BG/L team in the IBM Watson Research Center. He is currently the Manager of the Parallel Programming Models team in the Computer Sciences Department at the Barcelona Supercomputing Center (BSC).

Dr. Eduard Ayguadé received the Engineering degree in Telecommunications in 1986 and the Ph.D. degree in Computer Science in 1989, both from the Universitat Politècnica de Catalunya (UPC). Since 1987 he has been lecturing on computer organization and architecture and parallel programming models. Currently, and since 1997, he is full professor of the Computer Architecture Department at UPC. He is currently associate director for research on Computer Sciences at the Barcelona Supercomputing Center (BSC). His research interests cover the areas of multicore architectures, and programming models and compilers for high-performance architectures.

Priya Unnikrishnan is a Staff Software Engineer in the Compiler Group at the IBM Toronto Laboratory since 2003. She works on the IBM XL compilers focusing on OpenMP and automatic parallelization. Her areas of interest are parallel computing, parallelizing compilers, tools and multi-core architectures. She represents IBM at the OpenMP Language Committee. She received an MS degree in Computer Science and Engineering from The Pennsylvania State University in 2002.

Dr. Guansong Zhang is a Staff Software Engineer in the Compiler Group at the IBM Toronto Laboratory since 1999 in charge of OpenMP implementation and performance improvement for Fortran and C/C++ on PowerPC and Cell architecture and Performance related compiler optimization techniques, including array data flow analysis, loop optimization and auto parallelization. Previously, he was a research scientist at NPAC center at Syracuse University. He received his Ph.D. from the Harbin Institute of Technology in 1995.

Raul Silvera is a senior developer at the IBM Toronto Lab. After joining IBM in 1997, he has focused on compilation technology and made contributions in the areas of loop transformation, locality analysis, profile-directed optimization, automatic and user-directed parallelization, interprocedural optimizations and others. He is currently the technical lead for TPO, the mid-level optimizer used in the IBM XL Family of compilers. He has participated on the development of the OpenMP standard, and also on the ISO C++ Language standard committee.

References


on Languages and Compilers for Parallel Computing (LCPC), October 2007.


