TECHNICAL REPORT 03/2007

A Flexible and Portable Programming Model for SMP and Multi-cores

BSC-UPC

COMPUTER SCIENCES PROGRAM

Josep M. Perez
Rosa M. Badia
Jesus Labarta

A flexible and portable programming model for SMP and multi-cores

Josep M. Perez, Rosa M. Badia and Jesus Labarta*

June 18, 2007

Abstract

Parallel programming on SMP and multi-core architectures is hard. In this paper we present a programming model for those environments based on function level parallelism that strives to be easy to program for, flexible and portable. We first present the programming environment from the programmer perspective and show its benefits compared to other programming models. Then we show the performance related features of the supporting runtime, which further increase the advantages of the programming model.

1 Introduction

Current chip fabrication technologies allow to place several million transistors in a chip, enabling more complex designs each time. However, there are several issues that discourage the design of more complex uniprocessors: the increase in heat generation, the diminishing instruction-level parallelism gains, almost unchanged memory latency, the inherent complexity of designing a single core with a large number of transistors and the economical costs derived of this design. For these reasons, the current trend on chip manufacturing is to place multiple slower processor cores (multi-core) on a chip.

As a recent published Berkeley report [ABC+06] describes, in earlier times performance improvements have often been achieved by simply running applications on new generations of processors with minimal additional programming effort. While current chips have up to 8 cores, this trend may lead in the future to chips with as much as 1000 cores (many-cores). The report observes that current programming methodologies will have to drastically change as just recompiling and running the current sequential programs will no longer work. Applications are now being required to harness much higher degrees of parallelism in order to exploit the available hardware and to satisfy their growing demand for computing power. This is seen by many as a real revolution in computing.

Examples of current multi-core chips are several dual-core processors like the AMD Opteron or Athlon, the Intel Smithfield or Montecito, or the IBM Power4 or Power5. More challenging architectures are for example the Cell/B.E. processor designed by IBM, Sony and Toshiba, with nine cores (and heterogeneous)

*Barcelona Supercomputing Center - Centro Nacional de Supercomputación (BSC-CNS)

or the Niagara by Sun with eight cores, each of them being able to handle four threads. Even more, Intel recently announced the design of a research prototype with 80-core processors and a capacity of more than a trillion floating operations per second and using less electricity than a modern desktop chip. The chip is modularly designed and each tile has its own router built in the core, creating a network on a chip.

With such a perspective, the availability of suitable programming environments (i.e., compilers, communication libraries, and tools) offering a human-centric approach to exploiting parallelism will become essential for the programming productivity of multi-core systems.

In this paper, we present SMP superscalar (SMPSs), a programming environment focused on the ease of programming, portability and flexibility that is based on Cell superscalar (CellSs)[BPBL06]. While CellSs is tailored for the Cell/B.E. processor, the solution we present is tailored for multi-cores and Symmetric Multiprocessors (SMP) in general.

A SMPSs program is a sequential program annotated with pragmas that identify functions in the code that are candidates to be run in parallel in the different cores of the chip. The only requirement we place in these functions is that they must not have any collateral effects (only local variables and parameters can be accessed). At execution time, the SMPSs runtime detects the data-dependencies that exist between different instances of the annotated subroutines (called tasks from now on) and builds a task dependency graph. This dependency graph is then used for exploiting the inherent concurrency of the application by scheduling non-dependent tasks in the different processors.

The SMPSs programming environment consists of a source to source compiler and a supporting runtime library. The compiler translates C code with the aforementioned annotations into common C code with calls to the supporting runtime library. Then it compiles the resulting code using the platform C compiler. The details about the language syntax and the compiler can be found in [BPBL06].

This paper is organised as follows. In section 2 we present the programming model and the ideas behind it. Then, on section 3 we discuss about the runtime features that support and enhance the programming model. Finally, section 4 is devoted to conclusions and future work.

2 SMP superscalar Programming Model

2.1 Task Based Programming

SMP superscalar is a programming environment for parallel applications based on function level parallelism. In this model, the programmer selects a series of functions called *tasks* that will run in parallel. These functions are treated by the runtime as the unit of parallel computation.

Tasks are defined with a pragma annotation right before their function definition. The specific syntax of all the language directives is the same as for CellSs[BPBL06]. Task annotations indicate that the following function is a task and specify the directionality of each of the task parameters. Figure 1 shows two simple task definitions.

By combining the addresses and directionality of each parameter with those

```
#pragma css task input(A, B) inout(C)
void block_macc(double A[N][N], double B[N][N], double C[N][N])
{
  int i, j, k;
  for (i=0; i < N; i++)
    for (j=0; j < N; j++)
      for (k=0; k < N; k++)
        C[i][j] += A[i][k] * B[k][j];
}

#pragma css task input(A) inout(B)
void block_acc(double A[N][N], double B[N][N])
{
  int i, j;
  for (i=0; i < N; i++)
    for (j=0; j < N; j++)
      B[i][j] += A[i][j];
}
```

Figure 1: A task consisting of a matrix block multiplication with accumulation and an accumulation task.

of previous task invocations, the runtime is capable of analysing the dependencies at run time. This is a major difference between our work and other programming models like OpenMP workqueue extensions[SHPT00]. It removes the effort of analysing the data dependencies from the programmer and moves it into the supporting runtime library. Moreover, the runtime is aware of the data dependencies with enough detail that it can take advantage of them instead of being limited by them.

Figure 2 shows a simple section of code that invokes the *block_macc* and *block_acc* tasks shown previously in figure 1. While this code is simple, it has some data dependencies that would had required manual handling by the programmer under another programming models.

The actual task dependency graph when $N = 2$ is shown in figure 3. The tasks are labelled with their order in sequential order. The upper and middle row of the graph correspond to the *block_macc* tasks, while the lower row corresponds to the *block_acc* task. Note that even if the task invocations of the first double nested loop have dependencies between themselves (task 1 and 2, 3 and 4, 5 and 6, 7 and 8), the graph is capable of representing parallelism beyond the first two iterations of that loop.

While this code is simple and straightforward under a sequential point of view and has a good level of parallelism under our programming model, it is not adequate for data parallel and workqueue programming models. Under those models, the data dependencies in the code would prevent its parallelisation unless the code was transformed by switching loops. Furthermore, those programming models also require additional synchronisation points, which would also reduce their performance when the number of tasks of each loop is not a multiple of the number of processors used.

This last problem can be aleviated by performing loop merging, which again

```
int i, j, k;

for (i=0; i < N; i++)
  for (j=0; j < N; j++)
    for (k=0; k < N; k++)
      block_macc(bigA[i][k], bigB[k][j], bigC[i][j]);

for (i=0; i < N; i++)
  for (j=0; j < N; j++)
    for (k=0; k < N; k++)
      block_macc(bigD[i][k], bigE[k][j], bigF[i][j]);

for (i=0; i < N; i++)
  for (j=0; j < N; j++)
    block_acc(bigC[i][j], bigF[i][j]);
```

Figure 2: Simple code with tasks operating with matrix blocks that has some dependencies.

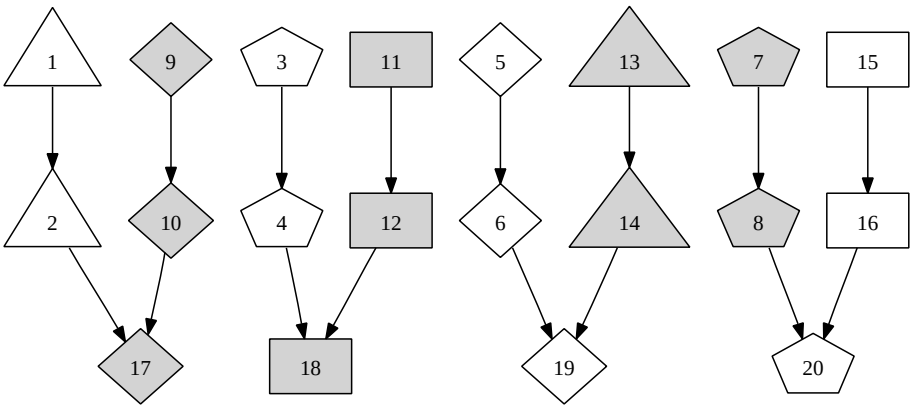

Figure 3: Task dependency graph for one execution of the code shown in figure 2 with $N = 2$. Lables corespond to execution in sequential order. Shade and shape combination corresponds to CPU that has been executed on.

```
int i, j, k;

for (k=0; k < N; k++)
  #pragma omp parallel taskq
  for (i=0; i < N; i++)
    for (j=0; j < N; j++) {
      #pragma omp task
      block_macc(bigA[i][k], bigB[k][j], bigC[i][j]);
      #pragma omp task
      block_macc(bigD[i][k], bigE[k][j], bigF[i][j]);
    }

#pragma omp parallel taskq
for (i=0; i < N; i++)
  for (j=0; j < N; j++)
    #pragma omp task
    block_acc(bigC[i][j], bigF[i][j]);
```

Figure 4: Taskqueue version of the same program after performing loop switching and loop merging.

forces the programmer to rearrange the program in terms of dependencies and still is not able to eliminate all synchronisation points. Figure 4 shows the equivalent code for the workqueue model from [SHPT00] after applying loop switching and loop merging.

2.2 Partial Synchronisation Points

While the underlying runtime is capable of handling all inter-task related data dependencies, it cannot handle dependencies with non task code. The best way to handle those cases is to create new tasks that encapsulate the relevant code, so that the runtime can take care of those dependencies. However, in some cases this is not possible or desirable, for example at the end of the program when writing the results of the whole execution to a file. Those cases can take advantage of synchronisation points.

Synchronisation points are partial kind of barrier. In SMP superscalar, they are associated to particular data that is going to be accessed. After a synchronisation point has been crossed, inline code is guaranteed to have all dependencies with the specified data resolved. In this regard, synchronisation points are partial, since they wait for specific values instead of waiting for all tasks to finish. Figure 5 shows synchronisation code for writing the results of the code from figure 2 to a file.

```
int i, j;

for (i=0; i < N; i++)
  for (j=0; j < N; j++) {
    #pragma css wait on (bigF[i][j])
    block_writeOut(bigF[i][j], outputFile);
  }
```

Figure 5: Code showing a synchronisation point.

3 Runtime Features

3.1 Data Dependency Control

Tasks in a program operate on data that is generated and consumed from task to task. These relations define a certain control flow that must be respected in order to execute the tasks and obtain the same results as in their corresponding sequential execution. SMP superscalar guarantees the consistency of the results by respecting the data dependencies between tasks. Dependency information is generated and kept in a task dependency graph at run time.

Task dependencies are calculated by analysing the direction (input, output or both), length and address of each parameter against those of previous tasks in sequential order. There are three kinds of data dependencies. Read after Write dependencies (RaW) are those between a task that reads data and the task that has written it. Write after Write (WaW) dependencies are those between a task that writes to a data location and a task that has previously written to it. Finally, write after read (WaR) dependencies are those between a task that writes to data and another that read it earlier.

The runtime is structured in a main thread that runs the non task code and populates the graph and a series of worker threads that consume and execute the tasks from the graph. As tasks are executed, their output parameters becomes available to the tasks that were dependant on them.

3.2 Data Dependency Reduction

Dependencies are one of the factors that determine how much parallelism can be extracted out of an application. Superscalar processors try to reduce dependencies between instructions by performing register renaming[SS95]. This technique is also used by SMP superscalar at run time in order to reduce task dependencies and increase the parallelism.

The renaming technique consists in storing temporary definitions of a program variable into temporary storage. That is, if a task writes to an array, renaming can replace that array by a temporary one and redirect all following reads of that definition to the temporary array. This effectively eliminates all WaR and WaW dependencies.

One clear disadvantage of renaming is that it increases the amount of memory usage. In order to limit the effects on memory usage, whenever a parameter is used as both input and output in a task and its input definition is not used by any other task, we do not rename it and instead allow its output definition

to reside in the same memory location as its input definition.

This enhancement reduces memory usage considerably in applications that perform many accumulations on parameters. This is the case of the dense blocked matrix multiplication, in which this technique avoids renaming altogether and allows all calculations to be performed in place.

3.3 Workload Distribution

One of the goals of SMP superscalar is to provide good performance. In these sense, the scheduling algorithm is designed according to three principles. First, trying to maximise the paralleelism. Second, trying to make task execution fast on the processors. And third, do not exceed the benefits of a simpler scheduling algorithm by applying a more computationally expensive one.

Our algorithm exploits data locality by taking advantage of the information in the graph. Since one task in the graph can only depend on another if the first consumes data generated by the second, we take advantage of that relation and try to run them in the same thread following a pseudo depth first traversal order.

Each thread has a ready task list associated to it. The main thread is responsible of running the main program by going through the non task user code, analysing the data dependencies and adding the tasks to the graph. New tasks that have no input dependencies are added to the main thread ready list where they can be picked up by any thread. Whenever the main thread has to wait for tasks to finish, it contributes to advancing the execution by executing tasks in the same way as the worker threads do.

Worker threads look for ready tasks first in their own list, then on the main thread ready list and then on the other thread lists. When a thread finishes running a task, it pushes all the task successors that have become ready into its ready list. While worker threads consume tasks from their own list in LIFO order, they steal them from other threads in FIFO order. That is, they consume the graph in a depth first order as long as they can can get ready tasks, and then steal tasks from other threads in a breadth first order when their ready lists become empty.

The idea behind this design is that each thread will be executing tasks in a different region of the graph and have little interference with other threads as long as there are ready tasks in that region or there are unexplored zones in the graph. Otherwise they will steal work from other threads in a way that tries to minimise the effect on the cache locality of that thread.

Figure 3 shows the task dependency graph of an execution of the code from figure 2 with $N = 2$. The shape and shade of each graph node indicates which CPU has it been executed on.

4 Conclusions and Future Work

We have presented a programming environment for SMP and multi-core chips that is easy, flexible and portable. It removes the burden of thinking in terms of data dependencies and allows the programmer to concentrate in the program itself. It is also very powerful in terms of parallelism extracting capabilities for regular and irregular algorithms.

We plan to analyse the effect of our current scheduling algorithm on the performance of various representative workloads. Other scheduling techniques will also be studied, including algorithms that bind data to processors. Finally, we plan to analyse the scalability under different regular and irregular problems and problem sizes.

References

[ABC+06] K. Asanovic, R. Bodik, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, D.A. Patterson, W.L. Plishker, J. Shalf, S.W. Williams, and K.A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, December 2006.

[BPBL06] Pieter Bellens, Josep M. Perez, Rosa M. Badia, and Jesus Labarta. Cellss: A programming model for the cell be architecture. In *proceedings of the ACM/IEEE SC 2006 Conference*, November 2006.

[SHPT00] Sanjiv Shah, Grant Haab, Paul Petersen, and Joe Throop. Flexible control structures for parallelism in openmp. *Concurrency and Computation: Practice and Experience*, (12):1219–1239, 2000.

[SS95] J. Smith and G. Sohi. The microarchitecture of superscalar processors. In *Proceedings of the IEEE*, volume 83, pages 1609–1624. IEEE, Dec. 1995.