International Journal of High Performance Computing Applications OnlineFirst, published on June 2, 2009 as doi:10.1177/1094342009106195

HIERARCHICAL TASK-BASED PROGRAMMING WITH STARSS

Judit Planas[1]
Rosa M. Badia[1, 2]
Eduard Ayguadé[1, 3]
Jesus Labarta[1, 3]

Abstract

Programming models for multicore and many-core systems are listed as one of the main challenges in the near future for computing research. These programming models should be able to exploit the underlying platform, but also should have good programmability to enable programmer productivity. With respect to the heterogeneity and hierarchy of the underlying platforms, the programming models should take them into account but they should also enable the programmer to be unaware of the complexity of the hardware. In this paper we present an extension of the StarSs syntax to support task hierarchy. A motivation for such a hierarchical approach is presented through experimentation with CellSs. A prototype implementation of such a hierarchical task-based programming model that combines a first task level with SMPSs and a second task level with CellSs is presented. The preliminary results obtained when executing a matrix multiplication and a Cholesky factorization show the viability and potential of the approach and the current issues raised.

Key words: SMP Superscalar, programming models for multicore, task scheduling, locality exploitation

1 Introduction

With the current trends in microprocessing fabrication where it is forecasted that hundreds to thousands of processors will be integrated into a single chip in the near future, there is no doubt about the need for programming models that easily enable the exploitation of these platforms. Furthermore, the future will bring heterogeneous and hybrid platforms, with different levels of compute and memory hierarchies, etc., that the programming models will have to take into account.

In this paper we review previous work towards the definition of the current Star Superscalar (StarSs) programming model by giving an overview of the two current available implementations: Cell Superscalar (CellSs) and SMP Superscalar (SMPSs). While CellSs tackles the specific Cell broadband engine (CBE) (CBEA 2, 2007) device, SMPSs is more generic and can be used both with shared-memory machines and in homogeneous multicore processors.

The CBE is a multicore chip that consists of a PowerPC processor element (PPE; a 64-bit, two-way multi-threaded, in-order PowerPC processor) and multiple synergistic processor elements (SPEs; in-order, 128-bit wide single instruction multiple data (SIMD) cores). All of them are connected to an element interconnect bus (EIB), that also couples main memory and input/output (I/O) devices. The SPEs only access main memory via direct memory access (DMA) transfers by programming their individual memory flow controllers (MFCs). For each SPE, data and code reside in its 256 kB local store (LS).

The MariCel PRACE[1] prototype at Barcelona Supercomputing Center (BSC) is an example of a hybrid computer, with 72 QS22 IBM Blade servers, each with two PowerXCell (the high-performance double-precision floating-point version of the Cell processor) at 3.2 GHz, and 12 GB of memory. In addition, the machine has 12 JS22 IBM Blade servers each with two Power6 processors and 8 GB of memory. The nodes are connected through a InfiniBand (16 Gb) network and the peak performance is 14.4 TFlops in two rack units.

A better known hybrid supercomputer is a current member of the TOP500: RoadRunner, also based in QS22 IBM Blade centers, although in this case two QS22 blades and

[1] BARCELONA SUPERCOMPUTING CENTER—CENTRO NACIONAL DE SUPERCOMPUTACIÓN (BSC–CNS), 08034 BARCELONA, SPAIN

[2] CONSEJO SUPERIOR DE INVESTIGACIONES CIENTÍFICAS (CSIC), 28006 MADRID, SPAIN

[3] UNIVERSITAT POLITÈCNICA DE CATALUNYA, 08034 BARCELONA, SPAIN
{JUDIT.PLANAS, ROSA.M.BADIA, EDUARD.AYGUADE, JESUS.LABARTA}@BSC.ES

The International Journal of High Performance Computing Applications,
Volume 23, No. 3, Fall 2009, pp. 000–000
DOI: 10.1177/1094342009106195

Figures 6–8, 10–13 appear in color online: http://hpc.sagepub.com

one IBM LS21 blade server are combined into a specialized “tri-blade” configuration for RoadRunner. The approach proposed to program RoadRunner is a hybrid and hierarchical approach where a process running in a host processor creates an accelerator process on an accelerator processor. This is implemented by means of the Data Communication and Synchronization (DaCS) Library and the Accelerated Library Framework (ALF) (Crawford et al., 2008).

The original StarSs model is a flat model with only one execution flow that generates one level of tasks. In this paper we present an extension of StarSs motivated by hierarchical architectures and by existing programming models with nested parallelism support. The current approach is to integrate in a hierarchical way our previous SMPSs and CellSs runtime implementations. A first demonstration of its utility is performed in a non-uniform memory access (NUMA) architecture like the QS22 blades but is not limited to them.

The structure of this paper is as follows: Section 2 presents the StarSs programming model. While Section 3 gives an overview of SMPSs and CellSs, Section 4 describes some experiments that motivate a hierarchical approach for the programming model. Section 5 describes the current hierarchical implementation of StarSs and gives some preliminary results. Section 6 gives an overview of existing programming models and, finally, Section 7 concludes the paper.

2 StarSs Programming Model

StarSs is a task-based programming model with two main objectives: to enable the automatic exploitation of the functional (task-level) parallelism and to keep applications unaware of the target execution platform. The starting point in StarSs is an application written in a sequential way in a traditional programming language (i.e., C/C++ or Fortran) and the target execution platform belongs to a range of parallel resources, such as a homogeneous/heterogeneous multicore chip, a shared-memory based system, a cluster or even a computational Grid. A first step is to identify in the sequential application the *tasks* that compose the application and the direction of the parameters. With this information, the StarSs environment is able to build at runtime a data-dependence graph of the tasks that compose the application. From this graph, the inherent task-level concurrency is made obvious and can be exploited. Furthermore, the environment is able to schedule tasks in the different parallel resources of the execution platform and perform other activities such as data transfer when necessary and synchronization. On task completion, the task graph is updated and new scheduling decisions are made.

2.1 StarSs Syntax

The StarSs approach is to declare, by means of pragmas, functions that are suitable for execution as tasks (see Figure 1a). For the arguments of these functions, the `clause-list` allows us to declare their use (see Figure 1b). The data references in those lists are used to dynamically compute, at runtime, dependencies among tasks. If the target architecture requires it, these clauses also specify data movement for the arguments in the function call, definition or header. Conceptually, `Input` will move variables in `data-reference-list` from the address space of the processing unit generating the task to the address space of the processing unit executing the task. `Output` will move back variables in `data-reference-list`. `Inout` combines the two effects. Once the task is ready for execution, the runtime system will move input variables. Once the task finishes execution, the runtime will move output variables, if necessary. The actual data *movement* can be avoided in a specific implementation if appropriated hardware support is available (i.e., symmetric multiprocessor (SMP)).

```
#pragma css task [clause-list]
    {function-header—function-definition}
```

(a)

```
input(data-reference-list)
output(data-reference-list)
inout(data-reference-list)
```

(b)

```
#pragma css target [clause-list]
```

(c)

```
device(device-name-ordered-list)
implements(function-name)
```

(d)

Fig. 1 StarSs syntax: (a) task definition; (b) task clauses; (c) target platform definition; (d) target clauses.

The former pragmas have been proposed before and are those implemented in CellSs and SMPSs. The extension to this syntax, proposed in this paper, is described in the following. To enable a task hierarchy, the programming model is extended to allow the instantiation of tasks within tasks. This is not a syntactical change, but a semantical change where each task makes up a private context for its subtasks. Data dependencies, as well as synchronization, are only considered between tasks in the same context. A given task waits for the end of its children tasks before finishing. These extensions enable the use of several task generators increasing the possibilities for concurrency exploitation.

The other new extension proposed in this paper is presented in Figures 1c and 1d. In heterogeneous architectures, the hierarchical approach can also consider heterogeneity. In order to target different architectures, the programming model includes an additional pragma that may precede a pragma `task` or the invocation of a function that is declared as a task (Figure 1c). The `target` pragma specifies that this implementation of the task is specific for the given target architecture, and that can be fed into the specific compiler (i.e., a synergistic processing unit (SPU) specific compiler in the case of the CBE, or a CUDA (Compute Unified Device Architecture) compiler in the case of graphics processing units (GPUs)). Some additional clauses can be used with this pragma `target` (Figure 1d). Clause `device` specifies the possible target architectures, specified in `device-name-ordered-list`, of where to offload the task. It is an ordered list that the runtime may use to decide where to execute the task. Clause `implements` is used to specify an alternative implementation for a function (Ayguade et al., 2009). There is a need for efficient implementations for each target architecture and for this reason the StarSs compiler should invoke the specific backend compiler for each case.

Two predefined combinations are already supported in the specification of StarSs, giving rise to what we call SMPSs (shared-memory multicore/SMP/ccNUMA architectures) and CellSs (for the CBE architecture).

StarSs also includes two additional pragmas for synchronization: `barrier` and `wait on`. Pragma `barrier` is used to wait for all tasks generated up to that point to be finished. Pragma `wait on` is used to wait for all of those tasks that generate variables in the `data-reference-list`[2].

2.2 Examples

Figure 2 shows the annotations that are needed to specify the parallelism in a simple application that implements a block matrix multiplication. The matrices are organized in a block data layout, where a double blocking level is applied, organizing the matrices into NBB × NBB big blocks, where each of them is composed of NSB × NSB smaller blocks of BS × BS elements.

The main program implements the matrix multiplication algorithm at the level of big blocks, calling to `sgemm2` tasks. The `sgemm2` tasks implement the matrix multiplication algorithm at the level of small blocks, calling to `sgemm1` tasks. Finally, `sgemm1` tasks implement the matrix multiplication algorithm at the level of the elements of the matrices (in this case, `floats`).

The pragma annotations identify functions `sgemm1` and `sgemm2` as tasks, with a block from C being read and written and blocks from A and B being read by the task. The task hierarchy is identified by the invocation order, being invocations to `sgemm2` first-level tasks and invocations to `sgemm1` second-level tasks. As denoted by the target pragma, these second-level tasks can be offloaded to Cell SPUs.

This code is in plain C and works correctly when compiling it with a regular C compiler and executing it sequentially. When compiled and executed within a StarSs environment, the runtime generates a hierarchical task dependence graph where tasks with no data dependencies between them can be executed concurrently.

Figures 3 and 4 show a Cholesky factorization implemented with the hierarchical StarSs proposed in this paper. The example is again based on matrices organized in a block data layout with a double blocking level as the matrix multiply example just described above. The main program (Figure 4) implements a possible algorithm to solve the Cholesky factorization at the level of big blocks, calling to tasks `sgemm2`, `strsm2`, `ssyrk2` and `spotrf2`. The `spotrf2` shown in Figure 4 also implements the same factorization algorithm but at the level of small blocks, calling this time to tasks `sgemm1`, `strsm1`, `ssyrk1` and `spotrf1`. The `sgemm2` task follows the same schema as the `sgemm2` task of the matrix multiply example, although the operation this time is slightly different (the `sgemm1` function accumulates in the C element by subtracting the product of A and B). The code of the `strsm2` and `ssyrk2` tasks is not shown in this paper. The `strsm2` is implemented without hierarchy (no further calls to second-level tasks) and the `ssyrk2` implements a similar schema to `sgemm2` with calls to the `sgemm1` task. Figure 3 shows the corresponding pragmas for the second-level tasks of this example. The target pragmas denote that these tasks can be offloaded to Cell SPUs.

3 Current Implementations

An instantiation of the hierarchical StarSs as described in the previous section may consider as its target a hierarchical hardware platform with a first level with SMP

```
#pragma css task input(A, B)  inout(C)
#pragma css target device(cell)
void sgemm1 (float C[BS][BS], float A[BS][BS], float B[BS][BS])
{
int i, j, k;
for (i=0; i < BS; i++)
   for (j=0; j < BS; j++)
      for (k=0; k < BS; k++)
          C[i][j] += A[i][k] * B[k][j];
}
#pragma css task input(A, B) inout(C)
void sgemm2 (float C[NSB][NSB][BS][BS], float A[NSB][NSB][BS][BS],
      float B[NSB][NSB][BS][BS])
{
int i, j, k;
for (i=0; i < NSB; i++)
   for (j=0; j < NSB; j++)
      for (k=0; k < NSB; k++)
         sgemm1(&C[i][j][0][0], &A[i][k][0][0], &B[k][j][0][0]);
}
int main  (int argc, char **argv) {
int i, j, k;

float A[NBB][NBB][NSB][NSB][BS][BS], B[NBB][NBB][NSB][NSB][BS][BS],
    C[NBB][NBB][NSB][NSB][BS][BS];

for (i =0; i < NBB; i++)
   for (j =0; j < NBB; j++)
       for (k=0; k < NBB; k++)
           sgemm2(&C[i][j][0][0][0][0], &A[i][k][0][0][0][0], &B[k][j][0][0][0][0]);
}
```

Fig. 2 Sample matrix multiply code in hierarchical StarSs.

```
#pragma css task input(A) inout(C)
#pragma css target device(cell)
void ssyrk1(float A[BS][BS], float C[BS][BS]);

#pragma css task inout(A)
#pragma css target device(cell)
void spotrf1(float A[BS][BS]);

#pragma css task input(T) inout(B)
#pragma css target device(cell)
void strsm1(float T[BS][BS], float B[BS][BS]);

#pragma css task input(A, B) inout(C)
#pragma css target device(cell)
void sgemm1 (float A[BS][BS], float B[BS][BS], float C[BS][BS]);
```

Fig. 3 Sample matrix multiply code in hierarchical StarSs.

```
#pragma css task input(NSB, A, B) inout(C)
void sgemm2(int NSB, float A[NSB][NSB][BS][BS], float B[NSB][NSB][BS][BS],
    float C[NSB][NSB][BS][BS]);

#pragma css task input(NSB, T) inout(B)
void strsm2(int NSB,float T[NSB][NSB][BS][BS], float B[NSB][NSB][BS][BS]);

#pragma css task input(NSB, A) inout(C)
void ssyrk2(int NSB, float A[NSB][NSB][BS][BS], float C[NSB][NSB][BS][BS]);

#pragma css task input (NSB) inout(A)
void spotrf2(int NSB, float A[NSB][NSB][BS][BS]){
int i, j, k;

 for (j = 0; j < NSB; j++) {
    for (k= 0; k< j; k++)
      for (i = j+1; i < NSB; i++)
        sgemm1(&A[i][k][0][0], &A[j][k][0][0], &A[i][j][0][0]);

    for (i = 0; i < j; i++)
         ssyrk1(&A[j][i][0][0], &A[j][j][0][0]);

    spotrf1(&A[j][j][0][0]);

    for (i = j+1; i < NSB; i++)
      strsm1(&A[j][j][0][0], &A[i][j][0][0]);

  }
}

int main  (int argc, char **argv) {
int i, j, k;

float A[NBB][NBB][NSB][NSB][BS][BS];

for (j = 0; j < NBB; j++) {
    for (k= 0; k< j; k++)
      for (i = j+1; i < NBB; i++)
        sgemm2(NSB,  &A[i][k][0][0][0][0], &A[j][k][0][0][0][0], &A[i][j][0][0][0][0]);
    for (i = 0; i < j; i++)
      ssyrk2(NSB, &A[j][i][0][0][0][0], &A[j][j][0][0][0][0]);

    spotrf2(NSB, &A[j][j][0][0][0][0]);

    for (i = j+1; i < NBB; i++)
      strsm2(NSB, &A[j][j][0][0], &A[i][j][0][0]);
  }
}
```

Fig. 4 Sample matrix multiply code in hierarchical StarSs.

multicore nodes and a second level with accelerators with local storage. Such an implementation is described in Section 5 considering QS22 blades as the target platform and SMPSs and CellSs as the building blocks for the runtime system. This section gives an overview of the CellSs and SMPSs implementations, outlining the commonalities and differences.

3.1 Compiler Overview

CellSs and SMPSs share the same pragma syntax and therefore the same annotated code applications can be run on both environments without portability problems (unless specific platform extensions are used, such as SPE intrinsics in the case of the CBE). The compiler infrastructure based on the Mercurium compiler (Gonzalez et al., 2004) is also shared by both systems, and it is composed of a C99 source-to-source compiler, a Fortran-95 source-to-source compiler and a common driver.

The driver, depending on each source filename suffix, invokes transparently the C compiler or the Fortran-95 compiler. The driver behaves similarly to a native compiler: it can compile individual files one at a time, several of which link several objects into an executable or perform all operations in a single step. The compilation process consists of processing the StarSs pragmas, transforming the code according to them, compiling these files with the corresponding native compiler and packing the object and additional information required for linking into a single object. In the case of the CBE, the compiler separates the code of the tasks that will be run in the SPEs from the other code and it is compiled with the specific SPE compiler.

The linking process consists of unpacking the object files, generating additional glue code required to join all object files into a single executable, compiling it, and finally linking all objects together with the SMPSs runtime to generate the final executable. In the CellSs case, specific glue code for the SPE side is generated and all of the SPE objects are linked and embedded together to meet the CBE binary format requirements.

3.2 Runtime Overview: Data-Dependence Analysis and Renaming

Another common specific feature of the current CellSs/SMPSs implementations is the automatic detection of data dependencies between tasks. When detecting dependencies, the CellSs/SMPSs runtimes are capable of classifying the type of data dependency: read after write (RaW), write after write (WaW) and write after read (WaR). While the first type is unavoidable and that is why those are also called true dependencies, the other type can be eliminated with the use of data renaming, inspired in the technique applied in superscalar processors (Smith and Sohi, 1995) or in optimizing compilers (Kuck et al., 1981). This renaming technique is provided transparently, although it has some overhead in terms of additional used memory and additional processing time, both to allocate the memory for the renamed copies and for memory synchronization of the last copy. Also, it can be argued that the data renaming is not an advantage in cases when there is enough parallelism, for example. However, there are cases when the use of renaming allows an application to run faster than counterparts of other programming users, thanks to the fact that the renaming technique allocates local copies of data (Perez et al., 2008).

The use of renaming memory can be controlled by the user by means of a configuration file and the authors are currently performing research into new techniques to reduce the overhead of data renaming and to reduce the number of renamings when they are not necessary.

3.3 Cellss Runtime Specifics

The CellSs runtime (Bellens et al., 2006; Perez et al., 2007) has been designed by taking into account the CBE specific architecture. It takes the benefit of the symmetric multithreaded nature of the PPE by throwing two threads into it: the *main* and *helper* thread. The main thread runs sequential code of the user application, performs an asynchronous call to the CellSs application programming interface (API) whenever a task is invoked and takes care of the synchronization between the sequential and parallel parts of the application. The call to the CellSs API on task invocation makes it possible to build the task graph. The helper thread is responsible for the scheduling tasks to the SPE threads (also called the worker threads), and the communication and synchronization with those threads. The SPE threads execute a three-step loop: waiting for notification of the scheduling decision from the helper thread, execution of a set of tasks and notification of completion. The execution of a set of tasks involves reading the task description (which includes information about the data involved in the computation), staging in the task input data, execution of the task and staging out of the task output data. Since tasks are scheduled in sets or bundles, the worker threads implement a double buffering scheme in order to overlap data transfers with computation. The default behavior of CellSs is to implement the double buffering scheme, but it is able to dynamically detect bursts of tasks where there is no memory space left and to switch to a single buffer scheme. The user can also configure application execution to disable all of the double buffering.

The worker thread also maintains a cache in the SPE local store. The basic idea is to keep those task arguments that have been used before in the SPE local store and to try to reuse them. Other features implemented in the worker thread are early callback, to notify the end of critical tasks or to minimize the stage out's, which reduces the number of copies from the local storage to main memory.

The task scheduling alternative considered so far for CellSs takes into account the structure of the task directed acyclic graph (DAG), scheduling together in the same SPE groups of tasks with a high connectivity level, to favor the exploitation of data locality. More details about the CellSs runtime can be found in Bellens et al. (2006, 2009) and Perez et al. (2007).

3.4 SMPSs Runtime Specifics

With regards to the thread organization, in SMPSs there is a main thread that is responsible for running the sequential code of the application, building the task DAG and for synchronizing the sequential and parallel parts of the application. The main worker is also able to execute tasks while it is idle. In SMPSs the task scheduling is distributed to the different SMPSs threads (worker threads). Each worker maintains its own ready list to favor data locality but workers share tasks between them to enable load balancing.

Since SMPSs assumes a shared-memory space, there is no need for data transfers[3]. In this case, the `input`, `output`, `inout` clauses are used to calculate the task data dependencies in order to build the task DAG. However, this does not mean that accessing data locally in NUMA systems is not important, as is shown by Badia et al. (2009) for the SMPSs case when used in an SGI Altix.

4 Examples and Motivation for a Hierarchical Model

In this section we describe some practical experimentation with CellSs to motivate and explain the reasons for a hierarchical approach in our programming model.

The performance of CellSs has been evolving significantly in recent years, and now provides a good performance. For example, while the matrix multiply example can almost reach the peak performance of the system when programmed directly with the IBM SDK, CellSs is able to obtain around 75% of the peak performance with minimum effort required from the programmer (for this example, we use a highly optimized 64 × 64 tile kernel, either from the IBM SDK or from http://www.tu-dresden.de/zih/cell/matmul). Figure 5 shows a comparison of different versions of a CellSs application that performs the matrix multiplication of size 2048 × 2048 floats. On the *Y*-axis we show performance in megaflops (average of 10 executions) and on the *X*-axis we show the number of SPEs used when running the benchmark. We first focus on the CellSs version 2.1, which scales linearly up to eight SPEs, with a large decrease in performance for a higher number of SPEs. However, from the very beginning (one SPE) is does not deliver peak performance.

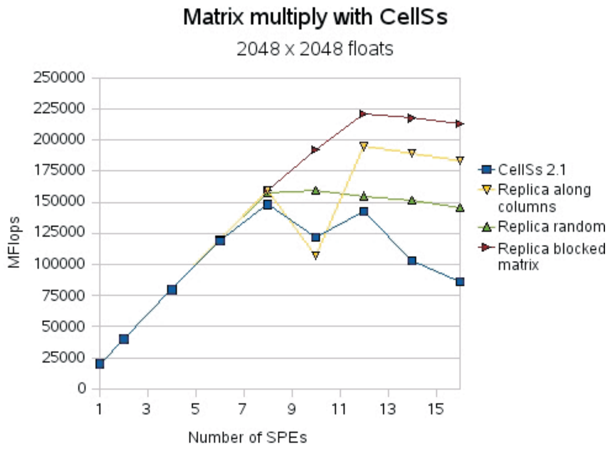


Fig. 5 Matrix multiply performance with CellSs.

When analyzing the reasons for the reduction in performance, we examine first the case when one SPE is used. For this case, the highly optimized kernel provides 25.2 GFlops for each 64 × 64 block, while CellSs with one SPE is delivering 20 GFlops. There are several possible reasons for this decrease of around 20% in performance: handling of the task DAG and synchronization in the PPE or DMA data handling in the SPE. For the case of one SPE, the PPE is introducing a very small initial overhead for generating the task graph (0.17% of the total time). From this moment, the SPE is busy all of the time, therefore the PPE does not add any extra overhead to the SPE computation. From the SPE side, while it is performing productive computation almost all of the time, a significant amount of time is invested in the management of the software cache (this time includes the time invested in synchronous writes to memory when replacements in the software cache are required) and data stage in programming, and also small periods of time are invested in waiting for DMA to finish and programming the DMA outs (the main part of the transfers are overlapped thanks to the double buffering). Figure 6 shows this decomposition of the SPE time in CellSs for this example (when one SPE is used).

This situation remains the same up to four or five SPEs, with a small incremental increase in the overhead for the software cache management noticeable from six SPEs although still small enough with up to eight SPEs. However, when running this benchmark with more than nine SPEs the situation is much worse. There are several aspects to consider at this point:

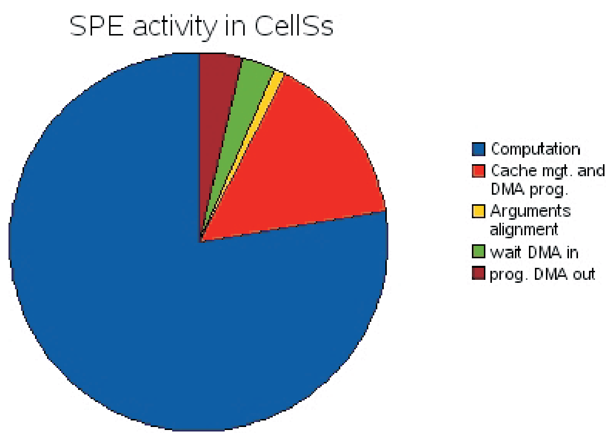


Fig. 6 SPE overheads decomposition in one SPE.

- From one to eight SPEs, the command line call to the NUMA API (*numactl*) has been used to allocate all SPEs in the same node. Also, the matrices are allocated in the memory module bound to the computational node.
- From nine SPEs, the SPE threads are allocated to two nodes, and with the default CellSs behavior, the data will be allocated to the memory node bound to the node where the main thread is running.

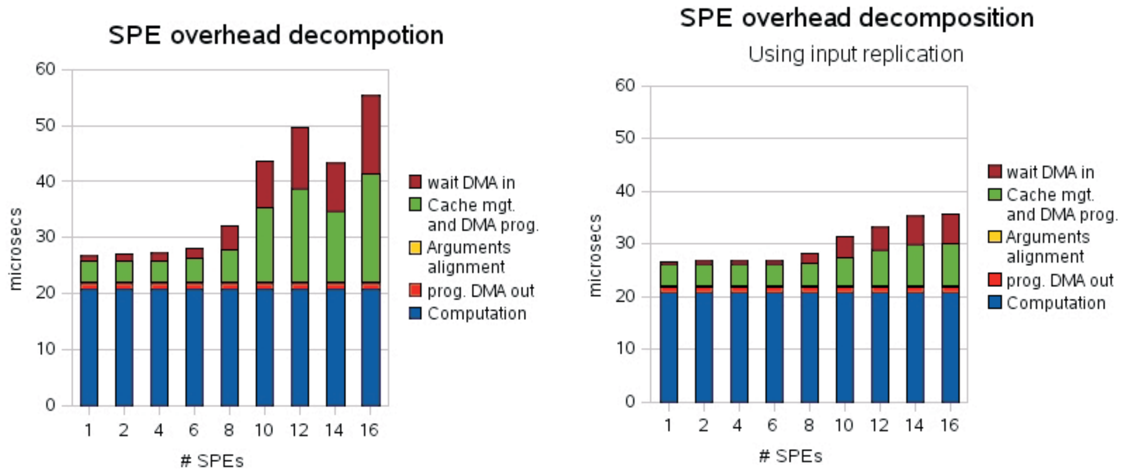
Taking this into account, there are two possible reasons for the reduction in performance when increasing the number of SPEs above eight: overheads in the main and helper thread (for DAG creation, scheduling, etc.) or reduction in the effective bandwidth with memory, due to the increase in the number of threads accessing it and due to the reduction in locality (threads allocated in different nodes to the data).

In Figure 7a we show the decomposition of the SPE time when the number of SPEs is increased (this figure is derived from analysis of Paraver tracefiles). The figure tries to capture the time invested in the SPE in the execution of the tasks but it is not showing the time that those SPEs are idling due to bottlenecks in the PPE threads. The figure shows that with up to eight SPEs, the decomposition of the time looks almost the same. From eight SPEs onwards, the time invested in software cache management and in waiting for the staged data increases significantly. To understand this fact, we performed some more experiments.

In the case of the IBM SDK matrix multiply example, several optimizations are applied to enable the perfect scaling of the example:

- The use of huge page allocations (16 MB pages) for reduced page table and translation lookaside buffer (TLB) thrashing.
- Replication of input matrices in the different NUMA nodes. It is not a very scalable trick, but demonstrates the importance of being able to exploit the available memory bandwidth in NUMA architectures.
- Padding the leading dimension of row- or column-order matrices so that improved distribution of memory accesses across all memory banks.

There are two different aspects that we wanted to measure: the increase of memory bandwidth due to the repli-



(a) (b)

Fig. 7 SPE overheads decomposition.

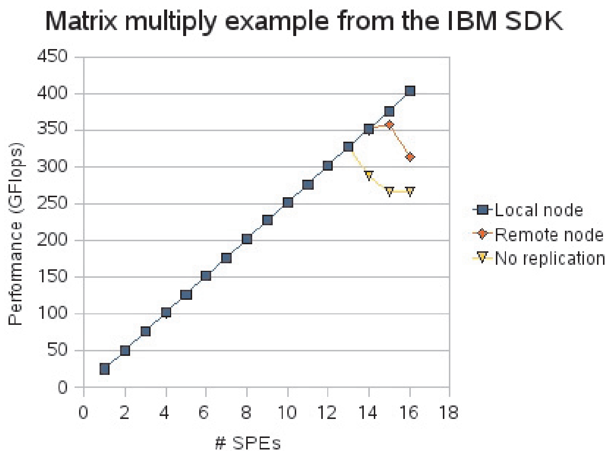


Fig. 8 Performance of IBM SDK matrix multiply when using replication.

cation of the data (requests to the same memory module are reduced) and the increase of memory bandwidth due to the increase in locality of the requests. A first attempt to explain this idea has been performed with the SDK example. We measured the performance of the example in the initial implementation and in a slightly modified implementation, where each thread was accessing the remote memory node (SPEs in node zero access replicas in memory node one and vice versa). The results are shown in Figure 8 where we can see that the performance of the example is the same up to 14 SPEs, with a reduction in the performance when crossing the data buffers with 15 and 16 SPEs. According to this, the impact of replication is very important. The figure also shows the performance when no replication is used.

We have applied the optimizations described in the list above to the CellSs matrix multiply example. The first optimization we tried is the use of huge pages. No significative difference was observed for this example (in fact, the authors did not measure any significative difference in the SDK example either when using large pages). The next step was to try exploit all of the memory bandwidth offered by the NUMA architecture of the system memory modules. In the SDK example, the input matrices A and B are replicated in the two memory nodes. Also, the SPE thread allocation forces the use of the copies allocated in the local node, to increase locality and bandwidth.

We have also applied the same idea in the CellSs example, replicating the input matrices in the two memory nodes. Although from the user program we cannot bind tasks to the physical SPEs, since this is done automatically by the runtime, we tried three different approaches to measure the impact of this strategy: a *random* approach, where the tasks were randomly called using one replica or another of the input matrices; *distributing replicas along columns*, that is, tasks accessing even columns of C were assigned one replica and odd columns were assigned the other; and *distributing by a blocked C matrix*, adding an additional level of blocking to matrix C, and contiguous blocks were assigned different replicas.

In all three cases the performance increases, especially when blocking the memory (in fact, this approach tries to favor the assignment of tasks accessing the local memory module), since CellSs follows a first in first out (FIFO) scheduling cycle when there is enough parallelism).

Figure 7b shows the decomposition of the SPE time when huge pages and replication are used (the blocking approach). The difference when compared with Figure 7a is quite impressive: while when using up to eight SPEs the figures are identical, the reduction in time that the SPEs invest in the management of the software cache in the local store (now with more bandwidth with memory for the synchronous writes due to replacements) and when waiting for staged data is quite large. If CellSs was able to guarantee that all tasks accessed the replica in the local memory module, these times should be the same as when using up to eight SPEs. However, this is not possible with the current CellSs implementation. Furthermore, keeping replicas of all of the input data does not look to be an efficient and scalable strategy unless done in a controlled way.

Another source of inefficiency in CellSs is the bottlenecks in the main and helper thread for creating and managing the tasks graph when using a large number of SPEs. While in a CellSs application we can use up to 16 SPEs, only two of the Power threads are used and the other two are left idling. The hierarchical approach proposed in this paper has the potential to solve these problems, and also opens new possibilities for expressing applications with a hierarchy of tasks.

5 Hierarchical Implementation

A prototype StarSs hierarchical implementation based on the syntax proposed in Section 2.1 that composes SMPSs and CellSs runtimes together has been implemented. The implementation is currently targeted to IBM QS22 blades, as those described in Section 1. In this platform, we consider a main program running on one of the two Power processors. This main program calls to coarse-grain tasks that are executed in the Power processors (SMPSs tasks). The SMPSs tasks themselves call to fine-grain tasks, which are executed in the SPEs (CellSs tasks).

The runtime of this hierarchical implementation is based on the SMPSs and CellSs, with the necessary modifications to allow them to run together (basically, the definition of new data structures that isolate the double tasking

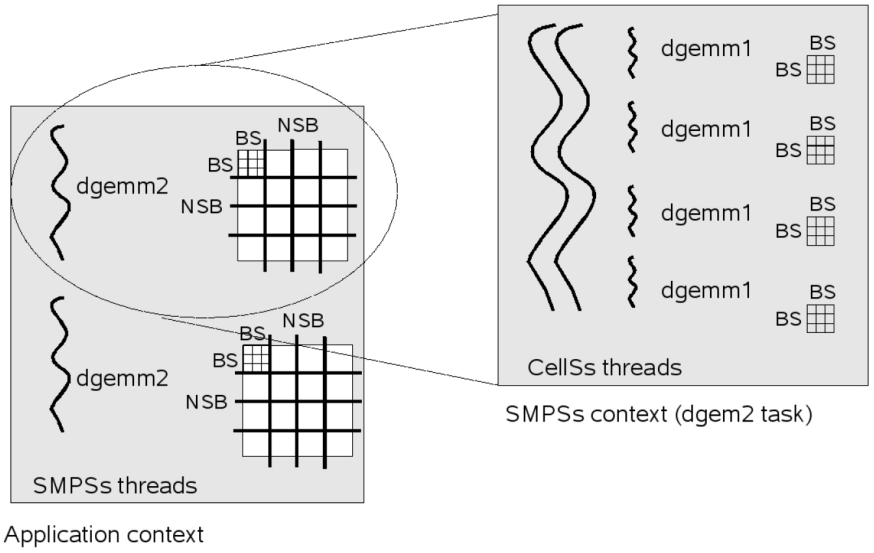


Fig. 9 Thread organization in hierarchical StarSs.

level). Currently, no other modification have been implemented in this version.

Taking into account the target platform, applications can be composed of up to two SMPSs threads. One of these SMPSs threads acts as global main thread, main thread for SMPSs tasks and main thread for CellSs tasks called in this thread. The other SMPSs thread, acts as worker SMPSs thread, but also as a main thread for CellSs. Each SMPSs thread has associated a helper thread (CellSs level) and a variable number of SPE threads. Currently, a symmetric configuration is assumed, with an identical number of SPE threads for each SMPSs thread. In this way, memory locality is exploited, since all threads bind to a SMPSs thread run in the same chip and can access the corresponding memory module, but forthcoming implementations can consider a variable number of SPE threads per SMPSs thread to consider different profile tasks and enable load balancing.

The data-dependency analysis is performed by considering each SMPSs task as an independent context. Therefore, the runtime seeks data dependencies between SMPSs tasks and inside the SMPSs looks for data dependencies between the CellSs tasks called in that task. Each SMPSs task is considered as a context, and therefore the data dependencies between CellSs tasks running in different SMPSs tasks are respected thanks to the data dependencies between the SMPSs tasks. Each SMPSs tasks waits for its child CellSs tasks to finish before completing the SMPSs-level task. Current implementation does not support nesting at the same level (i.e., SMPSs tasks inside SMPSs tasks or CellSs tasks inside CellSs tasks), but future implementations will.

The example shown in figure 2 has been implemented in the current prototype implementation, being the first level `dgemm2` tasks, SMPSs tasks and the second level `dgemm1` tasks, CellSs tasks (running inside SMPSs tasks). Figure 9 shows the thread organization of this application in the current implementation. A first level of `dgem` tasks is created in the application context. This first level of tasks are executed by SMPSs threads (the reasonable number in a QS22 blade would be two SMPSs threads) and these tasks access to a submatrices of the original ones of NSB × NSB blocks of BS × BS elements. Each of the SMPSs tasks (dgemm2) has its own SMPSs context, which creates a second level of CellSs tasks (`dgemm1` tasks). In each SMPSs context, we find a main thread, a helper thread (executed in the Power processor) and a variable number of SPE threads (four in the picture). The CellSs tasks operate in submatrices of BS × BS elements. Figure 10 shows the task DAG generated for this matrix multiply example when NBB = 3 and NSB = 2. Each large node in the figure represents the execution of a `dgemm2` task and each small node represents the execution of a `dgemm1` task.

Figure 11 shows results obtained with the matrix multiply example. The matrices are of 2048 × 2048 floats, organized in the first case in 4 × 4 big blocks (NBB) and 8 × 8 small blocks (NSB). The figures show two lines: one using one SMPSs thread (one CPU thread) and another with two SMPSs threads (in some cases these lines are

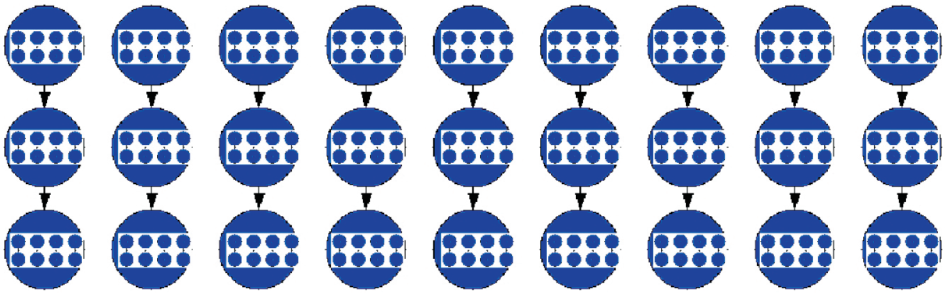


Fig. 10 Task DAG for the hierarchical implementation of the matrix multiply example (NBB = 3, NSB = 2).

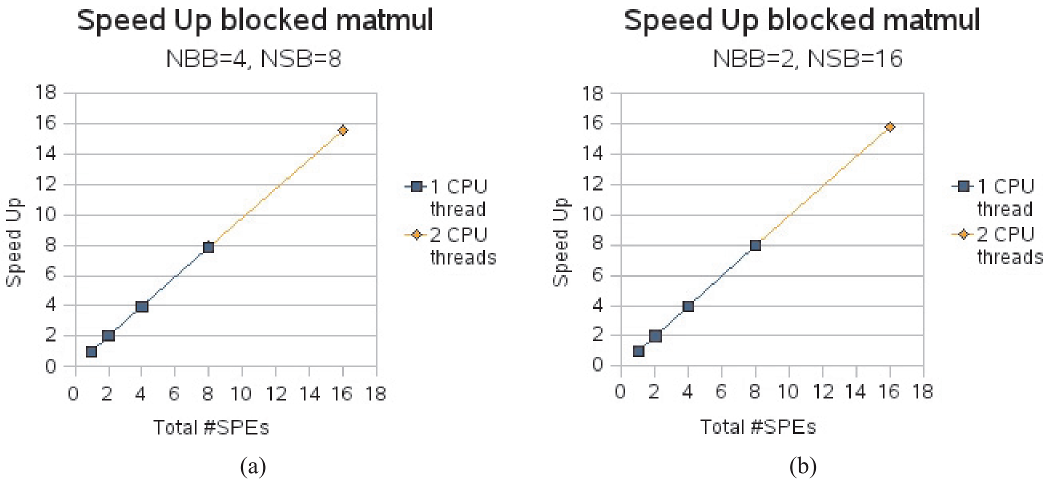


Fig. 11 Matrix multiply example, size 2048 × 2048 floats: (a) NBB = 4, NSB = 8; (b) NBB = 2, NSB = 16.

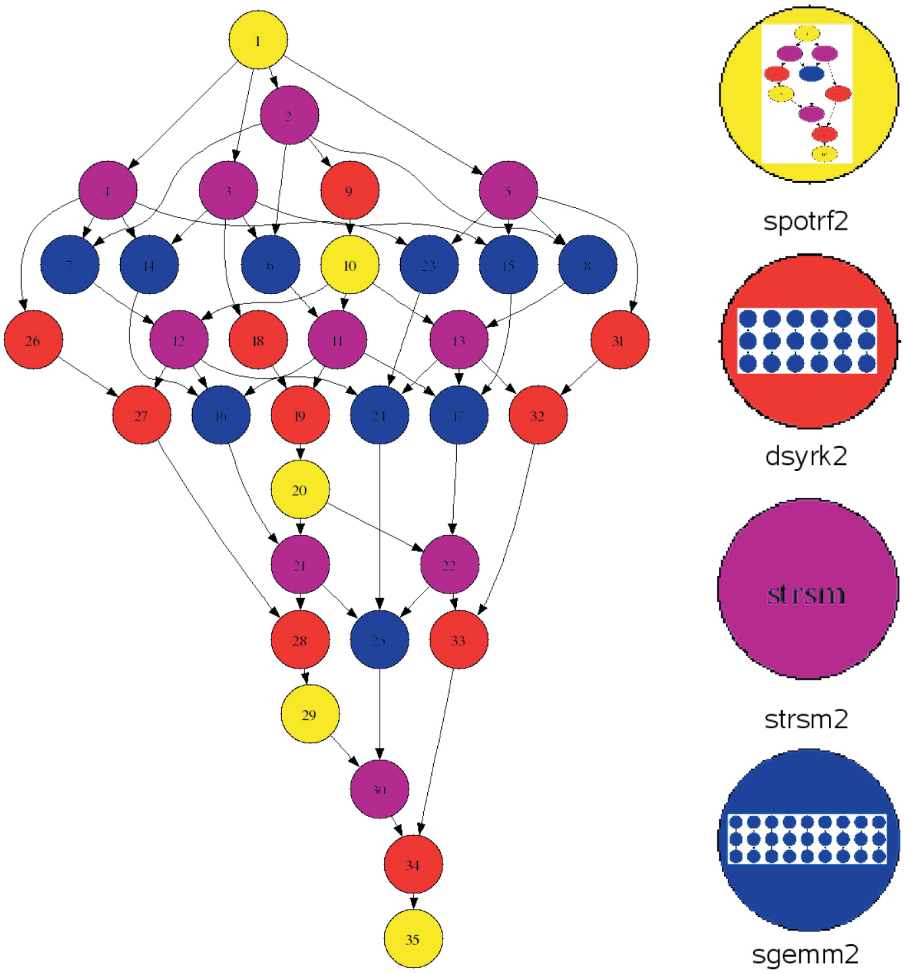
overlapped and difficult to differentiate). We have executed the examples with number of SPU varying from one to eight for each SMPSs thread, but the figure shows the speed increase against the total number of SPUs (therefore, with two CPUs, the number of SPU threads goes from 2 to 16 threads). For this problem the figures show good results, scaling perfectly up to 16 SPUs.

We have also run the Cholesky example presented in Section 2.2 in the same QS22-based platform. The task DAG that is generated for NBB = 5 and NSB = 3 is shown in Figure 12. As in the case of the matrix multiply example, large nodes represent first-level tasks, executed in the SMPSs threads (`spotrf2`, `strsm2`, `ssyrk2` and `sgemm2`). The figure has been split into two parts for clarity. Each of the nodes of Figure 12(a) has the internal structure represented in Figure 12(b). The small nodes represent second-level tasks, executed in the Cell SPUs (`spotrf1`, `strsm1`, `ssyrk1` and `sgemm1`).

The implementation of the Cholesky factorization is given in this paper as an example of programmability of the hierarchical StarSs approach. The hierarchical approach allows us to program applications in an incremental and modular way. For example, in this first implementation of the Cholesky the implementation of the `strsm2` task is sequential. After execution and analysis of the bottlenecks of the application, the programmer can decide which tasks to optimize or not, to derive specific implementations for given platforms, etc. Execution results for the Cholesky example are shown in Figure 13, as expected not so good as with the previous matrix multiply example.

6 Related Work

In the literature we can find several programming models that tackle the parallelization of applications from sequen-

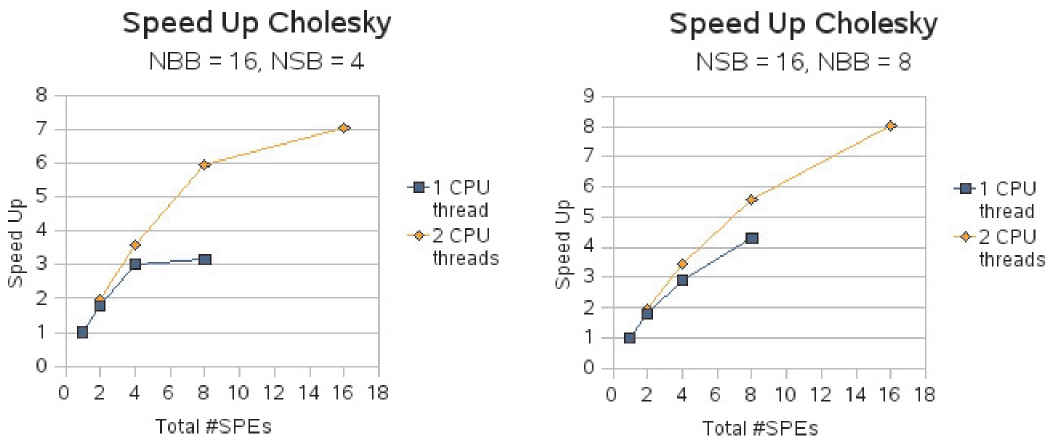


(a) (b)

Fig. 12 Task DAG for the hierarchical implementation of the Cholesky factorization example (NBB = 5, NSB = 3): (a) first-level DAG; (b) first-level nodes' structure.

tial codes. Between them, OpenMP[4] is a high exponent of them. OpenMP is based on a shared-memory model and on pragma annotations that are inserted into the code to give hints to the compiler about the existence of parallelism. The standard has been extended in its version 3.0 with a tasking model that allows us to address this other level of parallelism. Although this version of the standard does not support the detection of tasks' data dependencies there have been proposal to define tasks' precedence (Gonzalez et al., 2003) and task dependencies (Duran et al., 2009). Comparisons between SMPSs performance and OpenMP are shown in Perez et al. (2008) and Duran et al. (2009).

Another task-based programming model is Cilk (Frigo et al., 1998) which is a general-purpose programming language for multi-threaded parallel programming. In Cilk, the programmer identifies tasks with the *spawn* keyword and the *sync* keyword is used to wait for spawned tasks. Both OpenMP and Cilk consider nested tasks (tasks that generate new tasks) but the inter-task data-dependence detection is not supported and therefore the programmer has to consider this in the development of the application. Cilk scheduler is based on a work-stealing approach in such a way that is able to naturally exploit the data locality. While Cilk only supported parallel tasks, Cilk++ also supports parallel loops. Perez et al. (2008) also compared SMPSs with Cilk.



(a)

(b)

Fig. 13 Results for the Cholesky factorization example: (a) NBB = 16, NSB = 4, matrix size 4096 × 4096 floats; (b) NBB = 16, NSB = 8, matrix size 8192 × 8192 floats

The Parallel Linear Algebra for Scalable Multicore Architectures (PLASMA)[5] (Buttari et al., 2009) project is also addressing the problems that the linear algebra and high-performance computing community are facing due to the introduction of multicore architectures. PLASMA's goal is to create software frameworks that enable programmers to simplify the process of developing applications that can achieve both high performance and portability across a range of new architectures. With this objective, the project aims to develop programming models that enforce asynchronous, out-of-order scheduling of operations as the basis for the definition of a scalable and highly efficient software framework for computational linear algebra applications. Kurzak et al. (2009) presented a comparison between Cilk, SMPSs and an static scheduling mechanism currently used in PLASMA, showing results for Cholesky and *QR* factorizations. Although the static approach was giving the best results, SMPSs was very close, and taking into account the complexity of the task graphs that some applications can generate, a static (manual) approach is not possible.

SuperMatrix (Chan et al., 2007) is another programming paradigm that focuses on automatic parallelization although its application is limited to linear algebra matrix operations. SuperMatrix provides a library with a set of linear algebra routines that generates tasks internally and an API to build algorithms composed of those routines. A similarity of SuperMatrix with StarSs is that it is able to detect data dependencies between tasks, although it does not implement data renaming.

CAPS Hybrid Multicore Parallel Programming (HMPP) (Dolbeau et al., 2007) aims to simplify the use of hardware acceleration in conventional general-purpose applications while keeping the application code portable. The approach supports accelerator functions, written in the accelerator's own language. The main source code is written in a traditional programming language (C, Fortran) and makes calls to the accelerator functions. The framework provides specific compiler, library and other tools that help in generating the binaries. The approach makes use of directives. The main directives are `codelet` that allows the declaration of a codelet (a codelet is an accelerator function, with no side effects and no I/O) and `callsite` that allows codelets to be called from the main program code. HMPP itself does not exploit the concurrency of the code, but it is compatible with OpenMP and MPI and therefore the parallelism of the application is exploited by these other programming models.

An approach that somehow resembles StarSs is Mentat (Grimshaw, 1993), the basic idea of which is to let programmers only express what should be executed in parallel. Based in C++ and deeply object-oriented, in Mentat programmers indicate which classes have methods that are coarse grained enough to let them execute in parallel. Mentat supports the asynchronous execution of these operations using a macro dataflow model, where an operation can start as soon as all of its inputs are available. In addition to offering a high-level programming model, the runtime system takes over the communication, synchronization and load balancing of the applications.

With regards to programming models for the CBE, O'Brien et al. (2008) presented the support of OpenMP on the Cell processor, integrated in the XL compiler. The approach solves specific issues of the Cell processor: synchronization of the heterogeneous threads and specific code generation for the PPE and SPE sides, for example, support for the latter coming from the partitioning of the SPE code into multiple overlaid binary sections. Another specific feature of this approach is the memory management, where a software cache mechanism is part of the runtime library and it is used to access global variables. However, this implementation supports OpenMP version 2.5 but not OpenMP 3.0, therefore tasking is not supported.

Another approach to support the heterogeneity of the Cell is the ALF (IBM Corporation, 2007), that provides a programming environment for data and task parallel libraries and applications. ALF simplifies the task of development of computational libraries that encapsulate accelerated kernels. The framework then enables offloading of the computationally intensive kernels to the hardware accelerators.

ALF supports parallelism by means of multiple program multiple data (MPMD) paradigm. ALF supports also the data transfers, double buffering and load balancing for parallel tasks. These parallel tasks do not have any direct or indirect dependency between them, but ALF support the explicit definition of task execution order by means of the function `alf_task_depends_on`.

ALF can be combined with the DaCS (IBM Corporation, 2008) library that provides a set of services which ease the development of applications and application frameworks in a heterogeneous multi-tiered system. DaCS establishes a hierarchical topology of processing elements (DaCS elements (DEs)). In a hierarchy, a DE can be either a general-purpose processing element that acts as supervisor (host element (HE)) or as a processing element that executes the computing tasks designated by the HE (accelerator element (AE)). DaCS provides services for initializing, offloading computation to AEs, synchronization, and HE–AE communication. RoadRunner software development has been performed by means of combining ALF and DaCS (Crawford et al., 2008).

SP@CE is a programming model for streaming applications (Varbanescu et al., 2006), built as a component-based extension of the series–parallel contention (SPC) model of computation. In SP@CE, an streaming application is designed by providing a graph of computation kernels connected by data streams. The intermediate representation translates this graph into an expanded version of the application-dependency graph, which is then optimized and dynamically scheduled (using a job queue) on the hardware platform by the runtime system. To accommodate the CBE back-end, a new component type has been added to embed the SPE computation. Each SPE runs a tiny runtime system, based on a task state automaton, and follows the kernel execution step by step.

Sequoia (Fatahalian et al., 2006) is a programming language based on C++. Similarly to CellSs, it decomposes programs into tasks. In this case, one of the differences is that in Sequoia, tasks can call themselves recursively. While the top level (inner task implementation) recursively decomposes the problem into smaller tasks, the lower level (leaf task implementation) implements the SPE code itself. Whether a task call is bound to the inner task or the leaf task is determined by the runtime according to the user-specified task-mapping specification.

RapidMind[6] is another programming model for the CBE processor. It is based on a C++ template library and a runtime library that performs dynamic code generation. The template library allows writing and invoking of SPE code from within the PPE code. All SPE code is written using the template library.

7 Conclusions and Future Work

StarSs is a task-based programming model which aims to bring a efficient solution in forthcoming multicore architectures, both in terms of programmability and performance. CellSs and SMPSs are the two implementations targeted to different systems currently available. In this paper we have motivated and presented a hierarchical approach that combines both SMPSs and CellSs, with different levels of granularity, with CellSs tasks being children of SMPSs tasks. The results presented in this paper show that the approach is promising.

The current version of the hierarchical approach presents only modifications to enable the interoperation of SMPSs and CellSs. While the current solution statically assigns a given number of SPUs to each SMP tasks, we want to explore a dynamic approach, where each SMP task is given a different number of resources, depending on its needs. The next steps will look towards enabling the hierarchical approach through several nodes. Research towards bringing a solution for accelerator-based machines such as RoadRunner that do not require the whole duplication of data in the memory of the different levels of the hierarchy is also foreseen.

Acknowledgments

The researchers at BSC-UPC were supported by the Spanish Ministry of Science and Innovation (contract numbers TIN2007-60625 and CSD2007-00050), the European Commission in the context of the HiPEAC Network of Excellence (contract number IST-004408), and the MareIncognito project under the BSC-IBM collaboration agreement.

Authors' Biographies

Judit Planas has a degree in Informatics Engineering (2008) from the Technical University of Catalonia (UPC). Currently she is pursuing her Master's Degree in Computer Architecture, Networks and Systems (CANS) at the Computer Architecture Department of the UPC. Since July 2008 she has held an academic grant from the Barcelona Supercomputing Center to develop her studies and research in the framework of the center.

Rosa M. Badia has a PhD in Computer Science (1994) from the Technical University of Catalonia (UPC). Since 2008 she has been a Scientific Researcher from the Consejo Superior de Investigaciones Cientificas (CSIC) and manager of the Grid Computing and Cluster research group at the Barcelona Supercomputing Center (BSC). She has been involved in teaching and research activities at the UPC from 1989 to 2008, and where she has also been an Associated Professor since 1997. From 1999 to 2005 she has been involved in research and development activities at the European Center of Parallelism of Barcelona (CEPBA). Her current research interest are performance prediction and modeling of MPI programs and programming models for complex platforms (from multicore to the Grid). She has participated in several European projects and currently she is participating in projects HPC-Europa2, Brein, and is a member of HiPEAC2.

Eduard Ayguade received an engineering degree in Telecommunications in 1986 and PhD in Computer Science in 1989, both from the Universitat Politecnica de Catalunya (UPC), Spain. Since 1987 he has been lecturing in computer organization and architecture and parallel programming models. Currently, and since 1997, he has been full professor of the Computer Architecture Department at UPC. He is currently associated director for research on Computer Sciences at the Barcelona Supercomputing Center (BSC–CNS). His research interests cover the areas of processor microarchitectures, multicore architectures and programming environments models and their architectural support. He has published around 200 publications in these topics and participated in several research projects with other universities and industries, in framework of the European Union programmes or in direct collaboration with technology leading companies.

Jesus Labarta has been full professor at the Computer Architecture department at UPC since 1990. Since 1981 he has been lecturing on computer architecture, operating systems, computer networks and performance evaluation. His research interest has been centered on parallel computing, covering areas from multiprocessor architectures, memory hierarchy, parallelizing compilers and programming models, operating systems, parallelization of numerical kernels, metacomputing tools and performance analysis and prediction tools. He has lead the technical work of UPC in some 15 industrial R&D projects. Significant performance improvements were achieved in commercial codes owned by partners with whom he has cooperated. Since 1995 he has been director of CEPBA and currently he is director of the Computer Sciences Research Department at BSC.

Notes

[1] See http://www.prace-project.eu/.

[2] See http://www.bsc.es/media/1652.pdf.

[3] Although this copying can be performed to increase memory bandwidth

[4] See the OpenMP 3.0 Specification at http://www.openmp.org/.

[5] See http://icl.cs.utk.edu/plasma/index.html.

[6] See http://www.rapidmind.net/case-cell.php

References

Ayguade, E., Badia, R. M., Cabrera, D., Duran, A., Igual, F., Gonzalez, M., Jimenez, D., Labarta, J., Martorell, X., Mayo, R., Perez, J. M. and Quintana-Ortií, E. S. 2009. A proposal to extend the OpenMP tasking model for heterogeneous multicores. In *5th International Workshop on OpenMP*.

Badia, R. M., Perez, J. M., Ayguadé, E. and Labarta, J. 2009. Impact of the memory hierarchy on shared memory architectures in multicore. In *17th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*.

Bellens, P., Perez, J. M., Badia, R. M. and Labarta, J. 2006. CellSs: a programming model for the Cell BE architecture. In *Proceedings of the ACM/IEEE Supercomputing 2006 Conference*.

Bellens, P., Perez, J. M., Cabarcas, F., Ramirez, A., Badia, R. M. and Labarta, J. 2009. CellSs: scheduling techniques to better exploit memory hierarchy. *Scientific Programming* 17(1):77–95.

Buttari, A., Langou, J., Kurzak, J. and Dongarra, J. 2009. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing: Systems and Applications* 35:38–53.

CBEA_2 2007. *Cell Broadband Engine Programming Handbook, Version 1.1*, International Business Machines Corporation, Sony Computer Entertainment Incorporated, Toshiba Corporation.

Chan, E., Quintana-Orti, E. S., Quintana-Orti, G. and van de Geijn, R. 2007. SuperMatrix out-of-order scheduling of matrix operations for SMP and multi-core architectures. In *Proceedings of the 19th Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 116–125.

Crawford, C., Henning, P., Kistler, M. and Wright, C. 2008. Accelerating computing with the Cell broadband engine processor. In *Proceedings of the ACM International Conference on Computing Frontiers*.

Dolbeau, R., Bihan, S. and Bodin, F. 2007. HMPP: a hybrid multi-core parallel programming environment. In *First Workshop on General Purpose Processing on Graphics Processing Units*.

Duran, A., Ferrer, R., Ayguadé, E., Badia, R. M. and Labarta, J. 2009. A proposal to extend the OpenMP tasking model with dependent tasks. *International Journal of Parallel Programming* 37(3):292–305.

Fatahalian, K., Horn, D., Knight, T. J., Leem, L., Houston, M., Park, J. Y., Erez, M., Ren, M., Aiken, A., Dally, W. and Hanrahan, P. 2006. Sequoia: programming the memory hierarchy. In *ACM/IEEE Conference on Supercomputing*.

Frigo, M., Leiserson, C. E. and Randall, K. H. 1998. The implementation of the Cilk-5 multithreaded language. *SIGPLAN Notices* 33(5):212–223.

Gonzalez, M., Ayguadé, E., Martorell, X. and Labarta, J. 2003. Exploiting pipelined executions in OpenMP. In *Proceedings of the 32nd Annual International Conference on Parallel Processing*, pp. 153–160.

Gonzalez, M., Balart, J., Duran, A., Martorell, X. and Ayguadé, E. 2004. Nanos Mercurium: a research compiler for OpenMP. In *Proceedings of the European Workshop on OpenMP*.

Grimshaw, A. S. 1993. Easy-to-use object-oriented parallel programming with Mentat. *Computer* 26(5):39–51.

IBM Corporation 2007. *Accelerated Library Framework for Cell Broadband Engine Programmer's Guide and API Reference, Version 3.0*, available at: http://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/41838EDB5A15CCCD002573530063D465.

IBM Corporation 2008. *Data Communication and Synchronization Library Programmer's Guide and API Reference, Version 3.1*, available at: http://publib.boulder.ibm.com/infocenter/systems/scope/syssw/index.jsp?topic=/eicck/eicckkickoff.html.

Kuck, D. J., Kuhn, R. H., Padua, D. A., Leasure, B. and Wolfe, M. 1981. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, 207–218.

Kurzak, J., Ltaief, H., Dongarra, J., and Badia, R. 2009. Scheduling for numerical linear algebra library at scale. In *Trends in High Performance and Large Scale Computing*, IOS Press, Amsterdam.

O'Brien, K., O'Brien, K., Sura, Z., Chen, T. and Zhang, T. 2008. Supporting OpenMP on Cell. *International Journal of Parallel Programming* 36(2):289–311.

Perez, J., Badia, R. and Labarta, J. 2008. A dependency-aware task-based programming environment for multi-core architectures. In *Proceedings of IEEE Cluster Computing 2008*.

Perez, J. M., Bellens, P., Badia, R. M. and Labarta, J. 2007. CellSs: making it easier to program the Cell broadband engine processor. *IBM Journal of Research and Development* 51(5):593–603.

Smith, J. E. and Sohi, G. S. 1995. The microarchitecture of superscalar processors. *Proceedings of the IEEE* 83(12): 1609–1624.

Varbanescu, A., Nijhuis, M., Gonzalez-Escribano, A., Sips, H., Bos, H. and Bal, H. 2006. SP@CE—an SP-based programming model for consumer electronics streaming applications. In *Proceedings of LCPC 2006* (*Lecture Notes in Computer Science*, Vol. 4382), Springer, Berlin.