CellSs: Making it easier to program the Cell Broadband Engine processor

J. M. Perez
P. Bellens
R. M. Badia
J. Labarta

With the appearance of new multicore processor architectures, there is a need for new programming paradigms, especially for heterogeneous devices such as the Cell Broadband Engine™ (Cell/B.E.) processor. CellSs is a programming model that addresses the automatic exploitation of functional parallelism from a sequential application with annotations. The focus is on the flexibility and simplicity of the programming model. Although the concept and programming model are general enough to be extended to other devices, its current implementation has been tailored to the Cell/B.E. device. This paper presents an overview of CellSs and a newly implemented scheduling algorithm. An analysis of the results—both performance measures and a detailed analysis with performance analysis tools—was performed and is presented here.

Introduction and motivation

To design each generation of processors with higher performance than the last is becoming increasingly difficult because of the technological limitations imposed by their power consumption and heat generation. The current industry roadmap is based on multicore designs, that is, chips with multiple processors [1]. Each of the cores in these chips can offer less performance than the current single-core processors, but together they form a high-performing and energy-efficient device. Several examples are on the market: the AMD Opteron** and Athlon** processors; from Intel the dual-core P4 Pentium** D core codenamed Smithfield, the forthcoming dual-core Itanium** processor codenamed Montecito, and the quadcore processor codenamed Kentsfield; and the IBM POWER4*, POWER5*, and POWER6* processors. These are examples of homogeneous multicore processors. However, there are also examples of heterogeneous multicore processors, such as ClearSpeed** accelerator systems and the Ageia PhysX** physics processing unit.

Furthermore, Intel recently announced the design of a research prototype with 80 core processors and a capacity of more than a trillion flops that uses less electricity than a modern desktop chip. The chip is modularly designed, and each tile has its own router built into the core, creating a network on a chip.

The challenge now facing programmers is this: Applications must be ported to these new multicore architectures so that they can make use of threads and take advantage of all the possibilities offered by these devices. According to a recent Berkeley report [2], the current programming methodologies can be used with chips with two to eight cores, but not for systems with more than 16 or 32 processors per chip. Also in this report, the authors set a target of 1,000 cores per chip and reference a set of 13 dwarfs (a *dwarf* is an algorithmic method that captures a pattern of computation and communication) as benchmarks to be used to design and evaluate parallel programming models and architectures. Current programming methodologies should be shifted toward a more human-centric point of view to maximize programmer productivity, and the programming models should be independent of the processor count. A wide range of data types should be supported, as well as task-, word-, and bit-level parallelism.

0018-8646/07/$5.00 © 2007 IBM

The first-generation Cell Broadband Engine† (Cell/B.E.) processor [3] is composed of a 64-bit multithreaded IBM PowerPC* processor element (PPE) and eight synergistic processor elements (SPEs) connected by an internal high-bandwidth element interconnect bus (EIB). The PPE has two levels of on-chip cache and supports vector multimedia extensions (VMX[1]) to accelerate multimedia applications by using VMX single-instruction multiple-data (SIMD) units. The eight SPEs in a Cell/B.E. device are processors designed to accelerate media and streaming workloads. There are two important problems that the programmer is faced with when using the Cell/B.E. device: First, the SPE local memory is small (256 KB) and is not coherent with the PPE main memory. Each time a computation is to be executed in an SPE, the data must be transferred from main memory to the SPE local memory through a direct memory access (DMA) transfer. Second, the maximum performance of an SPE is obtained with vectorized code using single-precision floating-point (float) data.

CellSs [4] has been recently proposed as a programming model for multicore processors, and its current implementation is focused on the Cell/B.E. device. The programming model is based on simple annotations to a sequential code. The annotations identify independent parts of the code (*tasks*) without collateral effects (only local variables and parameters are accessed). A source-to-source compiler is used to generate the code for both the PPE and SPEs. At runtime, the system will try to concurrently execute tasks in different SPEs without data dependencies among them. To meet this objective, at runtime the system builds and schedules a task-dependency graph. Also, all data transfers between main memory and SPE local memory are handled by the system.

In this work we focus on offering tools that enable a flexible and high-level programming model for the Cell/B.E. processor while relying on other compilers [5, 6] (or those that may appear in the future) for code vectorization and other, lower level code optimizations.

In this paper, we provide an overview of CellSs and describe the new scheduling strategies implemented in the runtime library. Some experimental results and trace files of real executions are presented, and we review some proposals related to this work.

Overview of CellSs

CellSs is a programming model for multicore processors. Its current implementation is tailored to the Cell/B.E. device, but the programming model is general enough to be applied to other multicore processors or symmetric multiprocessors (SMPs).[2] While this section presents an overview of CellSs, the reader is referred to [4] for more detail.

CellSs syntax is based on code annotations, or *pragmas*, inserted in the application code. The current implementation is based on C language, and **Figure 1** shows a sample application with its corresponding pragmas (not all function code is shown, but sample codes can be downloaded from [7]). The application implements an LU factorization. The data structure is a hypermatrix.[3] At the first level, there is a matrix A of size $NB \times NB$ of pointers to floats. Each of these pointers addresses a block of $B \times B$ floats or has a null value to indicate that the block has all elements equal to zero. This strategy allows the easy representation of sparse matrices.

There are three types of pragmas: initialization and finalization pragmas, task pragmas, and synchronization pragmas. The initialization and finalization pragmas (`css start` and `css finish`) are optional and indicate the beginning and end of the CellSs applications. If they are not present in the user code, the compiler will automatically insert the start pragma at the beginning of the application and the finish pragma at the end. The task pragmas are inserted before the code of some functions in the application. The pragma specifies the direction of the parameters: input, output, or input and output; and the size for arrays or matrices. For example, in the case of Figure 1, four functions have been annotated with a task pragma: `lu0`, `bdiv`, `bmod`, and `fwd`. (The code can also have other nonannotated functions.) If we look closely at the pragma of `bmod`, we see that this function has two input parameters (`row` and `col`), which are matrices of size $B \times B$, and an input/output parameter (`inner`), also of size $B \times B$. (An input parameter is read only, while an input/output parameter can be read and written by the function.) The type of parameters (in this case, all of them are floats) is not indicated in the pragma since the compiler will extract this information from the function declaration. If the parameter size was already indicated in the function declaration, then an alternative pragma for this same case would have been the following:

```
#pragma css task input(row,col) inout(inner)
void bmod(float row[B][B], float col[B][B], float
  inner[B][B]){
    ...
}
```

In this case, the compiler would have extracted the size of the matrices and the type from the function declaration. It is important to note that the annotated

[1]VMX is a feature that enables a processor to perform vector (multiple-data) instructions in one step.

[2]An SMP is a homogeneous multiprocessor architecture in which several processors share a main memory.

[3]The representation of matrices as hypermatrices is a current limitation of CellSs but is planned to be overcome in future releases.

functions can access only the parameters and local variables; access to global data is not currently supported. Another limitation of the current version is that the size of the storage required for the parameters should not exceed the size of the local SPE storage; if it does, the application will fail. We plan to extend the system in the future to be able to deal with these features.

The synchronization pragma `css wait` is needed to access data that is generated by annotated functions. In this case, the pragma `css wait` is called before writing the resulting blocks in a file. The code shown in Figure 1 can be compiled with the GNU compiler collection (GCC) and locally tested and debugged in the programmer's workstation before compiling with the CellSs compiler.

The CellSs compiler is a source-to-source compiler. Starting with code that is annotated with pragmas, it generates two files: the main code, which is to be run in the PPE, and the task code, to be run in the SPEs. At the beginning of the main code, the compiler inserts the following application calls to the CellSs runtime: calls to initializing and finalizing functions; calls for registering the annotated functions; and calls to a CellSs runtime library primitive (`css_addTask`) wherever a call to one of the annotated functions is found. For the task code, an adapter function for each annotated function is generated. These adapters are called from the task main code, which is part of the CellSs runtime library. The files are then compiled with the GCC or the IBM XL C compiler to generate a single binary.

A CellSs application is called from the command line as a regular application. The main program of the application (the *master thread*) is run on the PPE. An additional thread (the *helper thread*) also runs in the PPE (**Figure 2**). In the initialization phase, the CellSs runtime starts as threads in the SPEs, with the number of threads being indicated by the user (this is defined with an environment variable). The task program is initiated in these threads. Whenever the main program calls to the `css_addTask` primitive, the master thread adds a node in a task graph that represents that task. It then looks for existing data dependencies between this task and tasks called previously. If a data dependency exists between two tasks, an edge between these tasks is added. Additionally, the master thread performs parameters renaming, a technique that implies the creation of new memory locations for output or input/output task parameters. This renaming technique allows the removal of all WAW (write-after-write) and WAR (write-after-read) dependencies, greatly increasing the parallelism of the task-dependency graphs. The helper thread is responsible for scheduling and further managing the task-dependency graph. Tasks that have no dependency can be scheduled. The helper thread selects an idle SPE and assigns a task to be run in it. The task program running in the SPE waits for assignments from the helper thread. In addition to

```
float *A[NB][NB];

#pragma css task inout(diag[B][B])
void lu0(float *diag){
   int i, j, k;

   for (k=0; k<BS; k++)
      for (i=k+1; i<BS; i++) {
         diag[i][k] = diag[i][k] / diag[k][k];
         for (j=k+1; j<BS; j++)
             diag[i][j] -= diag[i][k] * diag[k][j];
      }
}
#pragma css task input(diag[B][B]) inout(row[B][B])
void bdiv(float *diag, float *row){
...
}
#pragma css task input(row[B][B],col[B][B]) inout(inner[B][B])
void bmod(float *row, float *col, float *inner){
...
}
#pragma css task input(diag[B][B]) inout(col[B][B])
void fwd(float *diag, float *col){
...
}

void
write_matrix (FILE * file, float *matrix[NB][NB])
{
  int rows, columns;
  int i, j, ii, jj;

  fprintf (file, "%d\n %d\n", NB * B, NB * B);
  for (i = 0; i < NB; i++)
    for (ii = 0; ii < B; ii++)
      {
        for (j = 0; j < NB; j++){
#pragma css wait on(matrix[i][j])
          for (jj = 0; jj < B; jj++)
            fprintf (file, "%f ", matrix[i][j][ii][jj]);
        }
        fprintf (file, "\n");
      }

}

int main(int argc, char **argv) {
int ii, jj, kk;
FILE *fileC;
…
initialize (A);
#pragma css start
 for (kk=0; kk<NB; kk++) {
         lu0(A[kk][kk]);
      for (jj=kk+1; jj<NB; jj++)
        if (A[kk][jj] != NULL)
                    fwd(A[kk][kk], A[kk][jj]);
      for (ii=kk+1; ii<NB; ii++)
        if (A[ii][kk] != NULL) {
          bdiv (A[kk][kk], A[ii][kk]);
            for (jj=kk+1; jj<NB; jj++)
              if (A[kk][jj] != NULL) {
                 if (A[ii][jj]==NULL)
                   A[ii][jj]=allocate_clean_block();
                 bmod(A[ii][kk], A[kk][jj], A[ii][jj]);
               }
                    }
   }

  fileC = fopen (argv[3], "w");
  write_matrix (fileC, A);
  fclose (fileC);
  #pragma css finish
}
```

Figure 1

Sparse LU code.

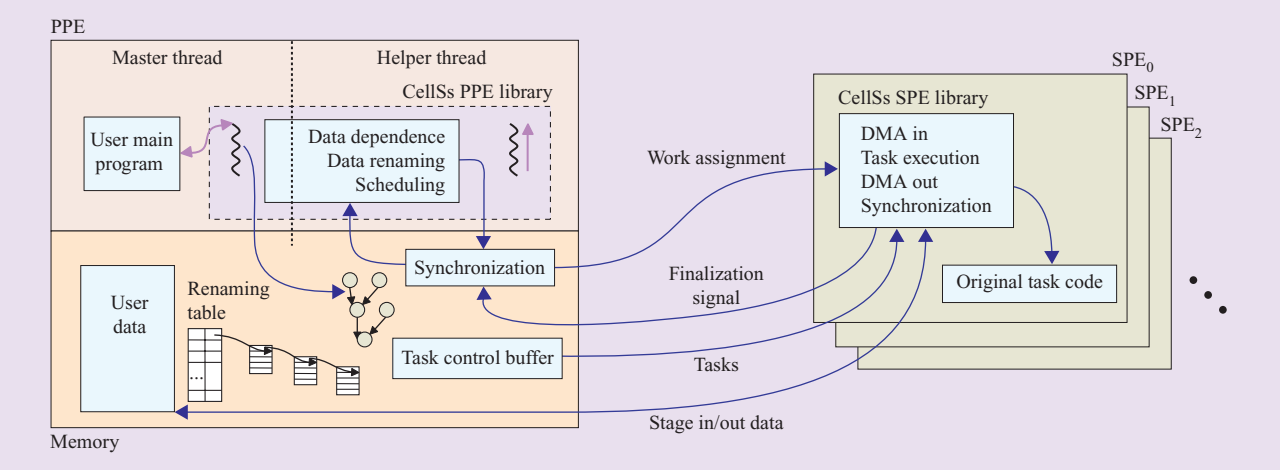


Figure 2

CellSs runtime behavior.

assigning the task, the task program receives information from the helper thread regarding the location of the parameters needed for this task. The data transfers back and forth between the main memory and the SPE local memory are performed by the CellSs SPE library transparently to the user code.

Once the task finishes, the task program notifies the helper thread, which then updates the task-dependency graph according to the current situation and schedules new tasks for execution in idle SPEs. Note that the helper thread is able to concurrently schedule several tasks in different SPEs, thereby exploiting the inherent parallelism of the application at the task (annotated-function) level.

Middleware for the Cell/B.E. processor

CellSs applications are composed of two types of binaries: the main program, which runs in the PPE, and the task program, which runs in the SPE. These binaries are obtained by compilation of the files generated by the CellSs compiler with the CellSs runtime libraries (CellSs PPE library and CellSs SPE library), as described in the overview above.

When the main program is started in the PPE (Figure 2), the task program is launched in each of the SPEs used for this execution. The task program waits for requests from the main program. To execute the annotated functions in the slave SPEs, the runtime prepares all of the necessary data to be transferred to the SPEs, requests the SPE to start a task, and synchronizes with the SPEs to be notified when a task finishes.

When the scheduling policy selects a task from the ready list and selects the SPE resources needed to execute the task, it builds a data structure, the *task control buffer*, that stores all of the information required by the SPE to locate the necessary data for the task. The task control buffer contains information such as the task type identifier and the location of each parameter. The task type identifier allows the SPE to know which task to execute from among all annotated functions. For each of the task parameters, the task control buffer includes the initial address and size of the parameter, as well as some flags indicating such characteristics as type and input or output.

The request from the main program to execute a task in a given SPE is done through a data structure located in main memory. One entry of this structure exists for each of the SPEs. When a task is ready for execution for a given SPE, the main program places the request in the corresponding entry, along with the address and size of the task control buffer.

The behavior of the task program run in the SPE is the following: If the task program is idle, it polls its corresponding entry in the data structure until a task request is detected. It then accesses the task control buffer, which contains all of the data required for the requested task, including data that is in the main memory and not yet transferred to local memory and data that is already in local memory. Once all of the necessary input data has been transferred in, the task is executed in the SPE by calling the annotated task through the generated adapter. When the task is finished, the SPE program transfers the output data to main memory.

Regarding the data alignment requirement in the SPEs, the task program aligns all data in position in multiples of 16 bytes. The task program also synchronizes with the main program when a task execution is finished. This synchronization is done through a main memory position.

The garbage collection of the task control buffers, as with the rest of data structures, is handled by the CellSs PPE library.

CellSs is based on the SPE threads provided with the Cell/B.E. system libraries.

Reducing scheduling overhead

The first attempt to implement the scheduling strategy for CellSs was based on scheduling one task to one SPE at a time. However, this turned out to produce a lot of overhead since the system takes a lot of time to schedule each task. To improve the situation, the tasks are now grouped in bundles that are scheduled to one SPE. To group the tasks, the following heuristics are followed:

- Tasks forming a chain are grouped. A chain is defined as a subgraph in which each task has at most one predecessor and at most one successor. Matrix multiplication is a typical application with such a data-dependence graph. Furthermore, in this case the reused data is not transferred out through the DMA (and it is not transferred in for the next task); only the output data generated by the last task in the bundle is copied back to main memory. This greatly reduces the number of data transfers for some examples. As shown in **Figure 3(a)**, the scheduler groups the tasks of one chain.
- Totally independent tasks are grouped. Some applications show a data-dependence graph with totally independent tasks. In this case, as shown in **Figure 3(b)**, bundles of consecutive tasks in the ready list are grouped.
- For more general data-dependence graph organizations, the scheduler begins from the first task in the ready list and tries to find a chain. However, if this chain does not exist, it takes the next task in the ready list and tries again to find a chain. This is repeated until the group of tasks has a given size. For example, **Figure 3(c)** shows how the scheduler first groups the tasks.

Since all SPEs are symmetric, the assignment of tasks to a given SPE is made according to availability and in a rotational manner.

To reduce the execution time of a bundle of tasks, double buffering is implemented. **Figure 4** shows the sequence of events that is followed to execute a bundle of tasks in the SPE. In this figure, all DMA inputs and outputs are decomposed in two parts: the time invested in programming the transfer and the wait time until the transfer has finished. First, the task control buffer of the bundle of tasks is transferred in. Once this information is available, the data needed for the first task is transferred

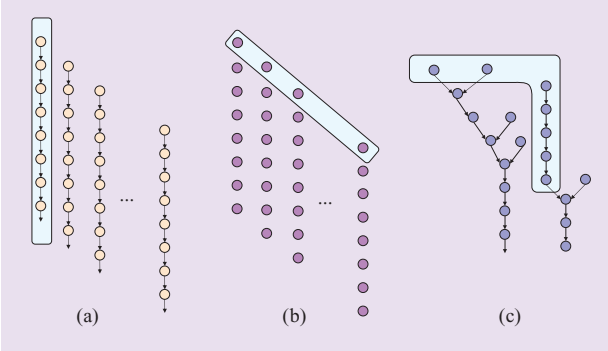


Figure 3

Sample tasks graphs: (a) tasks organized in chains; (b) totally independent tasks; (c) general dependency.

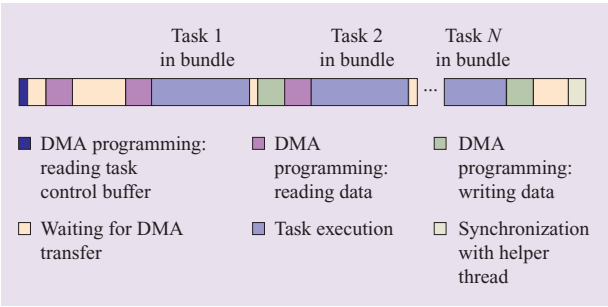


Figure 4

Double buffering in a task bundle.

in. Next, the transfer of the data needed for the second task is programmed, and the execution of the first task is overlapped with this second transfer. After the execution of the first task, a check is made to determine that the second transfer has finished and, if necessary, the DMA out of the output data is programmed. (This step is not needed, for example, for chains.) Then the DMA in of the data needed for the third task is programmed. Next, the execution of the second task is started. This is repeated for all tasks. At the end of the execution of all of the tasks, the DMA out of the output data of the last task in the group is programmed, and the task program waits for all of the DMA outs to finish. Finally, the task program synchronizes with the helper thread to indicate that the bundle of tasks has been executed.

Additionally, a prescheduling strategy has been implemented to reduce the time between the execution of task bundles. While a bundle is executed in a given SPE, the helper thread preschedules a new bundle of tasks to this SPE following the same scheduling strategy. It

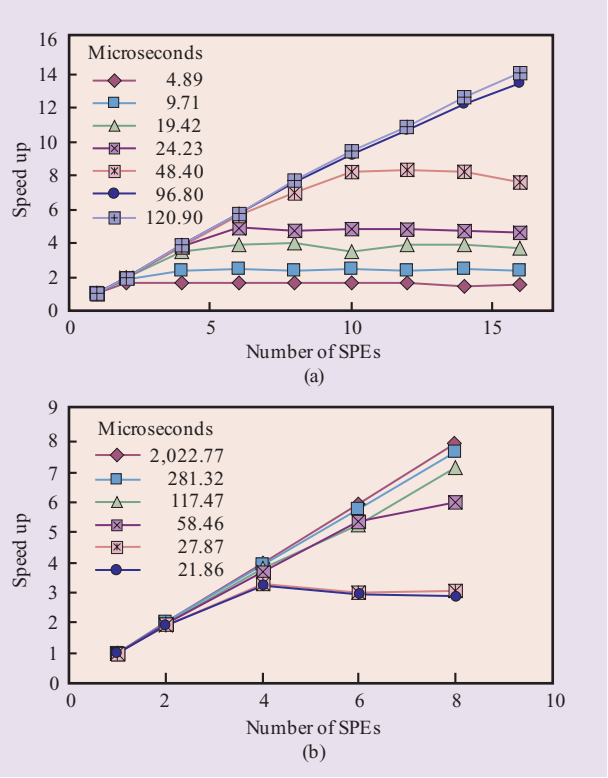


Figure 5

Scalability analysis of CellSs (a) with different task sizes and (b) of matrix multiplication.

does this in such a way that when the SPE finishes the execution of a bundle, a new bundle is ready to be executed. Once a task has been scheduled, it is removed from the ready list (but not from the data-dependence graph). The scheduler will try to preschedule the forthcoming bundle from tasks in the ready list (and their predecessors, in the case when chains exist). The tasks are removed from the data-dependence graph only when the SPE indicates that they have completed execution.

Tracing

A tracing mechanism has been implemented in the CellSs runtime; it generates postmortem trace files of the applications that conform to Paraver [8]. (A Paraver trace file is a collection of records ordered by time where information about the events and states that the application has passed through is stored.) These traces can then be analyzed with the Paraver graphical user interface, making it possible to do performance analysis at different levels, such as at the task level and the thread level. The interface also provides filtering and composing functions that can provide different views of the application, as well as a set of modules to calculate various statistics.

Although Paraver has its own tracing packages for Message Passing Interface (MPI), Open Multiprocessing (OpenMP**), and other programming models, in the CellSs case, a tracing component has been embedded in the CellSs runtime. The tracing component records events as they are signaled throughout the library. For example, it records when the main program enters or exits any function of the CellSs library (as a `css_init`), it records when an annotated function is called in the main program (when `css_registerTask` is called, and therefore, a node is added to the graph) or when a task is started or finished. The traces obtained make it possible to analyze the behavior of the CellSs runtime and that of the application in general. This tracing capability can be enabled or disabled by the user with a compilation flag.

Examples and results

To generate these results, we used the Cell/B.E. platform available at the Barcelona Supercomputing Center–Centro Nacional de Supercomputación (BSC-CNS)–composed four two-way Cell/B.E. processor–based blades: Two are DD2.0 prototypes while the other two are DD3.1 blades. The two prototype DD2.0 Cell/B.E. processor–based blades run at 2.4 GHz and have a total memory of 512 MB Rambus XDR** RAM (256 MB on each Cell/B.E. chip); the two DD3.1 Cell/B.E. processor–based blades run at 3.2 GHz and have a total memory of 1 GB XDR RAM (512 MB on each Cell/B.E. chip). The system runs both for libspe 1.1 and for libspe 2.0 [9].

Scalability analysis

For scalability, we used a simple test application that generates a given number of totally independent tasks. The application starts with a block-matrix, each block of 64×64 floats, and performs a transposition of each of the blocks. The task submitted to the SPE performs this individual transposition of the blocks. To test different sizes of tasks, we artificially increased the size by performing different numbers of iterations. The example has been run with task sizes, measured in microseconds, of 4.89, 9.71, 19.42, 24.23, 48.40, 96.80, and 120.90. **Figure 5(a)** shows the results obtained. As can be observed, the system starts to scale at task sizes of 48.40 μs, at least until eight SPEs, and almost perfectly at task sizes of 96.80 and 120.90 μs. The baseline in the speed-up calculation is the program executing with only one SPE.

Performance results of given examples

This section presents results for some examples. The first example is matrix multiplication. Again, the application operates with block-matrices, each block of 64×64 floats, and for the results given in this section, the matrix is of

16×16 blocks (the final matrix is $1{,}024 \times 1{,}024$ floats). We have tested the system with different versions of this application, from a scalar version (which will be the initial try for a nonexpert programmer) to one using the extremely well vectorized version given in the IBM Software Development Kit (SDK) [9]. In between, other simpler vectorization cases are considered. The range of duration of the block-multiply run in the SPEs goes from 2,022.77 μs in the worst case to 21.86 μs for the SDK case. **Figure 5(b)** shows these results. It is observed how all cases scale quite well with up to four SPEs, but with higher numbers of SPEs, it no longer scales for the two cases with faster tasks.

Additionally, **Table 1** shows the absolute performance values (in Gflops) for some of the cases in this example. With the current version, the higher performance is obtained with the kernel of the SDK and four SPEs (66.36 Gflops). The reason for this behavior becomes clear by analyzing trace files for four and eight SPEs: The master thread is busy adding tasks to the task-dependency graph most of its time (88% of the time). This factor and the task removal (in the helper thread) are the current bottlenecks of the system. This part of the code is especially complex: It requires the synchronization between the master and helper threads that access the same data structures and the different actions that have to be performed in a certain sequence. Currently, we are working on more efficient data structures that can alleviate this problem.

Sparse LU is another example application that has been tested. It presents a more challenging problem to the scheduler than the previous applications, since the task graph of the application is much more complex. **Figure 6(a)** shows the corresponding task graph for a small matrix size of 8×8 blocks. Different colors represent the different task types that are called in this application. **Figure 6(b)** shows the speed up obtained for this example with different matrix sizes (16×16, 32×32, and 64×64 blocks, with a block size of 64×64 floats for all cases). It is observed that the system is able to schedule the problem correctly for big matrix sizes.

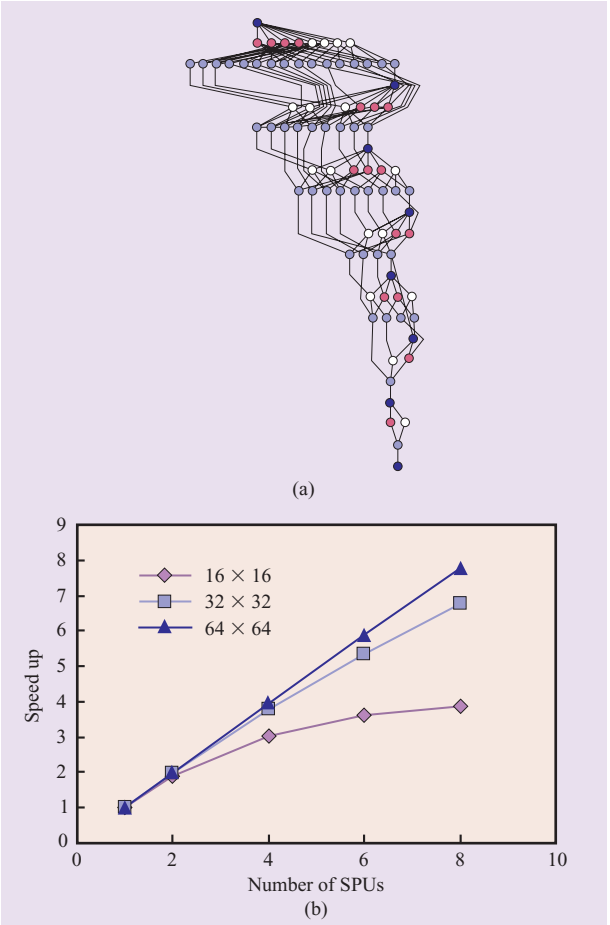


Figure 6

Sparse LU example: (a) task-dependence graph (8×8 matrix case); (b) speed up for the sparse LU case for different matrices.

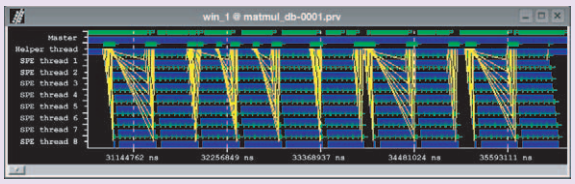
Table 1 Performance results for the matrix multiplication (Gflops).

No. of SPEs	*Task size* (μs)			
	117.47	*58.46*	*27.87*	*21.86*
1	4.29	8.16	19.46	20.48
2	8.34	16.27	37.24	39.24
4	16.46	30.08	63.38	66.36
6	21.28	44.02	58.74	59.81
8	30.81	48.32	59.50	58.88

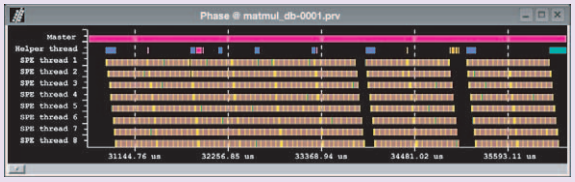
Performance analysis

This section presents an introduction of the type of performance analysis that can be done with the execution trace files extracted by CellSs and Paraver.

For the matrix multiplication example, the elements of the matrices are blocks of 64×64 single-precision floats, while the matrices are of 16×16 blocks. This schema generates 4,096 tasks, each task consisting of a 64×64 matrix multiplication. The 4,096 tasks are organized in chains of 16 dependent tasks, in which each task reads the result of the previous one. The CellSs scheduler is able to identify this graph organization and is able to group parts of these chains into bundles. **Figure 7(a)** shows the first window shown by Paraver when opening a trace file. The timeline is the x-axis, and each of the rows in the y-axis represents a thread of the application. The master and helper threads, shown at the top, are run in the PPE. Each of the remaining threads runs in a separate SPE. The



(a)



(b)

Figure 7

Paraver views: (a) state "as is"; (b) execution phases.

small green flags (vertical hash marks on the horizontal green lines) represent events. The yellow lines represent communications between the SPE threads and the helper thread. The blue color in the threads indicates activity.

With Paraver, the user can configure the type of view of the trace. For example, if a filter is used for a set of events and a semantic value is given to the periods between each event, the view shown in **Figure 7(b)** is obtained. The different colors represent different phases of each thread type. In the SPE threads, yellow indicates a DMA transfer and brown indicates when the SPE is executing the task. The figure shows very clearly how the runtime is able to schedule bundles of several tasks to be executed in the same SPE. Also, the shorter yellow states are due to the fact that the transfers are overlapped with the computations. From this view, the user can also capture the amount of time invested in each of the phases. The format in which data is presented to the user by Paraver is shown in **Table 2**. In this case, for example, the SPEs are busy almost all of the time executing the tasks (~92% of the time); the master thread invests most of its time (~70%) in adding new tasks; and the helper thread is simply waiting for SPE thread events to conclude for ~70% of the time.

Other types of information that can be extracted using Paraver include the following: the number of bytes transferred between main memory and the SPEs, the bandwidths obtained in the transfers, the type of tasks executed, and the identifier of the task, which indicates the order of generation and helps in analyzing task scheduling.

Related work

The IBM Research prototype Cell/B.E. Architecture OpenMP compiler [5] implements techniques for optimizing the execution of scalar code in SIMD units, subword optimization, and other techniques. For example, it implements *autoSIMDization*, which is the process of extracting SIMD parallelism from scalar loops. This feature generates vector instructions from scalar source code for the SPEs and VMX units of the PPE. It is also able to overlap data transfers with computation to enable the SPEs to process data that exceeds the local memory capacity.

Besides the low-level optimizations, this compiler enables the OpenMP programming model [10]. This approach provides programmers with the abstraction of a single shared-memory address space. Using OpenMP directives, programmers can specify regions of code that can be executed in parallel. From a single-body program, the compiler duplicates the necessary code, adds the required additional code to manage the coordination of the parallelization, and generates the corresponding binaries for the PPE and SPE cores. The PPE uses asynchronous signals to inform each SPE that work is available or that it should terminate. The SPEs use a mailbox to update the PPE on the status of their execution. The compiler implements a software cache mechanism to allow reuse of temporary buffers in the local memory, and therefore, there is no need for DMA transfers for all accesses to shared memory. Other features, such as code partitioning, have been implemented to allow applications that do not fit in the local SPE memory.

While the IBM approach is focused on OpenMP, we see CellSs as an alternative programming model to OpenMP. It is our view that CellSs is likely more flexible and can be used for a different range of applications. By contrast, OpenMP is primarily focused on the parallelization of fine-grain numerical loops. CellSs does, however, rely on the IBM compiler (or others that offer similar features) for autoSIMDization, the automatic vectorization of the SPE code. CellSs can also rely on GCC, for which similar autovectorization features are being implemented by a research group at the IBM Haifa Laboratory [6].

In the IBM Roadrunner system [11], the AMD Opteron processors are host elements and the Cell/B.E. blades are accelerator elements for the Opteron hosts. At the level of the Cell/B.E. blades, the PPE processors are hosts and the SPEs are accelerators.

The Accelerated Library Framework (ALF) [12] application programming interface (API) provides a set of functions to help programmers solve data-parallel problems on a hybrid system. ALF supports the single-program multiple-data (SPMD) programming style with a single program running on all accelerator elements at one time. ALF offers programmers an interface to easily partition data across a set of parallel processes without having to write architecturally dependent code. Its

Table 2 Data presented in Paraver view of the time invested in each phase by each thread.

	Master (%)	*Helper thread* (%)	*SPE thread 1* (%)	*SPE thread 2* (%)	*SPE thread 3* (%)	*SPE thread 4* (%)
Return to user code	4.73	—	—	—	—	—
Adding task	69.51	—	—	—	—	—
Schedule	—	7.36	—	—	—	—
Prepare bundle	—	2.56	—	—	—	—
Prepare bundle submission	—	0.76	—	—	—	—
Submit bundle	—	1.97	—	—	—	—
Attend task finished	—	4.09	—	—	—	—
Remove tasks	25.76	12.51	—	—	—	—
Low-level wait for events	—	70.76	—	—	—	—
Waiting for tasks	—	—	4.20	4.07	3.94	4.03
Getting task description	—	—	0.09	0.08	0.09	0.09
Task stage in	—	—	0.90	0.90	0.88	0.88
Task arguments alignment	—	—	0.27	0.25	0.27	0.28
Task execution	—	—	91.31	91.33	91.30	91.30
Task stage out	—	—	0.43	0.42	0.44	0.44
Task finished notification	—	—	0.05	0.05	0.05	0.05
Wait for DMA	—	—	2.76	2.89	3.03	2.94
Total	100.00	100.00	100.00	100.00	100.00	100.00
Average	33.33	14.29	12.50	12.50	12.50	12.50
Maximum	69.51	70.76	91.31	91.33	91.30	91.30
Minimum	4.73	0.76	0.05	0.05	0.05	0.05
Standard deviation	26.98	23.35	29.82	29.83	29.81	29.82
Coefficient of variation	0.81	1.63	2.39	2.39	2.39	2.39

features include data transfer management, parallel task management, double buffering, and data partitioning. The programming model is based on work queues. In ALF, a task is a function that receives a parameters list and a work packet identifier. The main program opens the parallelism by creating several work packets for a task and then waits until all of the work packets have finished (with a barrier). ALF is currently distributed with the IBM SDK 2.0. From our point of view, ALF is at a lower level than CellSs, and there is a possibility that CellSs based on ALF can be implemented.

Some similarities to CellSs can be found with thread-level speculation, especially as in the work proposed by Zhai et al. [13]. This work tries to solve the performance limitation introduced by the forwarding of scalar values between threads. In this approach, the compiler inserts explicit synchronization primitives (wait and signal) to communicate scalars that are used by multiple threads. These synchronization points define dependencies among the threads similar to the data-dependency analysis performed by CellSs. The difference is that in the former approach, the problem is tackled at compile time, and in CellSs, at execution time.

Sequoia [14] is a programming language based on C++. Similar to CellSs, it is based on the decomposition of 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.

Programs written in Sequoia contain the SPE code and the PPE code. The language has a construct that allows specifying loops with independent iterations. It also has a reduction construct and a sequential loop construct. These constructs are handled by the runtime library and are the only elements in Sequoia that determine the task-level parallelism. They also serve as barriers, which, in practice, makes them similar to basic parallel constructs from OpenMP without the `nowait` clause, and they are not very well suited for complex irregular algorithms. Similar to CellSs, the Sequoia runtime hides data transfer latency by overlapping computation and data transfers.

RapidMind [15] is another programming model for the Cell/B.E. 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 SPE code from within the PPE code. All SPE code is written using the template library.

The RapidMind C++ template library provides a set of data types, control flow macros, reduction operations, and common functions that allow the runtime library to capture a representation of the SPE code (retained code). The library data types have been designed to easily express SIMD operations and to pass those operations to the runtime library. The runtime in turn extracts parallelism from those operations by vectorizing the code and by splitting array and vector calculations on the different SPEs. It also has an optimizer that can perform loop invariant removal. Similar to CellSs, RapidMind assigns work to the SPEs dynamically and hides data transfer latency by overlapping computation and data transfers. Although RapidMind provides a good framework for programming the Cell/B.E. processor, CellSs offers a higher level and more flexible programming model.

Ohara et al. [16] present a new programming model, MPI microtask, based on the standard MPI programming model for distributed-memory parallel machines. In this model, programmers do not need to manage the SPE local store as long as they partition their application into a collection of microtasks that fit into the local store. Furthermore, preprocessor and runtime in this microtask system optimize the execution of the microtasks by exploiting explicit communications in the MPI model. There is a prototype available that includes a novel static scheduler for such optimizations. Initial experiments show encouraging results.

At the architectural level, the Cell/B.E. processor can be compared with SMPs. However, in the latter architecture, the threads share the memory space, and therefore, data can be easily shared, although a synchronization mechanism should exist to avoid conflicts. The Cell/B.E. SPEs have local memory, and data must be copied there by DMA before performing any calculation with them. This is one of the fundamental characteristics of the Cell/B.E. Architecture that makes its programming a challenge.

Another architecture with similarities to that of the Cell/B.E. is one that uses graphics processor units (GPUs). The latest series of this type of hardware presents a great potential for parallelism, with up to six vertex processors and up to 16 pixel processors. Although these processors are initially very specialized, because of their excellent performance/cost ratio, different approaches to use them in applications beyond their specialized graphical use have been presented [13, 17, 18]. GPUs can be programmed by means of C for Graphics (Cg) [19], a C-like high-level language. However, to be able to use the GPUs for general-purpose computation, there are some requirements that must be met, for example, the fact that the data structure must be arranged in arrays in order to be stored in the specific structures of the GPU. This is easy for applications dealing with matrices or arrays, but it is more difficult to accommodate data structures such as binary trees and to accommodate general programs that use pointers. Another limitation is that the computation may be inefficient in cases in which the program control flow is complex. To our knowledge, there are no approaches to hide these complexities from the programmers with environments equivalent to CellSs.

Conclusions and future work

This paper presents CellSs, an alternative to traditional parallel programming models. The objective is to be able to offer a simple and flexible programming model for parallel and heterogeneous architectures. Following the CellSs paradigm, input applications can be written as sequential programs. This paradigm is currently customized for the Cell/B.E. Architecture. The runtime builds a task-dependency graph of the calls to functions that are annotated in the user code and schedules these calls in the SPEs, handling all data transfers from and to the SPEs. In addition, a locality-aware scheduling algorithm has been implemented to reduce the amount of data that is transferred to and from the SPEs.

Although CellSs has some similarities with OpenMP because of the fact that both use pragmas to annotate the code, the semantics behind CellSs are quite different. In OpenMP, the programmer explicitly indicates what is parallel and what is not, while in CellSs, the programmer identifies pieces of code that are independent of one another. Thus, by annotating a function, we are not saying that it can be run in parallel: The system will be able to find the inherent parallelism of the application by building a data-dependency graph of the actual calls to the annotated functions. CellSs provides programmers

with a more flexible programming model with an adaptive parallelism level depending on the application input data. OpenMP is more appropriate for applications with parallelism at the loop level, while CellSs fits better for applications with parallelism at the function level. However, given the current 3.0 extensions of OpenMP, which define the concept of task, both programming models can converge into one in the future. We plan to perform the integration of OpenMP and CellSs and develop other extensions.

The initial results are promising, but there is a lot of work left, for example, the development of new annotations to be taken into account by the source-to-source compiler, improvement in the scheduling and data handling, and in general, improvement of the runtime performance.

Although the approach presented in this paper is focused on the Cell/B.E. Architecture, we consider CellSs to be generic enough for use (after tailoring the middleware) with other multicore architectures.

Acknowledgments

This work has been partially supported by the Comisión Interministerial de Ciencia y Tecnología (CICYT) under contract TIN2004-07739-CO2-01 and by the BSC-IBM Master R&D Collaboration agreement.

*Trademark, service mark, or registered trademark of International Business Machines Corporation in the United States, other countries, or both.

**Trademark, service mark, or registered trademark of Advanced Micro Devices, Inc., Intel Corporation, ClearSpeed Technology plc, AGEIA Technologies, Inc., OpenMP Architecture Review Board, or Rambus, Inc., in the United States, other countries, or both.

†Cell Broadband Engine is a trademark of Sony Computer Entertainment, Inc., in the United States, other countries, or both.

References

1. D. Geer, "Chip Makers Turn to Multicore Processors," *Computer* **38**, No. 5, 11–13 (2005).
2. K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, et al., "The Landscape of Parallel Computing Research: A View from Berkeley," *Technical Report EECS-2006-183*, University of California at Berkeley, 2006.
3. D. Pham, S. Asano, M. Bolliger, M. N. Day, H. P. Hofstee, C. Johns, J. Kahle, et al., "The Design and Implementation of a First-Generation Cell Processor," *Proceedings of the IEEE International Solid-State Circuits Conference*, San Francisco, CA, 2005, pp. 184–185.
4. P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta, "CellSs: A Programming Model for the Cell BE Architecture," *Proceedings of the ACM/IEEE SC2006 Conference on High Performance Networking and Computing*, Tampa, FL, 2006, p. 86.
5. A. E. Eichenberger, J. K. O'Brien, K. M. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, et al., "Using Advanced Compiler Technology to Exploit the Performance of the Cell Broadband Engine™ Architecture," *IBM Systems J.* **45**, No. 1, 59–84 (2006).
6. Cell Broadband Engine Technology, GNU GCC Tools Upgraded to Version 4.1.1, IBM alphaWorks; see *http://www.alphaworks.ibm.com/topics/cell*.
7. Code Example, Barcelona Supercomputing Center, 2007; see *http://www.bsc.es/plantillaH.php?cat_id=183*.
8. J. Labarta, S. Girona, V. Pillet, T. Cortes, and L. Gregoris, "DiP: A Parallel Program Development Environment," *Proceedings of the 2nd International Euro-Par Conference on Parallel Processing*, Lyon, France, 1996, pp. 665–674.
9. IBM Corporation, *IBM Software Development Kit (SDK) IBM Cell Broadband Engine Software Development Kit;* see *http://www.alphaworks.ibm.com/tech/cellsw*.
10. The Community of OpenMP Users, Researchers, Tool Developers, and Providers; see *http://www.compunity.org/*.
11. IBM Corporation (September 6, 2006). IBM to Build World's First Cell Broadband Engine Based Supercomputer. Press release; see *http://www-03.ibm.com/press/us/en/pressrelease/20210.wss*.
12. IBM Corporation, *Software Development Kit 2.1 Accelerated Library Framework Programmer's Guide and API Reference, Version 1.1*; see *http://df.unife.it/u/belletti/sdkdocs/ALFProgrammersGuideAndAPIRef_v1.1.pdf*.
13. A. Zhai, C. B. Colohan, J. G. Steffan, and T. C. Mowry, "Compiler Optimization of Scalar Value Communication Between Speculative Threads," *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, 2002, pp. 171–183.
14. K. Fatahalian, T. J. Knight, M. Houston, M. Erez, D. R Horn, L. Leem, J. Y. Park, et al., "Sequoia: Programming the Memory Hierarchy," *Proceedings of the ACM/IEEE Conference on Supercomputing*, Tampa, FL, 2006, p. 4.
15. RapidMind, Inc., "Cell BE Porting and Tuning with RapidMind: A Case Study," white paper; see *http://www.rapidmind.net/case-cell.php*.
16. M. Ohara, H. Inoue, Y. Sohda, H. Komatsu, and T. Nakatani, "MPI Microtask for Programming the Cell Broadband Engine™ Processor," *IBM Systems J.* **45**, No. 1, 85–102 (2006).
17. T. Yang and A. Gerasoulis, "A Fast Static Scheduling Algorithm for DAGs on an Unbounded Number of Processors," *Proceedings of the ACM/IEEE Conference on Supercomputing*, Albuquerque, NM, 1991, pp. 633–642.
18. N. Galoppo, N. K. Govindaraju, M. Henson, and D. Manocha, "LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware," *Proceedings of the ACM/IEEE Conference on Supercomputing*, Seattle, WA, 2005, p. 3.
19. Z. Fan, F. Qiu, A. Kaufman, and S. Yoakum-Stover, "GPU Cluster for High Performance Computing," *Proceedings of ACM/IEEE Conference on Supercomputing*, Pittsburgh, PA, 2004, p. 47.

Received March 14, 2007; accepted for publication March 28, 2007; Internet publication August 17, 2007

Josep M. Perez *Barcelona Supercomputing Center, Jordi Girona 29, 08034 Barcelona, Spain (josep.m.perez@bsc.es).*
Mr. Perez is a Researcher at the Barcelona Supercomputing Center. He holds an M.S. degree from the Technical University of Catalonia (Universitat Politècnica de Catalunya, or UPC) in Barcelona, where he is a doctoral candidate in computer science under the guidance of Dr. Rosa M. Badia. Mr. Perez's research interests include Grid programming models and multicore programming models.

Pieter Bellens *Barcelona Supercomputing Center, Jordi Girona 29, 08034 Barcelona, Spain (pieter.bellens@bsc.es).* Mr. Bellens is a Researcher at the Barcelona Supercomputing Center. He holds computer science and engineering degrees from the Katholieke Universiteit Leuven, Belgium, and is a doctoral candidate in computer science at the Technical University of Catalonia (UPC), under the guidance of Dr. Rosa M. Badia and Dr. Jesús Labarta. Mr. Bellens' research interests include parallel computing, the Cell/B.E. processor, and scheduling.

Rosa M. Badia *Barcelona Supercomputing Center, Jordi Girona 29, 08034 Barcelona, Spain (rosa.m.badia@bsc.es).*
Dr. Badia is the Manager of Grid computing and clusters at the Barcelona Supercomputing Center. She is also an Associate Professor at the Computer Architecture department of the Technical University of Catalonia (UPC). She received B.S. and Ph.D. degrees in computer science from the Technical University of Catalonia in 1989 and 1994, respectively. Since 1989, she has been lecturing at the UPC on computer organization and architecture and very-large-scale integration (VLSI) design, in both undergraduate and graduate programs. Her current research interests include performance prediction and modeling of message-passing interface programs, programming models for the Grid, resource management in the Grid, and programming models for multicore processors. Dr. Badia has participated in several international research projects.

Jesús Labarta *Barcelona Supercomputing Center, Jordi Girona 29, 08034 Barcelona, Spain (jesus.labarta@bsc.es).*
Professor Labarta is the Director of high-performance computing research at the Barcelona Supercomputing Center. He has been a full professor at the Computer Architecture department at the Technical University of Catalonia (UPC) since 1990. He has been lecturing on computer architecture, operating systems, computer networks, and performance evaluation since 1981. From 1995 to 2004 he was Director of Centro Europeo de Paralelismo de Barcelona (CEPBA) at the UPC. His research interest has been centered on parallel computing, covering areas from multiprocessor architecture, memory hierarchy, parallelizing compilers, operating systems, parallelization of numerical kernels, metacomputing tools, and performance analysis and prediction tools. Dr. Labarta has led the technical work of 15 industrial research and development projects at UPC.