Proc. 21st PARS - Workshop, Hamburg, Germany, May 31-Jun 1, 2007.
GI/ITG-Fachgruppe Parallel-Algorithmen, -Rechnerstrukturen und -Systemsoftware (PARS).
PARS-Mitteilungen **24, ISSN 0177-0454, GI, Dec. 2007**

Bulk-Synchronous Parallel Computing on the CELL Processor

Daniel Johansson, Mattias Eriksson, and Christoph Kessler

PELAB, Dept. of Computer Science (IDA)
Linköping university, Sweden
{x06danjo,mater,chrke}@ida.liu.se

Abstract. In order to ease programming of heterogeneous architectures with explicitly managed memory hierarchies such as the CELL processor, we propose a solution adopting the BSP model as implemented in the parallel programming language NestStep. This allows the programmer to write programs with a global address space and run them on the slave processors (SPEs) of CELL while keeping the large data structures in main memory. We have implemented the run-time system of NestStep on CELL and report on performance results that demonstrate the feasibility of the approach. The test programs scale very well as their execution time is mostly dominated by calculations, and only a fraction is spent in the various parts of the NestStep runtime system library. The library also has a relatively small memory footprint in the SPE's software managed local memory.

Key words: Cell processor, NestStep, bulk-synchronous parallel programming, global address space, run-time system

1 Introduction

The Cell Broadband Engine^{TM}, developed as a joint effort by IBM, Toshiba and Sony, is a heterogeneous multicore system. On a single chip, it accommodates

– one master processor core (called the *PPE*), which is a 64-bit dual-thread SMT PowerPC with level-1 and level-2 cache;
– 8 slave processor cores (called *Synergistic processing elements* or *SPE*s), which are 32-bit RISC processors with SIMD instructions operating on 128-bit data items per cycle; each SPE is equipped with a DMA controller and a local memory buffer of 256 KB for storing both data and code;
– a ring-based high-bandwidth interconnection network, and
– controllers for accessing (off-chip) main memory and I/O channels.

IBM provides a Linux-based operating system for Cell that runs on the PPE. Even for the Cell processor versions used in the recent Sony PlayStation-3^{TM}, the game operating system can be replaced by Linux. IBM also provides a complete development environment and a Cell simulator [3]. Using the (slower) cycle-accurate simulation mode in the simulator, it is possible to evaluate the performance of Cell programs even without having a hardware version of the Cell processor at hand.

The SPEs cannot access main memory directly; instead, each of them has a DMA controller that allows to move blocks of data and instructions between main memory and the SPE's local memory buffer. Parallelism resulting from overlapping DMA and SPE computation in time is one of several concerns in exploiting the performance potential of the Cell processor [1]. Other sources of parallelism that the programmer (or a parallelizing compiler) should exploit are the data parallelism provided by the SIMD instructions in the SPEs, and the task parallelism by running the PPE and/or eight SPEs in parallel.

Of course, additional parallelism potential could be gained by aggregating several Cell processors, e.g. in the IBM blade servers consisting of two Cell processors sharing one copy of the operating system [4], and racks containing 7 blades. A Cell-based supercomputer ("Roadrunner") with 16000 Opteron and 16000 Cell processors is planned by the Los Alamos National Laboratory [5], which accumulates to a theoretical peak performance of 1.6 petaflops. A new Cell generation with more PPE and many more SPE cores per chip is already planned.

Unfortunately, programming the Cell processor to utilize these sources of parallelism effectively is not an easy task. Communication and synchronization between PPE and SPEs is done by mailboxes and signals, and communication between SPEs and memory by DMA transfers; moreover, the local memory buffers of the SPEs are not included in the global address space. Hence, programming at this level reminds rather of a message passing system than a SMP chip multiprocessor.

Eichenberger *et al.* [1] proposed a virtual shared memory implementation across PPE and all SPEs by utilizing (part of) the local SPE memory buffers as software-controlled caches. This implies DMA traffic for each cache miss and also for maintaining cache coherence across the SPEs and the PPE. To get more efficient code, they use static analysis to minimize cache usage and optimize the placement of data transfers: by prefetching, software pipelining to overlap data move and computation, by improving data locality by loop tiling, and by agglomerating several DMA accesses into larger ones. In this way, they provided a SMP platform on which OpenMP programs can be executed.

In this paper, we propose an alternative way of realizing a shared address space on Cell. To control deterministic parallel execution, synchronization and memory consistency, we adopt the bulk-synchronous parallel (BSP) programming model [12]. The BSP model postulates that a parallel computation be structured as a barrier-separated sequence of supersteps, each consisting of a local computation phase and a global communication phase. The parallel programming language *NestStep* developed in earlier work [7, 8] extends the BSP model, replacing message passing by the emulation of a Combining CRCW BSP with a shared address space, realized by the compiler and the run-time system on top of a message-passing system. NestStep supports shared variables, shared objects and distributed shared arrays with block-cyclic distributions, in addition to ordinary thread-private variables. Memory consistency will be re-established at the end of a superstep; for each shared variable and array element, the policy of achieving consistency and resolving concurrent writes in a deterministic way within the same superstep can be programmed. In particular, reduction and multiprefix computations are supported. As superstep consistency must also hold for distributed shared arrays, read accesses to non-local array elements must be ordered by explicit prefetch statements at the end of the preceding superstep, and write accesses to non-local array elements are committed at the end of their superstep. The necessary communication for array accesses is scheduled at run-time by a BSP-compliant parallel version of the inspector-executor technique; the overhead for this is kept low by packing the access messages on top of the consistency-restoring and synchronizing messages at the end of each superstep [8].

Although being more restrictive compared to general shared memory programming in OpenMP, we see from a long-term perspective an advantage of using the NestStep/BSP model in its simplicity and scalability: Clusters consisting of multiple Cells connected by a message passing network could be programmed in NestStep in the same uniform way as a single Cell, while OpenMP basically relies on the existence of a shared memory platform and would need to be combined with MPI to a hybrid programming model.

We implement NestStep on Cell such that all computational NestStep code is running on the SPEs, while the PPE takes care of launching SPE threads and manages memory allocation/deallocation, committing combined values to main memory, and synchronization. The core of the NestStep run-time system on Cell is a loop running on the PPE polling combine request messages from the SPE mailboxes, handling them and then sending a message back to the SPEs with updated values of modified shared variables, telling them to continue executing. We find a quite large bottleneck in the SPE's 256 KB local store; for instance, we cannot store large array variables locally, and even the program size might become an issue. We have chosen to handle these problems by allowing the programmer to split a NestStep program into several parts that are run after each other, and also by storing shared variables in main memory and only use the local store as a kind of programmer/compiler-managed cache. We even introduced a special kind of global private variables that are stored in main memory, such that their state is preserved even when changing between consecutive SPE program parts.

We keep one data area dedicated for each SPE in main memory to store home versions of all variables created by a specific SPE: its distributed shared array segments, replicated shared variables and arrays, private variables stored in main memory, and consistency management structures such as combine lists communicated at the end of a superstep.

The cluster implementation of the NestStep runtime system [11] uses combine trees for handling the combining of updates to shared variable and replicated arrays at the end of a superstep. On Cell we adopted instead a flat combining mechanism, dedicating the PPE to act as central manager for combining. However, in future versions of Cell with many more SPE cores, trees should be used for combining as they are more scalable, distributing most of the combining work across the SPEs.

The remainder of this article is organized as follows: After illustrating some central NestStep language concepts in Section 2, we describe the structure of the NestStep runtime system for Cell in some more detail in Section 3. The experimental evaluation of the prototype implementation is described in Section 4, Section 5 discusses related work, and Section 6 concludes and lists issues for future work.

2 NestStep by example

We illustrate some of NestStep's language concepts by considering a simple example, parallel dot product. The following general version does not yet consider the memory hierarchy of Cell:

```
sh float s;   // shared variable – one local copy per processor
sh float a[N] < / >;   // block-wise distributed shared array
sh float b[N] < / >;   // block-wise distributed shared array
int i;   // private variable
...
step {    // executed by a group of processors
   forall (i, a)    // iterate over owned elements of a
      s = s + a[i] * b[i];
} combine ( s < + > );
...
```

The above example code contains one superstep (marked by the **step** statement) that is terminated by a global communication phase and an implicit barrier synchronization. The communication phase will restore consistency of all copies of a shared variable (here, s) written in the superstep. By the **combine** clause, the programmer can, for each variable, override its default combine policy by specifying how this should be done, here for instance by writing the global sum of all values of modified copies of s, denoted by $s <+>$; several other combine policies are also predefined in NestStep, and even user-defined combine functions are possible. Arrays a and b are block-wise distributed shared arrays, thus each of the p processors that see their declaration owns a contiguous partition of approximately size N/p exclusively. The **forall**(i, a) loop lets each executing processor's i variable iterate over its owned elements of a, such that all elements of a and b are read in the example above. After the superstep, s holds the global sum of all per-processor contributions, i.e., the dot product of a and b.

To run this example for large N efficiently on Cell with its size-limited software managed local store on each SPE, the baseline code needs to be extended for strip mining, by explicit management of buffers in the local stores and DMA statements to pre-fetch operands and, where necessary, poststore computed results, to overlap DMA transfers with computation. The resulting NestStep-Cell (source) code has 233 lines of code and can be found in [6].

3 NestStep-Cell Run-time System Structure

The Cell processor is different from previous targets for NestStep, mainly because of its very limited size of the local store. Because of this we need to extend the NestStep run-time system (a C library providing a kind of intermediate language for NestStep) in a few ways.

To be able to create programs that themselves are larger than the local store, we give the programmer (or a future Cell-aware NestStep frontend compiler) the possibility to store *all* NestStep variables in main memory and to read in only those that are currently being worked on to the smaller local memory for each node. We also support that a large program be split into several smaller program units that can run in sequence, where the values of variables are preserved between subsequent units.

Since the PPE has direct access to the memory while the SPEs do not, we decided to do all combining and calculations of combine functions (e.g., addition as in the example above) on the PPE. We are aware that this sequentializes the combining process, which will have a performance impact when combining large amounts of data. However, this makes implementation much easier, as we do not need to allocate buffers for system functions inside the already small SPE local store, and as we do not need to split combines into small workable chunks.

As a consequence, we decided to remove the possibility to have user-defined combine functions since the NestStep code runs on the SPEs and combining runs on the PPE. The alternative would be to either define the combine functions inside the PPE block of code and manage pointers into those from the SPEs, or move the combining to run on the SPEs.

In Fig. 1 we can see how the memory is managed in NestStep-Cell. We have a memory manager data structure both in main memory and in the LS of the SPE for each SPE thread. The memory manager is identical in LS and in main memory. Inside this data structure we store pointers to the location of the variables in the current scope of visibility, as well as metadata about how large distributed shared arrays are, their element type and so on. In the case of distributed shared arrays, the array is created in parts, one part per thread, that together form the entire data structure. In the memory managers we also store information about the owned range for the array, the size of the owned partition. In the access functions for reading and writing distributed shared array elements we have logic for calculating partition-local indices from their global counterparts and vice versa. In the case of replicated shared variables, we store one copy of the variable for each thread. For instance, in the dot product example above, the shared float variable s actually exists in 8 instances in main memory, one instance for each SPE thread. This replication is necessary to guarantee the BSP-compliant deterministic memory consistency model of NestStep.

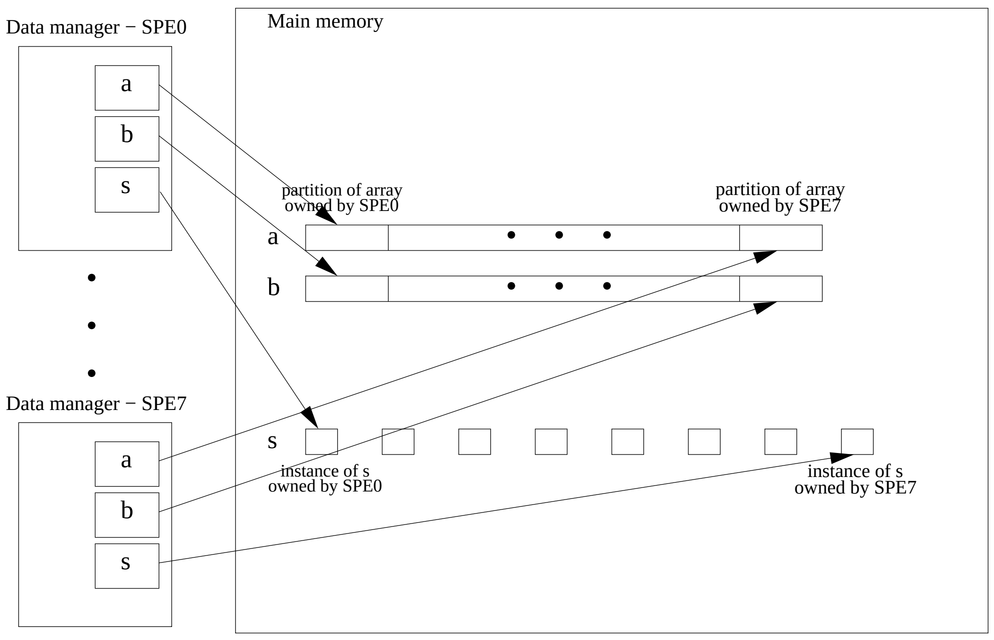

Data manager − SPE0
a
b
s
Main memory
partition of array owned by SPE0
partition of array owned by SPE7
a
b
s
instance of s owned by SPE0
instance of s owned by SPE7
Data manager − SPE7
a
b
s

Fig. 1. One memory manager for each SPE exists in main memory, with a copy in the SPE's local store. For a block distributed shared array, the memory manager points to the owned partition of the array that belongs to the corresponding SPE. For a shared variable, the memory manager points to the local copy of that variable that belongs to the SPE.

3.1 Implementation details

A NestStep-Cell program consists of PPE binary code and the SPE binaries, which are supposed to run alternatingly: The PPE program will begin by initializing the PPE part of the runtime-system, i.e., the memory managers on the PPE. It then loads the first SPE binary to all allocated SPEs and enters a waiting loop that polls the SPEs' mailboxes for messages. If the PPE finds a message (usually a request for combining or for allocating memory), it will handle it and continue polling the mailboxes. This will continue until the PPE has received a `DESTROY_THREAD` message from each SPE; then the PPE exits the loop and executes the next SPE binary that was linked in. When all the SPE binaries have terminated, the PPE cleans up its data structures and exit.

This message passing is done through the use of Cell's MFC mailboxes. These mailboxes only support passing of integers, therefore we have chosen to pass an integer value referring to a separate message data structure in the LS that the SPE thread can write into to pass additional data, such as the name of an array that is to be created, and the length of the array.

On the SPE side, the program will begin with a call to `NestStep_SPU_init(Addr64 argp)`. This function takes the address of the memory manager for this thread as an argument. The address is passed through the `main` function and has to be sent to the initializing function explicitly. The memory manager is then DMA'd to the SPE's LS. After this, the program is set up and can start executing user-level code, i.e. supersteps, requesting combining and other PPE services as encountered in the compiled NestStep code. Finally, the SPE program has to end with a call to `NestStep_SPU_finalize()` that sends the `DESTROY_THREAD` message to the PPE.

3.2 Specific constructs

With the addition of the possibility to have NestStep programs spanning several SPE programs, prefix variables being handled on the PPE, and the remote read/write requests being done on the PPE, we need to have private variables stored in main memory, too. This made us introduce two new data types to the NestStep runtime system, the *Private variable* (a private NestStep variable residing in main memory) and the *Private array* (a private array residing in main memory). These are very similar to shared variables and replicated shared arrays, respectively, in the sense that they reside in main memory, but with the difference that these can not be used in combine functions.

To manage NestStep's special data types and transfer them to and from main memory, we provide the following functions:

- `DMA_read_variable( lsaddr, var )`
  loads a variable `var` from main memory into local store at address `lsaddr`.
- `DMA_read_array( lsaddr, arr, lbound, ubound )`
  loads a contiguous section of an array `arr` (delimited by global indices `lbound` and `ubound`) from main memory into local store.
- `DMA_write_variable( lsaddr, var )`
  stores a variable from local store into main memory.
- `DMA_write_array( lsaddr, arr, lbound, ubound )`
  stores an array from local store into main memory.
- `DMA_wait( lsaddr, var )`
  waits for completion of an asynchronous DMA transfer of a variable.
- `DMA_wait( lsaddr, arr )`
  waits for completion of an asynchronous DMA transfer of an array.

These are all mapped to asynchronous DMA-read and DMA-write calls. On Cell, DMA transfer requests have a few restrictions on their parameters. If a transfer is larger than 16 byte, it needs to be 16-byte aligned, it needs to be a multiple of 16 byte in size, and it can be at maximum 16384 bytes in size. If a transfer is smaller than 16 byte, it can be naturally aligned (4, 8 or 16-byte aligned, depending on the transfer size). We allow 30 concurrent data transfers at a time, and the program has to do an explicit wait on all transfers in order to clear the DMA transfer flags.

3.3 Communication structure

Because we assigned combining work (including handling prefix variables) to the PPE, and because the current Cell processor only has at most 8 SPEs [1] we have chosen to remove the tree-based implementation of combining used in the NestStep runtime system for MPI clusters [11], and only use a flat tree for combining.

The NestStep construct `mirror( a[`l`:`r`] )` denotes a bulk prefetch operation for all remote elements in the section between (global) index positions l and r of a distributed shared array `a`, in order to make them available as a local copy at the entry of the following superstep. Likewise, the construct `update( a[`l`:`r`] )` denotes a bulk poststore operation for written local copies that should be committed to their remote home location at the end of the current superstep. `Mirror` instructions are necessary to enforce the BSP memory model for distributed shared arrays (accessing unmirrored remote elements would cause a run-time error) while `update` is just a hint by the programmer for the compiler to coordinate remote updates of the same array, avoiding many separate one-element update requests.

The handling of `mirror` and `update` requests has also been moved to the PPE since, on Cell, this just amounts to `memcpy` operations.[2] It might be faster if it could be managed via MFC; looking into this is a topic for future work.

[1] The IBM roadmap depicts Cell processors with at most 32 SPEs for the coming years.

[2] A parallelized inspector-executor realization of bulk remote array accesses on a cluster NestStep implementation was presented in earlier work [8].

4 Evaluation

We report on first results that should demonstrate how well NestStep's global-address-space BSP model can be implemented on a heterogeneous multicore architecture such as the Cell processor with our approach. Our test programs were initially benchmarked on a Sony PlayStation-3, which supports 6 SPEs on Cell. Later we got access to an IBM QS20 which allowed us to scale up to 8 SPEs. We did no benchmarking on the Cell System Simulator since the time for cycle-accurate simulation would be too high for larger problem instances. We used no SIMD operations in our test programs, thus the theoretical maximum performance in any test should be 1/4 of peak performance, or 50 GFLOPS, for single precision and 1/2 of peak performance, or 3 GFLOPS, for double precision. We did not try to optimize code outside the NestStep runtime system library, and thus the number of FLOPS achieved is less meaningful; instead, a comparative analysis showing how much time was spent in the different parts of the program should be more illustrative. Optimal behavior is that the program only spends time in the calculation part of the program. The time spent in DMA-wait is caused by some data transfers not being complete yet when the calculation should be done. Since the combining mechanism also acts as a barrier synchronization, time spent here can stem from load imbalance in a superstep, i.e., some calculations or data transfers on one SPU might have taken longer time to finish than on other SPUs.

Test programs For the purpose of testing the implementation, we provide four NestStep-Cell test programs: *Pi calculation* is a calculation-dominated program approximating π by numerical integration, where the sum loop over 10 million subintervals is parallelized. The *dot product* program calculates the dot product of two large (16M float elements) block-distributed arrays that do not fit in the local store. The *parallel prefix* program computes the prefix sums of a large block-distributed shared array of 8M integer elements and stores them in another block-distributed shared array. Finally, the *Jacobi* program contains a loop iterating over one-dimensional Jacobi relaxation steps with a 5-point stencil (smoothening a sampled signal by a low pass filter) on a one-dimensional block-distributed shared array of 7M float elements, until an accuracy criterion is met.

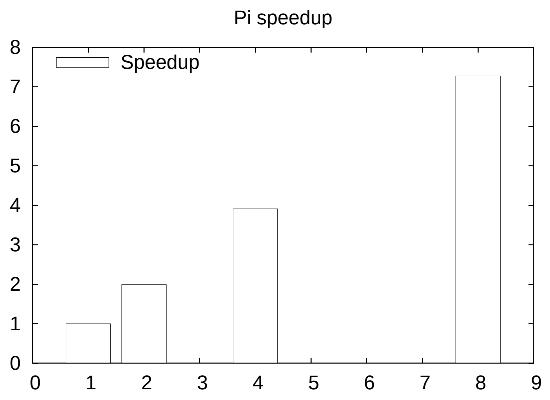
Results Fig. 2 and Table 1 show relative speedups and times required for the different phases of the test programs. We note that all test programs that have to shuffle major amounts of data between memory and SPEs (i.e., all but Pi calculation), the time spent in `DMA_wait` approximately decreases by half as we double the number of SPEs. This is because, for each doubling of the SPEs, each SPE only works on half its amount of data. We also see that the SPEs spend more time in the combine phase, because we need more time to handle combine requests, putting more (sequential) work on the PPE, as we add more SPE threads.

The pi calculation program scales very well; the relative speedup is at 7.3 for 8 SPUs and almost all of the time is spent inside the main calculation loop. This is to be expected since the program does not need to transfer much data on the bus.

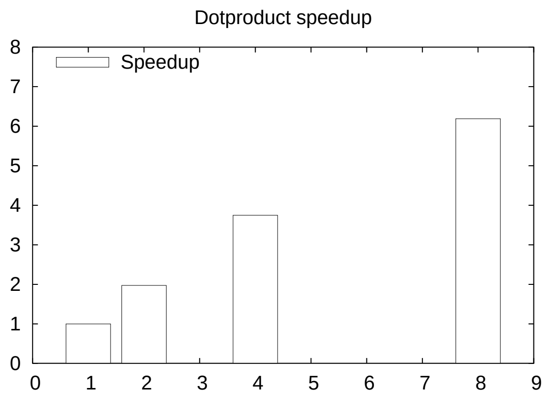
There is a lot of DMA traffic in the dot product program, but we still obtain a relative speedup of 6.2 when using 8 SPUs. This program also spends some time in combining, as can be seen in the time distribution table (b). The size of the arrays is $16 \cdot 1024^2$.

Parallel prefix calculation is quite similar to dot product with respect to memory access patterns, but it uses the runtime system more as prefix combining needs some more work by the PPE. We can see that the parallel prefix program achieves a speedup of 5.9 on 8 SPEs, which is almost the same as for dot product. The array size for this test is $8 \cdot 1024^2$.

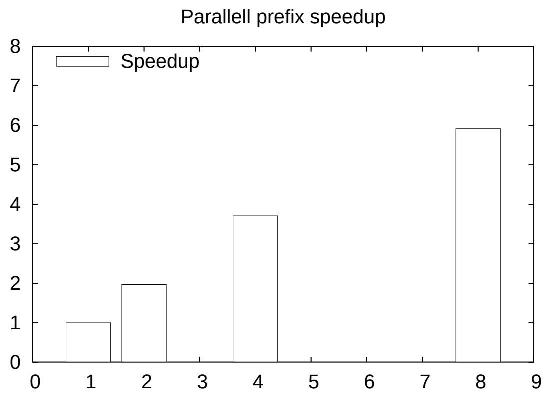
Jacobi calculations are more mixed in their memory accesses than the previous programs; it also spends more time in the runtime system. We still get a speedup of 6.1, and, as can be



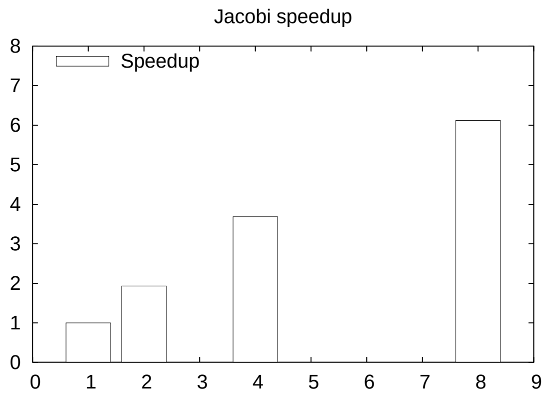
(a)



(b)



(c)



(d)

Fig. 2. Relative speedups of our test programs.

SPUs	1	2	4	8
Calculation	0.408	0.204	0.102	0.051
DMA Wait	0.000	0.000	0.000	0.000
Combine	0.000	0.001	0.002	0.005

(a) Pi calculation.

SPUs	1	2	4	8
Calculation	4.521	2.260	1.130	0.565
DMA Wait	0.034	0.016	0.008	0.004
Combine	0.012	0.034	0.080	0.168

(b) Dot product calculation.

SPUs	1	2	4	8
Calculation	5.948	2.974	1.487	0.744
DMA Wait	0.045	0.023	0.012	0.007
Combine	0.024	0.059	0.123	0.266

(c) Parallel prefix calculation

SPUs	1	2	4	8
Calculation	4.446	2.223	1.111	0.556
DMA Wait	0.197	0.112	0.058	0.029
Combine	0.005	0.068	0.091	0.174

(d) Jacobi calculation

Table 1. Times spent in Calculation, DMA and Combine phases.

seen in the time distribution table (d), the runtime system still takes quite moderate time compared to computations. The array size for this test is $7 \cdot 1024^2$.

As we can see from the time distribution diagrams, the NestStep-Cell run-time system library takes up some fraction of the program time. This can however be decreased if we were to use SIMD instructions during combines as well as use two PPE threads to parallelize the combining mechanism. The library's most important metric for Cell is however the size of `libNestStep_spuliba.a`, the part of the library that gets linked into all SPE binaries. This part takes up 53 KB of the SPE local store. This leaves 203 KB for the NestStep program and its buffered data. It is possible to further decrease the library's memory footprint by removing the debug structures and thus getting rid of the `stdio` library. This would however increase the difficulty of writing NestStep-Cell programs since one would no longer get any warnings when trying to read data that will cause the processor to stall, which can happen for several reasons. We therefore offer two versions of the library, a debug version and a small version.

5 Related Work

Knight *et al.* [9] chose another way to handle programming for architectures with different explicitly managed memory hierarchies. They present a compiler for *Sequoia* [2], a programming language for memory hierarchy aware parallel programs. In their compiler they treat memory as a tree of different memories with coupled processors and try to optimize the locality of code and data transfers. By optimizations to decrease the amount of copying and proper parallelization of code, they manage to get within a few percent of hand-tuned code or even outperform the alternatives.

Ohara *et al.* [10] propose a new programming model for Cell based on MPI, called *MPI microtask*. This programming model allows to partition the code into fragments where each fragment is small enough to fit into the local memory of the SPE. This abstracts the local store away from the programmer and makes it easier to program for the Cell processor.

IBM has also implemented OpenMP on Cell, see Eichenberger *et al.* [1], in the IBM xlc compiler. Code that ought to be run in parallel on both the PPE and the SPEs is compiled for each processor type. They implement a software cache on the SPEs, optimized by SIMD instructions and compile-time analysis, and set up the environment so that the PPE and the SPEs can work together as a gang.

6 Conclusions and Future Work

The goal of this research was to see how well the NestStep programming model suits the Cell processor. We have shown that it is possible to get good performance on our test programs using this programming model. We have developed a stable run-time system library that supports the basic NestStep functionality. The library indeed makes it easier to write SPMD programs for Cell. Our test programs scale very well. At least for the test programs, the degraded performance due to general purpose computing on the SPEs did not matter too much, as was shown in the distribution diagrams in the evaluation when compared against the core calculations. This might however not be a valid assumption for all kinds of programs. Even if it is easier to write this kind of programs with our library, it is not an easy task. Explicit multi-buffering makes an algorithm as simple as the dot product climb to over 200 lines of code, and the constraints on data transfer sizes and offsets of data transfers increases the burden for the programmer. However, if these problems can be overcome, then using this library on Cell is indeed a viable way to get programs that perform well.

There are a few issues for future work that could increase the usefulness of the library. A Cell-aware NestStep front end would allow the programmer to write programs with a less cumbersome syntax, as shown in Section 2. The nesting of supersteps defined in NestStep [7] is not implemented yet for Cell. Using several PPE threads for parallelizing the combining work, as well as using SIMD instructions in combining, could speed up that part of the program by a factor of up to 8 for single precision floating point variables; this would make the implementation more suited for Cell configurations with more than eight SPEs. In a current project we are working towards a library for numerical and non-numerical computations and parallel skeletons linkable from NestStep-Cell, which will encapsulate all double buffering and SIMDization, enabling concise, hardware-independent NestStep programming as illustrated in Section 2 even for Cell. Another possibility for future work is to extend the implementation to hybrid parallel systems containing more than a single Cell.

Acknowledgments. This research was funded by Ceniit 01.06 at Linköpings universitet; by Vetenskapsrådet (VR); by SSF RISE; by Vinnova SafeModSim; and by the CUGS graduate school. — We also thank Inge Gutheil and Willi Homberg from Forschungszentrum Jülich for giving us access to their Cell cluster Juice.

References

1. A. Eichenberger *et al.*: Using advanced compiler technology to exploit the performance of the Cell Broadband Engine^{TM} architecture. *IBM Systems Journal* **45**(1):59–84, 2006.
2. K. Fatahalian, T. J. Knight, M. Houston, M. Erez, D. Reiter Horn, L. Leem, J. Y. Park, M. Ren, A. Aiken, W. J. Dally, P. Hanrahan. Sequoia: Programming the memory hierarchy. In *Proc. ACM/IEEE Conf. on Supercomputing*, 2006.
3. IBM Corp.: Cell BE development environment. www.alphaworks.ibm.com/tech/cellsw/download (2006)
4. IBM Corp.: Cell Broadband Engine^{TM} processor-based systems White Paper. IBM, www.ibm.com, Sep. 2006
5. IBM Corp.: IBM to Build World's First Cell Broadband Engine Based Supercomputer. Press release, www-03.ibm.com/ press/us/en/ pressrelease/20210.wss, 6 sep. 2006
6. D. Johansson: Porting the NestStep run-time system to the IBM CELL processor. Master thesis, Linköping university, Dept. of computer and information science, to appear (2007).
7. C. W. Keßler. *NestStep*: Nested Parallelism and Virtual Shared Memory for the BSP Model. *The Journal of Supercomputing*, 17(3):245–262, Nov. 2000.
8. C. W. Kessler: Managing Distributed Shared Arrays in a Bulk-Synchronous Parallel Environment. *Concurrency and Computation: Practice and Experience* **16**:133-153, Wiley, 2004.
9. T. J. Knight, J. Y. Park, M. Ren, M. Houston, M. Erez, K. Fatahalian, A. Aiken, W. J. Dally, Pat Hanrahan. Compilation for explicitly managed memory hierarchies. In *Proc. 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'07)*, pp. 226–236, New York, NY, USA, 2007.
10. M. Ohara, H. Inoue, Y. Sohda, H. Komatsu, and T. Nakatani. MPI microtask for programming the CELL Broadband Engine processor. *IBM Syst. J.*, 45(1):85–102, 2006.
11. J. Sohl: *A scalable run-time system for NestStep on cluster supercomputers.* Master thesis, LITH-IDA-EX-06/011-SE, Linköping university, Sweden, 2006.
12. L. G. Valiant: A Bridging Model for Parallel Computation. *Communications of the ACM* **33**(8), 1990.