CG-Cell: An NPB Benchmark Implementation on Cell Broadband Engine

Dong Li, Song Huang, and Kirk Cameron

Department of Computer Science
Virginia Tech
{lid,huangs,Cameron}@cs.vt.edu

Abstract. The NAS Conjugate Gradient (CG) benchmark is an important scientific kernel used to evaluate machine performance and compare characteristics of different programming models. CG represents a computation and communication paradigm for sparse linear algebra, which is common in scientific fields. In this paper, we present the porting, performance optimization and evaluation of CG on Cell Broadband Engine (CBE). CBE, a heterogeneous multi-core processor with SIMD accelerators, is gaining attention and being deployed on supercomputers and high-end server architectures. We take advantages of CBE's particular architecture to optimize the performance of CG. We also quantify these optimizations and assess their impact. In addition, by exploring distributed nature of CBE, we present trade-off between parallelization and serialization, and Cell-specific data scheduling in its memory hierarchy. Our final result shows that the CG-Cell can achieve more than 4 times speedup over the performance of single comparable PowerPC Processor.

1 Introduction

The NAS Conjugate Gradient (CG) benchmark is often used to evaluate computer machine performance and compare characteristics of different programming models. It uses a conjugate gradient method to compute an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix. CG is one of the memory intensive benchmark in NAS kernels and is typical of unstructured grid computations, which tests irregular long distance communication by employing unstructured matrix vector multiplication [1].

The recent developments in semiconductor technology lead to the debuts of multi-core processors in the computing industry, such as IBM's Cell, Sun Microsystems' Niagara and AMD's Opteron. The multi-core helps multi-programmed workloads which could contain a mix of independent sequential tasks. It presents a chance to study new or existing parallel programming models. However many questions for programming on multi-core are still open, such as how to efficiently handle specific communication and computation patterns. Cell Broadband Engine (CBE), developed jointly by Sony, Toshiba and IBM, is a new heterogeneous multi-core platform. The Cell is a general-purpose microprocessor which offers a rich palette of thread-level and data-level parallelization options to the programmer.

S. Rao et al. (Eds.): ICDCN 2008, LNCS 4904, pp. 263–273, 2008.

The NPB CG presents a communication and computing pattern seen in sparse linear algebra [11]. It would be helpful to see how CG can be implemented on Cell, the new distributed and parallel scenario. Implementing CG on this special multi-core is a challenging topic. Firstly the essences of heterogeneous cores require careful consideration of task schedules while improving execution efficiency. Secondly Cell architecture shows an explicit and special memory hierarchy to users. On one hand, it presents a shared/global view of data to its nine cores through main memory. On the other hand, it presents a distributed view of data since eight of its nine cores have local memory. To achieve good performance on Cell, people need to carefully handle the data distribution and locality of references. Thirdly, due to its unusual architecture, unconventional Cell-specific code optimization approaches should be considered.

In this paper, we present how we solve the above problems in the implement CG on a real Cell multi-core. The main contributions of this paper include:

- We port NAS CG onto CBE. We present an example of how the communication and computation pattern of CG could be implemented on Cell and how we take advantage of the Cell's distributed multi-core nature. The result shows that CBE as a new architecture has good potential for this programming pattern with limited working data sets.
- We quantify Cell-specific code optimizations and assess their impacts using CG. These quantified results are beneficial for application developments on the Cell.
- We explore the parallelization methods on the Cell. We find that sometimes merely exposing task level parallelism is insufficient for high performance computing. We also find that parallelization on the Cell does not always mean performance speedup. Other factors, like overhead for creating threads and Direct Memory Access (DMA) communication, should be considerable when making decisions on parallelization.

The rest of this paper is organized as follows. Section 2 summarizes related works on programming support for Cell and studies of implementing CG using different programming models. Section 3 introduces kernel CG algorithm and section 4 outlines the Cell architecture. Section 5 presents step by step our CG porting and optimization process. In Section 6, we present the performance of parallel CG on Cell. In the end, we conclude the paper in Section 7.

2 Related Work

As a brand-new multi-core architecture, Cell has attracted many attentions in various research communities. Some researches focus on how to develop applications and speedup their performances. These include exploring new programming models and developing compiler supports. Other researches focus on analyzing the performance of the processor in terms of chip architecture.

Pieter et. al. [4] presents a simple and flexible programming model for Cell. It requires the input application source code to follow a certain paradigm. Then based on the paradigm annotation, a source to source compiler builds a task dependency graph of

functions calls and schedules these calls in the SPEs. A locality-aware scheduling algorithm was also implemented to reduce the amount of data that is transferred to and from the SPEs. Eichenberger et. al [5] present several compiler techniques targeting automatic generation of highly optimized Cell code. The techniques include compiler-assisted memory alignment, branch prediction, SIMD parallelization, and OpenMP task level parallelization. They present the user with a single shared memory image through compiler mediated partitioning of code and data and the automatic orchestration of data movement implied by the partitioning. However, we didn't use their compilers in this paper, because some optimizations, such as the communication optimization and the parallelism decision, may be hard to derive automatically in a compiler due to application complexities. Filip et al. [6] [7] designs a multigrain parallelism scheduler to automatically exploit the task level and loop level parallel in response to workload characteristics. The scheduler oversubscribes the PowerPC Processor Element (PPE) to strive for higher utilization of Synergistic Processor Elements (SPE). In addition, he explored the conditions under which loop-level parallelism within off-loaded code can be used. He ported a bio-informatics code (RAxML) with inherent multigrain parallelism as a case study.

Williams et al. [9] present an analytical framework to predict performance of program code written for Cell. They apply it to several key scientific computing kernels, including dense matrix multiply, sparse matrix vector multiply, stencil computation and 1D/2D FFTs. Driven by their observation, they propose modest micro-architectural modification to increase the efficiency of double-precision calculations. Kistler et. al [8] analyze the Cell processor's communication network, using a series of benchmarks involving DMA traffic patterns and synchronization protocols.

CG benchmark has attracted considerable attentions from high-performance computing community, due to its random communication in computation. Zhang et. al [12] parallelizes the CG using the global arrays shared memory programming model. Mills et. al [13] satisfies the CG memory access by introducing a remote memory access capability based on MPI communication of cached memory blocks between compute nodes and designated memory servers.

3 Kernel CG Description

The Conjugate Gradient Method (CG) is the most prominent iterative method for solving sparse systems of linear equations. The NAS CG benchmark uses the inverse power method to find an estimation of the largest eigenvalue of a symmetric positive definite sparse matrix with a random pattern of non-zeros. The inverse power method involves solving a linear system of equation $Az = x$ using the conjugate gradient method. The size of the system *n*, number of outer iteration as "niter" in Figure 1, and the shift λ for different problem sizes in the benchmark are clearly specified in its official document [2]. Figure 1 illustrates the main iteration in NAS CG benchmark.

```
Initialize random number generator
Use Makea() to generate sparse matrix
x = [1,1,...,1]^T
DO it =1, niter   # For CLASS A, niter=15
     Running the function conj_grad to solve the
     system Az = x and return || r ||,
     ζ = λ + 1/(x^T z)
     x = z / || z ||
ENDDO
```

Fig. 1. The main iteration in CG benchmark

```
z = 0
r = x
ρ = r^T r
p = r
do i = 1, 25
  q = Ap                                  Step 1
  α = ρ/(p^T q)                           Step 2
  z = z + αp                              Step 3
  ρ0 = ρ
  r = r − αq
  ρ = r^T r                               Step 4
  β = ρ/ρ0
  p = r + βp                              Step 5
enddo
compute residual norm explicitly : || r ||=|| x − Az ||   Step 6
                                                          Step 7
```

Fig. 2. Conjugate Gradient Method

The solution z to the linear system of equations $Az = x$ is to be approximated using the conjugate gradient (CG) method, which is implemented as in Figure2. In this paper, we use NPB2.3 OpenMP version as our start point.

4 Cell Broadband Engine

CBE is the first incarnation of a new family of microprocessors extending the 64-bit PowerPC architecture. It is a single-chip multiprocessor with nine processors operating on a shared, coherent memory. These nine processors are specified as one *PowerPC Processor Element (PPE)*, and eight *Synergistic Processor Elements (SPE)*. The PPE is a 64-bit, dual-thread PowerPC processor, while the SPE is the high-end computing engines optimized for running compute-intensive applications. The designation

synergistic for the SPE was chosen carefully because there is a mutual dependence between the PPE and the SPEs. The SPEs depend on the PPE to run the operating system, and, in many cases, the top-level control thread of an application; the PPE depends on the SPEs to provide the bulk of the application performance.

The PPE supports both the PowerPC instruction set and the Vector/SIMD multimedia extension instruction set [10]. In our current Play Station 3 (PS3) Cell, the PPE doesn't support the vectorization of double precision float point data (the data type of matrix element in CG).

Each SPE contains a RISC core (SPU), 256-KB, software-controlled Local Store (LS) for instructions and data, and a large (128-bit, 128-entry) unified register file. The SPEs support a special SIMD instruction set which could lead to computation vectorization. SPEs rely on asynchronous DMA transfers to move data and instructions between main storage and their LS. Data transferred between local storage and main memory must be 128-bit aligned and the size of each DMA transfer can be at most 16KB. The Memory Flow Controller (MFC), which handles DMA transfer, supports only DMA transfer sizes that are 1, 2, 4, 8 or multiples of 16 bytes long. Note that the LS in SPE has no memory protection, and memory access wraps from the end of LS back to the beginning. An SPU program is free to write anywhere in LS including its own instruction space. We need to avoid the corruption of the SPU program text when the stack area overflows into the program area.

The PPE and SPEs communicate coherently with each other and with main memory and I/O through the Element Interconnect Bus (EIB). The EIB is a 4-ring structure for data, and a tree structure for commands. The EIB's internal bandwidth is 96 bytes per cycle, thus achieving a peak bandwidth of 204.8GB/s. It can support more than 100 outstanding DMA memory requests between main storage and the SPEs.

5 Design and Analysis

In our implementation we ported CG to Cell in four steps: (i) porting the CG on the PPE; (ii) offloading the most time-consuming parts on one SPE; (iii) parallelizing the SPE code to run on multiple SPEs; (iv) optimizing the SPE code. These steps are outlined in the following sections.

The results reported in this section are obtained from a Cell multi-processor in a PS3. It has 256MB XDR RAM. The PPE have a 32KB L1 instruction cache, a 32 KB L1 data cache, and a 512KB unified L2 cache. The system runs Fedora Core 5 (Linux kernel 2.6.16), including Cell-Specific kernel patches. We compile our code using ppuxlc and spu-gcc in the Cell SDK 2.1 for ppu and spu respectively.

5.1 Porting CG on PPE

As our first step, we port CG onto PPE, which is actually a single Power 970 architecture compliant core. To port CG on PPE, we collect all functions distributed in several source files and move them into one source file. In original OpenMP implementation, the header files used to be automatically generated by the NPB script. We manually write the header files according to our input CLASS. These works

simplify both the program directory hierarchy and its build process. We call the current work the version 0.1.

In this version, we run the CG on one PPE thread with an input of CLASS A. The system matrix size for CLASS A is 14,000×14,000. It has 15 iteration, i.e. 15 conj_grad function calls as in Figure 1, which has total 1.50×10^9 FLOPs., The time for the 15 conj_grad function calls running on PPE is 14.90 seconds. We treat this result as our performance baseline for later sections.

5.2 Function Off-Loading

Our next step is to offload the computation intensive function conj_grad into SPE. We create one SPE thread to be in charge of the total 15 conj_grad function calls. To observe the initial speedup, it is better to use a small input matrix, which can be completely loaded into SPE 256KB LS. We shrink the input matrix size to 256×256, which occupies 30KB memory space. This mini-matrix allows us not to involve into the communication complexity of dividing data sets and focus only on the computation optimization. We will talk about dividing data sets later in Section 5.3.

Our initial computation optimization is to vectorize all the matrix data. Vector operations eliminate the overhead for scalar formatting and thus reduce the long latency caused by scalar load and store. The Vector/SIMD Multimedia Extension intrinsic and SPE SIMD operation provide supports to vector operation for PPE and SPE respectively. We could depend on auto-vectorizing compiler to do the vectorization by merging scalar data into a parallel-packed SIMD data structure. However such compiler must handle all the high-level language constructs and do not always produce optimal code. Therefore, we choose to do vectorization manually in our program. In addition, we only vectorize the function conj_grad running on the SPE, because PPE doesn't support SIMD operation for double precision floating point matrix data in CG.

Generally speaking, there are two methods for organizing data in SIMD vectors: Array-of-Structure (AOS) and Structure-of-Array (SOA) [3]. In this paper, we use SOA for vectorizing matrix data, i.e. across the vector both the data types and data interpretation are the same. This conforms to most data operations requirements in the algorithm and execute more efficiently than AOS organization. Although AOS produces small code sizes, it requires significant loop-unrolling to improve its efficiency and executes poorly. Note that here we don't vectorize step1 and step6 shown in Figure 2. The reason lies that these two steps do sparse matrix-vector multiplication. The data for computation in vector p and z are scattered around in the memory space (Figure 3). To vectorize the computing, we need to gather vector data into address-aligned continuous memory space. This requires extra memory copy operations which are proved to have big overhead by our experiences.

Based on the above work, our result shows that the new CG takes 35 milliseconds with mini-array as input, while the version 0.1 with the same input takes 8.1 milliseconds. This new CG's performance is even worse. Obviously the optimization with only data vectorization is not enough for performance improvement. This is because the vectorization can only happen on the SPE in our case and thus has limited benefits. Meanwhile for function off-loading we have to pay for the costs of transferring data between the main memory and SPE's LS, which counteracts the gains of vectorization.

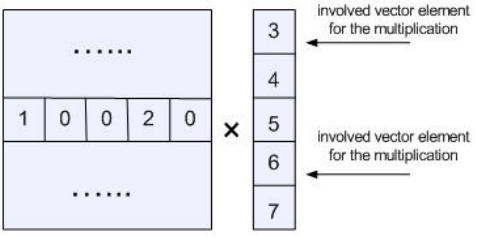

Fig. 3. An example of spare matrix-vector multiplication. One matrix row is doing multiplication with the vector. Only non-zero matrix data are considered. The involved vector elements for multiplication are not in continuous memory address.

5.3 Parallel Execution across Multiple SPEs

To fully take advantage of CBE multi-core architecture, we want to parallelize the program on 8 SPEs. An obvious method is to regard each conj_grad function call as a task and distribute 15 tasks across 8 SPEs. However due to data dependency on vector *x*, this task-level parallelization doesn't work. Due to the same reason (the data dependency on vector *p* in Figure 2), distribution of 25 iterations in the conj_grad function across 8 SPE doesn't work either.

As an alternative, we deliberately divide each loop into 5 steps and parallel them as shown in Figure 2. We also parallel two "residual norm" computing steps outside of conj_grad loops, as step 6 and step 7 in Figure 2. We create seven SPE modules corresponding to these steps and move the conj_grad loop into the PPE. Whenever the PPE program flow meets a step, it will load SPE program image corresponding to this step into each SPE LS and create threads to execute it. Then each SPE will fetch part of total matrix data from the main memory to the LS, do computing and/or update data in the main memory.

CG uses three arrays ("*rowstr*", "*colidx*" and "*a*") to record the sparse matrix in a compact way. The array *a* stores non-zero data elements of the matrix and put them in a row. The array *rowstr* records the position of first non-zero element in each matrix row. The array *colidx* records the column number for each non-zero data elements in the matrix. For CLASS A, the total size of these three array data is 4.41MB, which cannot be fitted into the LS (256KB) of a SPE.

The way we distribute the data is described in the following. The conj_grad function does computation row by row, so we distribute the data among SPEs at the granularity of row. In particular, each SPE processes 1/8 of the total rows and needs data for its 1/8 share of the total rows. For the array *rowstr*, we just evenly divide it into eight parts and copy them from the main memory to SPE LS. For the array *colidx* and *a*, even distribution doesn't work, because the number of nonzero elements varies from one row to the others. We need to locate the starting and the ending position for each 1/8 row block. This could be done with the help of *rowstr* array. The *rowstr* array data for 1/8 rows with CLASS A as input is 6.8KB and can be totally put in the LS.

However even though we distribute the array data in the above way, the array data for 1/8 row could still be bigger than the 256KB LS size. Therefore we strip-mine each 1/8 rows data by fetching a few row elements to local storage, and execute the corresponding loop iterations on this batch of elements. This operation continues until

all data are processed. We use an 8KB buffer for fetching array data. It should be noted that to avoid using up LS memory, the spaces used for buffer is much smaller than the size of the LS.

Both the step 1 ($q = Ap$) and step 6 (Az) do multiplication of matrix A with a vector. To reduce the unnecessary multiplication, the original CG only computes multiplications of the nonzero elements of A with the corresponding vector elements. To locate the needed vector elements for the multiplications, the CG depends on the array *colidx*. Although this simplification reduces the computing, we may have to copy the *colidx* and the vector from the main memory to the LS, as far as the parallel implementation is concerned. Since the *colidx* is too big to be totally placed in the LS, we could process it in the same way as we do to the array *a*. However for the vector *p* or *z*, the needed elements for the multiplication are scattered around the memory. It is highly inefficient to copy these elements one by one from the main memory to LS.

There are two ways to solve this problem. One is to ignore the unnecessary vector elements and firstly copy the needed vector elements into an address-aligned new array in the main memory. We call this array as "compressed vector". Then only the compressed vector is copied to the SPE and the SPE doesn't care about *colidx* at all. Although this method sounds promising, it has big overhead of copying. Since the vector *p* is updated each time at the step 5, we need to re-produce the compressed vector at each loop. According to our experiences this overhead easily beats the gains of parallelization. The second way is to completely put the vector *p* in the SPE LS and process the *colidx* in the same way as we do to the array *a*. This method needs to reserve enough LS space for the vector. In our implementation we manage to limit the data (three arrays) in the LS within 30KB (The section 5.4.2 describes how we use the double buffer to limit the size of array *a* and array *colidx* in the LS. The array *rowstr* is 6.8KB and totally placed in the LS. This sums up to 30KB) and the program image is less than 8KB, so the rest LS space can hold the whole vector *p* under the case of input CLASS A. For larger CLASS, there is a chance the vector can't be totally placed in the LS. We have to divide it into data blocks. The size of data blocks is determined by the available LS space. We leave this case as our future work.

After the parallelization in this step, we got the current version 0.2. This version takes 21.62 seconds with CLASS A as input. This result is even worse than our baseline (14.90 seconds). The next section will describe how we reduce the execution time by optimization.

5.4 Performance Optimization

We consider performance optimization from two aspects. One is to balance the tradeoff between parallelization and serialization based on the profiling results. The other is to take into consideration Cell architecture characters.

5.4.1 Parallel or Not

To figure out the reason why version 0.2(parallel version) does not gain performance speedup over version 0.1(serial version), we profile the conj_grad for both versions at the granularity of step. To make things simpler, we only run one loop for the conj_grad main iteration. The profiling results are shown in Figure 4. It is clear that step1 and step 6 gain great performance improvements from the parallelization, while other steps do

the opposite. Further analysis reveals that step1 and step6 have $O(n^2)$ multiplication while other steps have $O(n)$, where n is the matrix size. This means that step1 and step 6 take up a predominant percentage of computation if n is large. Larger computation intensiveness means more computing on each data movement (DMA transmission) between SPE LS and main memory. Only after the parallelization gains counteract the overhead of data movement could we get the performance improved. Thus, the overhead of data movement for parallelization should be considered. In addition, creating SPE threads is not free. Our experiment shows that the overhead of creating a single SPU thread costs 1.81 milliseconds, which is comparable to running 10 times of step 2 in one loop. Furthermore, more threads also mean more scheduling tasks and more competition in DMA bus. Due to those various factors, parallelization is not always beneficial to performance. Therefore, we have to carefully consider the balance between parallelization and serialization.

Based on the analysis above, we don't parallel steps 2-5 and 7. Results show (Figure 4 and Figure 6) that this optimization (we call it the version 0.3 hereafter) improves a factor of 6.093 over version 0.2, and a factor of 4.519 over the baseline (version 0.1).

5.4.2 Branch Reducing and Communication Optimization

The Cell SPE hardware assumes linear structure flow and produces no stall penalties from sequential instruction execution. A branch instruction has the potential of disrupting the assumed sequential flow. Specifically, a mispredicted branch incurs a penalty of approximately 18-19 cycles in the SPE. Branches also create scheduling barriers, reducing the opportunity of dual issue and covering up dependency stalls. Our next optimization is to reduce branches in SPE as much as possible. In step1 and step6, we need to use a function called getNextBlockSize, which deals with address alignment for DMA transfer and controls transfer block size within the limits of available SPE LS. This function employs several conditional branches. Moreover, considering the popularity of this function, these branches would be rather expensive for SPE. Therefore we move this function to PPE and use DMA-List to determine each transfer size before SPE takes over computation tasks. Each SPE gets its DMA-list after it starts new thread and uses the list to determine each transfer size.

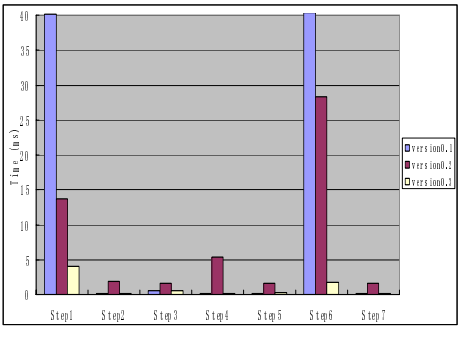

Fig. 4. conj_grad function profiling. The time for each step is for one loop in the function.

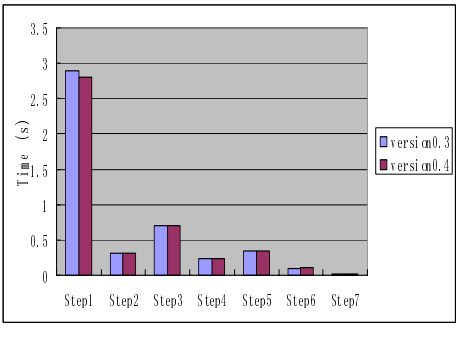

Fig. 5. Performance profiling of 15 times of the conj_grad function call

Another optimization we employ is to use double buffering in SPE. We allocate two buffers for array *a* and *colidx* respectively and each buffer holds 1024 array elements. When the SPE is working on one buffer, the other buffer is for data transferring. In this way, we can pipeline computation time with data transferring time. The total buffer size is 24KB, which is much smaller than the size of LS. This double buffering scheme maximize the time spent in the compute phase of a program and minimize the time spent waiting for DMA transfer to complete.

With the above two optimizations, our performance improves more than four times over the baseline. We call the current CG-Cell after optimization version 0.4, which is also our final version. Figure 5 depicts the profiling of the paralleled conj_grad at the granularity of step for version 0.3 and version 0.4.

6 Performance Evaluation

As a performance comparison, Figure 6 evaluates the total execution time for several important versions in CG_Cell. It shows that our version 0.4 has been improved a lot over version 0.1. Version 0.2, although fully paralleled, costs the most time. Since we eliminate the parallelization in the step 2 to step 5, version 0.3 is improved greatly. In version 0.4, since we have used the specific optimization for the Cell, the total execution time is less than 1/4.

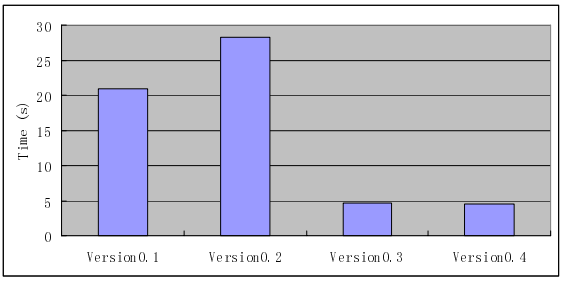

Fig. 6. Total running time of four versions

7 Conclusion

This paper presents the porting, optimization and evaluation of CG, an NPB kernel benchmark on the Cell Broadband Engine. We explore many Cell-specific optimizations and the performance implications of these optimizations. We explore the tradeoff between the parallelization and the serialization for the CG implementation. By offloading the most time-consuming function onto the SPE, we implement a parallel version of CG with less overhead. We also carefully divide and organize the data sets to fit into Cell memory hierarchy so that the data transfer is dropped as much as possible. We vectorize the computation, reduce the branches instructions on SPE program and use double buffer to hide memory access delay. Starting from a less

optimized CG implementation on PPE, we were able to boost performance on Cell by more than a factor of four.

References

[1] Bailey, D., Barszcz, E., et al.: The NAS Parallel Benchmarks—Summary and Preliminary Results. In: Proceedings of the 1991 ACM/IEEE Conference on SuperComputing (1991)

[2] Bailey, D., Harris, T., Saphir, W., Van der Vijingaart, R., Woo, A., Yarrow, M.: The NAS Parallel Benchmarks 2.0, Technical Report NAS-95-020, NASA Ames Research Center (1995)

[3] IBM Systems and Technology Group. Cell Broadband Engine Programming Tutorial Version 2.1, http://www.ibm.com/developerworks/power/cell/documents.html

[4] Bellens, P., Perez, J.M., Badia, R.M., Labarta, J.: Memory—CellSs: a programming model for the cell BE architecture. In: Proceedings of the 2006 ACM/IEEE conference on Supercomputing, Tampa, Florida (November 11-17, 2006)

[5] Eichenberger, A.E., et al.: Optimizing Compiler for a Cell processor. In: 14th International Conference on Parallel Architectures and Compilation Techniques, St. Louis, MO (September 2005)

[6] Blagojevic, F., Stamatakis, A., Antonopoulos, C., Nikolopoulos, D.S.: RAxML-CELL: Parallel Phylogenetic Tree Construction on the Cell Broadband Engine. In: Proceedings of the 21st IEEE/ACM International Parallel and Distributed Processing Symposium, IEEE Computer Society Press, Los Alamitos (2007)

[7] Blagojevic, F., Stamatakis, A., Antonopoulos, C., Nikolopoulos, D.S.: Dynamic Mulitgrain Parallelization on the Cell Broadband Engine. In: Proceedings of the 2007 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Jose, California, pp. 90–100 (March 2007)

[8] Kistler, M., Perrone, M., Petrini, F.: Cell Multi-processor Interconnection Network: Built for Speed. IEEE Micro, 26(3), (May-June 2006), http://hpc.pnl.gov/people/fabrizio/papers/ieeemicro0cell.pdf

[9] Williams, S., Shalf, J., Oliker, L., Kamil, S., Husbands, P., Yelick, K.: The potential of the Cell Processor for Scientific Computing. In: ACM International Conference on Computing Frontiers (May 3-6, 2006)

[10] PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technology Programming Environments Manual, http://www-306.ibm.com/chips/techlib

[11] Asanovic, K., Bodik, R., et al.: The Landscape of Parallel Computing Research: A View from Berkeley, http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.htm

[12] Zhang, Y., Tiparaju, V., Nieplocha, J., Hariri, S.: Parallelization of the NAS Conjugate Gradient Benchmark Using the Global Arrays Shared Memory Programming Model. In: IPDPS 2005. Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium - Workshop 4 - Volume 05 (2005)

[13] Mills, R., Yue, C., Stathopoulos, A., Nikolopoulos, D.S.: Runtime and Programming Support for Memory Adaptation in Scientific Applications via Local Disk and Remote Memory.Journal of Grid Computing, 5(2):213-234, ISSN 1570-7873, 1572-9184. Springer Verlag (June 2007)