A Container-Iterator Parallel Programming Model

Gerhard Zumbusch

Friedrich-Schiller-Universität Jena,
Institut für Angewandte Mathematik,
Ernst-Abbe-Platz 2, 07743 Jena, Germany
zumbusch@mathe.uni-jena.de
http://cse.mathe.uni-jena.de

Abstract. There are several parallel programming models available for numerical computations at different levels of expressibility and ease of use. For the development of new domain specific programming models, a splitting into a distributed data container and parallel data iterators is proposed. Data distribution is implemented in application specific libraries. Data iterators are directly analysed and compiled automatically into parallel code. Target architectures of the source-to-source translation include shared (pthreads, Cell SPE), distributed memory (MPI) and hybrid programming styles. A model applications for grid based hierarchical numerical methods and an auto-parallelizing compiler are introduced.

Keywords: parallel programming models, automatic parallelization, domain specific code generation, parallel numerical methods, multigrid, MPI, Posix threads, Cell processor.

1 Introduction

Development of parallel code for numerical computations is still an issue, for several reasons: On the one hand, current computers are parallel computers, with increasing degree of parallelism. On the other hand, parallelism is still very visible in the code: Current parallel programming models tend to be either limited in their applicability, limited in their efficiency or lead to a low level of parallel programming. Standard parallel programming models include thread libraries like POSIX threads [1] and Intel's TBB [2] on top, lightweight threads like in Cilk [3] and Concur (Microsoft) [4], and OpenMP [5] for shared memory computers and two- and single-sided message passing for distributed memory computers like in MPI [6]. A higher level approach is represented by the loop- and array-parallel HPF [7], Co-Array Fortran[8] and related Fortran version constructs to be generalized in projects like Chapel (Cray) [9] and Fortress (Sun) [10]. Distributed shared memory techniques are refined in OpenMP on clusters (Microsoft), and in the current evolution of splitted global address space languages like unified parallel C (UPC) [11] and in the X10 (IBM) project [12].

In addition to efficient general purpose parallel programming models, easier to use domain specific models are important. The simplicity of such a restricted

R. Wyrzykowski et al. (Eds.): PPAM 2007, LNCS 4967, pp. 1130–1139, 2008.
© Springer-Verlag Berlin Heidelberg 2008

computational model may have several advantages: For example, a single source code may compile to efficient sequential and parallel code for different types of parallel architectures, which is advantageous for code development and the lifetime of the code. Higher level program constructs may be automatically analysed and compiled into parallel code, which is not possible for general programming models. Compiler analysis may further assist and verify parallelization of the code, such that common mistakes can be eliminated.

The container-iterator programming model we propose here is supposed for the gap between expressibility and simplicity with numerical applications in mind. Several parallel Fortran extensions focus on parallel vector and array expressions. This is well suited for dense linear algebra and some finite difference methods, but may be less adequate for sparse matrices, non-uniform Finite Element grids, randomly distributed particles and other less structured numerical data. We try to attack this problem by a split of the code into an iterator which handles the non-uniformity of data in a container and a computational kernel applied to each element of the container. This is equivalent to parallel arrays for array-structured data with a slightly different syntax, but more flexible in other cases.

Shared memory programming models like OpenMP are not very well suited for distributed memory environments. The abstraction of a parallel loop with fixed bounds in early OpenMP revisions is a further severe restriction to the application. However, OpenMP is far simpler to use than the underlying thread programming model. We try to provide a programming model, which can be both compiled to shared memory and distributed memory models. In the case of a loop of independent operations, container-iterator is equivalent to OpenMP parallel loops with a slightly different syntax.

Virtual shared memory programming models like UPC, which combine the ease of use of shared memory (of certain coherence) with distributed memory architectures rely heavily on the underlying distributed shared memory engine and its coherence model. In the container-iterator model we try to create explicit message-passing code by a data dependence analysis of the compiler. This is possible due to the restriction to numerical applications and detailed knowledge of the container. However, it is independent of the computational kernel, as long as it really is a parallel algorithm.

Finally, parallel skeletons are based on a approach to parallel computing comparable to the container-iterator model. A collection of skeletons defines different patterns of algorithms like a linear loop or a divide and conquer tree, see e.g. [13]. The user code is derived from a specific skeleton. The parallelization is done by a compiler, which is able to transform each skeleton into parallel code. Some differences to the compiler-iterator model are: We start from a global persistent data decomposition transparent to the code, such that several parts of the algorithm can be applied to the container, one after the other. The user codes derived from an iterator may have different dependencies leading to different communication patterns, each of which can be detected by a compiler dependence analysis. Some communication patterns are currently not available in parallel skeletons. However,

`ForEach(iteration_variables, iterator, code);`: syntax.
`iteration_variables`: C++ declaration of access variables to container elements, e.g. '`int i, int j`' or '`tree *b`'.
`iterator`: instance of an iterator, includes container specification (like bounds or root pointer) and type of iteration.
`code`: regular C++ function body code. The `iteration_variables` are defined in the code and point to the current atom. Certain restrictions apply to the access of other atoms (dependency) and variables of outer scopes (read only or reduction).

Fig. 1. A formal parallel iterator

given appropriate compilation techniques and support libraries, the container iterator approach could be interpreted as a flavour of parallel skeletons.

The compilation technique we propose here uses static analysis. In fact it is incorporated in an experimental C++ compiler based on the current 4.2.1 Gnu gcc release. The remaining parts of the system are some pre- and post-processing facilities using m4 macro processor and perl scripts for code generation and C++ run-time libraries. A previous version of the system [14] was based on a speculative run-time analysis performed with a standard C++ compiler and different scripts. Related projects include the Rose compiler [15] and expression template libraries like Pooma [16].

2 The Container-Iterator Programming Model

In the domain of numerical software, good strategies to parallelize a given sequential implementation are often known. Larger sets of data can often be decomposed and mapped to coupled parallel tasks efficiently, e.g. many codes for the solution of partial differential equations, particle methods and other "local" methods which respect physical space. Some examples and many references can be found in [17]. Further, the sequential algorithm does not need to be changed, but often can be re-arranged according to the data decomposition.

We assume that data is organized in a large container of similar smaller units, called atoms. The sequential algorithms operate on all or parts of the atoms in a similar way and in a given order. Parallelization means to schedule operations on different atoms of an algorithmic step. This is a data parallel approach.

A formal description of the iterator can be found in figure 1. The syntax is similar to the C++ for-loop `for(`iteration variable declaration and initialization; upper bound; increment`){code}`, which is also used in the C++ STL [18] for iterators on arbitrary containers. We prefer to place the body `code` always right next to the loop iterator instead of a separate class definition (like in TBB [2]), which can also be done using the C++ Boost library lambda expressions [19] like in Concur [4].

Note that the container-iterator model does include, but is not limited to a contiguous data vector or an ascending access of totally ordered atoms. For example the model also includes different ways to access leaves of a tree. However, the access order and the parallelism of the operation follow from the data

dependency of the atoms. This can be detected by the compiler and need not be specified in the code. Note that we need more algorithmic patterns than just divide and conquer, see e.g. [20].

We restrict ourselves to numerical algorithms and a data container-iterator programming model. Further, we assume that a good domain specific data decomposition scheme is available. Now, numerical algorithms can be written as a sequence of (parallel) iterators and sequential operations. An iterator consists of a specification of the iteration space and a code fragment for the algorithmic atom. The atom will be executed for each element of the iteration space. The atom can be analysed using standard compilation techniques such as data dependency and flow analysis. The goal is to facilitate automatic parallelization. Compilation targets are shared memory and different distributed (virtual shared) memory systems with their respective low level programming models. hybrid architectures such as clusters of shared memory nodes, hierarchies of tightly and weakly coupled systems can be targeted with a mixture of message passing and multiple threads. Further, specialised architectures like the Cell processor [21] can be targeted, which will be described in more detail.

Note that this programming model also covers cases of loop and array parallelism addressed in OpenMP and some Fortran versions. It does not contain explicit parallel commands and it is not as expressive as UPC and as low level parallel programming models.

3 Sample Containers and Iterators

As an illustrative example for the container-iterator programming style, we show how it looks like for two domain specific extensions of C++, see figures 2 and 3. We have constructed a source-to-source compilation system which translates the domain specific code into standard C++ code with a domain specific library and a standard parallel programming model.

Figure 2 give a brief sketch of the different code stages of the source-to-source translation system. On top is a code sample of the container-iterator programming model. A one-dimensional grid, an iterator over an interval and a vector data structure are constructed. The `ForEach` loop features nearest neighbour access `y(-1)`, `y(+1)` and a global reduction `e`. On the left, the code is transformed into a sequential `for` loop. On the right, a data dependence analysis, based on the extended Gnu g++ compiler lists load, sore and reduction operations. This analysis is transformed into a pthread code (left), MPI code (middle) and a Cell processor code (right). Note that the loop is split into two parts for the pthread code and even into different files (for incompatible PPU and SPU binaries) for the Cell processor. Further, the necessary send and receive (MPI [6]) or get and put (Cell, see [26]) commands are generated using the data dependence analysis.

An application specific library provides data containers like uniform grids implemented as multi-dimensional arrays or a hierarchical decomposition of a set of particles implemented as quad-trees. In each case a geometric domain decomposition works well for parallelization. The applications in mind based on the containers include the solution of partial differential equations

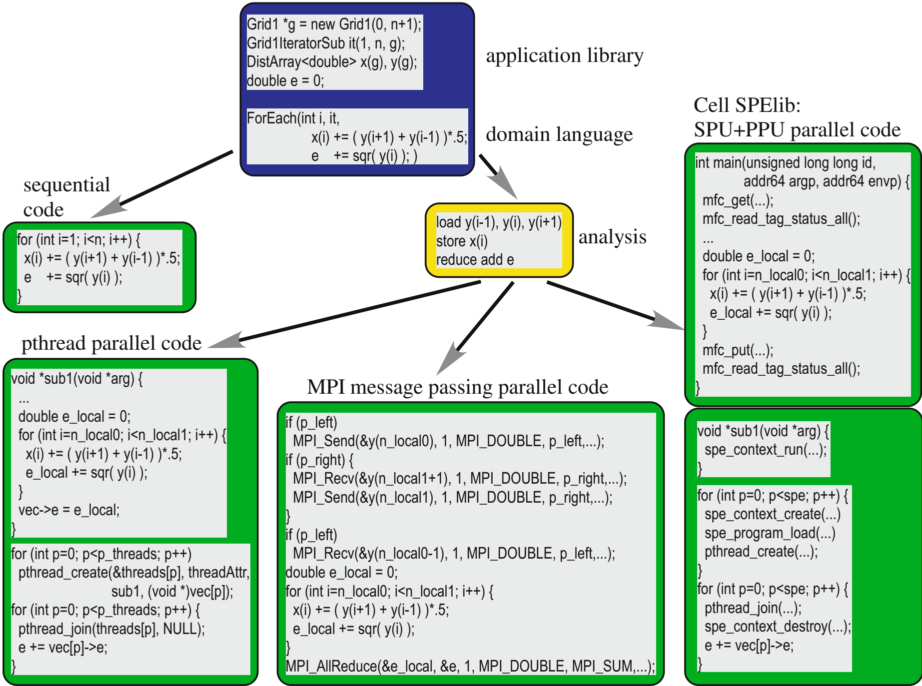

Fig. 2. Source-to-source transformation of the domain language to several target programming models (Posix threads, MPI message passing, Cell processor SPE library). Vector/ one dimensional grid example.

(finite elements, finite differences, multigrid solver, grid refinement), fast summation techniques (fast multipole expansion) and integral equations. Array operations may look like figure 2.

Particle methods, unstructured grids with refinement and fast summation techniques may require a programming style like in figure 3. Note that the operations on the tree nodes are no longer independent in some cases. However, a bottom-up or top-down iteration still offers enough parallelism for an efficient parallelisation. The basic access patterns for a fast multipole summation [22] are given for a binary tree, which easily generalises to quad- or oct-trees.

```
class tree : public KAryTree<class tree, 2>
public: // generic binary tree provides tree* child(int);
   complex<double> m, l, f, x;
tree *root = new tree;
TreeIterator<tree> iter(root);

ForEach(tree *b, iter, b->f = b->l; )

ForEach(tree *b, iter, `
   for (int i=0; i<2; i++)
      if (b->child(i)) b->child(i)->l += b->l; ')
```

```
ForEach(tree *b, iter, `
   for (int i=0; i<2; i++)
      if (b->child(i)) b->m += b->child(i)->m; ')

Require( list<tree*> inter, fetch );
ForEach(tree *b, iter, `
for (list<tree*>::const_iterator i = b->inter.begin();
      i != b->inter.end(); i++)
   b->l += log(abs(b->x - (*i)->x)) * (*i)->m; ')
```

Fig. 3. Binary tree examples: Declaration, embarrassingly parallel, top-down (no communication with replicated root), bottom-up (communication upwards), top-down with neighbourhood communication defined by a relation ‘fetch’

The analysis of the first `ForEach` in figure 3 gives a local read ‘l’ and a local write ‘f’ leading to an embarrassingly parallel loop over all atoms. The second `ForEach` gives a local read ‘l’ and child read and write ‘l’. This does only make sense for a top-down tree traversal. For distributed memory, the variable ‘l’ must be provided in the ghost atoms. The third `ForEach` gives the reverse local read and write ‘m’ and child read ‘m’, which only makes sense for bottom-up traversal. Variable ‘m’ has to be sent in message passing. Finally the last `ForEach` introduces an indirect addressing. Dependence analysis includes local variable ‘l’ and remote variables ‘m’ and ‘x’. The user provided relation ‘fetch’ gives a hierarchical hull of all candidates of the remote atoms. In the fast-multipole and some Finite Element algorithms the relation is based on the geometric distance of atoms.

4 A Sample Compilation System

The sample compilation system translates the extended C++ code using the container-iterator parallel programming model into plain C++ code using standard parallel programming models according to figure 2. The source-to-source translation seems to enable more flexibility than a direct single pass-compiler and can be considered as code generation [23]. Currently it is implemented using a general code dependence analysis tool and container/ iterator specific scripts. The tool uses the front end and some of the optimization stages of the back end of the Gnu gcc compiler (C++ front end, current release version 4.2.1). It dumps data type and dependency information extracted from the internal tree-ssa (static single assignement) language independent code representation. This includes names and types of outer-scope variables which are read, written, or updated in the code. Special mechanisms exist for updates leading to reduction operations.The post-processing for different targets uses a sequence of perl scripts. the code generation itself is performed by the m4 macro processor.

The idea of the parallelizing compiler is to do a certain global data dependence analysis. The user code is analysed for each atom separately, see `code` in figure 1. The `ForEach` loop is expanded into an iterator, which executes the atom for each node of the data container. For the parallel implementation, we control both global data distribution and the parallel iterator. Standard strategies include the use of distributed atoms with an “owner-computes” policy. Communication is be implemented through a small amount of replicated atoms like ghost zones or a common replicated root. Whether replicated data is computed locally, updated by message passing, put into shared memory or discarded in an algorithmic step depends on the parallel iterator and the communication pattern.

The data dependence analysis of each atom is passed to templates of parallel iterators. For each parallel target, be it message passing, threading or a mixed model, and for each possible communication pattern there exists a different parallel implementation of the iterator. One part of the templates can be found in the domain specific library. The remaining parts, including the selection and invocation of the correct template and the variables to be transferred or updated are located in the result of the source-to-source translation. The whole process can be considered as a compilation with a single coarse grain dependence

analysis. However, clever constructions of expression templates or parallel skeletons may lead to comparable results for certain containers, see Pooma [16].

Note that the source-to-source transformation process does create the same number of communication operations than hand coded examples. Hence message passing parallel code on distributed memory computers is expected to perform comparable. The performance of multi-threaded code on shared memory computers heavily depends on the data and memory layout. This is given by the application library, not the code transformation system. Hence, even in this case a competitive performance is expected.

5 Experiments

We consider a hierarchical, three-dimensional numerical grid/array code as a test example with pthread and Cell processor as target computer architecture. Message passing MPI results and results for numerical tree codes of the older (speculative analysis) container-iterator translation system can be found in [25] and [14] respectively.

The NAS multigrid benchmark code Fapin [24] implements a geometric multigrid $V_{0,1}$-cycle with one post-smoothing step for a Poisson equation on a set of nested three-dimensional cartesian grids with constant coefficients. The Fortran77 code was ported C++ using the distributed array classes, single precision floating point numbers and run on other data sets (7 levels, fine grid 129^3) than originally conceived, in order to fit into the main memory (256 Mbytes) of a Sony Playstation3 with a Cell processor.

First we consider the Posix thread (pthread) and the sequential versions of the code. Table 1 shows execution times for different platforms: A four dual-core AMD Opteron processor system at 1.8GHz (Linux, gcc 3.4.6), a two dual-core Intel Xeon 5110 processor system at 1.6GHz (Linux, g++ 3.4.6), a 1GHz six-core quad-issue Sun T1 processor (with limited floating point performance) (Solaris, SunStudio 11 CC) and a dual-core Intel Centrino Duo T2300 system at 1.66GHz (Windows XP, g++ 3.4.4).

The pthread implementation, which imposes slight parallel overhead, on a single processor core runs almost as fast as the sequential version. Increasing the number of cores gives good speedup. Note that computational work on coarser grids is performed on a single core for efficiency reasons, and finer grids are distributed to all cores. However, the exact break even point for the strategy

Table 1. Execution times of the multigrid example, wall clock in sec. Sequential and pthread parallel versions.

no. of threads	1	1	2	3	4	5	6	7	8	12	18	24
6 cores Sun T1	57.3	85.8	44.4	30.0	22.5	18.3	16.1	17.3	15.3	13.0	13.1	13.1
4 * 2 cores AMD64	2.10	2.11	0.93	1.41	1.26	1.16	1.11	1.06	1.05			
2 * 2 cores Xeon	1.61	1.59	1.07	0.83	0.77							
2 cores Centrino	1.70	2.06	1.14									

has not been optimised for each architecture. Note that the dual issue PowerPC processor of Cell also benefits from pthread parallelism, Windows seem increase the threading overhead, the AMD numbers show large variations due to memory affinity, and the test case is relatively small for large servers.

Now, we consider the genuine Cell processor implementation. Different programming models in C/C++ are possible, see [26]. The PPU (host) and SPU (worker) execute different binaries and can be considered as heterogeneous multi-threading. However, local memory size (256 kbytes) and main memory access (direct memory access DMA block transfer only) of the SPUs is limited and the performance characteristics of both core types are different, see [21]. We chose the function off-load programming model. The main code and data resides on the host PPU. Each for loop is executed on SPUs. The current binary SPU code is downloaded to the SPU, executed and the PPU wait for the SPU termination. The SPU features no direct memory access, no cache, but user instantiated DMA block transfers. Hence the SPU code loads a data block which fits into local memory, performs the for loop on the block and write data back to main memory. This process is repeated until all data is processed. In order to hide the effect of DMA memory access, a double buffer strategy is employed. Read and write operate on data in one buffer, computations on data in the other buffer. Afterwards both buffers are swapped. Further difficulties arise from the fact that DMA memory access has to be aligned to host cache lines of 128 bytes size.

Table 2 shows again preliminary execution times for different numbers of cores. The Sony Playstation3 is based on a Cell processor featuring a dual-issue PowerPC (PPU) and 7 synergistic units SPUs. Six SPUs are available to the user code. We present the numbers for sequential and pthread parallel PPU code for comparison. The next line shows numbers for complete function off-load to the SPUs. Note that the number of SPUs involved on coarser grids is smaller for reasons of efficiency. We show numbers with a mixed SPU/PPU execution model, where coarse grid computations remain on the PPU, while fine grids are distributed over the SPUs. The wall clock time with Yellow Dog Linux and IBM xlC 8.2 SPU and PPU compilers show the performance of the PowerPC PPU, an SPU implementation with non-overlapping communication (single buffer) and multi-buffer implemementation without and with explicit Altivec parallel SIMD instructions on the SPU.

We obtain a decreasing execution time for larger numbers of cores. The single SPU performance on small data sets is slow compared to the PPU and its theoretical performance. Some additional experiments may explain this: Usual optimization uses SIMD vector operations for 4 floating point values per word, which is in our does pay off. Start-up times and synchronisation of 6 SPUs accounts for 0.43s, the sequential PPU computations for 0.19s. A strategy to fuse several loops into one SPU code to avoid repeated code load (code size of roughly 85kbytes per ForEach) has to be weighted against additional synchronisation. The remaining time is spent by SPU initiated memory transfer and overlapped computations. Further, there is a strong influence of local buffer size. We expect larger local memory size to improve overall performance. However, preliminary

Table 2. Execution times of the multigrid example, wall clock in sec on Cell processor. PPU, SPU and SPU with Altivec SIMD instructions.

no. of threads	1	2	3	4	5	6
PPU	3.12	2.56				
SPU, single buffer	3.29	1.69	1.52	1.44	1.43	1.42
SPU	2.00	1.31	1.15	1.07	1.03	1.01
SPU+Altivec	1.79	0.97	0.89	0.85	0.83	0.83

results indicate major bottleneck of main memory size. Larger problem sizes are expected to give better vector and parallel efficiency.

6 Conclusions

A domain specific parallel programming model is presented, which allows for efficient automatic parallelization of numerical computations. However, it is designed as a set of domain dependent language extensions. An application specific library provides a (distributed) data container. Further, iterators are defined on the data containers. An application, which is written as a sequence of iterators can be parallelized: The iteration code is compiled into source code using standard parallel programming models.

Acknowledgements

I want to thank the anonymous referees for their helpful comments. This work was partially supported by DFG grant SFB/TR7 “gravitational wave astronomy”. Current versions of the compiler, libraries and benchmark codes are available at `http://parallel-for.sourceforge.net`.

References

1. Butenhof, D.R.: Programming with POSIX Threads. Addison-Wesley, Reading (1997)
2. Reinders, J.: Intel Threading Building Blocks. O’ Reilly (2007)
3. Blumofe, R.D., Joerg, C.F., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: An efficient multithreaded runtime system. In: 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming PPoPP 1995, pp. 207–216. ACM, New York (1995)
4. Sutter, H.: The Concur project: Some experimental concurrency abstractions for imperative languages (2006), slides at: `http://www.nwcpp.org/Downloads/2006/The_Concur_Project_-_NWCPP.pdf`
5. Chandra, R., Menon, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J.: Parallel programming in OpenMP. Morgan Kaufmann, San Francisco (2000)
6. Pacheco, P.: Parallel programming with MPI. Morgan Kaufmann, San Francisco (1996)

7. Koelbel, C.: The High Performance Fortran handbook. MIT Press, Cambridge (1993)
8. Numrich, R.W., Reid, J.: Co-array Fortran for parallel programming. ACM Fortran Forum 17(2), 1–31 (1998)
9. Chamberlain, B.L., Callahan, D., Zima, H.P.: Parallel programmability and the Chapel language. Int. J. high perf. computing 21(3), 231–312 (2007)
10. Steele, G.: Parallel programming and parallel abstractions in Fortress. In: Proc. 14th Int. Conf. on Parallel Architectures and Compilation Techniques, pp. 157–160. IEEE, Los Alamitos (2005)
11. Ghazawi, T.E., Carlson, W., Sterling, T.L.: Distributed shared-memory programming with UPC. Wiley, Chichester (2005)
12. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: An object-oriented approach to non-uniform cluster computing. In: Proc. 20th ACM conf. on object oriented programming, pp. 519–538. ACM Press, New York (2005)
13. Herrmann, C., Lengauer, C.: HDC: A higher-order language for divide-and-conquer. Parallel Proc. Let. 10(2/3), 239–250 (2000)
14. Zumbusch, G.: Data parallel iterators for hierarchical grid and tree algorithms. In: Nagel, W.E., Walter, W.V., Lehner, W. (eds.) Euro-Par 2006. LNCS, vol. 4128, pp. 625–634. Springer, Heidelberg (2006)
15. Quinlan, D., Schordan, M., Yi, Q., de Supinski, B.R.: Semantic-driven parallelization of loops operating on user-defined containers. In: Rauchwerger, L. (ed.) LCPC 2003. LNCS, vol. 2958, pp. 524–538. Springer, Heidelberg (2004)
16. Oldham, J.D.: POOMA. A C++ Toolkit for High-Performance Parallel Scientific Computing. CodeSourcery (2002)
17. Griebel, M., Knapek, S., Zumbusch, G.: Numerical simulation in molecular dynamics. Springer, Heidelberg (2007)
18. Austern, M.H.: Generic programming and the STL. Addison-Wesley, Reading (1999)
19. Järvi, J., Powell, G.: The lambda library: Lambda abstraction in C++. In: Proc. 2nd workshop on C++ Template Programming at OOPSLA 2001 (2001), `http://www.oonumerics.org/tmpw01`
20. Birken, K.: Semi-automatic parallelisation of dynamic, graph-based applications. In: Proc. Conf. ParCo 1997, pp. 269–276. Elsevier, Amsterdam (1998)
21. Kahle, J.A., Day, M.N., Hofstee, H.P., Johns, C.R., Maeurer, T.R., Shippy, D.: Introduction to the Cell multiprocessor. IBM J. Res. & Dev. 49(4/5), 589–604 (2005)
22. Warren, M.S., Salmon, J.K.: A portable parallel particle program. Comput. Phys. Commun. 87(1–2), 266–290 (1995)
23. Lengauer, C.: Program optimization in the domain of high-performance parallelism. In: Lengauer, C., Batory, D., Consel, C., Odersky, M. (eds.) Domain-Specific Program Generation. LNCS, vol. 3016, pp. 73–91. Springer, Heidelberg (2004)
24. Bailey, D.H., Barszcz, E., Barton, J.T., Browning, D.S., Carter, R.L., Dagum, L., Fatoohi, R.A., Frederickson, P.O., Lasinski, T.A., Schreiber, R.S., Simon, H.D., Venkatakrishnam, V., Weeratunga, S.K.: The NAS parallel benchmarks. Inter. J. Supercomp. Appl. 5(3), 63–73 (1991)
25. Zumbusch, G.: Data dependence analysis for the parallelization of numerical tree codes. In: Kågström, B., Elmroth, E., Dongarra, J., Waśniewski, J. (eds.) PARA 2006. LNCS, vol. 4699, pp. 890–899. Springer, Heidelberg (2007)
26. IBM: Cell Broadband Engine Programming Tutorial. 2.1 edn. (2007)