Portable Multi-Level Parallel Programming for Cell processor, GPU, and Clusters

Gerhard Zumbusch[1]

Friedrich-Schiller-Universität Jena, 07743 Jena, Germany,
gerhard.zumbusch@uni-jena.de,
WWW home page: http://cse.mathe.uni-jena.de

Abstract. High performance computers offer lots of parallelism at different levels of vectorization, thread parallelism, message-passing between distributed memory architectures and even function off-loading by hardware accelerators. Large scale numerical simulations often have lots of parallelism, which may be difficult to express in a high level programming language. A common abstract parallel programming style is proposed, which can be translated automatically into parallel code for one or a combination of common programming styles for different parallel architectures.

1 Introduction

There is a long history of parallel programming in the area of high performance computing. However, now even low end computing platforms are all parallel, based on multi-core processors, multimedia vector instructions and numerical co-processors. Standard techniques to express parallelism in a high level programming languages include message-passing library calls, such as MPI for distributed memory platforms. Shared memory platforms can be programmed additionally with a thread library such as Posix threads and alternatively with higher level constructs for loop paralelism like in OpenMP or parallel objects [1].This is applicable to symmetric multi-processing, multi-core processors and to some extend also to non-uniform mamemory access architecture. On the lower end of parallelism, compilers take care of an even finer scale of instruction parallelism for super-scalar and pipelined instruction units. However, this is not completely true for vectorization with multimedia SIMD instructions of different flavors [2].

Large numerical simulation codes have a longer life-time than that of several parallel platforms combined. It is of vital importance to have a portable code, which runs efficiently on many of the available or future parallel platforms. There are several projects to improve parallel programming for high performance computing which target multiple parallel platforms. These include high level general purpose languages like UPC and X10.There are different data-parallel Fortran extensions. Projects with slightly more narrow focus are Pooma [3], pC++ [1], or Rose [4]. Current GPU hardware is addressed by Brook [5], RapidMind [6], Hmpp and others, while the Cell processor is supported by CellSs [7], Sequoia [8], and again RapidMind. The projects are based on restrictions to certain types

of applications and algorithms like data streams, array data structures, index based array access in loops over index sets or operations on whole or slices of arrays.

We introduce a code generation system for source-to-source translation to different parallel programming paradigm such as multi-threading, message-passing, vectorization, GPU function off-loading, Cell processor and combinations thereof. The user code in standard C++ annotates data parallel iterators over data containers (visitor pattern). The implementation of the container classes depends on the parallel target architecture. The annotated user code is analyzed by a custom version of the front and middle end of a modified version of the Gnu g++ compiler. The analysis, together with the user code is then translated into a parallel C++ code, which can be compiled by the native host compiler . This way, the necessary send and receive operations, user initiated DMA data transfers, replication of global references and global reduction operations can be created.

2 Parallel Programming Paradigms

First of all, we will discuss some of the parallel programming paradigms. The goal is to illustrate the programming style before we introduce the source-to-source compiler in section 3. We consider the following example of a one-dimensional array and a loop over an index set. Note that iterator programming model presented is applied to more general containers like tree data structures and is not restricted to increment one loops.

There is an assignment (store `x`) with non-local array access (load `y`) and a global reduction (reduce add `s`) with local array access (load `y`). If `n` is large enough, it does make sense to distribute both loops to a number of processors.

In multi-threading, the index set is partitioned and mapped to different threads. A distributed memory message passing parallelisation additionally distributes the vectors. The non-local array access causes local neighbor communication operations prior to the computation. The reduction leads to a (tree based) all-to-all communication. A SIMD parallel version using multimedia instructions with 128 bit words can perform 4 operations per instruction. Hence, the loop advances by an increment of 4. with a trailing reduction operation.

The Cell implementation runs the sequential code on the PPU. Once the loop is reached, SPU loop code is loaded onto the SPUs. Each SPU performs a fraction of the operations like in the multi-threading case. However, the SPU code divides data further into blocks that fit into local memory. In order to overlap computation and data transfer, a multi-buffer strategy is employed. A load operation for future data block, a loop over the current data block, and a store operation on past data block are executed in parallel.

Table 1. Non-local memory access and reduction example.

```
for(int i=1;i<n;++i) { x[i]=.5*(y[i+1]+y[i-1]); s += sqr(y[i]);}
```

Table 2. Parallel loop syntax.

```
Grid1IteratorSub it(...);
ForEach(int i, it, 'x(i) = .5* (y(i+1) + y(i-1)); s += sqr(y(i));')
```

Table 3. Automatic code analysis.

```
block 1   load  'y' 'DistArray1<float>' [#-1,#+1]
          store 'x' 'DistArray1<float>' [#,#]
          reduce add 's' 'float'
```

3 The Parallel Code Generation

The basic assumption in our approach is that the user is able to mark parallel sections as parallel in the sense of data-parallel with possible local neighbor communication and possible reduction operations. A compiler may fail to identify such sections itself, because global code dependence analysis may be necessary to do so. Note that data layout and data distribution is also of vital importance. From a user perspective, good application specific data storage schemes may be available, which are again not known to a general compiler. Hence, we opt to put these into a run-time library.

In a previous effort to implement a similar project [9] for message-passing and multi-threading, a dynamic, speculative dependence analysis was performed. The current compilation system is based on an extension of the Gnu g++ 4.2 compiler and operates on the tree static single assignment (tree SSA) representation of the code. For each `ForEach` block, a table of variables loaded and stored is created. In the case of distributed objects, the precise relative element accessed is also included. This way it is possible to determine local access `(i)` and non local access. The data dependence analysis is also capable of tracing pointer accesses inside linked data structures like trees, e.g. see table 3.

There is a single data dependence analysis, but multiple code generators. We consider different target codes like multi-threading, message-passing, vectorization, procedure off-loading onto a graphics card (GPU) and a specialized Cell function off-loading model.The generated parallel code is linked to a platform dependent auxiliary runtime library.

4 Experimental Evaluation

We have chosen a single benchmark code for the evaluation of the compiler analysis, code generation system and the different programming models. A multigrid equation solver similar to a code of NAS parallel benchmark [10] is written with different sized arrays and `ForEach` loops. The code uses a nested set of three-dimensional arrays and grids. An iterative method for the solution of an equation system created by a constant finite difference stencil computes a solution. All functions are stored as arrays. The multigrid method improves the convergence rate of the iterative method by additional calculations on auxiliary coarser grids.

In figure 1 results are shown for a PC cluster of dual-core nodes, a Sony Playstation 3 with Cell processor and a couple of Nvidia graphics cards. We observe a good parallel scaling in proceesor cores, GPU engines and Cell SPUs.

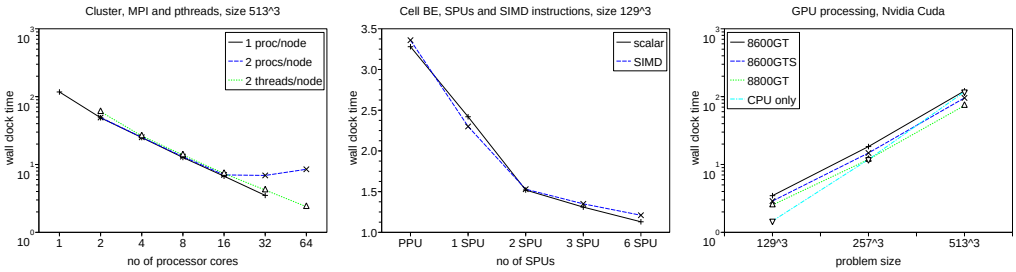

Fig. 1. Experiments for message-passing and threading, Cell processor SPU co-processors and SIMD instructions, and GPU computing.

Acknowledgment

The authors would like to thank DFG for partial support under 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. Bodin, F., Beckman, P., Gannon, D., Narayana, S., Yang, S.X.: Distributed pC++: Basic ideas for an object parallel language. Scient. Program. **2**(3) (1993) 7–22
2. 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) (2005) 589–604
3. Oldham, J.D.: POOMA. A C++ Toolkit for High-Performance Parallel Scientific Computing. CodeSourcery (2002)
4. Schordan, M., Quinlan, D.: A source-to-source architecture for user-defined optimizations. In Böszörmenyi, L., Schojer, P., eds.: Modular Programming Languages. Volume 2789 of LNCS., Springer (2003) 214–223
5. Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M., Hanrahan, P.: Brook for GPUs: Stream computing on graphics hardware. ACM Trans. Graph. **23**(3) (2004) 777–786
6. McCool, M.D.: Data-parallel programming on the Cell BE and the GPU using the RapidMind development platform. Technical report, RapidMind Inc. (2006) GSPx Multicore Applications Conference, Santa Clara.
7. Perez, J.M., Bellens, P., Badia, R.M., Labarta, J.: CellSs: Making it easier to program the Cell BE processor. IBM J. Res. Dev. **51**(5) (2007) 593–604
8. Fatahalian, K., Knight, T.J., Houston, M., Erez, M., Horn, D.R., Leem, L., Park, J.Y., Ren, M., Aiken, A., Dally, W.J., Hanrahan, P.: Sequoia: Programming the memory hierarchy. In: Supercomputing 2006. (2006) 4
9. Zumbusch, G.: Data parallel iterators for hierarchical grid and tree algorithms. In Nagel, W., Walter, W., Lehner, W., eds.: Euro-Par 2006 Parallel Processing. Volume 4128 of LNCS., Springer (2006) 625–634
10. 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) (1991) 63–73