Proceedings of the 21st IASTED International Conference
APPLIED INFORMATICS
February 10-13, 2003, Innsbruck, Austria

A Multithreaded Compiler Backend for High-Level Array Programming

Clemens Grelck
Institute of Software Technology and Programming Languages
University of Lübeck
Ratzeburger Allee 160
23538 Lübeck, Germany
E-mail: grelck@isp.uni-luebeck.de

ABSTRACT
Whenever large homogeneous data structures need to be processed in a non-trivial way, e.g. in computational sciences, image processing, or system simulation, high-level array programming in the style of APL offers a far more concise and abstract approach than traditional scalar languages such as C/C++ or FORTRAN-77. The same sort of applications often can also be characterized as performance critical and today represents the major domain for parallel processing.

This paper reports on the development of a compiler backend which allows to implicitly generate multithreaded code from high-level array program specifications. On shared memory multiprocessor systems, this code can be executed in parallel without any additional programming effort. After sketching out basic compilation schemes, optimizations on the runtime system are addressed and, finally, experimental runtime figures are presented.

KEY WORDS
Compilers, Array Programming, High-level Parallel Programming, High Performance Computing

1 Introduction

Processing of large homogeneous data structures in areas such as computational sciences, image processing, or system simulation constitutes the most prominent application domain for parallel processing. Despite the ubiquity of arrays in parallel programs, the prevailing programming languages used, i.e. C and FORTRAN-77, are inherently scalar and offer only minimal support for processing multi-dimensional arrays. They allow experienced programmers to achieve utmost runtime performance, but at the same time they enforce a very low-level programming style. As a consequence, programs are typically difficult to read and to understand. Explicit parallelization using common message passing libraries, e.g. MPI [9], PVM [8], or BSP [22], adds yet another dimension of complexity, which renders program development, debugging, and maintenance even more time-consuming and error-prone.

In contrast, high-level array languages like APL [15] or J [4, 16] make array processing almost as simple as dealing with scalars in traditional languages. Arrays are regarded as abstract data objects with a certain structure rather than as loose collections of individual data items or even direct mappings into main memory. Programming merely means the composition of basic homogeneous array operations, which are applicable to arrays of any shape including any number of dimensions, to form more complex, more application-specific ones. With memory management also being implicit, programs mostly abstract from concrete computing machinery. This abstract view together with the inherent concurrency of high-level array operations also make them attractive for implicit or automatic parallelization [1].

This paper reports on the development of compiler support for the implicit parallelization of array language programs for shared memory multiprocessors based on the multithreading standard PTHREADS [5]. The choice of this combination is motivated by several factors.

- Small to medium-sized shared memory multiprocessors represent an increasingly popular architecture, which is more wide-spread than traditional large-scale supercomputers.

- Despite the advent of OPENMP [6] parallel programming is dominated by the concept of message passing both as programming environment and as compilation target. However, message passing covers shared memory architectures only indirectly by specific versions of library implementations.

- A shared memory view on the level of the compilation target renders an explicit data decomposition obsolete. This does not only simplify the compilation process, but — even more important — it reduces communication requirements. With clever low-level implementations communication and synchronization overhead can be reduced with the consequence that even smaller computational tasks benefit from parallel execution.

Putting it all together, the approach described here addresses non-experts in parallel programming who want to

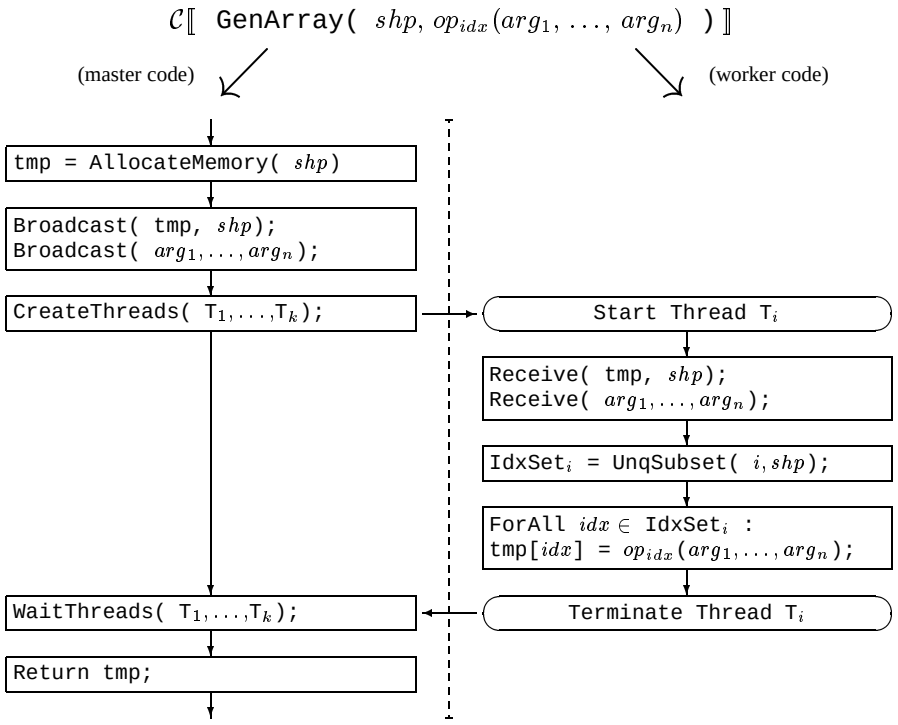

Figure 1. Compilation scheme for `GenArray` operations.

boost the performance of their programs with a multiprocessor machine next door rather than the whizz in a supercomputer lab with long experience and unlimited resources.

Although it is unlikely for a compiler-directed approach to achieve the same performance as a low-level program hand-coded by some expert, the latter should at least be approached. Hence, it is mandatory to start out from a sequential runtime performance that is not orders of magnitude away from traditional imperative programming environments. In this regard, array languages, as the ones mentioned before, must be considered inappropriate. They are implemented by interpreters which restricts static analysis and optimization; their flexibility renders compilation very difficult [7, 3, 2]. Therefore, the techniques described in this paper are not developed and implemented in the context of APL or J, but in the context of SAC (Single Assignment C). SAC [19, 21] is a functional array language, which offers almost the level of abstraction as APL [12]. However, some restrictions in conjunction with thorough optimizations allow to achieve runtime performance characteristics which are in a similar range as C or FORTRAN-77 [18, 20, 13].

The rest of this paper is organized as follows. Section 2 sketches out the basic compilation schemes; Section 3 introduces an enhanced runtime system. Scheduling work onto threads is addressed in Section 4. Section 5 investigates the runtime performance achieved, and Section 6 concludes.

2 Compiling Array Operations to Multi-threaded Code

Array languages provide a large set of array operations, either built-in or via libraries. However, almost all of them fall into one of two basic categories: they either create a new array whose elements are individually computed from some arguments based on their index positions or they perform a reduction operation. To abstract from individual properties of concrete operations we introduce two generalized operations representing these categories:

$$\texttt{GenArray(}\ shp, op_{idx}(arg_1, \ldots, arg_n)\ \texttt{)}\quad ,$$
$$\texttt{FoldArray(}\ shp, op_{idx}(arg_1, \ldots, arg_n), \textit{fold\_op}, \textit{neutral}\ \texttt{)}\quad .$$

`GenArray` creates a new array of shape shp, where shp denotes an integer vector defining both the number of dimensions as well as its extent in each dimension. Its elements are separately computed by some operation op_{idx}. Being parameterized over individual index positions, different operations may actually be realized by op_{idx} on disjoint areas of the result array. With its arity left unspecified any number of scalar and array arguments may occur. So, `GenArray` and `FoldArray` may be considered operational templates rather than higher-order functions.

Similar to `GenArray`, `FoldArray` evaluates the given function op_{idx} for each index position associated with the shape *shp*. Instead of using the results for initializing a new array, they are pairwise folded using the binary fold operation *fold_op* with neutral element *neutral*.

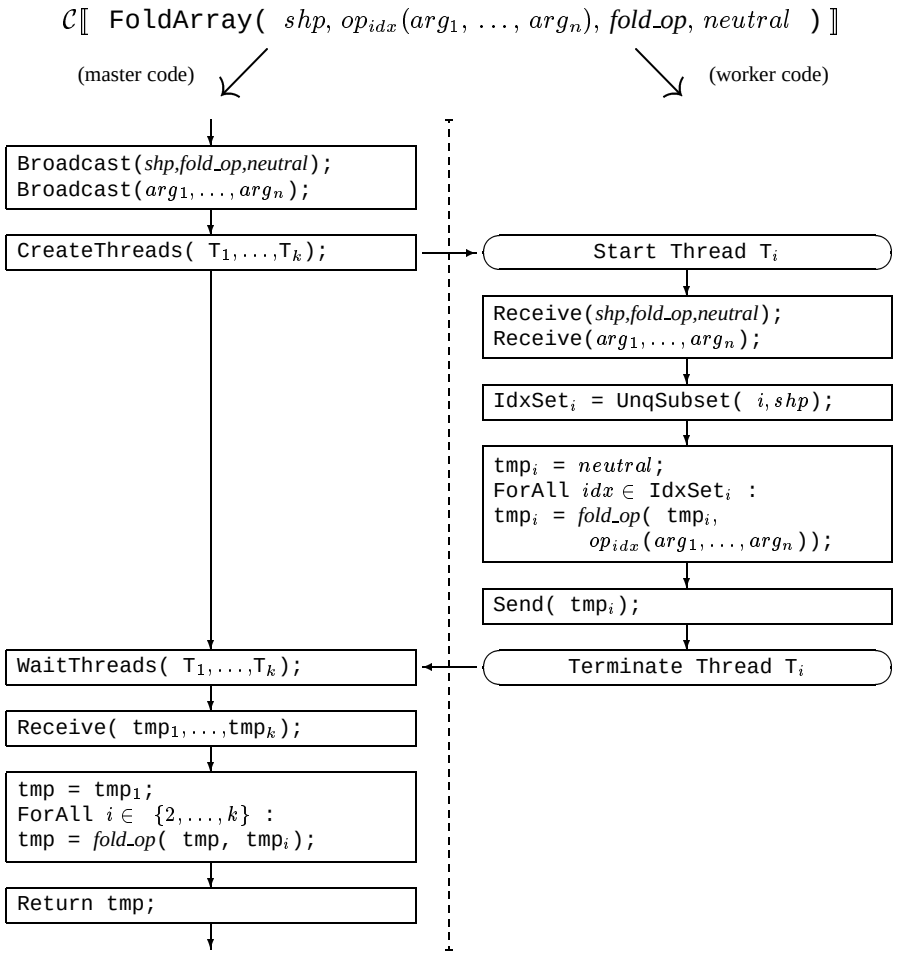

Figure 2. Compilation scheme for `FoldArray` operations.

Since the concrete sequence of folding operations is left unspecified, legal fold operations must be associative and commutative to ensure deterministic results.

Fig. 1 shows the compilation scheme for `GenArray` operations into imperative pseudo code. Whenever the initial or master thread evaluates a `GenArray` operation, it first allocates memory for storing the result array, which is referred to by some previously unused variable `tmp`. Due to the commitment to shared memory architectures, no explicit data decomposition is required, and implicit dynamic memory management for arrays can be adopted from sequential implementations with little or no alteration. Afterwards, the base address and the shape of the result array as well as the numerical arguments of the operation are broadcast, and, last but not least, the desired number of worker threads is created.

At first, it seems inconsistent to send data to worker threads prior to their creation. However, in a shared memory environment `send` and `receive` are nothing but copy operations to and from some specific memory buffer, which may exist independently of the threads themselves. Broadcasting data prior to thread creation allows for a non-blocking implementation of the corresponding `receive` operation, and, thus, reduces synchronization requirements among threads to their creation. It is also notable that communication of compound data structures such as arrays merely means sending references between threads, not the data itself.

All worker threads uniformly execute the code shown on the right hand side of Fig. 1, but each thread may identify itself by means of a unique ID. As a first step, a worker thread receives the necessary arguments to set up an appropriate execution environment. Then, based on its unique ID, each worker thread identifies a subspace of the total index space. Proper implementations of `UnqSubset` guarantee that each legal index position belongs to exactly one such index subspace. For each element of its individual index subspace a worker thread computes the corresponding numerical operation and initializes the result array accordingly. After having completed their individual computations, the worker threads terminate.

While worker threads cooperatively compute an array operation, the master thread just awaits their termination. As soon as the last worker thread has completed its share of work, the master thread returns the result to the surrounding context and continues with sequential program execution.

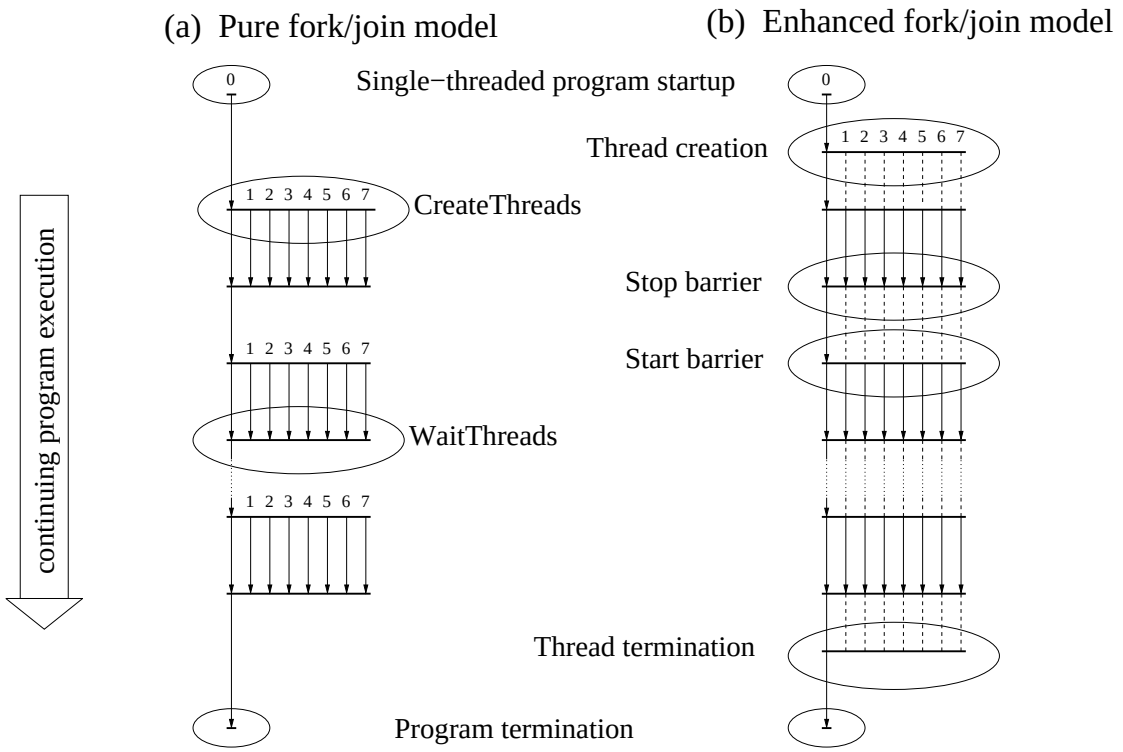

Figure 3. Illustration of execution models.

Code generation for `FoldArray`, as shown in Fig. 2, is quite similar. However, instead of computing elements of a target array, worker threads initialize a local accumulation variable $\texttt{tmp}_i$ by the neutral element of the fold operation and then perform the specified computations restricted to the individual index subspace identified before. Hence, each worker thread computes a partial fold result, which it sends back to the master thread prior to its termination.

The master thread awaits the termination of all worker threads before it receives their partial fold results. Once again, necessary thread management operations are exploited to ensure proper synchronization upon `send`/`receive` communication. Finally, the master thread itself combines the various partial fold results to generate the overall result.

3 Enhancing the Execution Model

Code derived by the compilation schemes introduced in the previous section executes as a sequence of steps alternatingly performed in single-threaded and in multithreaded mode. Following the multithreaded execution of one array operation, all worker threads are terminated during synchronization, and the same number of threads is created again for the multithreaded execution of the next array operation, as illustrated in Fig. 3.

This fork/join model is conceptually simple. Synchronization and communication are limited to thread creation and thread termination. Worker threads do not interact with each other at all. Unfortunately, frequent creation and termination of threads is a considerable source of overhead.

A solution which combines the conceptual simplicity of the fork/join approach with an efficient execution scheme is shown on the right hand side of Fig. 3. In the *enhanced fork/join model*, the desired number of worker threads is created once at program startup, and they remain active until the whole program terminates. All necessary synchronization among threads is realized by means of two tailor-made barriers: the *start barrier* and the *stop barrier*.

After creation, worker threads immediately hit a start barrier, which is lifted as soon as the master thread encounters the first array operation. The master thread and all worker threads thereupon activated share the computation of the array operation exactly as in the pure fork/join model. Worker threads that have completed their individual computations pass the following stop barrier and, with nothing else to do, immediately move on to the next start barrier. However, the master thread waits at the stop barrier for the last worker thread to arrive before it proceeds with subsequent (sequential) computations.

The combination of a stop barrier and a subsequent start barrier represents a full barrier synchronization, which is known to cause considerable runtime overhead and to scale poorly with the number of threads [14]. Therefore, the efficient implementation of start and stop barriers is crucial for runtime performance. Fig. 4 shows one such implementation. It is based on a global flag,

which is shared by all threads, and on one local flag within the scope of each worker thread. Assuming all flags are statically initialized, say to 1, worker threads executing START_BARRIER_WAIT block on the condition of the empty WHILE-loop. By inverting the global flag during execution of START_BARRIER_LIFT, the master thread lifts the barrier and, thus, activates the worker threads.

```
START_BARRIER_WAIT( )
{
  while (local_flag == MT_global_flag);
  local_flag = MT_global_flag;
}

START_BARRIER_LIFT( )
{
  MT_global_flag = 1 - MT_global_flag;
}
```

Figure 4. Implementation of start barriers.

This start barrier implementation completely avoids expensive thread synchronization mechanisms such as mutex locks. Although inverting the global flag while worker threads continuously check its value, constitutes a race condition, this is without problems. On the one hand, only the master thread has write access to the global flag. On the other hand, for worker threads solely the fact that the flag has changed its value is important and not the specific state of the memory location.

One may argue that iteratively reading the global flag in very short intervals of time generates heavy contention on memory. In fact, the opposite is true for modern shared memory multiprocessors with processor-specific cache memories and hardware cache coherence. As soon as an individual worker thread arrives at the start barrier, it loads the global flag into the local cache of the processor it currently is running on. From that point on, it only accesses the local cache when blocking on the global flag. However, when the master thread inverts the global flag and writes its new value back to memory, the cache coherence mechanism invalidates the local copies in all other caches. Worker threads only then reload the global flag from memory and, thereupon, proceed beyond the start barrier. In total, each thread performs at most two main memory accesses during execution of the start barrier. The sole assumption made on the memory consistency model is that write operations issued by one processor are noticed by others.

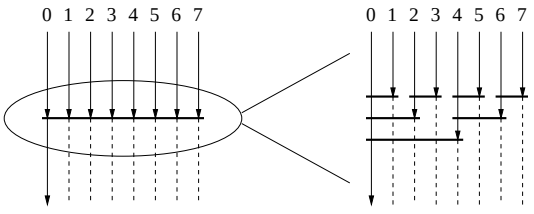

Figure 5. Organization of tree-structured stop barriers.

The stop barrier employs the same synchronization technique as the start barrier. However, its scalability is improved by a tree-like organization, as illustrated in Fig.5. Threads with an odd ID simply pass the stop barrier, immediately running into the subsequent start barrier. Each thread with an even ID n waits for thread $n+1$ to complete. Then, it either passes the stop barrier itself if its ID is not a multiple of 4, or it continues to wait for thread $n+2$ otherwise, and so on.

In the enhanced fork/join execution model the impact of thread creation and of thread termination on runtime performance decreases with growing overall program execution time. Nevertheless, it makes sense in principle to pay attention to the efficiency of their implementations. In a straightforward approach, the master thread creates all worker threads one after the other by means of a FOR-loop. Execution of productive code is delayed by a time which grows linearly with the number of threads.

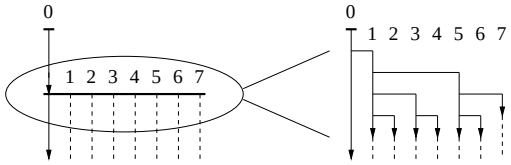

Figure 6. Organization of thread creation phase.

This initial delay can be easily reduced by having the worker threads participate in thread creation. This leads to a binary tree thread creation scheme similar to the stop barrier implementation. With this solution, the initial delay can be reduced to $\mathcal{O}\left(\lceil log_2 \texttt{NUM\_THREADS} \rceil\right)$. However, it may be further reduced to only $\mathcal{O}(1)$ by excluding the master thread from thread creation. As outlined in Fig. 6, the master thread creates exactly one worker thread and then immediately starts with the execution of productive code. Instead of the master thread, the first worker thread subsequently initiates a regular binary tree thread creation scheme. With this solution, thread creation almost completely overlaps with a program's sequential startup phase, e.g. reading input data from files.

4 A Note on Scheduling

Regardless of the execution model, the even distribution of workload among worker threads is a key issue for achieving good performance. The clear separation of scheduling code (UnqSubset) from computational code, as shown in Figs. 1 and 2, allows to plug-in various different scheduler implementations. The usual static workload distribution schemes are supported without interfering with actual code generation for the computational part of the code. This feature is particularly essential as code generation for array operations as general as the ones covered here can be quite challenging [11].

Unfortunately, the compilation schemes presented in

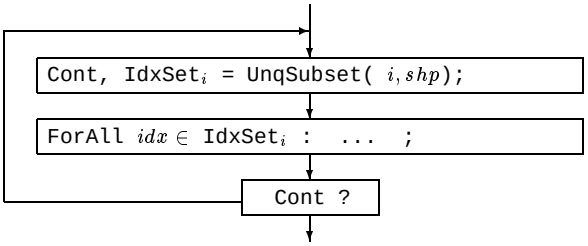

Figure 7. Enabling workload balancing.

Section 2 exclude any form of dynamic load balancing because each thread selects exactly one iteration subspace. However, dynamic workload adjustment relies on repeatedly assigning smaller portions of work. To overcome this limitation, the compilation schemes are extended as shown in Fig. 7. In addition to determining some iteration subspace, a scheduler implementation also provides a flag `cont` which decides whether or not the scheduler wishes to re-assign more work to the thread at a later stage of the computation. This solution extends the simple plug-in technique to a wide range of dynamic scheduling schemes without substantially changing the compilation process. However, certain scheduler implementations may require synchronization among threads on a lower level of abstraction and, hence, must be designed and selected carefully.

5 Runtime Performance

Experimental investigations of the runtime performance achieved by the methods described in the previous chapters have been made on 3 different machine architectures: a 4-processor SUN E650, a 12-processor SUN E4000, and a 72-processor SUN 15k. 2-dimensional Jacobi relaxation with a 4-point stencil served as a benchmark kernel. Although being quite simple, it is still not a trivial benchmark as it does require data exchange between processors after each iteration.

Fig. 8 shows speedups achieved by multithreaded program execution relative to sequential execution. Access to the two larger machines has been non-exclusive. Hence, not all processors could actually be used. Various grid sizes have been investigated, as shown for each machine individually. Since there is always a full synchronization after re-computation of the grid, its size determines the ratio between productive computation and administrative overhead inflicted by parallel execution.

Experiments which also cover extremely small grid sizes demonstrate the efficiency of the proposed solution. Superlinear speedups observed for some problem sizes on the SUN E15k are due to cache effects. For certain combinations of problem size and cache configuration taking one additional processor may result in a situation where all memory processed by each individual processor completely fits into its local caches.

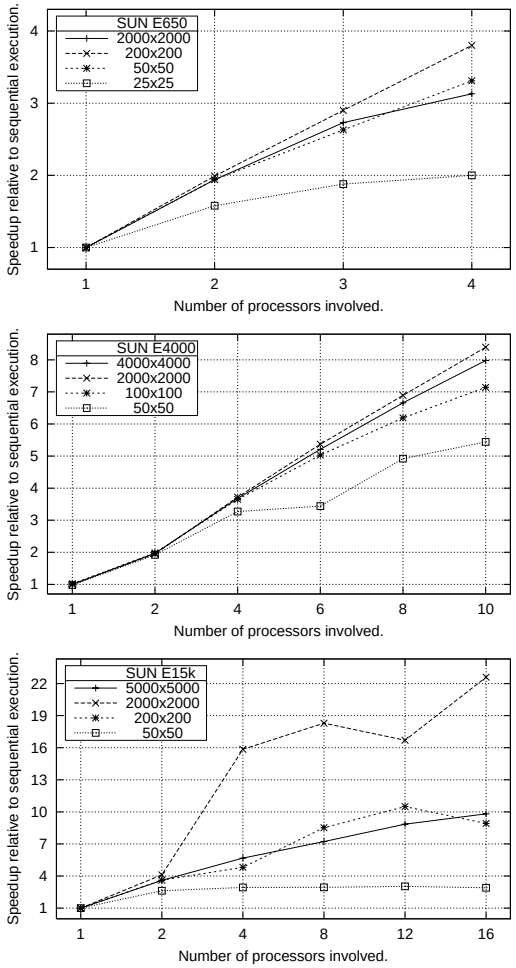

Figure 8. Speedups observed on different machines.

6 Conclusion and future work

High-level array programming offers the opportunity to write concise and elegant code in various application domains. Such programs also tend to exhibit large degrees of concurrency, which can be exploited for substantially reducing program runtimes by parallel execution. This paper describes compiler and runtime system support that allows exploitation of program-inherent concurrency without any additional programming effort. Experiments on three different SMP architectures demonstrate the suitability of the approach in principle. Additional information on technical realizations can be found in [10].

Opportunities for future improvements are still manifold. As SMP systems grow in size, it may often not be feasible to efficiently use all available processors to cooperatively compute a single array operation. Hence, it may be necessary to combine task and data parallel concepts similar to the *group SPMD model* [17]. In the context of SAC, concurrency between array operations can be detected easily due to its purely functional semantics.

References

[1] R. Bernecky. The Role of APL and J in High-Performance Computation. *APL Quote Quad*, 24(1):17–32, 1993.

[2] R. Bernecky. An Overview of the APEX Compiler. Technical Report 305/97, University of Toronto, Toronto, Canada, 1997.

[3] T. Budd. *An APL Compiler*. Springer-Verlag, Berlin, Germany, 1988.

[4] C. Burke. *J and APL*. Iverson Software Inc., Toronto, Canada, 1996.

[5] D.R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, Reading, Massachusetts, USA, 1997.

[6] L. Dagum and R. Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Transactions on Computational Science and Engineering*, 5(1), 1998.

[7] G.C. Driscoll and D.L. Orth. Compiling APL: The Yorktown APL Translator. *IBM Journal of Research and Development*, 30(6):583–593, 1986.

[8] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine*. MIT Press, Cambridge, Massachusetts, USA, 1994.

[9] A. Geist, W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, W. Saphir, T. Skjellum, and M. Snir. MPI-2: Extending the Message-Passing Interface. In *Proceedings of the 2nd European Conference on Parallel Processing (Euro-Par'96), Lyon, France*, volume 1123,1124 of *Lecture Notes in Computer Science*, pages 128–135. Springer-Verlag, Berlin, Germany, 1996.

[10] C. Grelck. *Implicit Shared Memory Multiprocessor Support for the Functional Programming Language SAC — Single Assignment C*. PhD thesis, Institute of Computer Science and Applied Mathematics, University of Kiel, Germany, 2001. Logos Verlag, Berlin, 2001.

[11] C. Grelck, D. Kreye, and S.-B. Scholz. On Code Generation for Multi-Generator WITH-Loops in SAC. In *Proc. 11th International Workshop on Implementation of Functional Languages (IFL'99), Lochem, The Netherlands, selected papers*, volume 1868 of *LNCS*. Springer, 2000.

[12] C. Grelck and S.-B. Scholz. Accelerating APL Programs with SAC. In O. Lefevre, editor, *Proceedings of the International Conference on Array Processing Languages (APL'99), Scranton, Pennsylvania, USA*, volume 29 of *APL Quote Quad*, pages 50–57. ACM Press, 1999.

[13] C. Grelck and S.-B. Scholz. HPF vs. SAC — A Case Study. In A. Bode, T. Ludwig, W. Karl, and R. Wismüller, editors, *Proceedings of the 6th European Conference on Parallel Processing (Euro-Par'00), Munich, Germany*, volume 1900 of *Lecture Notes in Computer Science*, pages 620–624. Springer-Verlag, Berlin, Germany, 2000.

[14] J.M.D. Hill and D.B. Skillicorn. Practical Barrier Synchronisation. In *Proceedings of the 6th Euromicro Workshop on Parallel and Distributed Processing (PDP'98), Barcelona, Spain*, pages 438–444. IEEE Computer Society Press, 1998.

[15] K.E. Iverson. *A Programming Language*. John Wiley, New York City, New York, USA, 1962.

[16] K.E. Iverson. *Programming in J*. Iverson Software Inc., Toronto, Canada, 1991.

[17] T. Rauber and G. Rünger. Deriving Structured Parallel Implementations for Numerical Methods. *Microprocessing and Microprogramming*, 41:589–608, 1995. Elsevier.

[18] S.-B. Scholz. On Programming Scientific Applications in SAC - A Functional Language Extended by a Subsystem for High-Level Array Operations. In *Proc. 8th International Workshop on Implementation of Functional Languages (IFL'96), Bonn, Germany, selected papers*, volume 1268 of *LNCS*. Springer, 1997.

[19] S.-B. Scholz. On Defining Application-Specific High-Level Array Operations by Means of Shape-Invariant Programming Facilities. In S. Picchi and M. Micocci, editors, *Proceedings of the International Conference on Array Processing Languages (APL'98), Rome, Italy*, pages 40–45. ACM Press, 1998.

[20] S.-B. Scholz. A Case Study: Effects of WITH-Loop Folding on the NAS Benchmark MG in SAC. In *Proc. 10th International Workshop on Implementation of Functional Languages (IFL'98), London, UK, selected papers*, volume 1595 of *LNCS*. Springer, 1999.

[21] Sven-Bodo Scholz. Single Assignment C — Efficient Support for High-Level Array Operations in a Functional Setting. *Journal of Functional Programming*. Accepted for publication.

[22] D.B. Skillicorn, J.M.D. Hill, and W.F. McColl. Questions and Answers about BSP. *Scientific Computing*, 6(3):249–274, 1997.