Lessons learned from the Shared Memory Parallelization of a Functional Array Language (Draft)

Clemens Grelck
University of Lübeck
Institute for Software Technology and Programming Languages
23569 Lübeck, Germany
grelck@isp.mu-luebeck.de

Abstract

This paper reports on the experiences made with the parallelization of a functional array language called SAC. The high-level approach together with a side-effect free semantics make it a good candidate for this purpose. In order to realize the project with limited man power, shared memory systems are selected as target architectures and the implementation based on PTHREADS. *These choices allow reuse of large parts of the existing compiler infrastructure as it avoids explicit data decomposition.*

In fact, this setting significantly simplified the creation of an initial solution. However, with respect to performance some architecture-specific pitfalls required careful attention, namely multithreaded memory management and the utilization of processor-private cache memories. These problems are discussed and solutions are outlined. In the end, it turned out that getting the right performance is at least as challenging in this setting as it is with a distributed memory target architecture.

1 Introduction

Functional programming languages are well-known for providing a very high level of abstraction. With higher-order functions, polymorphic type systems, implicit memory management, and functions which behave like mathematical functions following a call-by-value parameter passing convention, they allow for concise program specifications close to mathematical notations. Organizational details of program execution are left to language implementations, i.e. to compilers and runtime systems. By ruling out side-effects on a conceptual level and by defining program semantics based on the principle of context-free substitutions, functional programs usually are also considered ideal candidates for parallel execution.

However, program parallelization is usually motivated by the need to increase the runtime performance of programs beyond what can be accomplished by sequential execution on a single processor. To justify the additional effort both in hardware and in software, parallelization only makes sense for certain performance-critical application domains and it must start out from the basis of excellent sequential performance characteristics.

Unfortunately, functional programming is less well-known for these two aspects. All the programming amenities mentioned before have their price in terms of runtime overhead. Even more important, traditional parallel or high performance computing is dominated by array processing, whereas functional programming focuses on lists and trees as prevailing data structures. Hence, apart from the generally high level of abstraction, their support for array processing is often limited. Much worse, their runtime performance turns out to be clearly inferior to what is accomplished by imperative programs in uniprocessor environments [16, 15].

Unfortunately, this is not only an implementation problem, but, to some extent, inherent to the programming paradigm. Conceptually, functions consume arguments and create values from scratch. For small data objects this can be implemented fairly efficiently. However, operations on large homogeneous arrays which "change" just a few array elements should be performed destructively, whenever possible. Otherwise, large amounts of data would have to be copied, thus considerably degrading runtime performance with increasing array sizes.

Some functional languages, e.g. ML or ID, try to circumvent this problem by introducing arrays as non-functional, stateful data structures [23, 2] and by providing side-effecting operations on them. Though this approach allows for reasonable runtime performance characteristics, it mostly sacrifices the idea of high-level programming as far as arrays are involved, e.g., arrays have to be allocated, copied, and removed explicitly.

Languages which follow a lazy evaluation regime like CLEAN or HASKELL encounter some specific pitfalls with respect to efficient array processing. Whenever strictness cannot be inferred, several slightly modified versions of an array may have to be held simultaneously in different environments resulting in an explosion of memory requirements. Furthermore, the usage of garbage collectors rules out destructive implementations of array operations since it is generally undecidable whether or not additional references to argument arrays exist. The sole remedy to these problems is to implement arrays based on language facilities for the functionally sound realization of states, e.g. uniqueness types [34] or state monads [36, 19]. Both mechanisms guarantee that at most one reference to an array exists, and, hence, operations can always be performed destructively [20]. However, once again, a more low-level, imperative-like programming style is the consequence [33].

Very few functional languages are specifically designed with array processing in mind. After the development of Sisal [22, 4] has come to an end, SAC or **S**ingle **A**ssignment **C**[30] seems to be the most recent approach. SAC clearly builds upon the achievements of Sisal, yet it tries to alleviate some of its shortcomings and generally allows for significantly more abstract program specifications.

Unlike Sisal, SAC supports truly multi-dimensional arrays. Memory management for arrays again is based on reference counting [6, 7], which permits destructive implementations of array operations in certain situations. SAC provides only a few basic operations on arrays as built-in functions. In contrast, all aggregate array operations are specified by means of a SAC-specific array comprehension, the so-called WITH-*loop*, in SAC itself. They permit specifications which completely abstract from the dimensionalities of the arrays involved. Moreover, SAC allows to embed such general specifications within functions which are applicable to arrays of any dimensionality and size.

In fact, a similar functionality as provided by Fortran-90/95 or by interpreted array languages, e.g. APL or NIAL [18] can be implemented in SAC with almost no loss of generality [13]. This allows to adopt an APL-like programming style which constructs application programs by nesting pre-defined high-level, general-purpose array operations, whereas these basic building blocks themselves are implemented by means of WITH-loops.

Whenever during program development such an operation is found to be missing, it can easily be added to the repertoire and reused in future projects. A comprehensive selection of standard array operations is already provided as a library. Being implemented in the language itself rather than hard-wired into the compiler, they are easier to maintain, to extend, and to customize for varying requirements.

Notwithstanding the considerably higher level of abstraction, SAC has demonstrated its potential of outperforming Sisal as well as Fortran-77 in uniprocessor environments [29, 32]. Hence, SAC basically meets the aforementioned requirements for being an interesting candidate for parallelization. Typical SAC programs can be expected to spend considerable portions of their execution times in array operations. Since in SAC almost all such operations, in one way or another, boil down to nestings of WITH-loops in intermediate program representations, all efforts to generate concurrently executable code can be directed to this single language construct. Furthermore, preceding optimization steps [31, 32] often result in WITH-loops whose computational complexities per element exceed that of typical built-in aggregate array operations known from other languages.

Today, almost all high-level approaches in the field of parallel programming rely on MPI for their realization. Abiding to the leading industry standard ensures portability across a wide range of architectures. However, message passing inherently is a distributed memory programming paradigm. Shared memory systems are only indirectly supported by specific implementations of message passing routines, but the shared memory is not exposed at the programming level. This unnecessary indirection is likely to introduce additional overhead which could be avoided by a dedicated solution.

With the advent of processor-specific hierarchies of large and fast cache memories and the development from a physically shared memory to a shared address space on top of physically distributed memory shared memory multiprocessors have become an increasingly popular and wide-spread class of machines in recent years. The concept of having multiple threads of control within the shared address space of a single process perfectly matches this architecture. Therefore, the POSIX multithreading standard PTHREADS [17] is selected as a compilation target to ensure portability across different concrete machine implementations and operating systems.

However, the commitment to shared memory architectures also provides some concrete benefits for supporting parallel program execution. For instance, the existing internal representation of arrays may be adopted without modification, as an explicit data decomposition is obsolete. Hence, integral parts of the existing compiler framework can be reused, thus reducing precious man power.

The paper is organized as follows. Section 2 provides a very brief introduction into SAC. The compilation of SAC program specifications into multithreaded code is outlined in Section 3. Unfortunately, some specific pitfalls apply to achieving high performance on shared memory systems. They are addressed in Sections 4 and 5. Finally, Section 6 draws some conclusions.

2 SAC — Single Assignment C

The core language of SAC is a functional subset of C, a design which aims at simplifying adaptation for programmers with a background in imperative programming. This kernel is extended by multidimensional arrays as first class objects. With implicit memory management and a strict call-by-value semantics for function applications SAC allows for high-level array processing in the style of APL.

```
bool continue( double[+] A,
               double[+] A_old,
               double    eps)
{
  return( any( abs( A - A_old) >= eps));
}
```

Figure 1. SAC example: convergence test.

As an example, Fig. 1 shows a SAC implementation of the termination condition of some iterative algorithm. It takes two double precision floating point number arrays of arbitrary (but equal) shape (`double[+]`) and yields `true` as long as for any array element the absolute difference between its current and its old value exceeds the given convergence criterion `eps`.

What distinguishes SAC from other array languages is basically twofold. First, SAC allows for function abstractions on arrays of arbitrary shape, including varying numbers of dimensions. Second, almost all basic array operations, e.g. those shown in Fig. 1, are not hard-wired into the language as primitive operations, but they are implemented in SAC themselves based on a versatile array comprehension construct, the WITH-loop.

```
double[+] abs( double[+] A)
{
  res = with (. <= iv <= .)
        {
          if (A[iv] < 0.0) val = -A[iv]
          else             val =  A[iv]
        }
        genarray( shape(A), val)

  return( res);
}
```

Figure 2. SAC implementation of `abs`.

Fig. 2 shows the SAC implementation of the function `abs` used in the previous example. It consists of a single WITH-loop which creates a new array of the same shape as the argument array `A`. For each of its index positions, represented by the variable `iv`, the body is evaluated, and the result array is initialized accordingly.

More complex *generators*, i.e. the expression following the key word `with`, allow to restrict operations to subranges of arrays as well as to periodic grids. Moreover, additional WITH-loop variants may be used for manipulating existing arrays as well as for realizing reduction operations. More information on programming in SAC may be found in [14, 11] or at `http://www.informatik.uni-kiel.de/~sacbase/`.

3 Compilation into Multithreaded Code

Being a dedicated array language, it is reasonable to assume that typical SAC programs spend almost all of their execution time in compound array operations. Moreover, these compound array operations in one way or the other are all implemented in SAC itself by means of WITH-loops. Hence, all effort in compilation may be directed to this single language construct. The WITH-loop itself is extremely well-suited for parallel execution as it defines a new array whose elements may be computed independently of each other by definition.

Additionally, the commitment to shared memory architectures and multithreading as underlying organizational layer renders explicit data decomposition obsolete. In conjunction, these design choices allow for a fairly straightforward compilation scheme, as illustrated in Fig. 3.

Even if compiled for multithreaded execution, SAC programs are generally executed sequentially by a single thread of control. Only when it comes to execution of a WITH-loop and after allocation of memory for storing the complete result array, additional worker threads are created. Each worker thread first identifies a rectangular subrange of the iteration space covered by the WITH-loop. This is done using one of several interchangeable work distribution schemes. After that, worker threads compute and initialize pairwise disjoint parts of the result array. Facilities to repeatedly assign subarrays to the same worker thread, as indicated by the dashed arrows in Fig. 3, allow for including dynamic load balancing schemes. Having completed its individual share of work each thread runs upon a final synchronization barrier and terminates. With all worker threads being finished, the master thread resumes control and continues execution of the remaining program again in a sequential manner.

The compilation scheme sketched out so far directly leads to a classical fork/join execution model, as illustrated in Fig. 4a. Unfortunately, the repeated creation and termination of a possibly large number of threads during the execution of a program turns out to be too costly to be efficient in practice. Therefore, a more complex enhanced fork/join execution model, as shown in Fig. 4b, is used instead. Without sacrificing the simplicity of the simple fork/join model on a conceptual level, all worker threads are created right

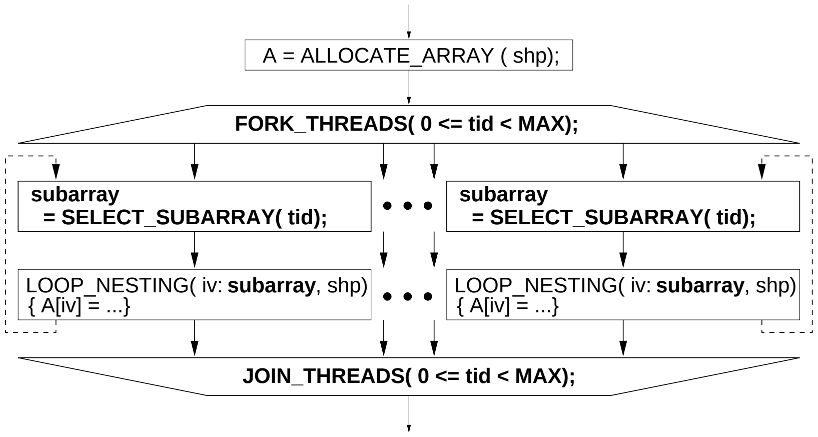
A = ALLOCATE_ARRAY (shp);
FORK_THREADS(0 <= tid < MAX);
subarray = SELECT_SUBARRAY(tid);
LOOP_NESTING(iv: subarray, shp) { A[iv] = ...}
JOIN_THREADS(0 <= tid < MAX);

Figure 3. Compiling WITH-loops into multithreaded code.

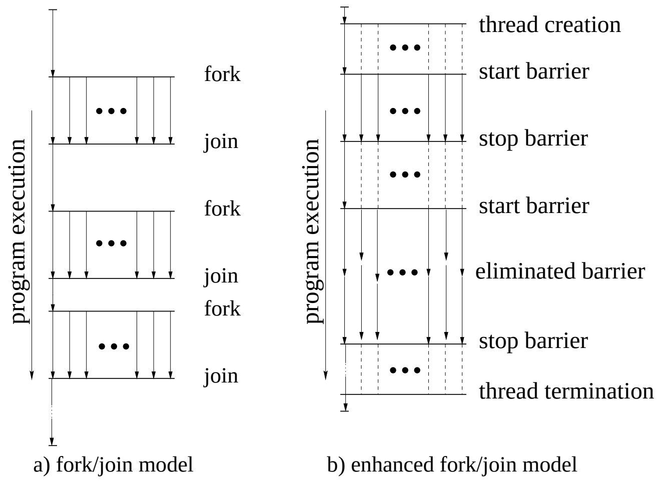

Figure 4. Multithreaded execution models.

after program execution starts and stay alive until the entire program terminates. Intermediate thread creations and terminations are replaced by tailor-made start and stop barriers. They are implemented solely by exploiting properties of underlying hardware cache coherence protocols and, hence, are fairly efficient.

However, even the most efficient barrier construct constitutes runtime overhead. Therefore, it is desirable to completely eliminate barriers between consecutive WITH-loops wherever data dependencies allow. This does not only save the time associated with executing the barrier code itself, but workload imbalances between consecutive WITH-loops are likely to compensate each other, as indicated in Fig. 4b. Here, the high-level functional approach pays off. With a semantics that completely avoids side-effects exact data dependencies can be identified with only modest effort.

With such a fine-tuned compilation scheme and runtime system reasonable speedups should be achievable for typical compound operations on arrays of suitable size. In fact, simple experiments back this assumption. For example, Fig. 5 shows speedups achieved by the multithreaded computation of the sum of all elements of a reasonably sized matrix relative to code compiled for sequential execution from the same source. The test system here as well as in all other experiments described in this paper is a 12-processor SUN Ultra Enterprise 4000 running SOLARIS-7.

4 Dynamic Memory Management

The performance data achieved by the parallel execution of the `sum` operation, as shown in Fig. 5, demonstrate that compilation scheme and runtime system support described in the previous section are well-suited to reach the goal of reasonable performance improvements with zero additional effort on the programmer's side. Unfortunately this observation does not hold for the general case. Fig. 6 shows

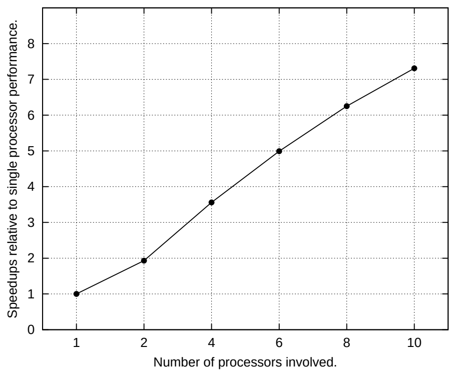

Figure 5. Speedup for computing sum of array elements.

"speedup" values achieved by a very similar operation: instead of summing up all elements of a matrix, they are only added row-wise yielding a vector of results. With a matrix of the same size as in the first experiment, this operation incurs almost the same number of machine instructions; parallel execution is just as trivial as in the first case. However, the observed performance gains achieved by parallel execution are nothing but disappointing: with every additional processor used the absolute program runtime grows.

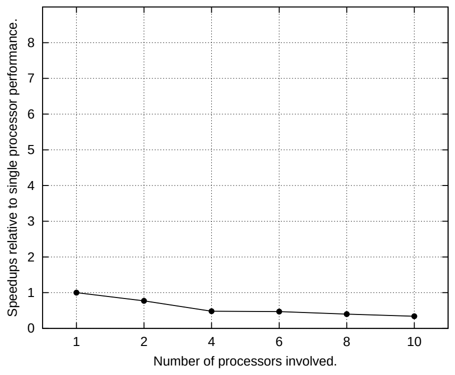

Figure 6. Speedup for computing row-wise sum of array elements.

So, the question is what is the fundamental difference between these two operations which are so similar at first glance, yet yield so different performance figures? Having a look at the two implementations reveals that the latter uses two nested WITH-loops, whereas the former does with a single one. Knowledge of their compilation into host machine code indicates that nested WITH-loops result in memory allocation/de-allocation operations during the multithreaded execution of the outer WITH-loop. In contrast, the first version of `sum` completely avoids memory management during multithreaded execution. These observations seem to be very SAC-specific at first. However, it is quite reasonable to assume that applications which are more complex and hence more realistic than the two versions of `sum` addressed here, are very likely to incur substantial memory management operations during multithreaded execution.

In SAC, as in other high-level languages, memory management for compound data structures like arrays is performed implicitly by the compiler and the runtime system. For this purpose, SAC uses a so-called *reference counting* scheme. Whenever memory for the creation of an array is allocated, it is associated with an additional reference counter which keeps track of the number of active references to this array. Special reference counting operations inserted by the compiler increment and decrement this counter during the lifetime of the array to keep track of varying numbers of references. As soon as the last reference becomes obsolete, all memory associated with the array — including its reference counter — is de-allocated.

Although this scheme incurs some additional overhead at runtime for managing reference counters, its advantages over alternatives, e.g. various forms of garbage collection, are persuasive in the context of arrays. First, precious memory can be reclaimed as soon as possible. Second, and even more important, any array with only a single remaining reference can be subject to a destructively implemented array operation without compromising the functional semantics.

Going back to the experiments which led to the diverging speedup graphs shown in Fig. 5 and in Fig. 6, it turns out that not reference counting itself can be made responsible for the observations. In fact, it is the allocation and de-allocation of memory (triggered by the reference counting mechanism) which causes the problems. In a multithreaded environment all threads share the same heap. However, the execution model, shown in Fig. 4, ensures that threads always write to pairwise disjoint memory locations while read operations to identical locations do not harm.

Unfortunately, the same is not true for operations modifying the internal structure of the heap, as any call to `malloc` or `free` does. In order to guarantee the integrity of internal heap data structures in the presence of multiple concurrent threads, access must be made single-threaded by means of critical regions. This is done by implementors of standard memory allocators to achieve correct behaviour in multithreaded environments. Unfortunately, this may serialize parallel execution through the back door though at the expense of severe runtime overhead. The associated performance impact can be observed in Fig. 6.

This frustrating experience led to the design and implementation of a new memory allocator tailor-made for the multithreaded runtime system of SAC and tightly integrated into it. Its basic organization is characterized by a hierarchy of multiple nested heaps, as sketched out in Fig. 7. At

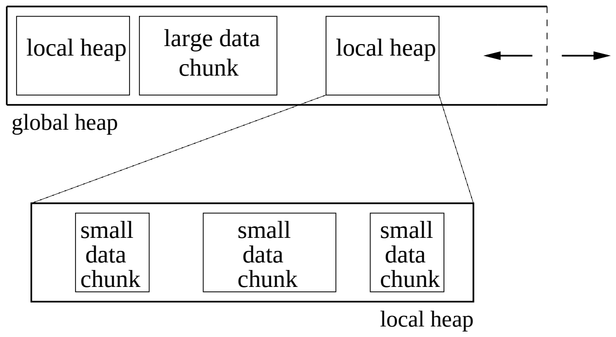

Figure 7. Organization of SAC heap manager.

the top of the hierarchy is a single *global heap*, which controls the entire address space of the process. It may grow or shrink during program execution, as additional memory is requested from the operating system or unused memory is released to it. Nevertheless, the global heap always represents a contiguous memory address space.

However, only relatively large chunks of memory are directly allocated from the global heap. Memory requests below some threshold size are satisfied from one of possibly several *local heaps*. A local heap is a contiguous piece of memory with a fixed size, which in turn is allocated from the global heap. Even more important: each thread is associated with its individual local heap(s).

This organization addresses both scalability and false sharing. On the one hand, each thread may allocate and de-allocate arrays of up to a certain size without any interaction with other threads. On the other hand, small amounts of memory are guaranteed to be allocated from different parts of the address space if requested by different threads. Furthermore, housekeeping data structures for maintaining local heaps are kept separate by different threads. This allows to keep them in processor-specific cache memories without invalidation by the cache coherency mechanism.

Access to the global heap still needs protection by means of synchronization primitives. However, their impact on runtime performance and scalability is negligible primarily for two reasons. On the one hand, expensive locking may actually be avoided in many cases because execution is known to be in single-threaded mode, anyways (cf. Fig. 4). On the other hand, runtime overhead inflicted by requests for large amounts of memory is amortized over the concurrent initialization of relatively many array elements.

Unfortunately, the concept of local heaps is not without drawbacks either. Since any thread must allocate at least one entire local heap regardless of its actual memory demands (which might be much less), memory fragmentation is increased. While this concept is clearly inappropriate for general-purpose allocators, the additional memory fragmentation seems to be acceptable in the context of SAC because the number of threads is limited by the number of processors available and, thus, remains rather small. Moreover, the data parallel approach taken by SAC definitely rules out certain allocation/de-allocation patterns which must be addressed by general-purpose allocators, e.g. the producer/consumer pattern, where some threads predominantly allocate memory whereas others predominantly de-allocate memory previously allocated by the former ones. In fact, the multithreaded execution model described in Section 3 guarantees that any memory allocated by one thread will be de-allocated by the same thread.

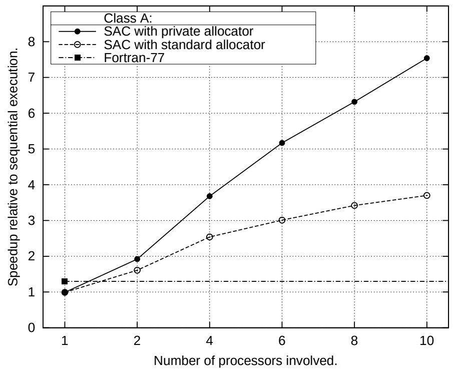

Figure 8. NAS benchmark MG.

Experiments involving more realistic code, i.e. SAC implementations of the NAS benchmarks MG and FT whose performance graphs are shwon in Fig.8 and in Fig.9, respectively, back the decision of implementing a specific memory allocator. They also demonstrate the huge impact of memory management overhead on runtime performance in the context of high-level programming environments.

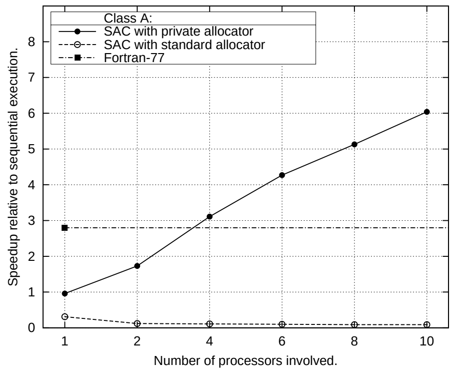

Figure 9. NAS benchmark FT.

Still, the question remains why didn't we simply take an off-the-shelf multithreaded memory allocator? The answer basically is twofold. The simple part is that when we first discovered the problem, suitable general-purpose multithreaded memory allocators were simply not available. This situation has changed meanwhile [3, 35]. Nevertheless, and this is the second part of the answer, a dedicated memory allocator may exploit specific knowledge both concerning general properties of the execution model as well as any other information dynamically available to the runtime system. Furthermore, the closer integration of the memory allocator into the runtime system allows to exploit compile time information on the code, e.g. memory chunk sizes, within the allocation and de-allocation routines.

5 Cache Utilization

It is well-known that data locality is a key issue for achieving high performance in multiprocessor environments [5, 21, 24]. On shared memory architectures efficient utilization of processor-specific hierarchies of cache memories is absolutely crucial. Any failure to satisfy memory requests from one of the cache memories does not only incur considerably slower main memory accesses. Compared to uniprocessor systems, the performance penalty associated with a main memory access typically is significantly higher. Depending on the concrete memory interface design data may have to be fetched from a remote node in physically distributed shared memory systems. In general, memory access times suffer from a much more complex hardware logic. Moreover, in one way or another, all processors compete for limited bandwidth resources. Hence, failure to efficiently exploit processor-specific cache memories usually results in very limited scalability and poor overall performance.

In order to illustrate and quantify the performance impact of varying degrees of cache utilization, we have made some experiments with a typical numerical application kernel: relaxation on a 3-dimensional grid with fixed boundary conditions. The same (compiled) SAC code is applied to grids of different size; more precisely, the cubic grids involved are varied from 16^3 to 528^3 elements in steps of 16 elements in each dimension. With double precision floating point numbers, this involves array sizes between 32KB and 1.1GB. The experiments are again performed on a SUN Ultra Enterprise 4000 system; their results are shown in Fig. 10. Startup overhead, which turns out to be significant for larger problem sizes, is eliminated from these figures by running each problem size with two different numbers of iterations. Then, the difference in iterations as well as the difference in measured execution times are taken to derive the average time needed to re-compute a single inner grid element. This guarantees to compare the runtime performance achieved with extremely different problem sizes on a reasonably fair basis.

It can be observed that the times required to re-compute a single inner grid element considerably vary among the problem sizes investigated. While 120nsec are sufficient to update an inner element of a grid of size 16^3, it takes up to 838nsec to complete the same operation in a grid of size 256^3 or 512^3. Although the same sequence of instructions is executed regardless of the problem size, the time required to do so varies by a factor of about 7 among the problem sizes investigated. Even when ignoring the three extremely poorly performing problem sizes 64^3, 256^3, and 512^3, a rather monotonous increase in program execution times can be observed. This alone accounts for a factor of 2.25 between the smallest problem size 16^3 and the largest one, i.e. 528^3.

Various optimization techniques have been developed in the context of Fortran-77 which aim at improving cache utilization [1, 28, 27]. Basically, they fall into one of two categories. Either the iteration order in loop nestings is manipulated or the memory layout. Examples for the former category are the unimodular transformations or tiling; examples for the latter kind are internal and external padding [25, 26]. Unfortunately, these optimization techniques are not without problems. Their application as well as certain parameters like tile or pad sizes have to be chosen very carefully. Inappropriate selections may significantly slow down program execution.

However, even before dealing with these considerations compilers must prove that iteration reordering or memory layout manipulations do not affect the meaning of the code. In low-level scalar languages, which are characterized by guaranteeing certain iteration orders and data representations, this prerequisite often prevents successful optimizations. At this point the design of high-level approaches clearly pays off. For example, SAC makes no assumptions on the representation of arrays in memory. Similarly, WITH-loops represent complex multi-dimensional iteration spaces without any explicit order. Moreover, a functional, side-effect free semantics simplifies the exact identification of array access patterns.

Exploiting these conceptual advantages, the SAC compiler addresses cache issues in four different ways. First, code generation for WITH-loops aims at achieving unit-stride memory accesses wherever feasible. Considering the potential complexity of WITH-loops, this turned out to be a challenging task [12]. Nevertheless, iteration space tiling is known to yield even better performance figures. Therefore, tiled code may be generated for WITH-loops based on a tailor-made tile size inference heuristics [9].

With respect to memory data layout, arrays in SAC are implicitly padded internally whenever another compiler heuristics considers padding suitable [8, 10]. In contrast, external padding as in Fortran-77 is no option for a high-level array programming environment. Whereas Fortran-77 provides a static overall memory layout, high-

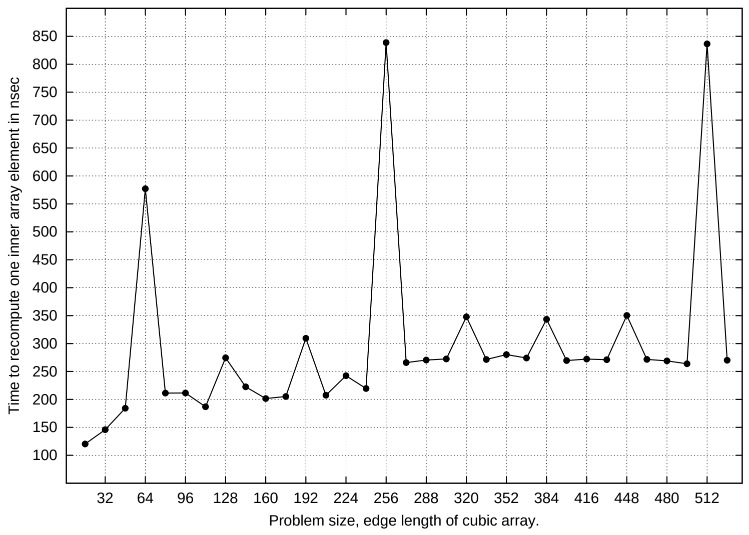

Figure 10. Performance impact of cache memories.

level approaches are typically characterized by highly dynamic memory requirements, which render global memory layout planning unfeasible. As a remedy, SAC applies so-called *array placement* [9]. Whenever a new array is computed based on existing ones, its newly allocated memory base address is carefully chosen to avoid cache conflicts at least in this operation.

In conjunction, these optimizations turn out to be quite successful in improving the average cache utilization and, hence, the overall runtime performance of SAC programs. Repeating the initial experiments with cache optimizations enabled yields the performance data shown in Fig. 11.

6 Conclusion

This paper reports on the experiences made by parallelizing the functional array language SAC. Instead of going the paved path of compiling SAC programs to MPI, PTHREADS was chosen as compilation target. This decision excludes distributed memory architectures for the time being, but it also allows to directly support shared memory system without the detour of a distributed memory programming model. As the commitment to shared memory architectures renders explicit data decomposition obsolete and allows to reuse the memory data layout used for sequential execution, the compilation effort is significantly reduced.

For similar reasons shared memory parallelization is often considered to be less challenging compared to the development of message passing backends. Indeed, the time to create the first successfully parallelizing compiler version was relatively short. However, achieving the desired speedups in program execution turned out to be a much more challenging task. A highly-tuned runtime system which avoids superfluous synchronizations and uses very efficient means for the remaining ones proved to be insufficient to yield reasonable performance figures in many cases.

Two problems characteristic to shared memory architectures had to be solved first. Whenever multiple threads of control concurrently issue memory management requests, they have to be satisfied from the same shared heap. With traditional memory allocators this results in severe performance degradation because access to internal heap data structures must be single-threaded to ensure their integrity, hence serializing program execution through the back door. Only the development of a specialized memory allocator, which is integrated into the multithreaded runtime system, allowed to achieve satisfying runtime performance values.

In shared memory systems, processor private cache memories represent the only “local” source of data, i.e. data which can be read though not even written without interaction with other processors. As a consequence, their efficient utilization is even more important for scalability and overall performance on shared memory systems as it is on distributed memory architectures. Several optimization techniques to this effect had to be incorporated into the compiler without which runtime performance often remains poor.

After all, the essence of this project is that PTHREADS is well-suited as a compilation target for shared memory architectures. However, achieving good speedups is about as challenging as with message passing. Although coming up with an initial solution is considerably more straightforward, it takes a lot more than that to actually succeed with respect to the runtime performance achieved.

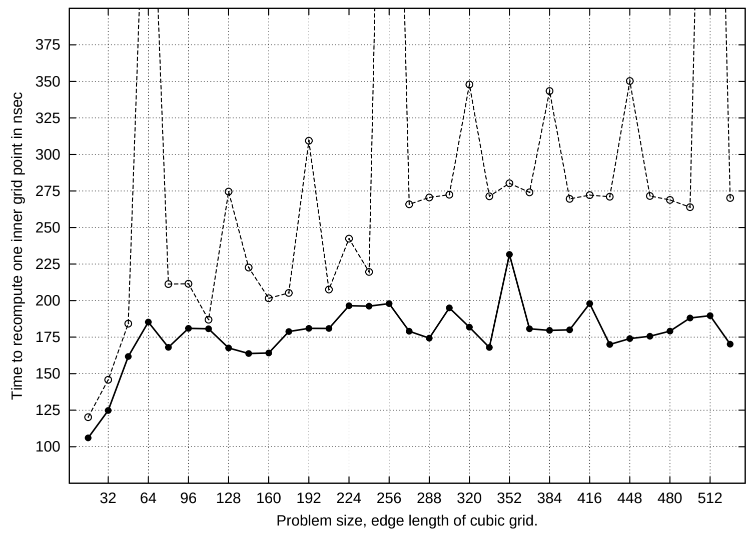

Figure 11. Performance impact of cache Optimizations.

References

[1] U. Banerjee. Unimodular Transformations of Double Loops. In *Proceedings of the 3rd International Workshop on Advances in Languages and Compilers for Parallel Processing (PCPC'90), Irvine, California, USA*, pages 192–219, 1990.

[2] P. Barth, R. Nikhil, and Arvind. M-Structures: Extending a Parallel, Non-Strict Functional Language with State. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA'91), Cambridge, Massachusetts, USA*, volume 523 of *Lecture Notes in Computer Science*, pages 538–568. Springer-Verlag, Berlin, Germany, 1991.

[3] E. Berger, K. McKinley, R. Blumofe, and P. Wilson. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX), Cambridge, Massachusetts, USA*, volume 35 of *SIGPLAN Notices*, pages 117–128. ACM Press, 2000.

[4] A. Böhm, D. Cann, R. Oldehoeft, and J. Feo. SISAL Reference Manual Language Version 2.0. CS 91-118, Colorado State University, Fort Collins, Colorado, USA, 1991.

[5] W. Bolosky, M. Scott, R. Fitzgerald, R. Fowler, and A. Cox. NUMA Policies and their Relation to Memory Architecture. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV), Palo Alto, California, USA*, volume 26 of *SIGPLAN Notices*, pages 212–221. ACM Press, 1991.

[6] D. Cann. Compilation Techniques for High Performance Applicative Computation. Technical Report CS-89-108, Lawrence Livermore National Laboratory, Livermore, California, USA, 1989.

[7] S. Fitzgerald and R. Oldehoeft. Update-in-place Analysis for True Multidimensional Arrays. In A. Böhm and J. Feo, editors, *Proceedings of the Conference on High Performance Functional Computing (HPFC'95), Denver, Colorado, USA*, pages 105–118. Lawrence Livermore National Laboratory, Livermore, California, USA, 1995.

[8] C. Grelck. Array Padding in the Functional Language SAC. In H. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'00), Las Vegas, Nevada, USA*, volume 5, pages 2553–2560. CSREA Press, 2000.

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

[10] C. Grelck. Improving Cache Effectiveness through Array Data Layout in SAC. In M. Mohnen and P. Koopman, editors, *Proceedings of the 12th International Workshop on Implementation of Functional Languages (IFL'00), Aachen, Germany, selected papers*, volume 2011 of *Lecture Notes in Computer Science*, pages 231–248. Springer-Verlag, Berlin, Germany, 2001.

[11] C. Grelck. Implementing the NAS Benchmark MG in SAC. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS'02), Fort Lauderdale, Florida, USA*. IEEE Computer Society Press, 2002.

[12] C. Grelck, D. Kreye, and S.-B. Scholz. On Code Generation for Multi-Generator WITH-Loops in SAC. In P. Koopman and C. Clack, editors, *Proceedings of the 11th International Workshop on Implementation of Functional Languages (IFL'99), Lochem, The Netherlands, selected papers*, volume 1868 of *Lecture Notes in Computer Science*, pages 77–94. Springer-Verlag, Berlin, Germany, 2000.

[13] 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.

[14] 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.

[15] J. Hammes, S. Sur, and W. Böhm. On the Effectiveness of Functional Language Features: NAS Benchmark FT. *Journal of Functional Programming*, 7(1):103–123, 1997.

[16] P. Hartel et al. Benchmarking Implementations of Functional Languages with "Pseudoknot", a Float-Intensive Benchmark. *Journal of Functional Programming*, 6(4), 1996.

[17] Institute of Electrical and Electronic Engineers, Inc. Information Technology — Portable Operating Systems Interface (POSIX) — Part: System Application Program Interface (API) — Amendment 2: Threads Extension [C Language]. IEEE Standard 1003.1c–1995, IEEE, New York City, New York, USA, 1995. also ISO/IEC 9945-1:1990b.

[18] M. Jenkins and W. Jenkins. *The Q'Nial Language and Reference Manual*. Nial Systems Ltd., Ottawa, Canada, 1993.

[19] S. P. Jones and P. Wadler. Imperative Functional Programming. In *Proceedings of the 20th ACM Symposium on Principles of Programming Languages (POPL'93), Charleston, South Carolina, USA*, pages 71–84. ACM Press, 1993.

[20] J. Launchbury and S. P. Jones. Lazy Functional State Threads. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'94), Orlando, Florida, USA*, volume 29 of *SIGPLAN Notices*, pages 24–35. ACM Press, 1994.

[21] E. Markatos and T. LeBlanc. Load Balancing versus Locality Management in Shared-Memory Multiprocessors. In *Proceedings of the 21st International Conference on Parallel Processing (ICPP'92), St. Charles, Illinois, USA*, 1992.

[22] J. McGraw, S. Skedzielewski, S. Allan, R. Oldehoeft, et al. Sisal: Streams and Iteration in a Single Assignment Language: Reference Manual Version 1.2. M 146, Lawrence Livermore National Laboratory, Livermore, California, USA, 1985.

[23] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts, USA, 1990.

[24] T. Mowry, M. Lam, and A. Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V), Boston, Massachusetts, USA*, volume 27 of *SIGPLAN Notices*, pages 62–73. ACM Press, 1992.

[25] P. Panda, H. Nakamura, N. Dutt, and A.Nicolau. A Data Alignment Technique for Improving Cache Performance. In *Proceedings of the International Conference on Computer Design: VLSI in Computers and Processors (ICCD'95), Austin, Texas, USA*, pages 587–592. IEEE Computer Society Press, 1997.

[26] G. Rivera and C.-W. Tseng. Data Transformations for Eliminating Conflict Misses. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98), Montréal, Canada*, volume 33 of *ACM SIGPLAN Notices*, pages 38–49. ACM Press, 1998.

[27] R. Saavedra, W. Mao, D. Park, J. Chame, and S. Moon. The Combined Effectiveness of Unimodular Transformations, Tiling, and Software Prefetching. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS'96), Honolulu, Hawaii, USA*, pages 39–45, 1996.

[28] V. Sarkar and R. Thekkath. A General Framework for Iteration-Reordering Loop Transformations. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'92), San Francisco, California, USA*, volume 27 of *SIGPLAN Notices*, pages 175–187. ACM Press, 1992.

[29] S.-B. Scholz. On Programming Scientific Applications in SAC - A Functional Language Extended by a Subsystem for High-Level Array Operations. In W. Kluge, editor, *Proceedings of the 8th International Workshop on Implementation of Functional Languages (IFL'96), Bonn, Germany, selected papers*, volume 1268 of *Lecture Notes in Computer Science*, pages 85–104. Springer-Verlag, Berlin, Germany, 1997.

[30] 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.

[31] S.-B. Scholz. With-loop-folding in SAC — Condensing Consecutive Array Operations. In C. Clack, T. Davie, and K. Hammond, editors, *Proceedings of the 9th International Workshop on Implementation of Functional Languages (IFL'97), St. Andrews, Scotland, UK, selected papers*, volume 1467 of *Lecture Notes in Computer Science*, pages 72–92. Springer-Verlag, Berlin, Germany, 1998.

[32] S.-B. Scholz. A Case Study: Effects of WITH-Loop Folding on the NAS Benchmark MG in SAC. In K. Hammond, T. Davie, and C. Clack, editors, *Proceedings of the 10th International Workshop on Implementation of Functional Languages (IFL'98), London, UK, selected papers*, volume 1595 of *Lecture Notes in Computer Science*, pages 216–228. Springer-Verlag, Berlin, Germany, 1999.

[33] P. Serrarens. Implementing the Conjugate Gradient Algorithm in a Functional Language. In W. Kluge, editor, *Proceedings of the 8th International Workshop on Implementation of Functional Languages (IFL'96), Bonn, Germany, selected papers*, volume 1268 of *Lecture Notes in Computer Science*, pages 125–140. Springer-Verlag, Berlin, Germany, 1997.

[34] S. Smetsers, E. Barendsen, M. van Eekelen, and M. Plasmeijer. Guaranteeing Safe Destructive Updates through a Type System with Uniqueness Information for Graphs. Technical report, University of Nijmegen, Nijmegen, The Netherlands, 1993.

[35] Sun Microsystems Inc. A Comparison of Memory Allocators in Multiprocessors. Solaris Developer Connection, Sun Microsystems Inc., Mountain View, California, USA, 2000.

[36] P. Wadler. Comprehending Monads. *Mathematical Structures in Computer Science*, 2(4), 1992.