Shared Memory Multiprocessor Support for SAC

Clemens Grelck

University of Kiel
Dept. of Computer Science and Applied Mathematics
D-24105 Kiel, Germany
e-mail: cg@informatik.uni-kiel.de

Abstract. SAC (Single Assignment C) is a strict, purely functional programming language primarily designed with numerical applications in mind. Particular emphasis is on efficient support for arrays both in terms of language expressiveness and in terms of runtime performance. Array operations in SAC are based on elementwise specifications using so-called WITH-loops. These language constructs are also well-suited for concurrent execution on multiprocessor systems.
This paper outlines an implicit approach to compile SAC programs for multi-threaded execution on shared memory architectures. Besides the basic compilation scheme, a brief overview of the runtime system is given. Finally, preliminary performance figures demonstrate that this approach is well-suited to achieve almost linear speedups.

1 Introduction

SAC (Single Assignment C) is a strict, first-order, purely functional programming language primarily designed with numerical applications in mind. Particular emphasis is on efficient support for array processing. Efficiency concerns are essentially twofold. On the one hand, SAC offers the opportunity of defining array operations on a high level of abstraction, including dimension-invariant program specifications which generally improves productivity in program development. On the other hand, sophisticated compilation schemes ensure efficiency in program execution. Extensive performance evaluations on a single though important kernel application (3-dimensional multigrid relaxation from the NAS benchmark [5]) show that SAC clearly outperforms its functional rival SISAL[17] both in terms of memory consumption and in terms of wallclock execution times[23]. Even the FORTRAN reference implementation of this benchmark is outperformed by about 10% with respect to execution times.

Although numerical computations represent just one application domain, certainly, this is a very important one with many applications in computational sciences. In these fields, the runtime performance of programs is the most crucial issue. However, numerical applications are often well-suited for non-sequential program execution. On the one hand, underlying algorithms expose a considerable amount of concurrency; on the other hand, the computational complexity

can be scaled easily with the computational power available. So, multiprocessor systems allow substantial reductions of application runtimes, and, consequently, computational sciences represent a major field of application for parallel processing. Therefore, sufficient support for concurrent program execution is particularly important for a language like SAC.

Due to the Church-Rosser-Property, purely functional languages are often considered well-suited for implicit non-sequential program execution, i.e., the language implementation is solely responsible for exploiting concurrency in multiprocessor environments. However, it turns out that determining where concurrent execution actually outweighs the administrative overhead inflicted by communication and synchronization is nearly as difficult as detecting where concurrent program execution is possible in imperative languages [25]. Many high-level features found in popular functional languages like HASKELL or CLEAN, e.g. higher-order functions, polymorphism, or lazy evaluation, make the necessary program analysis even harder.

As a consequence, recent developments are often in favour of explicit solutions for exploiting concurrency. Special language constructs allow application programmers to specify explicitly how programs are to be executed on multiple processors. Many different approaches have been proposed that reach from simple parallel map operations to full process management capabilities and even pure coordination languages [26, 19, 4, 9, 12, 16]. Although the actual degree of control varies significantly, explicit solutions have in common that, in the end, application programmers themselves are responsible for the efficient utilization of multiprocessor facilities. Programs have to be designed specifically for the execution in multiprocessor environments and, depending on the level of abstraction, possibly even for particular architectures or concrete machine configurations.

However, typical SAC applications spend most of their execution time in array operations. In contrast to load distribution on the level of function applications, elementwise defined array operations are a source of concurrency that is rather well-suited for implicit exploitation, as an array's size and structure can usually be determined in advance, often even at compile time. This allows for effective load distribution and balancing schemes. Implicit solutions for parallel program execution offer well-known advantages: being not polluted with explicit specifications, a program's source code is usually shorter, more concise, and easier to read and understand. Also, programming productivity is generally higher since no characteristics of potential target machines have to be taken into account which also improves program portability. Functional languages like SISAL[17], NESL[6], or ID[3], have already demonstrated that, following the so-called data parallel approach, good speedups may well be achieved without explicit specifications [11, 7, 13].

Successfully reducing application runtimes through non-sequential program execution makes it necessary to consider at least basic design characteristics of intended target hardware architectures. Having a look at recent developments in this area, two trends can be identified. Up to a modest number of processing facilities (usually ≤ 32) symmetric shared memory multiprocessors dominate. If

a decidedly larger number of processing sites is to be used, the trend is towards networks of entire workstations or even personal computers. Both approaches are characterized by reusing standard components for high performance computing which not only is more cost-effective than traditional supercomputers but also benefits from an apparently higher annual performance increase.

In our current approach for SAC, we focus on shared memory multiprocessors. Machines like the Sun Ultra Enterprise Series, the HP 3000/9000 series, or the SGI Origin have become wide-spread as workgroup or enterprise servers and already dominate the lower part of the Top500 list of the most powerful computing facilities worldwide [10]. Although their scalability is conceptually limited by the memory bottleneck, processor private hierarchies of fast and sufficiently large caches help to minimize contention on the main memory. Theoretical considerations like Amdahl's law [2], however, show that an application itself may be limited with respect to scalability anyway. Our current approach may also serve as a first step to be integrated into a more comprehensive solution covering networks of shared memory multiprocessors in the future.

As a low-level programming model, multi-threading just seems to be tailor-made for shared memory architectures. It allows for different (sequential) threads of control within the single address space of a process. Each thread has its private execution stack, but all threads share access to the same global data. This programming model exactly coincides with the hardware architecture of shared memory multiprocessors which is characterized by multiple execution facilities but uniform storage. To ensure portability between different concrete machines within the basic architectural model, the current implementation is based on POSIX-THREADS[18] as the major standard.

The rest of the paper is organized as follows: after a short introduction to SAC in Section 2, the basic concepts of our shared memory multiprocessor implementation are outlined in Section 3. Preliminary performance figures are presented in Section 4. Finally, Section 5 draws conclusions and discusses future work.

2 SAC — Single Assignment C

This section is to give a very brief overview of SAC. A more detailed introduction to the language may be found in [21, 24]; its strict, purely functional semantics is formally defined in [20].

The core language of SAC may be considered a functional subset of C, ruling out global variables and pointers to keep the language free of side effects. It is extended by the introduction of arrays as first class objects. An array is represented by two vectors: a *data vector* which contains the elements of the array, and a *shape vector* which provides structural information. The length of the shape vector specifies the dimensionality of the array whereas its elements define the array's extension in each dimension. Built-in functions allow to determine an array's dimension or shape and to extract array elements.

Complex array operations may be specified by means of so-called WITH-loops, a versatile language construct similar to the array comprehensions of

HASKELL or CLEAN and to the FOR-loops of SISAL. It allows the dimension-invariant, elementwise definition of operations on entire arrays as well as on subarrays selected through index ranges or strides.

$$
\begin{array}{lll}
\mathit{WithExpr} & \Rightarrow & \textbf{with} \;\; (\;\; \mathit{Generator} \;\;) \;\; \mathit{Operation} \\
\mathit{Generator} & \Rightarrow & \mathit{Expr}\;\; \mathit{Relop}\;\; \mathit{Identifier}\;\; \mathit{Relop}\;\; \mathit{Expr}\;\; [\;\; \mathit{Filter} \;\;] \\
\mathit{Relop} & \Rightarrow & < \;\; | \;\; <= \\
\mathit{Filter} & \Rightarrow & \textbf{step}\;\; \mathit{Expr}\;\; [\;\; \textbf{width}\;\; \mathit{Expr} \;\;] \\
\mathit{Operation} & \Rightarrow & \textbf{genarray} \;\; (\;\; \mathit{Expr} \;\; , \;\; \mathit{Expr} \;\;) \\
 & | & \textbf{modarray} \;\; (\;\; \mathit{Expr} \;\; , \;\; \mathit{Expr} \;\; , \;\; \mathit{Expr} \;\;) \\
 & | & \textbf{fold} \;\; (\;\; \mathit{FoldFun} \;\; , \;\; \mathit{Expr} \;\; , \;\; \mathit{Expr} \;\;)
\end{array}
$$

Fig. 1. The syntax of WITH-loops.

The syntax of WITH-loops is outlined in Fig. 1. A WITH-loop consists of two parts: a *generator part* and an *operation part*. The generator part defines a set of index vectors along with an index variable representing elements of this set. Two expressions that must evaluate to vectors of equal length, define the lower and the upper bounds of a range of index vectors. This continuous range may be restricted by a filter which defines strides of arbitrary widths. For instance, with a, b, s, and w denoting expressions that evaluate to vectors of length n,
`( a <= i_vec < b step s width w )` specifies the set of index vectors
$\{i\_vec \mid \forall_{i \in \{0,\ldots,n-1\}} : a_i \leq i\_vec_i < b_i \wedge (i\_vec_i - a_i) \text{ modulo } s_i < w_i\}$.
The operation part specifies the operation to be performed on each element of the index vector set defined by the generator. Three different operation parts exist. Let *shp* and *idx* denote SAC-expressions that evaluate to vectors, let *array* denote a SAC-expression that evaluates to an array, and let *expr* denote an arbitrary SAC-expression. Moreover, let *fold_op* be the name of a binary commutative and associative function with neutral element *neutral*. Then

- `genarray(` *shp*`,` *expr*`)` generates an array of shape *shp* whose elements are the values of *expr* for all index vectors from the specified set, and 0 otherwise;
- `modarray(` *array*`,` *idx*`,` *expr*`)` defines an array of shape `shape(`*array*`)` whose elements are the values of *expr* for all index vectors from the specified set, and the values of *array*`[`*idx*`]` at all other index positions;
- `fold(` *fold_op*`,` *neutral*`,` *expr*`)` allows the specification of reduction operations. Setting out with *neutral*, for each index vector from the specified set the value of *expr* is folded using *fold_op*.

The expressive power of the WITH-loop allows the specification of a comprehensive array library for SAC in the language itself. This library provides numerous dimension and shape independent high-level array operations similar

to those available in APL[15] or FORTRAN-90[1] as intrinsic functions, e.g. extensions of binary scalar operations to combinations of scalars and arrays as well as to arrays of equal shape by elementwise application, various types of subarray selection, concatenation of arrays along given axes, shifting and rotating arrays, or the reduction operations sum, product, any, and all. Since this library can easily be extended by any application programmer, SAC allows high-level programming without the restriction of a fixed set of built-in operations.

3 Implementation aspects

This section introduces the basic concepts of extending the SAC compiler in order to generate multi-threaded target code based on POSIX-THREADS. This thread API provides operations to dynamically create new threads and to synchronize them upon termination. As threads communicate with each other by means of global data, various synchronization primitives are available to ensure data integrity in the presence of simultaneous accesses by different threads. While on a uniprocessor, these are simply executed in a time-sharing mode, on a shared memory multiprocessor, the operating system scheduler may assign them to different processors for simultaneous execution. Thread scheduling is performed implicitly by the operating system; there is no means to explicitly assign threads to specific processors for execution.

For reasons already pointed out, concurrency in SAC program specifications is not to be exploited on the level of function applications but within elementwise defined array operations. Here, the design of arrays in SAC pays off. Since all high-level array operations are implemented by WITH-loops in SAC itself, we can focus entirely on this single though powerful language construct. Consequently, without any extra effort, the operations provided by the SAC array library benefit from multi-threaded execution just as any user-defined array operation.

```
A = with ( lb <= iv < ub step s width w )
    genarray( shp, e );
  ...
B = with ( lb <= iv < ub step s width w )
    modarray( A, iv, A[iv] + 1);
  ...
c = with ( lb <= iv < ub step s width w )
    fold( foldfun, neutral, B[iv]);
```

Fig. 2. SAC code example.

The compilation of WITH-loops into multi-threaded (imperative) pseudo code is outlined by means of a small example. The SAC code fragment in Fig. 2 features all three variants of the WITH-loop as introduced in Section 2. The variables `lb`, `ub`, `s`, and `w` that make up the generator parts as well as `shp` are assumed to be defined before the statements shown and to evaluate to vectors

of equal length. For reasons of simplicity, the same variable names are used in all three generator parts, however, their actual values may be different. First, an array `A` of shape `shp` is generated by means of a `genarray`-WITH-loop. The variable `e` is also assumed to be defined before and to evaluate to a scalar, say `int`. Next, a `modarray`-WITH-loop defines an array `B` identical to `A` except for the elements selected by the generator, which are incremented by 1. Finally, a `fold`-WITH-loop is used to fold selected elements of array `B` by the operation `foldfun` whose neutral element `neutral` is assumed to denote a constant.

```
A = ALLOCATE_ARRAY( shp);
LOOP_NESTING( iv: shape(A), lb, ub, s, w) {
  A[iv] = ? e : 0;
}
...
B = ALLOCATE_ARRAY( shape(A));
LOOP_NESTING( iv: shape(B), lb, ub, s, w) {
  B[iv] = ? A[i]+1 : A[i];
}
...
c = neutral;
LOOP_NESTING( iv: lb, ub, s, w) {
  c = foldfun( c, B[iv]);
}
```

Fig. 3. Compilation to sequential code[1].

As a starting point, the compilation of this example code fragment into sequential (imperative) pseudo code is outlined in Fig.3. After memory for the target array is allocated, all its elements are initialized in a nesting of (`for`-) loops either with the value of `e` or with `0`. The loop nesting defines a complete iteration of the variable `iv` on the target array; the concrete design however depends on `lb`, `ub`, `s`, and `w`. On this level of abstraction, the `genarray` and the `modarray` variants of the WITH-loop turn out to be identical, i.e., `modarray`-WITH-loops can be ignored from now on. The implementation of the `fold`-WITH-loop is slightly different. It starts with the initialization of the fold variable `c` with the neutral element of the fold operation. The loop nesting lets `iv` only iterate within the iteration space actually defined by `lb`, `ub`, `s`, and `w`. In each iteration step, the value of `c` is updated by folding its old value with the respective element of array `B`.

With this sequential implementation in mind, the basic idea of organizing the multi-threaded execution of a WITH-loop is straightforward. The corresponding iteration space has to be partioned into several disjoint subspaces, one for each thread. In the case of the `genarray` and the `modarray` variant, each thread then

[1] Here `A[iv] = ? e : 0;` denotes that in different parts of the loop nesting the operation is either `A[iv] = e;` or `A[iv] = 0;`.

simply initializes a disjoint part of the target array. In the case of a `fold`-WITH-loop, each thread computes a partial fold result. Afterwards, these partial results are again folded to form the overall result.

```
A = ALLOCATE_ARRAY( shp);
MT_EXECUTION( 0 <= tid < #THREADS) {
  do {
    sb, se, cont = SCHEDULE( tid, #THREADS, shape(A), lb, ub, s, w);
    LOOP_NESTING( iv: sb, se, shape(A), lb, ub, s, w) {
       A[iv] = ? e : 0;
    }
  } while (cont);
}
```

Fig. 4. Multi-threaded implementation of the `genarray`-WITH-loop.

The multi-threaded implementation of the `genarray`-WITH-loop of the example is outlined in Fig.4. The pseudo statement `MT_EXECUTION` denotes that the following code block is to be executed concurrently by multiple threads. The exact number of threads is specified by `#THREADS` which is considered a runtime constant. Although each thread executes the same code, threads can identify themselves by means of the variable `tid` whose value in the range `[0..#THREADS-1]` is unique for each thread.

In the presence of subranges and strides of different widths in multiple dimensions, the actual nesting of loops can be extremely complicated. An optimization called WITH-loop-folding[22] that allows for condensing several subsequent WITH-loops into a single, more powerful variant increases this complexity even further. For reasons of efficiency in compiler design, it is therefore highly recommendable to reuse the existing sequential compilation scheme for WITH-loops as far as possible. The solution here is to completely separate from the computation, i.e. from the loop nesting, the decision which thread actually initializes which array elements. In Fig. 4, this decision-making code is denoted by the pseudo statement `SCHEDULE` as this discipline is usually called loop scheduling.

The idea is that the loop scheduler defines a rectangular subrange of the original iteration space covered by the loop nesting, based on the total number of threads (`#THREADS`) and the thread ID (`tid`). This rectangular subspace is defined by the two vectors `sb` ('schedule begin') and `se` ('schedule end'). The original (sequential) loop nesting is only slightly modified in that each loop is restricted to the intersection between its original range and the iteration subspace defined by `sb` and `se`. Apart from reusing existing compilation schemes, strictly separating the scheduling from the computation offers the additional advantage that different scheduling strategies may easily be implemented and tested, and later on the compiler may choose the one which is most appropriate with respect to the overall array operation. Enclosing the scheduler and the loop nesting within a (`do`-) loop allows scheduler implementations that repeatedly assign different iteration subspaces to one thread. The scheduling code itself de-

cides whether or not a re-scheduling is required and stores this information using the local variable `cont`.

The basic organizational concepts of a multi-threaded implementation of WITH-loops as outlined in the context of the `genarray` variant may also be be applied to `modarray`- and `fold`-WITH-loops in a more or less straightforward way. Instead of going into more details, we now focus on the aspect of organizing a whole program with respect to multiple execution threads. This concerns such issues as where and how to create and terminate additional threads, thread synchronization, and inter-thread communication.

As a result of the compilation steps described so far, all WITH-loops from the original SAC program are replaced by `MT_EXECUTION` blocks. These blocks exactly indicate the code sections that actually are to be executed concurrently by multiple threads. This leads straightforwardly to a fork/join execution model as depicted on the left hand side of Fig. 5. The primary thread of an application process serves as a master thread (thread ID 0). Upon program startup, the master thread begins executing the program sequentially. Each time the master thread encounters an `MT_EXECUTION` block, it creates `#THREADS` - 1 so-called worker threads. Afterwards, the master thread and the worker threads jointly execute the `MT_EXECUTION` block as described before. Upon completing their computation, worker threads simply terminate. The master thread, however, has to wait until the last worker thread terminates, and thereupon continues with the execution of sequential code.

This fork/join model is conceptually simple and may be implemented straightforwardly. Synchronization and communication is exactly limited to thread creation and thread termination; the worker threads do not interact with each other in any way. However, in a concrete implementation, the performance achieved by a pure fork/join model turns out to be rather poor. Sufficient speedups may only be achieved for extremely large problem sizes or with extremely costly operations per element. The reason for this is that although thread creation is relatively cheap compared to process creation, it is still expensive in terms of machine instructions. So, creating new worker threads upon each multi-threaded WITH-loop-execution and terminating them afterwards is inefficient. [2]

A solution to this problem that combines the conceptual benefits of the fork/join approach with an efficient execution scheme is graphically outlined in the centre of Fig. 5. In the enhanced fork/join model, all worker threads are created exactly once at program startup and do not terminate until the whole program does so. The necessary synchronization and communication between the threads is implemented by means of two different types of barriers: each `MT_EXECUTION` block is enclosed within a *start barrier* and a *stop barrier*. After creation, worker threads immediately stop at a start barrier. This barrier is lifted when the master thread encounters the first `MT_EXECUTION` block. The master thread and all worker threads activated thereupon share the computation of the WITH-loop exactly as in the pure fork/join model. Worker threads which com-

[2] On one of our test machines, we measured $> 10,000$ clock cycles for creating just one (kernel) thread.

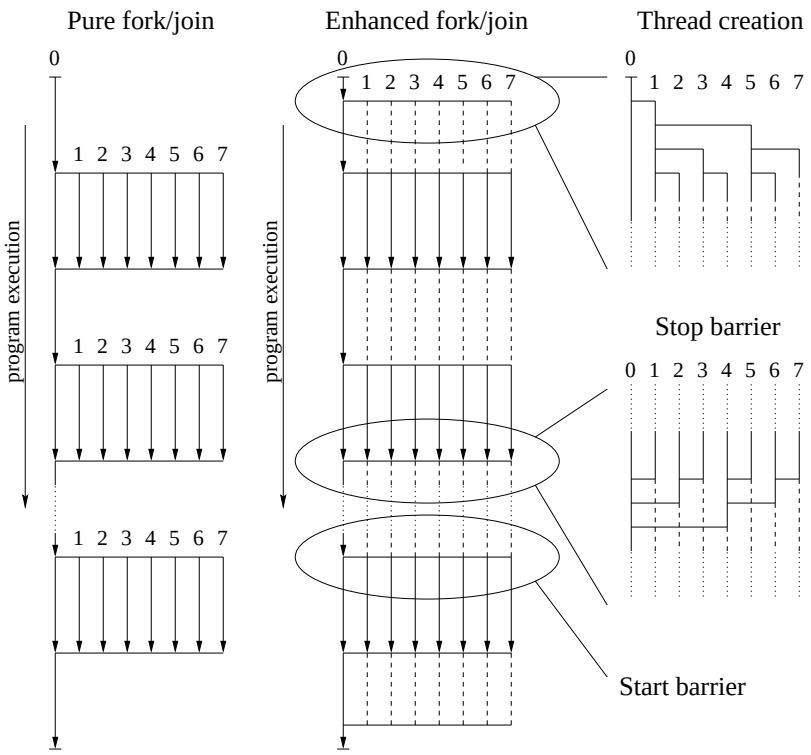

Fig. 5. Multi-threaded execution schemes.

plete their individual part of the computation, pass a stop barrier, and, with nothing else to do, immediately move on to the following start barrier. However, the master thread has to wait for the last worker thread to reach the stop barrier before it may proceed with further (sequential) computations.

Two major extensions to the compilation scheme described so far are required in order to implement this enhanced fork/join execution model. First, a function has to be specified that is executed by the worker threads upon creation, in the following called thread control function; second, the code within `MT_EXECUTION` blocks has to be abstracted out of its original context and lifted to a separate function definition in order to be accessible from the thread control function. These new functions are named WL-functions.

The thread control function is outlined in Fig. 6. It shows how the worker threads reach the start barrier immediately after creation. Before the master thread lifts this barrier, it stores the address of the WL-function to be executed in the global variable `WL_FUN_ADDR`. Upon activation, each worker thread retrieves this address and executes the respective function with its own unique thread ID as argument. Afterwards the worker threads stop again at the start barrier waiting for further activations.

```
void ThreadControl( int tid)
{
  wl_fun_t *wl_fun;
  do {
    START_BARRIER_WORKER();
    wl_fun = WL_FUN_ADDR;
    *wl_fun( tid);
  } forever;
}
```

Fig. 6. Thread control function.

If a block of code is to be abstracted out of its original context, it must first be transformed into a combinator. For this purpose, two sets of variables have to be inferred: the set IN of all variables referenced within the block but defined outside and the set OUT of the variables assigned a value within the block that is needed outside. To actually generate a new function definition, the set LOC of all identifiers exclusively used within the block is also required. For the `MT_EXECUTION` block outlined in Fig.4, these sets can easily be identified as

IN = { `A`, `e`, `lb`, `ub`, `s`, `w`}, OUT = ∅, LOC = { `iv`, `sb`, `se`, `cont`},

and for the `fold`-WITH-loop introduced with the initial example in Fig. 2 as

IN = { `B`, `lb`, `ub`, `s`, `w`}, OUT = { `c` }, LOC = { `iv`, `sb`, `se`, `cont`}.

With these sets of identifiers at hand, it is rather straightforward to construct a function definition and to replace the original code block by the respective function application. However, in our case, we have to observe that WL-functions are restricted in their signature since they have to be called from within the thread control function in a uniform way (see Fig. 6). As a consequence, an alternative parameter passing mechanism is required. The complete solutions for the `genarray`-WITH-loop of our example is outlined in Figs. 7 and 9.

At the original position of the `genarray`-WITH-loop, the `MT_EXECUTION` block is replaced by code which stores the value of each variable from the corresponding IN set within the global argument frame `ARG_FRAME` (Fig. 9). Afterwards, the address of the respective WL-function which actually contains the code to be executed concurrently, is stored in the global variable `WL_FUN_ADDR`. The master thread now activates the worker threads by reaching the start barrier and subsequently joins them in executing the WITH-loop through an ordinary call to the respective WL-function with its special thread ID `0` as argument.

In the following, all threads execute the same function (`WL_FUN_1`, Fig.7). This function definition has a local declaration for each variable from the corresponding IN, OUT, and LOC sets. Before any computations are done, the values of the IN variables, i.e. the 'arguments' of the WL-function, are retrieved from the global argument frame `ARG_FRAME`. The WL-function also contains the stop barrier. So, after returning from the application of a WL-function, the master thread may simply proceed with further (sequential) computations (Fig. 9).

Only minor extensions of this scheme are required for `fold`-WITH-loops as depicted in Figs. 8 and 10. The OUT variable `c` which is used to accumulate

```
void WL_FUN_1(int tid)
{
  int A[]  = ARG_FRAME.WL1.A;
  int e    = ARG_FRAME.WL1.e;
  int lb[] = ARG_FRAME.WL1.lb;
  ...
  int iv[], sb[], se[], cont;
  do {
    ...
  } while (cont);
  STOP_BARRIER(tid);
}
```

Fig. 7. WL-function: genarray.

```
int WL_FUN_3(int tid)
{
  int B[] = ARG_FRAME.WL3.B;
  ...
  int c, iv[], sb[], se[], cont;
  c = neutral;
  do {
    ...
  } while (cont);
  STOP_BARRIER_F(tid, foldfun, c);
  return(c)
}
```

Fig. 8. WL-function: fold.

```
A = ALLOCATE_ARRAY( shp);
ARG_FRAME.WL1.A  = A;
ARG_FRAME.WL1.e  = e;
  ...
WL_FUN_ADDR = &WL_FUN_1;
START_BARRIER_MASTER();
WL_FUN_1( 0);
```

Fig. 9. WL-context: genarray.

```
ARG_FRAME.WL3.B  = B;
ARG_FRAME.WL3.lb = lb;
ARG_FRAME.WL3.ub = ub;
  ...
WL_FUN_ADDR = &WL_FUN_3;
START_BARRIER_MASTER();
c = WL_FUN_3( 0);
```

Fig. 10. WL-context: fold.

the partial fold result private to each thread is also declared a local variable. However, a special variant of the stop barrier is required that takes care of folding the partial results of the various threads, i.e., behind the stop barrier, `c` represents the overall fold result which then is simply returned by the WL-function. The master thread may directly use this value for further computations while the worker threads just ignore the return value of the WL-function (Fig. 6).

Some issues of particular interest have not been addressed yet: the thread creation phase and the implementation of start and stop barriers. Since these represent the administrative overhead of a multi-threaded program, their efficient implementation is crucial to achieve good speedups.

In a straightforward implementation of the thread creation phase, the master thread starts all worker threads one after another by means of a `for`-loop. As a consequence, the execution of the actually productive code is delayed by a time that grows linearly with the number of threads. This delay can easily be reduced if the worker threads participate in thread creation. This leads to a tree-like creation scheme which reduces this initial delay to a factor of $\lceil log_2$ #`THREADS`$\rceil$. However, the initial delay factor may be further reduced to only 1 by excluding the master thread from the thread creation scheme as outlined on the upper right hand side of Fig. 5. The master thread creates exactly one worker thread and then immediately starts with the execution of the actual program. The first worker thread subsequently creates the other worker threads following a binary tree scheme. In this way, the administrative overhead due to thread creation overlaps

with the execution of a program's (sequential) startup phase, e.g. reading input data from files.

The combination of a stop barrier and a subsequent start barrier represents a full barrier synchronization which is known to scale poorly with the number of threads [14] and, therefore, is a major cause of overhead. However, scalability can be improved by organizing the barrier as a tree-like structure of pairwise synchronizations, as depicted on the lower right hand side of Fig.5. Threads with an odd ID simply pass the stop barrier, immediately stopping at the following 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. This concurrent synchronization scheme allows the master thread (thread ID 0) to synchronize itself with all worker threads in only $\lceil log_2 \ \#\texttt{THREADS} \rceil$ steps.

In the case of a `fold`-WITH-loop, the stop barrier is also responsible for folding the partial results of the single threads to form the overall result. Each time a thread synchronizes itself with another thread, it folds its own intermediate result with that of the other thread. This scheme is further improved by allowing threads which synchronize with several other threads to do so in any order. Then, a thread may already execute final fold operations while still waiting for other threads to complete their partial result. As in the thread creation phase, administrative overhead again overlaps with productive computation.

4 Preliminary performance evaluation

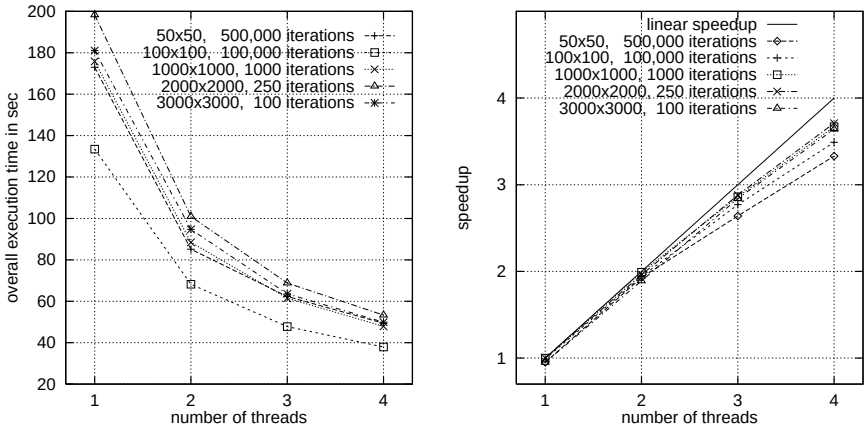

Fig. 11. 2-D Jacobi relaxation on 4-proc. Sun Ultra Enterprise 3000.

Preliminary performance tests of the current implementation described in the previous section have been made on two different machines: a Sun Ultra Enterprise 3000 with 4 processors and 512MB of memory and a Sun Ultra Enterprise 4000 featuring 12 processors and 7.5GB of memory. Both are running Solaris,

versions 2.5.1 and 2.6, respectively. A simplified variant of 2-dimensional Jacobi relaxation [8] served as a benchmark kernel. Test runs for various problem sizes have been made with up to 4 threads on the Enterprise 3000 and with up to 12 threads on the Enterprise 4000. Overall execution times achieved on the two machines are depicted in Figs. 11 and 12. The respective speedups relative to a program which from exactly the same SAC source code has been compiled for sequential execution are shown in Figs. 11 and 13.

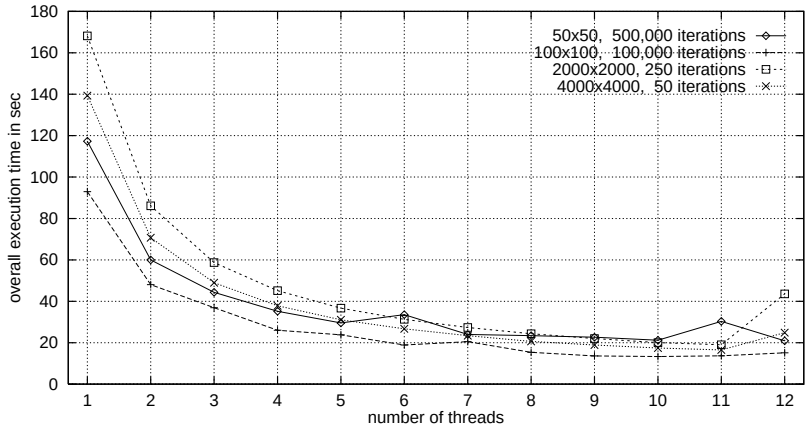

Fig. 12. 2-D Jacobi relaxation on 12-proc. Sun Ultra Enterprise 4000: execution times.

As the figures demonstrate, multi-threaded execution of the benchmark kernel yields substantial reductions in overall runtimes on both machines and for all problem sizes investigated. Speedups reach up to 3.71 on the 4-processor system and up to 8.83 on the 12-processor system. Considerable speedups are achieved even for relatively small problem sizes of only 100×100 or 50×50 array elements although they require very frequent synchronization among threads. Only for unfavourable combinations of array size and number of threads, speedups decrease due to load imbalances resulting from the simple loop scheduling mechanism.

It is important to note that multi-threading per sé produces nothing but overhead. Only when it comes to program execution on a multiprocessor, multi-threading enables the operating system scheduler to assign different threads to different processors for execution. As a consequence, speedups due to multi-threading can only be expected if different threads of an application actually run on different processors. However, the way the underlying operating system distributes runnable threads among the available processors on a given machine cannot be influenced by the application itself. Still, it is obvious that no additional speedup can be expected if the number of threads exceeds the number of available processors.

However, if exclusive access to a machine cannot be guaranteed as it is the case with the machines used for benchmarking, the execution of an application's threads may interfere with other user and system processes. As soon as the total

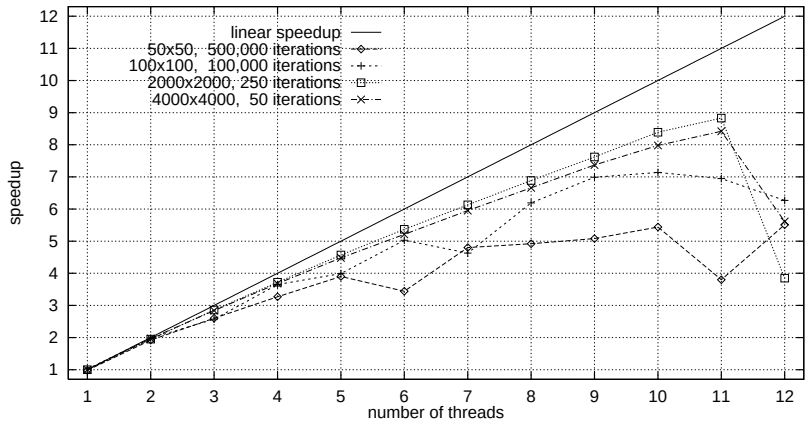

Fig. 13. 2-D Jacobi relaxation on 12-proc. Sun Ultra Enterprise 4000: speedups.

number of runnable threads in a system exceeds the number of processors available, the operating system scheduler is forced to assign several threads to the same processor for execution. In this case, a loop scheduler that statically assigns work to threads, like the one in our implementation, causes severe load imbalance, leading to a performance degradation. This is exactly what can be observed with the performance figures for the two larger test cases on the 12-processor machine. When these measurements were done, exactly one other process was constantly running in the system. Hence, up to 11 threads, execution time and speedup figures scale well, but drop dramatically if 12 threads are used.

5 Conclusions and future work

SAC is a programming language primarily designed with numerical applications in mind. A powerful language construct called WITH-loop allows the specification of high-level array operations independent of the operands' dimensionalities and shapes. Operations are defined elementwise on entire arrays or on subarrays selected by index ranges or strides. Despite the high level of abstraction in program specifications, sophisticated compilation schemes allow the transformation of WITH-loops into efficiently executable (sequential) code [21,23].

The elementwise specification of operations on (sufficiently large) arrays exposes a high amount of fine-grained concurrency. This paper describes a completely implicit approach to exploit this concurrency to speed up program execution on shared memory multiprocessors. A compilation scheme which transforms WITH-loops into multi-threaded target code is outlined along with the required runtime system. By completely separating the loop scheduling facility from the loops themselves, the existing sequential compilation scheme of WITH-loops can largely be reused. Moreover, this provides the opportunity to easily exchange the loop scheduling implementation in order to adjust load balancing strategies to the program structure or target system properties. An execution model for

multi-threaded programs is presented that overcomes the limitations of a simple fork/join oriented approach. Instead of repeatedly creating and terminating threads, they are created exactly once upon program startup while all synchronization is realized by a tailor-made variant of barrier synchronization.

Preliminary performance evaluations of our current implementation are made on two Sun Ultra Enterprise systems with 4 and 12 processors. A simplified version of 2-dimensional Jacobi relaxation is used as a benchmark kernel. Performance figures for various problem sizes demonstrate that even for relatively small problems substantial speedups are achieved on both systems reaching up to 3.71 or 8.83, respectively.

Future work will focus on reducing the negative performance impact of the synchronization barriers which complete each concurrently executed code segment. Since the barrier implementation itself is already highly optimized, the emphasis will be on improving the load balancing capabilities of the loop scheduler in order to cope with variations in computational complexity for different elements of the target array as well as with threads belonging to other processes on systems not used exclusively. An alternative approach is to identify larger sections of code that can be executed concurrently without intermediate synchronization, e.g., synchronization barriers between consecutive WITH-loops can be eliminated as far as there is no data dependence between them.

References

1. J.C. Adams, W.S. Brainerd, J.T. Martin, et al. *Fortran90 Handbook - Complete ANSI/ISO Reference*. McGraw-Hill, 1992. ISBN 0-07-000406-4.
2. G.M. Amdahl. Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. In *AFIPS Conf. Proc.*, pages 483–485. AFIPS Press, Reston, Va, 1967.
3. Arvind, K.P. Gostelow, and W. Plouffe. The ID-Report: An asynchronous Programming Language and Computing Machine. Technical Report 114, University of California at Irvine, 1978.
4. C. Aßmann. Coordinating Functional Processes Using Petri Nets. In Werner Kluge, editor, *Implementation of Functional Languages, 8th International Workshop, Bad Godesberg, Germany, September 1996, Selected Papers*, volume 1268 of *LNCS*, pages 162–183. Springer, 1997.
5. D. Bailey, E. Barszcz, J. Barton, et al. The NAS Parallel Benchmarks. RNR 94-007, NASA Ames Research Center, 1994.
6. G.E. Blelloch. NESL: A Nested Data-Parallel Language (Version 3.1). Technical Report CMU-CS-95-170, Carnegie Mellon University, Pittsburgh, PA, 1995.
7. G.E. Blelloch, S.Chatterjee, J.C. Hardwick, J. Sipelstein, and M.Zagha. Implementation of a Portable Nested Data-Parallel Language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, 1994.
8. D. Braess. *Finite Elemente*. Springer, 1996. ISBN 3-540-61905-4.
9. S. Breitinger, R. Loogen, and Y. Ortega-Mallen. Towards a Declarative Language for Parallel and Concurrent Programming. In *Proc. Glasgow Workshop on Functional Programming 1995*. Springer, Workshops in Computing, 1995.
10. J.J. Dongarra, H.W. Meuer, and E. Strohmaier. TOP500 Supercomputer Sites, 11th edition. In *Supercomputer '98 Conference, Mannheim, Germany*, 1998.

11. M. Haines and W. Böhm. Task Management, Virtual Shared Memory, and Multithreading in a Distributed Memory Implementation of SISAL. In A. Bode et al., editors, *PARLE '93*, volume 694 of *LNCS*, pages 12–23. Springer, 1993.
12. K. Hammond, H.-W. Loidl, S.L. Peyton Jones, and P. Trinder. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1), 1998.
13. J. Hicks, D. Chiou, B.S. Ang, and Arvind. Performance Studies of Id on the Monsoon Dataflow System. *Journal of Parallel and Distributed Computing*, 18(3):273–300, 1993.
14. J.M.D. Hill and D.B. Skillicorn. Practical Barrier Synchronisation. Technical Report TR-16-96, Programming Research Group, Oxford University Computing Laboratory, Oxford, England, 1996.
15. K.E. Iverson. *A Programming Language*. Wiley, New York, 1962.
16. S.L. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proc. 23rd ACM Symposium on Principles of Programming Languages (POPL'96), St. Petersburg Beach, Florida*, pages 295–308, 1996.
17. J.R. McGraw, S.K. Skedzielewski, S.J. Allan, R.R. Oldehoeft, et al. SISAL: Streams and Iteration in a Single Assignment Language: Reference Manual Version 1.2. M 146, Lawrence Livermore National Laboratory, LLNL, Livermore California, 1985.
18. Institute of Electrical and Inc. Electronic Engineers. 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, NY, 1995.
19. J.H. Reppy. CML: A higher-order concurrent language. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 293–305, 1991.
20. S.B. Scholz. **S***ingle* **A***ssignment* **C** *– Entwurf und Implementierung einer funktionalen C-Variante mit spezieller Unterstützung shape-invarianter Array-Operationen.* PhD thesis, Institut für Informatik und Praktische Mathematik, Universität Kiel, 1996.
21. 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, *Implementation of Functional Languages, 8th International Workshop, Bonn, Germany, September 1996, Selected Papers*, volume 1268 of *LNCS*, pages 85–104. Springer, 1997.
22. S.B. Scholz. WITH-loop-folding: Condensing Consecutive Array Operations. In C. Clack, T. Davie, and K. Hammond, editors, *Proceedings of the 9th International Workshop on the Implementation of Functional Languages '97, St. Andrews*, pages 225–242. University of St. Andrews, Scotland, 1997.
23. S.B. Scholz. A Case Study: Effects of WITH-Loop Folding on the NAS Mgrid Benchmark in SAC. In K. Hammond, T. Davie, and C. Clack, editors, *Proc. of the 10th International Workshop on Implemenatation of Functional Languages (IFL'98, London)*, pages 407–418. University College, London, 1998.
24. 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, *Proc. of the Array Processing Language Conference '98, Rome, Italy*, pages 40–45. ACM Press, 1998.
25. W. Schreiner. Parallel Functional Programming, An Annotated Bibliography (2nd edition). 93-24, Research Institute for Symbolic Computation (RISC), Johannes-Kepler-University, Linz, Austria, 1993.
26. P.R. Serrarens. Distributed arrays in Clean. In *Euro-Par '97*, LNCS. Springer, 1997.