SMARTS: Exploiting Temporal Locality and Parallelism through Vertical Execution

Suvas Vajracharya, Steve Karmesin, Peter Beckman,
James Crotinger, Allen Malony*, Sameer Shende*, Rod Oldehoeft, and Stephen Smith
Advanced Computing Laboratory
Los Alamos National Laboratory
Los Alamos, NM, U.S.A.

Abstract *In the solution of large-scale numerical problems, parallel computing is becoming simultaneously more important and more difficult. The complex organization of today's multiprocessors with several memory hierarchies has forced the scientific programmer to make a choice between simple but unscalable code and scalable but extremely complex code that does not port to other architectures.*

This paper describes how the SMARTS runtime system and the POOMA C++ class library for high-performance scientific computing work together to exploit data parallelism in scientific applications while hiding the details of managing parallelism and data locality from the user. We present innovative algorithms, based on the macro-dataflow model, for detecting data parallelism and efficiently executing data-parallel statements on shared-memory multiprocessors. We also describe how these algorithms can be implemented on clusters of SMPs.

Keywords: Data-parallelism, dependence-driven execution, runtime systems, barrier synchronization, loop scheduling, macro-dataflow, cache reuse, programming models, object-parallelism, data-parallel languages, object-oriented, data locality, scientific computation

1 Introduction

The source of most parallelism in numerical and scientific applications comes from independent loop iterations in data-parallel statements. This paper describes a system for discovering independent loop iterations through run-time dependence analysis and efficiently executing those iterations on multiprocessors with deep memory hierarchies.

Conventional models of data-parallel programming take advantage of *horizontal* parallelism in a stream of data-parallel statements. In horizontal data parallelism, the participating processors apply the same operations to different subsections of the data, one operation at a time. A barrier synchronization ensures that the next data-parallel statement is not started until all the processors are done with the current statement. In contrast, we define *vertical* parallelism as the concurrent execution of multiple data-parallel operations while respecting the data dependencies among those operations. The "vertical" here refers to the vertical direction in the stream of data-parallel statements appearing in a program. By exploiting both vertical and horizontal parallelism, we can reduce the idle time of participating processors waiting at barrier synchronizations. Furthermore, because we are applying multiple operations in a depth-first manner, a vertical execution will reuse cached data more effectively, as illustrated in Fig. 1. If the illustrated arrays do not fit in the machine's cache, horizontal execution forces re-loading the data values at each data-parallel operation. In contrast, vertical execution applies multiple operations to the same data before that data leaves the cache. Compiler optimizations such as loop fusion [6, 13, 27] and loop interchange [27, 1] restructure the source code to achieve the same end. However, these transformations cannot be applied in the presence of more complicated dependencies between the statements, a restriction that we wish to remove by using a macro-dataflow model.

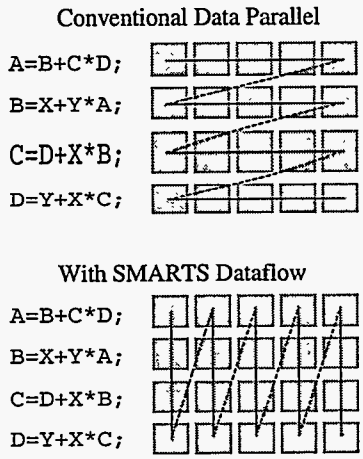

Figure 1: Horizontal vs. Vertical Execution. Each square represents a block of floating point operations.

In earlier papers, it was argued that a vertical execution improves temporal locality [24] and increases parallelism [23]. In this paper, we describe a macro-dataflow approach for *automating* vertical execution by generating and executing a dependence graph representing the dependencies among the data-parallel operations. While we show how these algorithms are general enough to be implemented on clusters of shared-memory multiprocessors (SMPs), the results in this paper are limited to our current experience on a single SMP.

*Dept. of Computer Science, University of Oregon

The remainder of the paper is organized as follows: Sections 3 and 5 describe how SMARTS, the Shared Memory Asynchronous RunTime System, extends the master-slave programming model and the single-program-multiple-data (SPMD) model to permit a vertical execution. Having described the runtime system, Sec. 6 shows how POOMA, a framework for scientific applications, uses the runtime system to encapsulate parallelism and the details of data locality for the application developer. Section 7 then compares the performance of different methods using vertical and horizontal execution on a shared-memory multiprocessor. Our ultimate goal is to run on clusters of SMPs. Section 8 describes how the same algorithm can be implemented to use both shared-memory and message passing, and argues that a vertical execution model is particularly well suited for hiding latency.

2 Related Work

Due to the higher degree of parallelism that asynchronous systems contain, the dataflow concept has appeared in many works. One issue that distinguishes the various works is the unit, or granularity, of the parallelism. A very fine granularity requires a significantly different design than does a coarse-grain computation. For example, dataflow computers such as the Manchester Dataflow Machine [9, 10] relied on special hardware to orchestrate parallelism at the level of individual floating-point arithmetic operations.

Most modern computer architectures do not have such special support hardware, and thus the dataflow concept is often implemented in software at a much coarser grain; i.e. *macro-dataflow*. In the tradition of true dataflow models, much of the work on macro-dataflow approaches, such as the work by Babb [2], Mentat [8, 7], and Cilk [3], has emphasized functional parallelism.

For data-parallel applications, functional parallelism has the disadvantage that the functional decomposition determines the granularity of parallelism. Finding the appropriate decomposition is often a difficult and inappropriate task for data-parallel applications.

In contrast, SMARTS determines the granularity of computation based on the decomposition, or layout, of the arrays, a much easier task because the user separately specifies data and functional definitions. This also has the advantage that the granularity can be determined dynamically, as in our earlier work [23]. A functional decomposition, in contrast, statically fixes the granularity when the functions are defined.

These difficulties have led to other approaches that use entire loops as nodes in dependence graphs, as in the work by Tang [21], Sisal [5], and the autoscheduling work by Moreira and Polychronopoulus [11]. A horizontal loop scheduler, such as guided-self scheduling [16], might then schedule these loops. In SMARTS, iterations within and across loops can be scheduled simultaneously because a node in the generated dependence graph, known as a SMARTS *iterate*, is an iteration subspace, not the entire loop.

We have used granularity to classify related works. Another dimension for comparison is the extent of user involvement in synchronizing the units of computation. Compile-time methods are attractive because they involve no user intervention for optimizations. Strictly compile-time methods [22, 26] restructure the loop and generate new code to improve locality and parallelism.

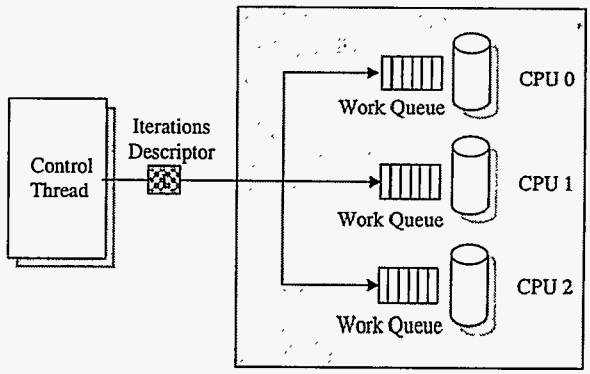

Figure 2: SCHE: Single Control Thread, Horizontal Execution.

For example, loop fusion takes two consecutive data-parallel statements (or loops) and combines them into one, such that each iteration of the loop applies both of the operations. Another transformation, loop interchange, swaps the inner loop of a nested loop with the outer loop to take advantage of better data locality in a vertical execution. While having the desired effect of better cache reuse, these methods are limited in that the compiler can only apply them when the dependencies are simple and apparent.

In general, while compile-time methods have the advantage that they incur little run-time overhead and that the automation frees the user from the details of locality and parallelism, they are limited in that compilers cannot apply many interesting optimizations that depend on the knowledge of dynamic information. Compile-time optimizations cannot be applied to situations where the time it takes to complete an operation varies at run-time, a common case on cache-based and parallel computers. Other methods, such as Cilk and Mentat, require the user to explicitly state the interaction and synchronization between the units of parallelism. SMARTS, used with a data-parallel package like POOMA, frees the application programmer from being concerned with interaction and synchronization, which we believe are the most difficult parts of parallel processing.

3 Master-Slave Work Queue Model

This section describes the master-slave work-queue model and its extension to take advantage of vertical parallelism. In the master-slave model, there is a single thread of control (the master thread) that distributes work to slave threads. In the case of executing data-parallel expressions, the master thread distributes loop iterations to the waiting slave threads.

In its simplest form, the actual number of iterations to be distributed can be described by an iteration descriptor containing the range of loop iterations. Because all threads must be working on a single data-parallel operation at a time, the operation does not need to be specified in each descriptor. A common way to distribute work is to enqueue the descriptors on a work queue, to be dequeued by idle processors.

Affinity scheduling, which has been shown to give favorable performance on NUMA architectures [15], uses per-processor work queues to improve data locality and to avoid contention for a global queue. Each element in the work queue consists of an iteration descriptor and a CPU number that represents the affinity of data

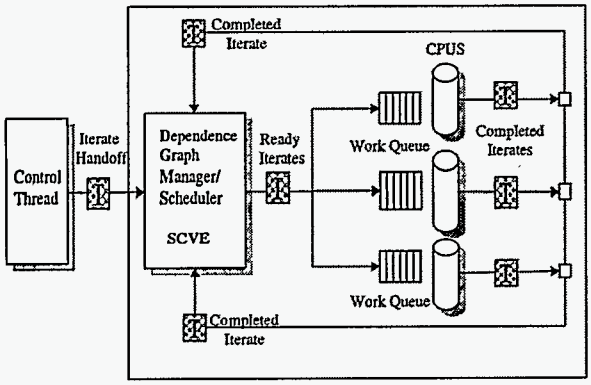

Figure 3: Single Control Thread, Vertical Execution.

```
class Iterate {
  public:
    Iterate(...);
    CPU affinity();
    IterDesc range;
  protected:
    DataObject *read;
    DataObject write;
    void operation();
};
```

```
class DataObject {
  public:
    DataObject(...)
    CPU affinity();
    DataDesc dataRegion;
    bool request(Iterate,Type);
    release();
  private:
    Fifo RequestQueue;
};
```

Figure 4: Iteration and data descriptors in SMARTS.

to that CPU. An idle processor looks for work in its own queue and executes the iterations described by the descriptor. If its local queue is empty, the idle process steals work from a remote queue of another processor. For the purpose of classification, we will call this simple method (shown in Fig. 2) SCHE, Single Control Horizontal Execution. Note that the control thread does not execute the loop iterations but simply describes what to compute. The run-time system, which knows the load distribution on the different CPUs and the affinity hint in the descriptor, decides where to actually execute the iterations.

In SMARTS, we take the idea of deferring execution a step further. Unlike the SCHE model, in which the descriptor is ready to run when the control thread hands it off to the run-time system, a SMARTS iterate is not ready to run until the run-time system has determined that all of the iterate's dependencies have been satisfied. This gives the run-time system another degree of freedom: the freedom to determine an order of operations, within the constraints of the dependencies, that improves cache data reuse and parallelism.

To incorporate vertical parallelism into the work-queue model, we need to add a few extensions. First, because we wish to execute multiple operations simultaneously or in an interleaved fashion, the work queue must be a polymorphic queue of descriptors containing not only the `<iteration range,CPU>` tuple, but also the operation that SMARTS is to perform when dependencies are satisfied. Our implementation uses a modified descriptor that contains `<operation, iteration range, CPU>`. This descriptor is represented by a C++ object that we call a SMARTS *iterate*. When an idle processor finds an iterate in the work queue, it removes it from the work queue and executes it on its domain by calling its (virtual) run method. By associating operations with data and using a polymorphic queue, we can interleave the execution of iterations from different data-parallel statements or schedule multiple loops (as also described in [14, 23, 24]). Because each iterate has all the information necessary to execute, iterates from many different data parallel statements can be in existence and ready to run at the same time.

4 Graph Generation and Execution

Another extension to SCHE is required to ensure that the data dependencies between the data parallel operations are not violated. Figure 3 schematically shows the new model. A SMARTS iterate must not run until earlier iterates that use the same data have finished. Thus, the iterates make reservations with the data to which they require access. These reservations are handled by another C++ class, the SMARTS *data-object*. The data-objects grant reservations in FIFO order, and iterates are eligible for execution when all of their reservations have been granted. When an iterate finishes, it notifies the associated data-objects that it is done, allowing each data-object to grant other reservations. We now discuss how iterate and data-object descriptors interact to build and use the dependence graph.

SMARTS maintains two types of descriptors, iterates representing the *operation* and data-objects representing the *data*.

A data-object is a gatekeeper for each block of data that is potentially shared between different iterates. It receives requests from iterates for read or write access and stores them in a FIFO queue. It grants one write request at a time, or any number of sequential read requests at a time. In our implementations, data-object is a concrete C++ class, `Smarts::DataObject`, that can be used via either inheritance or composition.

Iterates are represented by subclasses of `Smarts::Iterate`. Iterate subclasses represent the work to be done for a specific domain of a specific data parallel expression. The subclass constructor requests appropriate access to the data-objects involved in the subexpression, and those data-objects notify the `Iterate` base class when they are available. When all of the data-objects have signaled that they are available, the iterate becomes runnable and enrolls itself in the proper queue for execution.

Using these classes, execution of data parallel statements proceeds in the following steps.

When data parallel objects are constructed, they are decomposed into blocks, each of which is given a data-object. At construction time the queue of reservations in each data-object is empty. The format of the data and what it represents is determined by the library or language; SMARTS acts only as a gatekeeper.

The master thread later executes statements involving these data parallel objects. Let us call a given statement S_a, which uses J_a data parallel objects, P_{aj}, each of which has N_{aj} data-objects. S_a is evaluated on the domain DS_a.

For example, suppose the first data parallel statement, S_1, is

$$A = B + C * D,$$

where each of A, B, C and D are defined on [1..100] and are each decomposed into five equal pieces. Then $DS_1 = [1..100]$, $J_1 = 4$, and $N_{1j} = 5$ for all j.

To evaluate S_a, the master thread builds a set of K_a SMARTS iterates, I_{ak}, each of which has a domain DI_{ak}, the union of which is DS_a and the intersection of

which is empty. The domains DI_{ak} are chosen so that each of I_{ak} uses exactly one data-object for each of P_{aj}.

In many cases, the decompositions of the data parallel objects are chosen to align. If that is the case, we will have $K_a = N_{aj}$ for all j. That is, the number of iterates will be the same as the number of data-objects in each of the terms in the expression. If the decompositions of the data parallel objects are arbitrarily different, the number of iterates K_a could be as high as $\prod_j N_{aj}$.

The master thread's responsibility for statement S_a is to create K_a iterates, I_{ak}. When constructing each of those iterates, reservations are made with the data-objects that the iterates will use. If all of an iterate's data-objects are available, the iterate goes immediately to a runnable queue. Otherwise, the iterate waits until its data-objects notify it that they are available.

At that point, the master thread's responsibility for statement S_a is done, and it is free to go on to S_{a+1}.

The slave threads' responsibility is to pull iterates out of the runnable queue and execute them. All data dependencies have been satisfied for iterates in the runnable queue, so the slave threads don't need to check dependencies.

After an iterate is executed, it is destroyed by the slave thread, and the destructor releases the reservations on the data-objects that it used. Releasing a reservation pulls it off the front of the data-object's FIFO queue of reservations. If that reservation was the last of a block of read requests, then the following write request will be granted, and if it was a write request, the following single write or block of reads will be granted. If other read requests had been granted and are still outstanding, no new requests are granted.

If new requests were granted, they may have been the last ones required by some iterates. Those iterates would then move to the runnable queue, to be picked up by slave threads.

The overall pattern then is one of the master thread constructing iterates, those iterates making reservations with data-objects, slave threads executing and destroying iterates, and that destruction triggering the release of new iterates for execution. Figures 5 and 6 illustrate this cycle.

Because the dependencies impose a partial order on the execution of the iterates, rather than a full order, there may be many iterates available for execution at any given time. It turns out that there is a simple and effective technique to execute the iterates in an order that optimizes for cache reuse. Right after an iterate finishes, the data that it used is likely to be in cache, and one would like to run other iterates that will also use that data. If, when iterates are moved to the runnable queue, they are placed at the head of that queue, they will very likely need data that has just been used, and thus will achieve cache locality with very low computational overhead.

The programming model and scheduling algorithm that we described above is an extension of the affinity-work-queue scheduling method to include vertical parallelism or out-of-order execution. We will call this extended method SCVE, single control (thread), vertical execution. In the next section, we similarly extended the SPMD model of computation to use vertical execution; that is, we describe MCVE, multiple control vertical execution.

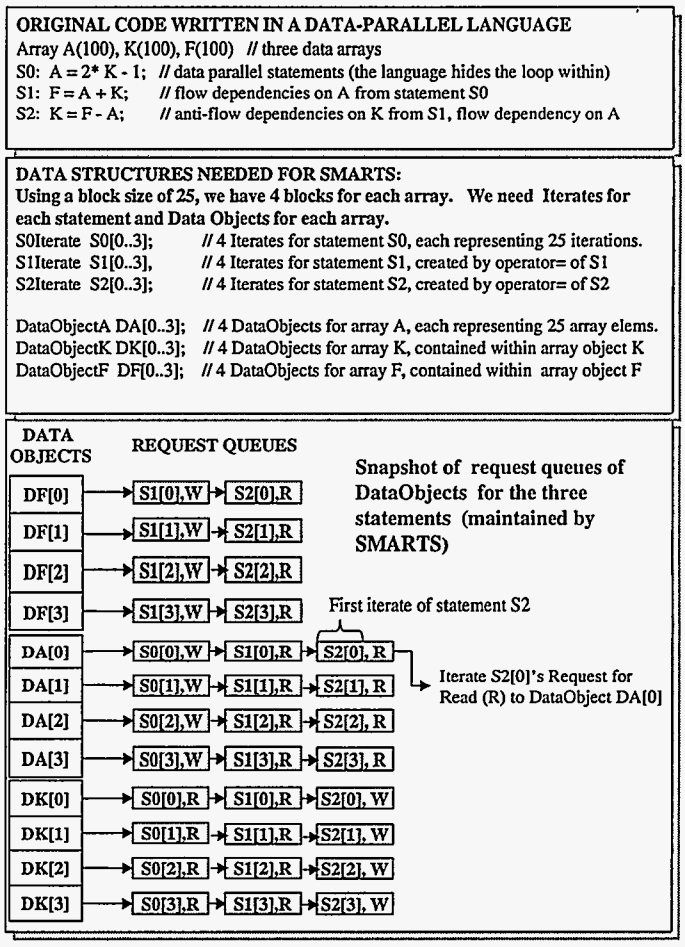

Figure 5: Snapshot of the SMARTS requests queues.

```
class SOIterate : public Iterate {
  public:
    Iterate(int I, GRAIN) {
      begin = I*GRAIN;
      end = (I+1)*GRAIN-1;
      DO_A[I] -> request(this, Write);
      DO_D[I] -> request(this, Read);
    }
    operation() {
      for (int I=begin; I < end; I++)
          A[I] = 2*D[I]-1;
    }
  private:
      int begin;
      int end;
}
DataObject DO_A[0..3], DO_D[0..3];
...
beginGeneration()
  for (I=0; I<4; I++)
    SMARTS::handoff(new SOIterate(I,25));
endGeneration();
```

Figure 6: Code to interface with SMARTS for data-parallel statement, A = 2*D-1.

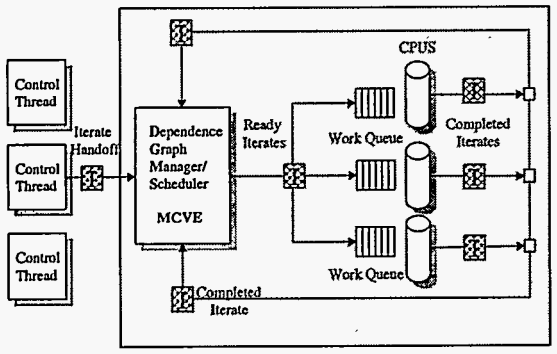

Figure 7: Single Control Thread, Vertical Parallelism.

5 Multiple Control Threads

In the conventional SPMD model of programming, a programmer spawns as many threads as there are processors. Each thread executes the same program, but computes on different sections of arrays and other data structures based on its unique thread identifier. For example, a parallel computation on a one-dimensional array of size N by P threads would involve each thread computing on a block of the array indexed by $(I * \frac{N}{P}..(I+1) * \frac{N}{P})$ where I is the thread identifier. The model is synchronous in the sense that every thread must perform a barrier synchronization after every data parallel statement.

Vertical parallelism can be introduced into this model. As shown in Fig. 7, the back-end remains similar to the back-end for SPVE model, except that the front-end, which generates the graph, needs to synchronize with other control threads. Because the SPVE model has only a single thread of control, there was no possibility of the iterates requesting data-objects out of order, a necessary condition for correct generation of the graph. With multiple control threads, we guarantee that no iterates from subsequent data-parallel statements make reservations with data-objects by keeping the master threads synchronized. Consequently no requests from statement $N+1$ would be honored by the data-object prior to honoring requests from statements $0..N$.

6 Detection of Parallelism Using POOMA

The POOMA (Parallel Object-Oriented Methods and Applications) framework [17] is a C++ class library for high performance scientific computations. POOMA includes data-parallel array and particle classes that hide the details of parallelism from the user. Several large applications currently use POOMA, including a multi-material hydrodynamics code and a particle-in-cell-based linear accelerator modeling code.

The original POOMA implemented parallelism in a lock-step fashion, using message passing. POOMA II, a major redesign aimed at improving flexibility and performance, includes thread-based evaluation and ability to use SMARTS [12]. Switching to the SMARTS data-flow model was a natural choice for several reasons. Most of the POOMA applications are currently run on clusters of SMPs. SMARTS permits a programming paradigm that uses shared memory to share data objects within a box instead of message passing. Furthermore, the data-flow model allows overlapping of communication with computation, hiding communication latency (which can be a significant factor for some of the more communication intensive applications). It is possible to implement latency-tolerant computation with asynchronous message passing, but the SMARTS programming model simplifies this task tremendously. Finally, as this paper demonstrates, SMARTS potentially increases cache reuse, allowing POOMA codes to run faster in both serial and threaded environments.

The details of generating SMARTS iterates that evaluate the data-parallel expressions are completely hidden from the user. POOMA codes are written using high level data parallel statements. For example,

```
#include "Pooma/Arrays.h"

// Jacobi iteration
void main(int argc,char* argv[])
{
  Pooma::initialize(argc,argv);

  int n=100;
  Array<2> V(n,n),b(n,n);

  V = 0.0;
  b = 0.0;
  b(n/2,n/2) = -1.0;

  Interval<1> I(1,n-2),J(1,n-2);

  for (int iter = 0; iter < 100; ++iter)
  {
    V(I,J) = 0.25* ((V(I+1,J) + V(I-1,J) +
             V(I,J+1) + V(I,J-1) - b(I,J));
  }
  cout << V << endl;
  Pooma::finalize();
}
```

Evaluation is performed by the array's assignment operator. Instead of performing the operations as they appear, POOMA uses expression templates [25] to construct expression objects that are passed to the assignment operator. These objects contain references to all arrays and scalars in the expression, and encode the form of the expression in a template argument. Arbitrary transformations can be performed on the expression at compile-time using template meta-programs [4]. Traditionally, expression templates are used to transform statements like `A = B + C` into efficient loops like:

```
for (i=0; i< n; ++i)
  A[i] = B[i] + C[i];
```

POOMA defers evaluation to an evaluator object, which in turn can generate the iterates that are passed to SMARTS. The dependency information is generated using the expression template mechanism to say, "for each array on the right-hand side, request a read lock" and "request a write lock for the array being assigned to."

To decompose problems in parallel, POOMA provides an array type that contains multiple blocks. Each block contains its own SMARTS data-object, so that multiple iterates can simultaneously act on a given array, provided that they act on separate blocks. In the multi-block case, POOMA generates at least one iterate for each block on the left-hand side and generates read requests for all the blocks that are touched on the right-hand side.

7 Experimental Results

This section describes our experience with SMARTS and POOMA on a shared-memory multiprocessor, the SGI Origin 2000. We show sequential speedups from better reuse of cached data, parallel speedup due to the asynchrony of the macro-dataflow model, and performance analysis using TAU [19].

7.1 Sequential Cases

To separate the performance benefits of temporal locality from parallelism, we initially restrict our experiments to sequential cases. First, we considered a trivially parallel example where we applied a logistic map for several iterations to an entire field. In POOMA, this is expressed as

```
for (i = 0; i < n; ++i) {
  b = k * a * (1.0 - a);
  a = b;
}
```

where a and b are POOMA arrays and k is a constant. The SMARTS iterate for a given block in each of the arrays only has dependencies on the corresponding block in the other array. Thus we can execute the entire loop for a given block before moving to the other blocks. Figure 8 compares three cases: The case labeled "serial" was implemented using arrays that contained a single block of memory, and SMARTS was not used. For small problem sizes, this case runs the fastest because there is no additional overhead, but once the arrays no longer fit in level-two cache the performance drops by more than a factor of two. We implemented the "blocked" case using arrays that contained 100 x 100 blocks of data, but again, we did not use SMARTS. Since each block needs control code for a loop, multi-block arrays introduce additional overhead. However, this overhead becomes less significant for larger problem sizes as cache effects become dominant. Finally, the case labeled "blocked+smarts" uses SMARTS to schedule the iterates for arrays with 100 x 100 blocks. SMARTS introduces additional overhead to generate and enforce dependencies, but, as in the "blocked" case, the benefits of cache reuse quickly overshadow this overhead as we increase the problem size. As shown in the figure, we are able to sustain almost a constant MFLOPS as we increase the problem size, leading eventually to a twofold speedup over a naive implementation.

For the previous example, it may be possible for some compilers to recognize that a loop-interchange can generate code that traverses the iteration subspace vertically. However, for applications with more complex data dependencies, this would not be possible. Stencil expressions are examples of such applications. The results in Fig. 9 are for a simple three-point stencil operation performed on a two-dimensional array. The POOMA code for the loop is

```
for (i = 0; i < n; ++i) {
  b(I,J) = c*(a(I+1,J) + a(I,J) + a(I-1,J));
  a = b;
}
```

where a and b are arrays, I and J are POOMA index objects and c is a constant. Using SMARTS for large problems gives superior results, but because of the overhead that SMARTS and POOMA introduce, we do not see a constant MFLOPS rate as we increase the problem size.

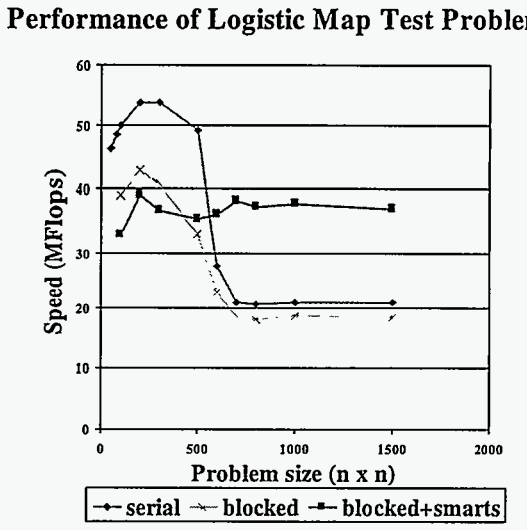

Figure 8: Performance for a trivially parallel example: Applying the logistic map to arrays. SMARTS allow us to get uniform performance, as problems become too large to fit in cache.

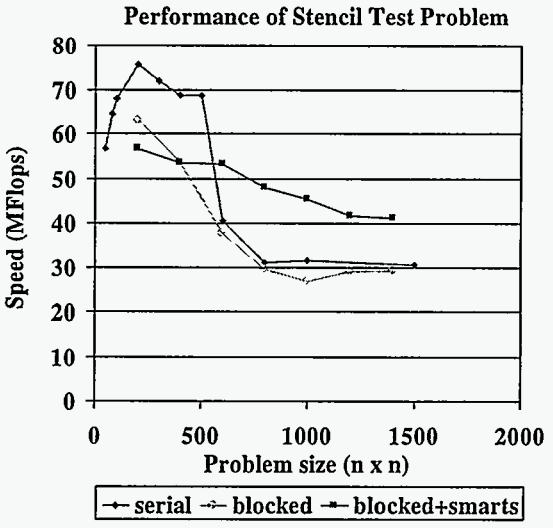

Figure 9: Performance for a simple stencil operation. Even with more complex dependencies, SMARTS provides better performance for large problems.

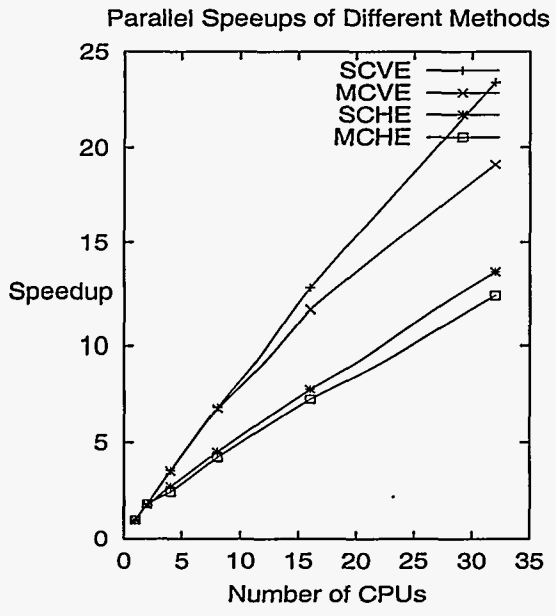

Figure 10: Speedups for different methods for Red/-Black SOR with problem size of 4096×4096.

7.2 Parallel Speedups

To study scaling of applications using SMARTS, we compared the performance of Red/Black SOR with various methods on a 32 processor SGI Origin 2000. We looked at four methods:

- SCHE: (Single control thread, horizontal execution). In our implementation of SCHE, we used affinity scheduling with multiple work queues.
- SCVE: (Single control thread, vertical execution) is currently our fastest implementation of vertical parallelism. It is a macro-dataflow extension of the affinity scheduler.
- MCHE: (Multiple control threads, horizontal execution) is an implementation of the SPMD model. We spawn as many threads as there are processors and each thread is bound to a particular CPU.
- MCVE: (Multiple control threads, vertical execution) is an experimental implementation of vertical execution built by extending the SPMD model. This experimental implementation has not been fine-tuned for performance.

In each of the methods above, we used a Morton order [18] to block-decompose the two-dimensional data matrix. Figure 10 shows the relative performance of the four methods on an otherwise idle machine. Speedups are relative to the best sequential method. To measure the time taken for each run, we took an average of 10 trials. Results were statistically significant with the 95% confidence interval being no greater than ∓ 1.87 seconds with an average of 45.8 seconds on runs with 32 processors.

7.3 Performance Analysis

We used the Tuning and Analysis Utilities (TAU) to analyze the performance of SMARTS. TAU uses timing instrumentation that is triggered at function entry and exit. The instrumentation is responsible for name registration, maintaining the function database,

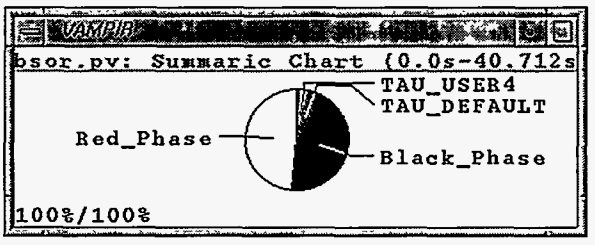

Figure 13: Pie chart of the time spent in Red/Black code segments over all threads. Figure shows a system overhead of 3.49%.

the callstack [20], and statistics. The overhead of this instrumentation is proportional to the frequency of triggers. For a single threaded program running on an SGI Origin 2000, the overhead associated with profiling is 0.8 microseconds for each entry and for each exit of a profiled block of code. A block can be any section of code and commonly consists of a single function or template. TAU can be configured to work with different thread packages such as SMARTS, Tulip, and pthreads. For multiple SMARTS threads, the overhead was 4 microseconds for each entry and for each exit. This is due to the cost associated with getting the thread identifier from the underlying thread system. An additional 8 to 40 microseconds is required for name registration, which happens the first time a profiled block is entered. The instrumentation is kept independent of the symbol table generated by a specific compiler. This makes TAU portable and gives it the flexibility to be used with multiple compilers, with or without optimizations, with multiple runtime systems, multiple operating systems, and multiple languages.

The model that TAU uses to profile parallel, multithreaded C++ programs maintains performance data for each thread, context, and node in use by an application. The profiling instrumentation needed to implement the model captures data for C++ functions, methods, basic blocks, and statement execution at these levels. The instrumentation is complicated, however, by advanced features in the C++ language, such as templates and namespaces. All C++ language features are supported in the TAU profiling instrumentation, which is available through an API at the library or application level. The API also provides selection of profiling groups for organizing and controlling instrumentation. From the profile data collected, TAU's profile analysis procedures can generate a wealth of performance information for the user. It can show the exclusive and inclusive time spent in each function with nanosecond resolution. For templated entities, it shows the breakup of time spent for each instantiation. Other data includes how many times each function was called, how many profiled functions were invoked by each function, and what the mean inclusive time per call was. On systems where available, TAU can also use hardware performance counters.

Figure 11 shows the TAU tool, *racy*, displaying the breakdown of the time spent in functions and templates for the Red/Black SOR example running on 32 processors using the SCVE scheduler. The method Iterate_Expr1:run() and Iterate_Expr2::run() refer to the red and black phases of computation, respectively. The other methods are system overheads.

TAU can also generate binary event traces that can be visualized with Vampir, a commercial trace visual-

ization tool. In Fig. 12[1], the black region represents the computation in the Black phase, the white region is the Red phase, and the gray region is the scheduling overhead. It shows the global timeline display in the upper window, where each horizontal set of blocks represents a single thread and the colors represent the profile group that the code belongs to. In the lower window, the parallelism view shows the groups with the number of processors plotted for the entire duration of the program. It is possible to zoom in to any segment on the timeline to explore the details of the trace.

In Fig. 13, the aggregate summary of the time spent in routines across all threads is shown as a pie chart. It shows that most of the time is spent doing useful computations in the Red and Black code segments. The overhead associated with thread scheduling and synchronization is also illustrated in the remaining groups of functions. According to figures Fig. 13 and Fig. 11, the total amount time spent on system overheads was only 3.49%. Despite the low overheads, we do not see quite linear speedup. This suggests that the memory system of the machine and method we have used for data placement along with affinity of data to cpu does not scale perfectly.

8 Future and Continuing Work

We have described how the SMARTS run-time system is implemented in shared-memory multiprocessors. Because the current technology for shared-memory interconnects does not scale to more than about 250 processors, we believe that the future of high-performance computing will be clusters of SMPs. These, however, create a challenging programming environment for application developers because neither the shared-memory model using threads nor the SPMD model with explicit message passing are sufficient to efficiently use the available hardware. Since the interface to SMARTS only involves the creation and the execution of graphs, an interface that is abstract enough to hide the details of the underlying memory model, the end-user need not see whether communication is within or across SMPs.

Another difficulty with data-parallel applications on clusters of SMPs is that the high communication latency requires that computation overlap communication. Conventional models use lightweight threads to achieve this overlap. We believe that the SMARTS runtime system improves upon this model in two important ways. First, the unit of concurrency in SMARTS, the SMARTS iterate, is much lighter than threads because iterates do not need to save and restore contexts. Consequently the cost of switching to another computation while waiting for communication is much cheaper. Second, the asynchrony of SMARTS permits a higher degree of parallelism, which gives the system something to do while waiting for communication. It is not sufficient to just be able to switch to a different thread efficiently; it is also necessary to have something to switch to! Figure 14 uses a one-dimensional stencil application to compare the amount of computation (shaded region) that an SMP box could be doing while waiting for communication using SMARTS with the maximum amount of inter-communication computation using a conventional model. The stencil code corresponding to the diagram is

[1]A color image is available at
http://www.acl.lanl.gov/tau/users/smarts

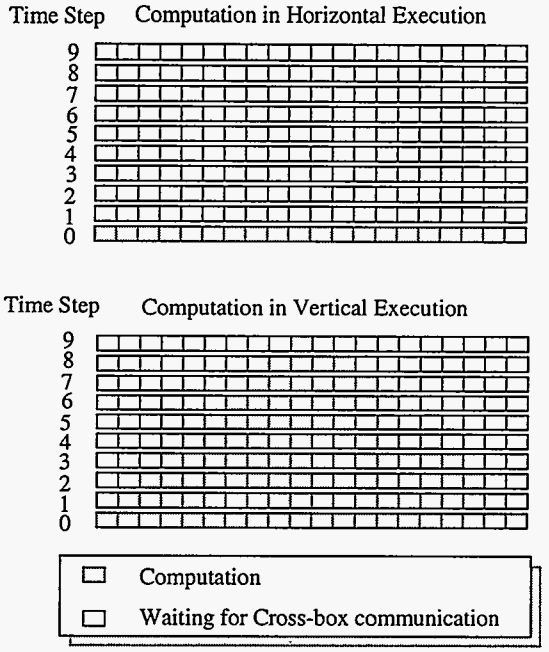

Figure 14: Maximum inter-communication computation in vertical and horizontal computations in 1d-Stencil Computation.

```
for (i = 0; i < 10; ++i) { // time steps
  a(I,J) =  c*(a(I+1,J) + a(I,J) + a(I-1,J));
}
```

In the conventional model (using horizontal execution), communication with the neighboring box must occur on every time step. In contrast, SMARTS allows us to aggressively compute some iterations from subsequent time-steps, restrained only by dependencies in the application. In the SMARTS version, each block of computation is initially ready to compute at the very first time step. Assuming that the communication from the boxes on the left and right side of the picture takes an infinite amount of time, we could still compute all but the leftmost block and rightmost block in the second time step. Similarly, at each of the subsequent time steps, we would continue computing the inner blocks of the array to form the pyramidal shape in the figure.

9 Conclusion

The growing inability of memory systems to keep up with processors makes computational overhead less of a factor in the total execution time. These technological trends necessitate a re-evaluation of the models of execution: methods and algorithms that were previously unusable because of their computational overhead now seem favorable and worth further investigation. This paper introduced a mechanism for improving the locality and parallelism of scientific applications by using vertical execution in which loop iterations of consecutive data-parallel statements are executed in an interleaved fashion. While this model does indeed incur more overhead, this overhead is insignificant compared to the relatively high cost of memory access.

Another important type of overhead is the amount of time required for the scientific programmer to parallelize and to optimize his application for data locality. The SMARTS run-time system and the POOMA frame-

work shield the end-users from these details, allowing them to make more efficient use of their expertise.

Acknowledgements

The research described here was performed under the auspices of the U. S. Department of Energy by Los Alamos National Laboratory under contract No. W-7405-Eng-36.

References

[1] J.R. Allen and K. Kennedy. Automatic loop interchange. *ACM SIGPLAN Notices*, 19(6):233–246, June 1985.

[2] Robert Babb. Parallel Processing With Large-Grain Data Flow Techniques. *IEEE Computer*, 17(7):55–61, July 1984.

[3] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Barbara, California, July 1995.

[4] James A. Crotinger, Julian Cummings, Scott Haney, William Humphrey, Steve Karmesin, John Reynders, Stephen Smith, and Timothy J. Williams. Generic Programming in POOMA and PETE. In *Proceedings of the Dagstuhl Seminar on Generic Programming*, to be published in Lecture Notes in Computer Science. Springer-Verlag, 1999.

[5] John T. Feo and David C. Cann. A Report on the Sisal Language Project. *Jounal of Parallel and Distributed Computing*, 10(4):81–89, December 1990.

[6] Guang R. Gao, Russ Olsen, Vivek Sarkar, and Radhika Thekkath. Collective Loop Fusion for Array Contraction. In *1992 Workshop on Languages and Compilers for Parallel Computing*, number 757 in Lecture Notes in Computer Science, pages 281–295, New Haven, Conn., August 1992. Berlin: Springer Verlag.

[7] A. S. Grimshaw. Easy to Use Object-Oriented Parallel Programming with Mentat. *IEEE Computer*, pages 39–51, May 1993.

[8] A. S. Grimshaw, W. T. Strayer, and P. Narayan. Dynamic Object-Oriented Parallel Processing. *IEEE Parallel and Distributed Technology: Systems and Applications*, pages 33–47, May 1993.

[9] J. Gurd, C. C. Kirkham, and A. P. W. Boehm. The Manchester Prototype Dataflow Computer. *Communication of the ACM*, 28:34–52, January 1985.

[10] J. Gurd, C. C. Kirkham, and A. P. W. Boehm. *The Manchester Dataflow Computing System*, pages 516,517,519,520,529. North-Holland, 1987.

[11] Jose Moreira and Constantine Polychronopoulos. Autoscheduling in a Shared Memory Multiprocessor. In *Proceedings of the IEEE/USP International Workshop on High Performance Computing Compilers and Tools for Parallel Processing*, March 1994.

[12] S. Karmesin, J. Crotinger, J. Cummings, S. Haney, W. Humphrey, J. Reynders, S. Smith, and T. J. Williams. Array Design and Expression Evaluation in POOMA II. In D. Caromel, R.R. Oldehoeft, and M. Tholburn, editors, *Computing in Object-Oriented Parallel Environments*, volume 1505 of *Lecture Notes in Computer Science*, pages 231–238. Springer-Verlag, 1998.

[13] Ken Kennedy and Kathryn S. McKinley. Maximizing Loop Parallelism and Improving Data Locality via Loop Fusion and Distribution. In *1993 Workshop on Languages and Compilers for Parallel Computing*, number 768 in Lecture Notes in Computer Science, pages 301–320, Portland, Ore., August 1993. Berlin: Springer Verlag.

[14] Induprakas Kodukula and Keshav Pingali. Transformations For Imperfectly Nested Loops. In *Supercomputing*, Nov 1996.

[15] E.P Markatos and T. J. LeBlanc. Load Balancing vs Locality Management in Shared Memory Multiprocessors. In *Intl. Conference on Parallel Processing*, pages 258–257, St. Charles, Illinois, August 1992.

[16] C. D. Polochronopoulous and D. Kuck. Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. *IEEE Transactions on Computers*, 36(12):1425–1439, December 1987.

[17] J.V.W. Reynders, P.J. Hinker, J.C. Cummings, S.R. Atlas, S. Banerjeee, W.F. Humphrey, S.R. Karmesin, K. Keahey, M. Srikant, and M. Tholburn. Pooma. In G.V. Wilson and P. Lu, editors, *Parallel Programming Using C++*. MIT Press, 1996.

[18] Hanan Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1990.

[19] S. Shende, A.D. Malony, J. Cuny, K. Lindlan, P. Beckman, and S. Karmesin. Portable Profiling and Tracing for Parallel Scientific Applications using C++. In *Proceedings of the 2nd SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 134–145. ACM, 1998.

[20] S. Shende, A.D. Malony, and S. Hackstadt. Dynamic Performance Callstack Sampling: Merging TAU and DAQV. In B. Kågström et al., editors, *Applied Parallel Computing, PARA'98*, Lecture Notes in Computer Science, No. 1541, pages 515–520. Springer-Verlag, 1998.

[21] P. Tang and P.C. Yew. Processor Self-Scheduling for Multiple Nested Parallel Loops. In *Proc. Int. Conf. on Parallel Processing*, pages 528–535. IEEE, August 1986.

[22] J. Torres, E. Ayguadi, J. Labarta, and M. Valero. Loop Parallelization: Revisiting Framework of Unimodular Transformations. In *Proceedings of the Fourth Euromicro Workshop on Parallel and Distributed Processing, IEEE Computer Society*, pages 420–427, January 1996.

[23] Suvas Vajracharya and Dirk Grunwald. Dependence-Driven Run-Time System. In *Proceedings of Language and Compilers for Parallel Computing*, pages 168–176, 1996.

[24] Suvas Vajracharya and Dirk Grunwald. Loop Reordering and Pre-fetching at Runtime. In *High Performance Networking and Computing*, November 1997.

[25] T.L. Veldhuizen. Expression Templates. *C++ Report*, 7(5):26–31, June 1995.

[26] Michael Edward Wolf. *Improving locality and parallelism in nested loops*. PhD thesis, Stanford University, August 1992.

[27] M.J. Wolfe. *Optimizing supercompilers for supercomputers*. PhD thesis, Univ. Illinois, Urbana, April 1987. Rep. 329.

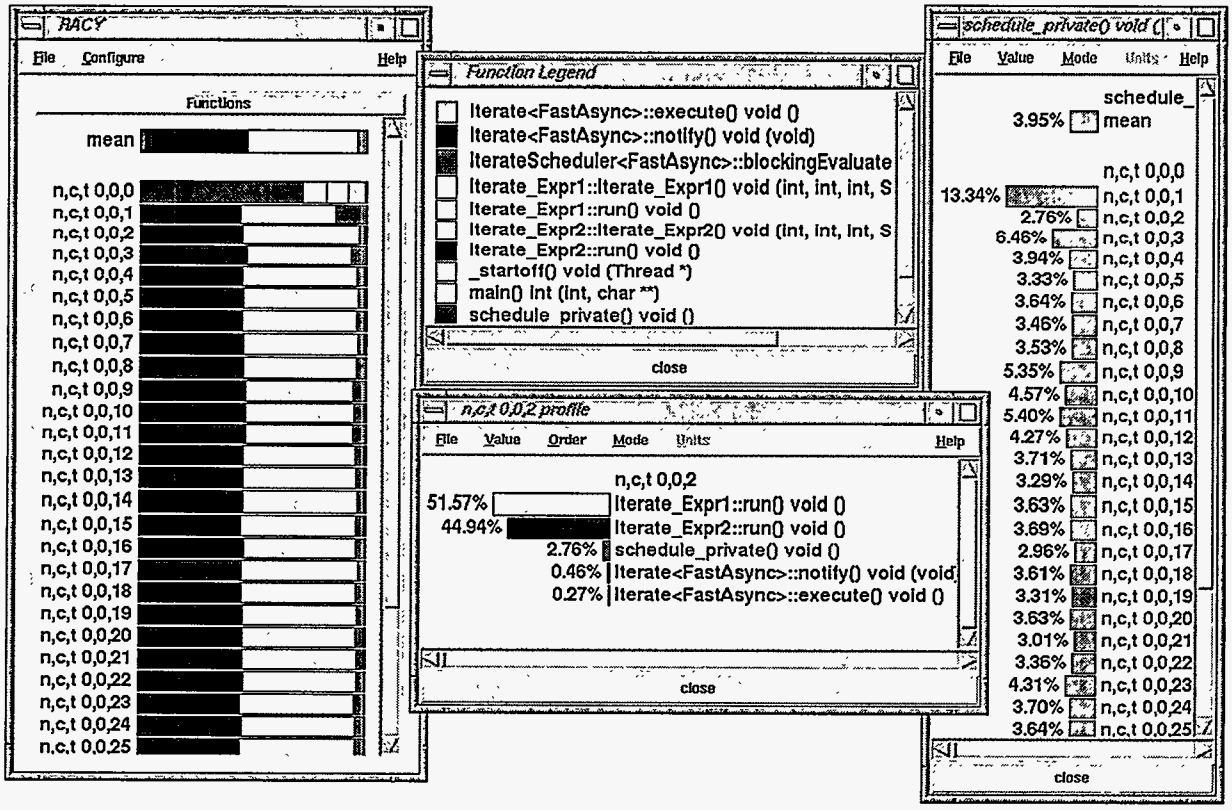

Figure 11: Profile of Red/Black SOR using SCVE scheduler displayed in RACY, a TAU profile visualization tool.

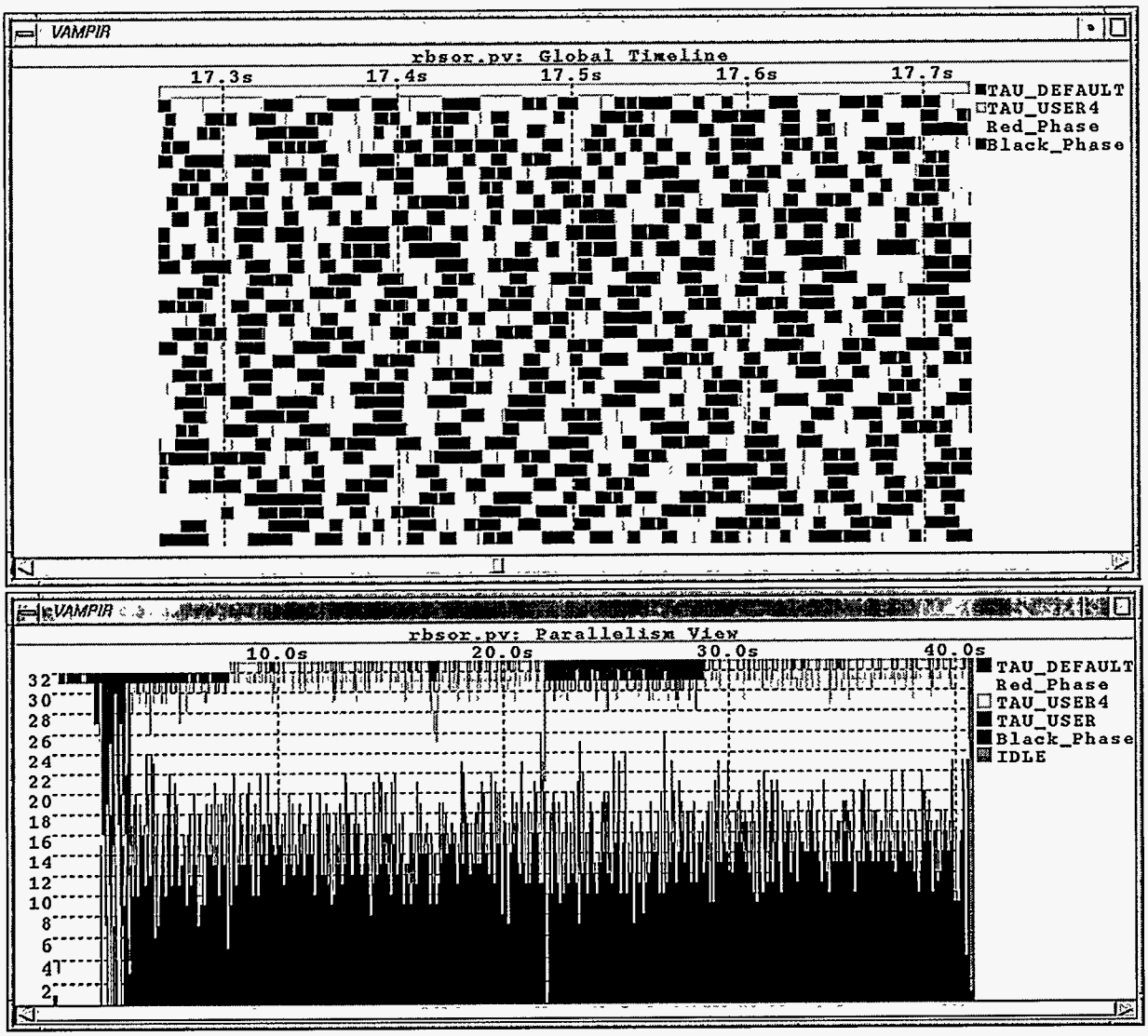

Figure 12: TAU event traces for Red/Black SOR visualized in Vampir.