Targeting a Shared-Address-Space Version of the Seismic Benchmark Seis1.1

Bill Pottenger and Rudolf Eigenmann

Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign
1308 West Main Street, Urbana, Illinois 61801-2307
217-333-6578, fax: 217-244-1351
potteng@csrd.uiuc.edu

Purdue University
School of Electrical and Computer Engineering
336 EE Building, West Lafayette, Indiana 47907
317-494-1741, fax: 317-494-6440
eigenman@ecn.purdue.edu

January 15, 1996

Abstract

We report on our experiences retargeting the seismic processing message-passing application Seis1.1 [11] to an SGI Challenge shared-memory multiprocessor. Our primary purpose in doing so is to provide a shared-address-space (SAS) version of the application for inclusion in the new high performance SPEChpc96 Benchmark Suite. As a result of this work we have determined the language constructs necessary to express Seis1.1 in the SAS programming model. In addition, we have characterized the performance of the SAS versus message-passing programming models for this application on a shared-memory multiprocessor.

Keywords: Shared Memory, Shared Memory Multiprocessor, Distributed Shared Memory, Distributed Shared Memory Multiprocessor, Computational Applications, Seismic Processing, Parallel Processing, Parallel Languages, Language Design, Shared Address Space, Message Passing, Performance Evaluation, Benchmarks, SMP, DSM, SAS, MP

1 Introduction

1.1 Application Overview

The computational application studied in this article is an industrial code representative of modern seismic processing programs used in the search for oil and gas. Seis1.1 [11] consists of a collection of seismic processing routines targeted at parallel processors. Fortran 77 routines for pre-stack and post-stack seismic processing and 3D finite difference modeling are included in the benchmark. Low-level C routines which implement parallel input and output, synchronization primitives, a system-independent message passing layer, and timers complete the suite. Taken together with message-passing primitives, these routines are designed to offer opportunities for parallelism in a portable manner.

Seis1.1 has been revised and adopted as a member of the new SPEChpc96 benchmark suite, which is being established under the sponsorship of the Standard Performance Evaluation Corporation (SPEC) [3]. SPEChpc96 is being defined by a joint effort of industrial members, high-performance computer vendors, and academic institutions. The primary goal is to determine a set of industrially significant applications that can be used to characterize the performance of high-performance computers across a wide range of machine organizations. A secondary goal is to identify a representative workload for high-performance machines which will be made available for scientific study. SPEChpc96 includes multiple program versions for each application, each targeted at a different class of machine architecture. Currently, both a message-passing and a sequential program version are provided. One of the goals of the work described herein is to develop a third version that best exploits the strengths of machine architectures that provide a shared address space.

Seis1.1 was originally written as a collection of serial Fortran codes which were later ported to a parallel, message-passing model [12]. In its current form, both the single and multiprocessor versions are generated from this explicitly parallel message-passing version. The message-passing version is based on a system independent message-passing layer which acts as an interface to commonly available communication libraries such as PVM. As a result, Seis1.1 can be ported with modest effort between parallel architectures. For machines that lack a global address space this is the only model available. Examples of such architectures include the Intel Paragon, the IBM SP2, and the CM5. Other architectures, among them many newer machines, provide hardware and/or software support for a shared-address-space (SAS) programming model. This is essentially a shared-memory model in which data is globally visible to all processes. In most cases, however, access times to the underlying physical memory are non-uniform in nature[1].

1.2 Motivational Background

Many diverse language standards and proposals exist for expressing parallelism. For example, the following languages and language extensions support shared-address-space (SAS) parallel programming:

- Message passing for SAS machines (e.g., MPI [5], PVM [6])
- High Performance Fortran (HPF) [4]
- High Performance C^{++} [7]
- Recent work on dpANS X3.252-199x (X3H5) [9]
- Vendor-specific dialects (e.g., compiler directives, *Step [8])

These languages and extensions are quite diverse, and as a result the following questions remain as yet unanswered:

1. How can portable parallel programming be accomplished in a way that machine-dependent features can be exploited?
2. Relative to well established languages, what is the minimum set of new language features that users must learn to achieve satisfactory application performance?
3. Given multi-language applications such as Seis1.1, what meta-language constructs are necessary to present a uniform interface to the application programmer?
4. Can the tradeoff between ease of use and achievable performance be quantified?

Application-based studies such as the work described herein complement language design projects. Our approach is to seek answers to these questions by first identifying the language elements needed to express important computational applications in the SAS programming model. This will then put us in a position where we can quantitatively define a new language or extend an existing one.

[1]sometimes termed Non-Uniform Memory Access or NUMA

1.3 Objectives

In this paper we describe the transformation of Seis1.1 from a message-passing form to a loop-parallel form based on the shared-address-space programming model. We discuss transformations that we found applicable, as well as other issues encountered. As mentioned above, the benchmark incorporates both a single-processor and message-passing version into one code, using #ifdef preprocessor directives to differentiate the two. The provision of a third version suitable for shared-address-space machines is one of the goals of this study. A second goal is to determine the language constructs necessary to express this program in a way that exploits a shared-address-space architecture efficiently.

2 Comparing the MP and SAS Models

2.1 Overview

Figure 1 pictorially compares the message-passing (MP) model and the shared-address-space (SAS) model. At the point marked FORK (normally at the start of the program), the MP model spawns processes, and control flow enters a parallel region. In the SAS model a parallel loop is started. In both models the number PROCS typically corresponds to the number of physical processors participating in the execution. Although loop parallelism is the most common form of parallelism in the SAS model, other forms, such as parallel tasks (each executing different code), can be implemented based on the same underlying execution scheme.

During execution of the parallel region, communication under the MP model takes place in the form of explicit send and receive operations. This requires that the data communicated be packaged efficiently so that architectural communication latencies do not limit performance. In the SAS model, communication usually takes place using shared-memory copy operations combined with barrier synchronizations[2]. Of course, many other forms of communication are possible. Data is always private in the MP model,

[2] Often provided implicitly, e.g., when exiting a parallel loop

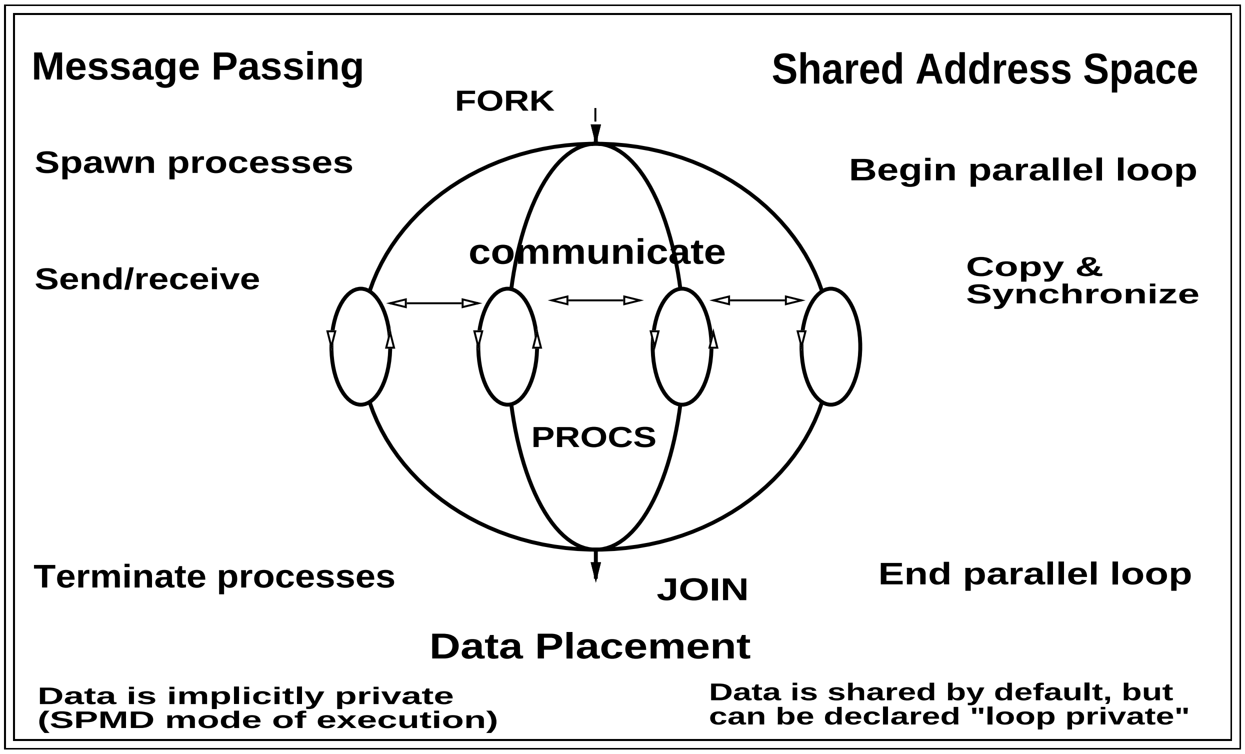

Figure 1: Message-Passing versus Shared-Address-Space Programming Models

whereas there can be both process-private and shared data in the SAS model.

Although the underlying execution scheme of both models is similar, the language constructs differ. More importantly, the SAS model distinguishes itself in that it allows the programmer to declare data that can be seen by all participating processes.

Our primary focus in this paper is on machines that support a hardware-based shared-address-space model (e.g., symmetric multiprocessors (SMPs) and distributed shared-memory (DSM) machines). Due to the similarity in the virtual shared-memory model presented to many users of message-passing machines[3], we expect our conclusions will apply to these architectures to a certain degree as well. However, a discussion of this point is beyond the scope of this paper.

[3]For example, HPF directives inserted in Fortran 77 code

3 Issues in Conversion: MP to SAS

Our starting point in the development of the shared-address-space version of Seis1.1 was the message-passing version described in Section 1.1. The following issues arose during the transformation:

- Conversion of the SMPD-style computation to loop-parallel form
- Data Privatization
  - Privatization of Fortran Common Blocks
  - Privatization of Static Global C Variables and Structures
- Replacement of Send/Receive Message Layers
  - Communication of Fortran Common Block Data
  - Implementation of Global Reduction and Data Collection
- Optimization of Barrier Synchronization
- Issues in Concurrent Access to Library Routines

In the following sections, each of these issues will be dealt with in turn.

3.1 Converting SPMD code to loop-parallel form

The Seis1.1 benchmark that served as the starting point for our work is written in a Single Program Multiple Data (SPMD) style. Under the SPMD execution model parallel processes are spawned at program start and each process then executes identical program text. Conditional control flow based on individual process IDs determines different paths of execution for different processes executing this text. For example, the code *if (pid = master_id) then <initialization>* is typically used for initializations that need only be done by the master process.

A key insight was the realization that a loop-parallel form of the program could be created by enclosing the *entire body* of the original SPMD program in a parallel loop with the number of iterations equal to the number of parallel processes. We term this transformation *de-SPMDizing*.

The following exemplifies the de-SPMDizing transformation expressed using the SGI `C$DOACROSS` directive:

```
program SPMD
if (not spawned) spawn(np)
if (pid = master_id)
    initialize()
decompose(n,np,pid,li,ll)
do j = li, ll
    ...
enddo
end
```

⇒

```
program de-SPMDized
initialize()
C$DOACROSS
do i = 1, np
    decompose(n,np,i,li,ll)
    do j = li, ll
        ...
    enddo
enddo
end
```

In the above, *np* is the number of processes, *pid* is the process id, and *li* and *ll* stand for the local init and local limit of each decomposed slice of the global iteration space, which ranges from 1 to n.

The SPMD code decomposes the iteration space of the loop yielding a local iteration range for each process involved in the computation. Each process then executes its slice of the iteration space. The same code is executed after the transformation, but wrapped in a parallel loop. Effectively, the parallel loop replaces the code which spawns the SPMD processes. The iteration space of the new parallel loop runs from one to the number of processes executing in parallel.

The transformation allows initializations that are conditionally executed under the *if (pid = master_id)* guard to be hoisted outside the parallel loop. This is exemplified by the relocation of the master process' call to *initialize()* in the code above.

Before closing this section, a brief note on the syntax of the SGI DOACROSS directive is in order. The DOACROSS directive syntax provides for the specification of several parameters including scheduling, data privatization, reductions, and last values. Data is by default shared, unless specifically declared otherwise. It must be noted that although the term DOACROSS implies that iterations can be executed in an ordered, synchronized fashion, SGI DOACROSS loops are fully parallel, and are more commonly referred to as DOALL loops.

3.2 Data Privatization

After making the de-SPMDizing transformation, it became necessary to privatize variables which, although implicitly private under the MP (message-passing) model, were by default shared under the

SAS model. When a variable is privatized, multiple independent instances coexist, one per process (as opposed to shared variables where a single instance is globally visible). Two broad classes of variables requiring privatization were discovered: array and scalar data declared in common blocks in Fortran, and static variables declared globally in C.

3.2.1 Privatization of Fortran Common Blocks

The MP version of Seis1.1 depends on process-private scalar and array data for its storage needs. As a result, some means of privatizing these common block variables was needed in the SAS version. As mentioned above, the SGI DOACROSS directive allows variables to be declared SHARED or LOCAL, corresponding to shared and private variables, respectively. This facility allows both scalar and array variables to be privatized. However, only arrays declared with non-symbolic dimensions can be privatized in this way. For example:

```
subroutine sub(a,n)
    real a(n)
C$DOACROSS LOCAL(a)
    do i = 1, n
        ...
    enddo
    end
```

Although the actual size of the array a is statically known outside the scope of subroutine sub, the above LOCAL declaration will result in a compilation error due to the symbolic dimension n. However, the SGI Challenge loader (the `ld` command) provides an alternate method for privatizing variables that are members of common blocks. Using the `-Xlocaldata name [,name]` switch, named common blocks can be privatized in their entirety. Common blocks listed after this command line flag are allocated by the Irix operating system in process-private memory.

Just as important is compiler support for the initialization of private data. However, no support is provided for such *copy-in* operations, as they are termed, to private variables declared using the DOACROSS LOCAL directive syntax. Nonetheless, support for copy-in of variables privatized using the aforementioned loader switch is supplied in the form of the `C$COPYIN` directive (discussed in Section 3.3.1 on common block data communication). Given this choice of constructs, all common blocks were privatized using the -Xlocaldata interface.

3.2.2 Privatization of Static Global C Variables and Structures

Seis1.1 includes several thousand lines of C code containing various global, static variables and structures. In the SPMD (MP) variant, all of these variables are process-private due to the fact that each process maintains its own address space. Under the SAS model, however, global C data structures are shared by default. As a result, either access to these variables must be synchronized or, where possible, the variables must be explicitly privatized.

We have identified two classes of C variables in Seis1.1: write-once variables that are initialized by the master and henceforth read by all processes, and variables that are written and then read repeatedly during the course of parallel execution.

Write-once variables are written in guarded statements such as

```
if (node = master)
    assignment
```

Several such variables are written during various initialization phases of the MP version. As discussed in Section 3.1, initialization code such as this can be hoisted outside the parallel loop. In one case, this is what was done. A second case involved initialization that occurs midway through the actual execution of the suite. Although this code could, in principle, also have been hoisted outside the parallel loop, the *if (pid = master_id)* guard and accompanying synchronization were simply left in place.

The second class of variables were privatizable. This was implemented in all cases save two using data expansion. In data expansion the shared data structures are reduplicated, creating independent copies for each process. The i-th process then uses the i-th copy of the structure. One of the two remaining cases was the process identification variable, which we replaced by a system call returning the value of the process id. The final case was resolved by recognizing that the structure in question was both defined and used in the same function, and could be allocated as a local variable on the function's call stack.

3.3 Replacement of Send/Receive Message Layers

As noted in the Introduction, Seis1.1 uses a system-independent message-passing layer. The actual underlying message-passing layer employed in release 1.1 is PVM [6]. In order to port the MP/PVM based version to the SAS model, the message-passing layer had to be replaced. In order to understand how this replacement was done, some background is necessary on the different modes of communication employed in the suite.

There are two distinct communication paradigms used in the MP version of Seis1.1: PVM-based sends and receives, and optional shared-memory copies for AT&T System V shared-memory architectures. AT&T System V interprocess communication (IPC) facilities allow two or more processes to share segments of their virtual address spaces [2]. These facilities, when combined with shared-memory copy operations, can be used to replace PVM message-passing primitives. This functionality is optionally enabled for array transpose operations which take place in one of the computationally intensive phases of the application. The variable SHMEM selects between the two schemes at compile time. If SHMEM is defined, array transpose operations take place using System V shared memory segments, otherwise they take place using PVM sends and receives. In either case, broadcast and reduction operations take place using PVM sends and receives. Thus, optional support for shared-memory array transpose operations is already present in the MP version of the Seis1.1 benchmark, and we have taken advantage of this facility in implementing the SAS version.

3.3.1 Communication of Fortran Common Block Data

Earlier, in Section 3.2.1, we discussed the privatization of common blocks. The structure of Seis1.1 is such that the master initializes its own common block data, and this data is then broadcast to the slaves and copied into each of their common blocks. In the MP/PVM version this broadcast takes

```
      COMMON /sys/ sys_start, ..., sys_end
      ...
C Broadcast system common from the master's common block to the slaves
C$COPYIN /sys/
      ...
C$DOACROSS
      do i=1,np
            ... access to variables in privatized copies of /sys/
      enddo
```

Figure 2: Copy-in using the SGI C$COPYIN Directive

place using PVM sends and receives. In the SAS version, however, some provision was needed for communicating the common block initial values to the slaves. This was accomplished using the SGI Fortran directive `C$COPYIN item [,item]`, where *item* is a member of a local common block privatized using the `-Xlocaldata` loader switch. *item* can be a "variable, an array, an individual element of an array, or the entire common block" [2]. An entire common block is copied-in using the standard Fortran notation "/name/", where *name* is the name of the common block. Data structures identified by name in the item list are copied from the master's copy of the data structure to the slave's copy at the program point where the `C$COPYIN` directive is placed. This copy-in is done at runtime by the master in a serial section of the code, with the result that the slave data structures are all initialized from the master prior to entry to a parallel section of code. Since all common blocks in the SAS version of Seis1.1 were privatized using the `-Xlocaldata` ld switch, the `C$COPYIN` directive was used to copy the master's initialized common block data into the slave common blocks.

Figure 2 depicts an example in which PVM-based broadcasts were originally made to slave processes by calling the routine sysbcst(sys_start, sys_end), where the variables sys_start and sys_end served as place holders used to mark the start and end of the sys common block. In the transformed code above, the call to sysbcst() was commented out and replaced by the `C$COPYIN /sys/` directive placed at the same point in the program.

3.3.2 Implementation of Global Reduction and Data Collection

The two remaining communication operations in Seis1.1 are a global reduction used in calculating an average amplitude for use as a checksum and the collection of timing information across processes. In the MP version, both of these operations are done using PVM sends and receives. The SAS version of the program employed array expansion to create global array structures visible to all processes. This was possible because the sizes of the data structures are statically known (i.e., the size and number of messages being passed is known at compile time).

The following exemplifies this transformation:

```
  if (node.eq.master) then                       if (node.eq.master) then
C Master node receives from                         call jsync()
C other nodes and accumulates                       do jn=1,nodes-1
    do jn=1,nodes-1                                    do j=1,5
C Receive amplitude array                                 do i=1,11
       call jrecv( jn, sa, len )                             amp(i,j) = amp(i,j) + amp_exp(i,j,jn)
C Accumulate total                                        enddo
       do j=1,5                                        enddo
          do i=1,11                                 enddo
             amp(i,j) = amp(i,j) + sa(i,j)      ⇒   else
          enddo                                     do j=1,5
       enddo                                           do i=1,11
    enddo                                                 amp_exp(i,j,node) = amp(i,j)
  else                                                 enddo
C Other nodes send amplitude array                  enddo
    call jsend( node, amp, len, master )            call jsync()
  endif                                          endif
```

In the above example, the routines *jrecv*() and *jsend*() are part of the system-independent message-passing layer discussed in the introduction to Section 3.3. These routines have been replaced by shared-memory copies to and from the global array *amp_exp*. During the reduction in the transformed code, each process copies its process-private data into its assigned section of *amp_exp*, and the master reduces this global array across these sections.

The collection of timing information is done in a similar way using a global array. Access to these arrays is synchronized to insure that the data values from all slave processes are present before the master attempts to read them.

```
void
barrier() {
        int i, delay = 1, delaymax = 1024, wasless;
        wasless = (* svptr < nt);
        * points_next_to_svptr = 0;
        if (test_then_add((unsigned long *) svptr,1) == rollover)
                * svptr = 0;
        while ((* svptr < nt) == wasless) {
                for (i=0; i < delay; i++) ;
                delay = (delay > delaymax) ? delaymax : delay * 2;
        }
}
```

Figure 3: Custom Coded Barrier with Backoff Delay

3.4 Optimization of Barrier Synchronization

Synchronization in the MP version of the program primarily occurs in the form of barriers, and is handled via PVM sends and receives. The SAS version initially employed a barrier based on AT&T System V shared-memory semaphores which was provided with the Seis1.1 benchmark. However, performance of the IPC semaphore-based barriers was dismal, and we replaced them with a custom-coded barrier. Using an algorithm provided in [1], a barrier based on the SGI atomic test-then-add primitive was developed. This proved to be much more efficient than the IPC-based barrier. Additional optimizations to the test-then-add barrier were investigated, including cache-line alignment of the barrier synchronization variable, and exponential backoff while spin-waiting at the barrier [10]. The code is displayed in Figure 3.

The global variable *nt* in Figure 3 stands for the number of threads, and is initialized prior to entry to the barrier. The global *rollover* is similarly initialized to $2 * nt - 1$. The alignment and size of the synchronization variable (to which the pointer *svptr* points) is not shown. On the SGI Challenge, cache lines are 128 bytes in length, and the synchronization variable was declared with a size of 128 bytes and aligned at an address in memory which corresponded to the beginning of a cache line. This was done to avoid false-sharing of the cache line containing the synchronization variable.

A second, minor optimization is the "pre-fetch" of the synchronization variable. This was accomplished by writing to an unused portion of the cache line containing the synchronization variable. Essentially this is a method of "priming" the state of the cache line in the coherence protocol to write-ownership in preparation for the following test-then-add operation.

A third optimization involved the insertion of a backoff delay in testing the synchronization variable while waiting at the barrier. Various values for the delay (powers of two) were tested empirically using an exponential backoff scheme. The measurements showed that although delays ≤ 1024 provided better performance, it was not significantly better than a delay of 0. We found this result noteworthy in that it was not necessarily expected.

3.5 Issues in Concurrent Access to Library Routines

The Seis1.1 benchmark does parallel input and output using low-level routines written in C. On the SGI Challenge, a system flag named `CONF_STHREADIOON` controls the function of standard C library I/O routines when executing in parallel. When set, all access to C library I/O routines is single threaded. Due to the fact that the SGI Fortran environment does not support parallel I/O[4], this flag is set off by default. As a result, it was necessary to insert a call to set the flag prior to performing I/O.

A second system specific requirement is a similar flag which forces malloc operations to be single-threaded. This is `CONF_STHREADMALLOCON`, short for "single threaded malloc on". The final flag is `CONF_STHREADMISCON`, a catch-all which means "single threaded miscellaneous on". This enables single threaded execution of miscellaneous routines in the C stdio library such as opendir, closedir, seekdir, etc. All three of these flags were set using the *usconfig* interface [2].

4 Results

Tables 1 and 2 summarize our timing results on an SGI Challenge multiprocessor. The measurements are based on runs made at the National Center for Supercomputing Applications (NCSA). The timings

[4] Since the time of writing, SGI has released support for parallel I/O in Fortran

	wallclock seconds	speedup
Serial	20.7	1
2 Processors		
Parallel MP	11.7	1.77
Parallel SAS	10.5	1.97
4 Processors		
Parallel MP	7.3	2.84
Parallel SAS	5.7	3.63
8 Processors		
Parallel MP	3.9	5.31
Parallel SAS	3.7	5.59

Table 1: Timings and Speedup for Seis1.1 MP and SAS Versions on an SGI Challenge, Small Problem

	wallclock min:sec	speedup
Serial	93:48.2	1
2 Processors		
Parallel MP	51:22.4	1.83
Parallel SAS	52:30.4	1.79
4 Processors		
Parallel MP	27:53.1	3.36
Parallel SAS	28:05.8	3.34
8 Processors		
Parallel MP	14:56.5	6.28
Parallel SAS	14:55.0	6.29

Table 2: Timings and Speedup for Seis1.1 MP and SAS Versions on an SGI Challenge, Medium Problem

are wall-clock (elapsed) execution time for small and medium problem sizes in two dimensions. Medium in this context refers to the second of four problem sizes available with Seis1.1. The medium problem was felt to provide enough work to be representative of the application's performance.

The timings were obtained during single-user mode on the machine. In addition, system services provided by the SGI operating system were used to manually increase process priorities to a real-time mode. The MP runs were made using versions 3.3.7 and 3.3.8 of a shared-memory implementation of PVM on the SGI Challenge. Multiple runs of the SAS version were made in order to optimize the code. Briefly, different runs were made under different scheduling conditions combined with different values for the backoff delay, as described in Section 3.4. For the SAS results reported above, system services were invoked to disable process migration.

The performance of the two variants is almost identical. This result was unexpected in that we initially believed we would see additional communications-related overhead in the message-passing version. However, after some reflection on the matter, we realized that this application has a fairly high computation to communication ratio (approximately 12) in the depth-migration phase of the medium problem size on eight processors. This is indicative of the fact that relatively little communication takes place in the medium problem size computation.

An interesting trend surfaces when comparing the computation to communication ratios and speedups of the small and medium problem sizes. The ratio for the small problem size for the depth-migration phase on eight processors is approximately four, roughly 1/3 that of the medium problem size. This trend is echoed in the speedups of the two problem sizes on eight processors as can be seen in Tables 1 and 2 above.

The similarity in performance under the MP and SAS models is indicative of the nature of the seismic processing algorithms used in this application. The decomposition of the problem domain results in data access patterns with high locality. A cache-based SMP such as the SGI Challenge is able to take advantage of this locality under both programming models.

5 Conclusion

Our primary goal was to produce a shared-address-space (SAS) version of the SPEChpc96 candidate Seis1.1. To date, we have reached this goal in the form of a version specific to the SGI Challenge multiprocessor. We have demonstrated that the SAS version of the suite performs as well as the original message-passing (MP) version. Given that the SAS programming model is often preferred by software engineers, the SAS version of the program developed in this effort is expected to be of interest to vendors that provide a SAS machine architecture.

Our secondary goal was to identify language constructs necessary for expressing programs efficiently in the SAS model. To this end we have identified the constructs necessary for porting Seis1.1 from the MP to the SAS model. Table 3 summarizes these constructs, many of which must be viewed as meta-language constructs. We have found that although we succeeded in expressing the application in a SAS version, the language support available did not make the task easy.

1. Construct for expressing loop-level parallelism
2. Data Privatization
   (a) Construct for loop-private declaration of Fortran data structures
   (b) Construct for privatization of C variables and structures
3. Communication
   (a) Construct enabling copy-in of various data structures
   (b) Construct enabling global reductions of data structures
   (c) Construct enabling global collection of data structures
4. Construct for expressing barrier synchronization
5. Construct providing concurrent access to standard library routines

Table 3: Language constructs used for expressing the Seis1.1 application in SAS form

Loop-level parallelism in Fortran is supported by the DOACROSS directive, which allows the declaration of loop-private data. However, for variables declared private using the DOACROSS directive syntax, it is not possible to perform the necessary initialization (copy-in) operations required by the application. Although alternative methods for the privatization and initialization of variables declared in Fortran common blocks exist, they are not integrated into language elements nor into a single directive syntax available to the programmer.

Seis1.1 is a hybrid C/Fortran code. As a result, it presents many new challenges which do not exist when working within a single language environment. For example, little if any support is provided for the privatization of C variables and structures in an inter-language environment. Similarly, concurrent access to library routines is complicated by the use of languages with different underlying assumptions about parallel I/O. Work-arounds such as data expansion can be employed, but the lack of consistent meta-language constructs for dealing with these issues is a hindrance to the development of multiple-language applications for shared-memory multiprocessors.

Finally, we have determined that the most important synchronization primitives are barrier operations. Although support for barrier synchronization does exist, the underlying implementation on the Challenge architecture performed poorly when compared to our custom implementation based on readily available library routines.

In response to the questions posed in the introduction concerning portability, minimal language extensions, meta-language constructs, and performance tradeoffs, our study of a large, industrially significant application has given us partial answers: we have identified a small number of constructs that, when implemented across languages available on shared-address-space machines, will significantly reduce the effort of porting applications to this class of machine. The constructs we have identified are minimal in that we have employed only those crucial to the accomplishment of our porting task.

Similar studies of additional industrially significant applications are needed to determine a full set of language extensions which can be applied across multiple language environments. We believe that if languages are extended with such a minimal set of constructs, the burden imposed on the user of parallel high-performance machines will become more manageable. The resulting increased accessibility of high-performance computing will benefit a wide user community.

References

[1] George Almasi and Allan Gottlieb. *Highly Parallel Computing.* Benjamin/Cummings, Redwood City, CA, 1989.

[2] SGI Documentation. Fortran 77 Programmer's Guide and associated man pages. Silicon Graphics, Inc., 1994.

[3] Rudolf Eigenmann and Siamak Hassanzadeh. Evaluating High-Performance Computer Technology through Industrially Significant Applications. *IEEE Computational Science & Engineering*, Spring 1996.

[4] High Performance Fortran Forum. High Performance Fortran Language Specification, version 1.0. Technical report, Rice University, Houston Texas, May 1993.

[5] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Technical report, University of Tennessee, Knoxville, Tennessee, May 1994.

[6] G. A. Geist, A. L. Beguelin, J. J. Dongarra, W. Jiang, R. J. Manchek, and V. S. Sunderam. *PVM: Parallel Virtual Machine - A Users Guide and Tutorial for Network Parallel Computing.* MIT Press, 1994.

[7] HPC++ Whitepapers and Draft Working Documents. The HPC++ working group. Technical report, Indiana University, 1995. Presented at Supercomputing '95, available at http://extreme.indiana.edu/hpc++/index.html.

[8] Kuck & Associates Inc. ∗Step User's Guide, Version 1.0, November 1995.

[9] Bruce Leisure. Parallel Processing Model for High Level Programming Languages. Technical report, Kuck & Associates, Inc., Champaign, Illinois 61820, April 5th, 1994. Available at http://www.kai.com/hints/csbparallelism.html.

[10] John M. Mellor-Crummey and Michael L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems, Vol 9, No 1*, pages 21–65, Feburary 1991.

[11] Society for Exploration Geophysics. *A Benchmark Suite for Parallel Seismic Processing*, 1991.

[12] Supercomputing 1992. *A Benchmark Suite for Parallel Seismic Processing*, 1992.