1-1-2002

Scalability and performance of MPI, HPF and OpenMP on an Origin 2000

Zhe Guan
Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/rtd

Recommended Citation
Guan, Zhe, "Scalability and performance of MPI, HPF and OpenMP on an Origin 2000" (2002).
Retrospective Theses and Dissertations. 19863.
https://lib.dr.iastate.edu/rtd/19863

This Thesis is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
Scalability and performance of MPI, HPF and OpenMP on an Origin 2000

by

Zhe Guan

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
Glenn R. Luecke, Co-major Professor
David Fernandez-Baca, Co-major Professor
Shashi Gadia

Iowa State University
Ames, Iowa
2002

Copyright © Zhe Guan, 2002. All rights reserved.
This is to certify that the master’s thesis of

Zhe Guan

has met the thesis requirements of Iowa State University

Signatures have been redacted for privacy
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF FIGURES</td>
<td>iv</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>v</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>vi</td>
</tr>
<tr>
<td>CHAPTER 1. GENERAL INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>CHAPTER 2. SCALABILITY AND PERFORMANCE OF MPI, HPF AND OpenMP ON AN SGI Origin 2000</td>
<td>3</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>3</td>
</tr>
<tr>
<td>1. INTRODUCTION</td>
<td>3</td>
</tr>
<tr>
<td>2. DATA DISTRIBUTION</td>
<td>5</td>
</tr>
<tr>
<td>3. TIMING METHODOLOGY</td>
<td>5</td>
</tr>
<tr>
<td>4. TESTS AND RESULTS</td>
<td>11</td>
</tr>
<tr>
<td>5. CONCLUSIONS</td>
<td>50</td>
</tr>
<tr>
<td>CHAPTER 3. GENERAL CONCLUSIONS</td>
<td>51</td>
</tr>
<tr>
<td>BIBLIOGRAPHY</td>
<td>52</td>
</tr>
<tr>
<td>APPENDIX</td>
<td>54</td>
</tr>
<tr>
<td>ACKNOWLEDGEMENTS</td>
<td>57</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

Figure 1. 51 trials for the HPF copy test with $n = 128 \times 1024$ and $p = 32$. 10
Figure 2. Timing results of the dot product test in milliseconds for $n = 16 \times 1024$. 15
Figure 3. Timing results of matrix times vectors with row blocking test in milliseconds for $n = 256$. 19
Figure 4. Timing results of matrix times vectors with row blocking test in milliseconds for $n = 256$. 20
Figure 5. Timing results of matrix times vectors with column blocking test in milliseconds for $n = 256$. 25
Figure 6. Timing results of matrix times vectors with column blocking test in milliseconds for $n = 256$. 26
Figure 7. Timing results of matrix times matrix test in milliseconds for $n = 128$. 31
Figure 8. Timing results of matrix times matrix test in milliseconds for $n = 128$. 32
Figure 9. Timing results of matrix transpose test in milliseconds for $n = 64$. 37
Figure 10. Timing results of Stencil test in milliseconds for $n = 256$. 41
Figure 11. Timing results of LU benchmark in seconds for Class B. 43
Figure 12. Timing results of copy test in milliseconds for $n = 128 \times 1024$. 46
Figure 13. Timing results of array replication test in milliseconds for $n = 128 \times 1024$. 49
LIST OF TABLES

Table 1. Timing Results of the dot product test in milliseconds for $n = 16 \times 1024$. 54
Table 2. Timing results of matrix times vectors with row blocking test in
milliseconds for $n = 256$. 54
Table 3. Timing results of matrix times vectors with column blocking test in
Milliseconds for $n = 256$. 54
Table 4. Timing results of matrix times matrix test in milliseconds for $n = 128$. 55
Table 5. Timing results of matrix transpose test in milliseconds for $n = 64$. 55
Table 6. Timing results of stencil test in milliseconds for $n = 256$. 55
Table 7. Timing results of LU benchmark in seconds for Class B. 56
Table 8. Timing results of copy test in milliseconds for $n = 128 \times 1024$. 56
Table 9. Timing results of array replication test in milliseconds for $n = 128 \times 1024$. 56
The purpose of this study is to investigate the scalability and performance of 9 HPF programs using two HPF compilers and to compare their performance with equivalent OpenMP and MPI programs on an SGI Origin 2000. In all cases either the MPI or OpenMP implementations performed and scaled the best. For most cases, at least one of the HPF compilers performed and scaled about the same as either the MPI or OpenMP implementations. For LU benchmark, one of the HPF compilers performed and scaled about the same as both the OpenMP and MPI implementations.
CHAPTER 1. GENERAL INTRODUCTION

INTRODUCTION

The High Performance Fortran (HPF) [1] standard was introduced in 1994. HPF is a high level parallel language whose primary goal was to enable one to write portable, efficient parallel programs for distributed memory parallel computers without explicitly specifying the message passing. Another goal of HPF was to develop a programming standard that allowed existing serial codes to be easily ported onto distributed memory machines.

OpenMP [2] provides a standard syntax for writing portable parallel programs for shared memory parallel computers. OpenMP contains a set of standard compiler directives that enables programmers to express shared-memory parallelism. Both OpenMP and HPF use the single-program multiple data (SPMD) programming model. Unlike HPF, OpenMP has no directives that specify data layout amongst processors.

The Message Passing Interface (MPI) [3] standard was introduced in 1994. MPI is a message-passing library, a collection of routines that enable communication among the processors in a distributed memory parallel computer. The MPI programming model supports only local data.

The purpose of this paper is to investigate the scalability and performance of 9 HPF programs using two HPF compilers and to compare their performance with equivalent OpenMP and MPI programs on an SGI Origin 2000. One of these 9 programs is the NAS Parallel LU Benchmark [5]. The Portland Group’s HPF [13] and the ADAPTOR [12] HPF compilers were used for this study. Both compilers scan the HPF code and generate Fortran code with calls to MPI routines. This Fortran code is then compiled and executed using SGI’s Fortran compiler and SGI’s MPI. The Portland Group’s HPF (PGHPF) uses either its own (the default option) or the MPI (the -Mmpi option) communication library for message passing. To make PGHPF performance comparable with that of MPI and ADAPTOR, the
authors chose the –Mmpi option. (For most tests, performance results were better using the –Mmpi option.)

The SGI Origin 2000 used for this paper is a 256-processor machine located in Eagan, Minnesota. The Origin machine is a 128-node machine and each node has two 250 MHz MIPS R10000 processors sharing a common memory. This processor has two levels of cache. The primary data-cache and instruction-cache are both two-way set associative, each of size 32*1024 bytes. The 4*1024*1024 byte secondary cache is used for both data and instructions. The communication network among nodes is a hypercube for up to 16 nodes. The communication network was a hypercube for up to 32 processors and is called a “fat bristled hypercube” for more than 32 processors since multiple hypercubes were interconnected via a CrayLink Interconnect [4]. For all tests, the IRIX 6.5 operating system, the MIPSpro Fortran compiler version 7.3.1.3m with –O3 –64 option, mpt 1.5.3, the PGHPF version 3.2 and ADAPTOR HPF compiler version 7.0 were used. When using the Portland Group’s HPF compiler, the –O3 option was used. As recommended by the author of ADAPTOR, no special options were used when compiling with ADAPTOR. ADAPTOR produces Fortran MPI code which is then executed using SGI’s Fortran 90 compiler and SGI’s MPI. The PGHPF compiler can be purchased from the Portland Group. The ADAPTOR HPF compiler is public domain software and can be obtained from the web site listed in [12]. All tests were run one at a time with nobody else using the machine.
CHAPTER 2. Scalability and Performance of MPI, HPF and OpenMP on an SGI Origin 2000

Glenn R. Luecke, Zhe Guan, Thomas Brandes
Iowa State University, Ames, Iowa 50011-2251, USA
SCAI, D-53754 Sankt Augustin, Germany
grl@iastate.edu, zheguan@iastate.edu, Thomas.Brandes@scai.fraunhofer.de
February 4, 2002

ABSTRACT. The purpose of this paper is to investigate the scalability and performance of 9 HPF programs using two HPF compilers and to compare their performance with equivalent OpenMP and MPI programs on an SGI Origin 2000. In all cases either the MPI or OpenMP implementations performed and scaled the best. For most cases, at least one of the HPF compilers performed and scaled about the same as either the MPI or OpenMP implementations. For LU benchmark, one of the HPF compilers performed and scaled about the same as both the OpenMP and MPI implementations.

1. INTRODUCTION

The High Performance Fortran (HPF) [1] standard was introduced in 1994. HPF is a high level parallel language whose primary goal was to enable one to write portable, efficient parallel programs for distributed memory parallel computers without explicitly specifying the message passing. Another goal of HPF was to develop a programming standard that allowed existing serial codes to be easily ported onto distributed memory machines.

OpenMP [2] provides a standard syntax for writing portable parallel programs for shared memory parallel computers. OpenMP contains a set of standard compiler directives that enables programmers to express shared-memory parallelism. Both OpenMP and HPF use the single-program multiple data (SPMD) programming model. Unlike HPF, OpenMP has no directives that specify data layout amongst processors.

The Message Passing Interface (MPI) [3] standard was introduced in 1994. MPI is a message-passing library, a collection of routines that enable communication among the processors in a distributed memory parallel computer. The MPI programming model supports only local data.
The purpose of this paper is to investigate the scalability and performance of 9 HPF programs using two HPF compilers and to compare their performance with equivalent OpenMP and MPI programs on an SGI Origin 2000. One of these 9 programs is the NAS Parallel LU Benchmark [5]. The Portland Group’s HPF [13] and the ADAPTOR [12] HPF compilers were used for this study. Both compilers scan the HPF code and generate Fortran code with calls to MPI routines. This Fortran code is then compiled and executed using SGI’s Fortran compiler and SGI’s MPI. The Portland Group’s HPF (PGHPF) uses either its own (the default option) or the MPI (the –Mmpi option) communication library for message passing. To make PGHPF performance comparable with that of MPI and ADAPTOR, the authors chose the –Mmpi option. (For most tests, performance results were better using the –Mmpi option.)

The SGI Origin 2000 used for this paper is a 256-processor machine located in Eagan, Minnesota. The Origin machine is a 128-node machine and each node has two 250 MHz MIPS R10000 processors sharing a common memory. This processor has two levels of cache. The primary data-cache and instruction-cache are both two-way set associative, each of size 32*1024 bytes. The 4*1024*1024 byte secondary cache is used for both data and instructions. The communication network among nodes is a hypercube for up to 16 nodes. The communication network was a hypercube for up to 32 processors and is called a “fat bristled hypercube” for more than 32 processors since multiple hypercubes were interconnected via a CrayLink Interconnect [4]. For all tests, the IRIX 6.5 operating system, the MIPSPro Fortran compiler version 7.3.1.3m with –O3 –64 option, mpt 1.5.3, the PGHPF version 3.2 and ADAPTOR HPF compiler version 7.0 were used. When using the Portland Group’s HPF compiler, the –O3 option was used. As recommended by the author of ADAPTOR, no special options were used when compiling with ADAPTOR. ADAPTOR produces Fortran MPI code which is then executed using SGI’s Fortran 90 compiler and SGI’s MPI. The PGHPF compiler can be purchased from the Portland Group. The ADAPTOR HPF compiler is public domain software and can be obtained from the web site listed in [12]. All tests were run one at a time with nobody else using the machine.
2. DATA DISTRIBUTION

OpenMP is designed for shared memory machines and hence does not contain any data distribution directives. Since the memory on the SGI Origin is not physically shared, SGI provides data distribution directives to allow users to specify how data is placed on processors. If no data distribution directives are used, then data is automatically distributed via the “first touch” mechanism [4] which places the data on the processor where it is first used. Since placement of data can significantly affect performance, data distribution directives were used for each test. There are two sets of data distribution directives available on the Origin: the `!$sgi distribute` directives, and the `!$sgi distribute_reshape` directives, see [4]. All tests use the `!$sgi distribute_reshape` in order to ensure that the data is distributed as specified and to ensure repeatability of performance data. On the SGI Origin 2000, data for MPI programs is local to the executing processor unless there is not enough local memory. When there is not enough local memory, memory on the other nodes is used. However, the local memory on the SGI origin used was large enough to hold the data for all the tests run for this paper.

3. TIMING METHODOLOGY

Timing is done by first flushing the caches on all processors by changing the values in the real*8 array flush with the size of the secondary cache. The cache was flushed by adding 0.1 to flush; however, any number small enough to prevent floating point overflow would also work. One could instead flush the cache by simply calling random_number(flush); however this takes significantly more execution time and hence was not used. \( p \) is the number of processors/threads used and \( n_{\text{trial}} \) is the number of timing tests performed in each single test. \( n_{\text{trial}} \) was set to 51 for all tests. The data was filtered as is described in [9].

The following template was used for timing the MPI tests.

\[
\text{integer, parameter :: ncache = \ldots ! number of 8 byte words in the secondary cache}
\]
integer, parameter :: ntrial = 51 ! number of trials timed
real*8 :: time(ntrial), local_time(ntrial)
call random_number(flush)

... do k = 1, ntrial
    flush(l :ncache) = flush(l :ncache) + 0.1d0 ! flush the cache
call mpi_barrier(mpi_comm_world, ierr)
    t1 = mpi_wtime() ! time in seconds
    t2 = mpi_wtime()

    ... MPI code to be timed ... (and assume A(l :n) is calculated here)

    local_time(k) = (mpi_wtime() – t2) – (t2 – t1)
call mpi_barrier(mpi_comm_world, ierr)
    ! prevent loop splitting of the “flush = flush + 0.1d0” statement
    flush(modulo(k, ncache) + 1) = flush(modulo(k, ncache) + 1) + A(modulo(k, n) + 1)
endo
call
mpi_allreduce(local_time, time, ntrial, mpi_real8, mpi_max, mpi_comm_world, ierr)

... print *, A(1!) prevent optimizing compiler from eliminating the calculation of A
Notice that overhead for calling the timer is subtracted when obtaining the time on each process. The maximum of the times measured on each processor are calculated by calling mpi_allreduce and placed in the \texttt{time(1:ntrial)} array. The statement

$$\text{flush(modulo(k,ncache) + 1) = flush(modulo(k,ncache) + 1) + A(modulo(k,n) + 1),}$$

where \(A\) is an array calculated in “the MPI code to be timed”, is needed to prevent some compilers from splitting the cache flushing out from the timing loop. The reason for the second call to mpiBarrier is to prevent an early processor from looping back and begin flushing its cache when it executes the statement

$$\text{flush = flush + 0.1d0}$$

When a processor flushes its cache, it creates high memory traffic. (The size of the secondary cache on the SGI Origin 2000 is large, 8 Mbytes.) The Origin 2000 is made up of nodes with each node consisting of 2 processors sharing a common memory. If one processor on a node finishes early and begins to flush its cache, then this saturates the memory bus making it impossible for the other processor on the same node to send or receive messages during this time.

The following template was used for timing the OpenMP tests.

\begin{verbatim}
    integer, parameter :: ncache = ... ! number of 8 byte words in the secondary cache
    integer, parameter :: ntrial = 51    ! number of trials timed
    real*8 :: time(ntrial), local_time(0:p-1,ntrial)
    call omp_set_num_threads(p)
    call random_number(flush)
    ...
    !$omp parallel default(none) firstprivate(flush) private(t,k) shared(time,local_time)
\end{verbatim}


\begin{verbatim}
... 
    t = timef() ! the first call may be undefined 
    do k = 1, ntrial 
        flush(1:ncache) = flush(1:ncache) + 0.1d0 ! flush the cache
        !$omp barrier 
        t1 = timef() ! time in milliseconds 
        t2 = timef() 
        ...
        !prevent loop splitting of the "flush = flush + 0.d0" statement 
        flush(modulo(k,ncache) + 1) = flush(modulo(k,ncache) + 1) + 
        A(modulo(k,n) + 1) 
        enddo 
        !$omp end parallel 
        time(1:ntrial) = maxval(local_time(0:p-1,1:ntrial),dim=1)*1.0d-3 ! time in seconds 
    ... 
    print *, A(1) !prevent optimizing compiler from eliminating the calculation of A
\end{verbatim}

To use the equivalent timing methodology for the HPF tests, one first must define the extrinsic function hpf_timef() as follows.

\begin{verbatim}
extrinsic (hpf_local) subroutine hpf_timef (t) 
    real*8 :: t(:), timef
    !hpfs distribute t(block) 
    t(1) = timef() ! time in milliseconds 
end subroutine
\end{verbatim}
The extrinsic subroutine hpf_timef is defined this way to allow the measuring of times on each process so the maximum over all processes can be taken, as is done for the OpenMP and MPI timing templates. There is no barrier in HPF. However, an implicit barrier is achieved by the statement

\[
\text{time}(k) = \text{maxval}((t_3 - t_2) - (t_2 - t_1))
\]

This is because \(t_1\), \(t_2\) and \(t_3\) are distributed arrays so maxval performs a reduction with the result broadcast to all processes. The HPF tests are timed using the following template with \(p\) being replaced by number_of_processors().

```fortran
integer, parameter :: ncache = ... ! number of 8 byte words in the secondary cache
integer, parameter :: ntrial = 51 ! number of trials timed
nflush = ncache*number_of_processors() ! number of 8 byte words in all caches
real*8 :: time(ntrial), flush(nflush), t, t1(p), t2(p), t3(p)
interface
  extrinsic (hpf_local) subroutine hpf_timef (t)
  real*8 :: t(:)
  !hpf$
  distribute t(block)
  end subroutine
end interface
!hpf$ processors proc(p)
!hpf$ distribute flush(block) onto proc
!hpf$ distribute (block) onto proc :: t1, t2, t3
  call random_number(flush)
  ...
  do k = 1, ntrial
    forall (i = 1, nflush) flush(i) = flush(i) + 0.1d0 ! flush the cache
    time(k) = maxval((t3 - t2) - (t2 - t1)) ! implicit barrier
```
call hpf_timef(t1)
call hpf_timef(t2)

... HPF code to be timed ...(and assume A(1:n) is calculated here)

call hpf_timef(t3)
time(k) = maxval((t3 - t2) - (t2 - t1)) ! implicit barrier
! prevent loop splitting of the “flush = flush + 0.1d0” statement
flush(modulo(k,ncache) + 1) = flush(modulo(k,ncache) + 1) + A(modulo(k,n) + 1)
enddo

... 
print *, A(1) ! prevent optimizing compiler from eliminating the calculation of A

Figure 1. 51 trials for the HPF copy test with n = 128*1024 and p = 32.

Figure 1 shows the timing data with 51 trials for the copy test (test 8) for the Adaptor HPF compiler with n = 128*1024 and with 32 processes. Notice that there are several “spikes” in
the data with the first spike being the first timing. This kind of behavior is typical of the data found for all tests. The first timing usually was significantly longer than most of the other timings (likely due to the additional setup time required for the first call to subroutines and functions), so the time for the first trial was always removed. The other spikes are probably due to the operating system interrupting the execution of the program. The average of the 50 trials (the first trial is removed) is 12.1 milliseconds, which is much longer than most of the other trials. The authors decided to measure times for each operation by first filtering out the spikes as follows. Compute the median value after the first time trial is removed. All times that are greater than 1.8 times this median value are then removed. The authors consider it to be inappropriate to remove more than 10% of the data. If more than 10% of the data is removed by the above procedure then only the largest 10% of the spikes are removed. Using this procedure, the filtered value for the time for figure 1 is 9.73 milliseconds instead of 12.1 milliseconds. For the 190 timings presented in this paper, only 9 had the maximum of 10% (i.e., 5) spikes removed.

All tests were run one at a time with nobody else using the machine. However, when running the same tests again on the same machine with nobody else using the machine, the performance results reported after filtering would sometimes vary from run to run. Typically, results would vary less than 5% but occasionally they would vary as much as 16%. This timing variation is likely caused by process/thread migration to a different physical processor.

4. TESTS AND PERFORMANCE RESULTS

For all tests $p$ denotes the number of processes/threads used for the test. All floating-point variables and arrays are declared as real*8. All tests were run with $p = 4, 8, 16, 32$ and 64. On the SGI Origin, dynamically allocated arrays sometimes do not perform as well as statically allocated arrays. For all tests, all arrays were statically allocated.
Test 1: Dot Product

This test measures the time to perform the dot product of two vectors that are block distributed across \( p \) processors. In all cases, the initialization of the variable sum to 0.0 was not included in the timing. One way to write the HPF code is:

```hpf
real*8 :: X(n*p), Y(n*p), sum
!hpf$ processors proc(p)
!hpf$ distribute X(block) onto proc
!hpf$ align Y(:) with X(:)
    call random_number(X)
    call random_number(Y)
    sum = 0.d0
    ...
    !hpf$ independent, reduction(sum)
    do i = 1, n*p
       sum = sum + X(i)*Y(i)
    enddo
```

Another way to write the HPF code is to use the `dot_product` parallel intrinsic:

```hpf
real*8 :: X(n*p), Y(n*p), sum
!hpf$ processors proc(p)
!hpf$ distribute X(block) onto proc
!hpf$ align Y(:) with X(:)
    call random_number(X)
    call random_number(Y)
    sum = 0.d0
    ...
    sum = dot_product(X,Y)
```
The OpenMP code for this test is:

```fortran
real*8 :: X(n*p), Y(n*p), sum
!$sgi distribute_reshape X(block), Y(block)
call random_number(X)
call random_number(Y)
sum = 0.d0
!$omp parallel shared(X, Y, sum) private(i)
...  
!$omp do schedule(static) reduction(+:sum)
doi = 1, n*p
    sum = sum + X(i)*Y(i)
endo
e!$omp end do
...  
!$omp end parallel
```

Notice that for the HPF implementation, the variable `sum` is global/shared, so its value either resides on a single processor or is replicated on all processors. (Many HPF implementations replicate `sum`.) For the OpenMP implementation `sum` is shared and hence it resides in the memory of a single node on the SGI Origin. Recall that all data is local/private for MPI codes, so we have a choice of having the final value of `sum` to be on only one processor (i.e. use `mpi_reduce`) or have the final value of `sum` to be on all processors (i.e. use `mpi_allreduce`). The authors chose to implement this test using `mpi_allreduce` since for both OpenMP and HPF, `sum` is available on all processors. The following is the MPI implementation of this test:

```fortran
real*8 :: X(n), Y(n), s, sum
call random_number(X)
```
call random_number(Y)
sum = 0.d0

... 

s = 0.d0
do i = 1, n
    s = s + X(i) * Y(i)
enddo

call mpi_allreduce (s, sum, 1, mpi_real8, mpi_sum, mpi_comm_world, ierror)

Performance results for test 1 are presented in Figure 2 and in Table 1 in the appendix. Notice that Adaptor performed roughly twice as fast as PGHPF. Adaptor should produce the same results for the intrinsic and non-intrinsic versions since Adaptor inlines the dot product intrinsic routine. Figure 2 shows that both the intrinsic and non-intrinsic versions perform about the same for Adaptor up to 32 processors. However, the non-intrinsic version does perform about 25% slower for 64 processors. The MPI and OpenMP versions of this test perform and scale about the same.
Figure 2: Timing results of the dot product test in milliseconds for $n = 16 \times 1024$. 
Test 2: Matrix Times Vector with Row-Blocking

This test measures the time to calculate the matrix-times-vector operation, \( y = y + Ax \), where \( y \) is block distributed, the rows of \( A \) are block distributed, and \( x \) is assumed to be replicated on all \( p \) processors. Notice no communication is required for this test. Thus, one way to write the HPF code for this test is:

```hpff
real*8 :: y(n*p), A(n*p, n), x(n)
!hpf$ processors proc(p)
!hpf$ distribute y(block) onto proc
!hpf$ align A(:,*) with y(:)
!hpf$ align x(:) with A(*,*)
call random_number(A)
call random_number(x)
call random_number(y)

...!
!hpf$ independent, new(j)
do i = 1, n*p
do j = 1, n
y(i) = y(i) + A(i,j) * x(j)
endo
do
endo
```

Another way to write the HPF code is to use the `matmul` parallel intrinsic:

```hpff
real*8 :: y(n*p), A(n*p, n), x(n)
!hpf$ processors proc(p)
!hpf$ distribute y(block) onto proc
!hpf$ align A(:,*) with y(:)
!hpf$ align x(:) with A(*,*)
call random_number(A)
```
The OpenMP code for this test is:

```c
real*8 :: y(n*p), A(n*p, n), x(n)
!$sgi distribute reshape y(block), A(block,*)
call random_number(A)
call random_number(x)
call random_number(y)
!$omp parallel shared(y, A, x) private(i, j)
!$omp do
  do i = 1, n*p
    do j = 1, n
      y(i) = y(i) + A(i,j) * x(j)
    enddo
  enddo
!$omp end do
!$omp end parallel
```

In the above HPF and OpenMP code, parallelization of the i-loop requires the i-loop to be written prior to the j-loop. Notice that then the accesses of A are not stride one. We expect the compiler to interchange loops within each parallel block for stride one access of elements of A.

The MPI code used for this test is:

```c
...
real*8 :: y(n), A(n,n), x(n)
call random_number(A)
call random_number(x)
call random_number(y)

... 
do j = 1, n
   do i = 1, n
      y(i) = y(i) + A(i,j)*x(j)
   enddo
endo
do
endo

In MPI code, x is replicated on all processors and y is block distributed across the p processors. Thus, no communication is required to execute this program. Notice that for the MPI implementation, we can write the loops with stride one accesses for A.

Performance results for this test are presented in figures 3 and 4 and table 2 in the appendix. Notice the poor performance and scalability of the intrinsic implementation for the PGHPF compiler. The non-intrinsic PGHPF implementation is only 4-8 times slower than MPI version, see figure 4. However, both the intrinsic and non-intrinsic Adaptor versions are only 2-4 times slower than the MPI version. Notice increase in times for the MPI version as the number of processors increases even though there is no communication in this test. This may be due to the operating system migrating MPI processes to other nodes during program execution. To help understand why the times increased for the MPI version as the number of processors increased, the MPI version was run on a PC cluster. This increase in time was not observed on the PC cluster.
Figure 3: Timing results of matrix times vectors with row blocking test in milliseconds for n = 256.
Figure 4: Timing results of matrix times vectors with row blocking test in milliseconds for \( n = 256 \).
Test 3: Matrix Times Vector with Column-Blocking

This test measures the time to calculate the matrix-times-vector operation, \( y = y + Ax \), where \( x \) is block distributed, the columns of \( A \) are block distributed, and \( y \) is assumed to be replicated among the \( p \) processors. One way to write the HPF code for this test is:

```hpf
real*8 :: y(n), A(n,n*p), x(n*p)
!hpf$ processors proc(p)
!hpf$ distribute x(block) onto proc
!hpf$ align A(*,:) with x(:)
call random_number(A)
call random_number(x)
call random_number(y)
...
!hpf$ independent, reduction(y)
do j = 1, n*p
do i = 1, n
y(i) = y(i) + A(i,j)*x(j)
enddo
dooo
```

Another way to write the HPF code is to use the `matmul` parallel intrinsic:

```hpf
real*8 :: y(n), A(n, n*p), x(n*p)
!hpf$ processors proc(p)
!hpf$ distribute x(block) onto proc
!hpf$ align A(*,:) with x(:)
call random_number(A)
call random_number(x)
call random_number(y)
...
```
y = y + matmul(A, x)

Notice that the alignment of $A(*,:)$ to $x(,:)$ causes $A$ to be column-block distributed among all processors.

The OpenMP code for this test is:

```plaintext
real*8 :: y(n), A(n, n*p), x(n*p)
!$sgi distribute_reshape A(*,block), x(block)
call random_number(A)
call random_number(x)
call random_number(y)
!$omp parallel shared(y, A, x) private(i)
  . . .
!$omp do schedule(static), reduction(+:y)
  do j = 1, n*p
    do i = 1, n
      y(i) = y(i) + A(i,j)*x(j)
    enddo
  enddo
!$omp end do
!$omp end parallel
```

At the time this study was done, SGI did not have OpenMP 2.0 available and hence reduction for arrays was not implemented. Therefore, this test is implemented as follows:

```plaintext
real*8 :: y(n), A(n, n*p), x(n*p), psum(n)
!$sgi distribute_reshape A(*,block), x(block)
call random_number(A)
```
call random_number(x)
call random_number(y)
!$omp parallel shared(y, A, x) private(psum, i)
    . . .
psum = 0.d0
!$omp do schedule(static)
do j = 1, n*p
    psum(l :n) = psum(l :n) + A(l:n,j)*x(j)
enddo
!$omp enddo
!$omp critical
    y(l :n) = y(l :n) + psum(l :n)
!$omp end critical
    . . .
!$omp end parallel

The MPI implementation is:

real*8 :: y(n), A(n,n), x(n), psum(n), sum(n)
call random_number(A)
call random_number(x)
call random_number(y)
. . .
psum  0.d0
sum = 0.d0
do j = 1, n
do i = 1, n
    psum(i) = psum(i) + A(i,j)*x(j)
enddo
dendo
call mpi_allreduce (psum, sum, n, mpi_real8, mpi_sum, mpi_comm_world, ierror)
y = y + sum

For the same reasons stated in test 1, mpi_allreduce was used instead of mpi_reduce for the MPI implementation.

Performance results for this test are presented in figure 5 and 6 and in table 3 in the appendix. Notice that poor performance and scalability of the intrinsic implementation with Adaptor. However, the non-intrinsic implementation with Adaptor performs and scales well compared with the MPI version, see figure 6. Notice that the OpenMP version performs roughly as well as the MPI version up to 16 processors, but does not scale nearly as well for 32 and 64 processors. The poor performance of the OpenMP is likely due to the fact that the reduction clause has not yet been implemented for arrays and the updating of y is done in a critical region.
Figure 5: Timing results of matrix times vectors with column blocking test in milliseconds for \( n = 256 \).
Figure 6: Timing results of matrix times vectors with column blocking test in milliseconds for $n = 256$. 
Test 4: Matrix-Matrix Multiplication

This test compares the performance of HPF, OpenMP and MPI implementations for the product of two matrices $C = C + AB$. For the HPF implementation $A$ is an $n \times (n*p)$ array, $B$ is an $(n*p) \times n$ array, and $C$ is an $n \times n$ array. One way to write the HPF code for this test is:

```hpf
real*8 :: C(n,n), A(n,n*p), B(n*p,n)
!hpf$ processors proc(p)
!hpf$ distribute A(*,block) onto proc
!hpf$ distribute B(block,*) onto proc
call random_number(A)
call random_number(B)
call random_number(C)

!hpf$ independent, reduction(C)
do k = 1, n*p
do i = 1, n
do j = 1, n
  C(i,j) = C(i,j) + A(i,k) * B(k,j)
enddo
doto
enddo
enddo
```

Another way to write the HPF code is to use the **matmul** parallel intrinsic:

```hpf
real*8 :: C(n,n), A(n,n*p), B(n*p,n)
!hpf$ processors proc(p)
!hpf$ distribute A(*,block) onto proc
!hpf$ distribute B(block,*) onto proc
call random_number(A)
call random_number(B)
call random_number(C)
```
\[ C = C + \text{matmul}(A, B) \]

The OpenMP version of this test is:

```fortran
real*8 :: A(n,n*p), B(n*p,n), C(n,n)
!$sgi distribute_reshape A(*,block), B(block,*)
call random_number(A)
call random_number(B)
call random_number(C)
!$omp parallel shared(A,B,C) private(i,j,k)
!$omp do reduction(+: C)
dok=l,n*p
do i = 1, n
do j = 1, n
\quad C(i,j) = C(i,j) + A(i,k) \times B(k,j)
enddo
enddo
!$omp end do
!$omp end parallel
```

Since reduction for arrays was yet not available on the SGI OpenMP Fortran compiler, this test was written as follows.

```fortran
real*8 :: A(n,n*p), B(n*p,n), C(n,n), local_C(n,n)
!$sgi distribute_reshape A(*,block), B(block,*)
call random_number(A)
```
call random_number(B)
call random_number(C)
!$omp parallel shared(A,B,C) private(i,j,k)

... 
local_C = 0.d0

!$omp do
  do k = 1, n*p
    do i = 1, n
      do j = 1, n
        local_C(i,j) = local_C(i,j) + A(i,k) * B(k,j)
      enddo
    enddo
  enddo
!$omp end do

!$omp critical
  C = C + local_C
!$omp end critical

... 
!$omp end parallel

The MPI version of this test is:

real*8 :: A(n,n), B(n,n), local_C(n,n), C(n,n)
call random_number(A)
call random_number(B)
call random_number(C)

... 
if (rank .eq. 0) then
  local_C = C
else
local_C = 0.0
endif
do k = 1, n
do i = 1, n
do j = 1, n
    local_C(i,j) = local_C(i,j) + A(i,k) * B(k,j)
endo
dendo
dendo
call mpi_allreduce(local_C,C,n*n,mpi_real8,mpi_sum,mpi_comm_world,ierror)

For the same reasons stated in test 1, mpi_allreduce was used instead of mpi_reduce for the MPI implementation.

Performance results for this test are presented in figures 7 and 8 and in table 4 in the appendix. Adaptor performance data is not available because when n = 128, memory overflow occurred on the Origin 2000. Notice that the intrinsic implementation for the PGHPF compiler did not perform nor scale nearly as well as non-intrinsic PGHPF implementation. The MPI and OpenMP versions of this test perform and scale about the same. The non-intrinsic PGHPF implementation is 5-6 times slower than both MPI and OpenMP versions. The poor scalability of the OpenMP version is likely due to the fact that the reduction clause for arrays is not yet implemented by SGI so the updating of C is done in a critical region. For both the OpenMP and MPI implementations, the compiler recognizes the matrix-times-matrix pattern and inlines efficient code to perform this operation. SGI’s Fortran compiler does recognize the matrix-matrix code generated by ADAPTOR, but the code generated by ADAPTOR is complicated and SGI’s compiler does not appear to be as efficient as the inlined code for the OpenMP and MPI versions.
Figure 7: Timing results of matrix times matrix test in milliseconds for $n = 128$. 
Figure 8: Timing results of matrix times matrix test in milliseconds for $n = 128$. 
Test 5: Matrix Transpose

Suppose A and B are real*8 arrays of dimension n*p, that are column-block distributed among p processors. This test measures the time to calculate B = transpose(A). One way to implement this transpose operation using HPF is to use the parallelized transpose intrinsic as follows:

```fortran
real*8 :: A(n*p, n*p), B(n*p, n*p)
!hpf$ processors proc(p)
!hpf$ distribute A(*,block) onto proc
!hpf$ align B(:,:) with A(:,:)
call random_number(A)
...
B = transpose(A)
```

Another way to write the HPF version of this transpose operation is:

```fortran
real*8 :: A(n*p, n*p), B(n*p, n*p)
!hpf$ processors proc(p)
!hpf$ distribute A(*,block) onto proc
!hpf$ align B(:,:) with A(:,:)
call random_number(A)
...
!hpf$ independent, new(i)
do j = 1, n*p
do i = 1, n*p
B(j,i) = A(i,j)
enddo
enddo
```

The equivalent OpenMP implementation is:
In the above OpenMP code, either the i-loop or the j-loop can be chosen to be the outer loop. In both cases there will be stride 1 loads from memory or stride 1 stores to memory, but not both. Tests show that on the SGI Origin the performance of the j-loop as the outer loop was faster, so this was used.

An MPI implementation for this transpose operation is:

```fortran
real*8 :: A(n*p, n), At(n, n*p), B(n*p, n), Btemp(n, n*p)
call random_number(A)

At = transpose(A)  ! perform a local transpose
! distribute local transposes & store in Btemp
call mpi_alltoall(At, n*n, mpi_real8, Btemp, n*n, mpi_real8, mpi_comm_world, ierr)
! copy Btemp to B
```
do k = 0, p-1
   do j = 1, n
      do i = 1, n
         B(k*n+i, j) = Btemp(i, k*n + j)
      enddo
   enddo
enddo

The following picture shows the MPI code implementation for 4 processors:

The authors considered implementing the MPI version of this test using MPI derived types as was done in [14, p222] for block-block distributed arrays. However, this example uses mpi_alltoallw which is not currently available on the SGI Origin. If one assumes that A and B are row-blocked instead of column-blocked, then one can use MPI derived types to implement the transpose. This is done as follows:

real*8 :: A(n, n*p), B(n, n*p)
integer :: disp(0:n*n-1), bls(0:n*n-1), transpose_type
call random_number(A)
do i = 0, n-1
do j = 0, n-1
   disp(i*n+j) = j*n + i
   bls(i*n + j) = 1
endo
dendo
call mpi_type_indexed(n*n,bls,disp,mpi_real8,transpose_type,ierr)
call mpi_type_commit(transpose_type,ierr)
   ...
call mpi_alltoall(A, n*n, mpi_real8, B, 1, transpose_type, mpi_comm_world, ierr)
   ...
call mpi_type_free(transpose_type,ierr)

Unfortunately when this test was run, there was a bug in the SGI implementation of mpi_alltoall when using the MPI derived types above. Thus, it was not possible to use this MPI implementation, so the previous MPI implementation was used.

Performance results for this test are presented in figures 9 and in table 5 in the appendix. Both HPF compilers scale and perform about the same as the MPI version. The OpenMP version performs and scales best for this test. If the authors had been able to use MPI derived data types to implement this test, the MPI implementation may have performed and scaled as well as the OpenMP implementation.
Figure 9: Timing results of matrix transpose test in milliseconds for $n = 64$. 
Test 6: Stencil Computation

The following test is the standard 5-point stencil calculation [11] used when solving Laplace’s equation in two dimensions. In this test, both $A(n, n*p)$ and $B(n, n*p)$ are distributed among $p$ processors. The HPF version is implemented as follows:

```hpf
real*8 :: A(n, n*p), B(n, n*p)
!hpf$ processors proc(p)
!hpf$ distribute A(*,block) onto proc
!hpf$ align B(:, :) with A(:, :)
call random_number(A)

B(1,1:n*p) = 0.d0
B(n,1:n*p) = 0.d0
!hpf$ independent, new(i)
do j = 1, n*p
  if ((j == 1) .or. (j == n*p)) then
    B(1:n,j) = 0.d0
  else
    do i = 2, n-1
      B(i, j) = 0.25d0*(A(i-1, j) + A(i+1,j) + A(i, j-1) + A(i, j + 1))
    enddo
  endif
endo
```

The OpenMP version is implemented as:

```omp
real*8 :: A(n, n*p), B(n, n*p)
!$sgi distribute_reshape A(*,block),B(*,block)
call random_number(A)
```
!$omp parallel shared(A, B) private(i,j)
    ...
    B(1,1:n*p) = 0.d0
    B(n,1:n*p) = 0.d0
 !$omp do schedule(static)
    do j = 1, n*p
        if ((j == 1) .or. (j == n*p)) then
            B(1:n,j) = 0.d0
        else
            do i = 2, n-1
                B(i, j) = 0.25d0*(A(i-1, j) + A(i+1,j) + A(i, j-1) + A(i, j+1))
            enddo
        endif
    enddo
 !$omp end do
...
 !$omp end parallel

The MPI version is implemented as:

real*8 :: A(n, 0:n+1), B(n,n)

... if (rank > 0) then
    call mpi_send(A(1,1), n, mpi_real8, rank-1,0,mpi_comm_world,ierr)
    call mpi_recv(A(1,0), n, mpi_real8, rank-1,1, mpi_comm_world,status,ierr)
endif
if (rank < p-1) then
    call mpi_send(A(1,n),n, mpi_real8, rank+1,1, mpi_comm_world,ierr)
    call mpi_recv(A(1,n+1), n, mpi_real8, rank+1,0,mpi_comm_world,status,ierr)
endif
B(1, 1:n) = 0.d0
B(1:n, 1) = 0.d0
do j = 1, n
    if ((rank == 0 .and. j == 1) .or. (rank == p-1 .and. j == n)) then
        B(1:n, j) = 0.d0
    else
        do i = 2, n-1
            B(i, j) = 0.25d0*(A(i-1, j) + A(i+1, j) + A(i, j-1) + A(i, j+1))
        enddo
    endif
enddo

In the MPI code for j = 1, 2, ..., p-2, p-1, processor of rank j sends column 1 to column n + 1 of A on processor rank j -1. For j = 0, 1, ..., p-2, processor of rank j sends column n of A to column 0 of A in the processor with rank j + 1.

Performance results for this test are presented in figure 10 and in Table 6 in the appendix. Notice that poor performance and scalability of the OpenMP. SGI has been informed of this performance defect of their compiler for this test. The Adaptor, PGHPF, and MPI implementations of this test perform and scale roughly the same.
Figure 10: Timing results of Stencil test in milliseconds for $n = 256$. 
Test 7: LU NAS BENCHMARK

In this section, the performance of the HPF, OpenMP and MPI versions of the LU NAS parallel benchmark will be evaluated. The NAS Parallel Benchmarks (NPB) [7] are a set of 8 programs derived from computational fluid dynamics (CFD) applications, and designed to help evaluate the performance of parallel supercomputers. For this study, we arbitrarily chose the LU benchmark to determine the relative performance of the HPF, OpenMP and MPI implementations on a more complex code than the previous tests. The LU benchmark is available with HPF, OpenMP and MPI versions. The authors used the versions without modification. There are five test cases with each of these benchmarks: classes A(small), B(medium), C(large), S(sample) and W(workstation). The authors chose the middle problem size B due to limited testing time available.

Performance results for this test are presented in figure 12 and in table 7 in the appendix. Notice that poor performance and scalability of Adaptor. The PGHPF, MPI and OpenMP versions of this test perform and scale about the same. This test is more complex than all the other tests in this paper and for and HPF compiler to perform and scale nearly as well as the OpenMP and MPI versions is excellent.
Figure 11: Timing results of LU benchmark in seconds for Class B.
The rest of the tests are commonly found in MPI programs but may also be used when writing HPF and OpenMP programs.

**Test 8: Copy**

Assume real, one-dimensional arrays $A$ and $B$ are block distributed across $p$ processors. This test measures the time to copy $B$ to $A$. Observe that no communication between processors is required for this test. The HPF code used is:

```hpfn
real*8 :: A(n*p), B(n*p)
!hpf$ processors proc(p)
!hpf$ distribute B(block) onto proc
!hpf$ align A(:,) with B(:,)
    call random_number(B)
    
    !hpf$ independent
    do i = 1, n*p
        A(i) = B(i)
    enddo
```

The OpenMP code for this test is:

```omp
real*8 :: A(n*p), B(n*p)
!$sgi distribute_reshape A(block), B(block)
    call random_number(B)

!$omp parallel shared(A, B) private(i)
    do i = 1, n*p
        A(i) = B(i)
    enddo
```
A(i) = B(i)
enddo
!$omp end do
...
!$omp end parallel

The MPI code for this test is:

real*8 :: A(n), B(n)
call random_number(B)
...
A = B

Performance results for this test are presented in figure 12 and in table 8 in the appendix. The PGHPF, MPI and OpenMP versions of this test perform and scale about the same and that they scale better than Adaptor.
Figure 12: Timing results of copy test in milliseconds for $n = 128 \times 1024$. 
Test 9: Array Replication

This test measures the time required to take array $A(1:n)$ and replicate it on all processors by copying it into array $B(1:n,1:p)$. One way to implement this in HPF is as follows:

```hpf
real*8 :: A(n), B(n,p)
!hpf$ processors proc(p)
!hpf$ distribute B(*,block) onto proc
!hpf$ align A(:) with B(:,1)
call random_number(A)
...
!hpf$ independent
do  i = 1, p
    B(1:n,i) = A(1:n)
endo
```

Another way to implement this in HPF is to use `spread` parallel intrinsic:

```hpf
real*8 :: A(n), B(n,p)
!hpf$ processors proc(p)
!hpf$ distribute B(*,block) onto proc
!hpf$ align A(:) with B(:,1)
call random_number(A)
...
B = spread(A, dim = 2, ncopies = p)
```

The OpenMP code for this test is:

```hpf
real*8 :: A(n), B(n,p)
!$sgi distribute_reshape B(*,block)
```
call random_number(A)  
!$omp parallel shared(A,B) private(i)  
  
  !$omp do schedule(static)  
  do i = 1, p  
    B(1:n,i) = A(1:n)  
  enddo  
$omp end do  
  
$omp end parallel

But this can be easily implemented in MPI with mpi_bcast and then copy A to B on each processor.

real*8 :: A(n), B(n)  
call random_number(A)  
  
  call mpi_bcast(A, n, mpi_real8, 0, mpi_comm_world, ierror)  
do i = 1, p  
    B(1:n,i) = A(1:n)  
  enddo

Performance results for this test are presented in figure 13 and in table 9 in the appendix. Notice that poor performance and scalability of both the intrinsic and non-intrinsic implementations of PGHPF. Both implementations for Adaptor perform and scale about the same as the MPI implementation.
Figure 13: Timing results of array replication test in milliseconds for $n = 128 \times 1024$. 
5. CONCLUSIONS

The purpose of this paper was to investigate the scalability and performance of 9 HPF programs using two HPF compilers and to compare their performance with equivalent OpenMP and MPI programs on an SGI Origin 2000. Test results were mixed but in all cases either the MPI or OpenMP implementations performed and scaled the best. For most cases, at least one of the HPF compilers performed and scaled about the same as either the MPI or OpenMP implementations. The most complex test used in this study was the NAS Parallel LU Benchmark [5]. For this test, one of the HPF compilers performed and scaled about the same as both the OpenMP and MPI implementations. The performance and scalability of an HPF (any) compiler depends on how well it has been implemented. When writing a parallel program, the programmer is limited to expressing the parallelism by the language being used. The results of this study suggest that the HPF syntax does not limit performance for the tests considered for this study. However, the performance of the two HPF compilers used for this study differed significantly for some of the tests. It is interesting to observe how similar the HPF code is to the OpenMP code for each of the tests.
CHAPTER 3. GENERAL CONCLUSIONS

The purpose of this paper was to investigate the scalability and performance of 9 HPF programs using two HPF compilers and to compare their performance with equivalent OpenMP and MPI programs on an SGI Origin 2000. Test results were mixed but in all cases either the MPI or OpenMP implementations performed and scaled the best. For most cases, at least one of the HPF compilers performed and scaled about the same as either the MPI or OpenMP implementations. The most complex test used in this study was the NAS Parallel LU Benchmark [5]. For this test, one of the HPF compilers performed and scaled about the same as both the OpenMP and MPI implementations. The performance and scalability of an HPF (any) compiler depends on how well it has been implemented. When writing a parallel program, the programmer is limited to expressing the parallelism by the language being used. The results of this study suggest that the HPF syntax does not limit performance for the tests considered for this study. However, the performance of the two HPF compilers used for this study differed significantly for some of the tests. It is interesting to observe how similar the HPF code is to the OpenMP code for each of the tests.
BIBLIOGRAPHY


2 The OpenMP ARB and KAI Software, a Division of Intel Americas, Inc. (KAI),
   *OpenMP specification, version 2.0*, http://www.openmp.org/ (date accessed: June 2001)


5. Numerical Aerospace Simulation, *NAS Benchmark*,
   http://www.nas.nasa.gov/Software/NPB/ (date accessed: July 2001)


8. Glenn R. Luecke, Ying Li, *Evaluating the Performance of High Performance Fortran Compilers on a NEC Cenju-4, a Cray T3E, an IBM SP and an SGI Origin 2000*


11. Advanced Computing Laboratory in Los Alamos National Laboratory, *Stencil test*,

12. Thomas Brandes, Institute Algorithms and Scientific Computer in Gesellschaft f
    Mathematik und Datenverarbeitung mbH Bonn (literally: Society for Mathematics
    (date accessed: August 2001)


APPENDIX

Table 1: Timing Results of the dot product test in milliseconds for n = 16 \times 1024.

<table>
<thead>
<tr>
<th>P</th>
<th>MPI</th>
<th>OpenMP</th>
<th>PGHPF non-intrinsic</th>
<th>PGHPF intrinsic</th>
<th>ADAPTOR non-intrinsic</th>
<th>ADAPTOR intrinsic</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6.15E-01</td>
<td>7.60E-01</td>
<td>2.06E+00</td>
<td>2.05E+00</td>
<td>1.07E+00</td>
<td>1.12E+00</td>
</tr>
<tr>
<td>8</td>
<td>7.53E-01</td>
<td>7.10E-01</td>
<td>2.08E+00</td>
<td>2.13E+00</td>
<td>1.20E+00</td>
<td>1.22E+00</td>
</tr>
<tr>
<td>16</td>
<td>6.33E-01</td>
<td>7.88E-01</td>
<td>2.17E+00</td>
<td>2.18E+00</td>
<td>1.31E+00</td>
<td>1.26E+00</td>
</tr>
<tr>
<td>32</td>
<td>1.24E+00</td>
<td>9.50E-01</td>
<td>2.29E+00</td>
<td>2.36E+00</td>
<td>1.37E+00</td>
<td>1.53E+00</td>
</tr>
<tr>
<td>64</td>
<td>1.62E+00</td>
<td>1.57E+00</td>
<td>2.58E+00</td>
<td>2.69E+00</td>
<td>2.21E+00</td>
<td>1.58E+00</td>
</tr>
</tbody>
</table>

Table 2: Timing results of matrix times vectors with row blocking test in milliseconds for n = 256.

<table>
<thead>
<tr>
<th>P</th>
<th>MPI</th>
<th>OpenMP</th>
<th>PGHPF non-intrinsic</th>
<th>PGHPF intrinsic</th>
<th>ADAPTOR non-intrinsic</th>
<th>ADAPTOR intrinsic</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3.50E-01</td>
<td>1.02E+00</td>
<td>3.91E+00</td>
<td>4.84E+00</td>
<td>1.50E+00</td>
<td>1.70E+00</td>
</tr>
<tr>
<td>8</td>
<td>3.35E-01</td>
<td>9.30E-01</td>
<td>4.09E+00</td>
<td>5.48E+00</td>
<td>1.73E+00</td>
<td>1.92E+00</td>
</tr>
<tr>
<td>16</td>
<td>4.73E-01</td>
<td>9.10E-01</td>
<td>4.11E+00</td>
<td>9.92E+00</td>
<td>2.13E+00</td>
<td>1.80E+00</td>
</tr>
<tr>
<td>32</td>
<td>7.86E-01</td>
<td>1.02E+00</td>
<td>4.23E+00</td>
<td>2.11E+01</td>
<td>2.42E+00</td>
<td>2.29E+00</td>
</tr>
<tr>
<td>64</td>
<td>1.17E+00</td>
<td>1.31E+00</td>
<td>4.31E+00</td>
<td>8.26E+01</td>
<td>2.61E+00</td>
<td>2.29E+00</td>
</tr>
</tbody>
</table>

Table 3: Timing results of matrix times vectors with column blocking test in milliseconds for n = 256.

<table>
<thead>
<tr>
<th>P</th>
<th>MPI</th>
<th>OpenMP</th>
<th>PGHPF non-intrinsic</th>
<th>PGHPF intrinsic</th>
<th>ADAPTOR non-intrinsic</th>
<th>ADAPTOR intrinsic</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1.17E+00</td>
<td>9.34E-01</td>
<td>4.00E+00</td>
<td>4.63E+00</td>
<td>1.82E+00</td>
<td>1.87E+01</td>
</tr>
<tr>
<td>8</td>
<td>1.63E+00</td>
<td>1.03E+00</td>
<td>4.23E+00</td>
<td>5.00E+00</td>
<td>2.22E+00</td>
<td>2.43E+01</td>
</tr>
<tr>
<td>16</td>
<td>1.47E+00</td>
<td>1.53E+00</td>
<td>4.39E+00</td>
<td>5.17E+00</td>
<td>2.44E+00</td>
<td>3.57E+01</td>
</tr>
<tr>
<td>32</td>
<td>1.93E+00</td>
<td>3.20E+00</td>
<td>4.78E+00</td>
<td>5.67E+00</td>
<td>3.46E+00</td>
<td>7.06E+01</td>
</tr>
<tr>
<td>64</td>
<td>3.23E+00</td>
<td>8.71E+00</td>
<td>5.20E+00</td>
<td>5.91E+00</td>
<td>3.46E+00</td>
<td>2.32E+02</td>
</tr>
</tbody>
</table>
Table 4: Timing results of matrix times matrix test in milliseconds for \( n = 128 \).

<table>
<thead>
<tr>
<th>P</th>
<th>MPI</th>
<th>OpenMP</th>
<th>PGHPF non-intrinsic</th>
<th>PGHPF intrinsic</th>
<th>ADAPTOR non-intrinsic</th>
<th>ADAPTOR intrinsic</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1.74E+01</td>
<td>1.39E+01</td>
<td>1.10E+02</td>
<td>2.54E+02</td>
<td>na</td>
<td>na</td>
</tr>
<tr>
<td>8</td>
<td>2.42E+01</td>
<td>1.82E+01</td>
<td>1.18E+02</td>
<td>3.58E+02</td>
<td>na</td>
<td>na</td>
</tr>
<tr>
<td>16</td>
<td>2.41E+01</td>
<td>3.08E+01</td>
<td>1.32E+02</td>
<td>9.35E+02</td>
<td>na</td>
<td>na</td>
</tr>
<tr>
<td>32</td>
<td>3.47E+01</td>
<td>5.69E+01</td>
<td>1.57E+02</td>
<td>4.44E+03</td>
<td>na</td>
<td>na</td>
</tr>
<tr>
<td>64</td>
<td>4.00E+01</td>
<td>1.05E+02</td>
<td>2.23E+02</td>
<td>2.00E+04</td>
<td>na</td>
<td>na</td>
</tr>
</tbody>
</table>

Table 5: Timing results of matrix transpose test in milliseconds for \( n = 64 \).

<table>
<thead>
<tr>
<th>P</th>
<th>MPI</th>
<th>OpenMP</th>
<th>PGHPF non-intrinsic</th>
<th>PGHPF intrinsic</th>
<th>ADAPTOR non-intrinsic</th>
<th>ADAPTOR intrinsic</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3.78E+00</td>
<td>1.54E+00</td>
<td>6.22E+00</td>
<td>9.38E+00</td>
<td>4.24E+00</td>
<td>4.13E+00</td>
</tr>
<tr>
<td>8</td>
<td>8.57E+00</td>
<td>3.63E+00</td>
<td>1.51E+01</td>
<td>2.19E+01</td>
<td>1.03E+01</td>
<td>1.15E+01</td>
</tr>
<tr>
<td>16</td>
<td>2.45E+01</td>
<td>1.00E+01</td>
<td>2.84E+01</td>
<td>4.61E+01</td>
<td>2.81E+01</td>
<td>2.85E+01</td>
</tr>
<tr>
<td>32</td>
<td>7.98E+01</td>
<td>3.94E+01</td>
<td>6.99E+01</td>
<td>1.12E+02</td>
<td>7.25E+01</td>
<td>7.71E+01</td>
</tr>
<tr>
<td>64</td>
<td>2.32E+02</td>
<td>1.33E+02</td>
<td>2.07E+02</td>
<td>2.99E+02</td>
<td>2.59E+02</td>
<td>2.42E+02</td>
</tr>
</tbody>
</table>

Table 6: Timing results of stencil test in milliseconds for \( n = 256 \).

<table>
<thead>
<tr>
<th>P</th>
<th>MPI</th>
<th>OpenMP</th>
<th>PGHPF</th>
<th>ADAPTOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2.23E+00</td>
<td>8.29E+00</td>
<td>1.24E+01</td>
<td>4.27E+00</td>
</tr>
<tr>
<td>8</td>
<td>2.53E+00</td>
<td>1.43E+01</td>
<td>1.45E+01</td>
<td>4.50E+00</td>
</tr>
<tr>
<td>16</td>
<td>3.17E+00</td>
<td>2.99E+01</td>
<td>1.56E+01</td>
<td>4.51E+00</td>
</tr>
<tr>
<td>32</td>
<td>4.85E+00</td>
<td>6.66E+01</td>
<td>2.14E+01</td>
<td>4.72E+00</td>
</tr>
<tr>
<td>64</td>
<td>9.76E+00</td>
<td>1.75E+02</td>
<td>1.96E+01</td>
<td>1.01E+01</td>
</tr>
</tbody>
</table>

Table 7: Timing results of LU benchmark in seconds for Class B.

<table>
<thead>
<tr>
<th>P</th>
<th>MPI</th>
<th>OpenMP</th>
<th>PGHPF</th>
<th>ADAPTOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1.23E+03</td>
<td>1.75E+03</td>
<td>1.90E+03</td>
<td>7.82E+03</td>
</tr>
<tr>
<td>8</td>
<td>6.47E+02</td>
<td>9.26E+02</td>
<td>1.22E+03</td>
<td>7.69E+03</td>
</tr>
<tr>
<td>16</td>
<td>3.27E+02</td>
<td>5.20E+02</td>
<td>7.50E+02</td>
<td>7.67E+03</td>
</tr>
<tr>
<td>32</td>
<td>1.46E+02</td>
<td>3.23E+02</td>
<td>4.89E+02</td>
<td>7.53E+03</td>
</tr>
<tr>
<td>64</td>
<td>8.90E+01</td>
<td>1.95E+02</td>
<td>2.90E+02</td>
<td>7.60E+03</td>
</tr>
</tbody>
</table>
Table 8: Timing results of copy test in milliseconds for $n = 128 \times 1024$.

<table>
<thead>
<tr>
<th>$P$</th>
<th>MPI</th>
<th>OpenMP</th>
<th>PGHPF</th>
<th>ADAPTOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1.08E+01</td>
<td>1.22E+01</td>
<td>1.10E+01</td>
<td>1.79E+01</td>
</tr>
<tr>
<td>8</td>
<td>1.10E+01</td>
<td>1.29E+01</td>
<td>1.10E+01</td>
<td>1.79E+01</td>
</tr>
<tr>
<td>16</td>
<td>1.21E+01</td>
<td>1.01E+01</td>
<td>1.11E+01</td>
<td>1.83E+01</td>
</tr>
<tr>
<td>32</td>
<td>1.23E+01</td>
<td>1.48E+01</td>
<td>1.71E+01</td>
<td>2.77E+01</td>
</tr>
<tr>
<td>64</td>
<td>1.29E+01</td>
<td>1.44E+01</td>
<td>1.85E+01</td>
<td>2.50E+01</td>
</tr>
</tbody>
</table>

Table 9: Timing results of array replication test in milliseconds for $n = 128 \times 1024$.

<table>
<thead>
<tr>
<th>$P$</th>
<th>MPI</th>
<th>OpenMP</th>
<th>PGHPF</th>
<th>ADAPTOR (spread)</th>
<th>ADAPTOR (spread)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3.13E+01</td>
<td>1.26E+01</td>
<td>4.50E+01</td>
<td>4.55E+01</td>
<td>4.56E+01</td>
</tr>
<tr>
<td>8</td>
<td>4.45E+01</td>
<td>1.31E+01</td>
<td>7.51E+01</td>
<td>6.68E+01</td>
<td>5.48E+01</td>
</tr>
<tr>
<td>16</td>
<td>5.63E+01</td>
<td>2.27E+01</td>
<td>1.33E+02</td>
<td>1.28E+02</td>
<td>6.57E+01</td>
</tr>
<tr>
<td>32</td>
<td>7.00E+01</td>
<td>5.06E+01</td>
<td>2.65E+02</td>
<td>2.51E+02</td>
<td>7.59E+01</td>
</tr>
<tr>
<td>64</td>
<td>8.47E+01</td>
<td>7.24E+01</td>
<td>5.44E+02</td>
<td>5.25E+02</td>
<td>9.61E+01</td>
</tr>
</tbody>
</table>
ACKNOWLEDGEMENTS

I would like to thank my advisor Dr. Glenn R. Luecke, for his guidance, patience, and support throughout this research. I would also like to thank my other committee members, Dr. David Fernandez-Baca, and Dr. Shashi Gadia, for their comments and advises to this work.

Special thanks to the staff members in the High Performance Computing Group: Dr. James Coyle, Dr. James Hoekstra, and Dr. Marina Kraeva, for their help during this research work. I would also like to thank Thomas E. Griffith from SGI, whose knowledge and experiences helped me out of my struggling with the SGI Origin 2000.