HPC Fortran Compilers

Lee Higbie

Arctic Region Supercomputing Center

ABSTRACT:
Fortran, a forgotten language outside of HPC, continues to be the most important for computationally intensive problems. This paper provides a detailed investigation of several of the Fortran compilers that are most heavily used in supercomputing. The investigation looks at the performance of the compilers on small code snippets. During the investigation, some problems with using PAPI on small code blocks were uncovered; these are also discussed.

KEYWORDS: Compiler Comparison, PAPI, Timing, Optimization, Performance

Introduction
Fortran is the dominant language for numeric supercomputer applications such as weather, climate and aircraft modeling. The applications often use millions of compute hours per year. For example, The Arctic Region Supercomputing Center allocates approximately 16 core-years\(^1\) per year for weather forecasting for the state of Alaska. Because of the execution expense of these models, Fortran compilers tend to offer extensive "code optimization" options. Major users often prefer to spend their time working in their field of specialization instead of learning about "another compiler" or a new language construct. The result is that many programs are compiled with default optimization or \(-Ox\), for the biggest \(x\) mentioned in the first screen or two of the compiler man page. If there is an option with "fast," that may be added.

With this background, it seemed reasonable to compare the compilers available on our machines. Most Fortran compiler performance studies have evaluated compiler performance based on the execution time of a few, mostly large programs.[1-4] This study has approached the analysis of Fortran compilers from a different angle, looking at the performance on a large number of very small code blocks. Each code snippet used in this study is one or more loops, only a few of which have more than five lines of code. We think that the performance of one or more major applications is probably a better indication of useful compiler quality than performance of small blocks of code, but such performance measures provide little or no help to the compiler-optimization writers or to the analysts programming an application or tuning one to execute more efficiently.

We think our approach will be of greater interest to compiler and optimizer creators and code developers because we highlight specific well or poorly compiled source code structures. The loops are small enough that optimizer writers can evaluate the compiler's internal optimization operation and people doing code development or optimization can see the types of structure that prevent compilers from producing good code.

This paper used all the Fortran 90/95 compilers on the XT5 supercomputers at the Arctic Region Supercomputing Center at the University of Alaska Fairbanks: those from Cray, the Gnu Project, PathScale and The Portland Group.\(^2\) The study consisted of compar-

---

1 Because the CPU chips in supercomputers have multiple computational "cores," this term is usually used to describe the basic computational resource on supercomputers. Nodes, which typically have between 4 and 32 cores and between 1 and 8 CPU chips, are the basic unit of allocation on many large systems.

2 Terminology for compilers mentioned in this paper:
"Cray" = "Cray Fortran," version 7.0.3
"Gnu" = "The gcc Fortran 90 compiler," gfortran, version 4.1.2(prerelease)
"PathScale" = "Qlogic PathScale Compiler Suite," versions 3.2
ing the execution speed of several hundred code snippets on each compiler. Some statistics on the relative performance of the compiled code are included in the figures.

The results were surprising for four reasons.

1. The execution time, as reported by the PAPI\(^4\) `papi\_get\_real\_cyc`, varied widely from one run to the next.

2. Changing the optimization from normal to high often did little to improve the performance on these small code blocks.

3. For each compiler tested, there were some snippets that ran substantially slower with “high” optimization.

**Caveats**

Any study like this is a snapshot of specific compiler versions on specific code.\(^3\) Making general inferences about compiler performance is unlikely to be useful. At a different time, i.e., with other compiler versions, the relative results are likely to change.

We did not attempt to comprehensively test compiler options. Only a few switches or options were tried and they were only tried on our small code snippets. How well a compiler does on a large program, specifically on your own program, is probably the only meaningful-to-you compiler metric.

**Background**

Each compiler tested has many, often dozens of “optimization” options.\(^3\) Facing the complexity of selecting the way to compile, we suspect most production program users try high optimization, and if that doesn’t work (the program doesn’t run or runs inaccurately), back off to the default optimization.

ARSC’s XT5s have the most common supercomputer architecture with hundreds of nodes, each with two or more i386 family processor chips. Thus, we do not think they pose any special compilation difficulties the way a more unusual architecture, such as vector or cell processors, might. ARSC has two XT5s, a small one named Ognip and a large one named Pingo.\(^6\) These tests were run standalone on nodes with 8 CPU cores.

**Test Procedure**

The program was compiled using each of the compilers and a few of the most common options:

- `O2`, or the default optimization level for each compiler
- `-fast` (or its equivalent), an option that is supposed to produce faster-running code. Because Cray

5 We doubt that any significant program has been or ever will be performance optimized. Requesting “optimization” from a compiler means requesting that it generate code that runs faster (or sometimes takes less space). “Hey, make it run a little faster” just doesn’t have the nice ring of “optimize.”

6 A Pingo is a large frost heave, typically a kilometer across and dozens of meters high. An Ognip is a Pingo that has collapsed (melted interior). Both words are derived from Inuit. In Alaska, some people (incorrectly?) call the smaller frost heaves common here “pingos.” (True? classic?) Pingos are common in northern Canada, but rare in Alaska. Pingos form in areas of permafrost.

---

\(^4\) Acronym Decoding

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARSC</td>
<td>Arctic Region Supercomputing Center</td>
</tr>
<tr>
<td>CNL</td>
<td>Compute node Linux</td>
</tr>
<tr>
<td>HPC</td>
<td>High performance computing</td>
</tr>
<tr>
<td>MFLOPS</td>
<td>Mega-floating point operations/second</td>
</tr>
<tr>
<td>MOPS</td>
<td>Mega-operations/second</td>
</tr>
<tr>
<td>PAPI</td>
<td>Performance application programming interface</td>
</tr>
</tbody>
</table>
recommends the -tp barcelona switch for the Portland Group compiler and PathScale has a switch, -ffast-math, that looked useful; we used both options.

Because code quality is difficult to assess directly and the size of source code structure space is so large and highly dimensioned, we planned to use execution time as a surrogate for compiler quality. We feel that this is a proper measure, in the sense that code execution time is what compiler optimization is all about, at least for HPC.\footnote{DoD's large Challenge Projects are often required to show the efficient operation on the machines they use. In this context, efficiency is usually measured only by the scaling of the program to large numbers of MPI tasks.}

Further, we doubt that the management of the memory hierarchy can be determined except by its execution time behavior. I.e., to measure how well the code generated by a compiler utilizes the memory system, we believe one has to use code execution time.

To make the performance data meaningful we timed each test loop three times in succession. The motivation for this approach was to guarantee that the time from program-start to loop timing varied:

```fortran
DO I=1, noTimings ! = 1 to 3
  timeStmp(tstNo, 1, I) = compTim()
<loop being timed>
  timeStmp(tstNo, 2, I) = compTim()
  call checkResult
call reinitialize
enddo
```

This loop structure is repeated for each of the 708 loops, ase [6]. The calls to checkResult and reinitialize have the side effect of flushing all data from the cache before the next loop. For many of the test loops, the code block above was embedded in an outer loop that doubled the iteration count of the test loop 20 times. In the sample graphs at the end of the paper or those at [6], you can see timing ratios for blocks of loops with increasing iteration counts.

Using the smallest timing from three re-executions of a loop appears to produce repeatable and reasonable results. The entire program was run 15+ times for each compiler and the minimum of the 15 minimal-times is the value used for this paper.

The test code program does not perform any I/O until the last few code snippets.

### Compiler Differences

The table below shows the execution speed ratios on Ognip comparing the time for the loops compiled with “-fast” to those compiled with -O2. If the ratio is greater than 1.0, then -O2 code performed faster than -fast. The column heading show the compiler and the “optimization” selections compared.

<table>
<thead>
<tr>
<th>Statistic / Compiler</th>
<th>Cray</th>
<th>Gnu</th>
<th>PathScale</th>
<th>PGI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum Time Ratio</td>
<td>1.35</td>
<td>1.50</td>
<td>2.28</td>
<td>1.17</td>
</tr>
<tr>
<td>99th Percentile</td>
<td>1.20</td>
<td>1.25</td>
<td>1.30</td>
<td>1.13</td>
</tr>
<tr>
<td>95th Percentile</td>
<td>1.05</td>
<td>1.14</td>
<td>1.15</td>
<td>1.05</td>
</tr>
<tr>
<td>50th Percentile</td>
<td>1.00</td>
<td>1.00</td>
<td>0.94</td>
<td>1.00</td>
</tr>
<tr>
<td>5th Percentile</td>
<td>0.96</td>
<td>0.92</td>
<td>0.39</td>
<td>0.94</td>
</tr>
<tr>
<td>1st Percentile</td>
<td>0.86</td>
<td>0.80</td>
<td>0.32</td>
<td>0.86</td>
</tr>
<tr>
<td>Minimum Time Ratio</td>
<td>0.72</td>
<td>0.56</td>
<td>0.04</td>
<td>0.61</td>
</tr>
</tbody>
</table>

The Gnu compiler with the fast option had the best speedup, nearly twice as fast, on the “loop”

```fortran
DO I=1, Nparhd ! = 1 to 128
  DO J=1, NSomeDat ! = 1 to 32
    DO K=1, nFewDat ! = 1 to 15
      XP1(I,J) = XP1(I,J) + XP2(I,K)* XP3(K,J)
    enddo
  enddo
endo
```

For this loop the -fast option caused preloading the XP2 values and fully unrolling the inner loop (the loop iteration counts are parameters).
The PathScale compiler’s -fast option slowed the execution of
\[ j = 0 \]
\[
\text{DO I=1, nFewDat} \\
K = nFewDat - I + 1 \\
J = J+1 \\
\]
\enddo
by more than a factor of 2. In this case, PathScale unrolled the loop with -fast. Our guess is that cache misses slowed execution of the unrolled code. At the other extreme, -fast increased execution speed of
\[ v1 = 2 \]
\[
\text{DO I = 1, NPARHD} \\
XS1(I) = XS2(I)**v1
\]
\enddo
by a factor of 25 for PathScale. In this case the compiler called a different function, vrs4_powf(), to evaluate the expression, instead powf(). No other compiler used the vrs4_powf() function, one that computes four exponentiations at a time. In effect, it unrolled the loop.

The Portland Group compiler was slowed by almost 20% on the set of loops
\[
\text{DO I=1,13} \\
\text{DO I=14,330 ! note overlap} \\
\text{DO I=34,nData ! nData = 600}
\]
apparently because of loop unrolling. At the other extreme, it sped up
\[ k = 1 \]
\[
\text{DO I=1, nParHD} \\
\text{IF(1sl(I)) THEN} \\
\text{ENDIF}
\]
by almost 40%.

Using our loop-by-loop technique for measuring compiler-to-compiler differences does not seem appropriate as we have noted. In fact, despite our efforts to use performance as an accurate surrogate for compiled code quality, we may have bad time values instead of compiled-code quality differences. Compiler writers may be interested in specific areas where their compiler’s relative performance is poor, so they can improve it. Thus the table of inter-compiler comparisons should not be viewed as comparing compiled code quality.

For example, the zero entries in the second table result from the random number generator, which took substantially longer in the PGI-compiled code than with the others. PGI’s code may be doing substantially better or more anticipatory work than the others.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum Time Ratio</td>
<td>1.23</td>
<td>3.86</td>
<td>29.42</td>
<td>3.99</td>
<td>3.07</td>
<td></td>
</tr>
<tr>
<td>99th Percentile</td>
<td>1.09</td>
<td>3.19</td>
<td>2.84</td>
<td>1.18</td>
<td>3.32</td>
<td>1.81</td>
</tr>
<tr>
<td>95th Percentile</td>
<td>1.04</td>
<td>1.94</td>
<td>1.74</td>
<td>1.06</td>
<td>2.00</td>
<td>1.52</td>
</tr>
<tr>
<td>50th Percentile</td>
<td>1.00</td>
<td>1.05</td>
<td>1.02</td>
<td>1.00</td>
<td>1.06</td>
<td>0.97</td>
</tr>
<tr>
<td>5th Percentile</td>
<td>0.95</td>
<td>0.73</td>
<td>0.58</td>
<td>0.95</td>
<td>0.72</td>
<td>0.28</td>
</tr>
<tr>
<td>1st Percentile</td>
<td>0.86</td>
<td>0.43</td>
<td>0.30</td>
<td>0.88</td>
<td>0.30</td>
<td>0.09</td>
</tr>
<tr>
<td>Minimum Time Ratio</td>
<td>0.79</td>
<td>0.00</td>
<td>0.00</td>
<td>0.71</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

As with the intra-compiler comparison above, we made an effort to see how the compilers “optimized” or failed to optimize code, by looking at the assembly language output for the loops producing the maxima or minima in the table above. Here we summarize those cases where this yielded useful insight.

The PGI compiler outperformed the Gnu compiler at O2 and fast optimization levels by the widest margin on a character string copies. The Gnu compiler compiled the code while PGI made a single call to __c_mcopy1. With -fast, Gnu unrolled the loop, but __c_mcopy1 was still nearly four times faster.

The PGI compiler had the best performance relative to the Pathscale compiler for fast optimization on
\[
\text{DO I=1, nData} \\
\text{ls1(I) = CH1(I:I) .EQ. CH2(I:I)} \\
\text{ENDDO}
\]
For this loop, it appears that PGI -fast is preloading the data, probably reducing cache miss time to achieve more than three times the performance of PathScale.
-fast.

The loop where the PGI -O2 compilation code ran orders-of-magnitude slower than either Gnu or Pathscale (but quite close to Cray’s compiler) is

```
DO I = 1, NPARHD
    XS1(I) = XS2(I)*XS3(i)
    CALL random_seed()
endo
```

Pathscale called ranf_4 and Gnu called _gfortran_random_seed while PGI called pghpf_rseed, which apparently slowed the execution tremendously for both levels of optimization.

Summary:
If performance on a code is not as expected, the easiest optimization is often to vary the compiler or compiler options. Changing from -fast to -O2 or conversely, may yield good results, especially for programs with loop counts near 64. If your program will compile with another compiler, this analysis suggests you should try it or try it on some of the hot-spot routines.

The big and long term program improvement is from enhancing the algorithms in heavily used code blocks and cleaning up their code. We feel certain that clean code will always be easier to analyze and optimize, both to programmers working on it and to compilers—it is not difficult to confuse a compiler. Clean, understandable code is the best defense against poor compiler performance.

Observations:

References and Bibliography

6. The entire program source code and performance result spreadsheets are at www.arsc.edu/~higbie/CompilerTests

About the Author
Lee Higbie is an HPC Specialist at the Arctic Region Supercomputing Center, P.O. Box 756020, Fairbanks, AK 99775-6020, +1-907-450-8688, higbie@arsc.edu. He has been working in supercomputing and parallelization for almost five decades.