A case study on the importance of compiler and other optimizations for improving super-scalar processor performance

M. Duvall[1], P. Andersen[1], J. Leggoe[1], A. Graham[1], D. Cooke[1] & J. Antonio[2]
[1]*Departments of Chemical Engineering and Computer Science, Texas Tech University, United States*
[2]*School of Computer Science, University of Oklahoma, United States*

Abstract

The importance of properly optimizing code for execution on super-scalar processors was investigated. Access to the domain specialist was not available during the optimization investigation. For this study of an existing serial FORTRAN application, the use of compiler switches, manual coding techniques, a commercial preprocessor utility (KAP), and a commercial parallelization utility (FORGE) showed the potential to affect execution performance by more than an order of magnitude. The application for the case study was a three-dimensional boundary element code that modeled spherical particle transport phenomena in a particle suspension. Separate experiments were conducted using two different processor platforms: a four node IBM SP (160Mhz POWER2 CPU) and a single node DEC Alpha (667Mhz 21164 CPU).

Execution times for the non-optimized, serial base case were 72 hours on a single IBM SP node and 66 hours on the DEC Alpha. Using a combination of compiler switches and manual optimizations, such as in-lining of inefficient subroutines, execution times were reduced to 7.5 hours on a single IBM SP node and 5.4 hours on the DEC Alpha. The use of the KAP pre-processor reduced execution time to 2.3 hours on the single IBM SP node. Using the parallelization software FORGE and four nodes on the IBM SP resulted in an execution time of 25.8 hours without compiler optimization and 3.0 hours using compiler switches for optimization.

1 Introduction

As the availability of cheaper and faster computers has increased, it has become feasible to develop suspension models in which the individual particles are included as discrete entities. These models are used for more detailed investigation of flow phenomena such as the effects of particle interactions on macroscopic suspension behavior. Acceleration of the solution process is considered essential to increase the number of particles that can be included in the model and to increase the period of real time that can be simulated. Both improvements would greatly increase the ability of the model to provide information on the evolution of phenomena such as particle migration in real flow situations.

A number of techniques to accelerate the solution process are available and documented in various books and papers. These techniques include the use of compiler switches, manual code optimization, commercial preprocessors, and conversion from serial to parallel code either manually or with the aid of commercial parallelization software. For optimization of serial code, Dowd and Severance [1] discuss a number of manual techniques to eliminate clutter in subroutine calls, loops, and branches. The authors also discuss in general terms the different levels of compiler optimization from no optimization to interprocedural analysis to floating-point optimization. The overview does not provide information on the amount of execution time saved using the different techniques or the order in which to apply the various techniques.

Stewart [2] provides a comparison of the execution time saving benefits of common compiler switches and information on which compiler switches to implement to reduce execution time. However, a comparison between manual implementation of a corresponding compiler switch (such as manual in-lining of subroutines) and use of the compiler switch is not made by Stewart. KAP [3], a preprocessor and optimizer, is described as being beneficial for code that 1) uses significant computation time; 2) has loop nests that use many local arrays; 3) uses a data set whose total size is larger than the cache of the machine; 4) have significant data reuse within a loop nest; and 5) has large loop iterations. As noted in [3], the benefits of KAP can be increased if compiler switches are used, but no information on the effectiveness of KAP plus compiler switches was available.

If serial code is converted to parallel code, the execution time can often be reduced. However, the execution time savings depends on the method used to create the parallel code. Ideally, the serial code should be completely rewritten as parallel code, which can be a time consuming task. Andersen [4] and Jelly [5] have proposed similar methodologies to reduce the time required to convert existing serial code into parallel code. However since both methodologies are iterative in nature, it is difficult to calculate up front the execution time savings that will be achieved by converting the serial code to parallel code. Commercial parallelization software such as FORGE [6], BERT 77 [7], or Visual KAP [3] can be used to analyze serial code, create parallel code, and estimate the speedup from converting serial to parallel code. However, commercial parallelization software is platform specific which limits software availability and portability of the parallel code. A combination of the techniques discussed by the above authors was needed to efficiently optimize the serial code and analyze parallel code speedup.

This paper outlines and discusses the steps taken to optimize a serial FORTRAN program to significantly reduce execution time, using many of the techniques listed above. Section 2 provides a description of the model and serial code. Section 3 discusses the path to serial optimization and associated execution time results. Section 4 focuses on the initial parallel optimization results. Finally, Sections 5 and 6 summarize all of the current results, review the methodology used to optimize the serial code, and discuss future plans to complete the development of the parallel version of the model code.

2 Model Overview

The existing application code used for this study was originally encoded as a serial FORTRAN program. A three dimensional boundary element model was used to simulate spherical particle transport in flows subjected to varying boundary conditions. At each time step, the model determined the velocity of all particles and updated particle positions using a fifth order Runge-Kutta algorithm. Integration step size was adjusted over time within the program to reduce the stiffness of the problem as the particles came into proximity with each other and the boundaries.

The base case study was the simulation of ten spherical particles flowing in a cylinder for ten seconds. The code consisted of 2500 lines of code with 25 subroutines. Most of the subroutines were nested and contained two or more sets of nested loops. The application code was executed on both a four node IBM SP (160Mhz POWER2 CPU) and on a single node DEC Alpha (667Mhz 21164 CPU). Execution of non-optimized code required 72 hours on a single node of the IBM SP and 66 hours on the DEC Alpha. Results from the non-optimized execution of the serial code on both the IBM SP and DEC Alpha were used to compare to the results from the serial and parallel optimizations of the FORTRAN code.

3 Serial Optimization

Given the large serial code with nested loops and subroutines – and no access to the domain specialist who designed the code – a methodology was needed to efficiently analyze the serial FORTRAN program to determine which manual optimizations and compiler switches to use. Table I summarizes the switches used for code optimization on the IBM SP and DEC ALPHA referred to through the remainder of this paper.

Table 1: Description of compiler optimization switches.

Compiler Switch	Description
O#	In-lining of intrinsic functions, global optimization, branch elimination using memory and compiler time intensive optimizations. # indicates level of optimization.
hot	Performs high order transformation of loops.
arch	Use processor specific instructions.
tune	Tune the code for a specific processor.
cache	Customize the cache for a specific cache configuration.
Q	In-line all subroutines that can be in-lined.
autodbl	Perform calculations using longer data types and full PWR2 floating point calculations.
float	Round single precision expressions only when stored as single precision.
unroll	Unroll loops to a given depth.
align	Align all COMMON block entities.
strict	Do not change semantics of program to account for floating point precision and exceptions.
KAP	Use KAP preprocessor code and includes.

To quickly and efficiently profile the code for subroutine dependence and execution time, *gprof* on the IBM SP was used [1]. *gprof* is a timing and profiling utility used to determine subroutine dependency and execution times. Figure 1 shows output from *gprof* as applied to the original code. Execution times from relevant subroutines are shown all of the figures. Subroutine dependency results from *gprof* are not shown any of the figures for clarity.

```
  %    cumulative    self                  self      total
 time    seconds    seconds     calls     ms/call   ms/call      name
 33.2     7436.27   7436.27   1991229483        0         0    .funds [9]
 20.2    11959.23   4522.96    219702144     0.02      0.06    .rqint [8]
 16.7    15696.24   3737.01          872  4285.56   4328.47  .decomp [10]
  4.9     19162.1   1102.25          872  1264.05  17838.99   .matvec [5]
  3.1    19852.65    690.55   1836824121        0         0   .der9t [13]
  2.2     20335.3    482.65     37879680     0.01      0.04   .rtint [12]
  2.1    20796.36    461.06   1747634267        0         0   .shp9t [15]
  1.5     21127.4    331.04   2081643706        0         0 .unormal [16]
```

Figure 1: Example *gprof* output from original serial code.

Some subroutines, such as *funds* and *rqint*, have a large number of calls with small execution times (self ms/call) resulting in large overhead from entering and exiting the subroutines without performing any significant number of calculations. These types of subroutines are good candidates for subroutine in-lining [1]. Also, parallelism is significantly reduced by these inefficient and nested subroutines, which could be a roadblock to developing good parallel code.

Figure 2 shows a sample of the results from rerunning the original code but using the compiler switch –Q for in-lining of subroutines.

```
  %     cumulative     self                    self       total
 time     seconds     seconds      calls      ms/call    ms/call        name
 38.3      7556.68    7556.68   1991229483          0          0    .funds [9]
 27.3     12930.02    5373.34    219702144       0.02       0.05    .rqint [8]
   19     16682.08    3752.06          872    4302.82    4302.82  .decomp [10]
  5.6     17785.45    1103.37          872    1265.33   17259.12   .matvec [5]
```

Figure 2: Example *gprof* output using –Q compiler switch.

From Figure 2, some of the subroutines were in-lined successively using the compiler switch -Q. However due to the nested nature of some subroutines, for example *funds* and *rqint*, some subroutines could not be in-lined using the –Q switch. Using manual in-lining [1] of subroutines on the original code resulted in the *gprof* output shown in Figure 3.

```
  %     cumulative     self                    self       total
 time     seconds     seconds      calls      ms/call    ms/call        name
 55.2     10058.58   10058.58          872   11535.07   11576.88   .decomp [6]
 35.4     16504.72    6446.14       474368      13.59       13.6     .int4 [8]
  8.9     18122.07    1617.35          872    1854.76    9253.07   .matvec [7]
  0.4     18194.99      72.92         1744      41.81      41.81    .solve [9]
```

Figure 3: Example *gprof* output using manual in-lining of subroutines.

The results from *gprof* also included the total execution time of the program. Thus using *gprof* and comparing subroutine and total execution times for different in-lining methods, we were able to determine that manual in-lining some subroutines would be beneficial to optimization by elimination of subroutine overhead. In the final optimization of the serial code, the use of manual in-lining contributed significantly to the decrease in the execution time. To determine whether using the compiler switch for in-lining would be successful, we examined the subroutine dependence (nested nature of the subroutines), which is also in the *gprof* output. Nested subroutines are not typically in-lined by the –Q compiler switch optimization.

With the inefficient subroutines in-lined, compiler switches could be used to further reduce the execution time of the program. Compiler switches have the disadvantage of possibly altering the semantics of the program leading to numerical differences in program solutions. The trade off for decreased execution time is often increased floating point and round off errors. Table 2 compares the execution times on the IBM SP for different compiler switches used on the original code and manually in-lined code. Two solutions from Table 2 showed numerical differences in precision from the original solution. The domain specialist was not available to provide information on the required accuracy in the model solution. In this case, having the domain specialist available would be an advantage to determine whether the serial solutions with numerical differences were still accurate enough to be considered useable.

Table 2: Optimized serial execution times on the IBM SP. An asterisk indicates the resulting solution differed numerically in precision from the original solution.

Compiler Switch	Execution Time Original Code [hr]	Execution Time Manual In-line [hr]
None	72.0	67.7
O3	7.9	7.4
O3, qarch, qtune	7.7	7.2
O3, qarch, qtune, qcache, qfloat	7.7	5.6
O3, qarch, qtune, qautodbl, qfloat	7.7	2.5*
O3, qarch, qfloat, qautodbl, qfloat, KAP	7.8	2.3*

After manually in-lining inefficient subroutines, the IBM SP compiler was able to significantly reduce the execution time. With the nested and inefficient subroutines in the original code, use of the KAP optimizer did not decrease the execution time over use of compiler switches only. Using the KAP optimizer in addition to compiler switches on the manually in-lined code provided an additional 8% decrease in execution time. Note that the numerical differences in the precision of the solution obtained with the KAP optimizer would still have to be validated by the domain specialist.

Table 3 shows the execution times from the Dec Alpha using optimization switches on the original code and manually in-lined code. No KAP preprocessor/optimizer was available for the DEC Alpha. Most of the decrease in execution time was due to level three optimization. While a maximum of five levels are available on the Dec Alpha, higher levels of optimization did not decrease execution time. However, manual in-lining allowed the DEC Alpha compiler to further decrease the execution time by 6.5%. Tuning the compiler for the DEC Alpha and unrolling loops [1] resulted in an additional 5% savings in execution time over -O3 optimization and manual in-lining of subroutines. The numerical solutions for different compiler switch settings on the DEC Alpha were all identical to the original solution on the DEC Alpha without optimization.

Table 3: Optimized serial execution times on the DEC Alpha.

Compiler Switch	Execution Time Original Code [hr]	Execution Time Manual In-line [hr]
None	77.0	No data
O3	6.1	5.7
O3, align, arch, tune, unroll	6.2	5.4

4 Parallel Optimization

A few commercial utilities exist for developing parallel code from serial code on the IBM SP [3, 6, 7]. Before spending considerable time reverse engineering the serial code to develop parallel code, we wanted an estimate of the speedup expected from parallelization. The effectiveness of the use of compiler switches for optimization of parallel code was also investigated. For the IBM SP, the test utility of choice was FORGE [6]. Utilizing a demonstration version of FORGE with default settings, the serial code with manual in-lining of subroutines was converted to parallel code. The code was then run on the four nodes of the IBM SP both with and without compiler switches. The results of the execution times are summarized in Table 4.

Table 4: Parallel execution time utilizing four IBM SP nodes. The parallel code was created using the commercial utility FORGE.

Compiler Switch	Execution Time for Parallel Code [hr]	Speedup (versus manually in-lined serial code with similar compiler switches)
None	25.8	2.6
O3	3.5	2.1
O3, qarch, qtune, qfloat, qcache	3.0	1.9

The average speedup was 2.2 when compared to the execution times for the manually in-lined code with similar compiler switches running on a single IBM SP node. The compiler switch –qautodbl caused severe numerical problems in the parallel solution when used with the parallel program as developed using FORGE on the IBM SP. Thus, speedup results are not provided for the –qautodbl compiler switch simulations. Also during execution of the parallel code with only ten particles, some of the nodes were under utilized as observed when analyzing the FORGE timing results. It is expected that as more particles are included in the model and as the parallel code created using FORGE is refined, under utilization of the nodes will not occur.

5 Conclusions

Utilizing manual optimization techniques, compiler switches, a commercial preprocessor/optimizer utility, and a commercial parallelization utility, the execution time for the serial code was decreased by an order of magnitude. For both serial and parallel code, level optimization compiler switches provided the largest decrease in execution time on both computer platforms tested. Utilizing architecture specific compiler switches further reduced execution times. Some compiler switches showed the potential to alter code semantics resulting in numerical precision errors in the model solution. A solution based on no optimization is still required to compare to optimized solutions.

Using manual optimization techniques such as in-lining of inefficient and deeply nested subroutines further decreased execution time by removing computational overhead and roadblocks to compiler optimization and parallelization. Commercial utilities to convert serial code to parallel code have improved greatly in the past few years. Using the basic settings for FORGE utility and compiler switches on the IBM SP resulted in a speedup of 2.2 when compared to single node execution times on the IBM SP.

The following methodology is suggested to efficiently optimize serial code, to test serial code for parallel application, and to decrease serial and parallel execution times without access to the domain specialist.

1) Use a profiling tool [1] to determine subroutine dependency and inefficiency.
2) Use manual optimization techniques [1] to reduce computational overhead and roadblocks to compiler optimization.
3) Add compiler switches [2] for level optimization to reduce execution time. The numerical results of the program need to be compared to the original solution as some compiler switches can alter the semantics of the program.
4) Add architecture specific compiler switches.
5) Use a commercial parallelization utility to check for benefits of parallelization of serial code [4] without the expense of writing parallel code by hand.

Using the above procedure, approximately 60 person-hours (excluding execution times) were required to optimize the serial code and to create the initial parallel code. Another 30 hours were required to research compiler switch settings and to review manuals for available commercial parallelization utilities.

6 Future Studies

The next step to decrease the execution time of the model will be to reverse engineer the serial code and to create the parallel MPI code using the methodology outlined by Andersen [4]. The serial program analysis output from FORGE and BERT 77 will be used to determine what changes to the serial code are required to create a more efficient parallel code. The parallel code created with FORGE will be further optimized to provide a better comparison between manual parallelization of serial code and using commercial software to parallelize code.

Special thanks go to the Department of Computer Science of Texas Tech University for access to the IBM SP and for their help in obtaining the results presented in this paper.

References

[1] Dowd, K. & Severance, C. R. Programming and tuning software (Section II). *High Performance Computing, Second Edition*, O'Reilly: Cambridge, pp. 79-169, July 1998.

[2] Stewart, K. Using the XL compiler options to improve application performance, *http://www.rs6000.ibm.com/resource/technology/options.html,* 1999.

[3] Kuck & Associates, Inc. KAP for IBM Fortran and C, *http://www.kai.com/productinfo.html*, 1999.

[4] Andersen, P. H., Pizzi, J., Zhu, R., Cao, Y., Bagert, D., Antonio, J., Lott, F., & Grieger, J. Evaluation of a methodology for the reverse engineering and parallelization of sequential code. *Proceedings of the International Symposium on Software for Parallel and Distributed Systems (PDSE '99)*. Computer Society: Los Angeles, CA, May 1999.

[5] Jelly, I., Gorton, I. & Croll, P. SEMPA: Software engineering methods for parallel scientific applications. *Software Engineering for Parallel and Distributed Systems*. Chapman & Hall: New York, 1996.

[6] Applied Parallel Research, Inc. FORGE and FORGE Explorer. *http://www.apri.com*, 1998.

[7] Paralogic. BERT 77: Automatic and efficient parallelizer for FORTRAN, *http://www.plogic.com/bert.html*, 1988.