LA-UR-05-5950

Approved for public release;
distribution is unlimited.

Title: INCREASING MCNP5 CALCULATION SPEED BY COMPILER OPTIMIZATION

Author(s): JESSE CHEATHAM & FORREST B. BROWN

Submitted to:

Increasing MCNP5 Calculation Speed by Compiler Optimization

Abstract

The speed performance of MCNP5 is examined using four different compilers with their different optimization options on the Lambda Linux computing cluster. Intel, Portland, Absoft, and Lahey compilers are compared on calculation times by computing eigenvalue and fixed-source problems with their default options as well as their optimizations that maintain solution accuracy. By choosing certain optimization options, a reduction of run time of around 40% was seen in some of the MCNP5 builds for a single processor.

Introduction

While work has been done in optimizing the MCNP code for speed, there has been less attention given to optimization of the code by compilers. To determine whether different compilers create significantly different run time speeds, four compilers (Intel, Portland, Absoft, and Lahey (Appendix 2)) are used to build MCNP5 on the Lambda Linux system (Appendix 1). These MCNP5 builds are then speed tested on an eigenvalue and fixed-source problem (Problem details in Appendix 4). All of the timed problems are then normalized to the performance of the Intel default build for the timing problem in question for cross compiler comparisons. Besides the default builds for each compiler, compiler specific optimizations are examined as well. All of the timed problems listed in the discussion section below passed their test for accuracy.

Guaranteeing the accuracy of the solution can be difficult when optimizing codes. Optimizing written code with a compiler generally means that the order in which the code was written will be rearranged in hopes of better stream lining the memory access and computation. A side effect of this process is that calculations that were coded to work one way may inadvertently be rearranged to produce very different effects, i.e., a wrong answer. To insure the accuracy of the MCNP build, there are 42 test problems that are run at the end of an installation and are compared to the anticipated outputs and MCTAL files. If the MCTAL files or the Outputs files differ significantly from the expected values, the MCNP5 build is deemed unusable.

Discussion

The running times for the MCNP5 builds on the Lambda cluster are shown in Tables (1(a,b) – 4(a,b)). Run times are recorded in seconds and Rtime and Itime are relative speed indicators. Rtime is the relative speed of the compiler with its default options, while Itime is the relative speed with respect to the Intel compiler defaults. Itime allows for cross compiler comparisons of calculation speed. In the comparison, it should be noted that the computation time recorded can vary by a couple of seconds each instance the MCNP5 build in question is run. Therefore, slight differences in calculation time may not indicate an actual speed increase. The default compiler options (default

build) are shown in gray, the optimal speed build for the compiler is shown in bold, and details of the options listed in the following tables may be examined in Appendix 3.

Intel's Fortran compiler showed the most positive response to optimization. The MCTAL files passed for all of its general O optimizations as well as some of the more advanced rearrangement schemes. As shown in Table 1a, optimized MCNP5 builds were doing the same eigenvalue problem in about 58% of the time of the default build. Table 1b also shows that the fixed source problem has similar success in completing the problem around 51% of the original time.

Table 1a: Intel Build Eigenvalue Test

Intel option	time (s)	Rtime	Itime
O0	404.62	100.0%	100.0%
O1	262.69	64.9%	64.9%
O2	263.95	65.2%	65.2%
O3	264.03	65.3%	65.3%
O0 prof_gen prof_use ipo	402.71	99.5%	99.5%
O1 prof_gen prof_use ipo	237.99	58.8%	58.8%
O2 prof_gen prof_use	248.83	61.5%	61.5%
O2 prof_gen prof_use ipo	**234.55**	**58.0%**	**58.0%**
O2 prof_gen prof_use ipo tpp6	237.08	58.6%	58.6%
O3 prof_gen prof_use ipo	240.45	59.4%	59.4%

Table 1b: Intel Build Fixed Source Test

Intel option	time (s)	Rtime	Itime
O0	136.87	100.0%	100.0%
O1	89.75	65.6%	65.6%
O2	89.7	65.5%	65.5%
O3	89.76	65.6%	65.6%
O0 prof_gen prof_use ipo	135.44	99.0%	99.0%
O1 prof_gen prof_use ipo	74.95	54.8%	54.8%
O2 prof (gen use)	72.23	52.8%	52.8%
O2 prof_gen prof_use ipo	74.96	54.8%	54.8%
O2 prof_gen prof_use ipo tpp6	**69.21**	**50.6%**	**50.6%**
O3 prof_gen prof_use ipo	72.33	52.8%	52.8%

The Portland compiler had a few more difficulties in being optimized. Since the O2 optimization failed for the compiler, much of the advanced code rearrangement could not be used since the O2 option is essential for it. Even so, the default Portland MCNP5 build performed 13% better than the default Intel build on the eigenvalue problem shown in Table 2a. The most optimized run took only 77% of the time of the default Intel run and 92% of the time of the default Portland run.

Table 2a: Portland Build Eigenvalue Test

Portland option	time (s)	Rtime	Itime
O0	349.97	103.5%	86.5%
O1	**309.7**	**91.6%**	**76.5%**
O0 Mrecursive	353.88	104.6%	87.5%
O1 Mprof	415.93	123.0%	102.8%
O1 Mrecursive	318.26	94.1%	78.7%
O1 tp px	338.27	100.0%	83.6%

Fixed source runs by the Portland build also demonstrate a good speed increase over the default Intel build as well as with respect to its own default build. The fixed source problem run time was only 59% of a default Intel build and was 85% of the default Portland build. This shows the fixed source problem was more responsive to optimization. As a side note, the Portland runs appeared to be one of the slowest in the timed tests of the 42 test problems. This is most likely due to the way that the Portland build loads and unloads information and therefore while Portland is good for long runs, it should not be used in fast running problems that must be completed over and over.

Table 2b: Portland Build Fixed Source Test

Portland option	time (s)	Rtime	Itime
O0	99.08	105.2%	72.4%
O1	**80.46**	**85.4%**	**58.8%**
O0 Mrecursive	103.85	110.3%	75.9%
O1 Mprof	164.53	174.7%	120.2%
O1 Mrecursive	84.75	90.0%	61.9%
O1 tp px	94.19	100.0%	68.8%

Absoft's compiler had similar troubles to Portland's compiler. Most of the advanced optimizations did not pass the MCTAL tests since its O2 optimization, which is required for most optimizations, failed. Overall, Absoft's compiler performed poorly in optimizing MCNP. Unfortunately, the default build for Absoft seemed to be the fastest that the code could be run for both fixed source and eigenvalue problems while maintaining accuracy. The default Absoft runs were comparable to the default Intel runs.

Table 3a: Absoft Build Eigenvalue Test

Absoft option	time (s)	Rtime	Itime
O0	548.35	136.7%	135.5%
O1	407.16	101.5%	100.6%
O1 cpu:p6	401.06	100.0%	99.1%
O1 fpic	481.02	119.9%	118.9%
O1 B24	**398.2**	**99.3%**	**98.4%**
O1 P	444.18	110.8%	109.8%

Table 3b: Absoft Build Fixed Source Test

Absoft option	time (s)	Rtime	Itime
O0	196.72	140.3%	143.7%
O1	140.36	100.1%	102.5%
O1 cpu:p6	**140.24**	**100.0%**	**102.5%**
O1 fpic	173.46	123.7%	126.7%
O1 B24	142.01	101.3%	103.8%
O1 P	180.64	128.8%	132.0%

Lahey's compiler was most difficult to ensure accuracy. The default runs of the Lahey compiler generate a number of differences in the MCTAL files. On top of that, the eigenvalue run produces a different eigenvalue, 0.99825, than all of the other compilers with their options which yielded the same value, 0.99662. Accepting that the default Lahey MCTAL files were so different to begin with, I continued with optimizations that yielded the differences in the same ball park.

The Lahey compiler runs for the eigenvalue problems were similar to the default Intel runs. The fastest optimization yielded a run time that was 93% of the default Intel run time. However, the fixed source problems were considerably slower. Even the most optimized Lahey run took longer than the default Intel run. The default Lahey run for the fixed source problem ran 50% longer than the Intel one.

Table 4a: Lahey Build Eigenvalue Test

Lahey option	time (s)	Rtime	Itime
o0	454.95	100.0%	112.4%
o1	377.67	83.0%	93.3%
o0 tpp	456.24	100.3%	112.8%
o0 prefetch	374.88	82.4%	92.6%
o0 unroll 4	453.5	99.7%	112.1%
o0 li	454.96	100.0%	112.4%
o0 x arg	451.81	99.3%	111.7%
o0 tp	456.29	100.3%	112.8%
o1 prefetch	379.76	83.5%	93.9%
o1 tp	**371.72**	**81.7%**	**91.9%**

Table 4b: Lahey Build Fixed Source Test

Lahey option	time (s)	Rtime	Itime
o0	202.65	100.00%	148.10%
o1	170.17	84.00%	124.30%
o0 tpp	199.09	98.20%	145.50%
o0 prefetch	173.87	85.80%	127.00%
o0 unroll 4	203.03	100.20%	148.30%
o0 li	203.51	100.40%	148.70%
o0 x arg	204.47	100.90%	149.40%
o0 tp	202.84	100.10%	148.20%
o1 prefetch	172.9	85.30%	126.30%
o1 tp	**170.05**	**83.90%**	**124.20%**

Conclusion

An optimized Intel compiler build solved both fixed source and eigenvalue problems with run times less than 60% of the default builds. Luckily, for this increased speed in calculation, there appears to be no noticeable change in accuracy of the solution according to the MCTAL files. Portland's compiler also made fast MCNP5 builds. The optimized Portland runs ran in less than 80% time of the default Intel eigenvalue runs and less than 60% of the fixed source run time. It may be possible to increase the speed of the Portland compiler by advanced optimization options if the Portland O2 option would pass the MCTAL file test.

Absoft's compiler did not perform as well as Intel's or Portland's. Even its optimized runs were comparable to the default Intel ones. Like Portland, Absoft failed its O2 optimizations and therefore cut off many of the code rearrangement options. Similarly, Lahey also failed the O2 optimization. However, besides giving large MCTAL differences even in its default run, its optimized run times still were comparable to Intel's default for an eigenvalue problem and near 50% longer in calculating a fixed source problem.

In conclusion, Intel's compiler optimized MCNP5 build performed the best out of all the compiler builds. These results, however, are limited in lifetime. With improvements in compilers as well as the operating systems and hard ware of the system in which they are being run, these results will become obsolete. It should also be noted that these results pertain only to the MCNP5 source code and may vary significantly with different types of codes.

Appendix

1. Computer type Lambda

Each backend node is a Compaq DL360 with two Intel Pentium-3, 1 GHz processors. Each backend node has 2 disk drives - a 9 GB system disk and an 18 GB scratch disk. The Lambda System is running RedHat Linux 2.4.10.

2. Compiler versions

2.a Intel -- intel-fortran_8.1.023

2.b Portland -- pgi_5.2-4

2.c Absoft -- absoft_9.0

2.d Lahey -- lahey_6.2c

3. Compiler Options

3.a Intel

-O0 Disables all -O<n> optimizations. On IA-32 and Intel(R) EM64T systems, this option sets the -fp option.

-O1 On IA-32 and Intel(R) EM64T systems, enables optimizations for speed. Also disables intrinsic recognition and the -fp option. This option is the same as the -O2 option.

On Itanium-based systems, the -O1 option enables optimizations for server applications (straight-line and branch-like code with a flat profile). Enables optimizations for speed, while being aware of code size. For example, this option disables software pipelining and loop unrolling.

-O2 or -O
This option is the default for optimizations. However, if -g is specified, the default is -O0.

On IA-32 and Intel(R) EM64T systems, this option is the same as the -O1 option.

On Itanium-based systems, the -O2 option enables optimizations for speed, including global code scheduling, software pipelining, predication, and speculation. It also enables:

o Inlining of intrinsics

o The following capabilities for performance gain: constant propagation, copy propagation, dead-code elimination, global register allocation, global instruction scheduling and control speculation, loop unrolling, optimized code selection, partial redundancy elimination, strength reduction/induction variable simplification, variable renaming, exception handling optimizations, tail recursions, peephole optimizations, structure assignment lowering and optimizations, and dead store elimination.

-O3 Enables -O2 optimizations plus more aggressive optimizations, such as prefetching, scalar replacement, and loop transformations. Enables optimizations for maximum speed, but does not guarantee higher performance unless loop and memory access transformations take place.

On IA-32 and Intel(R) EM64T systems, when the -O3 option is used with the -ax and -x options, it causes the compiler to perform more aggressive data dependency analysis than for -O2, which may result in longer compilation times.

On Itanium-based systems, the -O3 option enables optimizations for technical computing applications (loop-intensive code): loop optimizations and data prefetch.

-tpp6 (i32 only)
Optimizes for the Intel(R) Pentium(R) Pro, Intel(R) Pentium(R) II and Intel(R) Pentium(R) III processors.

-prof_gen
Instruments a program for profiling.

-prof_use
Enables use of profiling information during optimization.

-ipo[n]
Enables multifile interprocedural (IP) optimizations (between files). When you specify this option, the compiler performs inline function expansion for calls to functions defined in separate files.

3.b Portland

-O[level]

Set the optimization level. If -O is not specified, then the default level is 1 if -g is not specified, and 0 if -g is specified. If a number is not supplied with -O then the optimization level is set to 2. The optimization levels and their meanings are as follows:

0 A basic block is generated for each C statement. No scheduling is done between statements. No global optimizations are performed.

1 Scheduling within extended basic blocks is performed. Some register allocation is performed. No global optimizations are performed.

-Mrecursive -Mnorecursive (default)

Allocate (don't allocate) local variables on the stack, thus allowing recursion. SAVEd, data-initialized, or namelist members are always allocated statically, regardless of the setting of this switch.

-Mprof[=option[,option,...]]

Set profile options. Normally, the -ql, -qp, or -pg switches are used for this; however, on some systems, it is desirable to override the default method of profiling. See the PGI User's Guide, or the system profiler manual, for further information.

-tp px

blended code generation that will work on any x86-compatible processor

3.c Absoft

-O0 no optimizations

-O1 Turn on basic optimizations to make executable programs run faster. The basic optimizations are: common subexpression elimination, constant propagation, and branch straightening.

-cpu:type

Use the -cpu:type option to generate instructions specific to a particular processor. The recognized type arguments are:

486 non-Pentium class Intel processor
p5 first generation Pentium
p6 Pentium Pro, II, and III
p7 Pentium 4
athlon AMD Athlon and Duron

host automatically establishes processor based on the machine that the program is compiled on. If the CPU type cannot be established, p5 is assumed.B24

-P Cause the compiler to instrument the code for profiling with gprof(1).Fpic

-B86 Forces the compiler to remove indexed address expressions from within loops. For the X86, this often has the desirable effect of reducing instruction stalls for floating point access. However, because the index must still be calculated, additional integer operations must be performed. If the application needs to be as fast as possible, try running once with this option and once without.

3.d Lahey

--o0 | -O0
no optimizations

--o1 | -O
classical, memory, and interprocedural optimizations

--tp
generate Pentium code

--tpp
generate Pentium Pro code

--[n]prefetch
Athlon and Pentium III optimizations

--[n]unroll <value>
perform/control loop unrolling

--[n]li
Lahey intrinsic procedures

-x arg
inline code

4. Problem Details

4.a Eigenvalue Problem

The eigenvalue problem run was BAWXI2i which came from the MCNP criticality validation suite. The only change was the kcode card which became
“kcode 5000 1.0 10 50”

4.b Fixed Source Problem

The fixed source problem came from problem 12 of the 42 test problems found in testinp.tar. The only change was the nps card which became
“nps 105000”

REFERENCES

X-5 Monte Carlo Team, "MCNP – A General Monte Carlo N-Particle Transport Code, Version 5", LA-UR-03-1987, Los Alamos National Laboratory (2003).