

HPC Fall 2008 – Project 1

Robert van Engelen

Due date: October 21, 2008

1 Introduction

1.1 Account and Login Information

For this assignment you need an SCS (SC) account. Contact the instructor if you do not have an SCS (SC) account.

1.2 Setup

If you use `tcsh` as your main shell then edit your `.tcshrc` file in your home directory, or edit the `.cshrc` file. If you use `bash` then edit `.bashrc`. Add the following lines:

```
# Set paths to Sun compilers
setenv MANPATH ${MANPATH}:/opt/sun/sunstudio12/man:/opt/SUNWspro/man
set path = ( /opt/sun/sunstudio12/bin $path )
```

1.3 Download

Next, download the project source code from

```
http://www.cs.fsu.edu/~engelen/courses/HPC/Pr1.zip
```

The package bundles the following files:

- `Makefile`: a standard Makefile to build the project.
- `config.guess`: determines the platform
- `make.platform-comp`: platform- and compiler-specific files used by `Makefile`
- `timing.sh`: a script used by `make plot`, see Section 8
- `global.h`: global definitions
- `bench.c`: a benchmark wrapper program to time `sqmat_mult()` square matrix multiply using random matrices of varying dimensions
- `cputime.h` and `timeres.c`: `cputime()` timer
- `rdtsc.h`: Intel RDTSC timer used by `cputime()`
- `timeres.h` and `timeres.c`: determine timer resolution
- `sqmat.c`: simple version of square matrix multiply in C
- `sqmatb.c`: blocked version of square matrix multiply in C (incomplete)
- `sqmatf.f`: simple version of square matrix multiply `SQMULT` in Fortran (incomplete)
- `sqmatfc.c`: wrapper to invoke Fortran `SQMULT` from C
- `sqmatw.c`: Winograd version of square matrix multiply (incomplete)
- `sqblas.c`: wrapper to invoke BLAS3 `DGEMM` (incomplete)

1.4 Project Aims

For this assignment we will use two types of machines:

- `gp004`: a quad core AMD Opteron 846, 2 GHz
- `tempest`: a Sun UltraSPARC IIIi 1.2GHz

The characteristics of these machines are very different. The AMD Opteron supports the x86 CISC instruction set architecture (ISA) and has 4 cores (though only one core will be used in this assignment, multithreading will be part of another project assignment). Instructions are translated to μ Ops for out-of-order execution. The Opteron has a 64KB data and 64KB instruction L1 caches (no trace cache), a 512KB L2 cache per core, and a 2048KB shared

L3 cache. The UltraSPARC IIIi is an older (thus slower) RISC processor with a 64KB data and 32KB instruction cache, and 1024KB L2 cache (all 4-way associative).

The optimizations applied by the compiler to the code are different for these two different types of machines. Instruction scheduling (e.g. modulo scheduling) by the compiler is especially important for the UltraSPARC. Loop restructuring is important for both. Loop restructuring requires dependence testing to verify the absence of cross-iteration dependences (other than the dependences of the loop counter variables). If the compiler cannot prove absence of a cross-iteration dependence, a (nested) loop cannot be reordered. The onus of the proof is on the compiler. If it cannot disprove dependence, the loops are not reordered. Hints provided by the programmer in the code or as compiler options can help.

In this project we will investigate the effect of compiler optimizations, programming hints, and algorithmic changes on the performance of an example numerical application. A simple matrix multiply will be used as an application.

The following topics will be investigated:

- To benchmark the performance of the application, we will pick a timer for each machine to conduct our timing experiments.
- We will compile with increasing levels of optimizations and add compiler hints via program annotations to improve performance.
- To investigate the difference of programming languages and compiler optimizations, we will compare the performance of matrix multiply written in C versus Fortran.
- The performance of the implementations is compared to BLAS DGEMM of the Sun Performance libraries.
- We apply blocking techniques to improve the performance of matrix multiply for larger data sets.
- To understand the impact of algorithmic differences, an alternative formulation of matrix multiply using Winograd's algorithm will be implemented and compared.

These tasks are explained in more detail in the next sections.

1.5 Report

Write a report of your findings and submit this to the instructor for grading. Include performance graphs and explanations of your findings in your report. Your report must address all tasks listed in the sections below and also list the source code of the algorithms you wrote. You do not need to submit your source codes separately electronically.

2 Selecting a Timer for Benchmarking

The `cputime()` function defined for you in `cputime.h` and `cputime.c` returns the elapsed time in seconds, which is measured from the previous call to this function to the next¹. We use the `cputime()` function to determine the number of floating point operations per second (MFlops “megaflops”) of our square matrix multiply routine `sqmat_mult()`, by measuring the elapsed time of k calls to `sqmat_mult()`:

$$\text{MFlops} = \frac{2kn^3}{10^6 \cdot \text{cputime}} \quad (1)$$

where n is the matrix dimension used in a benchmark test and $2n^3$ floating point operations are needed for the matrix multiply. We chose `MINRUNS=2` and `MINSECS=0.1` as defined in `bench.c` to force at least two runs and an elapsed time of 0.1 seconds. The benchmark driver automatically adjusts the number of runs k of the benchmark code to meet these constraints.

What follows are some remarks about MFlops.

As a measure of performance we use MFlop/sec (or MFlops). MFlops is actually a measure of *useful work* performed, i.e. we don’t count integer arithmetic and address calculations needed to access the arrays. Though MFlops is a really a misnomer, because it puts the emphasis on the floating point operations, not the speed by which the output data is obtained. In fact, slower algorithms with a high MFlop count may look better than faster algorithms with a lower MFlop count if we are not careful to define a fair MFlop formula.

For example, suppose that we optimize the square matrix multiply algorithm for the case of diagonal matrices. A check for diagonality takes n^2 comparisons and computing the output takes only n floating point operations. Thus, if we would compare the performance of the optimized algorithm to the naïve algorithm based on Flop counts per second, then the naïve algorithm would definitely win!

Because algorithms have different floating point operation counts, the performance of an algorithm with fewer Flops could look worse even when the algorithm runs in the same time or faster. Therefore, it is more fair to keep using the general MFlops formula (1) and use it as a *scaled measure of performance* of square matrix multiply, which is only dependent on the data size n and `cputime`, and not dependent on the actual operations performed, where n is the matrix dimension and k is the number of benchmark runs that are timed.

To get started, follow these steps:

- Open a terminal and login from an SCS machine to `gp004` with `qlogin -q gp-int@gp004 -l sunstudio`

¹Important: You cannot invoke `cputime()` from multiple threads in a multi-threaded application because it is not thread safe. This is not a problem in this assignment in which all of our code is single threaded.

- Chdir to the directory where you installed the content of `Pr1.zip`
- Run `make COMP=suncc plot` which compiles code with Sun CC and executes a set of benchmark programs (this can take a while), with all incomplete programs generating “max error tolerance exceeded” (ignore this for now)
- Run `gnuplot -persist timing.gnuplot` to view the performance results of `sqmat`

Note that the results of the incomplete programs are not shown in the graph.

When the benchmark runs take place, you will see:

```
Timer resolution is 0.010000000 seconds
Timing precision is at least 1 digit(s)
```

The precision of the timings reported by the benchmark was only one digit! Hardly trustworthy when the running time is short (0.1 second). By keeping `MINSECS` small we avoided very long waiting times for the benchmarks to complete. However, if the resolution of `cputime` is too low then the timing precision in digits p_{digits} might be insufficient, since:

$$p_{\text{digits}} \geq \lfloor \log_{10}(t/r) \rfloor \quad (2)$$

where r is the resolution of `cputime()` in seconds and t_{sec} is the elapsed time in seconds. When $t = \text{MINSECS} = 0.1$ and $r = 0.01$ in Eq.(2) we have $p_{\text{digits}} = 1$.

To ensure a reasonable precision of our timings we can increase `MINSECS` or use a timer with a higher resolution. This should be done with care since high-resolution timers tend to roll over when the maximum elapsed time that they can represent is exceeded. Timers have a fixed bit-width (e.g. 32 bit or 64 bit) that limits the maximum elapsed time that can be represented.

To study the use of timers for benchmarking, you will need to investigate the resolution of the following timers:

- `USE_TIMES`: use `times()` to obtain CPU time (user + system time).
- `USE_GETRUSAGE`: use `getrusage()` to obtain CPU time (user + system time).
- `USE_GETTIMEOFDAY`: use `gettimeofday()` to obtain wall-clock time.
- `USE_RDTSC`: use the Intel RDTSC instruction to obtain high resolution wall-clock time. Only available with Intel IA32 and IA64. Be careful when using this with multicore processors, since these may have RDTSC clocks per core and a context switch to another core gives a different readout. To avoid this you must set the thread affinity.

- `USE_GETHRTIME`: use `gethrtime()` to obtain high resolution wall-clock time. Available on Sun Solaris only.

To select a timer for `cputime()` use one of the above “defines” (C/C++ `#define` macros). A macro (`#define`) is set with compiler option `-D`. You should set these compiler options in the appropriate `make.platform-comp` files for platform- and compiler-specific options that are used to complete the Makefile:

- `make.i686-pc-linux-gnu-suncc` for `gp004`
- `make.sparc-sun-solaris2.10-suncc` for `tempest`

Find a wall-clock timer for `gp004` and one for `tempest` that has the best resolution. Modify the make files to use the timers.

NOTE 1: always do a `make clean` whenever you change any file, including the source files and `make.platform-suncc` files. Then recompile with `make COMP=suncc plot` to compile and run all benchmarks using compiler `suncc`. Use `make COMP=suncc prog` to compile a specific program.

NOTE 2: make sure the usage load on the machines on which you are benchmarking is low. To check the CPU usage use the `top` command.

3 Optimization and Benchmarking

In this part of the assignment we will experiment with advanced compiler optimizations and program annotations to improve the speed of the matrix multiply implementation. These experiments are conducted on `gp004` and `tempest` and results are expected to be different based on the RISC versus x86 characteristics of these machines. Your report should discuss the results of the experiments for `gp004` and `tempest`.

For this assignment you need to use the Sun Studio 12 compiler documentation:

<http://developers.sun.com/sunstudio/documentation/product/compiler.jsp>

Select the C users guide, then Appendix B to browse the compiler options when required to answer the questions below.

- After compiling `sqmat` with `make COMP=suncc sqmat` run `er_src sqmat`. The `er_src` command lists the source code and optimizations that have been applied to it, if

any. Do the same on `gp004` and `tempest`. From the results of `er_src` it seems no optimizations were applied. But there will certainly be a few as we discussed in class. Which are these?

Use `make COMP=suncc plot` and `gnuplot -persist timing.gnuplot` to view the MFlops performance graphs². Save the graphs for your report using screen capture or edit `timing.gnuplot` and modify to `set term png; set output "myplot.png";` (see also Section 8). Rerun `gnuplot timing.gnuplot` to generate a PNG file.

Use `make clean` when switching platforms or after modifying the make files and program sources.

- Login to `gp004`. Change the `make.platform.suncc` files by adding the `-fast` optimization option for the C and Fortran compilers. Use `make clean` and `make COMP=suncc plot` to compile and view the performance graphs on `gp004`. Run `er_src sqmat` to view the optimizations. Explain these optimizations and relate them to the `-fast` option's compiler transformations (see compiler documentation on this option).

Repeat the above experiment on `tempest` by changing the make file for UltraSPARC, then compile and run on `tempest`. Explain the compiler optimizations.

- Change the `sqmat_mult` function in `sqmat` by adding the `restrict` type qualifier for the pointer arguments of this function (see lecture notes on Uniqueness of Addresses in Architecture I). On `gp004` use `make clean` and `make COMP=suncc plot` to compile and view the performance graphs for `gp004`. Run `er_src sqmat` to view the optimizations. Explain why these additional optimizations appear to be dependent on the use of `restrict`.

Repeat the above compilation and benchmark run on `tempest` and explain the results.

4 Fortran vs C

Fortran and C code compiles differently, mainly because of the underlying programming language properties that the compiler can exploit. A Fortran compiler that translates Fortran to C first to compile the resulting C code loses some of these properties along the way.

- Edit `sqmatf.f` and complete the function's implementation of square matrix multiply. Use the loop nest ordering `j` (outer), `i` (middle), and `k` (inner). On `gp004` compile and run with `make COMP=suncc plot` and use `er_src sqmatf` to see the optimizations. View the graph with `gnuplot` and compare performance to the C `sqmat` code.

²Unfortunately, `tempest` does not have `gnuplot` installed. Open a new terminal and log on a different machine e.g. `pamd` to run `gnuplot -persist timing.gnuplot`.

- Now use the loop ordering `j` (outer), `k` (middle), and `i` (inner) loop and rerun the experiment. Explain what is happening. Hint: something strange has happened to the code since it seems to call BLAS DGEMM (!). Compare performance results with the previous version that used a different loop ordering.
- Do the above two steps on `tempest` and compare performance results.

5 BLAS Level 3: DGEMM

BLAS Level 3 is an efficient implementation of matrix-matrix linear algebra operations. Several implementations are available by vendors and as open source.

- Determine how to use DGEMM in C by perusing `man dgemm`. Edit `sqblas.c` to call `dgemm` to perform a square matrix multiply.
- Compile and run with `make COMP=suncc plot`. Compare performances of the `sqmat`, `sqmatf`, and `sqblas` benchmarks on `gp004` and `tempest`.

6 Blocking

Loop blocking to enhance performance by increasing the locality of memory access is very effective. Blocking improves locality of memory accesses to speed up the multiplication of large matrices.

The block multiply is performed as follows:

```

for (i = 0; i < n; i += BLKSIZE)
  for (j = 0; j < n; j += BLKSIZE)
    for (k = 0; k < n; k += BLKSIZE)
      block_mult(A, B, C, i, j, k, n);

```

where `block_mult` multiplies $C = AB$ block-wise for each $BLKSIZE \times BLKSIZE$ block located at $C_{i,j}$, $A_{i,k}$, and $B_{k,j}$.

- Edit `sqmatb.c` and implement a blocked version of matrix multiply. More information can be found in the file itself.

- Determine two block sizes `BLKSIZE` that work well on `gp004` and `tempest`, respectively. Report how you found your block sizes. Show the results you obtained with `suncc` on `gp004` and `tempest`.
- Use `er_src sqmat` to determine which loops in `sqmat` were automatically blocked by the compiler and how this blocking compares to your blocked version `sqmatb`.
- There are certain matrix sizes for which the code runs faster or slower as you can see from spikes in the performance graphs. For which matrix sizes does the code run better than the average in the same neighborhood on the graphs?
- Compare performances of the `sqmat`, `sqmatf`, `sqmatb`, and `sqblas` benchmarks on `gp004` and `tempest` using `make COMP=suncc plot` and `gnuplot`.

7 Winograd's Algorithm

There are several algorithms that (theoretically) improve the speed of matrix multiply, such as Winograd's algorithm and Strassen's algorithm.

Winograd proposed the following formulas (here rewritten for $n \times n$ square matrices):

$$\begin{aligned}
 x_i &= \sum_{k=1}^{\lfloor n/2 \rfloor} A_{i,2k-1} A_{i,2k} \\
 y_j &= \sum_{k=1}^{\lfloor n/2 \rfloor} B_{2k-1,j} B_{2k,j} \\
 C_{i,j} &= -x_i - y_j + \sum_{k=1}^{\lfloor n/2 \rfloor} (A_{i,2k} + B_{2k-1,j})(A_{i,2k-1} + B_{2k,j}) \\
 &\quad + A_{i,n} B_{n,j} \quad \text{if } n \text{ is odd}
 \end{aligned}$$

The number of floating point operations performed in an $n \times n$ square matrix multiply is $2n^3$ (one add and one multiply per iteration of the $n \times n \times n$ loop). Winograd's method uses $2n^3 + 3n^2$ operations when n is even and $2n^3 + 5n^2$ when n is odd, but with only half the number of multiplications.

- Implement Winograd's algorithm in Fortran in `sqmatw.f`.
- Compile and run the benchmarks `sqmat`, `sqmatf`, `sqmatb`, `sqmatw`, and `sqblas` on `gp004` and `tempest` using `make COMP=suncc plot` and `gnuplot`.

- It is actually more important to find an algorithm with a better FP:M ratio than saving more floating point operations, since memory access is expensive. Thus, reducing the number of floating point operations should (hopefully) also reduce the number of distinct memory locations referenced. Determine the FP:M ratio of the basic square matrix multiply and compare it to the FP:M ratio of your implementation of Winograd's algorithm.

8 Plotting Help

To plot the data files of a set of benchmark programs, run `./timing.sh prog1 prog2...` followed by `gnuplot -persist timing.gnuplot`. Note that `gnuplot` is not available on `tempest`, so you should run `gnuplot` on `pamd` via another `xterm`.

To produce PNG graphics files of the plots for your report, edit `timing.gnuplot` and change the first part to:

```
set term png; set output 'myplot.png'; set grid; set xlabel 'Dim'; ...
```

Then run `gnuplot timing.gnuplot` to create the `myplot.png` file.