1 Introduction

1.1 HPC Account Setup

For this assignment you need an account on the FSU HPC cluster:

• If you are an SC student and you don’t have an account on the HPC yet, then:
  1. Use your favorite web browser to go to 
     \url{http://www.hpc.fsu.edu}
  2. Select ”Connecting” under ”Getting Started” on the left.
  3. Select ”HPC Account Application Details”
  4. Select ”Apply For An Account”
  5. Select ”Scientific Computing” from the drop-down list
  6. Select ”Non-Faculty Account” from the drop-down list
  7. Enter you FSUID and the rest of the information (memorize your password)
  8. Carefully read and accept the HPC user policies and select ”Request Account”

• If you are a CS student (or other department), then:
  1. Use your favorite web browser to go to 
     \url{http://www.hpc.fsu.edu}
  2. Select ”Connecting” under ”Getting Started” on the left.
  3. Select ”HPC Account Application Details”
  4. Select ”Apply For An Account”
  5. Select ”General Access” from the drop-down list
6. Select “Sponsored Account” from the drop-down list
7. Select ”van Engelen” from the Sponsor drop-down list
8. Enter you FSUID and the rest of the information (memorize your password)
9. Carefully read and accept the HPC user policies and select ”Request Account”

1.2 Login Procedure

Once your account is approved and activated (you will receive an email notification), you can log in to the ”submit” node of the HPC cluster as follows:

    ssh -Y submit.hpc.fsu.edu

Best is to log in from an Xterm, since we need to run the Sun Studio GUI and gnuplot. The ”submit” node should be primarily used to edit files and run GUI tools. If you haven’t done already, create or edit the .tcshrc (if you’re using tcsh), or .cshrc (if you’re using csh), or .bashrc (if you’re using bash) file and add this line for tcsh/csh users:

    source /usr/local/profile.d/sunstudio.csh

or for bash users:

    source /usr/local/profile.d/sunstudio.sh

We need to source that script to set the environment for Sun Studio tools and compilers. Open a new Xterm

    [yourname@submit ~]$ xterm &

and log in to the compute node via our classroom queue:

    [yourname@submit ~]$ msub -I -l nodes=1:ppn=4 -q sc_classroom
    qsub: waiting for job 3089994.torque.local to start
    qsub: job 3089994.torque.local ready
    [yourname@hpc-5-2 ~]$
You’re now working on one of the HPC compute nodes in interactive mode. Both the submit and compute nodes share the same file system (NFS). We access this node only to run jobs interactively and to collect profile statistics. We use the "submit" node to edit files (in a separate Xterm):

- **submit.hpc.fsu.edu node**: first node to log in with `ssh`, use only for editing files and to run Sun Studio GUI tools.

- **hpc-5-2 compute node**: next node to log in via the `sc_classroom` queue with `msub -I` to compile sources, run the executable, and/or collect profile statistics.

Note that the `sc_classroom` queue controls and possibly limits access to the compute nodes. This means that an overload of users causes further logins to be blocked until others log out. **Please only open one terminal to the compute node via msub to prevent others from being unnecessarily blocked.**

### 1.3 Download the Project Files

Download the project source code:

```
[yourname@submit ~]$ wget 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
• **timer.c**: timer precision test
• **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

In this project we will study the use of program timers, investigate and measure the impact of compiler optimizations, use program code hints, and apply algorithmic changes to improve the performance of a small numerical application: matrix multiplication.

The following topics will be investigated:

• To benchmark our performance improvements, we need to select an accurate timer to conduct our experiments.
• We will compile with increasing levels of optimizations and add compiler hints via program annotations to improve performance.
• To study the differences in programming languages and compiler optimizations, we will compare the performance of matrix multiply written in C versus Fortran.
• We apply loop blocking techniques to improve the performance of matrix multiply for larger data sets.
• The performance of our matrix multiply implementations is compared to BLAS DGEMM of the Sun Performance libraries.
• To understand the impact of algorithmic differences, we will implement an alternative formulation of matrix multiply using Winograd’s algorithm.

These parts are explained in more detail in the next sections.
1.5 Write a Project 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 parts listed in the sections below and also list the source code of the programs you wrote or modified. You should only submit the report. You do not need to submit your programs for evaluation.

2 Selecting an Accurate Timer for Benchmarking

The \texttt{cputime()} function defined for you in \texttt{cputime.h} and \texttt{cputime.c} returns the CPU or wall-clock time (depending on the choice of timer) in seconds, which is measured from the previous call to this function to the next\(^1\). We use the \texttt{cputime()} function to determine the number of floating point operations per second (MFlops “megaflops”) of our square matrix multiply routine \texttt{sqmat_mult()}, by measuring the elapsed time of \(k\) calls to \texttt{sqmat_mult()}:

\[
\text{MFlops} = \frac{2kn^3}{10^6 \cdot \text{cputime}}
\]

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 \texttt{MINRUNS=2} and \texttt{MINSECS=0.1} as defined in \texttt{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 critical observations 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 result 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 always win! But that would not be desirable as it is very slow in this case.

---

\(^1\)Important: You cannot invoke \texttt{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.
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 for matrix multiply (1) and use it as a scaled measure of performance of 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 study the accuracy of timers, first follow these steps to test the default TIMES timer:

- Log in to the HPC compute node.
- Chdir to the directory where you installed the content of Pr1.zip
- Do make timer to build a timer test program.
- Run ./timer

You will see:

Timer resolution is 0.010000000 seconds
With MINSEC=0.1 timing precision is at least 1 digit(s)

The precision of the default timer is only one digit when our benchmarks run for 0.1 second! By keeping MINSECS small we avoid very long waiting times for the benchmarks to complete. However, if the resolution of cputime is too low then the timing precision in digits $p_{\text{digits}}$ 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 $t_{\text{sec}}$ 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 accuracy 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**: uses times() to obtain CPU time (user + system time).
• **USE_GETRUSAGE**: uses `getrusage()` to obtain CPU time (user + system time).

• **USE_GETTIMEOFDAY**: uses `gettimeofday()` to obtain wall-clock time.

• **USE_RDTSC**: uses 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.

**Your reporting assignments:**

1. Modify the make file to test different timers by setting the `-D` compiler option specified in the file `make.x86_64-unknown-linux-gnu-suncc`. For each timer tested, report its precision. To select a timer for `cputime()` use one of the above “defines” (C/C++ `#define` macros). A macro (C `#define` in C code) is also set with the compiler option `-D`. You should set these compiler options in the `make.x86_64-unknown-linux-gnu-suncc` file to rebuild the code.

2. Report the timer which gives the best wall-clock time accuracy and works naturally in a multi-threaded machine. We prefer wall-clock time over CPU time, especially when we deal with parallel programs as we will do in the next project. You should use that timer for all timing experiments from now on.

**IMPORTANT:**

1. **Work with clean builds**: always do a `make clean` whenever you change any file, including changes to the source files, scripts, and the make files (such as your modified `make.x86_64-unknown-linux-gnu-suncc` file). Then recompile with `make` (such as `make timer` to rebuild the timer).

2. **Make sure the system is dedicated**: 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 Compiler Optimization and Performance Benchmarking

The optimizations applied by the compiler to optimize the code are different from CPU to CPU. Instruction scheduling (such as modulo scheduling) by the compiler is especially important for the VLIW, EPIC (Itanium), and RISC architectures. Loop restructuring is
important for all systems to optimize code. 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 part of the assignment we will experiment with compiler optimizations and program annotations to speed up matrix multiply.

If in doubt about the compiler options, you may want to check out 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.

Your next reporting assignments:

1. On the compute node do make plot. After completion (this will take a while), go back to the submit node and execute gnuplot -persist timing.gnuplot to view the MFlops performance graphs (some programs do not show, which is normal as we will work on them later). 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.

   This graph shows the baseline performance, without optimizations and should be in your report.

   Remember to always use make clean when modifying the makefiles and program sources.

2. We need to find the hotspot in the code. To do so we use sample profiling. On the compute node run collect -p on -o sample.er ./sqmat. Then on the submit node execute analyzer sample.er.

   Using the analyzer, find the most time consuming function and the line(s) of code in that function that take most of the execution time. Which line(s) is/are this?

3. Change the make.x86_64-unknown-linux-gnu-suncc by adding the -fast optimization option for the C and Fortran compilers, COFLAGS and FOFLAGS, respectively. This option is similar to -O3 to optimize code.

   Use make clean and make plot to compile and run the tests on the compute node.

   Put the new graph in your report and estimate the speedup obtained with the -fast option compared to the baseline performance.
Profile the `sqmat` program and find the hotspot in the source. What (loop) optimizations were applied to this part of the code that explain the performance increases?

4. The compiler seems to perform optimization under "dynamic-alias-disambiguation", which means the code is optimized by checking for the absence of aliases at runtime. We can either add `restrict` qualifiers to the arguments as follows:

```c
sqmat_mult(const double *restrict A, const double *restrict B,
           double *restrict C, int n)
```

which is safe, since we do not use this function with overlapping input/output arrays (note that A and B can overlap or be the same as these are input only and thus cannot lead to flow/anti dependences). Or we can ask the compiler to figure out if there are any aliases based on structure layout using the optimization flags `-xrestrict -xalias_level=layout`. Add these to `COFLAGS` and `FOFLAGS`. It is fine to do both (use `restrict` in code and compiler options).

Use `make clean` and `make plot` to compile and run the tests on the compute node.

Put the new graph in your report and estimate the speedup obtained with these "restrict" options compared to the baseline performance.

Profile the `sqmat` program and find the hotspot in the source. What (loop) optimizations were applied to this part of the code that explain the performance increases? Did it help to inform the compiler of the absence of argument aliases?

5. The compute node supports vector operations ("multimedia extensions", such as SSE). To enable these, change the `make.x86_64-unknown-linux-gnu-suncc` by adding the `-xvector=simd` optimization option for the C and Fortran compilers, `COFLAGS` and `FOFLAGS`, respectively.

Use `make clean` and `make plot` to compile and run the tests on the compute node.

Put the new graph in your report and estimate the speedup obtained with the option `-xvector=simd` compared to the baseline performance.

Profile the `sqmat` program and find the hotspot in the source. What (loop) optimizations were applied to this part of the code that explain the performance increases?

From now on, we will continue to use the `-fast`, `-xrestrict -xalias_level=layout`, and `-xvector=simd` optimization flags for C and Fortran.

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.

**Your next reporting assignments:**

1. Edit `sqmat.f` and complete the function’s implementation of square matrix multiply. For the multiply loop, use the loop nest ordering \( j \) (outer), \( i \) (middle), and \( k \) (inner).

2. Now use the loop ordering \( j \) (outer), \( k \) (middle), and \( i \) (inner) loop and rerun the experiment.

   Compile and run with `make clean` and `make plot`.

   Put the new graph in your report.

   Something has happened to the performance. To figure out what, profile the `sqmat` program. There are many new functions. Take a look at function `sqmat` and its source (select the function, then click on "source"). It seems that that whole matrix multiply was replaced by the compiler, but can you figure out by what?

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.

**Your next reporting assignments:**

1. Determine how to use DGEMM in C by perusing `man dgemm`. Edit `sqblas.c` to call `dgemm` to perform a square matrix multiply.

2. Compile and run with `make clean` and `make plot`.

   Put the new graph in your report.

   From the graph compare performances of the `sqmat`, `sqmatf`, and `sqblas` benchmarks. Why has `sqblas` a similar shape as `sqmatf`?

6 **Blocking**

Loop blocking can be very effective to enhance performance by increasing the locality of memory access. Blocking improves locality of memory accesses to speed up the multiplication of large matrices. It is tricky however to find a good block size.
The block multiply is performed as follows:

```c
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 $C = A B$ block-wise for each $BLKSIZE \times BLKSIZE$ block located at $C_{i,j}$, $A_{i,k}$, and $B_{k,j}$.

**Your next reporting assignments:**

1. Edit `sqmatb.c` and implement a blocked version of matrix multiply. More information can be found in the file itself. Show your code in your report.

2. Determine the block size $BLKSIZE$ that works well. Report how you found your block size.

3. On the compute node run `collect -h dcm,on,ecdm,on -o hc.er` to investigate the L1 and L2 cache miss rates for `sqmat`. Use the analyzer to find the miss rate for the hotspot loop that assigns $C$ in `sqmat`. Repeat this for `sqmatb` and analyze the hotspot loop that assigns $C_b$ in `sqmatb`. What is the difference between the profile results for `sqmat` and `sqmatb`, if any?

4. There are certain matrix sizes for which the `sqmatb` code runs faster or slower as you can see from spikes in the performance graphs. What is a possible explanation of the slower parts and what could be causing these spikes?

### 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):

$$
\begin{align*}
    x_i &= \sum_{k=1}^{\lfloor n/2 \rfloor} A_{i,2k-1}A_{i,2k} \\
    y_j &= \sum_{k=1}^{\lfloor n/2 \rfloor} B_{2k-1,j}B_{2k,j}
\end{align*}
$$
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.

Your next reporting assignments:

1. Implement Winograd’s algorithm in Fortran in \texttt{sqmatw.f}.

2. Compile and run the benchmarks \texttt{sqmat, sqmatf, sqmatb, sqmatw,} and \texttt{sqblas} using \texttt{make plot} and \texttt{gnuplot}. Show the graphs in your report.

3. 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 \texttt{./timing.sh prog1 prog2...} followed by \texttt{gnuplot -persist timing.gnuplot}.

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

\[
\text{set term png; set output ’myplot.png’; set grid; set xlabel ’Dim’; ...}
\]

Then run \texttt{gnuplot timing.gnuplot} to create the \texttt{myplot.png} file.

- End