High-Performance Scientific Computing (MATH-GA 2011/ CSCI-GA 2945)

Homework Set 6

Out: October 20, 2012 · Due: November 1, 2012

Since all of you are by now itching to get started on your projects, this will be the last homework set. (We may issue more problem sets later purely for your entertainment.)

Problem 1: MPI Bugs

Oh no! Uncle Blaise has written more parallel programs, this time with MPI, and they're once again all wrong. Can you help him out? :) In the repository at `https://github.com/hpc12/hw6-problem1`, you'll find seven MPI programs, each of which has one or more bugs, detailed in the comments at the start of the source file.

Once again, note that these are well-known programs, and it's super-simple to find solutions for them on the web. We're aware of that. To state the obvious, you'll get more out of this problem if you try to do them on your own.

Turn in a fixed version of each program with the same file name as in the source repository above, all in a subdirectory '`problem-1`' of your repository. Try to make sure that the output of

```
diff -u original/mpi_bugN.c hpc12-hw6-netid123/problem-1/mpi_bugN.c
```

makes sense, because that's what I'll be looking at.

Here's a summary of what's wrong with each program, and what the fixed version should do:

Program	Desired behavior
`mpi_bug1.c`	The message should be passed without any rank encountering a hang.
`mpi_bug2.c`	The message should be passed *without* undefined behavior–no wrong data, no crashes, etc.
`mpi_bug3.c`	The message should be passed successfully.
`mpi_bug4.c`	The final sum should be computed correctly. The code refers to `mpi_array.c`, which is present in the repository.
`mpi_bug5.c`	The program contains an endless loop. The endless loop will cease to make progress after a few thousand iterations. In a comment in the code, explain what's going on, referring to sections of the MPI standard as necessary. Next, ensure that the program makes progress without long hangs.
`mpi_bug6.c`	Requires 4 ranks. Both pairs of tests should complete without an error.
`mpi_bug7.c`	The broadcast should not hang.

Problem 2: Faster matrix multiplication

This problem is a more performance-conscious take on matrix multiplication, improving on what you did in homework 1. In particular, we will be a bit more mindful of processor architecture and the availability and size of close memory. As we saw in class, matrix multiplication–because of its theoretically high arithmetic intensity–stands a good chance of making use of this.

Some of the lower-level tuning techniques mentioned throughout will make more sense once you've learned what we'll cover in the lecture next Wednesday, but the problem is entirely doable without that knowledge. Also, in case you are wondering, these instructions may seem fairly lengthy, but the actual code you write will turn out relatively short and straightforward.

To make better use of close and far memories, we will be using two levels of blocking in our code, where each successive level of blocking corresponds to memory at an increasing distance from the processor. We'll see on Wednesday that these, aside from registers, are different-level 'caches', called "L1", "L2", and so on, up to "LLC", the 'last-level-cache'.

For this assignment, matrix multiplication is supposed to carry out this operation–note that we're adding onto the existing value of C:

```
for i = 1 to n
  for j = 1 to n
    for k = 1 to n
      C[i,j] = C[i,j] + A[i,k] * B[k,j]
```

All matrices in this assignment are stored in column-major order.

a) Write a test and timing harness (perhaps based on your work for homework 1). This should be your main program, which takes three arguments. We'll call the arguments `n`, `ntrips`, and `use_variant`.

- `n` is the size of the matrices to use.
- `ntrips` is the number of repetitions to perform inside the timing loop.
- `use_variant` is an integer which decides which variant of the matrix multiplication to call.

These variants referred to by the last parameter are (`use_variant == 0`)

```
void dgemm_simple(
  const int M, const double *A, const double *B, double *C)
```

and (`use_variant == 1`)

```
void dgemm_tuned(
  const int M, const double *A, const double *B, double *C)
```

and a few more as mentioned below.

For now, leave the definition of `dgemm_tuned` empty, and fill `dgemm_simple` with a simple triple-loop matrix multiplication. You may use `dgemm_simple` as the reference calculation to check against. (So initially you'll be checking `dgemm_simple` against itself–that's ok.)

Your program should then do the following, in order:

- Allocate and fill square of size $n \times n$ random numbers drawn reasonably uniformly from the interval $[1, 2)$.
- Run a test on the chosen variant, to make sure its results match the known-good calculation. Check that the results match to a tolerance of 10^{-5}. Because of the addition semantics on `C`, make sure to initialize `C` with zero before starting the test.
- Run `ntrips` matrix multiplications using the chosen variant.
- Output a timing of the previous step in units of millions of floating point operations per second. (MFlops/s) You may use the approximation that a matrix multiplication performs $2n^3$ flops.

By the way, if you're wondering about the funny name, `DGEMM` is the name commonly used by the BLAS[1] for 'double-precision general-form matrix multiply'.

Run your code with the simple variant for the following matrix sizes

> 31, 32, 96, 97, 127, 128, 129, 191, 192, 229, 255, 256, 257, 319, 320, 321, 417, 479, 480, 511, 512, 639, 640, 767, 768, 769, 1024

and report your performance findings. A shell command like the following can help you automate this task:

```
for n in 31 32 96 97 127 128 129 191 192 229 \
255 256 257 319 320 321 417 479 480 511 512 639 \
640 767 768 769 1024; do
  ./matmul $n 10 0
done
```

(Copy and paste is your friend here–you can even automate this by using a shell script. Shell scripting will be tool of the week on October 31.)

Report performance in the following format:

```
printf("use_variant: %d size: %d - performance: %g MFlops/s\n",
...);
```

Make sure to use the same machine for these performance numbers and the ones you report further down. In your report, please also provide the output of

```
cat /proc/cpuinfo | grep "model name"
```

(or find your processor model number in some other way). Keep repeated lines in the output. Also state the output of

```
uname -a
```

and whether you are running inside a virtual machine.

b) Write and benchmark a fixed-size lowest-level square matrix multiplication. The intention is for this to become the building block for the higher levels. The amount of data referenced by this routine should be a few kilobytes, and definitely not more than fits into the L1 cache of the machine you are targeting. Go with perhaps 8 kiB as a target value, for the subblocks of `A`, `B`, and `C` combined.

Use the following prototype for your function:

```
static void dgemm_lowest(
    const double*restrict A, const double*restrict B, double*restrict C)
```

Make sure to stick to this prototype. Notice the following things:

- There is no size passed to this function. This is a fixed-size computational kernel. The block size processed by this function should be a global constant, which we'll call `L1_BLK_SIZE`. The benefit of a fixed-size routine is that the compiler can do a better job generating code, simply because the loop bounds and strides are known a priori.

[1] `https://en.wikipedia.org/wiki/Basic_Linear_Algebra_Subprograms`

- `static` allows the compiler to do perform inter-procedural optimizations (because it guarantees that the compiler knows about all sites that call this function).
- `restrict` is a promise about something called 'pointer aliasing' which we'll cover next Wednesday.

For this routine to be useful to higher levels, it should be *adding* its result to `C`, just like the top-level routine.

Interface your test harness from the previous part to `dgemm_lowest`, by writing code that does the following:

- Check that the right-size matrix is being passed in.
- Copy the input matrices to the temporary arrays `a_block` and `b_block`, using the routines and data provided in this code snippet[2]. Note that the storage size is dictated by `L2_BLK_SIZE`, by ways of the value `ANOTHER_INTEGER` in this code, which we haven't fixed yet. Just set this to 1 for now.
- Make sure that the routine passes the correctness test.
- Run `dgemm_lowest` for `ntrips` loops.
- Output timing information as above.

Make the value `use_variant==2` enter this part of your code.

Carry out a systematic study of:

- loop ordering (realize that the loops *can* be interchanged without changing the meaning)
- block size

Write about your performance findings in your report, and (as a second step) try to explain them in terms of the pipelining issues we'll learn about next Wednesday.

c) Include an assembly listing of your lowest-level routine in your report. Finding this routine may require some searching, as the compiler may have '*inlined*' the code into a higher-level routine. (If you need help, try removing the `static` keyword. Make sure to add it back later.) Comment each line of what you believe to be the core computational loop with what you think the code is doing at that point. Use Google to find the meaning of instructions you don't understand.

d) Write a second-level matrix multiply that uses your lowest-level routine in the fashion depicted in the lecture slides for lecture 7, PDF pages 88–91. Note that this routine still works on a fixed size, given by `L2_BLK_SIZE`, which is an integer multiple of `L1_BLK_SIZE`.

Interface this code to the test harness as above, using `use_variant==3` as the switch value to enter this branch.

Again carry out a systematic study of:

- (block) loop ordering
- block size multiple (called `ANOTHER_INTEGER` in the code snippet given above).

Write about your performance findings in your report.

[2] `https://gist.github.com/4045f6dabd1d04192098`

e) Write a top-level matrix multiplication routine that uses your second-level routine, once again in the same fashion as described in the slides. This code should go into `dgemm_tuned`, which you left empty above.

Note that this routine will have to handle *all* matrix sizes, requiring you to handle computational blocking boundary cases. The recommended way of doing so is by filling unoccupied spots in and `a_block`, `b_block` with zeros, and only copying out the parts of `c_block` that are desired in the output matrix. Convince yourself that this does the right thing mathematically.

You may once again use the blocking helper routines given in the code snippet.

Report performance for the same matrix sizes as in part 1. Try and interpret your results.

f) Add OpenMP-based parallelization to your code, and check if you see a benefit. Answer the following questions in your report:

i. Which is the best level on which to add parallelization?

ii. Which loops (`i`, `j`, `k`?) are parallelizable, which ones are not?

iii. Report performance for the same matrix sizes as in part 1.

Turn in:

- a file `problem-2/dgemm.c` containing all the code described in this assignment.
- `problem-2/Makefile` that builds your code with your chosen compiler and flags.
- `problem-2/report.txt`

Hints:

- While not as satisfying as being able to see what manual optimizations cause what sort of speed-up, compiler flags can also make a large difference in execution speed.
  - `-O3` is your basic 'go-fast' option.
  - `-march=native -mtune=native` tell gcc to tune for the machine you are currently using.
  - `-ftree-vectorize`—see here[3] for more.

  Check this[4] and this[5] link for an overview of other options.

- It's somewhat more instructive to do this exercise on a real machine compared to the 'fake' one presented by the virtual machine. On my computer, the 64-bit virtual machine is about 20% slower than the 'real' processor, whereas the 32-bit machine loses about 50%. (32-bit Intel processors have a tiny register file–just 8 general purpose registers. That's one reason why the 32-bit machine does so poorly. If your OS is just 32-bit, you are effectively running your computer with the handbrake on–FYI. :)

  Part of the reason for this is that the virtual machine simulates a simpler processor, so the '`native`' tuning flags do not have their full range of benefits.

[3] `http://gcc.gnu.org/projects/tree-ssa/vectorization.html`
[4] `http://gcc.gnu.org/onlinedocs/gcc-4.7.2/gcc/i386-and-x86_002d64-Options.html`
[5] `http://gcc.gnu.org/onlinedocs/gcc-4.7.2/gcc/Invoking-GCC.html`

If you don't have access to a real Linux machine with a new gcc (version 4.7 or newer), you may want to consider using compute nodes on `cuda` or `bowery` for this exercise. Unfortunately, the gcc installed there is old and generates very slow code. You may therefore want to investigate using the Intel compiler. Use

```
module load intel/11.1.046
```

on Bowery and

```
module load intel/11.1.083
```

on cuda to make it available as `icc`. Then check this[6] link for an overview of optimization options.

- Once we've discussed caches, it may be helpful to use `valgrind --tool=cachegrind` to see the cache behavior of your code. To make sure the observed values correlate with your actual processor, find its cache sizes as shown in next Wednesday's class and match its parameters in Valgrind using

```
--D1=<size>,<assoc>,<line_size>
--LL=<size>,<assoc>,<line_size>
```

(You should match Valgrind's LLC to your L2 cache.)

- We'll hold a (light-hearted, not terribly serious) contest for the fastest code. We'll be using the compute nodes on the CUDA cluster for comparison. If you'd like to enter the contest, have a `Makefile` target `matmul-contest` that builds on the CUDA cluster, likely with the Intel compiler. You can use a compiler argument `-DCONTEST` and `#ifdef CONTEST/#else/#endif` in your source code to write contest-specific code or parameter values, should you need to.

  Entries will be ranked by their average performance across all of the above matrix sizes, and the top three entries will receive a bag of M&Ms.

[6] `http://software.intel.com/sites/default/files/compiler_qrg12.pdf`