CS 140 Assignment 1:
Performance Analysis of a Sequential Program

Assigned Monday, January 9, 2017

Due 11:55 pm Tuesday, January 17, 2017

The reason anyone uses a parallel computer, or writes a parallel program, is always performance: You want your computation to run faster, or cheaper, or on a bigger problem. Every program you write in this class will have performance as one of its goals.

This assignment will give you some practice measuring and maybe improving the performance of a program. The program isn't parallel; we'll get to that next week, but for now the program just runs on one processor. If you're trying to get good performance on a parallel code, the first step is always to make sure you're getting the best performance you can on the single-processor sequential code. The moral of this assignment is that performance is affected by lots of subtle things, some of which you can explain and some not.

We have supplied a program to multiply two matrices together. We'll look at some applications of this important computation later in the course, but for now it's just a particular task that we want to get to run fast. You will benchmark this code and some modifications of it, and you will present your analysis of its performance and scaling in graphs, in tables, and in writing.

The Gauchospace site has a link to a video lecture by Saman Amarasinghe at MIT about optimizing matrix multiply. He starts with a somewhat different code than ours, but it's interesting to watch.

You can do this assignment on any computer you like, including your laptop or CSIL or a single processor of Triton.

1 Mathematical background

A square *matrix* is an n-by-n array A of n^2 numbers. The entry in row i, column j of A is written either a_{ij} or $A(i, j)$. The rows and columns are numbered from 1 to n in traditional linear algebra notation, or sometimes from 0 to $n - 1$ in programs like our C code. Two same-sized matrices A and B can be multiplied together to give a third matrix C. Recall the definition: If the matrix $C = AB$, the elements of C are

$$C(i, j) = \sum_{k=1}^{n} A(i, k) \times B(k, j).$$

In words, each element of C is obtained from one row of A and one column of B by computing an inner product (that is, by adding up the elementwise products). The element in row i and column j of C is the inner product of all of row i of A with all of column j of B.

Every element of A contributes to n different elements of C (the ones in the same row), and every element of B contributes to n different elements of C (the ones in the same column). The total amount of work to compute all n^2 elements of C is n^3 multiplications of numbers, and n^3

additions of numbers. We call an addition or multiplication of two numbers a *flop* (for "floating point operation"); thus, computing the product of two n-by-n matrices uses $2n^3$ flops.

You would expect, therefore, if you measure the running time of a matrix multiplication program for matrices of different sizes n, the measured time will scale as $O(n^3)$. This assignment is to evaluate that expectation.

2 The code

We've written the first version of the matrix multiplication code and linked it to the course home page. You can build it on any Linux/Unix/OS-X machine by saying `make` in the `code` directory. (You may modify the `Makefile` for your system if you need to.) You then run it by saying `./matrix_multiply`. You can use command-line switches to run on matrices of different sizes. You can also use command-line switches to choose different versions of the algorithm; except there's only one version in the code we supply.

Read the code. The header file `matrix_multiply.h` defines a `matrix_t` type as a struct whose most important field is a vector of `double` (64-bit floating point) matrix elements. All the elements of the matrix are stored contiguously in memory, in what's called *column major order*: the elements in the first column come first, then the elements in the second column, and so on. The header file also defines a macro `element(X,i,j)` that expands to a pointer to the element in the (i, j) position of matrix X.

The file `matrix_multiply.c` defines some routines to allocate, free, and print matrices, and also contains the matrix multiplication routine `matrix_multiply_run_1`. That simple routine just implements the equation above to compute the elements of C from the elements of A and B.

3 Computation cost and communication cost

Notice that the definition leaves a lot of freedom to do things in different orders. The computations of the different elements of C are independent of each other; all the element multiplications are independent; and all the additions could be done in a different order because addition is commutative and associative. At least in principle, you could do the $2n^3$ flops in many, many different orders, and still get the right answer. For example, you could nest the three loops "`for i`", "`for j`", "`for k`" in any order—six different orders in all—and still compute the same result.

Why should the order matter? You're still doing the same amount of arithmetic, that is, the same amount of computation. However, you might be able to change the number of times you move the matrix elements around. Under the covers, this program moves data (that is, matrix elements) between main memory, cache, and processor registers. A parallel program would also have to move data between the memories of different processors.

Here is the most important fact in this course, and I'm already giving it away to you on the first day of class:

> *Almost all modern computer programs spend more time and energy moving data around than actually computing on it.*

In other words, the cost of communication is usually larger—often very much larger—than the cost of computation. As we'll see this is true for parallel computers that move data around between processors. But it's also true for most single-processor sequential codes. The question you'll ask over and over again in this course, and indeed the key question in most of performance analysis, is: *Where's the data?*

4 What experiments to do

Okay, so what do you need to do for this homework?

1. First, measure the scaling of the code we gave you, on whatever computer you're using. Run the code for a dozen or so values of n up to the largest matrices you can multiply in a few minutes. A good choice would be to use powers of two, say 16, 32, 64, up to 1024 or 2048 or whatever. (On my 5-year-old Macbook Air, the original code takes about 10 seconds for $n = 1000$.) Make a table of running time as a function of n. Plot a graph of the same data (using Matlab or any other plotting software).

   Also, plot a graph of the flops/second rate as a function of n. It's often a good idea to study performance scaling by looking at how a suitable *computation rate* scales with problem size. In this case, if time is really proportional to $2n^3$, the flops/second plot will be a horizontal line.

   Is it? If not, can you explain why not? Write a paragraph of analysis.

2. Second, try out all six possible nesting orders of the `ijk` loops in `matrix_multiply`. For each one, try a few different matrix sizes, including the biggest you can. Make another plot of flops/second as a function of n, with both the best and the worst loop orders, for a range of powers of two similar to that in the first experiment.

   Write a paragraph of analysis. What do you see? Can you explain it? (It's okay if you can't explain what's going on—we will study some of these questions later in class!)

3. Third, for extra credit, try modifying the program in any way you want. The only restriction is that you can't call any library routines for matrix or vector computation. Can you do better than any of the six variations above? As a challenge, see how big you can get n and keep your running time under two minutes.

   On my Macbook, I wrote a pure C version that improves the 1000-by-1000 running time from 10 seconds to 1 second. It's possible to do a lot better than that, actually.

5 What to turn in

Turn in all of the following, through the course Gauchospace page:

- All your source code, including all the files in the `code` directory whether you've modified them or not.
- A report in PDF form named `report.pdf`, containing any necessary description of your code, tables of your run time results, your plots and graphics, plus a description and interpretation of your results and the conclusions you draw from them.
- If you worked in a team, your report should include the names and CSIL accounts (or perm number if you don't have an account) of both members.
- Don't turn in any executable or .o files.

You may do this assignment alone or in a group of two people. (Later assignments must be done in groups of two.) Only one person needs to turn it in, but make sure both names are on the report.