14 Matrix-Vector Multiplication

• Problems 8.7, 8.10, 8.11

• Note that there seem to be some errors in section 8.6.8. The data declarations for grid_coords don’t seem right.

• Problem 8.11 asks about matrix transposition. What would you do if you had to transpose a row-blocked matrix and have the transpose matrix also blocked by rows?
15 Matrix Multiplication

• The inner computation is

\[ c_{ij} = \sum_{k=0}^{n-1} a_{ik} b_{kj} \]

• The outer computation results in a triple loop, with a matrix

\[(n \times \ell) \times (\ell \times m) = (n \times m)\]

requiring

\[n \cdot \ell \cdot m\]

floating point multiplications

• To simplify matters, we usually think of this as just an \(O(n^3)\) operation, which is what it would be if the matrices were all square.

• Look at Quinn’s code. He gets about 220 MFLOPS on small matrices, but then only about 80 MFLOPS on larger matrices

• This is almost certainly the effect of cache size.

• If we’re doing \(A \times B\) as matrices, and all of \(B\) fits in the cache, then we’ll do well. Otherwise, we will suffer a performance hit.
● Because of this cache issue, we usually do not block entire rows of a matrix in a processor, but rather do a recursive checkerboard breakdown of the matrix, as in Figure 11.5. This way, we can subdivide into blocks until things fit in cache, and the subdivision is in fact recursive.

● Doing this, Quinn’s code maintains high performance even as matrix size increases.
16 Row-wise Block Stripping

• Let’s do the Foster analysis of how to parallelize matrix multiplication.

• Clearly, all the multiplications to compute $c_{ij}$ are independent and can be done in parallel. If we had $n^3$ processors, and we could distribute data in a fast way, we would do all the multiplications in parallel in three ticks (one to fetch $a_{ik}$, one to fetch $b_{ik}$, one to do the multiplication) and then add things up in log time.

• Agglomeration: What’s the best way to combine these too-tiny mults together?

• Let’s look at the mults needed to do a single row of the result matrix.

• Let all tasks correspond to the same (by subscript) rows of matrices $A$, $B$, and $C$.

• But no single recursively-defined block is sufficient to compute an entire element of $C$, so we will have to move some data around.

• page 280, Figure 11.7

• We pass rows of the $B$ matrix to all tasks, and after $p$ steps we have seen all of the $B$ matrix
16.1 Analysis

• Let’s work with $n \times n$ matrices, and assume $p|n$ for convenience

• We initialize an $n/p \times n$ block of $C$ to zero

• Multiply $n/p \times n/p$ chunk of $A$ times an $n/p \times n$ chunk of $B$

• Add this into the running $C$

• Let $\chi$ be the cost of a multiplication followed by an addition

• Then the cost is

$$\chi \cdot \left( \frac{n}{p} \right) \cdot \left( \frac{n}{p} \right) \cdot n = \frac{\chi n^3}{p^2}$$

in CPU time

• The communication cost per transmission is

$$\lambda + (n/p)(n/\beta)$$

if done without overlap

• We have $p$ iterations of this outer loop, so we have $\chi n^3/p$ compute time and $p\lambda + n^2/\beta$ communication time
ISOEFFICIENCY

- The sequential algorithm is $\Theta(n^3)$
- The communication of the parallel algorithm is $\Theta(n^2)$
- We need for isoeficiency to have

  $$n^3 \geq Cpn^2$$

  which means

  $$n \geq Cp$$

  and once again this is bad, because memory on the host increases as $n^2$ as the $n \times n$ matrix increases in size.

- Overlap of data and computation: We can, in this method, do the overlap since the $n^3$ computation time is rather large in and of itself.
16.2 Another Idea

If you touch data, then you should use it as much as possible. In this algorithm we are moving

\[
\left(\frac{n}{p}\right) \cdot n = \frac{n^2}{p}
\]

elements in order to do

\[
\left(\frac{n}{p}\right) \cdot \left(\frac{n}{p}\right) \cdot n \cdot (\text{mult} + \text{add})
\]

operations. This comes to

\[
\frac{2n^3/p^2}{n^2/p} = \frac{2n}{p}
\]

FLOPS per data item moved.
16.3 Cannon’s Algorithm

- Do the checkerboard arrangement, which also happens to match up with the recursive breakdown of the matrix for caching
- \((n/\sqrt{p}) \times (n/\sqrt{p})\) multiplications, so \(\chi n^3/p^{3/2}\) mults per block
- There are \(\sqrt{p}\) iterations so the total compute time is

\[
\frac{\chi n^3}{p}
\]

and thus no loss of compute time over the sequential algorithm

- We assume bandwidth of \(\beta\) bytes per timestep, so 1/\(\beta\) to send a single element

- The initial distribution of data takes

\[
2 \left( \lambda + \frac{n^2}{p\beta} \right)
\]

since we can only send or receive each tick

- Each iteration, we need to move and receive a block of \(A\) and \(B\), which is a cost of

\[
2\sqrt{p} \left( \lambda + \frac{n^2}{p\beta} \right)
\]

so the total communication cost is

\[
2(\sqrt{p} + 1) \left( \lambda + \frac{n^2}{p\beta} \right)
\]
ISOEFFICIENCY

• $n^3$ sequential computation time

• Overhead is $p$ processors times $n^2 / \sqrt{p}$ equals $n^2 \sqrt{p}$

• We need

$$n^3 \geq C n^2 \sqrt{p}$$

which means

$$n \geq C \sqrt{p}$$

and this is now scalable. As $p$ increases, $n$ grows like $\sqrt{p}$, so memory $n^2$ grows like $p$, since the memory per processor will be constant

• And we can still overlap the data movement with the computation

• Further, we move $2n^2 / p$ elements (one block of $A$ and one of $B$), and we get to do $n^3 / p^{3/2}$ FLOPS. This works out to

$$\frac{n^3 / p^{3/2}}{2n^2 / p} = \frac{n}{2\sqrt{p}}$$

FLOPS per data move; much better than before.

• Problems 11.2, 11.3, 11.4