

# CS 140: Analysis of matrix-vector multiplication

January 24, 2008

Here is a summary of the analysis that I went through in class today for matrix-vector multiplication. I ought to know better than to try to do algebra on the board in real time. . . .

**The problem:**  $A$  is an  $n$ -by- $n$  matrix and  $x$  is a column vector. We are to compute  $y = A * x$  using  $p$  processors  $P_0, \dots, P_{p-1}$ . We will assume  $n$  is divisible by  $p$ . (A real library code wouldn't make this assumption and would be a little more complicated because of it, but the analysis would be essentially the same.)

**The algorithm:** We partition  $A$  among the processors by rows, so that processor  $P_k$  has rows  $kn/p + 1$  through  $(k + 1)n/p$ . We store all of  $x$  on  $P_0$ , and we will finish with all of  $y$  on  $P_0$ . There are three steps:

1. Send all of  $x$  to every processor (method A: use  $p - 1$  `MPI_send/MPI_recv`'s of  $n$  words each; method B: use one `MPI_Bcast` of  $n$  words with  $p - 1$  destinations).
2. Each processor uses  $x$  with its rows of  $A$  to compute its elements of  $y$  (no communication,  $n^2/p$  computation per processor).
3. Each processor sends its piece of  $y$  to  $P_0$ , which concatenates them. (method A: use  $p - 1$  `MPI_send/MPI_recv`'s of  $n$  words each; method B: use one `MPI_Reduce` of  $n/p$  words per processor).

**The analysis:** We will measure computation time as number of scalar multiplications. Thus the sequential time is

$$t_{\text{seq}} = n^2.$$

In the parallel algorithm each processor does  $n^2/p$  multiplications, so the maximum is the parallel computation time,

$$t_{\text{comp}} = n^2/p.$$

We will analyze communication two ways: First, we will count total communication in big-O terms; second, we will count each message and account for which messages happen at the same time.

**First way (total communication):** Step 1 sends a total of  $(p - 1)n$  words, and step 3 sends a total of  $(p - 1)n/p$  words, by either Method A or Method B. This adds up to  $pn - n/p$  words; this gives a total communication volume of  $O(pn)$ , so by this method we have

$$t_{\text{comm}} = pn$$

(ignoring lower-order terms and constant factors). Now parallel time is modeled as

$$t_{\text{par}} = t_{\text{comp}} + t_{\text{comm}} = n^2/p + pn.$$

We can see from this that when  $p$  is small enough, the first term dominates and the algorithm will get faster for fixed  $n$  as  $p$  increases. If  $p$  gets much larger the second term will dominate and performance will get worse with larger  $p$ , as communication dominates computation. We can't tell exactly where the tradeoff is without knowing the constant factors, but it will be when the two terms are on the same order, which is when  $p$  is on the order of  $\sqrt{n}$ .

Another way to see this is to compute the speedup, which is defined as

$$\text{speedup} = t_{\text{seq}}/t_{\text{par}} = \frac{n^2}{n^2/p + pn},$$

or better yet the parallel efficiency, which is defined as

$$\text{efficiency} = \text{speedup}/p;$$

we can express efficiency as a percentage and get an idea of how good our algorithm is. We have

$$\text{efficiency} = \frac{1}{p} \cdot \frac{n^2}{n^2/p + pn} = \frac{1}{1 + p^2/n}.$$

(You can check the algebra.) I like to express parallel efficiency as  $1/(1 + \text{something})$ , because then you can see clearly what is limiting the efficiency to be less than 1. In this case, if  $n$  is large compared to  $p^2$  (equivalently, if  $p$  is small compared to  $\sqrt{n}$ ), then  $p^2/n$  is small, and the efficiency is close to 1; if  $p^2/n$  starts to get larger, the efficiency drops.

**Second way (model each message):** Now let's compute  $t_{\text{comm}}$  using the more detailed approach where we model the cost of a message of length  $w$  as  $t_{\text{startup}} + wt_{\text{data}}$ , and assume that messages involving different processors can happen at the same time (possibly an overoptimistic assumption). This will be more complicated but it will give us a little better idea of how the efficiency depends on message time. With this approach, it will make a difference whether the algorithm uses sends and receives (Method A) or broadcasts and gathers (Method B). So, here goes...

**Second way, send/recieve algorithm:** We'll abbreviate  $t_{\text{startup}}$  and  $t_{\text{data}}$  as  $t_{\text{s}}$  and  $t_{\text{d}}$ . In step (1), each send/receive costs  $t_{\text{s}} + nt_{\text{d}}$ . Processors  $P_1$  through  $P_{p-1}$  may be doing receives at the same time, but processor  $P_0$  does the  $p-1$  sends one after the other. Thus  $P_0$  is the bottleneck, and it takes time  $(p-1)(t_{\text{s}} + nt_{\text{d}})$  for step (1). Processor  $P_0$  is also the bottleneck for step (3), taking time  $(p-1)(t_{\text{s}} + (n/p)t_{\text{d}})$ . Ignoring low-order terms (but not constant factors this time), we replace  $(p-1)$  by  $p$  and we replace  $pn + n$  by  $pn$ , and have  $t_{\text{comm}} = p(t_{\text{s}} + nt_{\text{d}}) + p(t_{\text{s}} + (n/p)t_{\text{d}}) = 2pt_{\text{s}} + pnt_{\text{d}}$ . Now  $t_{\text{comp}}$  is still  $n^2/p$ , so we have

$$t_{\text{par}} = t_{\text{comp}} + t_{\text{comm}} = n^2/p + 2pt_{\text{s}} + pnt_{\text{d}}.$$

Since  $t_{\text{seq}}$  is still  $n^2$ , we have

$$\text{efficiency} = \frac{t_{\text{seq}}}{p \cdot t_{\text{par}}} = \frac{n^2}{p \cdot (n^2/p + 2pt_{\text{s}} + pnt_{\text{d}})} = \frac{1}{1 + (2p^2/n^2)t_{\text{s}} + (p^2/n)t_{\text{d}}}.$$

As  $n$  goes to infinity, the  $(p^2/n)t_{\text{d}}$  term dominates the  $(2p^2/n^2)t_{\text{s}}$  term, but since the constant  $t_{\text{s}}$  is larger than the constant  $t_{\text{d}}$  (perhaps by a factor of 1000 or so) we keep both terms in the parallel efficiency.

**Second way, broadcast/gather algorithm:** Finally, we count messages in the broadcast/gather version of the basic parallel algorithm. This depends on how the `MPI_Bcast` and

`MPI_Gather` library routines are implemented. In practice, we hope that they are implemented as efficiently as possible for each different parallel machine we run on. In theory, we will model both broadcast and gather as if they use a binary tree of processors: a broadcast from processor  $P_0$ , for example, first sends the data from  $P_0$  to  $P_1$  and  $P_2$ ; then  $P_1$  sends the data to  $P_3$  and  $P_4$  while  $P_2$  sends to  $P_5$  and  $P_6$ ; and so on. At each stage two messages are sent, and after  $\log_2 p$  stages every processor has received the message. Thus we model the cost of broadcasting  $n$  words from  $P_0$  as  $(2 \log_2 p)t_s + (2n \log_2 p)t_d$ . The gather is similar; it costs  $(2 \log_2 p)t_s + (2(n/p) \log_2 p)t_d$ . Then we have

$$t_{\text{par}} = t_{\text{comp}} + t_{\text{comm}} = n^2/p + (4 \log_2 p)t_s + (2n \log p)t_d$$

(noting that the  $t_d$  term in the gather cost is lower-order). Thus,

$$\text{efficiency} = \frac{t_{\text{seq}}}{p \cdot t_{\text{par}}} = \frac{1}{1 + ((4p \log_2 p)/n^2)t_s + ((2p \log_2 p)/n)t_d}.$$

Untangling the terms in the denominator that limit parallel efficiency, the broadcast/gather algorithm replaces a factor of  $p$  by a factor of  $2 \log p$ . If  $p$  is small this is about the same, but for larger  $p$  it should make a difference. Part of Homework 3 is to see whether you can detect this difference.