Lecture 5

1 Overview.

Today, we give a simple model for parallel computers, design a simple algorithm to run on the computer, and describe an implementation of a restricted version of the model.

2 A simple model for parallel computers.

We will use Valiant's Bulk Synchronous Processing model of parallel computers. In this model, the computer proceeds in steps where one can compute for some time and have a round of communication. The computer is parameterized by p, the number of processors, L the "latency" parameter, and g the bandwidth parameter.

In each step takes time,

$$C_{\max} + (L + h_{\max} g),$$

where $C_{\max}$ is the maximum computation time over all processors, and $h_{\max}$ is the maximum number of messages sent or recieved by any one processor.

Intuitively, g is giving you cost per unit message, while L is giving you fixed cost per step (perhaps measuring latency.)

3 A simple application.

Physicial simulation problems often consists of a graph of some form and a computation that proceeds by exchanging information along the edges of this graph and updating values at the nodes. For example, when one simulates the weather generally activity at one "grid point" depends on the activity at neighboring grid points.

A canonical view of this is a repeated matrix-vector multiplication, where when one views the nonzeros in the matrix as the adjacency matrix of the graph. (The computation of eignvectors, or the Gauss-Seidel iteration, and even conjugate-gradient methods for solving linear systems follow this structure.)

A bit more precisely, we are given a sparse matrix A of dimension n, a vector x and wish to compute $x^t = A^t x$. (By the way, repeated squaring is a bad idea since the matrix becomes dense.)

We consider the graph $G = (V, E)$ be an n-node an edge (u, v) when $A[u, v]$ is nonzero, we define $w(u, v)$ to be $A[u, v]$. (We assume the matrix is symetric and thus the graph is undirected.) Furthermore, x_v^t denotes the value of the vth element of the vectore x^t. Clearly,

$$x_v^{t+1} = \sum_{e=(u,v)} w(u,v) x_u^t.$$

Thus, we define the neighbors of v, $N(v)$, to be the set of nodes u other than v with an edge to v, and the neighbors of a set of nodes S, $N(S)$, to be the neighbors of any node in S but not in S.

This computation is local in the sense that each value at step $t+1$ depends only on values at step t. Moreover, for any set of nodes S, the values only depend on the values of S and the values of the neighbors of S, $N(S)$. This leads to a natural parallel implementation.

- The nodes V are partitioned into $S_0, \ldots, S_p$, where S_i are the home nodes of processor i.
- At each timestep t, each processor has all the values of its home nodes, and all the values for $N(S_i)$.

With this invariant, the computation can proceed by computing its new values locally, and then sending each value to whomever needs a copy.

The total number of messages received by a processor i is at most $|N(S_i)|$ and the total number of messages sent by a processor i is at most $d|N(V - S_i)|$ since any node in S_i that has a neighbor in $V - S_i$ to d other processors (where d is the maximum degree of any node in the graph.) Thus, the total communication time is

$$(\max_{0 \le i \le p} |E_i| + L + g \max_{0 \le i \le p} \max(dN(V - S_i), N(S_i))$$

where E_i is the number of edges incident to processor i in each step of the itrative relaxation.

What is the goal here? We wish the runtime to be as small as possible. Thus, each $|E_i|$ should be around $|E|/p$, and $|N(S_i|$ be kept as small as possible. Clearly these may be competing concerns.

For some types of matrices that arise in practice, this can be done in a reasonable manner. Typically, in physical simulations, the volume (or the amount of computation) grows faster than the surface area. For example, let's consider a 2-dimensional grid. Here, an $N = k \times k$-node grid can be divided into p pieces (assume that p is a perfect square), such that each piece has size N/p and for each piece $N(S_i)$ (and $N(V - S_i)$) has size $4k/\sqrt{p} = 4\sqrt{N/p}$. Thus, the running time becomes

$$T(4N/p + g4\sqrt{N}/p + L).$$

When is this efficient? Well, one notion is that the total speedup over a "good" sequential algorithm is about equal to the number of processors, let's say within a factor of 2. Notice, in this case, that the extra overhead above is what we pay for communication. Notice, further that as N gets larger this term gets relatively smaller compared to the N/p computation that one performs in each step. So, by choosing a large enough N, we will obtain an "efficient" parallel algorithm.

In particular, our "efficiency" is above 50% when

$$4N/p > L + g4\sqrt{N/p}.$$

Or in rough terms, our algorithm is efficient when $N >> Lp$ and $N >> g^2p$.

Question 1: If one had a 3 dimensional grid with $N = k \times k \times k$, how large would N have to be for one to get an efficient parallel algorithm. (Here, the latter form would be fine, i.e. $N >> \ldots$ and $N >> \ldots$.)

3.1 An aside.

Let me remark about this style of analysis. Does this make sense? Well, if a problem is very small why run it on a parallel computer? If on the other hand if it is very very large, one might presume that it is easy to parallelize (this is not always the case, but bear with me). So, one notion of quality of an algorithm is how large does your problem need to be, in order to produce a good speedup?

On the other hand, this model ignored things like overlapping communication and computation, and also bulked up communications into large chunks. We suggest that this only changes the interesting problem sizes by a bit. That is, hopefully the problem is large enough to get good speedup. If not, with all the other tricks one could possibly speedup on somewhat smaller problems. But, the complexity is likely not worth getting this perhaps small set of problems that could be sped up. (This is essentially the argument that Valiant made when he first described the BSP model.)