Computer Science 400
Parallel Processing
Siena College
Fall 2008

Topic Notes: Parallel Algorithms

Algorithm Design for Distributed Memory

When programming for a message-passing/distributed-memory environment, we have several considerations that will drive our approach:

- distribute the computational workload among the processes
- distribute the memory requirements among the processes
- minimize the interprocess communication
  - minimize the number of messages
  - minimize the volume of data transferred
  - maximize message concurrency
  - minimize the number of neighbors with which communication is needed

Recall that our motivations for parallel processing may be either computational speedup, computational scaling, or both.

When the motivation is on speedup, we want to focus on a balanced computational workload. When the motivation is scaling, we want to balance and minimize the per-process memory usage.

We always want to reduce communication as much as we can, since that is pure overhead introduced by our parallel implementation.

Foster's Design Methodology

Ian Foster suggests a design approach for parallel algorithm design. His four steps:

1. *Partition* the problem by dividing it into pieces that we'll call *primitive tasks*.
2. Determine how the primitive tasks need to communicate with each other. These can be
   - *local communication* where a primitive task will need information from some of its neighbors to continue computation,
   - *global communication* where many or all primitive tasks need to contribute.
3. *Agglomerate* the primitive tasks into groups that will be assigned together to a processor.

- we can reduce communication by agglomerating tasks that would need to communicate with each other

4. *Map* the agglomerated tasks to processors.
   - If the number of agglomerated tasks is equal to the number of processors, this is easy.
   - If there are more agglomerated tasks than processors, we will map them to balance the load and minimize communication

Let's think about this in terms of our Game of Life simulation.

Here, we can consider one update of one cell to be a primitive task. Each primitive task needs to communicate with the task representing itself in the previous step and the tasks representing each of its 8 neighbors from the previous step to perform its update.

One thing we probably realized before but which is clear from the communication pattern is that we cannot do tasks in iteration 2 until at least some of the tasks from iteration 1 are completed.

So an obvious first agglomeration is to group all tasks that are responsible for a given cell over all iterations.

Our implementation, however, continues to agglomerate. We group all primitive tasks that represent a row of our simulation.

We take it a step further and agglomerate the rows to form a number of tasks equal to the number of processes we'll start.

So we have a model for what we implemented. Let's analyze it.

Let χ represent the time needed to compute $cell_{i,j,t}$ given values of $cell_{i-1,j-1,t-1}$, $cell_{i,j-1,t-1}$, $cell_{i+1,j-1,t-1}$, $cell_{i-1,j,t-1}$, $cell_{i,j,t-1}$, $cell_{i+1,j,t-1}$, $cell_{i-1,j+1,t-1}$, $cell_{i,j+1,t-1}$, and $cell_{i+1,j+1,t-1}$.

Using a single processor to compute the n^2 cells for one iteration requires $n^2\chi$ time. To compute m iterations, we need $mn^2\chi$ time.

Using p processors with the decomposition we described previously, we can perform our groups of rows concurrently. Thus, the time for one iteration is $\frac{n^2}{p}\chi$. If n does not divide evenly by p, we will have more rows assigned to some processors, so we use the maximum, denoted by $\lceil\frac{n^2}{p}\rceil\chi$, to represent our per-iteration computational cost.

However, we have introduced communication costs. Let's let λ represent the cost to send one value from one process to another. For each iteration, we need to send and receive either one or two rows to our neighbors. So at a maximum, we will send $2n$ values at a cost of $2n\lambda$.

This gives us a cost model for our parallel implementation:

$$m(\lceil\frac{n^2}{p}\rceil\chi + 2n\lambda)$$

There is an additional cost for the message passing that we have not accurately accounted for. We don't send $2n$ messages, each with a cost of λ at each step – we send 2 messages of length n.

There are two parts to the cost of sending a message:

- *latency* – the cost of sending any message
- *bandwidth* – the cost of sending a given amount of data in a message

Sending 100 messages of size 1 is much more expensive than sending 1 message of size 100, mainly because we incur the latency cost once per message, while the bandwidth cost is the same.

If our communication system has a bandwidth capability of β, typically measured in units such as MB/sec, we can express the cost of sending a message of size n as $\lambda + \frac{n}{\beta}$.

Applying this to our cost model for the Game of Life, we have the 2 messages of size n at each step to get:

$$m(\lceil\frac{n^2}{p}\rceil\chi + 2(\lambda + \frac{n}{\beta}))$$

Sieve of Eratosthenes

Chapter 5 of Quinn describes a method for finding prime numbers called the *Sieve of Eratosthenes*. This will be the first of a few problems we'll use to refine our parallelization skills.

We first consider what would be the primitive tasks. The marking of the multiples of a particular value is at the heart of the computation, and it is these markings that are the primitive tasks.

A data decomposition is most natural here, since our array of candidate primes can be distributed. We will associate a single task with each of the candidate primes.

Of course, we know from our experiences so far that we will need to group these tasks appropriately. There are two main options, each of which we've seen in the OpenMP context: interleaved decomposition, and block decomposition.

If we want to agglomerate p tasks from our array of n numbers, there are quickly some problems with the interleaved decomposition. So we use a block decomposition, assigning about $\frac{n}{p}$ entries to each.

In many of our examples, we have used block decompositions but have further assumed that we could divide n by p evenly. Section 5.4 in Quinn does not make this assumption.

So, our approach:

- distribute the n-element "sieve" array, initialize to 0's (unmarked)
- start with `k=2`
- `while (k<=sqrt(n))`
  - mark all local multiples of `k`

  - compute next k (next unmarked)

- print the resulting primes

We make a further assumption that all values of k will be found by our master process. This is valid as long as $n \geq p^2$, which is a perfectly reasonable assumption (though our program should check it). This means we always know that the master process can find the next value of k and broadcast it to the others.

Quinn has a detailed analysis of this algorithm that is worth a look.

Once we have a version that does the broadcast, can we eliminate that broadcast?

We can, by using *replicated computation*. As long as we assume that $n >> p^2$, there seems to be little harm in having a second little array replicated on each process that computes the primes between 2 and $\sqrt{n}$.

This eliminates the cost of the broadcast, but is it worth doing the replicated computation? Well, we already had to wait on all other processes for the master to compute the next prime. So there's really no time lost there – the others were just waiting in the broadcast operation anyway. The cost is in additional memory usage, which is small for large enough values of n.

Quinn suggests a further enhancement that tries to achieve better cache utilization. By reordering our loops to mark multiples of all primes within a given subrange, we can keep data in cache longer, which is often the most significant factor affecting performance of our programs.

Finally, Quinn exercise 5.9 suggests a functional decomposition approach and 5.10 asks you to name some disadvantages.

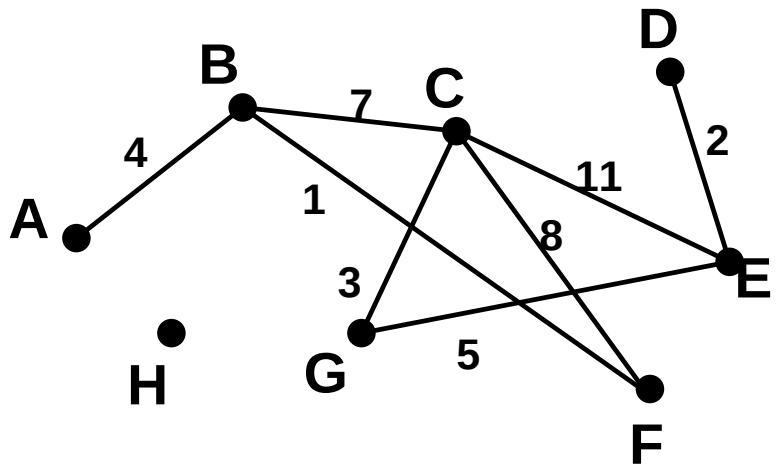
Floyd's Algorithm

Many computationally and memory intensive problems involve graph structures and computations on those graph structures. We will consider one for now.

First, recall the definition of a graph:

A *graph* G is a collection of *nodes* or *vertices*, in a set V, joined by *edges* in a set E. Vertices typically have labels. Edges can also have labels (often weights).

The graph structure represents relationships (the edges) among the objects stored (the vertices).



Graphs can be *directed* or *undirected*. In a directed graph, each edge represents a one-way connection. For undirected graphs, edges connect two vertices mutually.

Chapter 6 of Quinn examines *Floyd's Algorithm*, which is used to solve the all-pairs shortest paths problem for a directed graph.

See Figure 6.1 in Quinn for an example of a graph and its representation as an *adjacency matrix*, then a solution to the all-pairs shortest path problem.

The procedure works by numbering the vertices of the graph $v_1, v_2, ..., v_n$. We start by setting the "best known" distance between v_i and v_j as the weight of the edge between v_i and v_j, if it exists, ∞ otherwise. Then we consider paths from v_i to v_j that pass through v_1. The shortest path is now either the weight of the edge from v_i to v_j or a path from v_i to v_1 to v_j. Then, we consider paths that include v_2, then v_3, and so on. At step k, we need to check if the path from v_i to v_k followed by the path from v_k to v_j is shorter than the best (so far) known path from v_i to v_j.

This is implemented with a triple-nested for-loop, so the serial version of $O(n^3)$. Below, A is the adjacency matrix of the graph, and D is the matrix of shortest paths:

```
// intialize matrix a with edge weights where they exist,
// with MAXINT where they do not
for (k=0; k<n; k++)
  for (i=0; i<n; i++)
    for (j=0; j<n; j++)
      a[i][j] = min(a[i][j], a[i][k]+a[k][j]);
// entries of a now contain length of shortest paths
```

What about parallelization?

Our primitive tasks will be the n^2 entries of the matrix, and each primitive task does the k iterations for its entry.

Let's start by thinking about how an algorithm would work if we had n^2 processors, one for each of these primitive tasks.

At first, let's not worry about data distribution. Suppose we have a shared memory environment. Can we do the primitive tasks in any order?

It might not seem so at first, since at iteration k, we need to make sure we don't change the values we're using during this iteration before we make use of them.

The reason for this is that we are using only values in row k and column k at during iteration k. Values in row k will not change during iteration k since the assignment for entries i row k is:

```
a[k][j] = min(a[k][j], a[k][k]+a[k][j])
```

But all of our values are positive, so there is no way the above statement can change entry `a[k][j]`.

A similar argument shows that values in column k will not change during iteration k.

Thus, we can perform the assignments at a given iteration in any order and/or concurrently.

With shared memory and threads, this is simple - just parallelize our inner `for` loops and we're set.

With message passing and distributed memory, matters become more complex.

Each `a[i][j]` will be owned by a process. What other information does that process need to perform the update of its own data element at iteration k?

It needs the value in row k from its column and from column k in its row.

This boils down to this procedure:

```
for (k=0; k<n; k++) {
   each process P[k][j] broadcasts its value to the P[*][j] processes
   each process P[i][k] broadcasts its value to the P[i][*] processes
   each process waits to receive the needed values
   each process P[i][j] computes its new a[i][j]
}
```

So every process participates in 2 broadcasts of a single value to n processes at each iteration, plus its small computation.

Even assuming all communication can occur concurrently, this is a lot of communication.

Our cost using the cost model from Quinn:

Let χ be the cost of an entry update. Our computation time in each iteration is χ, since all n^2 operations are happening concurrently. The communication cost is $2 \log n \lambda$, since we have to wait for 2 broadcasts among n processes of a single value.

So over n iterations, our cost is:

$$n(\chi + 2\log n\lambda)$$

Most likely, the cost of messaging in this case will far exceed the cost of computation. Clearly, we will need to agglomerate primitive tasks to increase the work done by each and to reduce the needed communication.

Obvious choices are to agglomerate rows or to agglomerate columns. Either can be done, but we will choose rows, since that is more natural given the row-major order of array storage in C.

If we group by rows, our row broadcast goes away. Each process will only need to broadcast its entry from column k at iteration k to each other process. Quinn does this, and groups rows onto an arbitrary number of processors p, where $p <= n$.

We have removed half of the communication and made the computation within each row more significant. This seems good.

How does this affect our cost model. Let's assume that we do as Quinn does, and have some number of processors $p < n$ and we group rows on processors.

Each iteration involves n^2 updates, each of which costs χ. With p processors, the computation cost is the cost of computing on the most heavily loaded processor, which will be assigned $\lceil n/p \rceil$ rows, each of which costs $n\chi$. The communication cost is now a single broadcast of $\lceil n/p \rceil$ values among p processors, which will cost $\lceil \log p \rceil (\lambda + \frac{\lceil n/p \rceil}{\beta})$. So our full expression over n iterations:

$$n(\lceil n/p \rceil n\chi + \lceil \log p \rceil (\lambda + \frac{\lceil n/p \rceil}{\beta}))$$

Even better, let's think about it in terms of p processors. Furthermore, we'll assume that the processors are arranged logically as a $\sqrt{p} \times \sqrt{p}$ matrix, and the matrix a is broken into p blocks of size $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$. Each processor updates its own submatrix at each iteration.

What information does each processor need at the k^{th} iteration? It needs information from the processors that contain the k^{th} row and k^{th} column. The processor that contains the k^{th} row or the k^{th} column will need to broadcast that row/column to every other processor that has part of that row/column.

Pseudocode for this procedure:

```
for (k=0; k<n; k++) {
   each process P[i][j] that has a segment of the kth row of a
      broadcasts it to the P[*][j] processes
   each process P[i][j] that has a segment of the kth column of a
      broadcasts it to the P[i][*] processes
   each process waits to receive the needed segments
   each process P[i][j] computes its local part of a
}
```

In each iteration, the k^{th} row and k^{th} column processors do a one-to-all broadcast along the row or column of $\sqrt{p}$ processors. Each such processor broadcasts $\frac{n}{\sqrt{p}}$ elements, for a cost of $O(\frac{n}{\sqrt{p}} \log p)$ for communication.

The computation at each iteration is $O(\frac{n^2}{p})$

Our cost model becomes:

$$n(\frac{n^2}{p}\chi + \lceil \log p \rceil (\lambda + \frac{n/\sqrt{p}}{\beta}))$$