ECE669: Parallel Computer Architecture
Fall 2004 **Handout #2**

Homework # 2 Due: October 6

Programming Multiprocessors: Parallelism, Communication, and Synchronization

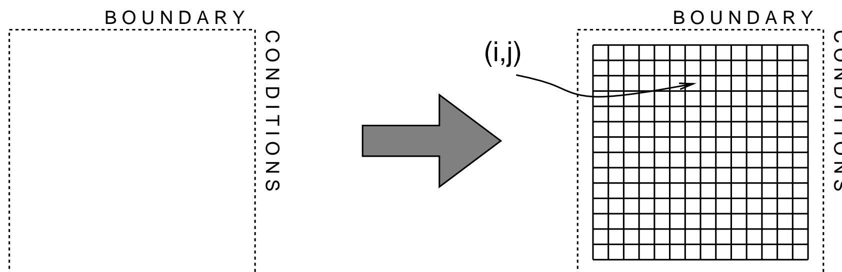
1 Introduction

When developing multiprocessor applications, we attempt to exploit parallelism to achieve increased performance. With increased parallelism, however, comes increased interprocessor communication. Since real multiprocessors can provide only a finite amount of network bandwidth, this tension between parallelism and communication has significant ramifications which we need to keep in mind when designing and programming multiprocessors. In this exercise, you will examine several of these issues.

The chief aim of the exercises contained here is to familiarize you with the problems of partitioning parallel programs and data structures and the judicious use of synchronization to allow correct operation while minimizing serialization.

2 Jacobi Relaxation

Jacobi relaxation is an iterative algorithm which, given a set of boundary conditions, finds (discretized) solutions to differential equations of the form $\nabla^2\mathcal{A} + \mathcal{B} = 0$. As we've seen in lecture, we begin by choosing the grid which will form the basis of our discretization:

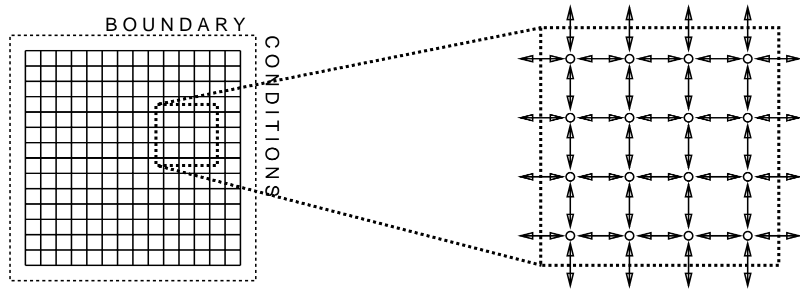


To find a solution on a grid, we repeatedly apply the following iterative step until we converge on a solution.

$$A^{k+1}_{\mathrm{i,j}} = \frac{A^k_{\mathrm{i+1,j}} + A^k_{\mathrm{i-1,j}} + A^k_{\mathrm{i,j+1}} + A^k_{\mathrm{i,j-1}}}{4} + b_{\mathrm{i,j}} \qquad (1)$$

Jacobi differs from other iterative relaxation algorithms in that the update of each point (at iteration step $k+1$) requires the *previous* values of the neighboring points (from iteration step k).

We use a simple graphical representation to capture those features we are interested in and abstract away excess detail. In this representation, graph nodes represent fixed amounts of computation; graph edges represent communication between computation nodes. In such a representation, a single iteration of Jacobi relaxation looks as follows:



In this graph, each node represents the computation required to compute a new value for a single grid point. To compute a new value for a grid point, Jacobi relaxation dictates that we average the previous values of each of the neighboring grid points – thus each node is connected to its four neighboring nodes with an edge that represents the communication of two grid values (one in each direction).

The following figure shows one step of a relaxation on a four by four grid. The A^k matrix contains the values of the grid at step k, and the A^{k+1} matrix has the values for the next step. The boundary conditions are shown on the edges of the A^k and A^{k+1} matrices. The B matrix, which is constant, contains the values that are added to the grid points on each step according to Equation 1.

A^k

	0	0	0	0	
4	6	4	5	0	0
4	8	5	8	3	0
4	1	8	7	8	0
4	4	3	4	2	0
	0	0	0	0	

B

4	0	1	2
0	–3	–1	0
2	0	–3	1
2	0	–3	5

A^{k+1}

	0	0	0	0	
4	8	4	4	4	0
4	4	4	4	4	0
4	8	4	4	4	0
4	4	4	0	8	0
	0	0	0	0	

You might want to think about the values in the A^{k+2} matrix after the next relaxation step.

There are a number of ways to partition the data in this grid to a bunch of processors. Figure 1 shows two possible ways to partition the four by four grid to four processors. One partition allocates one row of the matrix to each processor, and the other allocates a two by two submesh of the matrix to each processor. For every step in the relaxation, *each processor will calculate the new values for its own matrix cells.* To accomplish this calculation, the processors need to get cell values from neighboring processors. The values on the arrows between the nodes represent the data that must be communicated between processors.

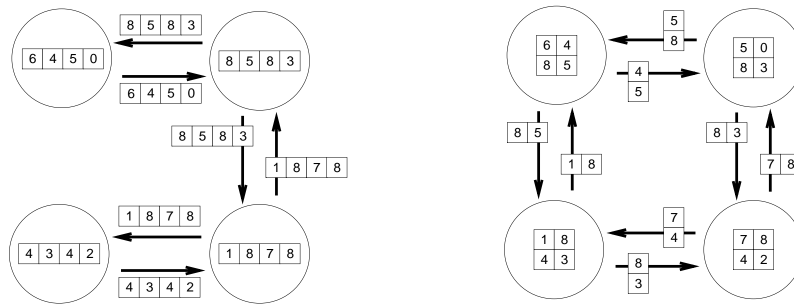
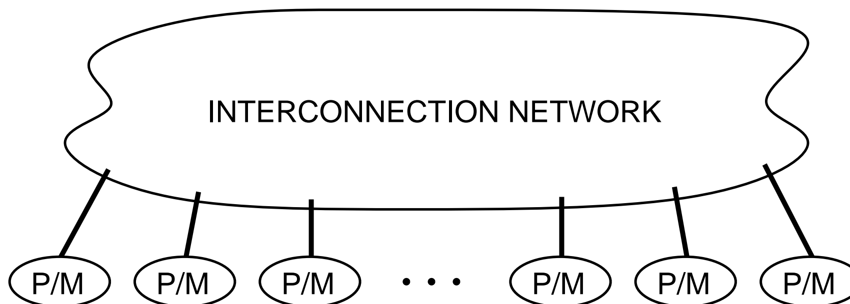


Figure 1: Partitioning by rows and by square tiles.

3 Machine Model

The machine model which we'll be using in this lab is relatively simple – a distributed-memory multiprocessor in which some number of processor/memory nodes communicate via an interconnection network. Each processor contains some amount of memory in which it stores the data partition for which it is responsible.



In this machine model, we assume the interconnection network provides uniform access – all interprocessor communication is equally expensive in terms of network resources consumed and communication latency!

Note that we don't make any assumptions about what programming model (e.g. shared memory, message passing, etc.) this machine provides to the end user (yet).

Given an application's graphical representation (such as that described above for Jacobi relaxation), we "program" a P-processor machine by deciding which processor should perform the computation represented by each of the nodes in the graph. Since we could equivalently view this process as one of dividing the graph into (at most) P pieces, we usually refer to this as the *partitioning problem*.

Ex 1: For each of the two partitions in Figure 1 how should the B matrix be distributed to the processors? How should the boundary values be distributed?

Ex 2: Using total amount of communicated data as a metric, which of the two partitions in Figure 1 is better?

4 A Simple Performance Metric

The total running time of the program is the ideal metric of goodness – one partition of a program graph is better than another if it results in a shorter running time. But how do we determine running time for a particular partition running on a P-processor machine? If we assume that there is no overlap between computation and communication, we can estimate the running time as the sum of the computation time and the communication time.

$$T = (\text{time to compute}) + (\text{time to communicate}) \tag{2}$$

We start by determining the following information from the program graph and partition:

- w_i – the total amount of computation for processor i (in abstract "computation units")
- c_i – the total amount of communication invoked by processor i which cannot be resolved on that processor (in abstract "communication units")

For simplicity, assume that we always partition things such that all processors get the same amount of work, w, and invoke the same amount of external communication c. ($w = w_1 = \cdots = w_i$ and $c = c_1 = \cdots = c_i$)

Given w and c, we might compute running time T by summing the computation and communication times required by one processor, assuming that there is no overlap between communication and computation. Since all the processors are running in parallel and doing the same amount of work, the running time of a single processor should be the same as the running time of the entire application.

Thus,

$$T = sw + lc \tag{3}$$

where

- s is a measure of processor speed – a processor requires s time units to complete one unit of computation
- l is a measure of network latency – the network requires l time units to transport one unit of communication

Here is a table that summarizes the variables used in the above formulae:

T	running time, the metric of a partition
w	amount of computation (work) for a processor
s	processor speed, in terms of time units to complete on unit of computation
c	amount of communication for a processor
l	network latency, in terms of time units to transport one unit of communication
P	number of processors

Caveat: If we use Equation 3 to guide our partitioning decisions for Jacobi, we'll find that running a finer-grained partition on a larger number of processors is *always* preferable. Empirically, we know this isn't true – sometimes the increased communication requirements of a finer-grained partition impose enough of a load on the interconnection network that the total running time is actually longer than it might be with a coarser partition. This effect stems from the fact that communication bandwidth is a finite resource in real interconnection networks. Along with increased communication comes increased contention for this resource, and, as with any finite resource, increased contention results in increased service latency. However, we shall defer the issues of contention for network resources to a later date.

5 More Exercises:

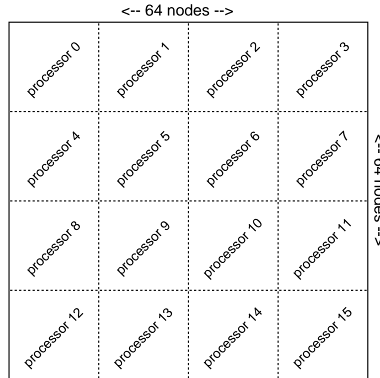
The following exercises examine further details of the scenario given in the previous section. Unless stated otherwise, assume $s = 10$, and $l = 1$. Assume that the Jacobi problem grid is of size $n \times n = 64 \times 64$.

Ex 3: 64 Processors, Strips

If you use $P = 64$ processors and partition the Jacobi graph into 64 strips, each 64 nodes wide and one node high, what are the appropriate values of w and c, and T? How much of a speedup is this over the sequential running time? Recall that a processor is responsible for updating the values in its partition.

Ex 4: 16 Processors, Tiles

Instead of partitioning the Jacobi graph into long, narrow strips, what if we partitioned it into square tiles? With 16 processors, each processor would get a 16 by 16 node tile, as shown below:



What are the appropriate values of w and c? Using these values, what values do you get for T? How much of a speedup is this over the sequential running time?

Ex 5: A General Expression for Jacobi

Assuming a $n \times n$ Jacobi grid, derive an expression for the amount of communication for one partition, when the aspect ratio of each partition is given as $a : b$. The aspect ratio is specified as $a : b$, where a is the size of the x dimension of the partition, and b is the size of the y dimension of the partition. Furthermore, assume there are P processors and that each processor gets an equal amount of work.

Ex 6: Optimal Partitioning

Prove that the volume of communication per partition is minimized when $a = b$. Assume as before that there are P processors, and that each processor must get an equal amount of work.

The next three problems focus on various methods of specifying the *speedup* of a computation. In general, the speedup $S(P)$ is the ratio of the sequential running time and the parallel running time, $T(1)/T(P)$. Depending on how other parameters (e.g., problem size) are constrained in the above computation of speedup, we get several notions of speedup:

Ex 7: Ordinary Speedup

What is the speedup for the optimal partitioning, when there are P processors, and when the problem size is fixed at $N = n \times n$. This computation yields the most common form of speedup.

Ex 8: Scaled Speedup

What is the speedup for the optimal partitioning, when there are P processors, and when

the problem size grows in proportion to the number of processors.[1] That is, problem size $N \propto P$, where $N = n^2$.

In other words, compute the scaled speedup as $T(N(P),\ 1)/T(N(P),\ P)$, where $T(N(P),\ P)$ denotes the running time for a problem of size $N(P)$ on P processors, and where $N(P)\ \propto\ P$. Assume that N with one processor is 4096.

For example, if we want to compute the scaled speedup with $P = 4$ processors, we would divide the sequential running time for the problem size $N = 4096$ with the parallel running time on four processors for a problem of size $N = 4096 \times 4$,

Ex 9: Asymptotic Speedup

What is the asymptotic speedup[2] for the optimal partitioning for a fixed problem of size $N = n \times n$. Asymptotic speedup is computed as the maximum speedup achievable with any number of processors, keeping problem size fixed. The asymptotic speedup is given as a function of N, the problem size.

(Optional) Ex 10: Speedup

For each of the following situations, which of the above notions of speedup (ordinary, scaled, asymptotic) is most appropriate?

a) You have a problem that you wish to solve faster by running it in parallel.
b) Your problem is larger but you expect to have a larger parallel machine.
c) You have as big a machine as you wish, and want to solve the problem as fast as possible.

Where might you encounter these situations is the real world?

Ex 11: Optimal Rectangular Partitioning

Consider the following computation performed for each (i,j) on an $n \times n$ grid.

$$A^{k+1}_{\mathrm{i,j}} = \frac{A^k_{\mathrm{i+x,j}} + A^k_{\mathrm{i-x,j}} + A^k_{\mathrm{i,j+y}} + A^k_{\mathrm{i,j-y}}}{4}$$

Assume $n >> P$, $n >> x$ and $n >> y$. Derive the aspect ratio that minimizes communication for the communication pattern inherent in the computation shown above.

(Optional) Ex 12: Optimal Rectangular Partitioning

Does your answer to the above question change if the some additional terms are included

[1] Reevaluating Amdahl's Law. John L. Gustafson. CACM, May 1988.
[2] Scalability of Parallel Machines. Dan Nussbaum and Anant Agarwal. CACM, March 1991.

in the iteration step as shown below,

$$A^{k+1}_{\mathrm{i,j}} = \frac{A^k_{\mathrm{i+x,j}} + A^k_{\mathrm{i+x',j}} + A^k_{\mathrm{i-x,j}} + A^k_{\mathrm{i-x',j}} + A^k_{\mathrm{i,j+y}} + A^k_{\mathrm{i,j-y}}}{6}$$

where $x > x'$. Discuss briefly.

Ex 13: Matrix Vector Product

Matrix-vector products of the form $r = As$, where A is a $N \times N$ matrix and r and s are column vectors with N elements each, commonly occur in many numerical computations. Assume $N > P$, where P, the number of processing nodes, divides N perfectly.

Suppose s is partitioned into P contiguous chunks, each of size N/P, and each assigned to a processor. (Element i of s is assigned to processor $\lfloor \frac{i}{(N/P)} \rfloor$.) The result vector r is similarly partitioned and distributed. Assume each processor is responsible for performing all the necessary operations to determine the portion of r assigned to it.

(a) How would you distribute the matrix A to the processors to minimize communication. Assume A is a dense matrix?

(Optional) (b) How would you distribute the matrix A to the processors to minimize communication, if A is a tridiagonal matrix? If A is known a priori, can you completely avoid distributing A?