Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.846: Parallel Computing
Spring 2008, Agarwal Handout #9

Homework # 1 **Due: Feb 28**

Multicore Programming: An Introduction

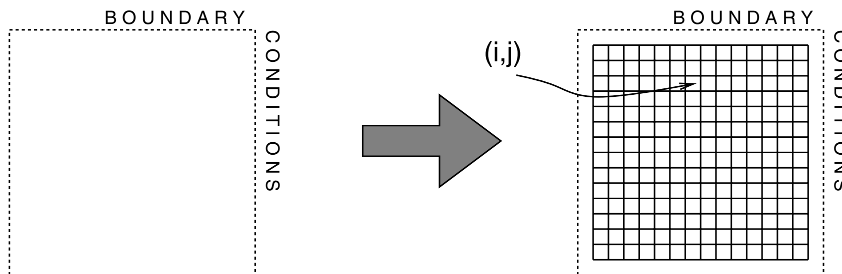
1 Introduction

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 homework, 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, the TILE64 development environment, and the iLib API for parallel programming.

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:

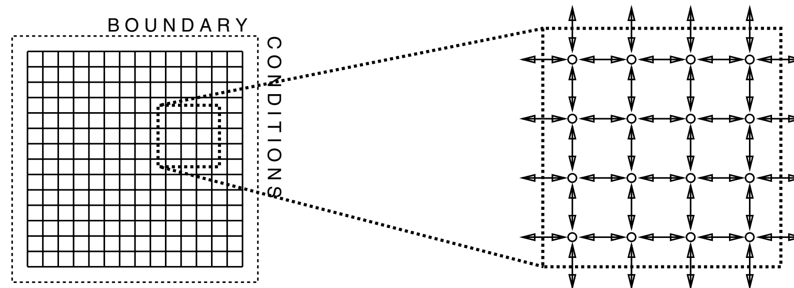


To find a solution on a grid, we repeatedly apply the following iterative step until we converge on a solution.

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

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 sub-grid 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.

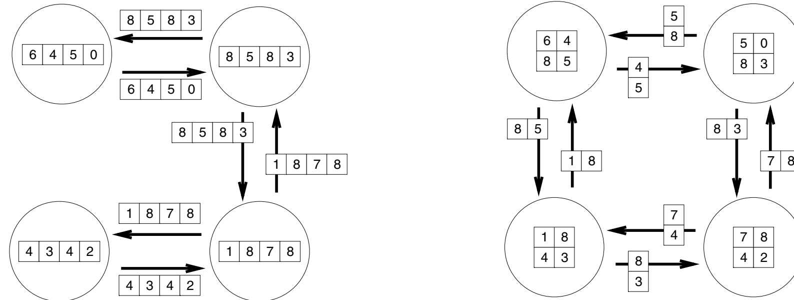
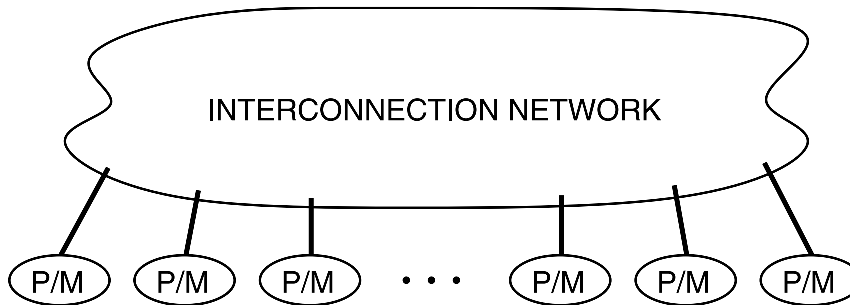


Figure 1: Partitioning by rows and by square tiles.

3 Machine Model

The machine model which we'll be using in the written exercises for this homework 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 with an array representation (such as that described above for Jacobi relaxation), we "program" a P-processor machine by deciding which processor should perform the computation required to produce each of the elements in the array. Since we could equivalently view this process as one of dividing the array 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 describe how the 4 by 4 arrays are distributed across 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 $l = 1$, but show results both symbolically (using s, l) and numerically (substituting the above values). Assume that the Jacobi problem grid is of size $n \times n = 64 \times 64$. Count one unit of 'work' for computing one matrix element, and assume all communications are serialized (*i.e.*, you can't transfer data in four directions at once).

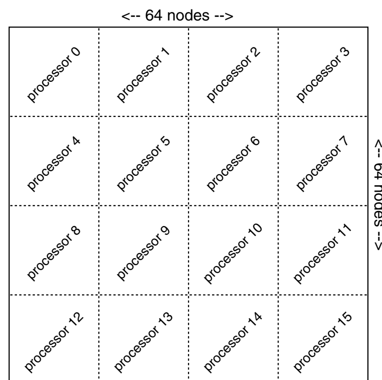
Ex 3: 64 Processors, Strips

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. Also, don't worry too much about boundary conditions, since it is the central nodes (with more work to do) that determine the maximum speedup of the problem.

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

You have an $n \times n$ Jacobi grid, and P processors each of which gets an equal amount of work. Derive an expression for the amount of communication for one partition, when the aspect ratio of each partition is given as r. This ratio r is equal to a/b, where a is the size of the partition in the x dimension and the b is the size in the y dimension; for example, a ratio of 1/2 would mean that the partitions are twice as tall (in the y axis) as they are wide (in the x axis). (Note, however, that you are not given the actual values of a and b; you must derive them to compute the communication traffic.)

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 a problem of size $N = 4096 \times 4$, with the parallel running time for four processors for the same problem 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.

Ex 10: Speedup

Can you identify in one or two sentences the circumstances under which each of the above notions of speedup is most appropriate.

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 $a : b$ that minimizes communication for the communication pattern inherent in the computation shown above. Assume the data is distributed in the same way as for standard Jacobi, and that i, x, and a all refer to the same axis of the array.

(Optional) Ex 12: Optimal Rectangular Partitioning

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

Does your answer to the above question change if the some additional terms are included 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, e.g., conjugate gradient. 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 (*i.e.*, at compile time), can you completely avoid distributing A?

Ex 14: Parallel Programming Exercises

The following exercise requires use of iLib and the TILE64 programming environment. Make sure you have read the handout "Laboratory Information I" and the iLib API reference manual before starting these exercises.

Parallelize the following sequential C code for a one-dimensional Jacobi relaxation with $n = 64$. Parallelize this code across 16 processors using iLib. Use `ilib_proc_go_parallel()` to create multiple processes. Use iLib messages to communicate intermediate results. For this exercise, do not use iLib channels or shared memory. Attach your iLib code and the program output to your homework. Please modify your program so that the output array is only printed once.

In addition to the result, the program will print out the number of cycles it took to execute. What is the speedup of your sixteen tile implementation?

Change the `SIZE` value to 2048. What, if any, changes do you have to make to the code? Why did you have to make those changes? What is the speedup now?

For this problem size (which is failry small) messaging is a bad choice of communication mechanism. The high overhead of the message is not amortized by the amount of work done per message. In future assignements we will explore both lower overhead communication mechanisms and larger problem sizes.

Note that the TILE64 processor does not have native support for floating point, so we implement this program with integer arithmetic. For large values of ITERS this will be inaccurate, but for the values used here, it is not an issue.

```
#include <stdio.h>
#include <sys/archlib.h>

#define SIZE 64
#define ARRAYSIZE (SIZE+2)
#define ITERS 12

void print_array(int *array)
{
  int i;
  for (i = 1; i <= SIZE; ++i)
    printf("%d ", array[i]);
  printf("\n");
}

void update_array(int *oldarr, int *newarr)
{
  int from, to, i;

  from = 1;
  to = SIZE;

  for (i = from; i <= to; ++i)
    newarr[i] = (oldarr[i-1] + oldarr[i+1])/2;
}

#define SWAP(a,b,t) (((t) = (a)), ((a) = (b)), ((b) = (t)))

int main(void)
{
  int i, t;
  int *oldarr, *newarr, *tmp;
  int start, stop;

  oldarr = (int *) malloc(ARRAYSIZE * sizeof(int));
  newarr = (int *) malloc(ARRAYSIZE * sizeof(int));
  oldarr[0] = newarr[0] = 32768.0;
  for (i = 1; i < ARRAYSIZE; ++i)
    oldarr[i] = newarr[i] = 0.0;

  start = get_cycle_count();
  for (t = 0; t < ITERS; ++t) {
```

```
    update_array(oldarr, newarr);
    SWAP(oldarr, newarr, tmp);
  }
  stop = get_cycle_count();

  print_array(oldarr);
  printf("Program took %d cycles\n", stop-start);
  return 0;
}
```