

# GEORGIA INSTITUTE OF TECHNOLOGY

College of Computing

## CS/EE6760 — Parallel Computer Architecture I

Fall 1998

---

CS/EE6760  
Homework 3

Issued: November 10, 1998  
Due: November 19, 1998

---

**Purpose:** This homework introduces programming models and some simple performance models for parallel machines. The goal is to develop intuition and a back-of-the-envelope strategy for evaluating the suitability of an architecture to an application.

**Reading:** H&P Sections 8.1 and 8.2  
[Dally91] (*optional, but a useful high-level view*)  
[Bertsekas89] and [Lubeck88] (*application*)  
[vonEicken92] (*“Active Messages” communication model*)

**Problems:**

1. Partitioning.
2. Communication Overhead.
3. Parallel Programming.

## Introduction

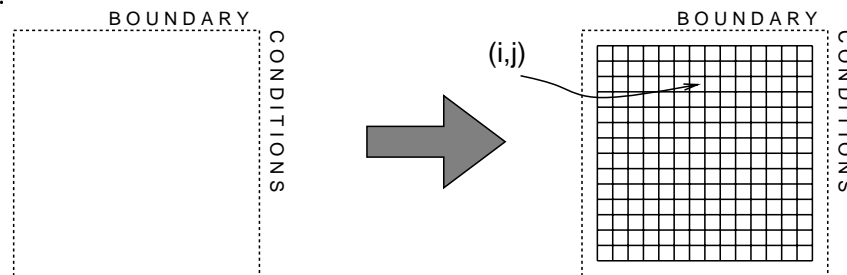
When developing multiprocessor applications, we attempt to exploit parallelism to achieve increased performance. With increased parallelism, however, comes increased interprocessor communication. Since real multiprocessors can provide only a finite amount of network bandwidth, this tension between parallelism 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.

Multiprocessor programs also require the introduction of synchronization to coordinate multiple processes. While different languages offer varied forms of synchronization, in this homework you will explore producer-consumer style synchronization, implemented using `barrier()` constructs for shared-memory and via messages in message-passing.

The goal of these exercises is to familiarize you with the problems of partitioning parallel programs and data structures. In addition, you will develop a simple model of parallel program performance that gives insights into the demands of applications and the capabilities of machines.

## Example Application: Jacobi Relaxation

*Jacobi relaxation* is an iterative algorithm which, given a set of boundary conditions, finds (discretized) solutions to differential equations of the form  $\nabla^2 \mathcal{A} + \mathcal{B} = 0$  (see [Bertsekas89]). As we've seen in lecture, we begin by choosing the grid which will form the basis of our discretization:



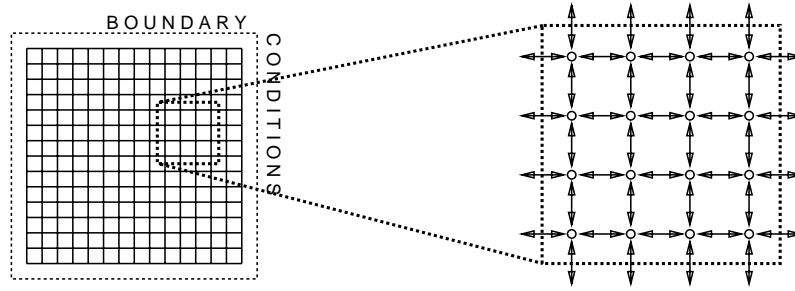
To find a solution on a grid, we repeatedly apply the following iterative step until we converge on a solution.

$$A_{i,j}^{k+1} = \frac{A_{i+1,j}^k + A_{i-1,j}^k + A_{i,j+1}^k + A_{i,j-1}^k}{4} + b_{i,j}$$

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).

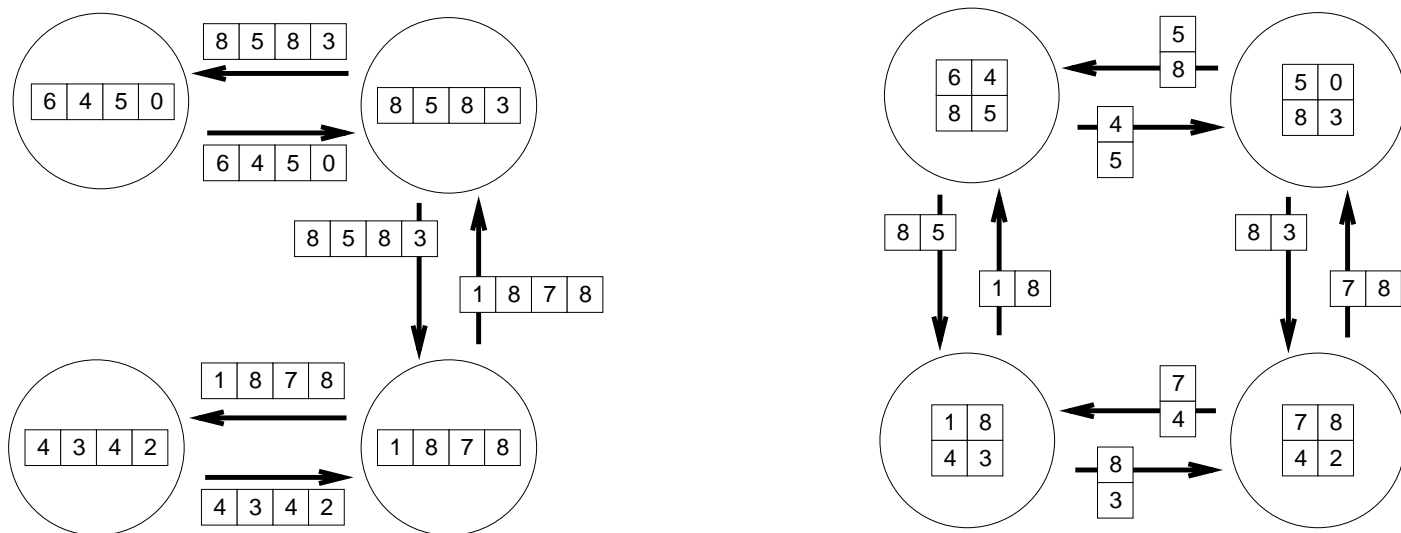
The following figure shows one step of a relaxation on a four by four grid. The  $A^k$  matrix contains the values of the grid at step  $k$ , and the  $A^{k+1}$  matrix has the values for the next step. The boundary conditions are shown on the edges of the  $A^k$  and  $A^{k+1}$  matrices. The  $B$ 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

$$\begin{array}{c}
 \begin{array}{cccc}
 & 0 & 0 & 0 & 0 \\
 4 & \begin{array}{|c|c|c|c|} \hline 6 & 4 & 5 & 0 \\ \hline \end{array} & 0 \\
 4 & \begin{array}{|c|c|c|c|} \hline 8 & 5 & 8 & 3 \\ \hline \end{array} & 0 \\
 4 & \begin{array}{|c|c|c|c|} \hline 1 & 8 & 7 & 8 \\ \hline \end{array} & 0 \\
 4 & \begin{array}{|c|c|c|c|} \hline 4 & 3 & 4 & 2 \\ \hline \end{array} & 0 \\
 & 0 & 0 & 0 & 0
 \end{array}
 \end{array}
 \quad \longrightarrow \quad
 \begin{array}{c}
 \begin{array}{cccc}
 & 0 & 0 & 0 & 0 \\
 4 & \begin{array}{|c|c|c|c|} \hline 8 & 4 & 4 & 4 \\ \hline \end{array} & 0 \\
 4 & \begin{array}{|c|c|c|c|} \hline 4 & 4 & 4 & 4 \\ \hline \end{array} & 0 \\
 4 & \begin{array}{|c|c|c|c|} \hline 8 & 4 & 4 & 4 \\ \hline \end{array} & 0 \\
 4 & \begin{array}{|c|c|c|c|} \hline 4 & 4 & 0 & 8 \\ \hline \end{array} & 0 \\
 & 0 & 0 & 0 & 0
 \end{array}
 \end{array}$$
  

$$\mathbf{B} = \begin{array}{|c|c|c|c|} \hline 4 & 0 & 1 & 2 \\ \hline 0 & -3 & -1 & 0 \\ \hline 2 & 0 & -3 & 1 \\ \hline 2 & 0 & -3 & 5 \\ \hline \end{array}$$

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

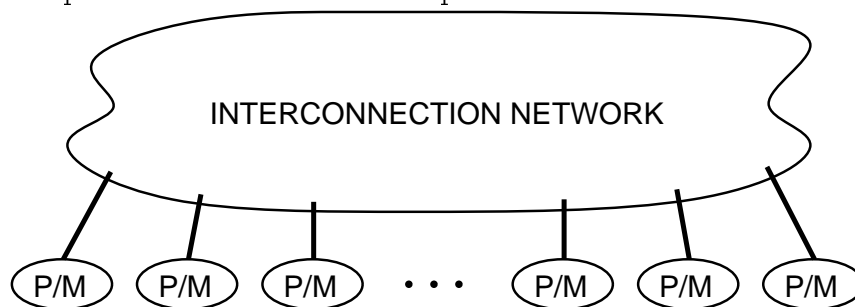


**Figure 1:** Partitioning by rows and by square tiles.

on the arrows between the nodes represent the data that must be communicated between processors.

### Machine Model

The machine model which we'll be using in 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'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*.

**A:** (*Warmup – nothing to turn in*) 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?

**B:** (*Warmup – nothing to turn in*) Using total amount of communicated data as a metric, which of the two partitions in Figure 1 is better?

### Problem 1: Partitioning

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.

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

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  $c$ . ( $w = w_1 = \dots = w_i$  and  $c = c_1 = \dots = c_i$ )

Given  $w$  and  $c$ , we initially compute the running time  $T$  by summing the computation and communication times required by one processor. 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,

$$T = s \cdot w + l \cdot c$$

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:

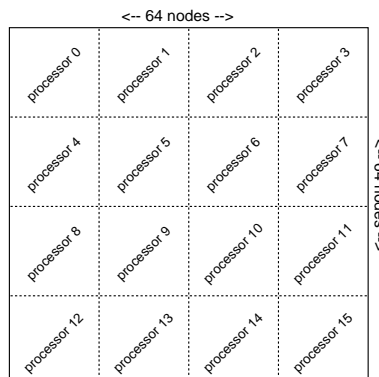
$T$	running time, the metric of a partition
$w$	amount of computation (work) for a processor
$s$	processor speed, in terms of time units to complete on unit of computation
$c$	amount of communication for a processor
$l$	network latency, in terms of time units to transport one unit of communication
$P$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

**Caveat:** The simple equation  $T = s \cdot w + l \cdot c$  models only processor and network *bandwidth*. In particular, if we use this equation 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 – eventually the costs of communication will overwhelm the speedup so that the total running time is actually *longer* than it might be with a coarser partition. We will ignore the effects of overhead in this problem but come back to them in the next problem.

The following exercises consider the Jacobi algorithm using the model above. Unless stated otherwise, assume  $s = 10$ , and  $l = 1$ . Assume that the Jacobi problem grid is of size  $n \times n = 64 \times 64$ .

**A:** 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.

**B:** 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?

**C:** Assuming an  $n \times n$  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.

**D:** 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.

**E: Optimal Rectangular Partitioning.** Consider the following computation performed for each  $(i,j)$  on an  $n \times n$  grid.

$$A_{i,j}^{k+1} = \frac{A_{i+x,j}^k + A_{i-x,j}^k + A_{i,j+y}^k + A_{i,j-y}^k}{4}$$

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

**F: (Optional)** Does your answer to the previous question change if the some additional terms are included in the iteration step as shown below, ■

$$A_{i,j}^{k+1} = \frac{A_{i+x,j}^k + A_{i+x',j}^k + A_{i-x,j}^k + A_{i-x',j}^k + A_{i,j+y}^k + A_{i,j-y}^k}{6}$$

where  $x > x'$ ? Discuss briefly.

## Problem 2: Communication Overhead

In this problem, we'll look at the costs of communication and computation more closely and (slightly) more realistically.

As discussed in class, the performance of an interconnect can be summarized in three parameters: the time-of-flight **latency** across the wire, the maximum **bandwidth** of the wire and the interface **overhead** (hardware and software) at the endpoints. Here are parameters that resemble some real networks:

Network	Wire Latency (S)	Bandwidth (bytes/S)	Overhead (S)
<b>T3E:</b>	$0.1 \cdot 10^{-6}$	$150 \cdot 10^6$	$0.5 \cdot 10^{-6}$
<b>Myrinet:</b>	$1 \cdot 10^{-6}$	$150 \cdot 10^6$	$2 \cdot 10^{-6}$
<b>ATM (uNet):</b>	$22 \cdot 10^{-6}$	$80 \cdot 10^6$	$5 \cdot 10^{-6}$
<b>ATM (TCP):</b>	$22 \cdot 10^{-6}$	$80 \cdot 10^6$	$50 \cdot 10^{-6}$

**A:** Endpoint overhead limits the effective bandwidth of an interconnect for “short” messages. How long (in bytes) must a message be to achieve half of the maximum bandwidth (the “3dB point”) in these networks? The overhead is for one endpoint (assume sending and receiving costs the same). Give two answers for each network: first, the bandwidth assuming two processors communicate completely synchronously (no messages may overlap) and, second, the bandwidth assuming messages may be overlapped/pipelined.

Simple model of applications: compute time and communication volume. You can give these parameters as a function of the number of processors and the size of the data set to get a gross feel for how an application will perform on an architecture.

At this point, we can nail down some constants, include overhead and get a much better estimate of performance. Assume the processor in all the machines above is capable of 1 floating-point operation per cycle at a clock rate of 300MHz. Ignore all other instructions (i.e. assume they are covered by instruction-level parallelism). Jacobi requires 4 FADDs plus one FMUL/cell, so, for instance, a  $64 \times 64$  grid has a sequential execution time of

$$5 \cdot 64 \cdot 64 = 20480 \text{ cycles per iteration}$$

Pick your favorite set of network parameters above and use that one set of parameters to answer the rest of the questions in this problem.

**B:** Ignoring overhead (i.e. considering only bandwidth), what is the execution time for one iteration of Jacobi on 16 processors using square tiles?

**C:** Now, considering overhead (but still ignoring latency), what is the execution time for one iteration of Jacobi on 16 processors using square tiles? Note that each processor will have to send four messages and receive four messages.

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 parallel running time and the sequential running time,  $T(P)/T(1)$ . Depending on how other parameters (e.g., problem size) are constrained in the above computation of speedup, we get several notions of speedup:

**D: Ordinary Speedup.** What is the speedup for the optimal (square tile) partitioning, when there are  $P$  processors, and when the problem size is fixed at  $N = n \times n$ ? Use your expression for the amount of communication and include the effect of message overhead. This computation yields the most common form of speedup.

**E: 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?<sup>1</sup> That is, problem size  $N \propto P$ , where  $N = n^2$ .

In other words, compute the scaled speedup as

$$T(N(P), P)/T(N(P), 1)$$

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

**F: Asymptotic Speedup.** Asymptotic speedup is defined as the maximum speedup achievable with any number of processors, keeping problem size fixed.<sup>2</sup> The asymptotic speedup is given as a function of  $N$ , the problem size. What is the asymptotic speedup for the optimal partitioning for a fixed problem of size  $N = n \times n$ .

**G: (Optional)** Can you identify the circumstances under which each of the above notions of speedup is most appropriate?

---

<sup>1</sup> Reevaluating Amdahl's Law. John L. Gustafson. CACM, May 1988.

<sup>2</sup> Scalability of Parallel Machines. Dan Nussbaum and Anant Agarwal. CACM, March 1991.

### Problem 3: Parallel Programming

This problem will explore writing a trivial parallel program using a shared memory programming model.

In a shared-memory model, all processors read and write variables in the same global address space. Communication takes place, but is implicit. As a further simplification, we'll assume a SPMD – Single-Program-Multiple-Data – model that places one logical thread on each processor in the machine. The threads do not operate in lockstep, but the code for every thread is identical.

Threads have access to a couple of variables so that they can tell which processor they are on:

```
NPROCS    --- variable indicating the number of processors
            (assume that N in Jacobi is a multiple of NPROCS).
MYPID     --- processor number (from 0 to NPROCS - 1) of this
            processor.
```

Shared memory takes care of communication implicitly, but synchronization must be explicit. The usual locks/mutex/condition variables used for multi-threaded programming on a single processor are possible synchronization primitives. A common primitive in SPMD programs, though, is `barrier()`. The semantics of a barrier are that all threads (processors) must reach the barrier before any thread can proceed past the barrier.

For example, the code fragment below is for a 16-processor machine in which all 16 processors cooperate to take the dot-product of a 16-entry array.

```
shared double array[16];      /* initialized elsewhere */
shared double result;        /* written by dotprod, below */

void dotprod()                /* called in parallel on all processors */
{
    int i;

    /*
     * 1. All processors square one entry of the array in parallel,
     *    then we wait (at the barrier) to make sure all are done.
     *    Since each processor does the same amount of work, the
     *    wait should be essentially zero.
     */
    array[MYPID] = array[MYPID] * array[MYPID];
    barrier();

    /*
     * 2. processor zero alone computes the result. All the processors
     *    wait for the result to become valid (at the barrier) before
     *    proceeding. Since only processor 0 is working, the others
     *    will wait and do nothing.
     */
    if (MYPID == 0)
    {
        result = 0.0;
        for (i = 0; i < 16; i++)
            result = result + array[i];
    }
    barrier();
}
```

The next page shows some C code for a sequential implementation of a one-dimensional Jacobi relaxation. There are two pieces of cleverness to explain:

- The arrays are two elements longer than necessary to provide space for boundary conditions. Boundary conditions are kept as constants in `a[0]` and `a[N+1]` — the code alters only `a[1]` through `a[N]`.
- Since Jacobi requires that each iteration requires the values from the *previous* iteration, we compute the values back and forth between two arrays rather than changing them in-place in one array.

```

#define N 64
#define ITERS 10

double b[N+2]           /* array of constants */
double old_a[N+2];     /* array w/boundary conditions */
double new_a[N+2];     /* array w/boundary conditions */

void jacobi()
{
    int i, t;

    for (t = 1; t <= ITERS; t = t + 2)
    {
        for (i = 1; i <= N; i++)
        {
            new_a[i] = old_a[i-1] + old_a[i+1] + b[i];
        }
        for (i = 1; i <= N; i++)
        {
            old_a[i] = new_a[i-1] + new_a[i+1] + b[i];
        }
    }
}

```

**A:** Write a new version of the `jacobi()` loop to compute in parallel on a shared-memory machine using `NPROCS`, `MYPID` and `barrier()`. Assume `N` is a multiple of `NPROCS`.

**B:** Where does the communication occur in your shared memory application? Ignore the communication that obviously takes place within `barrier()`. How many bytes *must* be sent/received per processor per iteration? As we will see shortly, cache coherence protocols tend to multiply the number of bytes *actually* sent/received by a nontrivial factor.

**C:** Describe briefly how a message-passing version of the application (e.g. using Active Messages) would be written.