Parallel Algorithms – Part 2

The Design of Parallel Algorithms

A major challenge facing computer scientists today, given the existence of massively parallel machines, is to design algorithms which exploit this parallelism. There are three main approaches to the design of parallel algorithms.

1. Modify existing sequential algorithms exploiting those parts of the algorithm that are naturally parallelizable. To some extent, this is what we did in the last set of notes with the algorithm to find the largest key from a set of keys. The tournament method (also called the binary fan-in technique) is not unique to parallel algorithms, indeed the same technique can be applied sequentially, however, that part of the algorithm is inherently parallel.
2. Design an entirely new parallel algorithm that may have no natural sequential analog.
3. For some problems, such as finding roots, the same sequential algorithm is run on many different processors concurrently with different seed values until one of the processors reports "success". That is, all the processors start running a sequential algorithm with different initial conditions, and the first processor to achieve the desired result "wins the race."

All three of these strategies are viable in certain situations and we will see examples of each as we explore parallel algorithms further.

Architectural Constraints When Designing a Parallel Algorithm

A number of constraints arise when designing parallel algorithms that do not occur when designing sequential algorithms. These constraints are imposed by the architecture of the particular parallel machine on which the algorithm is intended to be executed. We eluded to some of these constraints in the last section of notes and now we will expand this discussion somewhat. There are five basic constraints that we will examine in some detail, these are:

1. Single instruction versus multiple instruction architecture. Do all the processors execute the same instruction or different instructions concurrently?
2. The number and type of processors that are available.

3. Does the architecture support the PRAM model of shared memory or does it support a distributed memory through an interconnection network?
4. Communication constraints. PRAM models have read/write restrictions while interconnection networks must specify (through a graph) the direct connections that exist between processors.
5. I/O constraints. How is the connection to the "outside world" handled.

Single Instruction vs. Multiple Instruction

A single-processor computer can only execute one instruction at a time. A parallel computer with *p* processors can execute *p* instructions concurrently. Each processor may operate on possibly different data. In the PRAM model, each processor executes the same instruction on possibly different data concurrently in a synchronized manner. This common instruction also contains information that can instruct a given processor to remain idle (*masked out*) during a given step. The operations of each processor must be controlled by a front-end processor (*central control)* and a global clock. During each time interval of the global clock, all the processors concurrently perform the same operation (input or output data, perform computations on data, read from local memories, communicate between processors, and so forth). Since the operations pulse through the system in regular clock intervals, this model is often referred to as *systolic computing*. Parallel computers that follow this model are called SIMD (**S**ingle **I**nstruction **M**ultiple **D**ata) machines. Parallel machines that allow different instructions to be performed at the same time on possibly different data are called MIMD (**M**ultiple **I**nstruction **M**ultiple **D**ata) machines. Since SIMD machines are conceptually simpler and easier to implement, we'll focus on SIMD machines.

Number and Type of Processor

In practice, computer manufacturers must decide whether to build a coarse-grain computer, one which has tens or hundreds of powerful processors, or a fine-grain computer, one which has thousands and thousands of relatively simple processors. For our modeling purposes we will consider that the processors available are powerful enough to execute all the normal instructions of a serial computer.

There are two approaches to designing parallel algorithms with respect to the number of processors available. The first approach is to design the algorithm where the number of processors used by the algorithm is an input parameter. In this approach, the number of processors *p* does not depend on the input size *n.* The second approach is to allow the number of processors to grow with the

size of the input. Thus, the number of processors *p* is not an input parameter but is a function *p(n)* of the input size *n*. Most modeling of parallel algorithms is done using this second approach. Further, an algorithm designed under this second approach can always be converted into an algorithm suitable for a fixed number of processors. For example, if an algorithm has been designed that utilizes $p(n) = n^2$ processors, then for each parallel step of the original algorithm, in the converted algorithm the work is divided between among the *p* processors into at most n^2/p sequential steps done by each processor.

PRAM and Mesh Model Communication Constraints

PRAM Model Communication Constraints

Since the PRAM model uses a shared memory, the possibility of a conflict arises if two or more processors attempt to read or write from the same memory location simultaneously. There are four models for dealing with such conflict: EREW (exclusive-read, exclusive-write), CREW (concurrent-read, exclusive-write), ERCW (exclusive-read, concurrent-write), and CRCW (concurrent-read, concurrent-write). The ERCW is of little theoretical or practical interest, so we will consider it no further. Figure 1 illustrates the four conflict models of PRAMs.

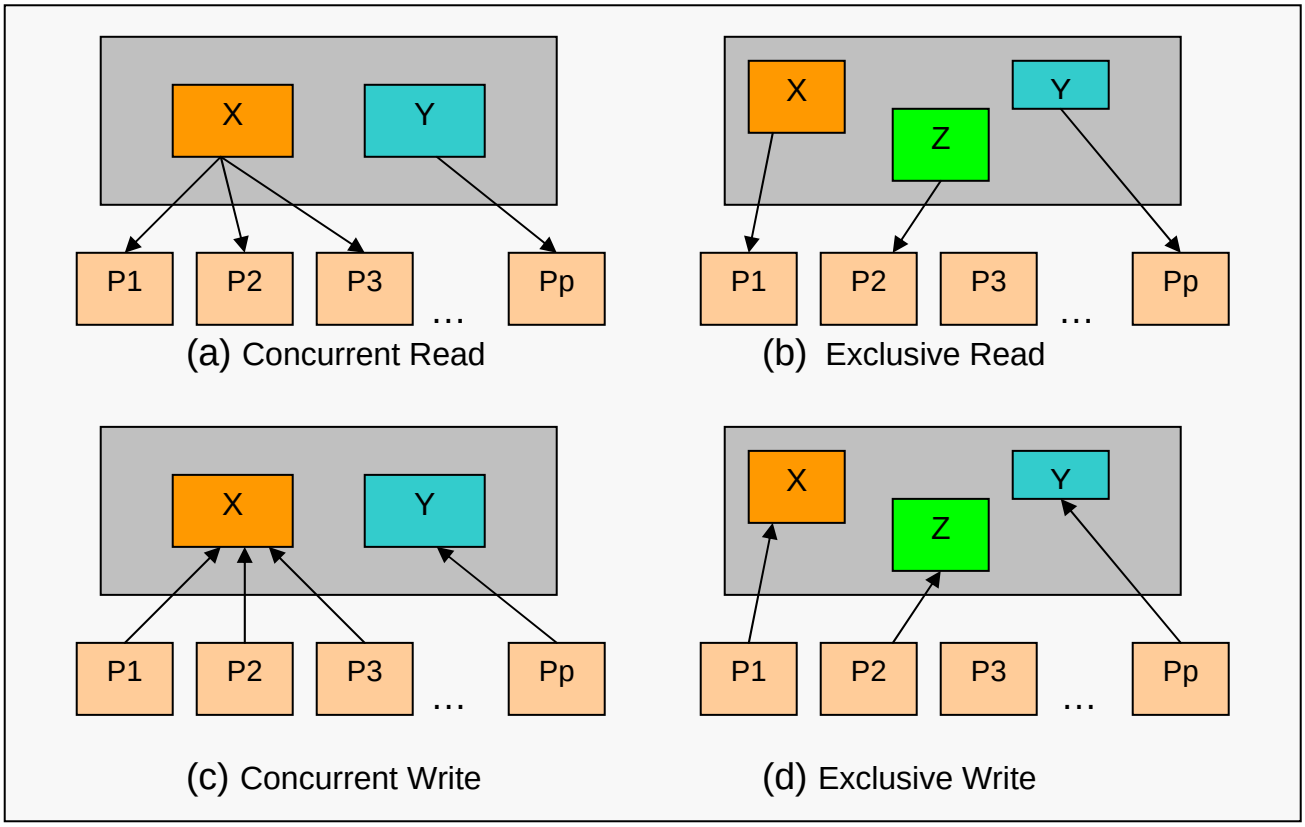


Figure 1 - Read/Write Possibilities for PRAMs.

Algorithms for EREW PRAMs are assumed to be in error if a read or write conflict ever occurs. CRCW PRAMs allow multiple processors to read and write from the same memory location concurrently. Resolution of concurrent writes is handled in various fashions. A commonly used technique only allows concurrent writes when all the processors are attempting to write the same value. Another method, which can be applied to numeric data, is to write the sum of all these values. Still other methods involve allowing a randomly chosen processor among the contending processors to write its value, or establishing a total ordering of the processors and allow the processor with the smallest value (typically *pid* based) to write first, and so on.

The EREW model is the most realistic of the PRAM models to build in practice. Further, any algorithm designed for an EREW PRAM will run without alteration on the other PRAM models. Unless otherwise noted, all PRAM algorithms assume the EREW model. With the current state of technology, EREW PRAMs are difficult to build (although there are efforts currently underway to do so). Nevertheless, the PRAM model is still a very good model for theoretical results and the initial design of a parallel algorithm without the burden of processor communication details getting in the way of the design.

Mesh Model Communication Constraints

Most parallel computers built today more closely follow the guidelines of the hypercube and degree-bounded network models (also called *mesh* models). This means that there is an interconnection network which links the processors together. In these models, each processor has its own RAM with no common shared memory accessible to each processor. Since we are focusing on SIMD machines, variables in processor memories each have instantiations in *every* processor, and are thus called *parallel* or *distributed* variables. In other words, if *x* is a distributed variable, then each processor in the network has a memory location reserved for its own version of *x*.

In the interconnection network models, the assumption is that each processor has sufficient memory to handle the various tasks to which it will be applied. Nevertheless, parallel algorithms are usually written so that they require only a constant (independent of input size) number of distributed variables. Once again, statements which involve distributed variables might only be executed in a subset of the available processors. Certain processors can be masked out at certain steps.

Information is communicated between processors using messages sent along the network. Messages pass along routes in the network where each link in a route is between directly connected (adjacent) processors. To avoid routing

conflicts most parallel algorithms will assume communication occurs in each step between adjacent processors only. While this is not necessarily true, in general for parallel machines, it again, makes the algorithms somewhat easier to develop and analyze if we can remove such detail from the algorithm.

To describe how communication takes place between adjacent processors P_X and P_Y, suppose the central control instructs P_Y to assign to the variable *y* in its local memory the value of the variable *x* in the local memory of P_X. This is accomplished as follows: P_X reads the value of local variable *x* and sends this value along a link in the network to P_Y. Upon receiving this message, P_Y writes this value into its copy of *y*. [In parallel pseudo-code: $\mathbf{P}_Y$:y $\Leftarrow$ $\mathbf{P}_X$:x].

For the time being, we'll focus on the mesh model for developing parallel algorithms using the interconnection network model. For our example, we'll use the two-dimensional mesh shown in Figure 2.

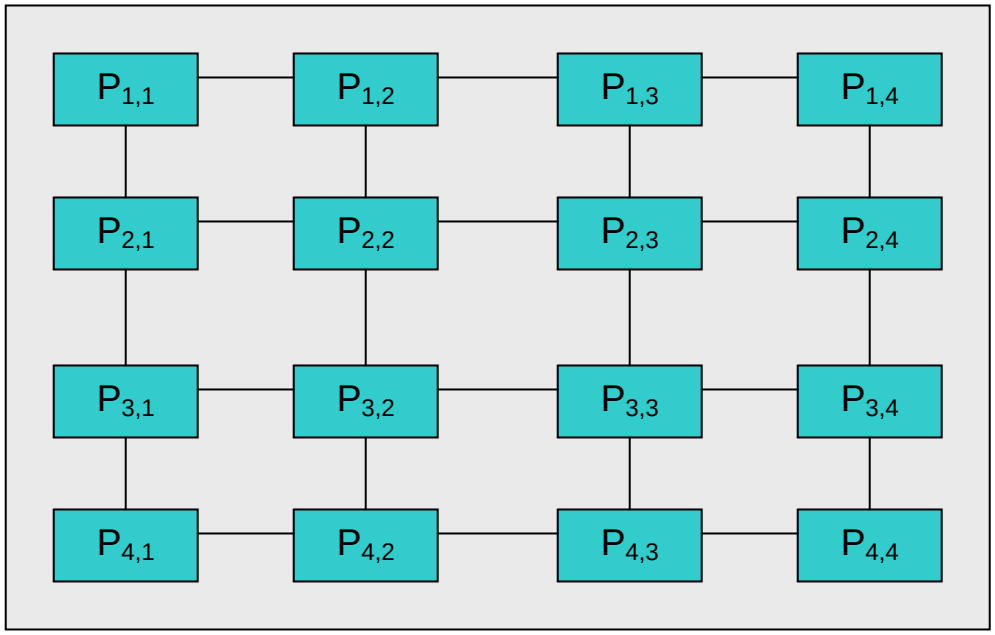


Figure 2 – Two dimensional mesh of degree 4 ($M_{4,4}$).

The two dimensional mesh $M_{q,q}$ with $p = q^2$ processors $P_{i,j}$, $i, j \in \{1, \ldots, q\}$ has $P_{i,j}$ directly connected with $P_{r,s}$ iff, $i = r$ and $|j-s| = 1$ or $|i-r| = 1$ (see Figure 2 above).

I/O Constraints

As with any computer, a parallel machine must have some mechanism to read "outside world" data from external input devices into the processor's local memories, as well as to write data from these memories to external output devices. Most parallel algorithm development takes a very high-level approach

to this type of constraint, leaving the exact nature of the I/O mechanism unspecified. Similar to single processor systems, we assume the availability of suitable parallel versions of *read* and *write* statements.

With the PRAM model, the assumption is that central control has already placed input data for the algorithms into the shared memory. This fact means that relatively little use is made of read/write statements in the PRAM model. Similarly, the interconnection network models assumes that central control has already distributed data to the local memories of the relevant processors without specifying the mechanism for accomplishing it. However, more use is made of read/write statements in interconnection models than in the PRAM model, since in the former models, it is important to specify how the data gets distributed to the processor's local memories. Further, in interconnection models, when a procedure is called or a function is invoked, the algorithm must specifically delineate those processors than have meaningful data supplied to them on input or which are supplying meaningful output data.

Algorithm Development

Let's now examine the development of algorithms on both PRAM and interconnection models of parallel systems. As a running example for both types, let's consider the problem of searching, a fundamental problem tackled by computer systems everyday. Assume that we have a list L[1:n] of size *n* and we are searching the list for the occurrence of some key value *x*.

Example: Searching in the PRAM Model

We want to keep things as simple as possible while introducing parallel algorithms, so let's assume that we have a least *n* processors P_1, P_2, …, P_n available. A parallel search algorithm might be executed in a single step where processor P_i compares *x* to L[i], $1 \leq i \leq n$. What constraints have we assumed with this algorithm?

First, we assumed that each processor accessed (via a read) the memory location containing *x* simultaneously. This means that we have assume a concurrent read model.

Second, we assumed that a successful search can be signaled in a single step. If L[i] = *x*, then we simply have P_i write *i* to a memory location *index*. No problem arises, if *x* occurs at most once in *L*, however, if *x* occurs several times in the list, several processors will attempt simultaneous access to *index*. Thus, we assumed that our PRAM machine allowed for concurrent writes. This, of

course, implies that we have established a protocol for resolving write contention. Thus, we have assumed a CRCW PRAM machine (model).

Now let's assume the more realistic EREW PRAM model and see the different assumptions that must be in place for this algorithm to be successful. For each processor to have access to the value of *x* simultaneously in the EREW PRAM, we need to allocate an auxiliary array *temp*[1:n] and assign the value of *x* to each array element *temp*[i], $1 \le i \le n$. Assigning the value of *x* to each entry in *temp*[1:n] can be achieved by assigning *x* to *temp*[1] and then *broadcasting x* to the other positions in the array as follows: (For simplicity assume that $n = 2^k$ for some nonnegative integer *k*.) In the first step, P_i reads *x* and writes it to *temp*[1]. In the second step, P_1 reads *temp*[1] and writes it to *temp*[2]. In the third step, processors P_1 and P_2 read *temp*[1] and *temp*[2], respectively, and write *x* to *temp*[3] and *temp*[4], respectively. This broadcasting process is illustrated in Figure 3 when $x = 5$ and $n = 16$.

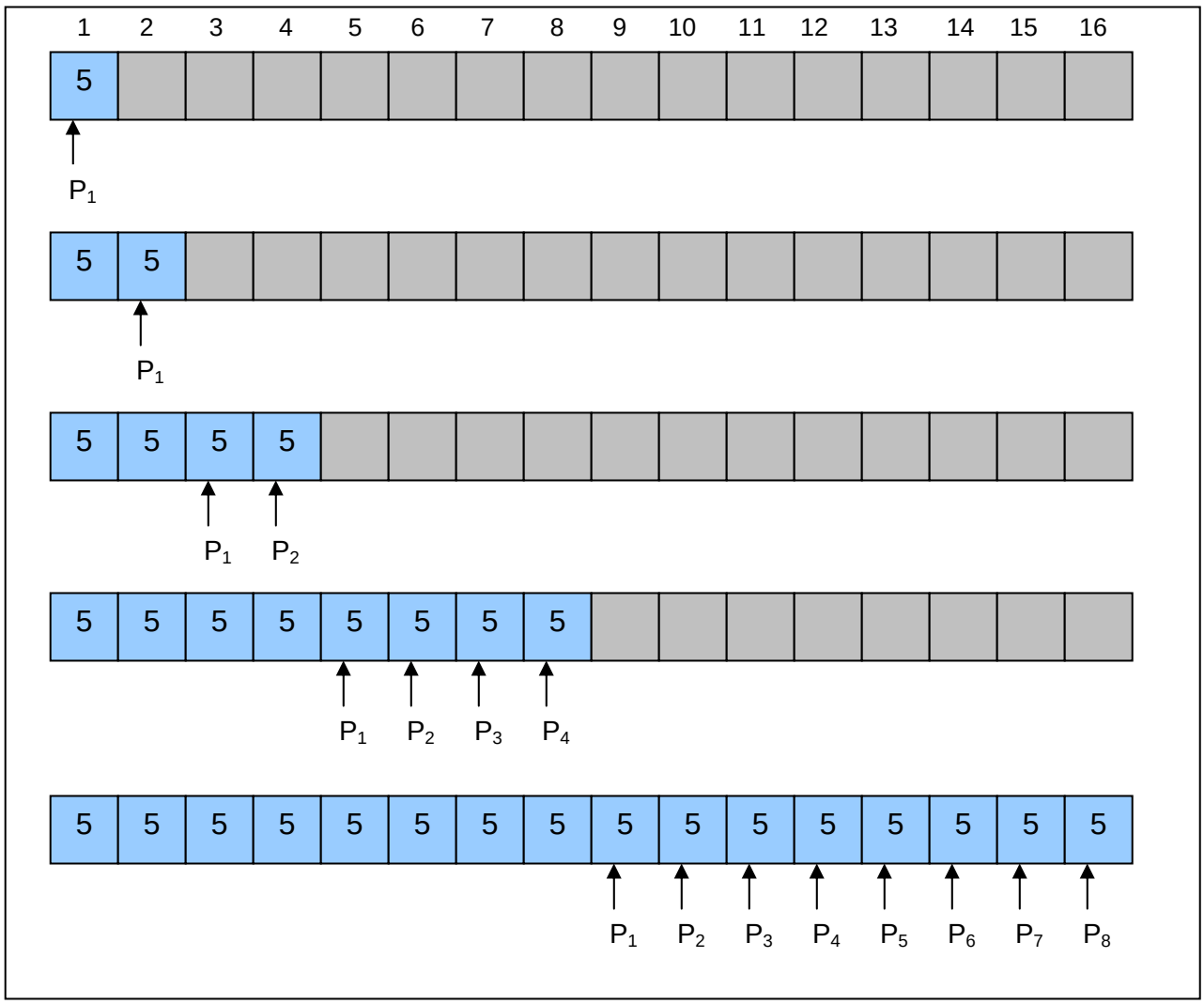


Figure 3 – Broadcasting the value of *x* into the array in $\log_2 n$ steps.

As Figure 3 illustrates, the broadcasting of x is complete after $log_2 n$ steps. Figure 3 also illustrates that not all processors must be active at every step in SIMD processing.

After x has been broadcast and *temp* has been filled, then, in parallel, processor P_i compares L[i] to *temp*[i] = x and writes i in *temp*[i] if L[i] = x; otherwise it writes the value ∞ (maxint) in *temp*[i]. The array *temp*[1:n] now contains the results of the search. The parallel search step is illustrated in Figure 4.

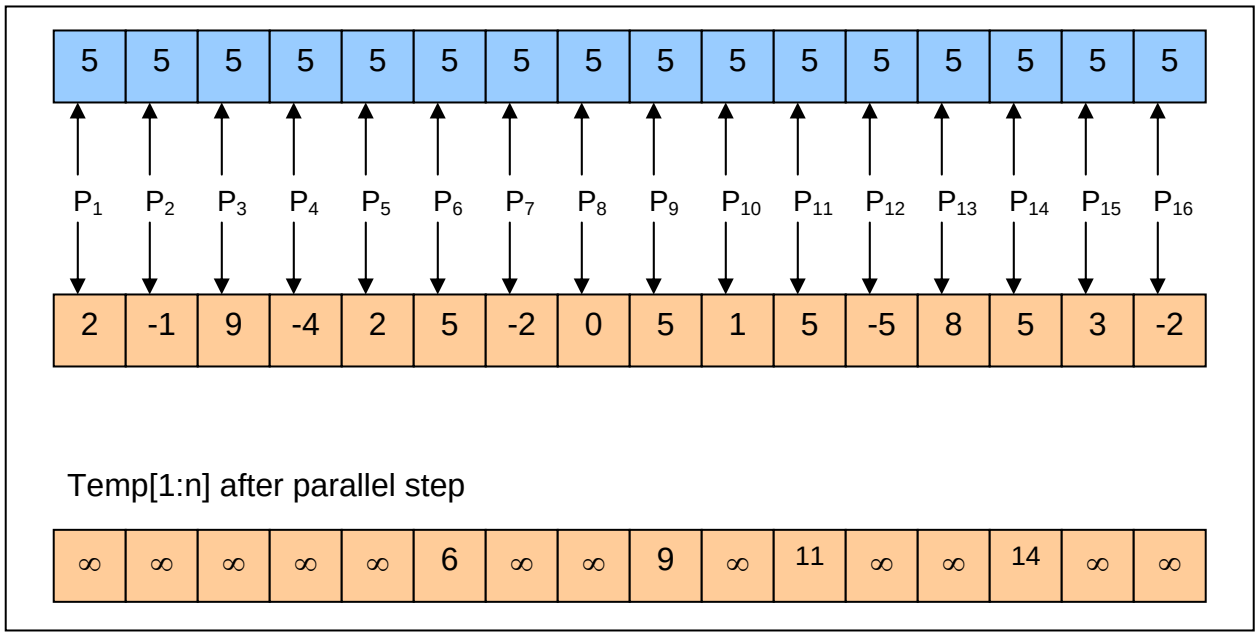


Figure 4 – Single parallel comparison step between search elements and list elements.

However, we are still left with the problem of signaling a successful search. Note that the value that we wish to return is nothing more than the minimum value in the *temp* array. Using the binary fan-in technique (see previous day's notes), we can obtain a straightforward parallel algorithm for determining the minimum of a set of n numbers on an EREW PRAM processor with $n/2$ processors. Using the binary fan-in technique we can reduce the 15 sequential steps required by a sequential processor to only four parallel steps, thereby achieving a speed-up of 15/4 over the sequential algorithm.

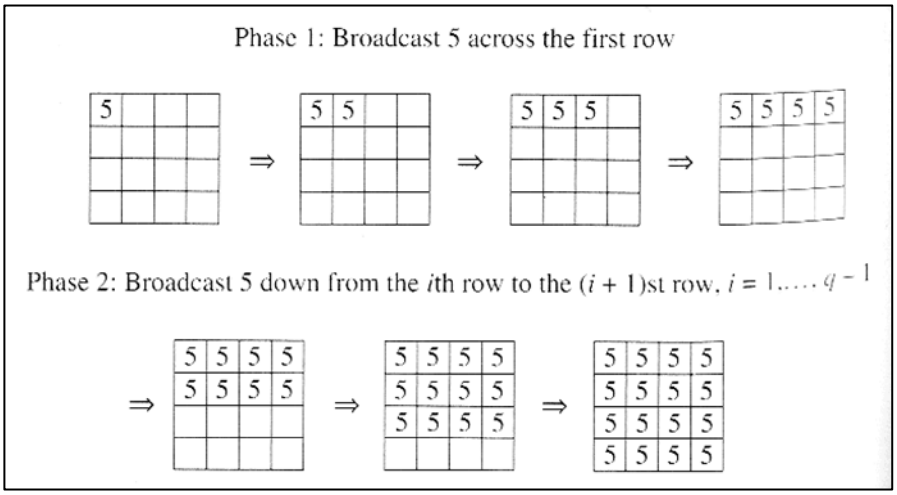
The basic operation for a sequential search algorithm was the comparison of a list element to the search element. In the parallel algorithm all such comparisons are made in a single step. Therefore, we must choose another basic operation to make a meaningful statement about the complexity (the number of parallel basic operations) of our parallel search algorithm. We could use either the number of parallel assignment statements performed when broadcasting the search element or the number of parallel comparison steps in

computing the index of the minimum value in *temp*[1:n] in the final phase. Either choice yields a complexity of $log_2 n$.

Example: Searching in the Two-dimensional Mesh Model

Now let's use the same searching example and develop a parallel algorithm for a two-dimensional mesh model (interconnection network model) and see how the constraints of the model differ for our algorithm. Let's consider the two-dimensional mesh $M_{q.q}$, and as before, for convenience, let's assume that the list has size $n = q^2$. As before, since the search element is a variable parameter and not a constant, it is not possible to initialize the distributed variable **P**:*x* with the value of the search element. We will assume that *x* is a front-end variable, and initially only $P_{1,1}$:*x* is assigned the value of *x*. We assume that the list *L*[1:n] has been input to the distributed variable *L* in row-major order. Therefore, element *L*[k] is assigned to processor $P_{i,j}$'s local instantiation of *L*, where: $k = (i-1)q+j$. Each instantiation of the distributed variable *index* is assigned the row-major value of its associated processor, which, unlike the assignment of the search element to the variable *x*, can be done in a single parallel step since it is a purely local computation (we assume, as typically the case, that each processor $P_{x,y}$ knows its own index *x,y*). Figure 5 illustrates the distribution of initial values in the mesh.

As with the PRAM, we need to broadcast the value of the search element throughout the distributed variable *x*. The following simple two-phase procedure will broadcast the value of $x = \mathbf{P}_{1,1}$: *x* throughout the entire distributed variable $\mathbf{P}_{x,y}$:*x*, $1 \le x, y \le q$ using $2q-2$ parallel communication steps. In the first phase, *x* is broadcast across the first row in *q-1* steps, where in the *i* th step, processor $P_{1,j}$ communicates *x* to its neighbor $P_{1,\,j+1}$ on the right, $i = 1, 2, \ldots, q-1$. In the second phase, *x* is broadcast down row by row. This is illustrated below:



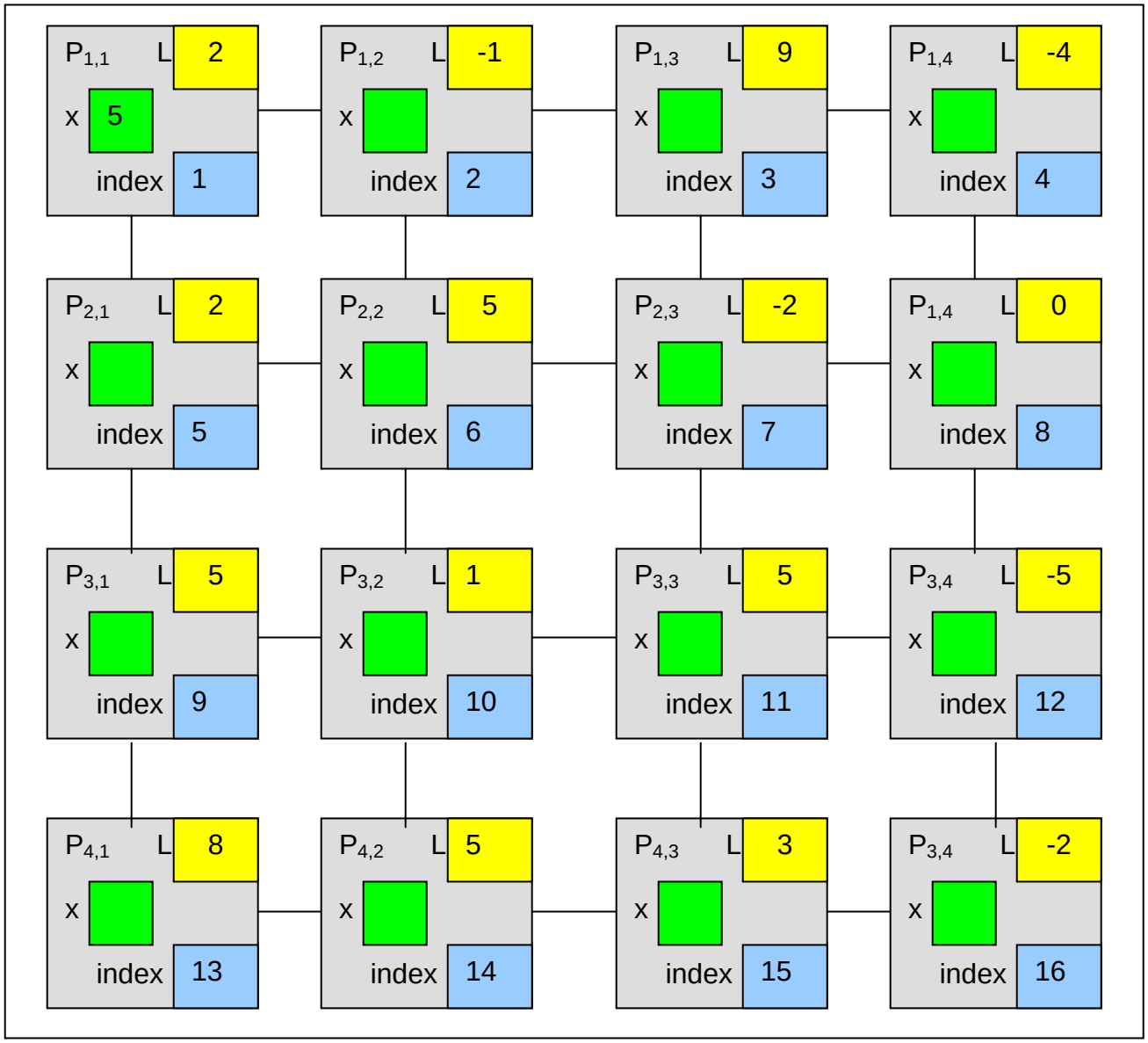


Figure 5 – Initial states of the distributed variables *L, x,* and *index* in the mesh $M_{4,4}$.

After the search element has been broadcast to all *n* processors (so that each $\mathbf{P}_{i,j}$: *x* contains the value of the search element, in a single parallel comparison step each processor $P_{i,j}$ compares $\mathbf{P}_{i,j}$: *x* to its list element $\mathbf{P}_{i,j}$: *L* and writes the value ∞ to $\mathbf{P}_{i,j}$: *index* if the search element is not equal to the list element.

After the single parallel comparison step, the distributed variable *index* contains the results of the search. Figure 6 illustrates the configuration of the mesh after the parallel comparison step has completed.

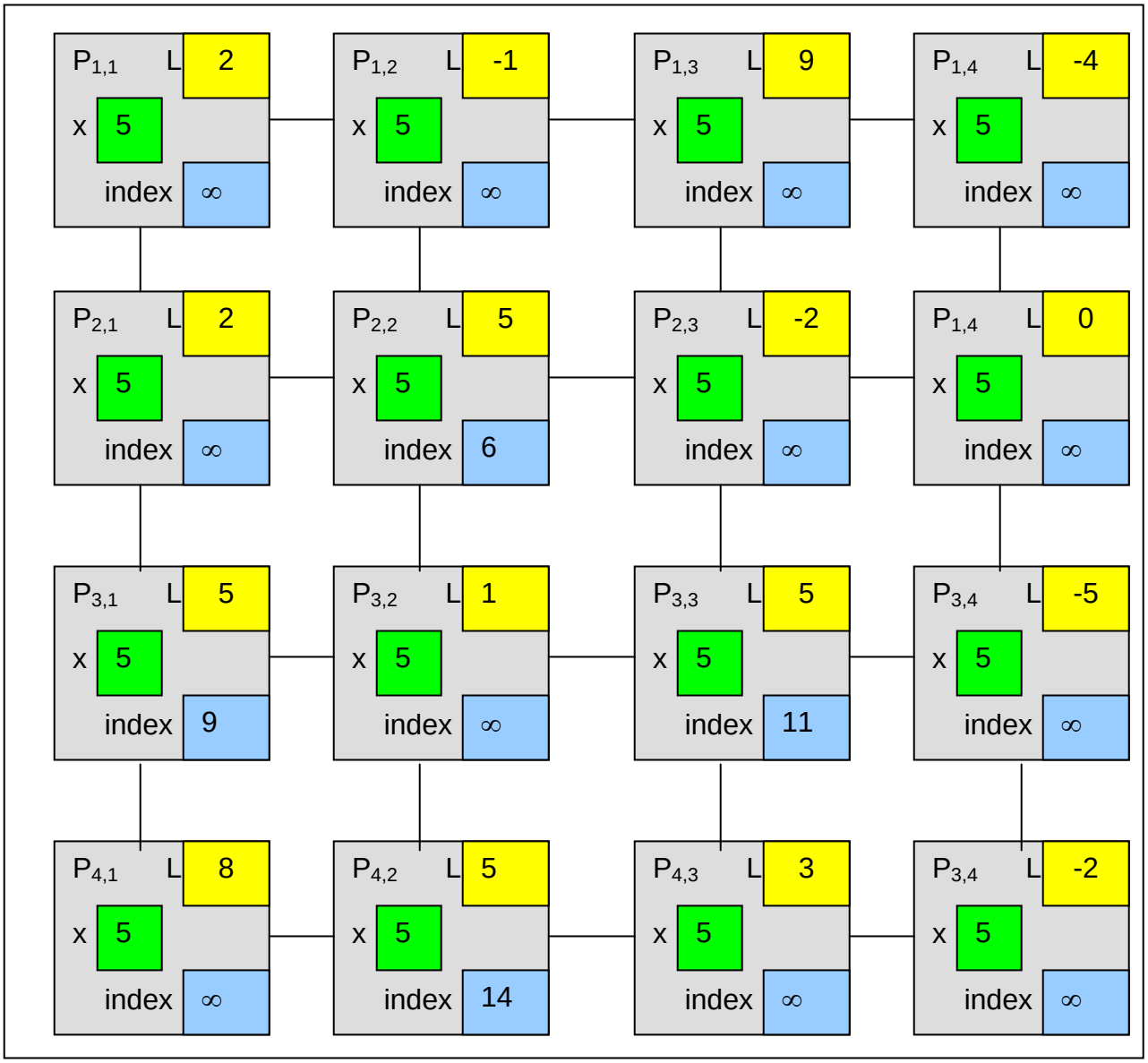


Figure 6 – State of the distributed variables in the mesh after value 5 has been broadcast throughout *x* and the single parallel comparison step has been performed.

As before, the mesh now contains the results of the search, but how do we return the result? Typically, whenever a scalar-valued function defined on an interconnection terminates, the value to be returned by the function resides in a particular processor's local instantiation of a suitable distributed variable. Let's assume that processor $P_{1,1}$ is our designated processor to return the value of the search in its instantiation of the distributed variable *index*. In other words, at the termination of the search, we want $\mathbf{P}_{1,1}$: *index* to hold the smallest index *i* such that *L*[i] = *x*, or ∞ if no such index exists.

To compute this minimum value, we need to perform what amounts to a reverse broadcast procedure. In phase 1, column minimums are computed as shown in Figure 7.

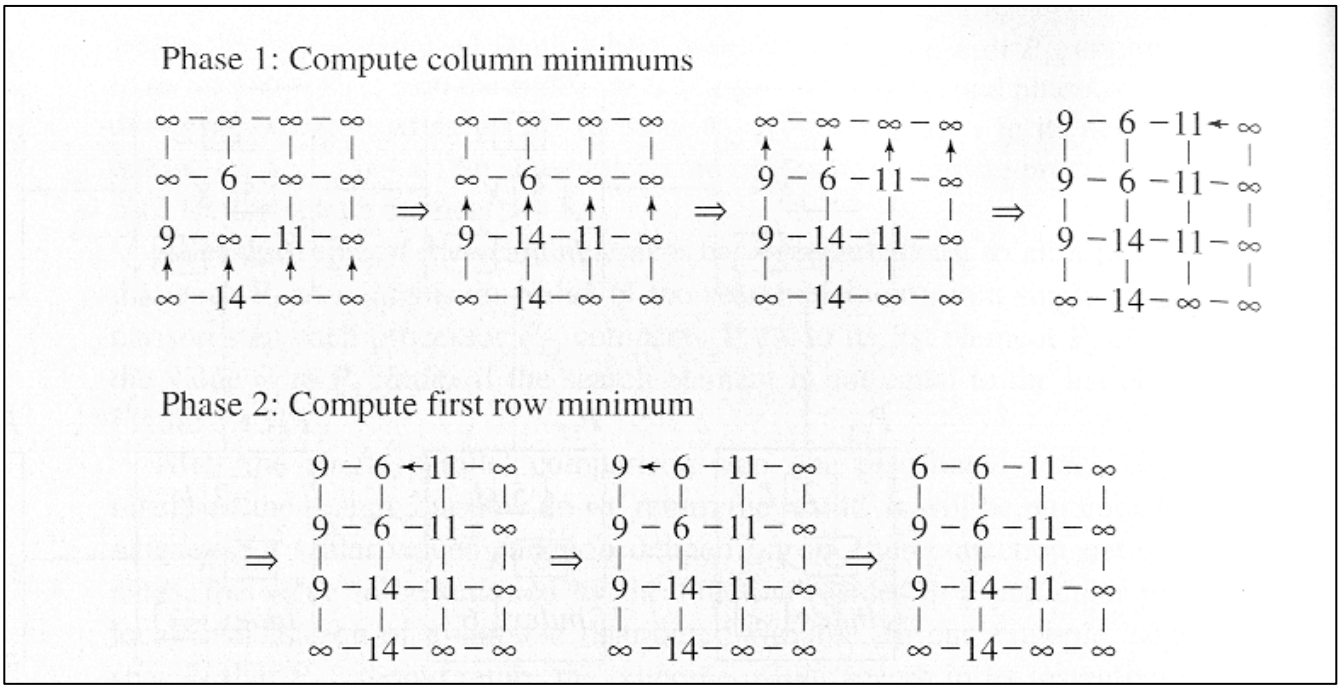


Figure 7 – Computing the minimum value on a 2-d mesh. Arrows indicate how the minimums are computed at each state. Final result is placed in $P_{1,1}$.

After completion of the computation of the minimum index value, the mesh will have the state as shown in Figure 8.

Similar to the search algorithm we developed for the PRAM model, the search algorithm for the 2-d mesh performs a single parallel comparison between the search element and the elements of the list. Therefore, the communication complexity is the appropriate metric of the complexity of the mesh search algorithm. The number of communication steps for both the broadcast phase and the reverse-broadcast phase is 2*q-2*. Thus, the complexity of the mesh algorithm is $4q-4 = 4\sqrt{n} - 4$, so we have achieved a speed-up of:

$$\frac{n}{4\sqrt{n}-4}$$

over the sequential search algorithm using the 2-d mesh parallel algorithm.

Figure 8 – Final state of the distributed variables in the 2-d mesh upon completion of the searching algorithm.