Introduction to Parallel Algorithms

Introduction

For the bulk of the term, we have discussed advanced data structures and some of the applications to which they have been suited for representing data. Each of these data structures was designed to provide efficient access to the data stored in the structure. Some of these data structures were subtle variants of a more general type of structure, with the subtle variation occurring to enhance access to the data for certain situations arising in the data. For example, the prefix B^+-tree enhanced access through the use of prefixes that exist in the key values maintained in the structure. Don't lose sight of the fact that a data structure's sole purpose is to maintain and provide access to data which is used to support the algorithm which uses that data. Data structures, no matter how complex, often represent a trade-off in terms of time and space, as there is typically no optimal data structure which covers all possible problem instances. Recall that this is the very reason that so many variants of many data structures exist.

We now change our focus from the underlying data structures supporting the algorithm to the algorithms themselves. In some respects, a data structure is chosen because of how naturally it represents the data which defines a problem. For example, trees naturally fit with data that represent hierarchical relationships. Algorithms too exhibit this characteristic, for example, divide and conquer algorithms arise from the natural problem solving strategy of dividing a complex problem into smaller, more manageable pieces. What we are about to examine are algorithms which carry this natural technique much further.

Most models of computation represent the computer as a general-purpose, deterministic, random access machine (a vonNeumann machine). Algorithms which can be executed by vonNeumann type machines are called *sequential algorithms* (sometimes also called *serial algorithms*). We are about to examine models of computation that present a much different machine, one in which several instructions can be executed simultaneously. Generally, referred to as, *parallel machines* or *parallel computers*, these are computers which have more than one processor operating in parallel. Over the years, there have been many different models of parallel computation that have been developed. As with sequential machines, parallel machines are best suited to certain classes of problems and to take advantage of a parallel architecture, algorithms must be developed specifically for the parallel architecture. We will see several parallel models and discuss their relative merits and weaknesses.

In recent years, as microprocessors have become cheaper and the technology for interconnecting them has improved, it has become both possible and practical to build general-purpose parallel computers containing a very large number of processors. Parallel algorithms are natural for many applications. In image processing, for example in vision systems for robots, different parts of a scene can be processed simultaneously in much the same way that you process a scene in parallel. Parallelism can speed up the computation for graphics displays (i.e., Intel's AGP and similar systems). In search problems, different parts of the database can be searched in parallel. Simulation programs often do some computation to update the states of a large number of components in the system being simulated; these can be done in parallel for each simulated time step. Artificial intelligence applications (which include image processing and a lot of searching) can also benefit from parallel computation.

Parallelism

If the number of processors in parallel computers were small, say somewhere between two and six, then there would be a practical advantage to using them for some problems in which computation could be speeded up by some small constant factor. However, when discussing the performance of computational algorithms we often ignore small constants (recall Big-Oh, etc.) which would make such machines, and their algorithms, rather uninteresting. Parallel algorithms become interesting from a computational complexity point of view when the number of processors is very large, larger than the input size for many of the actual problem instances for which the algorithm in question is utilized. This is where significant speed-up and interesting algorithms can be found and this is the only area of parallel computation that we will examine.

How much can parallelism do for us? Suppose that a sequential algorithm for a problem does W(n) operations in the worst case for an input of size *n*, and assume that we have *p* processors. Then the best that we can hope for from a parallel algorithm is to run in W(n)/p time. Furthermore, we can't guarantee to achieve the speed up in all cases. Consider the following example which illustrates a fundamental problem with parallel computation.

> Suppose that our problem is putting on our socks and shoes. Let's assume for this problem that a processor is a pair of hands. A common sequential algorithm is: put on the right sock, put on the right shoe, put on the left sock, put on the left shoe. (What algorithm do you use?) As a sequential algorithm this takes 4 time units. If we have two processors we can assign one to each foot and accomplish the task in 2 time units instead of four. However, if we have four processors, we can't cut the time down to one time unit, because the socks must go on before the shoes.

There are several general-purpose and special-purpose models of parallel computers that correspond to various (either real or theoretical) hardware designs. As we examine some of these models we will look at some parallel algorithms within the model which will illustrate the techniques of parallel computation. We won't always give the most efficient algorithm, but ones that illustrate the concepts of the model well and are also fairly easy to grasp. Some aspects of parallel computation are quite difficult to grasp and our intention here is to give you some background into an area of computation that may soon become the dominant model of computation.

The PRAM Model

The *parallel random access memory* (PRAM, pronounced "p ram") model of parallel computation consists of p general-purpose processors, P_0, P_1, …, P_{p-1}, all of which are connected to a large shared, random access memory M, which is treated as a (very large) array of integers. This model is illustrated in Figure 1.

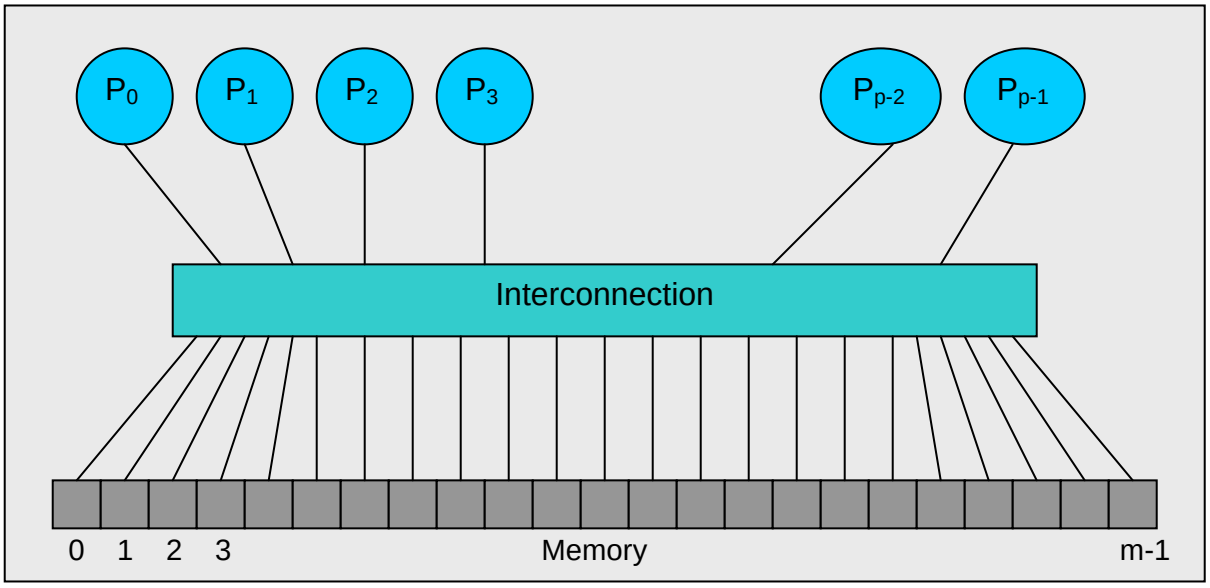

Figure 1 – PRAM model.

The processors have a private memory (local memory) for computational use, but all communication among them takes place via the shared memory. Unless it is otherwise indicated, the input for an algorithm is assumed to be in the first n memory cells, and the output is to be placed in cell 0 (or an initial sequence of cells). All memory cells that do not contain input are assumed to contain zero when a PRAM algorithm (program) begins execution.

All the processors run the same program, but each processor "knows" its own index (called the *processor id* or *pid*), and it "knows" the input size, usually designated as *n*, sometimes as a pair *(n,m)* or some other small, fixed set of parameters. The program (algorithm) may instruct processors to do different things depending on their *pid*s. Frequently, a processor uses its *pid* to calculate the index of the memory cell from which to read or into which to write.

PRAM processors are synchronized; that is, they all begin each step at the same time, all read at the same time, and all write at the same time, within each step. Some processors might not read or write in certain steps. Each time a step has two phases; the read phase, in which each processor may read from a memory cell, and the write phase, in which each processor may write to a memory cell. Each phase may include some O(1) computation using local variables before and after its read or write. The time allowed for these computations is the same for all processors and all steps so that their reading and writing remain synchronized. The model allows processors to do lengthy (but O(1)) computations in one step because for parallel algorithms, communication among processors through the shared memory (i.e., reading and writing) is expected to take considerably longer than local operations within one processor. There are several different models with different assumptions about how much information fits in one memory cell and which local operations are available. The algorithms that we will deal with a bit later, work with the weakest of these assumptions, so these algorithms will be robust in this sense.

In the PRAM model, any number of processors may read the same memory cell concurrently (i.e., at the same step). This is known as the *concurrent read* model. There are also several models that prohibit concurrent reads, known as *exclusive read* models. There are also several variants of the PRAM model that differ in the fashion in which they handle write conflicts. Initially, we will consider only algorithms that do not exhibit write conflicts and later we will consider how to handle write conflicts.

Several programming languages exist for describing parallel algorithms, but for now we will use a mixture of English and pseudocode before examining any specific language. Types are usually omitted in function headers in PRAM algorithms since the model only supports integers and arrays and we'll make the types clear in the context. PRAM algorithms can contain *for* and *while* loops as each processor can keep track of a loop index and do the appropriate incrementing and testing during the computational phases of its execution.

The use of arrays in the PRAM model mirrors the use of arrays in any high-level programming language such as Java. That is, the compiler decides on some fixed arrangement of the arrays in memory following the input, and translates

array references to instructions to compute specific memory addresses. For example, if the input occupies *n* cells, and *alpha* is the third *k*-element array, the compiler translates an instruction telling processor P_i to read *alpha[j]* into PRAM instructions to compute *index = n + 2 * k + j*, and then read *M[index]*. The address computation is completed within one PRAM step.

PRAM vs. Other Parallel Models

Although the PRAM model provides a good framework for developing and analyzing algorithms for parallel machines, the model would be difficult and expensive to provide in actual hardware. The PRAM assumes a complex communication network that allows all processors to access any memory cell at the same time, in one time step, and to write in any cell in one time step. Thus, any processor can communicate with any other in two steps: One processor writes some data in a memory location on one step, and the other processor reads that location on the next step. Other parallel computation models do not have a shared memory, thus restricting communication between processors.

A model that more closely resembles some actual hardware is the *hypercube.* A hypercube has 2^d processors for some *d* (the *dimension*), each connected to its neighbors. Figure 2 shows a hypercube of dimension 3.

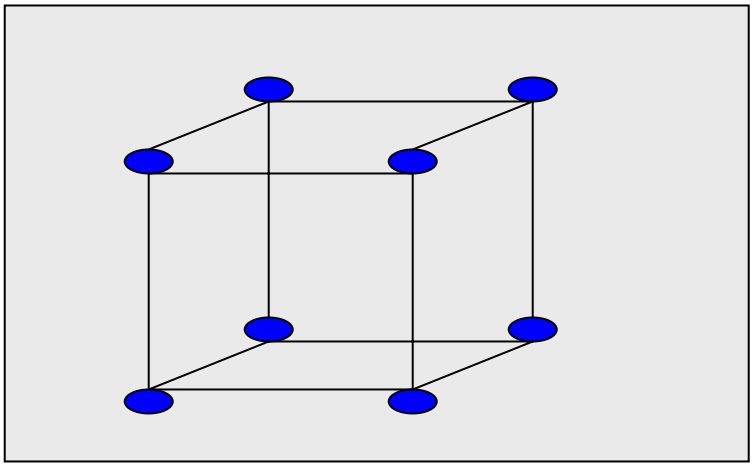

Figure 2 – Hypercube of dimension 3 (degree 3).

In the hypercube, each processor has its own memory and communicates with the other processors by sending messages. At each step each processor may do some computation, then send a message to one of its neighbors. To communicate with a non-neighbor, a processor may send a message that includes routing information indicating the ultimate destination; the message might take as many as *d* time steps to reach its destination. In a hypercube with *p* processors, each processor is connected to $log_2 p$ other processors.

Another class of models, called *bounded-degree networks*, restricts the connections still further. In a bounded-degree network, each processor is directly connected to at most *d* other processors, for some constant *d* (the *degree*). There are different designs for bounded-degree networks, however, an 8×8 network is illustrated in Figure 3.

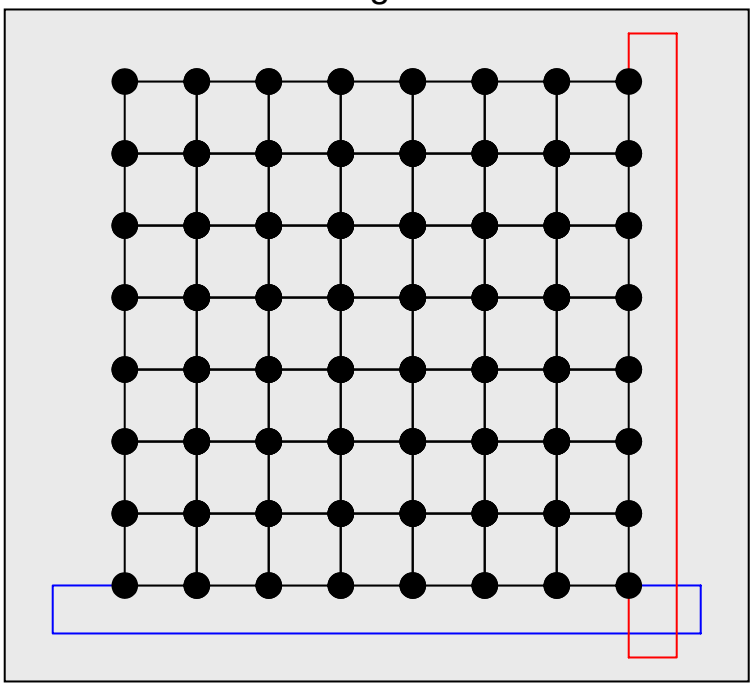

Figure 3 – Bounded-degree network (degree 4).

Hypercubes and bounded-degree networks are more realistic models than the PRAM model, but algorithms for them can be much harder to specify and analyze. The routing of messages among the processors, an interesting problem in itself, is eliminated in the PRAM model.

The PRAM model, while not very practical, is conceptually easy to work with when developing algorithms. Therefore, a lot of effort has gone to finding efficient techniques to simulate PRAM computations on other parallel models, particularly models that do not have shared memory. For example, each PRAM step can be simulated in approximately $O(\log_2 p)$ steps on a bounded-degree network. Thus we can develop algorithms for the PRAM, and know that these algorithms can be translated into algorithms for actual machines. The translation may even be done automatically by a translator program.

Parallel Computations and Intractability

Recall that the class of problems *P* was defined to distinguish between tractable and intractable problems. Class *P* consists of those problems that can be solved in polynomially bounded time. For parallel computation, too, problems are classified according to their use of resources: time and processors. The

class NC^1 consists of problems that can be solved by a parallel algorithm with p (the number of processors) bounded by a polynomial in the input size, and the number of time steps bounded by a polynomial in the *logarithm* of the input size. More succinctly, if the input size is n, then $p(n) \in O(n^k)$ for some constant k, and the worst-case time, $T(n)$, is $O(\log^m n)$ for some constant m. (Recall that $\log^m n = (\log n)^m$).

The time bound for the class *NC*, sometimes referred to as "poly-log time" because it is a polynomial in the log of n, is quite small – but we expect parallel algorithms to run quite fast. The bound on the number of processors is not so small. Even for $k = 1$, it may not be practical to use n^k processors for moderately large input. The reasons for using a polynomial bound, rather than some specific exponent, in the definition of *NC* are similar to the reasons for using a polynomial bound on time to define the class *P*. For one, the class of problems that can be solved in poly-log time using a polynomially bounded number of processors is independent of the specific parallel computation model that is chosen from a large class of models that would be considered as "reasonable". Thus, *NC* is independent of whether we are using a PRAM or bounded-degree network model. Secondly, if a problem *cannot* be solved quickly with a polynomial number of processors, then that is a strong statement about how hard the problem is. In fact, for many algorithms (certainly most of the ones that we will examine), the number of processors is O(n).

PRAM Algorithms

This section will introduce some commonly used techniques for PRAM computation by developing some simple PRAM algorithms that will illustrate the "flavor" of parallel algorithms.

In general, PRAM algorithms are "theoretical" in the sense that they demonstrate that a problem can be solved within a time that is in some asymptotic order class. There are no real PRAMs that magically have more processors for larger inputs, without limit. Therefore, there is little point in trying to optimize constant factors, since the algorithm will not actually run as it is. Instead we will try to present the algorithms as simply as possible and with as much clarity as the model allows.

Consider the problem of finding the largest key in an array of n keys. A common sequential algorithm to solve this problem is to proceed through the array comparing *max*, the largest key found so far, to each remaining key. After

[1] The class NC was defined and named by Steven Cook in 1985 as an abbreviation for "Nick's class." The name refers to Nick Peppenger who studied the same class of problems earlier in 1979, but did so in terms of circuit complexity rather than parallel computation. The class NC has several other equivalent definitions.

each comparison, *max* may change; we can't do the next comparison in parallel because we don't know which value to use for the next compare. However, if a different approach is used to solve the problem then parallelism can help solve the problem faster. Consider, as a case in point, the tournament method for solving this problem. In the tournament method, elements are paired off and compared in "rounds". In succeeding rounds, the winners from the previous round are paired off and compared as shown in Figure 4.

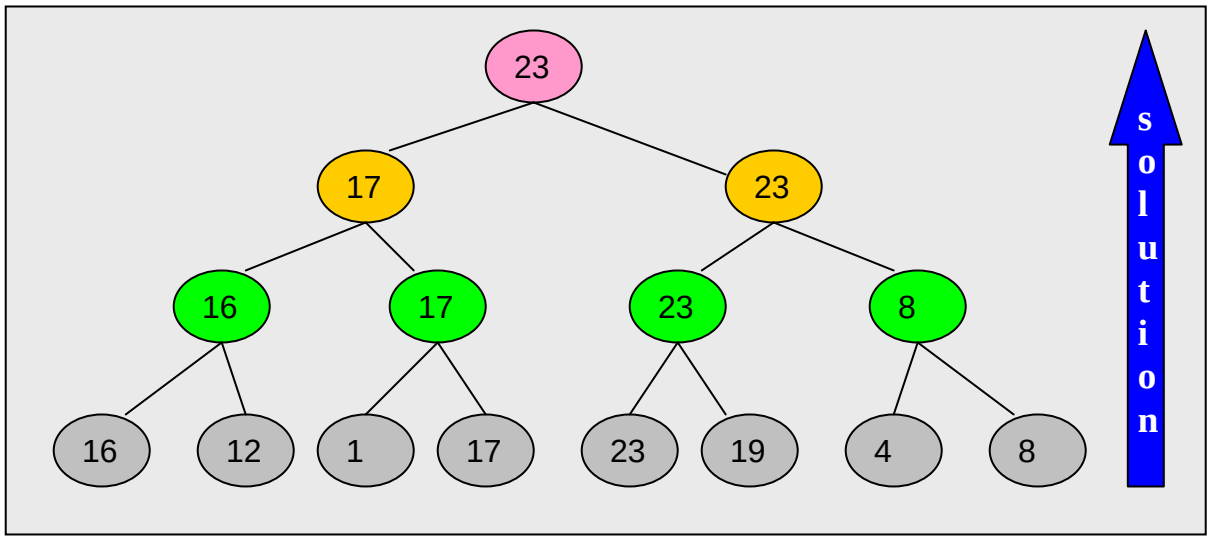

Figure 4 – Tournament tree for finding maximum key from *n* keys.

In Figure 4, the array of *n* keys is represented by the leaf nodes of the tournament tree. Clearly, the largest key is found in $\lceil \log_2 n \rceil$ rounds. All of the comparisons that occur in one round can be performed concurrently (in parallel). Therefore, the tournament method gives naturally gives us a parallel algorithm to solve the maximum key problem.

In a tournament, the number of keys under consideration at each round decreases by half, so the number of processors needed at each round decreases by half. However, to keep the description of the algorithm simple and clear, we'll specify the same instructions for all processors at each time step. The extra work being done by the processors which are no longer "in the tournament" may be confusing, so before looking at the actual algorithm consider Figure 5 which illustrates the actual work that is being done which contributes to the answer. A straight line represents a *read* operation, a zigzag line represents a *write* operation; a processor writes the largest key it has seen in the memory cell with the same number as the processor (i.e., processor P_i writes in M[i]). A *circle* represents a binary operation that "combines" two values, in this case it is a comparison operation that selects the maximum of two keys. "Bookkeeping" operations fit in around the reads and writes. If a *read* line comes into P_x from the column of P_y, that means that P_x reads from M[y], since that is where P_y wrote. Figure 6 shows the activity of all the processors

for our example. The shaded areas of Figure 6 correspond to computations shown in Figure 5 that actually affect the final answer.

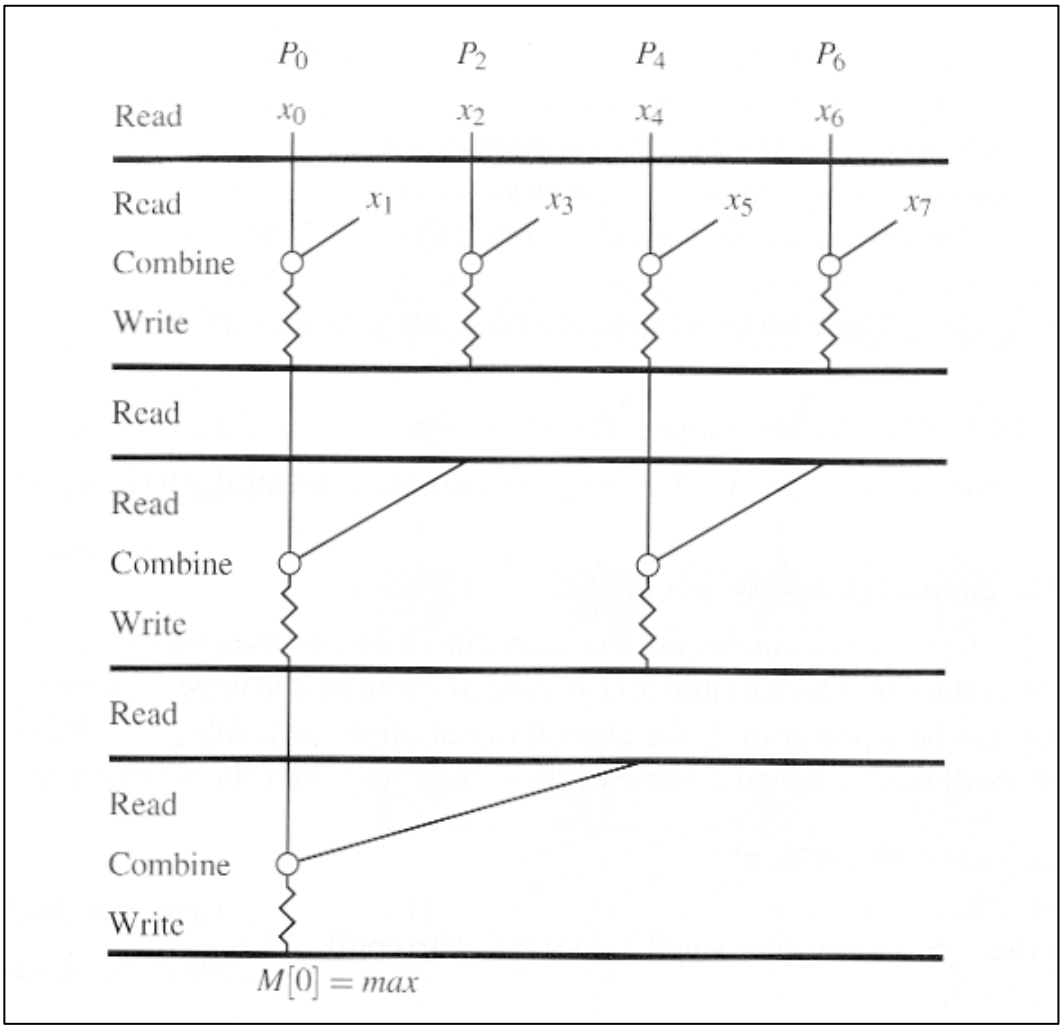

Figure 5 – A parallel tournament. Write steps are not shown for cycles in which no processor writes.

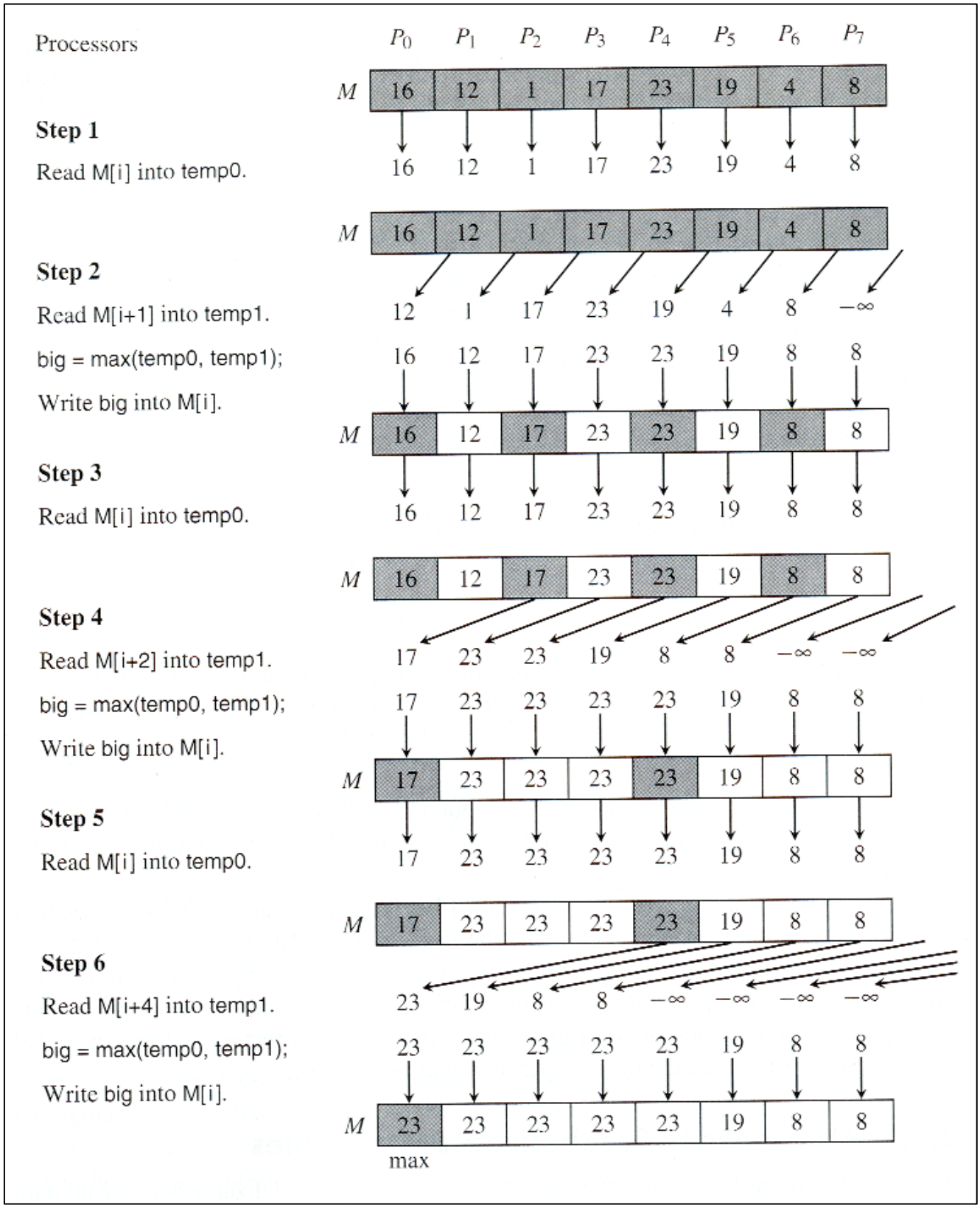

Figure 6 – A parallel tournament showing the activity of all the processors.

Parallel Tournament Algorithm for Finding Maximum Key of *n* Keys

Input: Keys x[0], x[1], …, x[n-1], initially in memory cells M[0], M[1], …, M[n-1]. An integer *n*.

Output: The largest key will be left in M[0].

Comment: Each processor carries out the algorithm using its own index number (pid) for a unique offset into *M.* The variable *incr* is used to compute the upper cell number to read. Since *n* may not be a power of 2, the algorithm initializes cells M[n], …, M[2*n-1] with $-\infty$ (some small value), because some of these cells will enter the tournament.

```
parallel_tournament_max_key(M, n)
   int  incr;
        write −∞ into M[n+pid];
        incr = 1;
        while (incr < n)
        {     key big, temp0, temp1;
              read M[pid] into temp0;
              read M[pid+incr] into temp1;
              big = max(temp0, temp1);
              write big into M[pid];
              incr =   2 * incr;
        }
```