

Parallel Scheduling Algorithms

ELIEZER DEKEL and SARTAJ SAHNI

University of Minnesota, Minneapolis, Minnesota

(Received March 1981; accepted April 1982)

Parallel algorithms are given for scheduling problems such as scheduling to minimize the number of tardy jobs, job sequencing with deadlines, scheduling to minimize earliness and tardiness penalties, channel assignment, and minimizing the mean finish time. The shared memory model of parallel computers is used to obtain fast algorithms.

WITH THE CONTINUING dramatic decline in the cost of hardware, it is becoming feasible to build economical computers with thousands of processors. In fact, Batcher [1979] describes a computer (MPP) with 16,384 processors that is currently being built for NASA. In coming years, one can expect to see computers with a hundred thousand or even a million processing elements. This expectation has motivated the study of parallel algorithms.

Since the complexity of a parallel algorithm depends on the architecture of the parallel computer on which it is run, it is necessary to keep the architecture in mind when designing the algorithm. Several parallel architectures have been proposed and studied. In this paper, we deal directly with only the single instruction stream, multiple data stream (SIMD) model. SIMD computers have the following characteristics.

1. They consist of p processing elements (PEs). The PEs are indexed $0, 1, \dots, p-1$ and an individual PE may be referenced as in $PE(i)$. Each PE is capable of performing the standard arithmetic and logical operations. In addition, each PE knows its index.
2. Each PE has some local memory.
3. The PEs are synchronized and operate under the control of a single instruction stream.
4. An enable/disable mask can be used to select a subset of the PEs that is to perform an instruction. Only the enabled PEs will perform the instruction. The remaining PEs will be idle. All enabled PEs execute the same instruction (though using possibly different data). The set of enabled PEs can change from instruction to instruction.

Subject classification: 584 parallel scheduling algorithms.

To see how an SIMD computer may be used for parallel computing, consider the problem of adding together the 8 numbers a_0, \dots, a_7 on an SIMD computer with 4 PEs. This summation can be accomplished using the scheme described in Figure 1.

The number below each node in the tree of Figure 1 is a PE index. During Step 1 of the parallel algorithm, all four PEs are active (or enabled). Each executes an add instruction during this step. PEs 0, 1, 2, and 3, respectively, compute $a_0 + a_1$, $a_2 + a_3$, $a_4 + a_5$, and $a_6 + a_7$. In the next step, PEs 0 and 1 are active while PEs 2 and 3 are idle. In this step, PEs 0 and 1, respectively, compute $\sum_0^3 a_i$ and $\sum_4^7 a_i$. PE 0, for example, computes $\sum_0^3 a_i$ by adding together the sums computed in the first step by PEs 0 and 1. Finally, in Step 3, PE 0 uses the results computed in Step 2 by PEs 0 and 1 to obtain $\sum_0^7 a_i$. During this step, PEs 1, 2, and 3 are idle.

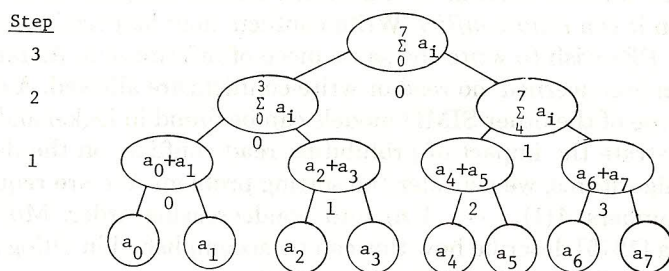


Figure 1. Parallel computation of $\sum_0^7 a_i$.

Note that if only 1 PE were available, then 7 steps would be needed to accomplish this summation. One should also note that the scheme just described can be generalized to add n numbers in $O(\log n)$ time using $n/2$ PEs. Actually, it is possible to add n numbers in $O(\log n)$ time using only $(n/\log n)$ PEs (see Savage [1978] or Dekel and Sahni [1981]).

Throughout this paper, we shall deal with only the SIMD model explicitly. Our techniques and algorithms readily adapt to the other models (e.g., multiple instruction stream multiple data stream (MIMD) and data flow models). This observation follows directly from the definition of these other models. An MIMD computer, for example, differs from an SIMD computer in that the PEs are asynchronous and operate under the control of individual instruction streams. This makes it possible for different PEs to execute different instructions at any given time (i.e., one PE can perform an add while another is performing a subtract, etc.). One readily sees that an SIMD algorithm can be run on an MIMD computer by simply replicating the single instruction stream over the available PEs and synchronizing at the end of each instruction.

In our earlier discussion of the parallel summation algorithm, we ignored such details as how PE 0 in Step 2 obtained the result of Step 1

from PE 1. In many SIMD models, the time required to communicate data from PE to PE often dominates the overall complexity of the algorithm. Several interprocessor communication models for SIMD computers have been proposed in the literature. Siegel [1979] summarizes some popular communication models.

The communication overhead of an algorithm varies from one communication model to another. To simplify the discussion, we deal only with the shared memory model (SMM) in this paper. This model has no communication delay. In a shared memory computer, there is a large common memory that is shared by all the PEs. It is assumed that any PE can access any word of this common memory in $O(1)$ time. When two or more PEs access the same word simultaneously, we say that a *conflict* has occurred. If all the PEs (at least two) that simultaneously access the same word wish to write in it, it is called a *write conflict*. If all wish to read, then it is a *read conflict*. Write conflicts may be permitted so long as all the PEs wish to write the same piece of information. As far as our discussion is concerned, no read or write conflicts are allowed. A description of some of the other SIMD models can be found in Dekel and Sahni.

To illustrate the impact of prohibiting read conflicts on the design of parallel algorithms, we consider the sorting problem. We are required to sort n numbers $A(1), \dots, A(n)$ into nondecreasing order. Muller and Preparata [1975] describe how this can be accomplished in $O(\log n)$ time using n^2 PEs. The steps involved are as follows.

1. Set $C(i, p) = 1$ if either of the following is true:
 - (a) $p \leq i$ and $A(p) \leq A(i)$
 - (b) $p > i$ and $A(p) < A(i)$
 Set $C(i, p) = 0$ otherwise. $1 \leq i \leq n, 1 \leq p \leq n$.
2. Compute $R(i) = \sum_{p=1}^n C(i, p)$, $1 \leq i \leq n$.

Note that $R(i)$ gives the position $A(i)$ is to occupy in the sorted sequence. Once $R(i)$ is known, we simply move $A(i)$ to position $R(i)$.

Performing Step 1 in parallel introduces many read conflicts. For example, $A(1)$ is needed in the computation of $C(1, p)$ and $C(p, 1)$, $1 \leq p \leq n$. To avoid read conflicts, we must first make $2n - 1$ copies of each of the $A(i)$ s. Once this has been done, each of the available n^2 PEs can compute a different $C(i, p)$ using individual copies of $A(i)$ and $A(p)$.

$2n$ copies of $A(i)$ can be made in $O(\log n)$ time using n PEs as shown in Figure 2. Hence, using n^2 PEs, $2n$ copies of each of the $A(i)$ s can be obtained in $O(\log n)$ time. Following this, the n^2 values $C(i, p)$, $1 \leq i \leq n$, $1 \leq p \leq n$ can be computed in $O(1)$ time. Finally, all the $R(i)$ s can be obtained in parallel in $O(\log n)$ time by using $n/2$ PEs to compute each of the $R(i)$ s. The overall complexity of the parallel sorting algorithm is $O(\log n)$.

