RESEARCH ARTICLE OPEN ACCESS

An Investigation about the Models of Parallel Computation

Mehrdad Hashemi[1], Shabnam Ahari[2]
[1]Department of applied mathematics and cybernetic, Baku State University, Baku, Azerbaijan
[2]Department of computer engineering, Islamic Azad university, Ahar, Iran

Abstract
In this paper we discuss on a basic facts about parallel processing. Suppose the fastest sequential algorithm for doing a computation with parameter n has execution time of T(n). Then the fastest parallel algorithm with m processors has execution time >= T(n)/m. If you could find a faster parallel algorithm, you could execute it sequentially by having a sequential computer simulate parallelism and get a faster sequential algorithm. This would contradict the fact that the given sequential algorithm is the fastest possible. We are making the assumption that the cost of simulating parallel algorithms by sequential ones is negligible.
Keywords: Parallel, computation, models, complexity.

I. INTRODUCTION

A PRAM [1] computer follows the exclusion read exclusion write (EREW) scheme of memory access if, in one program step, each memory location can be written or read by at most a single processor. It isn't hard to see that it is optimal in the sense that it will always take at least *n* steps to sort *n* numbers on that computer. For instance, some numbers might start out *n* – 1 positions away from their final destination in the sorted sequence and they can only move one position per program step. On the other hand it turns out that the PRAM-EREW computer described above can sort *n* numbers in *O*(lg2 *n*) program steps using an old algorithm due to Batcher. The difference is that even if only one processor can access one memory location at a time it is very significant that all processors can access all of the available memory in a single program step.

A comparator is a type of device (a computer-circuit, for instance) with two inputs and two outputs:

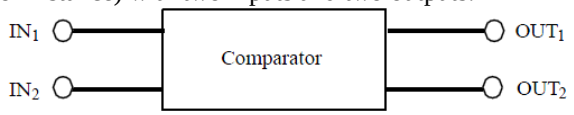

figure 1. A sample of *comparator*

Such that:
• OUT1 = min(IN1, IN2)
• OUT2 = max(IN1, IN2)

The standard notation for a comparator (when it is part of a larger network) is the more compact diagram:

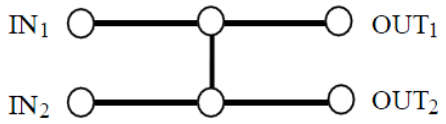

Figure 2. The standard notation for a comparator

A sorting network is a comparator network that has the additional property:

The data that appears at the output vertices is the result of sorting the data that was at the input vertices.

A merging network is defined to be a comparator network with the property that, if we subdivide the inputs into two subsets of equal sizes, and insert sorted data into each of these subsets, the output is the result of merging the input-sequences together. If a comparator network correctly sorts all input-sequences drawn from the set {0, 1}, then it correctly sorts any input-sequence of numbers, so that it constitutes a sorting network. Similarly, if a comparator-network whose inputs are subdivided into two equal sets correctly merges all pairs of 0-1-sequences, then it correctly merges all pairs of number sequences.

Suppose that a sequence of numbers will be called bitonic [2] if either of the following two conditions is satisfied:
• It starts out being monotonically increasing up to some point and then becomes monotonically decreasing.
• It starts out being monotonically decreasing up to some point and then becomes monotonically increasing.

A sequence of 0's and 1's will be called clean if it consists entirely of 0's or entirely of 1's.
For instance the sequence {4, 3, 2, 1, 3, 5, 7} is bitonic. We will present an algorithm that correctly sorts all bitonic sequences. This will turn out to imply an efficient algorithm for merging all pairs of sorted sequences, and then, an associated algorithm for sorting all sequences.

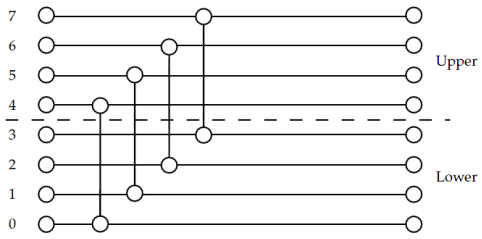

Figure 3. A bitonic halver of size 8

Given a bitonic sequence of size $\{a_0, \ldots, a_{n-1}\}$, where $n = 2m$, a *bitonic halver* is a comparator network that performs the following sequence of compare-exchange operations:

for i Ã 0 **to** $m - 1$ **do in parallel**
  if ($a_i < a_{i+m}$) **then swap**(a_i, a_{i+m})
end for

Note that a bitonic halver performs some limited sorting of its input.

II. RELATIONS BETWEEN PRAM MODELS

In this section we will use the sorting algorithm to compare several variations on the PRAM models of computation. We begin by describing two models that appear to be substantially stronger than the EREW model:

CREW — Concurrent Read, Exclusive Write [3]. In this case any number of processors can read from a memory location in one program step, but at most one processor can write to a location at a time. In some sense this model is the one that is most commonly used in the development of algorithms.

CRCW — Concurrent Read, Concurrent Write [3]. In this case any number of processors can read from or write to a common memory location in one program step. The outcome of a concurrent write operation depends on the particular model of computation being used (i.e. this case breaks up into a number of sub-cases).

It is a somewhat surprising result, due to Vishkin [4] that these models can be effectively simulated by the EREW model. The original statement is as follows:

If an algorithm on the CRCW model of memory access executes in a time units using b processors then it can be simulated on the EREW model using O(a lg2 n) -time and b processors. The RAM must be increased by a factor of O(b). This theorem uses the Batcher sorting algorithm in an essential way. If we substitute the (equally usable) EREW version of the Cole sorting algorithm, we get the following theorem:

Improved Vishkin Simulation Theorem If an algorithm on the CRCW model of memory access executes in a time units using b processors then it can be simulated on the EREW model using O(a lg n) -time and b processors. The RAM must be increased by a factor of O(b). Incidentally, we are assuming the SIMD model of program control. The algorithm works by simulating the read and write operations in a single program step of the CRCW machine.

III. COMPLEXITY CLASSES AND THE PARALLEL PROCESSING THESIS

In this section we will be concerned with various theoretical issues connected with parallel processing. We will study the question of what calculations can be efficiently done in parallel and in what sense. We present the so-called Parallel Processing Thesis of Fortune and Wyllie [5]. It essentially shows that execution-time on a parallel computer corresponds in some sense to space on a sequential computer. The arguments used by Fortune and Wyllie also give some insight into why the execution time of many parallel algorithms is a power of a logarithm of the complexity of the problem. One of the most interesting theoretical questions that arise in this field is whether there exist inherently sequential problems. These are essentially computations for which it is impossible to find parallel algorithms that are substantially faster than the fastest sequential algorithms. This is a subtle question, because there are many problems that appear to be inherently sequential at first glance but have fast parallel algorithms. In many cases the fast parallel algorithms approach the problem from a completely different angle than the preferred sequential algorithms. One of the most glaring examples of this is the problem of matrix inversion, where:

1. The fastest sequential algorithm (i.e., a form of Gaussian Elimination) only lends itself to a limited amount of parallelization.
2. The fastest parallel algorithm would be extremely bad from a sequential point of view.

This should not be too surprising, in many cases the fastest sequential algorithms are the ones that reduce the amount of parallelism in the computations to a minimum. First it is necessary to make precise what we mean by a parallel algorithm being substantially faster than the corresponding sequential algorithm. Here are some of the algorithms that have been considered so far:

1. Forming cumulative sums of n numbers. The sequential algorithm has an execution time of O(n). The parallel algorithm has an execution time of O(log n) using O(n) processors;
2. Sorting n numbers by performing comparisons. The best sequential algorithms have an asymptotic execution time of O(n log n). The best parallel algorithms have asymptotic execution times of O(log n) using O(n) processors;
3. Inversion of an n × n non-sparse matrix. The best sequential algorithms use Gaussian Elimination and have an execution time of O(n3). The asymptotically fastest known parallel algorithms have an execution time of O(lg2 n) using (n× 2.376) processors.

The general pattern that emerges is:

• We have a sequential algorithm that executes in an amount of time that is bounded by a polynomial function of the input-size. The class of such problems is denoted P;
• We have parallel algorithms that execute in an amount of time that is bounded by a polynomial of the logarithm of the input-size, and use a number of processors bounded by a polynomial of the input size. The class of these problems is denoted NC; As has been remarked before, NC µ P — any algorithm for a problem in NC can be sequentially simulated in an amount of time that is bounded by a polynomial function of the original input. Our question of whether inherently sequential problems exist boils down to the question of whether there exist any problems in P \ NC, or the question of whether NC = P. As of this writing 1991 this question is still open. We will discuss some partial results in this direction. They give a natural relationship between parallel execution time and the amount of RAM required by sequential algorithms. From this we can deduce some rather weak results regarding sequential execution time.
It is first necessary to define the complexity of computations in a fairly rigorous sense. We will consider general problems equipped with:
• An encoding scheme for the input-data. This is some procedure, chosen in advance, for representing the input-data as a string in some language associated with the problem. For instance, the general sorting problem might get its input-data in a string of the form $\{a_1, a_2, \ldots\}$, where the "a_i" are bit-strings representing the numbers to be sorted.
• A complexity parameter that is proportional to the size of the input-string. For instance, depending on how one defines the sorting problem, the complexity parameter might be equal to the number of items to be sorted, or the total number of symbols required to represent these data-items. These two definitions of the sorting problem differ in significant ways. For instance if we assume that inputs are all distinct, then it requires O(n log n) symbols to represent n numbers. This is due to the fact that "log n" bits are needed to count from 0 to n − 1 so (at least) this many bits are needed to represent each number in a set of n distinct numbers. In this case, it makes a big difference whether one defines the complexity-parameter to be the number of data-items or the size (in bits) of the input. If (as is usual) we assume the all inputs to a sorting-algorithm can be represented by a bounded number of bits, then the number of input items is proportional to the actual size of the input.

REFERENCES

[1] J. Brenner, J. Keller, C. Kessler, Executing PRAM Programs on GPUs, Procedia Computer Science, Vol 9, 2012, pp. 1799-1806.
[2] http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/oddn.htm
[3] M. Snir, On parallel searching. Society for Industrial and Applied Mathematics. SIAM J. COMPUT., Vol. 14, No. 3, August 1985, pp. 688-708.
[4] Uzi Vishkin, Implementation of simultaneous memory access in models that forbid it. J. Algorithms 4 1983, pp. 45–50.
[5] S. Fortune and J. Wyllie. Parallelism in random access machines. ACM Symposium on the Theory of Computing, vol. 10, 1978, pp. 114–118.
[6] F. Yu, K. Ko, On parallel complexity of analytic functions. Theoretical Computer Science, Volumes 489–490, 10 June 2013, PP. 48-53.