Can Parallel Algorithms Enhance Serial Implementation? (Extended Abstract)

Uzi Vishkin*
University of Maryland &
Tel Aviv University

Abstract

The broad **thesis** *presented in this paper suggests that the serial emulation of a parallel algorithm has the potential advantage of running on a serial machine faster than a standard serial algorithm for the same problem. It is too early to reach definite conclusions regarding the significance of this thesis. However, using some imagination, validity of the thesis and some arguments supporting it may lead to several far-reaching outcomes:*
(1) Reliance on "predictability of reference" in the design of computer systems will increase.
(2) Parallel algorithms will be taught as part of the standard computer science and engineering undergraduate curriculum irrespective of whether (or when) parallel processing will become ubiquitous in the general-purpose computing world.
(3) A strategic agenda for high-performance parallel computing: A multi-stage agenda, which in no stage compromises user-friendliness of the programmer's model, and thereby potentially alleviates the so-called "parallel software crisis", is discussed.
Stimulating a debate is one goal of our presentation.

1 Introduction

Given a problem, we want to find a serial algorithm for it whose actual running time is fast. On presently available computer organizations, only a small part of the computer memory is fast and the rest is slower (for instance, main memory is slower than the cache and disks are slower than the main memory; more levels are also possible in this hierarchy). Therefore, efficient computer implementation of an algorithm that needs large memory space may depend on how often the processing element needs to suspend its operation (or more precisely, the total time of such suspensions), waiting for data to be fetched from slow memories.
The main concrete observation of the paper is very simple. Consider the serial emulation of a parallel algorithm. **Just before the serial emulation of a round of the parallel algorithm begins, the whole list of memory addresses needed during this round is readily available; and, we can start fetching all these addresses from secondary memories at this time,** so that these data will be ready at the fast memories when requested by the CPU.

*Partially supported by NSF grants CCR-8906949 and 9111348.

This suggests that a parallel algorithm that is implemented properly on a serial machine may pay a smaller penalty for the use of large memory and thus run faster than a comparable serial algorithm.

Our basic argument is that the following idea deserves careful engineering prototyping: **use computer systems (or update their design as needed) so that an implementation of such an algorithm will result in better serial running time.** Specifically:

1. Given a list of future memory requests, the memory management system should enable them to be prefetched (i.e., get the required data into the cache) efficiently. In other words, the memory management system should be able to take advantage of "predictability of reference", in the same spirit as memory management systems take advantage of "locality of reference".

2. Given a computer program describing a parallel algorithm, the compiler should be able to identify all instructions that can be performed in parallel, and thereby extract a list of memory requests for future need, as per item 1 above.

Programming languages which enable to express parallelism should be available. Henceforth, we assume that such a programming language is available, since designing such a language is clearly doable. Actually, some languages already exist: FORTRAN 90 [MR] and SETL [BCHSZ] are two examples.

For the user, this would mean the following. 1. Given a problem, and an algorithm for it, try to redesign the algorithm into an efficient parallel one. The goal is that the redesigned algorithm will run in the smallest possible number of parallel rounds without increasing its total number of operations. Often a sufficiently parallel algorithm can be derived in a straightforward way. We remark that for the applications described in this paper even a moderate level of parallelism can already be helpful; of course, additional parallelism may be even better. For example, a serial emulation of a parallel algorithm whose parallel running time is ten times faster than its serial counterpart may, theoretically, already yield better performance.
2. If it is impossible to derive a parallel algorithm from a serial one, design a new parallel algorithm for the problem. We do not expect the possible need for development of a new parallel algorithm to be a significant disadvantage:

Parallel algorithmics has undergone major developments in the last few years (for brevity, we reference here only [Ja] and [Vi94]). So parallel algorithms and techniques for many problems already exist. In any case, the development of a new parallel algorithm is of independent interest, since a general long term trend in computer science is in the direction of parallelism and a new parallel algorithm will be added to a long lasting knowledge base. Recall that *optimal* parallel algorithms are those where the total number of operations is the same as the serial time complexity of the problem being considered. Typically, researchers on efficient parallel algorithms have been interested in designing optimal parallel algorithms (even if their running time could not be upper bounded by $O(\log^k n)$, where n is the input length and k is a constant, as demonstrated in a 1977 thesis by Eckstein, [RC], [SV], [VS], and [KRS]). A notion of performance which is similar to the one of optimal algorithms, but not identical, is needed in the context of this paper. A serial emulation of a parallel algorithm is compared with the fastest serial algorithm for the same problem, where the serial emulation may have a slightly higher running time. How much higher depends on how advantageous the implementation of a parallel algorithm (as advocated in the present paper) is going to be, *taking into account constant factors*. Also, getting the absolute fastest possible parallel running time is not as significant, since beyond a certain level of parallelism the incremental advantage of additional parallelism is very small.
"Thinking in parallel" is an "alien culture" and therefore, if needed, has to be taught in school. So, an interesting outcome of our approach is that parallel algorithms would have to be taught as part of the standard computer science and engineering undergraduate curriculum irrespective of whether (or when) parallel processing will become ubiquitous in the general-purpose computing world. This curriculum update should occur in anticipation of future changes since one of the objectives of such a curriculum is to prepare young graduates for a life-time professional career. These thoughts led the author to work on [Vi94].

Prefetching in the context of hardware is discussed in [Jo] and references therein. A recent paper that advocates the use of software prefetching is [CKP]; a helpful heuristic that guides compilers to identify cases where prefetching is possible is suggested. This heuristic will not be able to identify opportunities for prefetching that are described in this paper. However, the literature seems to emphasize another approach for getting more efficient algorithms in cases where large memories are involved. As a representative of this literature, we select [ACF] which suggests revising algorithms to work more efficiently on a particular model of memory organization. Such an approach essentially requires programming algorithms in a far less user-friendly language, a trend which is not considered desirable in modern computer science. Comparison with these works is not entirely appropriate since [ACF], for instance, assumes a relatively low upper bound on the throughput of secondary memories, where throughput is defined as the rate at which these memories can transfer data. In Section 6, we observe that upper bounds on throughput have led to parallel programming languages which are not sufficiently user-friendly. While some Cray machines, for instance, enable alleviating the throughput problem in the context of serial processing (or up to 8 processors), one of the main conclusions of this paper is that **designers of (massively) parallel computers will apparently have to improve the throughput of their machines if they want to efficiently support a user-friendly parallel programming language**.

We also note that our approach circumvents the need for on-line paging strategies as in [FKLMSY].

Our presentation proceeds as follows. Section 2.1 presents the main problem addressed in this paper. Section 2.2 outlines a measurement model, which enables quantifying the quality of a solution for the problem. Our broad thesis, and the concrete suggestion for the main problem, are described in Section 3. Section 4 discusses our suggestion with respect to concrete examples. Section 5 extends Section 3. Section 6 discusses the relevance of this work for parallel computation. Section 7 concludes with yet another broad perspective on our thesis.

2 The Model

2.1 The essence of the problem

A serial algorithm that might be particularly hard to implement efficiently on a serial machine is one in which the memory access in each step depends on the previous step. We will use several examples in this paper. Our lead example in the present and the next section is the *list-ranking* problem. Later we use other examples. The input for the list-ranking problem consists of several disjoint linked lists, whose total number of elements is denoted n - these lists are stored in some arbitrary order in an array of size n; each element has a pointer to its successor in the list or is the last in its list. Only the first element in each list does not have a predecessor. The problem is to find for each element the length of the path, counting pointers, from it to the end of its list.

A trivial serial list-ranking algorithm will perform a serial traversal of each linked list. If n is large then at any given time most of the list will be stored on secondary memory (on a typical computer). Therefore, each advance to a new element of the list may entail issuing a memory request to secondary memory. Thus, list-ranking may take a considerable time.

2.2 A nonincreasing-incremental-cost model

A typical machine has local registers, cache and some secondary storage (e.g., main memory and disks). Consider a request issued by the processor for an address. Looking for simplifications that will not hide the principles, we define the *cost of the request* as the number of lost cycles in which the processor had to suspend its operations and wait until the contents of the address reaches the registers. Specifically: (i) If the address lies in the cache, the cost is unit time. (ii) If the address lies in one of the secondary memories, the cost is A time, for some $A > 1$. *Explanation of assumption (ii).* Actually, the request will be sent towards the secondary memory containing it, then, first the address will be transferred from the secondary memory to the cache as part of a "fetch unit", and second, the desired address will be picked and forwarded to the registers. In order to maintain this paper at the level of principles, we will consistently: (1) ignore the time for forwarding a request to the memory containing it, but will only discuss the path of the address from the memory towards the registers; also (2) we assume the same cost formula (e.g., same A) for all secondary memories. Generalization to hierarchies of secondary storage, with different retrieval times appears to not to be too difficult. We use the term *fetch unit* as a generic name for a memory page, or a memory line, containing the desired element. For common organizations of main memory a typical value for A will be 5, [HP]. *In case the secondary memory is on a disk these values are drastically higher (by [HP] one million (!) is a typical value).* [1]

The list-ranking example (revisited). Under these assumptions, the implementation of the serial list-ranking algorithm takes at least nA time in the worst case, since there are at least n accesses to memory and all of them may be to secondary memory.

Consider now r address requests issued by the processor in r successive clock cycles. Assume that all these addresses lie in secondary memory. *Their COST* is upper bounded by $F(r)$, where F is a monotonically non-decreasing function which satisfies the following: (1) $F(1) = A$, where A is as before; (2) $F(i+1) - F(i) \leq F(i) - F(i-1)$ for $i > 1$. One useful example is $F(r) = A + (r-1)B$, where $B < A$. We will use this example below for illustrative purposes and refer to it as the *linear model* for F.

The reason for the name "nonincreasing incremental cost" is that the value A in the cost formula represents some set-up cost, regardless whether one or more requests are to be satisfied, and then for every $i > 1$ requests, the incremental cost of satisfying one more (i.e., an $(i+1)st$) request does not increase. In order to obtain good results for our suggestions, we would like the series $F(i)/i$ (representing the average cost of a request, where i requests are made) to converge to a small number. The next section mentions briefly several implementation possibilities, and discusses how they would affect the cost function F.

There is one central item which is missing from our model: *given an algorithm, how to identify its memory requests?* For this, we must have a clear idea as to what should be defined as a *single* memory access. Unfortunately, what exactly is to be considered a single operation is an unresolved issue in theoretical computer science. (This reason, among others, prevents the development of "robust theory" for some serious computer science problems where "only constant factors are involved".) The following guidelines help us to partially circumvent this "single operation" difficulty. In counting memory accesses below we are guided by the level of specification of our own description, which resembles high-level programming language, such as PASCAL. While our count may be insufficient for anticipating the exact number of memory requests, it may be sufficient for the more modest purpose of a crude comparison of two algorithms which are quite similar (as is actually done later). Since the details of our count offer little insight, we defer their presentation to the analyses of the algorithms in a later section.

[1] On a typical high performance computer the following access times are not out of line: registers - 10 ns, cache access 20-60 ns, memory access 80-200 ns, and disk access 20 ms.

3 The New Suggestion

Take *any* parallel algorithm for some problem; call it $P-ALG$. Given some input, denote by T the number of rounds of the algorithm and by w_i the number of operations performed in round i, $1 \leq i \leq T$, and suppose that each operation involves at most one memory access. Let $w = \sum_{i=1}^{T} w_i$. Consider a serial emulation of $P-ALG$ as follows. The emulation handles one round at a time. It proceeds by successively emulating the ("virtual") rounds of the parallel algorithm. At each round i, the w_i operations are emulated one at a time.

MAIN OBSERVATION: *When the emulation of round i begins, the whole list of memory addresses needed in round i is readily available.*

Before proceeding to an explanation of the main observation, we make a short detour and restate the thesis of this paper in a somewhat **broader** form.

THESIS: **A parallel program is potentially an invaluable asset for efficient implementation purposes. Since a parallel program allows concurrent tasks to be processed in any order, or even simultaneously, it also gives a lot more freedom to lower level implementations: cache, prefetch, pipelining, compiler, code rearrangement, estimators of working set and code behavior in Operating Systems etc. This flexibility is very structured, and might be more useful than information about execution that is gathered from: (1) statistical analysis of execution and heuristics; (2) statical analysis of flow diagrams; and (3) ad-hoc optimizations in run-time. Further, the parallel paradigm allows more than just predicting future memory requests. It allows the lower levels to alter the execution so that it will be best suited to the internal organization of the lower level. For instance, loading several independent tasks into the registers (which is an instance of the general prefetch idea), may result in better utilization of the processing unit.**

These examples are meant to encourage exploring opportunities for taking advantage of parallel programs, well beyond what is suggested in this paper.

Explanation of the main observation. Similar to the way PRAM algorithms are specified, the w_i instructions of the operations to be performed in round i, have serial numbers from 1 to w_i. Our experience with parallel algorithms is that in many of them all processors perform essentially the same program, and therefore there is a very concise way for representing parallel programs. However, in the most general case, we may need to store these instructions (including the address that each instruction needs) in some designated space. If the cache is not big enough we may need to store them in some designated part of the secondary memory, and have in the cache an "instruction area" which is devoted for instructions that are brought into the cache. We assume that the instructions are in place (in the secondary memory) by the time round $i-1$ ends. The emulation of round i begins by *instruction prefetch*. The size of the instruction area in the cache dictates the size of the fetch unit containing instructions (since we want them to be large to minimize transfer time). Instruction fetch units start migrating into the instruction area of the cache and continue as long as it is not full. This suggests that round i will be emulated in several subrounds; each subround has the same number, say p, of instructions. We refer to the emulation of each subround as a "time window". Soon after the first instruction fetch unit reaches the cache, requests for all the addresses that will be needed in the next "time window" start to go out. These data (i.e., contents of these addresses) will be ready at the cache when requested by the CPU. For simplicity, the corollary below assumes that $w_i \leq p$ (this simplifies things since if $w_i \leq p$, then the instruction area never fills up and there is only one subround). It also does not charge for instruction prefetching, since this should overlap for the most part with data prefetch.

MAIN COROLLARY: Satisfying the memory requests for round i of algorithm $P-ALG$ will take $F(w_i)$ time, and for the whole algorithm, $\sum_{i=1}^{T} F(w_i)$ time. In the linear model for F, this will be $A + (w_i - 1)B$ time for round i and $AT + (w-T)B$ time for the whole algorithm. (We do not discuss possibilities for overlapping parallel rounds or subrounds in order to maintain this paper at the principles level.)

The list-ranking example (revisited). Pick an efficient parallel algorithm for the list-ranking problem that runs in $O(\log n)$ rounds using a total of $O(n)$ operations, and consider its serial emulation. If n is large then at any given time most of the list is likely to be stored in secondary memory, and satisfying the memory requests will take time proportional to $(\log n)A + nB$ (in the linear model for F), as opposed to time proportional to nA in the serial algorithm. So, this comparison comes down to the relative values of A and B, and to the constant factors in the bound for the list-ranking algorithms involved. This is discussed in a later section.

The next issue is what values we can expect from the cost function F, and what can be done to improve it. In particular, the CHALLENGE is to get the value of $F(i)/i$, as i increases, to be as close to 1 (i.e., clock cycle time) as possible. However, even getting the value of $F(i)/i$ to be considerably smaller than $F(1)$ will be significant. We discuss several (semi) specific suggestions, and hope to stimulate the reader to think of others.

1. *Distributing addresses at random.* Assume that the secondary memory consists of several (say m) separate banks, called memory modules, each module connected by a separate bus to a separate part of the cache. (Think about a "star graph" with the cache as the center node and each memory module connected to it by an edge.) The intuitive idea is to distribute the addresses at random initially among the m memory modules hoping that, with high probability, the w_i memory requests will be partitioned nearly equally among the memory modules. This random distribution idea is implemented by randomly selecting a hash function from some set of easy-to-evaluate hash functions, as in [MV]. If $w_i \gg m$ then with high probability we will have no more than say $2w_i/m$ requests from any module (for a formal discussion see Fact 2 in [DM] which is based on an analysis by [KRS]). Suppose the random distribution idea is applied and the values of w_i and m imply the $2w_i/m$ upper bound; with a slight abuse of the linear model, we apply it for evaluating the transmission time from a module to its cache. Such transmission time is upper bounded by $A+2w_iB/m$. With some assumptions on A and B this can be made not to exceed w_i. This may mean that, subject to proper assumptions (on m and w_i, among others), the value of B in the linear model for F may be 1, reflecting the rate at which data is supplied by the caches; this is very attractive. We also note that these assumptions may be considerably *relaxed.* Since busses tend to have considerable bandwidth, a single cache, a single bus and (as before) m memory modules may be sufficient. Also, it appears that the bus transfer time is unlikely to be a bottleneck, because the fetch unit size (or, in other words additional data that have to be transmitted along with a requested datum) is becoming smaller (see [HP]), permitting a more effective use of the bus bandwidth. At the same time, the present approach copes well with the fact that bus access time is less predictable. Finally, we note that in case the secondary memory is main memory, this whole suggestion falls within the area of memory banks and interleaved memories (see the example concerning a Cray machine in Section 6 and [HP]). Next, we mention a possibility (which has limited promise by available technology because of throughput problems): *it is possible to use a considerable number of disks for memory modules.* (It appears that technology advances in the right direction, since it already enables effective use of over a hundred disks. For instance, a configuration of 9 Mass Storage Cards with 84 disk drives on each, for a total of 756 disk drives, is theoretically permitted in the Encore Multimax computer.) This last option can be combined with direct memory access, as described later. Another intriguing possibility is that of using, instead of disks, the fast memories of other computers that are connected by a local area network (LAN), such as an Ethernet, to the computer on which the algorithm under consideration is being run. By [St], a computer on an Ethernet can send a block of over $1K$ bytes within roughly the same time (about 1ms) as sending a single byte; a more up-to-date technology allows several thousands bytes in a block. While the latency may not improve in the near future, the size of the block is likely to increase without changing the transmission time, and

as discussed later, the current paper may provide an additional incentive for further increase in block size (without decrease in transmission time).
2. *Use Direct Memory Access.* See [Vi93].
3. *Other kinds of main memory.* See [Vi93].
4. *Locality of reference.* Following [HP], we distinguish two kinds of locality of reference properties. (i) *Locality in time:* If an item is referenced, it will tend to be referenced again soon; and (ii) *Locality in space:* If an item is referenced, nearby items tend to be referenced soon. The design of many computer systems is based on the assumption that the execution of most computer programs satisfies these properties; evaluating the validity of this assumption is beyond the scope of this paper. Throughout this paper we elaborate only on a case where locality in space occurs, since as explained in [Vi93] we see no inherent reason for why locality in space should be satisfied in a "typical" serial program, but not in a serial emulation of a "typical" parallel program.
It is interesting to mention that while the tendency of many serial algorithms to demonstrate some locality-of-reference has been experimentally demonstrated, our suggestion for using parallel algorithm for faster performance on suitably designed machines is based on proven (!) predictability-of-reference.

4 Two Concrete Examples

We discuss two examples. For the first, a pencil-and-paper kind of analysis is being used, while for the second, experimental results are reported.

4.1 First example: Depth-first search of graphs

Our lead example in this subsection is the the problem of depth-first search traversal of graphs. A serial and a parallel algorithm will be presented. *The main goal of the example is to show that (for dense graphs) the parallel algorithm can result in faster serial implementation.*

The reader may wonder why we selected this problem. We offer several reasons: (1) its simplicity; (2) in the standard serial algorithm for depth-first search memory access in each step depends on the previous step, as per the subsection on "the essence of the problem" above; and (3) its selection and analysis reinforce the claim that a parallel algorithm need not achieve poly-logarithmic time to be useful, and the constants in the running time analysis play an important role. We start by defining the problem.
Input: An undirected graph $G(V,E)$. There is a vertex array of length n, where $V = \{1,...,n\}$ and an edge-array of length $2m$, where $m = |E|$. Each vertex v has a *record* (using PASCAL terminology, or *structure* using C terminology) $S(v)$, where $S(v).1$ is a pointer to the edge-array, and the $deg(v)$ edges that are incident on v occupy locations $S(v).1,...,S(v).1 + deg(v) - 1$ in the array (we define $S(v).2 = deg(v)$). In other words, an edge connecting vertices u and v has two copies in the edge-array: (i) $< u,v >$ in the list of edges incident on vertex u, and (ii) $< v,u >$ in the list of edges incident on vertex v. We assume that each of these copies has a pointer to the other copy, and that the graph has no parallel edges.

The depth-first search (DFS) numbering problem: Starting from vertex 1, number the vertices in the connected component of 1 according to the order in which they are visited in a depth-first traversal of G.

We first give a standard serial algorithm for this problem. Define the following recursive routine:

$DFS(v)$
$dfnumber \leftarrow dfnumber + 1;$
$dfnumber(v) \leftarrow dfnumber$
Go to location $S(v).1$ in the edge-array
for every edge $< v,u >$ *do*
if $dfnumber(u) = 0$
then $DFS(u)$

The algorithm follows by initializing $dfnumber(v) \leftarrow 0$, for every vertex v, and calling $DFS(1)$.

MEMORY ACCESS analysis. For comparison of the running time of this algorithm with the parallel algorithm that follows, we count only memory accesses and refrain from counting other operations. This is justified since: (1) in this paper we care only about memory accesses; and (2) the time needed for operations that do not involve memory access in each of these algorithms is dominated by memory access operations. We will use the common method of analysis where each operation (in our case, memory access) is charged to an edge or a vertex. For each vertex v, we have a $DFS(v)$ call, one $dfnumber$ increment, one assignment into $dfnumber(v)$ (we do not count this as a memory access; note that $dfnumber(v)$ was brought into memory just prior to the $DFS(v)$ call; a compiler can be designed not to return $dfnumber(v)$ to memory till the assignment into $dfnumber(v)$; see more on the issue of writes into secondary memory using the "write-back' policy under "memory consistency" in the next section), an advance to the edge list through $S(v)$ (which we count as two memory accesses) and one return from the DFS call (which we count as one memory access). The return from a DFS call may be counted as more than one memory access, since we have to get back to the same state in the calling routine. We count this as one since it will be to our disadvantage in the comparison with the parallel algorithm. Having n vertices, this amounts to a total of $3n$ memory accesses. For every copy of an edge $< v,u >$, we advance to it (a memory access), read $dfnumber(u)$ (a memory access), and check whether $dfnumber(u) \neq 0$. This gives 2 memory accesses per copy of an edge, and a total of $4m$ memory accesses for all $2m$ copies. We conclude: *the depth-first search method takes a total of* $3n + 4m$ *memory accesses* under this count.

Next, we provide a parallel variant of this algorithm. While unaware of a reference, we will not be surprised if this variant has appeared in print. A variant that uses concurrent writes by several processors to the same memory location is given in [VS], and one that uses more space is given in a 1977 University of Iowa Doctoral Thesis by D. Eckstein. For each vertex

v, we maintain a doubly-linked list of all its incident edges $< v, u >$ whose other vertex (namely u) has not yet been visited by the depth-first search traversal. This is done using two pointers: $N(< v, u >)$ (to the next edge) and $P(< v, u >)$ (to the preceding edge). The main difference with respect to the serial DFS above is that the parallel DFS never advances on an edge leading to a vertex that has already been visited. The recursive routine is:

$P - DFS(v)$
$dfnumber \leftarrow dfnumber + 1$;
$dfnumber(v) \leftarrow dfnumber$
for every edge $< v, u >$ *pardo* (i.e., *do* in parallel)
$P(N(< u, v >)) \leftarrow P(< u, v >)$;
$N(P(< u, v >)) \leftarrow N(< u, v >)$
(i.e., discard edge $< u, v >$ from the doubly-linked list of u)
for every edge $< v, u >$ on the doubly-linked list of v *do*
$P - DFS(u)$

The algorithm starts with the following initializations: (i) for every vertex v $dfnumber(v) \leftarrow 0$; and (ii) for every edge $< u, v >$ in the edge-list: (a) $P(< u, v >)$ is the edge preceding $< u, v >$ in the edge list; and (b) $N(< u, v >)$ is the edge succeeding $< u, v >$ in the edge list. Finally, the traversal is initiated by calling $P - DFS(1)$.

The only subtle point in showing that this parallel algorithm is correct is the following: the "discard" operations *never* applies to two successive edges in any of the doubly-linked lists.

Memory access analysis. The parallel DFS runs in n rounds, each corresponding to some vertex v. Each round consists of several subrounds. In each subround, several operations can be done in parallel. The following subrounds are associated with vertex v: a DFS(v) call, one *dfnumber* increment and one assignment into $dfnumber(v)$, an advance to the edge list using $S(v)$ (which values two memory accesses that have to be performed one after the other) and one return from the DFS call (which is counted as one memory access, in order to be consistent with the analysis of the serial algorithm). This provides 3 (non successive) subrounds with a single memory access in each. Having n vertices, this amounts to a total of $3n$ single subrounds each consisting of a single memory access.
Next come the subrounds in which parallel operations are performed. In parallel, for every copy of an edge $< v, u >$, we advance to it (a memory access), advance to its anti-parallel copy $< u, v >$ (a memory access), advance (simultaneously) to $P(< u, v >)$ and $N(< u, v >)$ (two memory accesses in one subround), and update them. This provides additional 3 subrounds, per vertex, with 4 memory accesses per copy of an edge, and a total of $3n$ subrounds and $8m$ operations. So, *the total number of subrounds is* $6n$ *and the total number of operations is* $3n + 8m$.
Comment. From the point of view of the theory of parallel algorithms, the asymptotic time upper bound of the parallel DFS (including all operations) is $O(m/p + n)$, which means that if the graph is not too sparse (i.e., $m \gg n$) then we have an efficient parallel algorithm for up to $O(m/n)$ processors.

Comparison of the serial DFS with the parallel DFS in the linear model for the cost function F: The serial DFS algorithm will take $(3n + 4m)A$ time, and the parallel DFS will take $6nA + (3n + 8m - 6n)B = 6nA + (8m - 3n)B$ time. We are ready to try some concrete values: (1) Let $A = 10B$ and $m = 100n$, then the parallel DFS will run nearly 5 times faster. (2) Let $A = 100B$ and $m = 1000n$, then the parallel DFS will run nearly 50 times faster.
Using an asymptotic comparison style, the comparison becomes mA memory access time for serial DFS versus $nA + mB$ memory access time in parallel DFS (assuming that $m \gg n$ and $A \gg B$). We feel that this style is less relevant than in "traditional" computer science applications of asymptotic analysis.

4.2 Second example: Suffix trees

A more exciting example is the construction of *suffix trees*, a fundamental tool in text processing. The suffix tree data-structure is probably the most useful mechanism known for finding initial similarities within a very long string of characters, and as such is strongly related to the development of molecular sequence comparison algorithms, as per the grand challenges of the U.S. High Performance Computing and Communication Program. The linear-time serial algorithms (by P. Weiner, or E.M. McCreight [Mc]), which are considered theoretically superior because of the asymptotic upper bounds on their running time, seem to suffer from the following problem: their memory access often depends on the preceding step, as in the serial depth-first search, or serial list-ranking algorithms.

Next, we review some preliminary experimental work that, using simulations, compares McCreight's serial algorithm against a serially-emulated parallel algorithm [AILSV]. Henceforth, these algorithms are referred to as Algorithm M and Algorithm P, respectively. We considered it more appropriate to base our experiment with Algorithm M on code written by others, rather than by ourselves. William Chang kindly agreed to provide us with his code for Algorithm M.

We first sum up the experimental results, then give some more information about the experiments themselves, and finally briefly discuss the relevance of the results. Since the problem of suffix tree construction is of particular interest to us where the input is large and the data structures required by an algorithm do not fit into the fast memory of a computer, our experiments used an input string of length 10^6. We assumed a fast memory whose number of pages is $2K$, each of size $1K$ bytes. We are aware that larger fast memories are available but found it more convenient to experiment with these sizes.

Using the linear model for the cost function F for comparing the two algorithms, we found that the two algorithms had similar performance where the ratio A/B was around 250.
For $A/B = 10$, Algorithm M was 14.14 times faster than algorithm P.
For $A/B = 10^2$, Algorithm M was 2.26 times faster

than algorithm P.
For $A/B = 10^3$, Algorithm P was 4.15 times faster than algorithm M.

We also used another cost model, which will be described by the function G. The function G will sum up the cost of the memory accesses of an algorithm, as follows: (i) if an address request hits the fast memory, the cost is B; (ii) if a single address request misses the fast memory and has to be fetched from the slow memory, the cost is A (so far this is similar to the linear cost model and A/B may be 10^6); (iii) consider r address requests issued by the processor in r successive clock cycles, and assume that all these addresses lie in the slow memory; then their cost is $\lceil r/C \rceil A$, where $A/C > B$.
This function G captures the possibility of forwarding up to C addresses to the fast memory for the price of forwarding one address. Additional justification for using this cost function is given later in this subsection.

For $A/B = 10^6$, and $C = 300$, Algorithm P was not slower than algorithm M.
For $C = 10^3$, Algorithm P was 2.76 times faster than algorithm M.
For $C = 10^4$, Algorithm P was 27.38 times faster than algorithm M.
For $C = 10^5$ Algorithm P was 243.93 times faster than algorithm M.

More on the experiment. Cache policies: the results reported are for LRU (the page to be removed from the fast memory is the least recently used). We tried also FIFO (first-in first-out), and Algorithm M did not have better performance than with LRU. Source of input: we used files of alphabet symbols only that were extracted from the most recent messages on bulletin boards of two electronic news groups (comp.parallel and comp.theory). We tried also files whose characters were generated by a random function and Algorithm M had slightly more fast memory misses on these randomly generated kind of input. Alphabet size was 52 (all upper and lower case letters).

The values with respect to the linear cost function F are related to performance of disks, as discussed in the previous section. The figures on disk performance justified considering the function F for values of A/B up to 10^3.

The possibility of using the fast memories of other computers which are connected by a local area network justifies the use of the second cost function G as a first approximation (note that where $C = A/B$, the functions G and the linear cost function for F describe similar costs). There will be a "master" computer which actually runs the algorithm. Having a computer on an Ethernet sending back a block with at least $1.2K$ bytes in response to memory requests sent to it by the master computer can be done within roughly the same time (about 1ms) as sending a single byte. Therefore, the total time for handling 300 words (of 4 bytes each) is only slightly higher than for handling a single word. This makes the preliminary results reported for $C = 300$ appear possible by available computers and local area networks. While it is not clear whether latency will decrease in the near future the size of the block is likely to increase (possibly, without changing the transmission time) and the current paper may give additional motivation for further improvements in this direction. Our experimental results actually demonstrate that such development may lead to an implementation of Algorithm P which is **several orders of magnitude faster** than that of Algorithm M.

4.3 Additional examples

List ranking. For the *list-ranking* problem, we actually suggest an algorithm which is referred to, in [CV], as the "Basic list-ranking algorithm". This algorithm runs in $O(\log n \log\log n)$ time and $O(n)$ operation, and the constant factors hidden by the big-oh notation are 8 for both time and operations. The reason for mentioning this algorithm is that it seems to imply fewer operations than in other parallel list-ranking algorithms and at the same time provide considerable parallelism. The count was done in a similar way to the depth-first search example above and is omitted here. The total number of operations of the serial list-ranking algorithm given earlier is 2. So, for the linear model for F, the comparison is between $8(\log n \log\log n)A + 8nB$ and $2nA$. The simple randomized algorithms in [AM] and [Vi84b] achieve also similar performance. See [Vi93] for another example.

5 Caveats, Extensions and Discussion

• *Memory consistency.* The main issue is how to avoid a situation where an outdated copy of an address is interpreted as having a correct value. See [Vi93].

• *How much parallelism depends on the serial machine and the algorithms.* Figuring out which parallel algorithm yields the fastest running time on a serial computer, will involve a trade-off between the *level of parallelism* and the *total number of operations (including constant factors)*.

• *Throughput.* Our aforementioned challenge can be stated in terms of "throughput": secondary memories need to be capable of providing data at a sufficient rate for the processing unit. **The idea of random distribution of addresses on Cray-like memories appears to meet these throughput requirements**, as explained in the next paragraph.

• *Hardware technology for implementation of random distribution of addresses.* See [HP] regarding the the technology of memory banks. Since 1982 Cray machines have multiple memory pipelines. On the Cray X-MP (see [HJ]), one can make from 1 to 64 memory requests at a time. These memory requests do not have to be in any pattern, but can be scattered across memory. The requests are sent off serially, one per clock cycle, and after a latency of about 17 clock cycles (this can depend on collisions), the first element comes back. After that, the remaining elements come back one per clock cycle (again, there can be further delays due to collisions). The requests are guaranteed to come back in the same order as they were made. This means that if one can create blocks of memory requests (using parallelism), they can actually be received at an effective throughput of one request per clock cycle. The memory is actually organized into many banks, such that if all the requests are to the same bank, the throughput is significantly slower. Finally, we note that such memories are quite expensive.

• *Languages and compiler technology.* The final goal is that the compiler will be able to extract from a user-friendly program as much information as possible for predicting future memory address needs. This calls for an effort concerning both languages and compilers. Given a programming language, the challenge for compiler writers is well defined; the work by [CKP] is in this spirit. Next, consider design of programming languages which will enable expressing relevant information, that will later be used by the compiler to improve its predictions. A language in which parallelism can be expressed, will make it possible for a compiler to create the code for predicting memory accesses in the compiled program (due to parallelism) as advocated in the present paper. In addition, it will be nice to allow the programmer to have the option to express all his/her knowledge concerning memory access dependencies using the programming language (if he/she so desires, and not at the expense of user-friendliness).

• *Looser parallel specifications are sufficient.* Observe that the whole processor allocation, which is an essential part of a PRAM algorithm, will typically not be such a serious issue for extracting the parallelism needed here. A starting point for a weaker model than the PRAM for describing parallelism might be *Informal Data-Parallelism*, (originally presented in [SV] using Brent's theorem), as per [Vi94]; this has also been called the *work-time presentation paradigm* for parallel algorithms, as elucidated by the author in [Ja]. This paradigm suggests preceding a "full-PRAM" description of a parallel algorithm by a high-level description of work and time only. Specifically, this includes the operations to be performed at each parallel round, and even this can be done in an implicit way. Such a high-level description might be enough here.

6 Relevance for Parallel Computers

Most of the above considerations are valid also for the design of a parallel machine that will support the PRAM model of parallel computation.

This paragraph includes some background. As explained elsewhere (e.g., [Vi84a]), the PRAM should be viewed as a programmer's model for a parallel machine and not as a physical parallel machine, and improvement in the parallel running time of a PRAM algorithm can benefit us in reducing the actual running time in several ways. Specifically, [Va] (also in a CACM 1990 paper by L. Valiant) and [Vi84a] advocate *slackness in processors* as explained next. Suppose we are given an efficient PRAM algorithm and a (real) parallel machine with p_1 processors, on which we wish to simulate the algorithm. Suppose that the PRAM algorithm is specified for p_2 PRAM processors. (It is a trivial observation that such an algorithm scales also into an efficient algorithm for $< p_2$ processors.) In this case, *processor slackness* is defined as the ratio p_2/p_1. Informally, each of the above papers specifies ways for more efficient simulation by the real machine. Even if p_1 is fixed, having larger and larger p_2 (and therefore larger processor slackness) leads to improvements in efficiency. Several parallel machine designs take advantage of processor slackness; this includes the 1981 HEP design by B.J. Smith and the more recent design [ACCKPS], as well as [AKP]

In other words, the concept of processor slackness considers *parallelism as a resource.* Our thesis takes advantage of this resource for the degenerate case where the computer has a single processor. Informally, when replacing 1 by p_1 processors, requests to remote memories will be managed efficiently by using simulations of parallel algorithms on a parallel machine with p_1 processors. Namely, processor slackness together with a properly designed memory management system can also be used in a parallel machine for circumventing memory requests whose implementation is very costly, especially where locality of reference is not satisfied. Throughput will become even more critical than in the context of serial algorithms: retrieval of memory should be done at a rate compatible with the speed of the processors. It is interesting to mention that following experiments with implementation of parallel sorting algorithms, it has been observed that "throughput bottlenecks of this nature" exist in modern parallel machines such as the CM-2, by Thinking Machines Corporation [BLMPSZ]. Throughput bottlenecks occurred even at the Cray Y-MP [ZB], when interpreted as a parallel machine with 512 processors. (However, considering the Cray X-MP and Cray Y-MP machines as having between 1 and 8 Cray-1-like CPUs [HJ], as we do in the present paper, alleviates throughput bottlenecks.) There might be some insight in observing that, for both serial and parallel processing, alleviating throughput bottlenecks is helpful for supporting a user-friendly language.

Building a parallel machine that supports the PRAM is a considerable challenge. It is possible that **adoption of our more concrete ideas for serial computers will enable a more gradual transition into machines that support a "virtual PRAM" (i.e., as a programmer's model).** This is since developing the right tools for the memory management system of a serial computer may be a *first step* towards building a general-purpose parallel computer that uses such system. This gradual approach has the following potential advantage relative to some ongoing parallel machine projects. Ideally, all stages of a parallel machine project should aim at supporting a single ("robust") parallel programming language. Identifying a robust language might be the most significant challenge facing parallel computation today. To understand the importance of this issue, assume that we fail to define such a robust parallel language and imagine a computer industry decision maker that considers whether to invest several human-years in writing code for a certain parallel programming language. By the time the programming project will have been finished, the language is likely to become, or about to become, obsolete. The only reasonable business decision under this circumstances is to write code for some existing robust programming language, which presently would mean a serial programming language. This lack of a robust language is part of the **"par-**

allel software crisis" [P]. Following a meeting of an industry advisory board panel [P], pessimism is expressed about reaching agreement on a robust parallel programming language, because of the following. While, there seems to be agreement that a shared-memory ("PRAM-like") language will be ideal from the user end, present parallel machines require that parallel programs include considerably more machine-specific information for getting the best performance. It remains to be seen whether their design can afford supporting a programming language that will be considered satisfactory by "the user". Unfortunately, machines that do not support a robust parallel programming language may remain an academic exercise, since from the industry perspective, the test for successful parallel computers is their usability at the end. In contrast to this, our multi-stage approach is based on **using a shared-memory parallel programming language already in its first stage**.

The companion paper [Vi92] considers this multi-stage approach and adds the following perspective. The new generation of "serial machines" is far from being literally serial. Each possible state of a machine can be described by listing the contents of all its *data cells*, where data cells include memory cells and registers. For instance, pipelining with, say, s single cycle stages may be described by associating a data cell with each stage; all s cells may change in a single cycle of the machine. More generally, a transition function may describe all possible changes of data cells that can occur in a single cycle; the number of data cells that change in a cycle define the *machine-parallelism* of the cycle; a machine is *literally serial* if this number is never more than one. We claim that literally serial machines hardly exist and that considerable increase in machine-parallelism is to be expected. Machine-parallelism comes in such forms as pipelining, or in connection with the Very-Long Instruction Word (VLIW) technology, to mention just two. Still the "serial programmer's model" of computers as being taught in courses on algorithms and data structure in a standard computer science curriculum has not changed in the last several decades. The current paper suggests that parallel algorithms can benefit this new generation of machines, if these machines are adapted to be programmable by parallel algorithms and be designed to take advantage of their parallelism (by mapping "program parallelism" into their machine-parallelism). Another necessary condition for this to happen is that users will be capable to design parallel algorithms. Parallelism is a concern which is missing from "traditional" algorithm design, and the skill of designing parallel algorithms is based on first principles. Therefore, users should be taught this skill in school. Perhaps the main contribution of this paper is in demonstrating that parallel algorithms can be exploited to make more effective use of machine-parallelism irrespective of whether the machine itself is classified as "massively parallel", or even "parallel". Since the computer science curriculum is meant to prepare young graduates for a life time career, it is timely to put on the agenda a curriculum update to include the topic of parallel algorithms.

In [Vi88] I made the following **prediction**: A future parallel computer, which does not support the PRAM model of parallel computation (as a programmer's model), will not be competitive, as a general-purpose computer, with the strongest non parallel computer that will become available at the same time. This prediction is yet to be proven wrong, but is there reason for optimism? In recent years, multiprocessors based on inexpensive microprocessors, each with a large local memory ("shared nothing architectures"), allowed drastic increase in the size of fast memories, and led to a success of parallel data base systems [DG], increasing the interest of that community in faster parallel computation. Hash partitioning of data and hash join appear to become main stream techniques in this context. Note that [MV] actually suggested hash partitioning of memory addresses among the local memories of parallel processors as a **general method** for emulating a PRAM programmer's model on a similar shared nothing architecture! **Facing problems which are similar to possible bottlenecks targeted by commercial data base applications is a reason for some optimism.**

7 Conclusion

We encourage research in several directions. We encourage more experimental research to study the advantage of serially emulating more complicated parallel algorithms in order to cope with delays caused by computer memories. The present paper reports such work for the suffix tree problem; we encourage others to do the same for other problems.

This paper provides some evidence that prefetching will increase the effectiveness of serial machines. The challenge is twofold: (1) Advance the prefetching facility in hardware; and (2) find more ways to take advantage of software prefetching, either in the spirit of [CKP] or the present paper.

The fine structure of parallel programs may be beneficial in ways which are beyond the scope of this paper. Our thesis, in section 3, calls for exploring such opportunities.

Acknowledgements

The suffix tree programming was done by Suleyman Sahinalp. The code for McCreight's algorithm was written by William Chang, while at Berkeley. Guy Blelloch led me to the information about the Cray X-MP machine. Scott Carson led me to the information about the Encore Multimax computer. Olafur Gudmundsson was very helpful in providing technical data. Yossi Gil and Yossi Matias were particularly energetic in making thoughtful suggestions. I also got numerous useful comments from Richard Cole, Mike Franklin, Joseph JáJá, Hong Che Liu, Bill Pugh, Jim Purtilo, and Neal Young. All this help is gratefully acknowledged.

References

[AKP] F. Abolhassan, J. Keller and W.J. Paul. On the cost-effectiveness of PRAMs. In *Proc. 3rd IEEE Symp. on Par. and Dist. Proc.*, 1991, Dallas, TX.

[ACF] B. Alpern, L. Carter and E. Feig. Uniform memory hierarchies. In *Proc. 31st Ann. IEEE Symp. on Found. of Comp. Sci.*, 600-608, 1990.

[ACCKPS] R. Alverson, D. Callahan, D. Cumming, B. Koblenz, A. Porterfield and B. Smith. The Tera computer system. In *Proc. Int. Conf. on Supercomp.*, 1990, Amsterdam.

[AM] R.J. Anderson and G.L. Miller. A simple randomized parallel algorithm for list ranking. Inf. Proc. Lett., 33(5):269-273, 1990.

[AILSV] A. Apostolico, C. Iliopoulos, G.M. Landau, B. Schieber, and U. Vishkin. Parallel construction of a suffix tree with applications. *Algorithmica*, 3:347-365, 1988.

[BLMPSZ] G.E. Blelloch, C.E. Leiserson, B.M. Maggs, C.G. Plaxton, S.J. Smith, and M. Zagha. A comparison of sorting algorithms for the Connection Machine CM-2. In *Proc. 3rd SPAA*, 3-16, 1991.

[BCHSZ] G.E. Blelloch, S. Chatterjee, J.C. Harwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. In *Proc. 4th ACM PPOPP*, 102–111, 1993.

[CKP] D. Callahan, K. Kennedy and A. Porterfield. Software prefetching. In *Proc. 4th ACM Int. Conf. on Arch. Support for Prog. Lang. and Op. Sys., Santa Clara, CA* 40-52, 1991.

[CV] R. Cole and U. Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Inf. and Cont.*, 70:32–53, 1986.

[DG] D. DeWitt and J. Gray. Parallel database systems: the future of high-performance data base systems. *CACM*, 35:85–98, 1992.

[DM] M. Dietzfelbinger and F. Meyer auf der Heide. How to distribute a dictionary in a complete network. In *Proc. 22nd ACM Symp. on Theory of Comp.*, 117–127, 1990.

[FKLMSY] A. Fiat, R.M. Karp, M. Luby, L. McGeoch, D. Sleator, and N. Young. Competitive paging algorithms. *J. Algorithms*, 12,4: 685-699, 1991.

[HP] J.L. Hennessy and D.A. Patterson. *Computer Architecture A Qualitative Approach.* Morgan Kaufmann, San Mateo, California, 1990.

[HJ] R.W. Hockney and C.R. Jesshope. *Parallel Computers 2.* Adam Hilger, Philadelphia, PA, 1988.

[Ja] J. JáJá. *An Introduction to Parallel Algorithms.* Addison-Wesley, Reading, MA, 1992.

[Jo] N. Jouppi. Improving direct-mapped cache performance by the addition of a small fully associative cache and prefetch buffers. In *Proc. 14th Annual Conf. Int. Symp. on Comp. Arch.*, 1990.

[KRS] C.P. Kruskal, L. Rudolph, and M. Snir. A complexity theory of efficient parallel algorithms. In *Theoretical Computer Science*, 71:95-132, 1990.

[Mc] E.M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23:262–272, 1976.

[MV] K. Mehlhorn, and U. Vishkin. Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories. *Acta Informatica*, 21:339-374, 1984.

[MR] M. Metcalf and J. Reid. *Fortran 90 Explained.* Oxford University Press, New York, 1990.

[P] C.M. Pancake. Software support for parallel computing: where are we headed? *CACM*, 34,11: 53–64, 1991.

[RC] E. Reghbati and D.G. Corneil. Parallel computations in graph theory. *SIAM J. Computing*, 7: 230-237, 1978.

[SV] Y. Shiloach and U. Vishkin. An $O(nlog^2 n)$ parallel Max-Flow Algorithm. *J. Algorithms*, 3:128–146, 1982.

[St] W. Stallings. *Handbook of Computer Communications Standards, Volume 2.* Macmillan, 1987.

[Va] L.G. Valiant. General purpose parallel architectures. In Handbook of Theor. Comp. Sci.: Volume A, (Editor J. van Leeuwen), MIT Press/Elsevier, 1990, 942–971.

[Vi84a] U. Vishkin. A parallel-design distributed-implementation (PDDI) general purpose computer. *Theor. Comp. Sci.*, 32:157–172, 1984.

[Vi84b] U. Vishkin. Randomized speed-ups in parallel computations. In *Proc. 16th Ann. ACM Symp. on Theory of Comp.*, 230–239, 1984.

[Vi88] U. Vishkin. PRAM algorithms: teach and preach. In collection of position papers, IBM-NSF Workshop on “Opportunities and Constraints of Parallel Computing”, IBM Almaden, 1988.

[Vi92] U. Vishkin. A case for the PRAM as a standard programmer's model. In *Proc. 1st Heinz Nixdorf Symp. on Par. Arch. and Their Efficient Use, 1992*, Vol. 678, Lect. Notes in Comp. Sci., Springer, 1993.

[Vi93] U. Vishkin. Can parallel algorithms enhance serial implementation? Univ. of Maryland Inst. for Advanced Computer Studies, UMIACS-TR-91-145.1, 1991, revised 1993.

[Vi94] U. Vishkin. Thinking in Parallel: Some Basic Data-Parallel Algorithms and Techniques. Monograph, in preparation.

[VS] J.S. Vitter, and R.A. Simmons. New classes for parallel complexity: a study of unification and other complete problems for P. *IEEE Trans. on Computers*, C-35,5:403-418, 1986.

[ZB] M. Zagha and G.E. Blelloch. Radix sort of vector multiprocessors. In *Proc. Supercomp. 91.*