Exploiting Fine Grained Parallelism on the SPE*

Emanuele Milani and Nicola Zago

Department of Information Engineering,
University of Padova, Padova, ITALY
{milaniem,zagonico}@dei.unipd.it

Abstract. In this paper we propose a simulation of Work-Time framework algorithms on the recently proposed Speculative Prefetcher and Evaluator (SPE) processor, using a pipelined hierarchical memory. This allows us to inherit the efficency of work-optimal parallel algorithms in this new model.

Keywords: computational models, hierarchical pipelined memory, work-time simulation, efficient merge

1 Introduction

The *Random Access Machine* (RAM) is an idealized sequential computational model, in which the time to access any memory location is independent from memory size [8]. This assumption is unsuitable for physical machines, because of the principles of maximum information density and maximum information speed [4]. Indeed their combination imposes a minimum access latency, which grows with the size of the memory. RAM complexity is therefore an automatic lower bound for a given problem on any real sequential machine.

One of the main aims of recent literature is to devise implementable machine designs which hide or limit the latency impact, matching the ideal lower bounds. To this purpose, two major algorithmic strategies have been investigated: *locality* and *concurrency* of memory accesses. At the same time, models suitable to exhibit and measure them have been devised, respectively by means of hierarchical memories and pipelined memories.

Among the first we recall the Hierarchical Memory Model (HMM) [1], which is characterized by a non-decreasing function $a(x)$ that describes the access time to location x, implying that locations near to the processor take a lower time to be accessed. The Block Transfer (BT) [2] model extends the HMM, allowing the transfer of B adjacent locations starting from address x in $a(x) + B$ steps. These models encourage the design of algorithms exhibiting *temporal locality*, that is to use more often memory locations with a lower address, since they have a lower access time. The BT considers also *spatial locality*, or the access

* This work was supported, in part, by MIUR-PRIN Project *AlgoDEEP*, by PAT-INFN Project *AuroraScience*, and by the University of Padova Projects *STPD08JA32* and *CPDA099949*.

of adjacent data in a short time window. These models are good simplification of actual computer memories, which are hierarchically divided in several levels – from fast yet small caches to slow yet huge mass storage – among which data is transferred in blocks. A special case of BT is the *Disk Model* (DM) [15], which has been used to model the disk bottleneck and study the disk I/O efficiency of algorithms. Although general simulations of RAM algorithms in these models yield a worst case slowdown proportional to the memory latency, algorithms exhibiting locality can reach the same performance, as in *matrix multiplication* [1]. Nevertheless, algorithms which need to read the whole input present superlinear lower bounds; for example the *touch problem* – which consists in accessing each of the n elements in input and triavially solvable in n step in RAM model – has $\Theta(n) = n \log^* n$ complexity in the BT model with access function $a(x) = \log x$ [2].

On the other hand, *Pipelined Memories* (PM) [14] allow latency hiding through overlap of accesses. In particular they can perform k independent requests to the memory of M locations waiting only $O(a(M)+k)$ step for receiving all the responses. One should note that, unlike BT, accesses need not involve adjacent locations, nevertheless we have to know in advance enough independent requests to amortize the latency cost. PMs can solve the touch problem in linear time, still they have superlinear performance in problems where there are strong dependencies among instructions.

Recently [3] introduced a pipelined and hierarchical memory design which complies with physical constraints. This, jointly with the SPE processor, forms the Pipelined Hierarchical Memory Machine (PHMM), which is able to match RAM complexity ($O(1)$ slowdown) on wide classes of programs, exploiting both concurrency and locality.

Before memory models, these strategies were already been extensively studied in parallel computing, since concurrency allows independent executions among the processors and locality limits communication among these. This fact is pointed out by several works which show how to effectively simulate parallel models in memory models, partially carrying the knowledge of parallel computing in this field. For example, in [6] and [14], general Parallel Random Access Machine (PRAM, the ideal parallel model) simulations are proposed respectively on DM and PM, deriving new upper bounds for some problems on these memory models exploiting previously known parallel results. In [9] is shown how to turn the submachine locality of the Decomposable Bulk Synchronous Parallel model (D-BSP, a parallel model where also communication and synchronization costs are considered) in locality of references for the HMM.

Our paper is related to these works; in fact we propose a simulation of Work-Time (WT) framework [12] – a parallel framework which highlights parallelism and critical path of parallel algorithms – on the PHMM. This simulation, when applied to work-optimal WT algorithms, provides optimal algorithms for the PHMM. In particular we use it to obtain an optimal merge implementation, improving the previously best known result, which has superlinear complexity.

We also show how the whole exploitation of available parallelism can lead to a simulation with a huge memory footprint and degrading its performance.

The paper is organized as follows: in Sections 2 and 3 we recall concepts about SPE and parallelism. Section 4 contains our simulation, whose applications are shown in Sect. 5. Conclusions and further research directions are in Sect. 6.

2 The Speculative Prefetcher and Evaluator

The Speculative Prefetcher and Evaluator is a processor design which is able to exploit a *Pipelined Hierarchical Memory* (PHM) while complying with the physical constraints discussed in [4]. Both have been introduced in [3].

The memory features a size M and latency access function $a(x)$. It can accept a request per cycle for an arbitrary location x, guaranteeing a response within $a(x)$ cycles. One should note that the latency of k requests is determined by the location with higher address accessed, unlike PMs where latency is always $a(M)$.

SPE has an Instruction Generator Unit (IGU), connected to an instruction PHM, and an Instruction Execution Unit (IEU), connected to a data PHM of M locations. Both IGU and IEU have $O(k)$ constant–sized units called *stations* and arranged as linear arrays. Parameter k denotes the *processor size* and is choosen to match the worst case latency $a(M)$.

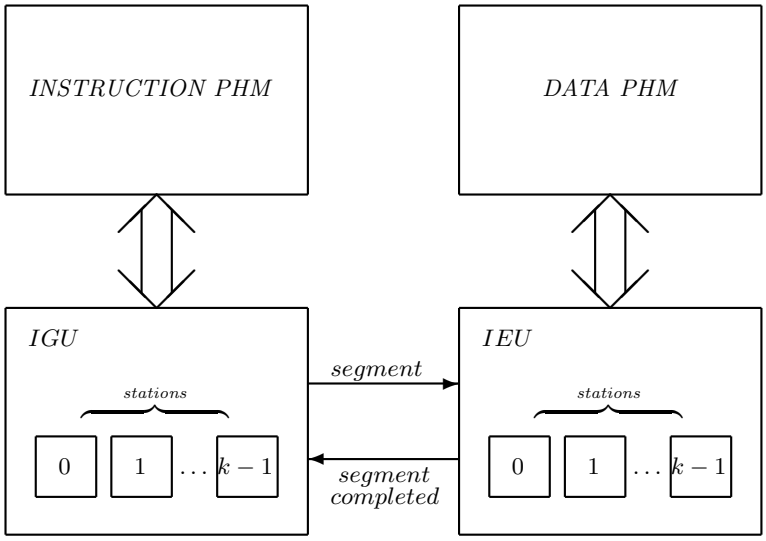

Fig. 1. Scheme of the PHMM architecture: IGU reads instructions from the *instruction PHM*, produces a *segment* and passes it to IEU stations. Here data are speculatively prefetched and instructions are executed, solving possible invalid data with the internal forwarding or reading them from *data PHM*.

The computation consists in a sequence of *stages*. In each stage the IGU reads instructions from the instructions PHM, assembling a *segment* (the number of stations used in the IEU) of machine code instructions for the IEU. In particular IGU predicts possible branches of the code and then loads the obtained sequential segment in the IEU, one instruction per station. The execution consists in a series of *rounds* in which instructions are executed. Each round is divided in two parts: in the first the IEU speculatively prefetches the operands of all the instructions of the segment, both directly and indirectly addressed; in the second, all the stations execute sequentially with the speculative operands they hold. In case of mispredicted operands, due for example to indirectly addressed operands whose base address has been changed by previous stations, another round takes place.

Anyway, some memory accesses are avoided through a mechanism of internal forwarding among stations. This mechanism allows the stations to forward their computed result along the linear array connecting them. In this way, if the value of an operand is modified by a station, making invalid the prefetched values of the followings stations, the new value will be immediately available to them without new memory accesses, but simply receiving it from the linear array.

One should note that the load of $a(M)$ instructions per round gives us an amortized $O(1)$ set up time per station. Moreover the segment size can be dinamically decreased to take advantage from the hierarchical structure of the memory. When the speculative prefetch or the internal forwarding succeed in avoiding further memory accesses, also each instruction execution takes $O(1)$ step.

In order to understand the segment assembly problems, we recall the definition of dependency-related concepts.

Definition 1. *There is* Functional Dependency *(FD) between instructions I_j and I_k if I_j modifies $m[x]$, and I_k uses the content of $m[x]$ as operand or to indirectly address the output location, while no operation among I_j and I_k modifies $m[x]$.*

Definition 2. *There is* Address Dependency *(AD) between instructions I_j and I_k if I_j modifies $m[x]$, and I_k uses the content of $m[x]$ to indirectly access to an operand, while no operation among I_j and I_k modifies $m[x]$.*

Clearly, memory accesses due to FD can be avoided in SPE thanks to internal forwarding. Instead AD is harder to be addressed and requires a new fetch of operands.

Definition 3. *Given the instruction stream $(I_1, I_2, \ldots, I_N)$ produced by the execution of a SPE program $\mathcal{P}$ on a particular input, its* address dependence depth *D is the maximum length of a subsequence $I_{j_1}, I_{j_2}, \ldots, I_{j_L}$ with $j_1 < j_2 < \cdots < j_L$ where subsequent instructions have address dependency.*

Let us recall two important program categories.

Definition 4. *A program $\mathscr{P}$ is* straigth–line *if it consists of only data processing instructions.*

Definition 5. *A program $\mathscr{P}$ is* direct–flow *if it is straigth–line and does not use indirect addressing.*

In [3] it is shown that any N instructions straigth–line program with address dependence depth D and accessing memory locations with address smaller than M can be executed by an SPE with PHM in time $T = O((D+1)(N+a(M)))$. Moreover, a direct–flow program can be executed in $T = O(N + a(M))$.

In particular the following result from [3] holds:

Proposition 1. *A program consisting in nested for loops where the only branches are those related to the cycles, can be executed in $T = O(D(N + a(M)))$.*

Proposition 1 applies to wide classes of programs, such as FFT and Matrix Multiplication.

We quote the following example, which intuitively shows how this occurs.

Example 1. Let's consider the execution of a C-like code that increments every element of an array: `for i=1 to k; A[i] = A[i]+1`. Using the naive branch prediction policy that always reenters the loop, the IGU can unroll the loop in `i=1; m[i]=m[i]+1; i=i+1; m[i]=m[i]+1; ....` At this point, the SPE speculatively calculates all the i values in the first round, prefetching the right operands at the beginning of the second one, whose speculative execution correctly completes the segment execution. So it can resolve address dependencies in O(1) amortized time.

3 Parallel Computing Background

Definition 6. *A* Parallel Random Access Machine *(PRAM) [10, 11] is an abstract parallel machine model, that consists in a collection of P synchronous processors and M shared memory locations.*

Definition 7. *A* PRAM program *is a sequence of parallel steps, each of which specifies an instruction per processor.*

Beside the number of nodes, the computational power of a PRAM is determined by which shared memory operations are permitted. Within a step, in fact, each memory location may or may not be accessed by more than one processor. In other words, a PRAM can be provided with either an *exclusive read* (resp. *exclusive write*) memory, or a *concurrent read* (resp. *concurrent write*) memory. Moreover, when concurrent writes are allowed, a contention policy must be specified in order to determine the actual memory state after the access. The most studied configurations, in order of increasing power, are exclusive read exclusive write (EREW), concurrent read exclusive write (CREW), concurrent read concurrent write (CRCW).

Both SIMD and MIMD versions have been studied, anyway they are equivalent [7] if they feature the same memory access policy.

Definition 8. *The* Work-Time model *(WT) [12] is a parallel programming model in which an algorithm $\mathscr{A}_{WT}$ consists in an ordered sequence of T sets $s_0, \ldots, s_{T-1}$ of independent operations on M_{WT} memory locations. Different sets may differ in size and therefore exhibit more or less parallelism. Let $|s_i| = p_i$, then we define the* work W *of $\mathscr{A}_{WT}$ as $W = \sum_{i=0}^{T-1} p_i$.*

It should be noted that, since it is always possible to simulate $\mathscr{A}_{WT}$ on a RAM in time $T_{RAM} = W$, lower bounds on RAM complexity automatically hold also for the work. In particular, let T^*_{RAM} be the best RAM complexity for a given problem. Then, the equivalent WT algorithm $\mathscr{A}_{WT}$ is *work–optimal* if and only if W is $O(T^*_{RAM})$.

WT algorithms are meant to be executed by PRAMs, by means of a schedule. A sufficient condition for a valid schedule of $\mathscr{A}_{WT}$ in a PRAM is that each operation in s_i is executed after all operations in s_{i-1} and before any operation in s_{i+1}. This schedule allows us to apply *Brent's Theorem* [5] and to execute $\mathscr{A}_{WT}$ in a PRAM with P processors in a time $O(\frac{W}{P} + T)$.

On the other hand, it is not clear how we can reschedule a PRAM program for P processors as the processor number increases, since possible dependencies between steps are not stated explicitly. For this reason WT framework is much more convenient if we need to extract dependencies and available parallelism.

The major construct of the WT model is the `pardo`, which specifies a parallel step with a syntax similar to a traditional `for`. The main difference is that the cycle index denotes just the index of an element of the set of instructions and can not be modified by the instructions. For example

```
for j, 1 ≤ j ≤ p pardo
      operation_j
```

denotes a set of p independent operations, whose execution order is irrelevant.

Since WT framework is a very high level model, it is important to pay attention to some hidden low level details. In particular, one should note that in each parallel step:

F1 guarantees that all the reads take place before any write;
F2 allows addresses to be expressed in a high level fashion.

Therefore any simulation or implementation of an algorithm which relies on such features has to provide them. For the sake of clarity we avoid to explicitly address these issues by resorting to a slightly more constrained yet equivalent case. We then show how to map the general case to this.

Let us introduce a class of WT algorithms.

Definition 9. *A step of WT algorithm $\mathscr{A}_{WT}$ is* CRCW decoupled *if any concurrently accessed memory location is either read or written. $\mathscr{A}_{WT}$ is itself* CRCW decoupled *if this condition holds for each step.*

Any CRCW decoupled algorithm does not rely on (F1). In the opposite case, it is possible to devise an equivalent CRCW decoupled algorithm with the same

work and time complexity. In fact, it suffices to split each parallel step into two sub–steps. The first fills an auxiliary array with the operation inputs, while the second performs the actual execution, reading from the array. In the worst case, the memory overhead is $O(p)$.

Consider now, without loss of generality a SIMD parallel step in the WT framework, which executes on an initial memory state $\mathscr{M}_i$ and leads to final state $\mathscr{M}_{i+1}$. Available parallelism and the memory locations that must be read or written (both usually parametrized with the size of the input) are indicated by a `pardo` statement. In particular an index j is used to distinguish each concurrent operation. Note that the memory to store one step is constant and therefore the whole program takes $O(T)$ memory.

One can think the operands of operation j to be a function of j. Typically, such function is simple enough to be expressed by the addressing modes of a modern instruction set (for example a base address and an index–dependent offset). In the most general case, when (F2) is fully exploited, the function can be explicitly used to prepare an auxiliary operand vector. This preliminary phase can be implemented with a memory overhead depending on how much parallelism we want exploit (up to p_i). In particular for the SPE, $O(k)$ memory locations are sufficient.

4 Simulation of WT Algorithms

One way to write efficient programs for SPE is to exploit the parallelism of Work–Time algorithms. Parallel steps can be efficiently coded into programs, also in case of concurrent memory accesses, which can be implemented with little effort. In fact, the address dependence depth of the resulting segments is $O(1)$, and memory accesses can be fully pipelined.

However, it must be noted that a trivial static unrolling of a `pardo` statement could lead to an SPE program with $O(W)$ size. In this case, instruction fetch latencies could be larger that data latencies, thus hindering time efficiency.

A more complicated unrolling, proportional to processor size, can be devised, which leads to programs with $O(kT)$ size. This last strategy is not the most compact; still it is interesting because it also applies to the SP processor (see [3]). On the contrary SP does not efficiently support the strategy described next.

Without loss of generality (see Section 3), let us restrict our scope to CRCW decoupled WT algorithms. The resulting simulation is itself rather simple. If only exclusive writes are used, WT statement

```
for j, 1 ≤ j ≤ p pardo
      operation_j
```

can be coded for SPE with a loop in the form:

```
segmentsize(min(k, p))
for j, 1 ≤ j ≤ p do
      instructions_j
```

where $\texttt{instructions}_j$ is the SPE coding for the high level WT operation.

As for concurrent writes, contention policies are quite different one from another, and therefore different approaches are needed for their implementation. For example, in case of reduction–like policies, such as MAX, + or logical AND, it is sufficient to append the appropriate reduction instruction to the core of the loop. The resulting SPE code would look like:

```
segmentsize(min(k, p))
for j, 1 ≤ j ≤ p do
        instructions_j
        acc ← max{acc; output_j}
```

where $output_j$ is the result of $\texttt{instructions}_j$, the reduce operation is a MAX and the final result is accumulated in variable *acc*.

Another common policy, *priority* CW PRAM, can be implemented recurring to *predicated instructions*, whose output is committed to memory only if a certain condition is verified.

It must be noted that the correctness of the simulation relies on the following fatcs:

- SPE instructions are chosen to match the corresponding WT operations;
- each SPE instruction gets the right operands;
- memory writes are consistent with the policy specified by $\mathscr{A}_{WT}$.

Hence, Proposition 2 holds true.

Proposition 2. *Given WT parallel step s_i, it is possible to implement an equivalent SPE program $\mathscr{P}$, such that they both lead from memory state $\mathscr{M}_{i-1}$ to $\mathscr{M}_i$.*

Before examining the time complexity of the simulation, we must consider its memory footprint.

Proposition 3. *Let $M_{WT}^{(i)}$ (resp. $M_{WT}^{(i+1)}$) be the size of memory state $\mathscr{M}_i$ (resp. $\mathscr{M}_{i+1}$), and let M_{PH} be the amount of memory needed by $\mathscr{P}$. Then, both M_{PH} and $M_{WT}^{(i+1)}$ are $O(M_{WT}^{(i)} + p)$.*

Proof. Since p is the number of operations of the parallel step, $M_{WT}^{(i+1)}$ is $O(M_{WT}^{(i)} + p)$.

As for M_{PH}, an auxiliary operand vector only needs $O(\min\{p, k\}) = O(p)$ extra space. □

Proposition 4. *WT CRCW parallel step with p available parallelism can be translated into a SPE program $\mathscr{P}$ with $O(p + a(M_{PH}))$ time complexity.*

Proof. Consider the for loop which implements the simulation. Its body has $O(1)$ address dependence depth. Therefore, as in Example 1, the IGU is able to roll out $\lceil p/k \rceil$ segments with $O(k + a(M_{PH}))$ time complexity each. More precisely, at least $\lceil p/k \rceil - 1$ segments have $O(k)$ complexity, since $k \geq a(M_{total}) \geq a(M_{PH})$. Summing up, we get $O(p + a(M_{PH}))$.

As for reduction–like CWs, the simulation adds a functional dependency for each concurrent WT operation. Anyway, the internal forwarding mechanism of SPE can deal with them at a constant multiplicative slowdown. The same holds for CW implementations based on predicated instructions. □

Next we show how Proposition 4 can be repeatedly applied in order to get a whole SPE implementation of $\mathscr{A}_{WT}$. Correctness follows from the fact that each single application produces the same memory state as the correspondent parallel step.

Theorem 1. *Consider WT algorithm $\mathscr{A}_{WT}$, with W work and T time complexity. Then an equivalent SPE program $\mathscr{P}_A$ can be written, with complexity $O(W + Ta(M_{PH}))$, where M_{PH} is $\max_i\{M_{PH}^{(i)}\}$.*

Proof. $\mathscr{P}_A$ can be obtained with T applications of Proposition 4. The resulting complexity is therefore $\sum_i O(p_i + a(M_{PH}^{(i)}))$, which is $O(W + Ta(M_{PH}))$. □

One should note that this simulation results in a program of $O(T)$ instructions. Therefore instruction memory latencies can be ignored.

Corollary 1. *Let $\mathscr{A}_{WT}$ be a work–optimal WT algorithm. If $Ta(M_{PH})$ is $O(W)$, then there exists a SPE simulation of $\mathscr{A}_{WT}$ with optimal RAM complexity.*

As for M_{PH}, an immediate consequence of Prop. 3 follows.

Proposition 5. *$M_{PH} = \max_i\{M_{PH}^{(i)}\}$ is bounded from below by the maximum available parallelism of $\mathscr{A}_{WT}$ plus input size; from above by the work and the input size. Formally: $n + \max_i\{p_i\} \leq M_{PH} \leq n + W$.*

In general, any work–optimal parallel algorithm with polylogarithmic time complexity is a good candidate for efficient implementations on the SPE, if $a(x) < x$.

Anyway, another metric emerges from Prop. 5. In fact, exploiting all the available parallelism can affect memory usage, therefore increasing worst case latencies. Actually, once the condition $Ta(M_{PH}) = O(W)$ is met, any further parallelism would just increase memory footprint.

Section 5 contains a clear example of how this metric can be used.

5 Applications

Corollary 1 can be successfully applied to work–optimal WT algorithms which exhibit polylogarithmic time T, whenever the memory access function of the PHM is x^α, $0 < \alpha < 1$ or $\log x$. Finding *connected components* of a dense, undirected graph, for example, can be done with $T = \log^2 n$ and $W = n^2$ (see [12]) where input size is $O(n^2)$, n being the number of nodes. Therefore, we can implement a program for SPE with $O(n^2 + \log^2 n \cdot a(n^2)) = O(n^2)$ complexity, which is optimal.

An analogous result holds for the problem of merging two sorted lists of n elements. In this case we can resort to the work–optimal algorithm in [13], which results in a linear SPE program.

Some further analysis is needed, though, when these solutions are used as subroutines of larger programs. In particular, the underlying assumption that all the input is stored in the fastest memory locations may not hold. Consider for example an iterative bottom–up implementation of mergesort for an SPE with $a(x) = x^\alpha$. At iteration j, we have to merge pairs of 2^j-sized lists, with the ith pair starting at position $2^{j+1}i$. Each such merge has a $O(2^j + a(2^{j+1}i))$ complexity, which results to be superlinear if $O(2^j) < O(a(2^{j+1}i))$. For example the overall complexity of step $j = 0$ is $O(n^{1+\alpha})$.

In this case, a technique similar to the execution of consecutive searches of [14] can be employed. Basically, instead of merging one pair of sublists at a time, whenever the size of the subinstances is small enough, all the merges advance "concurrently". Therefore, it is possible to obtain segments of independent instructions, which are executed efficiently.

An interesting example is matrix multiplication of two $n \times n$ square matrices (input size is $O(n^2)$). Implementing the standard WT algorithm with n^3 parallelism yields an optimal SPE implementation $\mathscr{P}$, as far as time complexity is concerned. The result is achieved even if no locality is exploited. Anyway, $\mathscr{P}$ requires $\Omega(n^3)$ space to be executed. In other words, a SPE with M available memory size, could not multiply matrices bigger than $M^{1/3} \times M^{1/3}$. On the other hand, implementing a WT algorithm with n^2 available parallelism yields a both time and space optimal SPE program.

6 Conclusions and Future Work

This paper shows a general technique for exploiting the parallelism expressed by the WT framework in the SPE. In particular, it shows how concurrent operations can be sequentially executed in a pipelined–efficient way, and how such efficiency can be measured. Besides, it is also shown how too much parallelism can negatively affect memory usage, also when time complexity is not compromised.

The more straightforward research line involves assessing a larger group of problems and algorithms, possibly mapping whole computational categories to classes of SPE programs.

A second line is directed to the integration of this work with memory hierarchy exploitation.

Finally, as a link between coarse grained parallelism and memory hierarchy exploitation has already been proved [9], it would be interesting to see if and how such link exists also for pipelined memory exploitation.

References

[1] Aggarwal, A., Alpern, B., Chandra, A. K., and Snir, M.: A model for hierarchical memory. In *Proceedings of the 19th ACM Symposium on Theory of Computing*. ACM, New York, 305–314, 1987.

[2] Aggarwal, A., Chandra, A. K., and Snir, M.: Hierarchical memory with block transfer. In *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society Press, Los Alamitos, 204–216, 1987.

[3] Bilardi, G., Ekanadham, K., and Pattnaik, P.: On approximating the ideal random access machine by physical machines. *Journal of the ACM 56*, 5, 1–57, August 2009.

[4] Bilardi, G., and Preparata, F.: Horizons of parallel computation. *J. Parall. Distrib. Comput. 27*, 2, 172–182, 1995.

[5] Brent, R. P.: The Parallel Evaluation of General Arithmetic Expressions. *Journal of the ACM 21*, 2, 201–206, April 1974.

[6] Chiang, Y., Goodrich, M. T., Grove, E. F., Tamassia, R., Vengroff, D. E., and Vitter, J. S.: External Memory Graph Algorithms. In *Proceeding SODA '95 Proceedings of the sixth annual ACM-SIAM Symposium on Discrete Algorithms*, 139–149, 1995.

[7] Collins, R. J.: MIMD Emulation on the Connection Machine. Technical Report CSD-910004, Dept. of Computer Science, Univ. of California, Los Angeles, Feb. 1988.

[8] Cook, S. A., Reckhow, R. A.: Time-bounded random access machines. *J. Comput. Syst. Sci. 7*, 4, 354–375, 1973.

[9] Fantozzi, C., Pietracaprina, A. A., Pucci, G.: Translating Submachine Locality into Locality of Reference. *Journal of Parallel and Distributed Computing 66*, 5, 633–646, 2006.

[10] Fortune, S., and Wyllie, J.: Parallelism in Random Access Machines. In *Proceeding STOC '78 Proceedings of the tenth annual ACM symposium on Theory of computing*, 114–118, 1978.

[11] Goldschlager, L. M.: A Unified Approach to models of Synchronous Parallel Machines. In *Proceeding STOC '78 Proceedings of the tenth annual ACM symposium on Theory of computing*, 89–94, 1978.

[12] Jájá, J. F.: An introduction to parallel algorithms. Addison Wesley Longman Publishing Co., Inc. Redwood City, CA, USA, 1992. ISBN:0-201-54856-9.

[13] Kruskal, C.: Searching, Merging and Sorting in Parallel Computation. *IEEE Transactions on Computers - TC 32*, 10, 942–946, 1983.

[14] Luccio, F., and Pagli, L.: A model of sequential computation with pipelined access to memory. *Math. Syst. Theory 26*, 4, 343–356, 1993.

[15] Vitter, J. S.: External Memory Algorithms. In *Proceedings of the 6th Annual European Symposium on Algorithms*, Springer-Verlag, Berlin, Germany, 1–25, 1998.