Communication-Processor Tradeoffs in Limited Resources PRAM *

Adnan Agbaria
CS. Dept. Technion city 32000, Haifa, Israel.
Email: adnan@cs.technion.ac.il

Yosi Ben-Asher
CS. Dept. Haifa university 31905, Haifa, Israel.
Fax: 972-4-8249331, Email: yosi@cs.haifa.ac.il

Ilan Newman
CS. Dept. Haifa university 31905, Haifa, Israel.
Fax: 972-4-8249331, Email: ilan@cs.haifa.ac.il

Abstract

We consider a simple restriction of the PRAM model (called PPRAM), where the input is arbitrarily partitioned between a fixed set of p processors and the shared memory is restricted to m cells. This model allows for investigating the tradeoffs/ bottlenecks with respect to the communication bandwidth (modeled by the shared memory size m) and the number of processors p. The model is quite simple and allows the design of optimal algorithms without losing the effect of communication bottlenecks.

We have focused on the PPRAM complexity of problems that have $\tilde{O}(n)$ sequential solutions (where n is the input size), and where $m \leq p \leq n$. We show essentially tight time bounds (up to logarithmic factors) for several problems in this model such as summing, Boolean threshold, routing, integer sorting, list reversal and k-selection. We get typically two sorts of complexity behaviours for these problems: One type is $\tilde{O}(n/p+p/m)$ which means that the time scales with the number of processors and with memory size (in appropriate ranges) but not with both. The other is $\tilde{O}(n/m)$ which means that the running time does not scale with p and reflects a communication bottleneck (as long as $m < p$). We are not aware of any problem whose complexity scales with both p and m (e.g $O(\frac{n}{\sqrt{m \cdot p}})$). This might explain why in actual implementations one often fails to get p-scalability for p close to n.

*An extended abstract of this paper appeared in SPAA99

Index terms- Parallel, PRAM(m), BSP, communication bandwidth, bounded number of processors.

1 Introduction

We propose and investigate a PRAM like model that is resource bounded and referred hereafter as PPRAM (Partitioned PRAM). The model is a natural blend between the DMM [23] and the theoretical model of Vishkin and Wigderson [33]. The PPRAM model is syntactically very close to the model in [33] which is referred hereafter as the PRAM(m). Both models are abstract PRAM models (without underlying communication network). In both there are p processors and m shared memory cells. In addition, the focus of study is similar: In both, when considering a computational task for a problem of size n, the interesting range of parameters is of $m \leq p \leq n$. The reason in both cases, is to model simultaneously the effect of low communication bandwidth (modeled by the bound on the number shared memory cells which, in turn, provides the only mean of communication between the processors) and bounded number of processors. However, unlike in PRAM(m) in which the input is presented in an additional read-only tape and is accessible to all processors, the input here is partitioned between the processors so that each processor gets n/p pieces of the input (much like the way the input is in the DMM). Indeed for some problems (as discussed ahead) this difference results in very different time bounds. The rational for the input convention, as in the DMM model, is that, in reality, algorithms are typically being cascaded, namely, the output of one is the input for the next (or the input just comes from the local disk of each processor). The requirement that the input is *evenly partitioned* is somewhat restrictive, however, this allows for more refined look into the complexity as unbalanced partition typically causes severe communication bottleneck.

The purpose of this work is first to introduce the PPRAM model which is very simple theoretically, both for proving lower bounds, and for developing algorithms (as there is no underlying network, and communication bottleneck is modeled in a very simplistic way). It allows easy development of algorithms that are optimally designed for any pair of m and p, yielding optimal implementations on real machines by experimentally finding out the 'effective' m and p. In Section 6 we show experiments implementing matrix multiplication in HPF to that effect.

Our second goal, following [33], is to theoretically investigate the communication-processors tradeoffs in a 'partitioned input' model for specific simple problems which are the building blocks of many parallel algorithms (e.g summing, Boolean threshold, routing, integer sorting, list reversal, k-selection etc.). We take into the time account read/write steps as well as internal steps, each with a unit cost. All algorithms and lower bounds (except for Observation 2.3) are for the deterministic case. We get the following results for the CRCW model (hereafter $\tilde{O}(), \tilde{\Theta}()$ represent the complexity disregarding logarithmic factors):

$\tilde{\Theta}(\frac{n}{p} + \frac{p}{m})$ time complexity for k-selection, minimum and sum. These problems scale nicely with either p or m but not with both.

$\tilde{\Theta}(\frac{n}{m})$ time complexity for integer sorting, $k - k$ routing, permutation routing, list reversal. These problems scale nicely with m but are nearly not affected by the increase in p (as long as $m < p < n$).

In addition we investigate the complexity of computing Boolean threshold functions for which we determine the complexity to be $\tilde{\Theta}(\frac{n}{p} + min(\frac{k}{m}, \frac{p}{m}))$ (here k is the threshold). We note that our PPRAM algorithms for Boolean threshold improve upon the previously best known algorithms even for the stronger PRAM(m) model of Vishkin et al. [33] (see [29, 8] for previous work on this problem).

Interestingly enough, we were not been able to identify problems that scale with both m and p (e.g. $O(\frac{n}{\sqrt{mp}})$, of [27] for randomized list reversal). This might, in turn, contribute to the understanding of why the running time of theoretically 'well parallelized' algorithms do not in fact scale with the increase in the number of resources (especially for small enough problems). The last point brings up another interesting issue: In practice (and theory) one would often like to analyze the time complexity for relatively 'small' problems (small n) for which unexpected non-scalable behaviour occurs. It is folklore that in such ranges, one typically fails to get speedups. Theory (asymptotic analysis) can not treat such framework directly. In our framework, one gets some intuition for such cases by letting $p \approx n$ and varying m. Again, the two types of tradeoffs we get, none of which exhibits simultaneous scalability in both p and m, may serve as a possible explanation to these speedup failures.

We end this discussion by noting that imposing different read/write conventions, result in different variants of the PPRAM model. Along the sequel, unless otherwise stated, we consider the strongest CRCW-PRIORITY for lower bounds, and either CREW or CRCW COMMON for upper bounds. As it turns out, upper and lower bounds usually match (some times, up to $logn$ factors) hence showing somewhat tight results in the various models at once.

Related work: Restricting the shared memory size of PRAM models has been suggested by Vishkin and Wigderson [33] that introduced the PRAM(m) model as discussed above. They proved the first lower and upper bounds for the sum and related problems (k-sensitive functions). This was later pursued by [29, 8]. Mansour et al. [27], showed that the complexity of the list reversal problem (for randomized algorithms) is $\tilde{\Theta}(\frac{n}{\sqrt{mp}})$. Comparing this to the $\tilde{\Theta}(\frac{n}{m})$ for PPRAM (Section 4.2) best exhibits the difference caused by the different input convention: Each processor in PRAM(m) can access any input in $O(1)$ time. In PPRAM this would be achieved by routing every input to the local memory of each processor. While the 'easier' input accessibility of PRAM(m) allows for faster running time and, in particular, scalability with both p and m, the 'more realistic' PPRAM is not scalable at all with p for the list-reversal problem. For PPRAM the list reversal problem is computationally 'equivalent' to a routing problem, while in PRAM(m) routing is less meaningful. This last issue further motivates research of the PPRAM model: While routing (of inputs) has no meaning at all for the PRAM(m) model, solving computational tasks often involves routing (redistributing) internal data items among the processors. At any such point, the PRAM(m) is reduced to PPRAM.

Other results related to this model are the results of [2] showing $\Theta(\frac{n \log m}{m})$ bound for sorting where multiple memory references to the same location are served using a FIFO queue (see also [1] for further work on sorting in PRAM(m)). Aggarwal et al. [4] showed upper and lower bounds for the number of shared memory references of matrix operations, sorting, and tree evaluation for the standard PRAM.

Recently Adler et al. [3] presented a related model called $QSM(m)$, which is a shared memory variant of the BSP model [19, 34]. Similar to our model, the input is partitioned between the processors, and similar to the BSP model, the computation in the $QSM(m)$ can be arbitrarily divided into super steps in which the contention is being averaged over the entire superstep. There are number of differences between the PPRAM with m cells and the $QSM(m)$: In PPRAM one may consider the various CRCW, and CREW models while the $QSM(m)$ is based on the queue-read queue-write contention rule introduced in [17]. On the other hand, the PPRAM does not support super-steps, which can be used to hide the contention in the $QSM(m)$. Similarly, PPRAM does not average the effect of the communication bandwidth (total number of shared memory references divided by m) and instead, limits the global bandwidth (using at most m cells) at every step. The focus of the corresponding studies is also different. While we are interested in communication vs. number of processors tradeoffs, scalability and speedup within the PPRAM model, the focus of the study in [3] is the relation of the $QSM(m)$ to the BSP model [19, 34], in particular with respect to global vs. local bandwidth (network throughput) limitations.

Organization: In section 2 we discuss the PPRAM self-scalability properties and the relationship to the QSM(m) model. Section 3 contains results on Boolean threshold functions. In section 4 we discuss routing related problems (including sorting and list reversal). Section

5 contains results on k-selection related problems. Some experimental results of an algorithm for matrix multiplication that is developed for the PPRAM model and is implemented on a real machine is presented in section 6. Finally, Section 7 contains some open problems and concluding remarks.

2 Comparison to QSM(m) and self-scalability

We present here some simple observations on the relation of PPRAM to other models and its self-scalability with respect to m.

In the rest of the sequel we denote by PPRAM(n, p, m) the PPRAM with p processors m memory cells and problem size of size n, with $n \geq p \geq m$. We denote by PPRAM(p, m) the same model but for which n is meaningless (or not specified).

Comparing the EREW-PPRAM(p, m) to the $QSM(m)$, one immediately observes that any algorithm in EREW-PPRAM(p, m) can be simulated on the $QSM(m)$ with the same number of processors and time. On the other hand we have:

Observation 2.1 *Let $\mathcal{A}$ be an algorithm for QSM(m) with cost t per superstep, using p processors. Assume that the pattern of memory references during a super step does not depend on the values being computed during the super step, then any one super step of $\mathcal{A}$ can be simulated on EREW-PPRAM(p, m) in at most $8t$ steps.*

Proof of Observation 2.1: The idea of the proof is quite standard and appears widely when simulating the communication step of PRAM models (e.g [13], [17]): The pattern of memory references during a super step is fixed, and can be represented by a bipartite graph $G = (P, M, E)$, where each node on the P side represents a processor, each node on the M side represents a memory cell that is referenced during the super step, and E is composed of all pairs (p, i) where the processor p reads or writes from location i.

With this notation, the definition of the time of a superstep is given by $time_{QSM} = max(d_{max}, \frac{|E|}{m})$ where d_{max} is the maximum degree of G.

To simulate a super step of $QSM(m)$ on PPRAM(p, m) a mapping of the memory of the QSM into the processors of the PPRAM will be defined below.

Assume first that $time_{QSM} = d_{max}$. We cluster the nodes of P and of M into 'super nodes' to obtain a new bipartite multi-graph on 'super nodes', $G' = (P', M', E')$ with edge set $E' = \{(p', i') | \ (p, i) \in E, \ p \in P', i \in M'\}$. We do that in such a way that the degree of each super node is in the range $[d_{max}, 2d_{max}]$, which can be done greedily in an obvious way. Note that G' may contain parallel edges.

A simple calculation shows that $|P'|, |M'| \leq \frac{|E|}{d_{max}} \leq m \leq p$. Hence, each cluster of memory cells, represented by a single node of M', can be mapped into one PPRAM processor arbitrarily. Now, by a well known Theorem of König [24, 26], the edges of G' can be decomposed into at most $2d_{max}$ matchings. Each matching represent some read/writes into some memory locations. But as $|P'|, |M'| \leq m \leq p$ it can be realized in 4 steps of PPRAM(p, m): First the processors that are in charge of the memory cells write into the m PPRAM(p, m) shared memory cells (each has one such cell). Then these processors that wish to read, read their data. In the next two steps, a write phase is simulated similarly. Hence the total of at most $2d_{max}$ matchings can be simulated within $8d_{max}$ steps.

Assume now that $time_{QSM} = \frac{|E|}{m} = \alpha \geq d_{max}$. By the same reasoning a bipartite graph G' can be defined, by clustering nodes in P and M, so that the degree of each super node is between $\lfloor \alpha \rfloor$ and $\lfloor 2\alpha \rfloor$. As before $|P'|, |M'| \leq m$ and the edges of G' can be decomposed into at most 2α matchings, hence the communication pattern can be simulated within at most 8α steps. ▮

We note that Observation 2.1 considers one super step simulation. This does not carry automatically to the simulation of an entire algorithm within the same time factor due to the fact that rearranging memory cells into processors might be required. Indeed, not every $QSM(m)$ algorithm can be simulated by EREW-PPRAM in constant slow down as shown in [17] for the QRQW model (which is identical to $QSM(m)$ with $p = m$ processors). In particular, $(2-2)$-routing can not be done in constant time in the EREW-PPRAM with m processors and m memory cells (Theorem 4.9 in this text), yet takes $O(1)$ steps in the QSM model (with $p = m$) [3].

We now state a very simple observation which addresses the self-simulation of PPRAM with respect to m, and proves to be quite handy for proving lower bounds.

Observation 2.2 *PPRAM(p, m) can be simulated deterministically by PPRAM$(p, 1)$ in a step by step manner with overhead factor of $O(m)$.*

Proof: We assume that we have at hand $p+m$ processors in the simulating machine instead of just p (simulating $m+p \leq 2p$ processors by p processors can be performed trivially with a slowdown of $O(1)$). We assume that each step of the simulated machine is divided into two sub steps: first a write step, then a read step. We assign m of the processors to be managers of the 'virtual memory' of size m. Now we just perform the operation on the m memory cells serially in m phases of 3 steps each: At phase i, $i = 1, ..., m$, the manager of the ith cell first writes its value (the old value of the ith cell) into the memory. Then, these processors that wish to write to the ith memory cell write into the cell. Finally all processors that wish to read the ith cell, along with the manager of the ith cell read the value of the cell. Note that this can be done for CREW, and the various CRCWs. ∎

An additional issue of concern is what can be said about simulating the standard PRAM on PPRAM.

Observation 2.3 *Let $c \geq 1$ be any constant then CRCW-PRAM(p, p^c) can be simulated in a step by step manner by a CRCW-PPRAM(p, m) with overhead factor of $O(\frac{p}{m} \log\log\log(p)(\log^* p)^2)$ with probability $1 - O(p^{-l})$ for any constant $l \geq 0$.*

Proof: By the Theorem of Czumaj et al. [16], CRCW-PRAM with p processors and p^c memory cells (for constant c) can be simulated by a DMM of p processors with overhead of $O(\log\log\log(p)(\log^* p)^2)$. This is done by a randomized algorithm which succeeds with probability of at least $1 - O(p^{-l})$ for any chosen constant $l > 0$. This randomized algorithm requires a suitable (somewhat 'random') partition of the input in the DMM. The DMM with p processors is equivalent to PPRAM with p processors and p memory cells. Now, the number of memory cells may be decreased to m paying p/m factor by observation 2.2. ∎

Note, for the above to be sound, we need the same input partition in the PPRAM as used by the DMM. The above theorem claims nothing for the case where worst case partition is considered.

3 Boolean threshold functions

Threshold functions are a fundamental set of simple Boolean functions: $T_k^n : \{0,1\}^n \longrightarrow \{0,1\}$ defined as, $T_k^n(x_1, ..., x_n) = 1$ if $\Sigma_1^n x_i \geq k$ and '0' otherwise. For $k = 1$ it is the OR function, for $k = n$ it is the AND function and for $k = n/2$ it is the majority function. Surprisingly enough, the complexity of Threshold functions is not exactly known for the bounded resources PRAM. We Summarize the best known results (previous to this work), for the PRAM(m) model:

- A lower bound of $\Omega(\sqrt{k/m})$ which is tight for $k = \Theta(n)$, with $p = \sqrt{nm}$ even for CREW [33].

- A lower bound of $\Omega(\frac{1}{m}(min\{k, n^{1/2-\epsilon}\}))$ and an upper bound of $O(min\{k, n^{1/2-\epsilon}\})$ for $m = 1$ with the COMMON write resolution [8].

If the number of processors is bounded by p then a trivial lower bound of $\Omega(n/p)$ is also true.

We note that the results above leave a gap between the upper and lower bounds for $m > 1$. We will show in the next subsection a tight (up to $\log n$ factors) algorithm for T_k^n in the weaker CRCW-COMMON PPRAM model. In particular this yields an algorithm of the same complexity for the PRAM(m) model, narrowing the gap above.

Fact 3.1 *Summing up n numbers each occupying one memory cell takes* $\Theta(n/p + p/m + \log m)$ *on PPRAM*(n,p,m). ▮

We remark that the upper bound holds for the weakest CREW model while the lower bound holds for the strongest PRIORITY model.

Algorithm for T_k^n in PPRAM-COMMON: We first present an algorithm that is optimal for relatively small k based on the algorithm of [11]: First the p processors scan their local input and compute the number of 1's they each have. This takes $O(n/p)$ steps. At this point each processor contains one integer, and they have to determine whether their sum is at least k. We denote this problem by $A(p,m,k)$ where p is the number of processors, m is the number of memory cells and k is the threshold. The algorithm for $A(p,m,k)$ will be recursive. In the sequel we often write quantities of the form p/m and abuse them for integers; to be exact $\lceil\rceil$, or $\lfloor\rfloor$ should be use, when necessary. This, however, will not affect the final results, and hence omitted to ease the presentation.

Algorithm $A(p,m,k)$

1. The p processors are divided into m groups of equal size. In each group, using one cell, the processors compute $A(p',1,k')$ with $p' = p/m$ and $k' = 2k/m$. This can be done in $O(k/m)$ using a trivial ARBITRARY algorithm, and in time $O(\frac{k}{m}\log\frac{p}{k})$ in the COMMON model, using the algorithm of [11, 12]: To compute $A(p',1,k')$, divide the processors into $2k'$ groups of $p'/(2k')$ processors each. In each group it is decided whether there is a positive input at all (by using one step for each group in total of $2k'$ steps). If there are k' or more groups in which the answer is positive then we are done. Otherwise, the total number of inputs decreases by half (as there are at least k' groups with no positive inputs) and we repeat the process recursively, ending if $p' \leq 2k'$ for which the sum of inputs is computed in $O(k')$ steps. It is important to note that this algorithm can be made to end with the actual sum of the inputs in the case that the sum of the inputs is below the threshold.

2. The processors compute how many of the m groups answered '1' in the first phase. This can be done trivially in $O(\log m)$. If there are more then $m/2$ such groups then the total sum is at least $\frac{m}{2}\frac{2k}{m} = k$ and thus the algorithm stops with answer '1'. Otherwise there are at least $m/2$ groups each with a sum being less than $2k/m$. The processors in these groups compute their total sum, k_0, which can be done in $O(\log m)$.

3. Let $k_1 = k - k_0$. At this point it remains to compute $A(p/2, m, k_1)$ on the input that is restricted to these processors in the groups with sum at least $\frac{2k}{m}$ (at most $p/2$ processors), which is done recursively. It needs to be seen that the remaining processors can again divide themselves into m equal groups effectively - this is a standard 'prefix sum' operation which is left for the reader to verify.

Claim 3.2 *The algorithm above takes $O(\frac{n}{p} + (\frac{k}{m}\log\frac{p}{k} + \log m)\log\frac{p}{m})$ steps for the COMMON model and $O(\frac{n}{p} + (\frac{k}{m} + \log m)\log\frac{p}{m})$ steps for the ARBITRARY model.*

Proof: The initial phase of scanning the input takes $O(n/p)$ steps. The algorithm for $A(p, m, k)$ is done recursively which gives the following recursion (disregarding multiplicative constant factors): $t(p, m, k) = \frac{k}{m}\log\frac{p}{k} + \log m + t(p/2, m, k)$ for the COMMON model and $t(p, m, k) = \frac{k}{m} + \log m + t(p/2, m, k)$ for the ARBITRARY model, with $t(m, m, k) = \log m$ for both models. This implies that $t(p, m, k) = O((\frac{k}{m}\log\frac{p}{k} + \log m)\log\frac{p}{m})$ for COMMON and $t(p, m, k) = O((\frac{k}{m} + \log m)\log\frac{p}{m})$ for ARBITRARY. Thus the whole algorithm has the claimed complexity. ▮

Theorem 3.3 *T_k^n can be computed in $O(\frac{n}{p} + min\{(\frac{k}{m}\log\frac{p}{k} + \log m)\log\frac{p}{m}, \frac{p}{m} + \log m\})$ steps on COMMON-PPRAM(n, p, m).*

Proof: Follows from the algorithm above and the algorithm of Fact 3.1.

Theorem 3.4 *The complexity of T_k^n is $\Omega(\frac{n}{p} + min(\frac{k}{m}, \frac{p}{m}))$ even for the PRIORITY model.*

Proof: A lower bound of $\frac{n}{p}$ is true from obvious reasons. We assume w.l.o.g that $k \leq n/2$ and also assume that $m = 1$. For $k \leq p$ a lower bound of $\Omega(k)$ is true, as an adversary can assign to each of the processors that succeeds writing in any of the first $k - 1$ writes exactly one '1' input without committing itself to the final result (this does not restrict any information that might be written in the first $k-1$ writes, in particular the word size may be unbounded). For $k > p$ values are assigned to all $\frac{n}{2p}$ inputs occupying the first $\frac{p}{2}$ processors as follows: '1' is assigned to all inputs $x_1, ..., x_r$, $r = k - \frac{p}{2}$ and '0' is assigned to $x_{r+1}, ..., x_{n/2}$. This reduces T_k^n to $T_{p'}^{p'}$ on the remaining $p' = \frac{p}{2}$ processors for which the previous argument holds. Using Observation 2.2 we get the desired lower bound for general m. ▮

4 Routing, Integer Sorting and related problems

Integer sorting (where the numbers are assumed to be small with respect to the number of processors) is a very well studied and important problem. Routing by itself is not an interesting computational problem in the PRAM context. However, for bounded resources models, routing (and information dispersal in general) is a basic building block for many algorithms ([7, 34, 18, 27]). In particular, as we shall see, routing and integer sorting are quite related in the PPRAM model. Our best algorithm for routing will based on an algorithm for sorting which in turn will be based on a special case of routing. In turn, the list reversal problem is a special case of routing too.

We show in this section that all these problems have running time of $\tilde{\Theta}(\frac{n}{m})$. The main technical part is to avoid logarithmic factors whenever we can, and proving matching lower bounds.

4.1 Routing and Sorting

We are interested in the routing problem that is known as the $(r-r)$-routing, or r-relation:

Definition 1 *The problem of $r-r$-routing in the PPRAM(n,p,m) is the following: each processor has $r = n/p$ data items, each with a destination, so that each processor is the destination of n/p data items. The problem is to distribute the items to their destinations. The items are considered* **atomic** *and cannot be encoded, concatenated, or take part in any computation.*

The operation of r-relation was studied in the context of the implementation of certain parallel algorithms, as well as in simulations of shared-memory models such as PRAM on distributed-memory machines [7, 34, 18, 27]. Valiant [34] showed that an efficient distributed implementation of this operation implies efficient realization of universal shared-memory models for parallel programming (such as the BSP).

We are going to use the following fixed r-relation:

Definition 2 *Consider a $r \times p$ matrix M whose columns are indexed by the processors, each contains the r elements within the processor. The $transpose(r,p)$ for $r \geq p$ is defined by looking at M as being composed of r/p blocks of sizes $p \times p$, one on top of the other, and transposing each. For $r < p$ this is defined analogously for $r \times r$ blocks.*

One basic standard feature of r-relation is that if it is fixed in advanced (in particular, every processor knows the global routing requirement) it can be implemented in $O(\frac{n}{m})$ steps:

Observation 4.1 *Any fixed, globally known r-relation can be implemented in $O(\frac{rp}{m}) = O(\frac{n}{m})$ steps.*

Proof: The observation is quite standard, it uses König Theorem [24, 26] to decompose the communication graph into r disjoint matching in a very similar way as in the proof of Observation 2.1.

Definition 3 *The problem of integer sorting in the PPRAM(n,p,m) is the following: each processor has $r = n/p$ data items in the range $\{1, ..., n^c\}$ for some constant c. At the end the numbers have to be sorted, so that each processor contains r items sorted within each processor. In addition, for every $i < j$, for every number x in p_i and every number y in p_j, $x \leq y$.*

The output order defined above is usually referred to as "column major order". Other orders might be considered as well. All such orders are computationally equivalent within complexity $O(n/m)$ as permuting one to another is just a fixed r-relation and can be realized in $O(\frac{n}{m})$ steps by Observation 4.1.

We first develop an algorithm for integer sorting. A basic building block is provided by the following lemma of Gafni and Marberg [28]

Lemma 4.2 *[28] sorting on the $s \times t$ mesh can be done in 16 phases where each phase is sorting in parallel along rows or along columns or performing a certain fixed permutation.*

Here sorting refers to rearrange the items in the mesh so that at the end the items are in the column-major order.

Theorem 4.3 *The problem of integer sorting in the EREW-PPRAM(n, p, m) can be solved in $O(\frac{n}{m} min(\log m, 2^{\frac{\log m}{\log k}}))$ steps where $k = n/m$. (Note, for $m = O(n^{1-\epsilon})$, for any fixed ϵ, the complexity is $O(\frac{n}{m})$).*

Proof: The processors divides themselves into m groups of p/m consecutive processors in each. The problem of integer sorting on (n, p, m) PPRAM is reduced to the integer sorting on (n, m, m) PPRAM in $O(n/m)$ steps: Using one cell for each of the m groups, in parallel, all data of each group is transferred to one representative processor. This is done in $O(n/m)$ steps. Then the sort is solved for the remaining (n, m, m) problem. In the end each representative, using one cell, sends the appropriate n/p items to the appropriate processor for each of the p/m processors in the group. This again is done in $O(n/m)$ steps. Hence the proof follows from Lemma 4.4 and Lemma 4.6 below. ∎

Lemma 4.4 *Let $\frac{n}{m} \geq m/2$, then integer sort in the EREW-PPRAM(n, m, m) can be solved in $O(\frac{n}{m})$ steps.*

Proof: Assume first that $\frac{n}{m} = k \geq m$. We think of the items as being arranged in an array of $k \times m$ where each processor represents a column. By Lemma 4.2 this array can be sorted by 16 phases, each being either sorting columns or rows in parallel, or performing a certain fixed permutation. Any fixed permutation is a fixed k relation and can be done in $O(n/m)$ steps by Observation 4.1. Sorting the columns in parallel is done by any linear time algorithm for integer sorting within each processor in $O(k) = O(n/m)$ steps. To sort rows, we first do $transpose(k, m)$. This is a pre-defined k-relation and can be done in $O(n/m)$ steps by Observation 4.1. After transposition, each column (processor) contains k/m old rows (we assume here w.l.o.g that k/m is integer). Hence, all rows can be sorted within the processors in $O(k) = O(\frac{n}{m})$ steps (again using any linear time integer sort). At this point, another transpose is done which brings the (now sorted) rows back to their original places. Thus every phase and also the total time takes $O(n/m)$ steps.

Assume now that $m/2 \leq k < m$. The same algorithm as above may be applied except that a row might occupy two processors. Then to sort rows, pairs of processors corresponding to rows inform each other of the entire row and sort it within the processor in $O(k)$ steps. ∎

The difficulty with $k << m$ is that rows now are too big to fit within a column, hence we cannot sort rows this efficiently. To handle this case, we present another sorting algorithm (that also may be thought as sorting a rectangular array). The high level algorithm is a generalization of the odd-even merge sort of Batcher [10, 31, 22] and will be presented here in its abstract form:

Algorithm Sort-Array(k,m), for $k < m/2$:
The following is an algorithm for sorting k lists of size m (each) into one sorted list (also depicted in figure 1); hereafter we think of the k lists as being arranged in a $k \times m$ array, at the end the sorted list will be read by taking the first column, then the next and so on, namely, the sort will correspond to column major order as defined before.

1. Each list (row) is sorted.

2. A $transpose(k, m)$ is performed on the $k \times m$ array. This distributes the elements of each list (row) in order, equally among all other rows.

3. Each row is being sorted again.

4. We now view each list as concatenated from $\frac{m}{2k}$ sublists of size $2k$ each. In turn this defines a decomposition of the $k \times m$ array into $\frac{m}{2k}$ blocks of size $k \times 2k$ each. Each of these blocks is sorted separately in parallel.

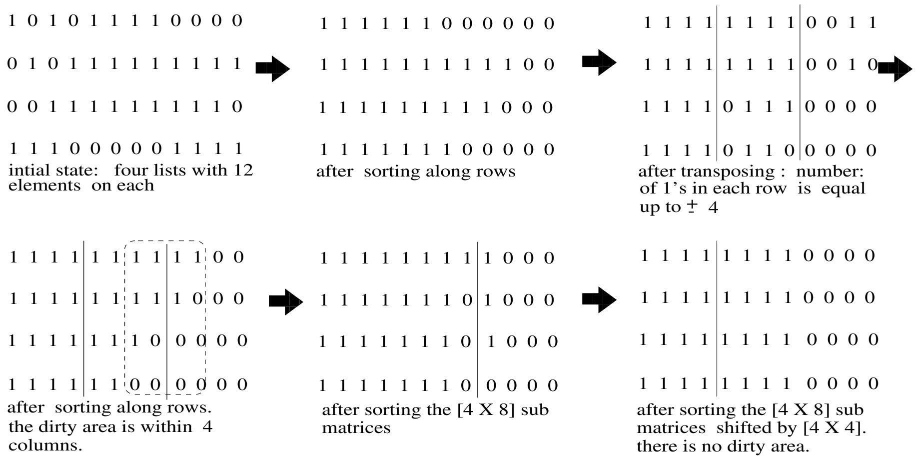

Figure 1: *Different stages of the algorithm for sorting k lists.*

5. We now view the array as composed of a first left block of size $k \times k$ followed again by disjoint blocks of size $k \times 2k$. Each of these blocks is being sorted (including the first one).

Lemma 4.5 *At the end of algorithm Sort-Array(k,m), the $(k \times m)$-array is sorted (in column major order).*

Proof: The proof of this fact is a generalization of the proof of Batcher sort, observing that the algorithm above is oblivious (see [31]) and using the 0-1 principle. ∎

Lemma 4.6 *Let $\frac{n}{m} = k < m/2$, then integer sorting can be done in $O(\frac{n}{m} min\{\log m, 2^{\frac{\log m}{\log k}}\})$ steps in EREW-PPRAM(n, m, m).*

Proof: An adaptation of algorithm Sort-Array(k,m) to PPRAM is straight forwards except for phases 1 and 3 which requires sorting each row efficiently. Indeed phase 2 is just a fix permutation and takes $O(n/m)$ steps (by Observation 4.1). Phases 4,5 are sorting in parallel the sub arrays of sizes $k \times 2k$ and can be done in $O(n/m)$ steps by Lemma 4.4.

To sort the rows we use two different algorithms depending whether $\log m \leq 2^{\frac{\log m}{\log k}}$ or not. We can always use any $O(\log m)$-parallel algorithm that sort m elements using m cells (e.g [5]) to sort one row. Thus sorting the k rows will result in $O(k \log m) = O(\frac{n}{m} \log m)$ many steps. If $\log m \leq 2^{\frac{\log m}{\log k}}$, this is the preferred way which contributes the first term in the claimed complexity.

If $\log m > 2^{\frac{\log m}{\log k}}$, a more efficient way to sort all rows is the following: First a $transpose(k, m)$ is performed so that each row occupies a $k \times \frac{m}{k}$ sub array. Then these sub arrays are sorted, each with m/k processors, using m/k cells, recursively. Then the array is transposed back to an array of which each row is sorted. Hence in this case the whole algorithm become recursive. Let $t(k, m, m)$ be the time it takes to integer sort $n = km$ elements, with m processors each containing k elements, using m cells. By the above algorithm we get the recursion: $t(k, m, m) \leq O(k) + 2t(k, \frac{m}{k}, \frac{m}{k})$ where the first term comes from the transpose operations and the final sorting of $k \times 2k$ sub arrays. The second term comes from the two calls for sorting rows which is done by recursive sorting sub arrays. The recursion ends at iteration i for which $m_i = \frac{m}{k^i} < k$ in which we are at the situation of Lemma 4.4 and thus is done in $O(k)$ steps. Solving the recursion yields $t(k, m, m) = O(k \cdot 2^{\frac{\log m}{\log k}})$ which contributes the other term in the statement. ▮

Theorem 4.7 *Let $k = \frac{n}{m}$ then $(r - r)$-routing can be done in $O(\frac{n}{m} min\{\log m, 2^{\frac{\log m}{\log k}}\})$ steps in EREW-PPRAM(n, p, m).*

Proof: Immediately follows from Theorem 4.3 as routing can be reduced to sorting the items by destinations in an obvious way. ▮

Theorem 4.8 *$(r - r)$-routing requires $\Omega(n/m)$ steps even for PRIORITY-PPRAM.*

Proof Of Theorem 4.8: Follows from Theorem 4.13 as list reversal is a special case of $(r - r)$-routing. ▮

We do not know whether the (possible) logarithmic factor multiplying $\frac{n}{m}$ in Theorem 4.7 is essential, however, rephrasing a result of [18] to our context yields:

Theorem 4.9 *[18] $(2 - 2)$-routing for EREW-PPRAM with $m = p$ requires $\Omega(\sqrt{\log \log n})$ steps on the average even for randomized algorithms.*

4.2 List reversal

The list reversal problem is the following: The input is a permutation π on n elements given as n pairs $\{(1, \pi(1)), (2, \pi(2)), \ldots, (n, \pi(n))\}$. In the PPRAM model the n pairs are distributed in an arbitrary way between the processors so that each processor has $k = n/p$ pairs. The goal is that P_i will know some n/p fixed predetermined pre-images, e.g to be specific, we require that for all $i = 1 \ldots p$, P_i will know $\pi^{-1}(j)$ for all j such that $(i - 1)\frac{n}{p} < j \leq i\frac{n}{p}$.

We investigate the list reversal problem as it reveals an essential difference between the PRAM(m) and the PPRAM. In the former [27] determined the randomized complexity (up to $poly(\log n)$ factors) to be $\tilde{\Theta}(\frac{n}{\sqrt{pm}})$ (for word size $\Theta(\log n)$). Here we determine its exact complexity for the PPRAM model in the following Theorems. We note here that in particular, the results depend crucially on the word size. We are interested mainly, as in [27], in word size of $\Theta(\log n)$. We compare this with the case of unbounded word size. We will also treat 1-bit word size for technical reasons. We get the following, nearly tight, results.

Theorem 4.10 *List reversal can be solved in $O(n/m+\log m)$ steps on EREW-PPRAM(n,p,m) with word size of $\Theta(\log n)$.*

Theorem 4.11 *Solving the list reversal problem on PPRAM(n,p,m) with word size $\Theta(\log n)$ requires $\Omega(n/m)$ steps even for the PRIORITY model.*

Proof of Theorem 4.10: The list reversal problem for PPRAM(n, p, m) can be reduced to that on PPRAM(n, m, m) with an additional cost of $O(\frac{n}{m})$. This is done by grouping the p processors into m groups and collecting within each group all data into one processor. Hence from now on we concentrate on the (n, m, m) case.

The (n, m, m) list-reversal is a special case of a $k = n/m$ relation, namely a processor containing a pair $(i, \pi(i) = j)$ has to send to the processor that must hold $\pi^{-1}(j)$ the value $\pi^{-1}(j) = i$. Hence every input pair $(i, \pi(i) = j)$ defines a 'destination' processor which is the processor that must hold i as the answer and is called the destination of i. Therefore for $m < k$ the result follows directly from Theorem 4.7. Using the same algorithm for the case of $m > k$ we may get a possible logarithmic factor. To avoid this we use a deterministic adaptation of the load balancing ideas of [13] that is described in the following Lemma.

Lemma 4.12 *Let $k = \frac{n}{m} < m/2$, then list reversal for EREW-PPRAM(n, m, m) can be solved in $O(n/m + \log m)$ steps.*

Proof: We color the set $\{1, ..., n\}$ by k colors in correspondence with the list-reversal definitions so that at the end each processor will contain one element of each color. To be specific we color the data $(i, \pi(i) = j)$ by $j\ mod(k)$. Thus each color class has size m. Note that each processor is the destination of k distinct colors. Assume that in addition, we are guaranteed that each processor has items of distinct colors. In that case, the list reversal would trivially take $2k$ steps: At each pair of steps only items of color i are being treated; at odd steps processor holding an item of color i writes it in the cell that corresponds to the destination processor. In the even steps, processors read from their cells their values.

The same idea carries on even if the guarantee is somewhat relaxed: Assume that each processor contains at most 3 items of the same color. Again the whole problem then would be solved in $4k$ steps: In the first three steps of each phase, processors write their value (of color i) in the proper cells, while at the 4th step processors read their values.

In what follows, our aim is to redistribute the items so to achieve the guarantee as before. We now describe the rest of the algorithm:

1. We divide the m processors into m/k groups, each containing k processors.

2. In each group of k processors the elements are sorted by their color using the k cells. This can be done in $O(k)$ steps (Lemma 4.4).

3. The elements of the same color are grouped together in each processor and are represented by an abstract data item colored with the corresponding color and the processor id. As there are at most k colors it is easy to see that at most $2k$ abstract items may be created in each group of k processors. At this point, each abstract item is associated with the processor it resides in (called the 'origin').

4. Within each group of k processors, the (at most $2k$) abstract items are redistributed so that each processor contains at most 2 items. This can be trivially done in $O(k)$ steps by letting the k processors expose their abstract items, one after the other using just one cell per group.

5. At this point, each of the m processors contains at most 2 abstract items. The whole set of abstract items is sorted by color using the m cells in $O(\log m)$ steps by any optimal sorting algorithms (e.g [5]), (Theorem 4.3 also gives this time bound).

6. Here we assume for a moment that we have two sets of $p' = m$ processors each. One that represents the set of processors of phase 3 with the abstract items that they are the origin of, and the other represents the set of processors at the end of the previous phase, each containing two abstract items which contain the origin name.

   Now each origin processor, using a memory location designated to it, scans through the abstract items it originates (in step 3 above), and for each, it writes the true inputs that it represents. As it contains a total of k true elements, this can be done in k steps. This is repeated twice, namely for $2k$ steps. At the same time, each processor of the second group, for its first abstract item - reads the content of the cell designated with the known origin for each of the first k steps, till it finds the true items belonging to the first abstract item it contains. It repeats the process for the second abstract item in the next k steps.

   At the end, each processor of the second group has the $2k$ items originally associated with the two abstract item it has. Note also that since all the abstract items were sorted by color, so are the true items at this point.

   Now, this could be done in $2k$ steps with $2m$ processors. As we really have only m processors, each playing its part in each of the groups, this will result in a slow down by a factor of 4 in a straight forwards manner.

7. At this point each processor contains at most $2k$ true items. In addition, items are being globally sorted by colors so that the first processor contains the 'small' colors, the next processor contains higher colors etc. We assume that each processor contains exactly $2k$ elements (by inserting dummy elements if necessary) and perform $transpose(2k, m)$ in $O(k)$ steps (Observation 4.1).

   As the items were sorted by colors before the transpose, it is well known (and easy to observe) that after the transpose, each color class is nearly evenly distributed among each processor, namely, each processor contains at most 3 items of each color. At this point we have achieved the guarantee assumption that we were looking for.

The algorithm above takes $O(k + \log m)$ steps, where the $\log m$ term comes from the sorting in phase 5. We note that in [13] a randomized algorithm (for a slightly different model) is developed by replacing the sort phase by randomized "padding-sort" or "semi-sort" [9], achieving the complexity of $O(k + \log^* m)$. This can be done here too, however, as we do not consider randomized algorithms in this work, we will not comment further on this direction. ∎

Proof of Theorem 4.11: Follows directly from Theorem 4.13 below, Observation 2.2 together with the following observation: PRIORITY-PPRAM with one k-bit cell can be obviously simulated in a step-by-step manner by PRIORITY-PPRAM with one 1-bit cell and slow down of k. ∎

Theorem 4.13 *Any algorithm for the list reversal problem for PRIORITY-PPRAM$(n, p, 1)$ with word size of $O(1)$ requires $\Omega(n \log n)$ steps to end.*

Proof: Consider the following two players communication complexity problem (see [25] for a comprehensive ref. on communication complexity). The input is a permutation π on n elements given as n pairs $\{(1, \pi(1)), (2, \pi(2)), ..., (n, \pi(n))\}$. The first player gets the first $n/2$ pairs and the second player get the rest. The goal is that the first player will know $\pi^{-1}(j)$ for all $j \leq n/2$ and second player has to know $\pi^{-1}(j)$ for all $j > n/2$. The complexity is the total number of bits that have to be transmitted by the players in order to accomplish the task. The players are assumed to be computationally unbounded.

It is quite easy to see that the communication problem above can be reduced to the list-reversal in PPRAM$(n, p, 1)$ with 1-bit word size: Assume that there is an algorithm A that solve the list-reversal problem for the PPRAM$(n, p, 1)$ within t steps, then a protocol that solves the communication problem with total communication at most t is obtained as follows: the first player simulates the actions of A on the first $p/2$ processors, and the second player simulates the actions of A on the rest of the processors. Hence, a lower bound for the communication complexity problem yields the same bound for the relevant PPRAM model. Indeed, a lower bound of $\Omega(n \log n)$ for the communication complexity problem is easy to prove due to a standard adversary argument: Assume that the partition of the input is so that the first player gets all pairs $(i, \pi(i))$ with $i > n/2$ and focus on the second player: To start with, he has any of the $(\frac{n}{2})!$ possible outputs, thus it takes $\Omega(n \log n)$ bits in order to restrict this set to the unique correct output. ∎

4.3 List reversal with unbounded word size

The case of unbounded word size is interesting from the theoretical point of view as it exhibits a different behaviour then in the bounded case. This model might have its own practical interest in the scenario in which the bandwidth of the channel is very high (e.g processors that are hooked together via a fast bus, optical device or radio communication), or when there is a significant overhead for creating a message, and thus one would attempt to use big messages.

Theorem 4.14 *The list reversal problem can be solved in time $O(\frac{n}{p} + \frac{p}{m} + \log m)$ on a CREW-PPRAM(n, p, m) with word size $\Omega(n \log n)$.*

Proof: Straight forwards by compressing all the information into one word. ∎

Theorem 4.15 *Any algorithm for the list reversal problem in PRIORITY-PPRAM(n, p, m) with unbounded word size takes at least $\Omega(p/m + n/p)$ steps.*

Proof: The n/p term is obvious. We prove a lower bound for PPRAM$(p, p, 1)$ of $\Omega(p)$ which implies the claim of the Theorem. Assume that the permutation π is cyclic. An adversary argument works as follows: Divide the p processor into $p/4$ groups of 4 processors each, and with a fixed cyclic order among the $p/4$ groups. In each group of 4 processors fix two of them: One that will be called the 'first' and the other the 'last'. The other two will be called the 'free' processors of the group. The adversary commits herself to inputs that are cyclic consistent with the fixed order of the groups and so that the 'last' processor of the ith group will be pointed to the 'first' processor of the $(i+1)th$ group. With all this known to all the processors, there are $2^{p/4}$ possible inputs; namely fixing the order between the two 'free' processors within each group. We prove by induction, that after the ith step of the algorithm, $0 \leq i \leq \frac{p}{4} - 1$ there are some $k \geq \frac{p}{4} - i$ groups, so that any order of the 'free' processors within these k groups is consistent with the i first steps of the algorithm.

Proving this will prove the theorem, as for $i < \frac{p}{4}$ there is at least $k \geq 1$ groups for which any order among the 'free' processor is still consistent. This means that the list reversal can not be determined yet, as in any of the above groups, the 'last' processor has yet two possible processors that might be pointed to it.

To prove the above inductive claim for the base case, note that after the 0th step (before any step) all $2^{p/4}$ inputs are consistent.

Assume that after the ith step there are $k \geq \frac{p}{4} - i$ groups for which any order among the 'free' processors is consistent. Let us look at the $(i+1)$th step, if there is any attempt of any processor within these k groups to write (conditioned on his input), the adversary will fix the input for the highest priority processor among all these that wish to write, making him the one that will write, and in addition will let everybody know the internal order within the group of the processor that wrote. Clearly that leaves a set of $k-1$ groups for which any order of the 'free' processors within these groups is still consistent with the last step and hence with all previous steps. If there is no processor within the above k groups that attempts to write on the $(i+1)$th step then after this step any order of the 'free' processors within these k groups is still consistent. ∎

5 The k-selection problem

The k-selection problem is the following: Given n inputs from a totally ordered set, find the input which is bigger (or equal) than $k-1$ of the inputs and smaller (or equal) than the rest.

The k-selection problem has been well studied in different models of computation, as well as in various resource bounded PRAMs (e.g [30], [22], [6], [21]). It is reducible to sorting, and it contains the problem of finding the minimum, maximum and median. It also contains T_k^n as a special case. As will be shown below, all these problems can be solved on COMMON-PPRAM(n,p,m) in $\tilde{\Theta}(\frac{n}{p} + \frac{p}{m})$ steps. Some special cases (of inputs in a bounded range) are discussed too.

5.1 The MIN problem

The MIN problem was investigated in bounded resource PRAMs by Shiloach and Vishkin [32]. They showed an $O(\log\log n)$ algorithm for CRCW-PRAM with $O(n/\log\log n)$ processors. In the PPRAM model it is trivial that one can get an $O(n/p + p/m + \log\log\frac{m^2}{p})$ (by using the Shiloach-Vishkin algorithm for the final stage after reducing the problem to size m and remembering that for $p \geq m^2$ the MIN for m values takes $O(1)$ steps). A more interesting case is when there is an a priori bound on the size of an interval that contains all the values:

Theorem 5.1 *The MIN problem of n integral values in the range* $\{r+1, ..., r+R\}$ *can be solved in* $O(\frac{n}{p} + min\{\frac{p}{m} + \log\log\frac{m^2}{p}, \ \frac{\log R}{\log m}\})$ *steps on COMMON-PPRAM*(n, p, m).

Theorem 5.2 *Solving the MIN problem of n values in the range* $\{r+1, ..., r+R\}$ *requires* $\Omega(\frac{n}{p} + min\{\frac{\log R}{\log m}, \frac{p}{m}\})$ *steps for ARBITRARY-PPRAM*(n, p, m).

Proof of Theorem 5.1: The following algorithm solves the MIN problem: Each processor computes the minimum of its n/p values which reduces the problem to finding the minimum of p values. Then to get the $p/m + \log\log\frac{m^2}{p}$ term one does not exploit the range restriction and just uses the algorithm for the general case. To exploit the range restriction we split the range $\{r+1, ..., r+R\}$ into m intervals of equal sizes, and allocate one memory cell, initialized to '0', for each such interval. Each processor writes '1' into the cell that corresponds to the interval that contains its value. As a result, the minimum is in the interval that corresponds to the smallest indexed cell that contains the value '1'. Finding this cell cab be done in $O(1)$ steps using the idea of Vishkin-Wigderson [33] (it can be done trivially with m^2 processors using m cells. For m processors it is reduced in two steps to a problem of size $\sqrt{m}$ for which again the number of processors available is quadratic). After locating the appropriate interval, the problem is solved recursively within this subinterval, yielding the stated upper bound. ∎

Proof of Theorem 5.2: The n/p term is trivial. Now assume that there are p values, one in each processor. Let $Min(R, p)$ denote the problem of finding the minimum where there are p processors each with one integer input in the range $\{r+1, ..., r+R\}$ for some fixed known r. The proof is formally by induction on $(R+p)$, assuming also that the lower bound holds even when the array of m cells is initialized in an arbitrary manner (but independent of the inputs). We assume w.l.o.g that R is divisible by $m+1$. We show that if there is an algorithm for $Min(R, p)$ in t steps then we can construct an algorithm for $Min(\frac{R}{m+1}, p-m)$ in $t-1$ steps. This in turn yields the relation $t(R, p) \geq 1 + t(\frac{R}{m+1}, p-m)$ where $t()$ is the complexity of the appropriate problem. Realizing that $t(m, m) \geq 1$ yields the result.

Indeed, let A be an algorithm that solves $Min(R, p)$. The argument involves an adversary argument: The adversary divides the range of size R into $m+1$ consecutive equal parts, $R_0, R_1, .., R_m$ each of size $R/(m+1)$. Its aim is that after the first step of the algorithm, it will fix a set P' of $p-m$ processors that didn't write, and one of the subranges $R_i \in \{R_0, ...R_m\}$, such that all the values in R_i are still consistent as inputs for these $p-m$ processors, and so that the minimum of the original problem is the minimum of these values. Assuming one can do this fixing a subrange R_i, then there would be an algorithm for $Min((\frac{R}{m+1}, p-m)$

in $t-1$ steps: The $p-m$ processors play the role of the fixed set P' of $p-m$ processors found by the adversary for A, the inputs to $Min((\frac{R}{m+1}, p-m)$ which are in some range $\{r_0+1, ..., r_0+R/(m+1)\}$ are mapped naturally into R'. In addition, as some m of the processors of A may write some values to the memory, we assume that the memory is initialized to contain these values. Now they run the algorithm A while skipping the first step, as it is known to everybody and is independent of the input to $Min(\frac{R}{m+1}, p-m)$. By our assumptions as guaranteed by the adversary choices, the input will be found at the end of the run taking $t-1$ steps.

We now describe how the adversary chooses P' and R_i. Looking at the first step of the algorithm A, each processor may attempt to write at any cell on some values. The adversary will try to fix values for at most m processors and force them to write into the m cells while keeping the invariant that the minimum still can be any value of a given subrange of size $R/(m+1)$. This is done as follows: The adversary passes form the upper range R_m down to R_1 trying to assign a value from each subrange to one processor and so that these m processors will write into different m cells. Once it succeeds, the set P' will be all other processors, and the subrange untouched yet that still might contain the minimum is R_0. Formally, assume that there is a processor that writes into a cell for some value in the upper range R_m then the adversary assigns this value to the processor and lets him succeed in writing. It then moves to R_{m-1} choosing a processor and a value to be written in a different cell and so on. If it indeed succeeds all the way, we are done.

Otherwise, assume that while doing the process above there is an $i \leq m$ for which no processor that is not assigned a value yet, that attempts to write to an available cell on any value in the range R_i. Remember also that so far the adversary committed itself to values only in subranges $R_j,\ j > i$. We set P' to be all the available processors so far (excluding some that will get the value $r+R$ just to reduce P' to be exactly of size $p-m$), and the subrange still untouched and consistent is R_i. This completes the description of the adversary strategy and concludes the proof. ∎

We leave open here the exact complexity of the bounded range MIN problem for the PRIORITY write resolution model.

5.2 k-selection for general k

For very small (large) k's, the solution for the minimum gives a trivial algorithms by repeatedly finding the minimum k times. For general k, the best result we know (disregarding $\log n$ factors) is $\tilde{O}(n/p + p/m)$. This can be done by the parallelization of the $O(n)$ serial algorithm for the median. We note that k-selection can obviously be reduced to sorting, however, sorting takes $\tilde{\Theta}(\frac{n}{m})$ in the PPRAM model (see Section 4.1). It is interesting to note that while in the general PRAM model both these problems are well parallelized, are similar in nature and their complexity differs only by a logarithmic factor even in the sequential model, here they behave quite differently. We sketch the algorithm for k-selection, although it is a rather straightforward adaptation of the sequential algorithm, for the sake of being self-contained.

We first start with the basic building block which is an algorithm for the PPRAM(mr, m, m) namely, there are $p' = m$ processors each having r values.

Fact 5.3 *Finding the kth element can be done in $O(r \log r + \log^2 m)$ steps on PPRAM(mr, m, m).*

Proof: The following is an adaptation of the liner time sequential algorithm, [14]:

1. Each processor internally sorts his r elements (in $O(r \log r)$ steps).
2. repeat while total number of elements is at least $\log m$,
   (i) Each processor finds the median among its elements ($O(\log r)$ steps).
   (ii) Globally a 'splitter' which is the median of the m medians is found. This is done using the m cells, by sorting in parallel (in $O(\log m)$ steps).
   (iii) Each processor splits its relevant set of inputs into these elements that are larger than, and small than the splitter ($O(\log r)$ steps).
   (iv) The number of elements that are larger than the global splitter is computed. This is just computing a sum which can be done trivially in $O(\log m)$ steps. Then each processor proceeds to next step with the relevant local set (either these elements that are larger or these that are smaller than the global splitter, according to where the final result must be).
3. If the number of items drops below $\log m$ the right element is found, trivially, using one cell in $O(\log m)$ time.

As at each step the total number of elements is dropped down by a constant fraction (see [14]), the total number of iterations of the repeat loop above is $O(\log(rm))$ yielding a total of $O(r \log r + \log^2 m)$ for the algorithm.

The algorithm for the general case, based on the same idea gives:

Theorem 5.4 *Finding the kth element can be done in $O((\frac{n}{p} + \frac{p}{m} \log \frac{p}{m} + \log^2 m) \log n)$ steps on PPRAM(n, p, m).*

Proof: Again we parallelize the classical recursive algorithm of [14], in a very similar way.

1. Each processor internally sorts his n/p elements (in $O(\frac{n}{p}\log\frac{n}{p})$ steps).

2. repeat while total number of elements is at least $\log m$,

   (i) Each processor finds the median among its elements ($O(\log\frac{n}{p})$ steps).

   (ii) To find a splitter (a median of the medians), the p processors group themselves into m group of p/m processor in each, and with a designated processor per group. Using one cell, the designated processor in each group collects the medians from its group members (in $O(\frac{p}{m})$ steps).

   (iii) Now the median of the medians can be computed by the m designated processors, using Fact 5.3 with $r = \frac{p}{m}$.

   (iv) Each processor splits its relevant set of inputs (to either the elements that are larger than, or small than the splitter). This is done in $O(\log\frac{n}{p})$ steps as the inputs are sorted.

   (v) The number of elements that are larger than the global splitter is computed. This is just computing a sum which can be done in $O(\frac{n}{p} + \frac{p}{m} + \log m)$. Then each processor proceeds to next step with the relevant local set (either these elements that are larger or these that are smaller than the global splitter, according to where the final result must be).

3. If the number of items drops below $\log m$ the answer is found, trivially, using one cell in $O(\log m)$ time.

By the standard analysis of the algorithm of [14], the repeat loop can be performed at most $O(\log n)$ times, from which the total complexity follows. ∎

6 Experiments with Matrix multiplication

For algorithms that theoretically scale with both p and m but not with both at the same time (as in the $\frac{n}{p} + \frac{p}{m}$ behaviour), it would be nice to see an implementation for which such a behaviour is witnessed (even though the real machine + software has no explicit m in its description). Here we present such an attempt. We implemented a simple matrix multiplication PPRAM algorithm (based on the 'naive' matrix multiplication). We compiled it using HPF (high performance Fortran) which gives a 'shared memory' model as an interface for the user, and run it on the IBM SP2 parallel machine with fixed number of processors, varying m. We chose to check matrix multiplication using the HPF compiler as the HPF has special feature to detect and optimize naive matrix multiplication [20]. In turn this allows us to compare the naive matrix multiplication (optimized by the HPF complier) and the PPRAM version (optimized for best m). It is interesting to note that our algorithm (optimized for best m) runs, in some cases, slightly faster than the optimized 'naive' matrix multiplication. Note that we don't try to compete with any sophisticated matrix multiplication algorithm. Our aim is to demonstrate that by taking a fixed algorithmic idea (in this case the naive matrix multiplication) and designing it as a PPRAM algorithm, it might yield an implementation that competes well with the result of optimized compiler (designed for the actual hardware parameters).

We chose to design the algorithm in EREW-PPRAM as the SP2 does not support true broadcast. We use a well known technique suggested by Aggarwal et al. [4] for reducing shared memory references in $n \times n$ matrix multiplication $C = A \cdot B$ (also used in BSP matrix multiplications). We use here the following partition of the inputs: The input matrices, A and B, are partitioned into $\sqrt{p} \times \sqrt{p}$ blocks of size $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$, denoted by $A_{i,j}, B_{i,j}$ $(i, j = 1, \ldots, \sqrt{p})$. We also index the processors by $(i, j) \in \{1, ... \sqrt{p}\} \times \{1, ... \sqrt{p}\}$. Then the (i, j) processor gets as input $A_{i,j}$ and $B_{i,j}$. Let the output matrix C be also divided into blocks in the same manner, then by the definition of matrix multiplication $C_{i,j} = \Sigma_{k=1}^{\sqrt{p}} A_{i,k} \times B_{k,j}$. At the end of the algorithm processor (i, j) will have the block $C_{i,j}$.

Now the algorithm is as follows: each processor (i, j) gets all blocks $A_{i,k}, B_{k,j}$, $k = 1, ..., \sqrt{p}$ and computes internally its result $C_{i,j}$. Sending the inputs is just a globally known $2\sqrt{p} \cdot \frac{n^2}{p}$-relation and hence can be done in $O(\frac{n^2 \cdot \sqrt{p}}{m})$ steps. The internal computation is done by the naive serial matrix multiplication in $O(\frac{n^3}{p})$ resulting in $O(n^2 \cdot (\frac{n}{p} + \frac{\sqrt{p}}{m}))$ steps.

The naive HPF code for matrix multiplication has three nested loops:

$$
\begin{array}{l}
for\ i = 1 \ldots n\ do \\
\quad for\ j = 1 \ldots n\ do \\
\quad\quad for\ k = 1 \ldots n\ do\ C[i,j] = C[i,j] + A[i,k] \cdot B[k,j];
\end{array}
.
$$

The HPF compiler optimally partitions these into blocks and allocates them to different processors. The matrices are also partitioned between the processors and the communication is minimized to the messages needed to distribute $B[,]$ to the suitable processors. We used $p = 16$ processors in all the experiments. There are two communication media in SP2: a standard Ethernet line connecting all the processors called 'ip' and a low diameter communication network with special protocol called 'us'. The two curves in Figure 1 describe the execution time of the PPRAM algorithm for 'ip' and 'us', as a function of m, for multiplying 100×100 matrices. The results shows that for our algorithm optimal values were obtained for $m = 8$ achieving 0.1062 time units for 'ip' communication medium and 0.0527 times units

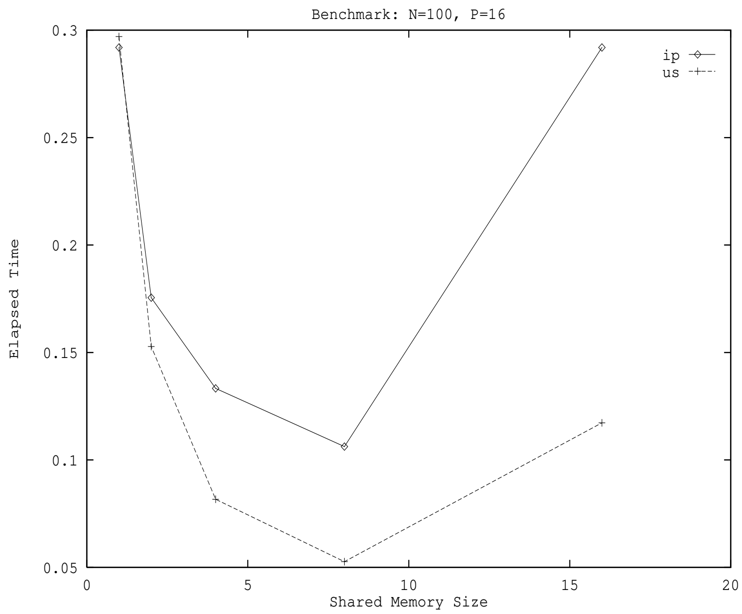

Figure 2: *Results of matrix multiplication.*

for 'us'. The naive version of matrix multiplication, obtained 0.082 times units with 'us' and 0.07 time units with 'ip'.

7 Concluding remarks and open problems

We have demonstrated in Sections 4, 5, the existence of problems with two different types of complexity behaviour. The one, which was typical to routing related problems was $\tilde{\Theta}(\frac{n}{m})$. The other type is $\tilde{\Theta}(\frac{n}{p}+\frac{P}{m})$, namely it exhibits different behaviour according to the relative sizes of n, p, m; there is one region in which there is scalability in p but not in m and the other in which there is scalability in m but not in p. In both types of behaviour one would expect to see non-scalability effects in small intervals of parameters. This might offer some explanation to why in real life situations, one fails to see scalability for some regions of bandwidth and number of processors. One open question that is interesting in this respect, is whether there is a more complex type of behaviour for a 'natural' $\tilde{\Theta}(n)$ sequential problems (e.g that scales with both m and p for a 'wide' range of parameters).

Another open question is to make the results tight (in many cases there is a gap of logarithmic factor between the upper bound and the lower bound). A related question is to better understand the particular concurrent read/write conventions, as some times the above gaps are possibly due to these different conventions.

References

[1] M. Adler. New coding techniques for improved bandwidth utilization. In *Proc. 37th IEEE Symp. Foundations of Computer Science*, pages 173–182, 1996.

[2] M. Adler, J. Byers, and R. Karp. Parallel sorting with limited bandwidth. In *Proc. 7th ACM Symp. on Parallel Algorithms and Architectures*, pages 129–136, 1995.

[3] M. Adler, P. B. Gibbons, Y. Matias, and V. Ramachandran. Modeling parallel bandwidth: local vs. global restriction. *Algorithmica, special issue on coarse-grained parallel algorithms*, (3-4):381404, 1999.

[4] A. Aggarwal, A. K. Chandra, and M. Snir. Communication complexity of PRAMs. *Theoretical Comput. Sci.*, 71(1):3–28, Mar 1990.

[5] M. Ajtai, J. Komlós, and E. Szemerédi. An $O(n \log n)$ sorting network. *Combinatorica*, 3:1–19, 1983.

[6] S. G. Akl. The design and analysis of parallel algorithms. *Prentice-Hall Int.*, 1989.

[7] R.J. Anderson and G.L. Miller. Optical communication for pointer based algorithms. Technical Report CRI 88-14, Comp. Sci. Dept., USC, 1988.

[8] Y. Azar. Lower bounds for threshold and symmetric functions in parallel algorithms. *SIAM J.Comput*, 21(2):329–338, 1992.

[9] H. Bast and T. Hagerup. Fast parallel space allocation, estimation and integer sorting. *Information and Computation*, 123(1):72–110, November 1995.

[10] K. E. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Comput. Conf.*, pages 307–314, 1968.

[11] Y. Ben-Asher and I. Newman. Decision trees with boolean threshold queries. *Journal of Computer and System Sciences*, 6(1):495–502, 1995.

[12] Y.Ben-Asher and I.Newman. Geometric approach for optimal routing on mesh with buses. *Journal of Computer and System Sciences*, 54(3):475–486, 1997.

[13] P. Berthomé, T. Duboux, T. Hagerup, I. Newman, and A. Schuster. Self-simulation for the passive optical star model. *J. of Algorithms*, 34(1):128-147, 2000.

[14] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *J. of Computer and System Sciences*, 7(4):448–461, 1973.

[15] R. Cole. Parallel merge sort. *SIAM J. Comput.*, 17(4):770–785, 1988.

[16] A. Czumaj, F. Meyer auf der Heide, and V. Stemann. Contention resolution in hashing based shared memory simulations. *SIAM J. Comp.*, 29(5):1703–1739, 2000.

[17] P. B. Gibbons, Y. Matias, and V. Ramachandran. The queue-read queue-write pram model: Accounting for contention in parallel algorithms. *SIAM J. on Computing*, 28(2):733–769, 1998.

[18] L.A. Goldberg, M. Jerrum, and P.D. MacKenzie. An $\Omega(\sqrt{\log\log n})$ Lower Bound for Routing in Optical Networks. *SIAM Journal on Computing*, 27(4):1083–1098, 1998.

[19] S. Gotz. Algorithms in cgm, bsp, and *bsp** models - a survey. *Carleton University, Ottawa*, 1996.

[20] M. Gupta, S. Midkiff, E. Schonberg, V. Seshadri, D. Shields, K. Wang, W. Ching and T. Ngo. An HPF Compiler for the IBM SP2. *Proc. of Supercomputing '95*, San Diego, CA, November 1995.

[21] E. Hao, P. D. MacKenzie, and Q. F. Stout. Selection on the reconfigurable mesh. In *4th Symp. on the Frontiers of Massively Parallel Computation*, pages 38–45, 1992.

[22] J. Jaja. An introduction to parallel algorithms. 1992.

[23] R. M. Karp, M. Luby, and F. Meyer auf der Heide. Efficient pram simulation on a distributed memory machine. *Algorithmica*, 16(4/5): 517-542 (1996).

[24] D. Konig. Line systems and determinants. *Math. Térmeszettud. Értes'*, 33:221–229, 1915. In Hungarian.

[25] E. Kushilevitz and N. Nisan. *Communication Complexity.* Cambridge university press, 1997.

[26] L. Lovász and M.D. Plummer. *Matching theory.* North Holland, Amsterdam, 1986.

[27] Y. Mansour, N. Nisan, and U. Vishkin. Trade-offs between communication throughput and parallel time. In *Proc. 26'th Annual ACM Symp. on Theory of Computing*, pages 372–381, 1994.

[28] J. M. Marberg and E. Gafni. Sorting in constant number of row and column phases on a mesh. *Algorithmica*, 3:561–572, 1988.

[29] M. Li and Y. Yesha. Resource bounds for parallel computation of threshold and symmetric functions. *J.Computer and System Sciences*, 42:119–137, 1991.

[30] J. I. Munro and M. S. Paterson. Selection and sorting with limited storage. *Theor. Comput. Sci.*, 12:315–323, 1980.

[31] T. Leighton. Introduction to parallel algorithms and architectures: Arrays. trees. hypercubes. *Morgan Kaufman publishers, Inc.*, 1992.

[32] Y.Shiloach and U.Vishkin. Finding the maximum, merging and sorting in a parallel computation model. *Journal of Algorithms*, 2:88–102, 1981.

[33] U.Vishkin and A.Wigderson. Trade-offs between depth and width in parallel computation. *SIAM J.Computing*, 14(2):303–314, 1985.

[34] L.G. Valiant. A bridging model for parallel computation. *Communication of the ACM*, 33(8):103–111, 1990.