ELSEVIER

Theoretical Computer Science 245 (2000) 151–173

Theoretical Computer Science

www.elsevier.com/locate/tcs

access machine ☆

Bogdan S. Chlebus[a], Artur Czumaj[b,1], Leszek Gąsieniec[c], Mirosław Kowaluk[a], Wojciech Plandowski[a]

[a] *Instytut Informatyki, Uniwersytet Warszawski, Banacha 2, 02-097 Warszawa, Poland*
[b] *Department of Computer and Information Science, New Jersey Institute of Technology, University Heights, Newark, NJ 07102-1982, USA*
[c] *Department of Computer Science, The University of Liverpool, Chadwick Building, Peach Street, Liverpool L69 7ZF, UK*

Abstract

We describe a number of algorithms for the model for parallel computation called parallel alternating-direction access machine (PADAM). This model has the memory modules of the global memory arranged as a two-dimensional array, with each processor assigned to a row and a column, the processors can switch synchronously between row and column access modes. We study the issues of inter-processor communication and of efficient use of memory on the PADAM, and develop: an optimal routing scheme among memory modules, algorithms enhancing random access of processors to all memory blocks, and general simulations of shared memory machines. Finally, we present optimal algorithms for the problems of selection, merging, and sorting.

Keywords: Algorithm; Parallel computation; Routing; Simulation

1. Introduction

One of fundamental goals in parallel processing is to develop effective models for parallel computation. The purpose they serve is to provide suitable abstraction which allows us to concentrate on the high-level structure of design while developing

☆ Work partially supported by EC Cooperative Action IC-1000 (project ALTEC: *Algorithms* for Future Technologies) and a research grant from Matsushita Electric Industrial Company Ltd.

[1] Work partially supported by DFG-Graduiertenkolleg "Parallele Rechnernetzwerke in der Produktionstechnik", ME 872/4-1, by EU ESPRIT Long Term Research Project 20244 (ALCOM-IT), and by DFG Leibniz Grant Me872/6-1.

E-mail addresses: chlebus@mimuw.edu.pl (B.S. Chlebus), Leszek@csc.liv.ac.uk (L. Gąsieniec), kowaluk@mimuw.edu.pl (M. Kowaluk), wojtekpl@mimuw.edu.pl (W. Plandowski).

0304-3975/00/$ - see front matter
PII: S0304-3975(99)00280-7

parallel algorithms. In this paper, we investigate the model of parallel computation called parallel alternating-direction access machine (PADAM). PADAM is an abstraction of the architecture of the machine ADENART of Matsushita (see, e.g., [11, 12, 21] and [30, Chapter 3.4.11]) and the prototype USC/OMP developed at the University of Southern California [8]. This model has been previously considered in [9, 25] (cf. also [6, Chapter 9.2.3]) under the name of orthogonal multiprocessor (OMP). The key feature of ADENART, OMP and their abstraction PADAM is the structured organization of the global memory: the memory modules are arranged as a two-dimensional array and each processor has the assigned row and column that it is able to access. The processors switch simultaneously between the column and row access modes and then access the assigned groups of memory modules without conflicts.

The goal of the designers of ADENART and OMP was to develop a special-purpose parallel computer oriented mainly towards scientific applications, able to solve quickly problems in matrix algebra, PDEs, image processing, computer vision, etc. For such purposes it is crucial to be able to perform efficiently complex operations on large matrices. The design was also motivated by the assumption that if each processor has an assigned set of memory modules disjoint from the memory of other processors, then a fast access to them can be implemented with modest hardware requirements (there are two such assignments for the PADAM, and the processors may switch between them). The inventors of OMP provided in [9] (see also [25]) many arguments why the orthogonal multiprocessor is an attractive architecture for parallel scientific computations. They also presented successful simulation performance results of the behavior of the prototype machine USC/OMP in [7]. The Matsushita ADENART was tested on numerical and computer-graphics applications in [11]. The results of further successful tests of the NAS Parallel Benchmark were presented in [22] (see also [24] for comparison of these results with other existing supercomputers). This provides ample evidence that the architecture can be successfully applied in the area of scientific and engineering applications.

In this research we develop algorithmic techniques that could be used to enhance PADAM-like architectures to make them viable general-purpose computers. The main difficulty in designing algorithms on the PADAM is caused by the restricted memory-access pattern. Therefore, our study is mainly concentrated on the issues of efficient use of the available memory. The first problem we consider is that of routing among memory units, which is an important generic problem of communication in a parallel setting. Then we design schemes to streamline access of processors to the memory units. Next, we develop general shared memory simulations. Finally, we present algorithms for selected problems (selection, merging, sorting) as case studies.

1.1. Models of computation

Parallel alternating-direction access machine consists of a set of n processors and a global memory. The global memory is organized as an $n \times n$ array of *memory units* (MUs). In one step a processor can either execute an instruction of its local data or access a MU for reading or writing. Let $P(i)$ denote the ith processor and $M[i, j]$ de-

note the MU in the ith row and the jth column. Processor $P(i)$ can access the MUs in the ith row and in the ith column. The system is always in one of two states: either in the *row-access mode* or in the *column-access mode*. While in the row-access mode, all the processors may access their rows of MUs, that is, processor $P(i)$ can read or write to MU $M[i,k]$, for any k, $1 \leqslant k \leqslant n$. Similarly, if the computer is in the column-access mode, processor $P(i)$ can access $M[k,i]$, for any k, $1 \leqslant k \leqslant n$. Switching modes is programmable, that is, there is an instruction available, say `switch`, to change between the row and column access modes. All the processors execute the same program. Computations are globally synchronized; in particular, if a loop is executed then the next instruction will be started only after all the processors have already exited the loop. The same holds for the `switch` instruction, that is, the processors which are ready to execute it remain idle waiting until the others, still performing their computations, are ready to switch the access mode (see Fig. 1).

Notice that with this restricted access pattern, processors can still communicate directly in constant time. Namely, when processor $P(i)$ needs to send a message to processor $P(k)$, it first writes its value into $M[i,k]$, in the row-access mode, then $P(k)$ waits till the change of mode to the column-access is done, and finally $P(k)$ reads $M[i,k]$ (this takes time $\mathcal{O}(1)$ if there is a constant delay between switching the modes).

We make no general assumptions on the size of a MU. The size of memory *required* by the presented algorithms is proportional to the size needed to store input and output. For instance, the 1–1 routing algorithm can be implemented on a PADAM with each MU of size $\mathcal{O}(1)$, but in the case of h–h routing the MUs need to have the capacity to accommodate h packets each, and nothing more. Also in the case of the randomized PRAM simulation, MUs should be large enough to store up to $\mathcal{O}(t \cdot p)$ words after t steps of the simulation of a p-processor PRAM.

A *distributed memory machine* (DMM) consists of a number of synchronized processors and memory modules (see [14, 19, 29]). Each module can be accessed by any processor in one step, with some semantics to resolve conflicts for access. The DMM has been studied under various names in the literature, usually the authors assumed a specific way to resolve conflicts for access to modules; examples are *direct connection machine* in [14] and S^*PRAM in [29].

A *parallel random access machine* (PRAM) consists of a number of synchronized processors that share a global memory, and every basic CPU operation, including accesses to the shared memory, is executed in a single time step (cf. [10, 13]). So the PRAM is like a DMM in which each module has the capacity of one word. The PRAM model has proven to be particularly appealing and popular among parallel models (see [10]). The reason is that its key feature, the random access to memory, facilitates the design and analysis of algorithms. However, this very property of the PRAM model is unrealistic, because the cost of memory access can hardly be neglected (cf. [1]).

The DMM model is considered more realistic than the PRAM one, as it captures the phenomena occurring if the global memory is partitioned into modules. Simulating a PRAM on a DMM has been a subject of extensive study (see [19, 20, 29]). Notwithstanding its similarity to the DMM and PRAM, the PADAM architecture is more restricted than that of

DMM, because each MU can be accessed directly by at most two processors. We show that the PADAM is strictly weaker than both the DMM and PRAM (see Theorems 5 and 9).

The PADAM architecture is related also to that of a mesh (cf. [15]) and a mesh with buses (called also mesh with broadcasting) (cf. [23, 27, 28]). All these architectures have the underlying $n \times n$ array. The mesh and the mesh with buses consist of n times as many processors as the PADAM. Another difference between the mesh and the PADAM is that each processor of the mesh can communicate only with its four neighbors in the grid, whereas any pair of processors of the PADAM can communicate directly in constant time. The difference between the mesh with buses and the PADAM is that the former consists of n^2 *processors* that can use buses for fast communication in rows and columns, whereas the n-processor PADAM consists of an $n \times n$ array of *memory units* and n processors that have a random access to their assigned rows and columns. Clearly, the mesh with buses is a stronger model than PADAM, with respect to communication capabilities.

1.2. Overview of results

There are four groups of results: (1) routing is studied as an important generic problem; (2) a direct access to the MUs is implemented; (3) general shared memory simulations are provided; (4) algorithms for selection, merging and sorting are developed, as case studies.

1.2.1. Routing

The problem of routing is specified as follows: there are a number of packets residing in the memory units and the task is to relocate them to other memory units that are their destinations. In the case of fixed-connection networks, where nodes store packets which are to be redistributed, the problem of routing is fundamental because this is how the communication between processors can be performed (see [15, 16]). For a machine with a global memory and no restrictions on the access of processors to memory modules (this is the case of the PRAM), routing can be performed trivially by a sequence of read–write operations. The PADAM processors cannot fetch the variable values or write arbitrarily, and if they need to learn values stored in the regions of memory which they cannot reach directly, then they resort to a routing procedure. This makes routing important for PADAM. If $\mathcal{O}(h)$ packets are in each MU and $\mathcal{O}(h)$ packets are addressed to any MU then this is an instance of h–h routing. We present an algorithm solving any h–h routing problem on the PADAM with n processors in time $\mathcal{O}(h \cdot n)$ with queues of size $\mathcal{O}(h)$. The algorithm is deterministic and asymptotically optimal. Recently, other routing algorithms were developed for PADAM, see [4], both deterministic and randomized, having smaller constants in their performance bounds.

Even though PADAM resembles meshes and meshes with buses, for which routing was studied extensively (cf. [17, 23, 26–28] and [16, Section 1.7]), there is no apparent and general way to adapt techniques from these models. In the $n \times n$ mesh, there are n^2 processors that can move n^2 packets in one step, but only to the adjacent nodes. This

also holds in the $n \times n$ mesh with buses, and additionally one item can be broadcast in unit time to all the nodes in its row or column. It is not clear how these computing capabilities could be translated to the PADAM, where there are n processors only, each can handle one packet at a time to transfer it to a possibly distant node (as opposed to the mesh) but only to one at a time (as opposed to the mesh with buses).

The high-level description of the routing algorithm described in this paper is as follows. The array of MUs is partitioned into $\sqrt{n}$ *horizontal* and *vertical strips*, an intersection of a vertical and horizontal strip is a *block*. The algorithm proceeds in three phases: (1) routing to the target vertical strips; (2) routing to target blocks; (3) routing within blocks.

1.2.2. Enhancing access to memory

Next, we study ways of implementing a programmable access of processors to the whole memory. We consider two possible approaches. The low-level one is to implement access of the processors to any memory module, similarly as they access the modules in their assigned rows and columns, but additionally taking care of possible conflicts for access. This is actually an emulation of the DMM with exactly the same processors and MUs as in the emulating machine. We call this *emulation of random access to memory units*. The high-level approach is to *simulate* general *shared memory*, that is, to simulate a PRAM which has an arbitrary number of processors and memory words. In this paper the term *simulation* usually means a general PRAM simulation, and *emulation* denotes the DMM emulation as described above. The presented DMM emulations are simpler than PRAM simulations, and are expected to be significantly faster and easier to implement on real machines. The motivation for developing emulations is *not* to provide general DMM simulations, because next we develop PRAM simulations, but rather to enhance the ease of programming on ADENART/OMP without slowing down the programs significantly. We develop two DMM emulations, one deterministic and the other randomized, and two PRAM simulations, one deterministic and the other randomized. The randomized emulation and simulation are significantly faster than the corresponding deterministic algorithms.

The deterministic emulation is implemented by a specialized routing algorithm, the main difference with the general case is that the number of packets is equal to the number of processors rather than to the number of MUs. A single access step can be performed in time $\mathcal{O}(\sqrt{n})$, this is shown to be optimal by a matching lower bound. The randomized emulation operates with the expected slowdown $\mathcal{O}(\log n)$, this again is optimal. The algorithm is very simple and could be of practical value. The idea is to rotate conceptually the rows by random and independent cyclic shifts, creating new physical addresses of MUs corresponding to the virtual addresses used by the programmer. This makes the physical column-addresses (of the packets in a batch of memory requests) distributed evenly among the columns with high probability, thus avoiding unbalanced load of the processors.

The deterministic PRAM simulation is an extension and modification of the deterministic DMM emulation. We show that a PRAM with p processors, for $p \leqslant n^2$, can

be simulated deterministically step-by-step with the slowdown $\mathcal{O}(\sqrt{p})$, which is optimal. Next we present randomized PRAM simulations. It is shown that one step of the p-processor CRCW PRAM can be simulated on the n-processor PADAM in the optimal time $\mathcal{O}(\lceil p/n \rceil)$, provided $p = \Omega(n^{1+\varepsilon})$, for any constant $\varepsilon > 0$. For smaller values of p, the simulation has delay $\mathcal{O}(\log p \cdot \lceil p/n \rceil)$. The simulations are based on the universal hashing of the PRAM memory among the MUs of the PADAM.

1.2.3. Case studies

We also consider three specific problems of selection, merging and sorting. These problems are investigated as case studies. Our aim is to show that each of these problems can be solved optimally on the PADAM, by a relatively simple algorithm.

1.3. Miscellany

If some property depending on n holds with the probability $1 - n^{-a}$, for any constant a, then the property is said to hold *with high probability*, abbreviated w.h.p. All the presented randomized algorithms are *Las Vegas*, in the sense that they always output the correct solution.

The n-processor PADAM can simulate any p-processor constant degree network (e.g. binary trees, butterflies and comparator-sorting networks) with delay $\mathcal{O}(\lceil p/n \rceil)$. A step of the network is performed by communicating with each of the neighbors. Since each processor knows its neighbors, the corresponding processors of the PADAM know where to look in the memory of messages.

We often refer to certain indexings of the MUs determined by linear orderings. Define the *row-major order* as follows: the MUs in row i precede the MUs in row j, if $i < j$, and all the rows are ordered from left to right. The *column-major order* is defined similarly.

1.4. Paper organization

In Section 2 we present an optimal deterministic h–h routing algorithm. Section 3 includes two methods of emulating random access to memory of the PADAM. In Section 4 we consider general simulations of a PRAM. Section 5 includes the algorithms for selection, merging, and sorting.

A preliminary version of this research appeared as [3].

2. Routing

Suppose that each MU stores at most h packets, and the packets need to be rearranged among the MUs in such a way that each MU receives at most h new packets. This is the *h–h routing* problem.

We first describe an optimal deterministic 1–1 routing algorithm operating in time $\mathcal{O}(n)$. Let us suppose that initially the packets are stored in the MUs, at most one packet at each $M[i,j]$. The array M of MUs is partitioned into $\sqrt{n}$ *horizontal strips*,

which are segments of $\sqrt{n}$ rows, and similarly into $\sqrt{n}$ *vertical strips* comprised of $\sqrt{n}$ columns. The strips are numbered from left to right and from the bottom up, respectively. The $\sqrt{n}$ MUs in a row that are in the same vertical strip make a *horizontal segment*, the *vertical segments* are defined similarly. An intersection of a horizontal and vertical strip is alled a *block*. There are n blocks, each having n elements organized as an $\sqrt{n} \times \sqrt{n}$ array. The algorithm can be conceptually divided into three phases:

PADAM ROUTING

Step 1: Route the packets to their vertical strips.

Step 2: Route the packets to their blocks.

Step 3: Finish routing in the blocks.

These steps are actually subroutines, in the sense that a solution to implementing Step i can be used in designing solution for Step j, for $j<i$. The phases are presented in the reversed order, to facilitate the comprehension.

Step 3: *Routing packets inside the blocks*: Let us suppose that the packets have already been moved to their destination blocks. To complete routing, we convert blocks into rows, then route packets in the rows, and convert rows back to blocks. The details follow (see Fig. 2).

Enumerate the blocks in the (regular) row-major order, which is analogous to the ordering of the array of MUs. We move all the packets from the ith block to the ith row. To this end, the kth row inside a row of blocks, for $1 \leqslant k \leqslant \sqrt{n}$, is shifted cyclically $(k-1) \cdot \sqrt{n}$ columns to the right. More precisely, a packet in the row i and column j, for $1 \leqslant i, j \leqslant n$, is moved to the column number $(j + ((i-1) \bmod \sqrt{n}) \cdot \sqrt{n}) \bmod n$ in the ith row. Now all the packets from a block are in different columns. Permute the packets in the columns so that all the packets from the ith block are moved to the ith row.

This determines a translation of addresses in the block to addresses in the row. Permute the packets in each row, translating block addresses to row addresses, to achieve the desired routing in the block. Finally, convert back rows to blocks in the reversed way, the ith row to the ith block.

Step 2: *Routing packets inside the vertical strips*: Suppose that the packets have already been moved to their destination vertical strips. We show how to move the packets to their destination blocks. Consider a vertical strip. The first stage is to sort the packets in each column on their block addresses. This can be done in time $\mathcal{O}(n)$, one processor per column, as follows. The processor scans the column and finds the number a_i of packets with destination address in the ith block. The number a_i is stored at the top MU of the ith vertical segment of the column. Next the processor computes the partial sums of these numbers: $f_1 = 0$, and $f_k = a_1 + \cdots + a_{k-1}$, for $k > 1$. Now the ith packet among those going to the jth block is moved to the location $i + f_j$ in the column. This can be done by one additional scan of the column, and completes sorting.

The sorted packets in each column are partitioned into two categories: $\mathcal{A}$*-packets* and $\mathcal{B}$*-packets*, as follows. For each block, if there are i packets in the column with

this destination address, then $i-(i \bmod \sqrt{n})$ are designated as $\mathscr{A}$-packets and the remaining ones as the $\mathscr{B}$-packets. Next the columns are sorted in a stable way such that the $\mathscr{A}$-packets precede the $\mathscr{B}$-packets. This can be done in time $\mathcal{O}(n)$ similarly as sorting on the block addresses. The $\mathscr{A}$-packets are dealt with first. Afterwards, we add some dummy $\mathscr{B}$-packets to make exactly $\sqrt{n}$ packets with each block address, and the $\mathscr{B}$-packets are sorted on their block addresses. This operation moves them to their blocks. The dummy packets are then discarded.

It remains to consider the $\mathscr{A}$-packets in detail. At this moment, every vertical segments contains only $\mathscr{A}$-packets with the same block address. Rearrange the $\mathscr{A}$-packets in such a way that those in the rows congruent to i modulo $\sqrt{n}$ go to the ith block (we refer to this permutation of rows in the vertical strips as σ). Next solve the routing problem in each block treating the original block addresses as row addresses inside the block (this is done similarly as in Step 3). Afterwards the packets are rearranged according to the permutation σ^{-1} (see Fig. 3).

Lemma 1. *At this stage of the algorithm the packets are in their proper destination blocks.*

Proof. Consider a block B just after permuting columns according to σ. If there is a packet in B with some block address k then it was taken from a vertical segment of packets with the same block address and moved as the only representative of them to B. Hence there are at most $\sqrt{n}$ packets in B with the block address k. All these packets are to be routed to the kth row in B. Notice that the column addresses do not matter since the packets are to be routed to their destination blocks to whatever column, and this is achieved by first routing the packets to their rows inside blocks. □

Step 1: *Routing the packets to their vertical strips*: First the packets in each row are sorted on their destination vertical-strip addresses. This is done in time $\mathcal{O}(n)$ similarly as sorting columns on their block addresses. Next, for each vertical-strip address, if there are some i packets with this destination address, then $i-(i \bmod \sqrt{n})$ are designated as $\mathscr{A}$-packets and the remaining ones as $\mathscr{B}$-packets. When the $\mathscr{A}$-packets have been routed, the $\mathscr{B}$-packets are padded with the dummy packets to make exactly $\sqrt{n}$ packets with each vertical-strip address in each row.

The packets are sorted in each row and this operation places them in the proper vertical strip. It remains to show how to handle the $\mathscr{A}$-packets. First the $\mathscr{A}$-packets and $\mathscr{B}$-packets are sorted in a stable way in the rows preserving that the $\mathscr{A}$-packets precede the $\mathscr{B}$-packets. Now we permute the $\mathscr{A}$-packets so that those in the columns congruent to i modulo $\sqrt{n}$ are moved to the ith vertical strip (we refer to this permutation of columns as γ). When inside the vertical strips, treat the original vertical-strip addresses as column addresses inside the block, and move the packets to the respective blocks (this is done similarly as in Step 2).

Next move the packets from the ith block to the ith column inside vertical strips, and finish the routing by rearranging the packets according to the permutation γ^{-1} (see Fig. 4).

Lemma 2. *Routing to the vertical strips is correct and can be accomplished in time* $\mathcal{O}(n)$.

Proof. Consider the $\mathcal{A}$-packets in a vertical strip V just after applying the permutation γ. There are at most n packets in V with the same original vertical-strip destination addresses, which are now treated as the block addresses inside V. This guarantees that there is enough room in each block to accommodate the packets. □

h–h routing: Adaptation to the *h–h* case is as follows. While routing inside the blocks, all the packets from a block are reallocated to a row, then the routing in that row is performed. While routing inside vertical strips, we need to define indexing of the locations of packets in a column to determine the sorting order, and then the $\mathcal{A}$-packets and $\mathcal{B}$-packets. The indexing of packet locations is such that all the packets from the MU in a row i are to precede the packets in the MU in the row j, if $i<j$. For each vertical-strip address, if there are some i packets with this destination address, then some $i-(i \bmod (\sqrt{n}\cdot h))$ of them are designated as $\mathcal{A}$-packets and the remaining ones as $\mathcal{B}$-packets. The phase of routing packets to their destination vertical strips is modified accordingly.

Traditionally, all the packets stored in one unit of storage are referred to as the *queue*.

Theorem 1. *The algorithm* PADAM ROUTING *solves an instance of the h–h routing problem on the* PADAM *with n processors in time* $\mathcal{O}(h\cdot n)$, *while the queues are of size* $\mathcal{O}(h)$.

3. Emulating DMM

In this section we show how to implement an access of each processor to any memory unit on the PADAM. This is the same as emulating a DMM with the same processors and MUs. Two emulations are presented. One is deterministic and uses the original memory addressing of the PADAM, the other is randomized, the randomization is used in shifting cyclically the addresses in rows. Each processor creates its memory request as a packet storing some address X of a MU, and memory word w in X; the packet additionally carries the bit defining the request as either reading or writing, and, in the case of a writing request, it carries the additional value y. If the request is for reading then the meaning is that the word w in MU X is to be read (and then brought back to the processor). Otherwise, the meaning is that the value y is to be written into the word w in MU X. The set of packets created by all the processors that are

to be realized concurrently is called a *batch of memory requests*. We do not assume any specific semantics regarding conflicts for access of MUs, but all the popular ones (like EREW, CREW, arbitrary CRCW, priority CRCW) can be handled with natural stipulations augmenting the given algorithm.

3.1. Deterministic emulation

The emulation is performed in a step-by-step manner. Each processor creates a packet and places it in its row in the first column. The packets are first organized into groups determined by the MU addresses. For each group, only one packet is designated to be routed, its selection depends on the conflict-of-access-resolution protocol. In the case of a writing step, this packet will carry the value which is to be written. In the case of a reading step, this packet will be also routed back by a reversed route and will bring the value retrieved from the accessed MU. This value will be distributed among the packets in the group, if necessary, and finally these packets will be sent back to their original rows.

EMULATION E1

Step 1: The packets have been placed in the first column. Sort them on their destination-column addresses by emulating the Batcher's sorting network (see [15]). Now the groups of packets with the same addresses are in contiguous blocks of locations in the first column.

Step 2: Select one packet (if any) from each group. The selection is made depending on the conflict-resolution protocol.

Step 3: The selected packets are routed up to fill a contiguous block of MUs in the first column, including the top row, one packet per MU. This is accomplished by first computing the addresses for the packets by the parallel prefix algorithm executed on a simulated tree, and then by routing on a simulated butterfly by applying a monotone-routing algorithm (see [15]). After this the selected packets remain sorted on their destination-column addresses. We consider only the selected packets from now on.

Step 4: For each group of the packets with the same destination-column address, if there are some i packets in the group then designate $i-(i \bmod \sqrt{n})$ packets as $\mathscr{A}$-packets, the remainder are the $\mathscr{B}$-packets. This can be done on a simulated tree. The $\mathscr{A}$-packets are then routed to a contiguous block of locations, preserving their relative order. This is accomplished similarly as in Step 3.

We first deal with the $\mathscr{A}$-packets.

Step 5: The ith vertical segment of $\sqrt{n}$ $\mathscr{A}$-packets is routed $i-1$ columns to the right. Then the processors switch to the column-access mode. The ith processor, for $i \leqslant \sqrt{n}$, looks for $\sqrt{n}$ packets in its column in the consecutive rows starting from the row $(i-1)\sqrt{n}+1$. If there are any packets there then they are moved to their destination rows.

Step 6: The processors switch to the row-access mode. Each processor reads the first $\sqrt{n}$ MUs in its column looking for packets, and if it finds any then they are moved to their destination columns.

Lemma 3. *The $\mathscr{A}$-packets are routed in time $\mathcal{O}(\sqrt{n})$.*

Proof. All the packets that have been in the same vertical segment and were moved to the same column in Step 5 have the same destination-column address and hence distinct destination-row addresses. This implies that there is at most one packet in every MU in the first $\sqrt{n}$ columns after Step 5. □

This completes routing of the $\mathscr{A}$-packets. The next steps take care of the $\mathscr{B}$-packets. They are already in a contiguous block of memory locations, and the packets with the same destination-column addresses also make contiguous block. These blocks will now be moved to their destination columns. Before that we need to inform the processors where they are to look for packets in their columns.

Step 7: For each packet which is the first in a group of packets with the same column address j, and which is in the row i, create a *special* packet to carry the number i and send it to the jth processor, say to $M[j,j]$. To this end, the special packet from the ith row is first moved to the column $\lfloor (i-1)/\sqrt{n} \rfloor + 1$. Then the processors switch to the column-access mode and the kth processor, for $k \leqslant \sqrt{n}$, scans $\sqrt{n}$ MUs, starting from the row $(k-1)\sqrt{n}+1$. If a packet is found then it is routed to its destination row. Finally, the processors switch to the row-access mode and scan the first $\sqrt{n}$ columns and move the packets (if any) to their column addresses. The special packets are handled in time $\mathcal{O}(\sqrt{n})$, because there is never more than a single packet in each MU, this follows from the fact that they have distinct row addresses.

Step 8: Now the $\mathscr{B}$-packets are moved to their destination columns in one step. The processors switch to the column-access mode and place themselves at the rows whose numbers have just been distributed by the special packets. Then they move the packets waiting there to their destination rows.

This completes the emulation and proves:

Theorem 2. *The algorithm* E1 *on a* PADAM *with n processors performs one batch of memory request in time $\mathcal{O}(\sqrt{n})$.*

The following matching lower bound holds.

Theorem 3. *Any emulation of random access to memory units on the n-processor* PADAM *requires $\Omega(\sqrt{n})$ operations per batch of memory request in the worst case.*

Proof. Consider a batch of requests to n different MUs which are located in $\sqrt{n}$ rows and $\sqrt{n}$ columns. The PADAM is either in row-access or column-access mode. In any case, in one step at most $\sqrt{n}$ processors can access the required MUs. □

3.2. Randomized emulation

The emulation is based on an *offset addressing* mechanism. It is determined by a sequence a_i of *offsets*, where $1 \leqslant i \leqslant n$ and $1 \leqslant a_i \leqslant n$. Instead of $M[i,j]$, the access to

the ith row and the jth column is meant to be to $M'[i,j]=M[i,(j+a_i) \bmod n]$. Just before the computation starts, each $P(i)$ generates a random offset a_i and stores it in $M[i,1]$. Then the input is read and stored in the MUs. The following is an emulation of a single memory reference.

EMULATION E2

Step 1: Each processor $P(i)$ generates a memory access request and places the respective (ordinary) packet p_i in $M[i,1]$. Assume that packet address are all distinct, otherwise proceed as in E1.

Step 2: The packets are sorted on their row addresses. The processor $P(i)$ such that the packet at $M[i,1]$ is the first in a block of packets with the same row address, say r_i, generates a packet to inquire about the offset a_{r_i}. The inquiring packets are routed to their respective rows in the first column on a simulated butterfly, and then routed back (these are instances of monotone routing). The offsets they bring are disseminated among other packets with the same row address on a simulated tree.

Step 3: The packets p_i are sorted on their offset column addresses k_i and stored again in the column $M[*,1]$.

Step 4: Each processor $P(i)$ checks to see if the packet residing now in $M[i,1]$ is the first in the block of packets with the same key, say k_i. If this is the case then it generates a special packet storing i and the address $M[k_i,1]$.

Step 5: The special packets are routed to their destination addresses on a simulated butterfly. Then each processor $P(i)$ checks $M[i,1]$ for such a packet.

Step 6: Each processor $P(i)$ moves the (ordinary) packet from $M[i,1]$ to $M[i,k_i]$, where k_i is the key of the packet.

Step 7: The processors switch to the column-access mode and processor $P(j)$ scans consecutive locations in the column j, starting from the address delivered by the special packet (if any) and moves the packets waiting there (if any) to their final destinations.

Theorem 4. *The algorithm* E2 *performs one batch of memory requests on a* PADAM *with n processors in time* $\mathcal{O}(\log n)$ *w.h.p.*

Proof. The sortings in Steps 2 and 3 are performed by emulating the randomized hypercubic sorting algorithm of Leighton and Plaxton [18]. Hence the Steps 1–6 are performed in time $\mathcal{O}(\log n)$. The time of Step 7 depends on the maximum number of packets in a column. Consider a specific column number j. A packet in row i goes to $M[i,j]$ with the probability $1/n$ since the offset a_i was selected randomly. If all the packets were in different rows then we would have n independent selections of a column, each selected with the probability $1/n$. It is well known that every column would be selected at most $\mathcal{O}(\log n/\log\log n)$ times w.h.p. It may happen that many packets are in the same row. Then our estimates hold true as well because if one of these packets is selected for a particular column then the remaining packets must go to other columns. □

The following matching lower bound holds.

Theorem 5. *Any randomized emulation of random memory access on the n-processor* PADAM *has the expected number of* $\Omega(\log n)$ *steps per batch of memory requests.*

Proof. Consider the special case when MU stores $\mathcal{O}(1)$ words and we emulate the EREW DMM. Then the lower bound follows from Theorem 9. □

4. Shared memory simulations

In this section we consider simulations of the random-access shared-memory architecture on the PADAM. First, we describe a deterministic simulation of a PRAM with some restrictions on the number of processors and the size of the shared memory. Next, we show a general randomized simulation of a p-processor CRCW PRAM with unbounded shared memory on the n-processor PADAM. The simulation has an optimal delay for $p=\mathcal{O}(n)$ and for $p=\Omega(n^{1+\varepsilon})$, for any constant $\varepsilon>0$. If t is the running time of a CRCW PRAM algorithm and $pt>n^2$, then it is assumed that each MU of the PADAM can store $\Omega(pt/n^2)$ words.

4.1. Deterministic simulation

We present a deterministic simulation of a PRAM with memory $\mathcal{O}(n^2)$ and p processors, for $p\leqslant n^2$. The algorithm uses $\mathcal{O}(1)$ memory of each MU, but, alternatively, could be easily modified to an emulation of a DMM with p processors and the MUs of the PADAM. It is possible to use the algorithm E1 directly to obtain a similar simulation, or the respective DMM emulation, as follows. Divide the PRAM processors into $\lceil p/n\rceil$ groups of at most n processors. To simulate one step, perform $\lceil p/n\rceil$ simulation phases, in each one taking care of n processors. Each phase takes $\mathcal{O}(\sqrt{n})$ steps, and the delay is $\mathcal{O}(p/\sqrt{n})$. We present an alternative simulation with delay $\mathcal{O}(\sqrt{p})$, which is better if $p>n$.

The simulation is an extension of emulation E1, so we just sketch it. Initially, the PRAM processors are divided into groups of size p/n and the groups assigned to the PADAM processors. Let $\mathscr{K}$ be the set of MUs $M[i,j]$ with $i,j\leqslant\lceil\sqrt{p}\rceil$.

SIMULATION S1

Step 1: The processors create packets storing the memory requests, and place them in their rows in the first p/n columns. Next, the packets are allocated to the square $\mathscr{K}$, one packet per MU. This can be accomplished in a straightforward way in time $p/n+\sqrt{p}\leqslant 2\cdot\sqrt{p}$.

Step 2: The packets in $\mathscr{K}$ are sorted on their destination-column addresses in column major order in time $\mathcal{O}(\sqrt{p})$.

A packet is defined to be an $\mathscr{A}$-packet if all the packets in its column in $\mathscr{K}$ have the same destination-column address. The remaining packets are $\mathscr{B}$-packets.

Step 3: The $\mathscr{A}$-packets are moved to their destination rows. Then the processors switch to the row access mode, scan the first $\sqrt{p}$ MUs in their rows and move the $\mathscr{A}$-packets residing there to their final destinations.

Step 4: The $\mathscr{B}$-packets with the same column address are in contiguous blocks in at most two columns. For each such block(s) of packets going to the column i create a special packet p_i with the row addresses of the blocks. Then the special packets are routed to the diagonal MUs, p_i to $M[i,i]$, and retrieved by the processors.

Step 5: The $\mathscr{B}$-packets are routed along the rows to their destination columns. Then the processors switch to the column access mode and deliver the $\mathscr{B}$-packets to their final destinations.

Theorem 6. *A p-processor* PRAM *with $\mathcal{O}(n^2)$ memory words can be simulated on the n-processor* PADAM *with delay $\mathcal{O}(\sqrt{p})$, provided $p \leqslant n^2$.*

To prove Theorem 6, we need only to show how to sort efficiently in Step 2; this is described in Theorem 13.

4.2. Randomized simulation

In this subsection a randomized shared memory simulation is presented. We assume the strongest CRCW model of computation on the PRAM. The algorithm hashes the shared memory of the PRAM into the (usually smaller) address space of the PADAM. First, we give an optimal simulation of a n-processor CRCW PRAM on the n-processor PADAM. Next, we extend it to the case of a p-processor CRCW PRAM, for any p. Our algorithm achieves the optimal delay $\mathcal{O}(\lceil p/n \rceil)$ for $p \geqslant n^{1+\varepsilon}$. Since our simulation hashes the shared memory of the PRAM, we do not assume any bounds on the size of the memory of the PRAM.

A PRAM consists of p processors $\bar{P}(1), \bar{P}(2), \ldots, \bar{P}(p)$ and a shared memory with cells $U = \{0, \ldots, u-1\}$, where u is arbitrary. The PADAM has n processors $P(1), P(2), \ldots, P(n)$. In our simulations the shared memory cells of the PRAM are distributed among the rows of the PADAM using a hash function $h : U \to \{1, \ldots, n\}$ drawn at random from the universal class of hash functions $\mathscr{R}(n,n,d)$ introduced by Dietzfelbinger and Meyer auf der Heide [5]. This family of hash functions has the following useful property:

Fact 1 (Dietzfelbinger and Meyer auf der Heide [5]). *If h is a random hash function from class $\mathscr{R}(n,n,d)$, then, for a constant d sufficiently large, and for any $X \subseteq U$, w.h.p. $\max_{1 \leqslant i \leqslant n} \{|h^{-1}(i) \cap X|\} = \mathcal{O}(|X|/n + \log n)$. $\mathscr{R}(n,n,d)$ consists of functions $h = h(f, a_0, \ldots, a_{n-1})$, where f is a polynomial of degree $d-1$ in the field of residues of some large prime taken modulo $n^2, a_0, \ldots, a_{n-1} \in \{0, \ldots, n-1\}$, and $h(x) = (f(x) \bmod n + a_{f(x)} \mathrm{div}_n) \bmod n$ for $x \in U$.*

Lemma 4. *Any n requests to the hash values of $h \in \mathscr{R}(n,n,d)$ can be evaluated on the n-processor* PADAM *in time $\mathcal{O}(\log n)$.*

Proof. Suppose processor $P(i)$ needs $h(x_i)$, for some x_i. First, processor $P(1)$ generates a random f in time $\mathcal{O}(d)$. Next, each processor $P(i)$ generates a random $a_{i-1} \in \{0,\ldots,n-1\}$. Function f is distributed to all the processors on an emulated binary tree. The notation $f_1(x)$ means $f(x)\mathrm{div}\, n$. Packets with values $f_1(x_1),\ldots,f_1(x_n)$ are sorted. Each a_i is routed to the first x such that $f_1(x)=i$, and next broadcast to the other locations storing the sorted values of f_1 equal to i. Finally, the packets with $a_{f_1(x_i)}$ are routed back to $P(i)$. □

We have actually shown that any p requests of values of $h \in \mathcal{R}(n,n,d)$ can be evaluated on the n-processors PADAM in time needed to perform integer sorting of p numbers.

A dictionary is a data structure supporting two operations:

LOOKUP(x) — returns the information associated with x that is currently stored in the dictionary; and

INSERT(x,c) — stores x together with the information c associated with x in the dictionary; if x was already stored then it is first removed from the dictionary.

Fact 2 (Dietzfelbinger and Meyer auf der Heide [5]). *There is a randomized implementation of a dictionary which uses $\mathcal{O}(n)$ space and that performs each of a sequence of $\mathcal{O}(n)$ dictionary operations in constant time w.h.p.*

At the beginning of the simulation we choose randomly a hash function $h: U \to \{1,\ldots,n\}$ from $\mathcal{R}(n,n,d)$. Function h defines the row $h(x)$ of the PADAM in which cell x of the PRAM will be stored. Let $\mathcal{M}=\{m_1,\ldots,m_p\} \subseteq U$. In a given PRAM step, $\bar{P}(i)$ wants to access cell m_i and either read its content as x_i (in the reading phase) or write y_i to m_i (in the writing phase). We refer to the respective pair (m_i,x_i) or (m_i,y_i) as the packet μ_i.

SIMULATION S2

Step 1: Each processor $P(i)$ creates a packet μ_i, places it in $M[i,1]$, and evaluates $h(m_i)$.

Step 2: All the packets are sorted on their PRAM addresses from $\mathcal{M}$. Now the groups of packets with the same addresses are in contiguous block of locations in the first column. Select one packet from each group according to the writing-conflict-resolution protocol.

Step 3: The selected packets are routed up to fill a contiguous block of MUs in the first column, including the top row, one packet per MU. We consider only these packets from now on.

Step 4: These packets are sorted on their destination-row addresses $h(m_i)$. Using the obtained sorted sequence, each packet computes the number of packets with the same destination row and the number of packets with the same destination row which precede it in the sorted sequence. This is accomplished by emulating a binary tree.

Step 5: The first packet destined for each row is moved to that row, as on a binary tree.

Step 6: The remaining packets are handled as follows. If the ith packet μ_i (with respect to the order obtained by sorting in Step 4) was moved in Step 5 to row k and there are j packets to be delivered to row k, then in the next $j-1$ moves: processor $P(i+l)$ writes μ_{i+l} to $M[i+l,k]$ in move l and $P(k)$ reads μ_{i+l} from that cell. Afterwards $P(k)$ stores all the packets destined for row k.

Step 7: Each processor $P(i)$ maintains a local dictionary in its row i. In the writing phase, $P(i)$ executes operation INSERT(m_j, y_j) for every packet μ_j with $h(m_j)=i$. In the reading phase, $P(i)$ executes operation $x_j=$ LOOKUP(m_j) for every $h(m_j)=i$.

Step 8: The values read are distributed among the read packets left in Step 2.

Theorem 7. *Simulations* s2 *performs one step of the n-processor* CRCW PRAM *on the n-processor* PADAM *in time* $\mathcal{O}(\log n)$ *w.h.p.*

Proof. The correctness of the simulation is straightforward and the optimality follows from Theorem 5. The following time estimates hold w.h.p. Steps 3,5–7 are performed in time $\mathcal{O}(\log n)$. Sortings are performed in time $\mathcal{O}(\log n)$ by a simulation of the sorting circuit of Leighton and Plaxton [18]. Fact 1 ensures that in each step each processor performs only $O(\log n)$ operations on its dictionary. Step 7 needs $\mathcal{O}(\log n)$ time, by Fact 2. □

The algorithm above can be extended to simulate the p-processor PRAM on the n-processor PADAM for any p. We use the following simple class of hash functions. Let q be a prime and $U=\{0,\ldots,u-1\}$ for $u\leqslant q$. For $s\geqslant 1$ define the set $\mathcal{H}_s^1=\{h_{a,b}\colon 0\leqslant a,b<q\}$ where $h_{a,b}(x)=(a+bx \bmod q) \bmod s$, for $x\in U$ and $0\leqslant a,b<q$.

Fact 3 (Dietzfelbinger and Meyer auf der Heide [5]). *Let* $h: U\to\{0,\ldots,s-1\}$ *be a random function from class* $\mathcal{H}_s^1$, *then for any* $X\subseteq U$, *if* $2|X|^{c+2}\leqslant s$ *then* $\Pr(\max_{1\leqslant i\leqslant s}\{|h^{-1}(i)\cap X|\}\geqslant 1-|X|^{-c}$.

Observe that a function $h\in\mathcal{H}_s^1$ can be stored in $O(1)$ cells, and can be generated and evaluated in constant time by one processor. Hence, it can be generated by $P(1)$ and broadcast by emulating a binary tree to all the processors of the PADAM in time $\mathcal{O}(\log n)$. Afterwards each processor can evaluate h in constant time.

Theorem 8. *One step of the p-processor* CRCW PRAM *can be simulated on the n-processor* PADAM *in the optimal time* $\mathcal{O}(\lceil p/n\rceil)$, *provided* $p=\Omega(n^{1+\varepsilon})$ *for any constant* $\varepsilon>0$. *For smaller values of p the simulation has delay* $\mathcal{O}(\log p\cdot\lceil p/n\rceil)$.

Proof. The simulation is an extension of s2. The case $p<n$ is straightforward. If $p>n$, then we can randomly divide p requests to the memory of the PRAM step into batches of size at most n. Then we perform the algorithm s2 for each batch independently. For $p=n^{O(1)}$ and $T=n^{O(1)}$ this yields the simulation of one step of the p-processor CRCW PRAM on the n-processor PADAM with delay $\mathcal{O}(\log n\cdot\lceil p/n\rceil)$ w.h.p. Now we sketch the optimal simulation for $p=\Theta(n^{1+\varepsilon})$, for any constant $\varepsilon>0$.

We proceed similarly as in the simulation for $p=n$. Now however, Steps 2 and 4 use integer sorting instead of general sorting. Theorem 14 shows that p integers can be sorted in the optimal time $\mathcal{O}(p/n)$ on the n-processor PADAM, for $p=n^{1+\varepsilon}$. In order to use integer sorting we have to reduce the problem of eliminating all requests to the same address in the PRAM to the integer case. We can do this by employing a random hash function $\bar{h}: U \to \{1,\ldots,2\cdot p^{c+2}\}$ from class $\mathcal{H}^1_{2\cdot p^{c+2}}$. Fact 3 implies that the set $\{m_i \in \mathcal{M}: \exists_{m_j \neq m_i} \bar{h}(m_i)=\bar{h}(m_j)\}$ is empty with the probability at least $1-p^{-c}$. Hence, we can apply integer sorting on $\bar{h}(m_1),\ldots,\bar{h}(m_p)$ to store the addresses with the same value in a contiguous subsequence of the sorted sequence w.h.p.

The algorithm can be extended to simulate a p-processor CRCW PRAM on the n-processor PADAM also when p is arbitrarily large. □

In this proof, we have essentially reduced the problem of simulating the p-processor CRCW PRAM on an n-processor PADAM to the problem of sorting p integer numbers on the n-processor PADAM. Thus, any improvement of such an algorithm for $p=\omega(n)$ and $p=o(n^{1+\varepsilon})$, for any constant $\varepsilon>0$, would also improve our simulation.

4.3. Lower bound for shared memory simulations

In this section we show that emulation E2 and simulation S2 are optimal. This follows from the lower bound on randomized PRAM simulations given in Theorem 9.

We will use the following fact:

Fact 4 (Yao [32]). *Let P_1 be the probability that a fixed randomized algorithm solves problem A in T steps, minimized over all the possible inputs. Let P_2 be the probability that a deterministic algorithm solves A in T steps, subject to a fixed probability distribution $\mathcal{D}$ on the inputs, maximized over all the deterministic algorithms operating in T steps. Then $P_1 \leqslant P_2$.*

Theorem 9. *Any randomized simulation of the n-processor* EREW PRAM *on the* PADAM, *that operates correctly with the probability larger than $\frac{1}{2}+\varepsilon$, for some constant $\varepsilon>0$, requires time $\Omega(\log n)$.*

Proof. Consider the following problem: given n bits, $x_1,\ldots,x_n$, of which at most one is a 1 and the others are 0's, compute OR of $x_1,\ldots,x_n$, that is, verify whether there exists i with $x_i=1$. Clearly, this problem can be solved in constant time on the n-processor EREW PRAM. We show that there exists an input distribution $\mathcal{D}$ such that any deterministic PADAM algorithm (with arbitrary number of processors) which solves that problem with probability $(\frac{1}{2}+\varepsilon)$ must run in time $\Omega(\log n)$. Combined with Fact 4, this will establish the theorem. Our proof is similar to that of the lower bound for the CROW PRAM for the same problem given by Snir.

Let I_j, for $1 \leqslant j \leqslant n$, be the input with $x_j=1$ and 0's everywhere else, and let I_0 be the input consisting of all 0's. We say that a processor P or a memory cell C *depends*

on the input variable x_j at time t, if its state on input I_0 is distinct from its state on input I_j at time t. We will prove by induction on t the following fact: Any processor P or memory cell C at time t depends on at most 3^t variables.

The basis of induction is clear. Consider the inductive step. Observe that at time t a processor P can depend on:

(1) any variable it depends on at time $t-1$, and

(2) any variable that the memory cell C_P depended on, if it read C_P during the last step.

By the inductive hypothesis, there are at most 3^{t-1} variables that P depended on at time $t-1$, and at most 3^{t-1} variables that C_P depended on at time $t-1$. Hence processor P can depend on at most $2 \cdot 3^{t-1} \leqslant 3^t$ variables.

Now let us consider memory cell C at time t. It can depend on:

(1) any variable it depends on at time $t-1$, and

(2) any variable that all the processors P_C depended on, where P_C are exactly the processors allowed to write into C during the last step.

Observe that at most two PADAM processors can write into a given memory cell. This implies that memory cell C at time t depends on at most $3 \cdot 3^{t-1} = 3^t$ variables.

Let P_1 be the processor writing the answer and let $\mathscr{D}$ be the following input distribution: $\Pr[I_0 \text{ is the chosen input}] = \frac{1}{2}$ and $\Pr[I_j \text{ is the chosen input}] = 1/2n$, for $1 \leqslant j \leqslant n$. After t steps P_1 depends on at most 3^t variables $x_{i_1}, \ldots, x_{i_{3^t}}$. With probability $3^t/2n$ one of these variables has value 1. Then P_1 gives the correct answer. If none of the variables is 1, then P_1 has to guess the answer to be either 1 or 0, basing this decision only on the fact that all the variables in $\{x_{i_1}, \ldots, x_{i_{3^t}}\}$ are zeros. If it decides to output 0, then it fails for all the inputs I_j with $x_j \notin \{x_{i_1}, \ldots, x_{i_{3^t}}\}$, and gives the wrong answer with the probability $(n-3^t)/2n$. Otherwise, it fails on I_0, and gives the wrong answer with the probability $\frac{1}{2}$. Hence the algorithm has the success probability at most $(n+3^t)/2n$.

Thus in order to make the success probability at least $\frac{1}{2}+\varepsilon$, we must have $3^t/2n \geqslant \varepsilon$. Therefore, if the PADAM algorithm runs in time t, then $3^t \geqslant 2n\varepsilon$. □

5. Case studies: selection, merging, sorting

In the previous sections we considered algorithmic problems related to the communication issues on the PADAM. Now we present optimal algorithms for the problems of selection, merging and sorting. They are examples of solutions on the PADAM of the classical algorithmic problems, and they are included as case studies. Some algorithms for specific problems were developed in [9, 25] in particular for matrix multiplication, LU decomposition, triangular systems, solving linear system, FFT, linear programming, solving partial differential equation (PDE), and polynomial manipulations. In [9] a nonoptimal sorting algorithm was given.

For the sake of simplicity of the exposition, we assume that there are exactly n^2 keys which are stored in the global memory, one key per MU, unless stated otherwise.

5.1. Selection

Given a sequence X of n^2 keys and an integer k, $1 \leqslant k \leqslant n^2$, the task is to find the kth key in X. This key is defined by the property that there are less then k keys in X smaller than x, and for each $y \in X$, $y > x$, there are at least k keys in X which are smaller than y. The notation $|Y|$ means the number of elements in the set Y. The median of Y is the $\lceil |Y|/2 \rceil$th key in Y. In the following presentation of the algorithm, it is assumed that all the keys are distinct to simplify the exposition, the general case can be handled along similar lines.

PADAM SELECTION

Step 1: The processors find in parallel the median of the keys in each row. They place the medians in the first column, next switch to the column-access mode, and the first processor finds the median z of the row medians.

Step 2: The value of z is made known to all the processors who count the number k_z of occurrences of keys smaller than z. Let $S_z = \{x \in X \mid x < z\}$ and $k_z = |S_z|$. If $k_z < k$ then let $X_1 := X - S_z$ and $k_1 := k - k_z$, otherwise let $X_1 := S_z$ and $k_1 := k$.

Step 3: The keys in X_1 are ranked in the row-major order, that is, a key x in row i obtains the rank r if there are $r-1$ keys from X_1 in the rows with numbers smaller than i and in the row i to the left of x. This is accomplished by combining prefix computations in the rows and one column.

Step 4: The keys from X_1 are moved to the square array A of the MUs $M[i,j]$ such that $i,j \leqslant \lceil \sqrt{X_1} \rceil$. This is accomplished by the algorithm PADAM ROUTING. The key of rank r in X_1 is routed to the rth MU in A with respect to the row-major order.

Step 5: Recursively, the k_1th key in X_1 is found in A by the processors $P(i)$ such that $i \leqslant \lceil \sqrt{X_1} \rceil$.

Theorem 10. *The algorithm* PADAM SELECTION *finds the kth key in a set of* n^2 *keys in time* $O(n)$ *on the* PADAM *with n processors.*

Proof. The first step is performed by executing the sequential selection algorithm of Blum et al. [2] in each row in time $\mathcal{O}(n)$. The Steps 2–4 are also completed within this time bound. There are at most $\lceil 3n^2/4 \rceil$ keys in the set X_1 because z is the median of the row medians. The total time bound $T(n)$ of the algorithm satisfies the equation $T(n) = T(\lceil \sqrt{3}n/2 \rceil) + \mathcal{O}(n)$, hence $T(n) = \mathcal{O}(n)$. □

5.2. Merging

The task is to combine two sorted sequences into one sorted sequence. Assume that one of two input sequences occupies the first $n/2$ rows, and the other the remaining rows.

PADAM MERGE

Step 1: Rearrange the keys in such a way that the upper sequence is sorted in the regular row-major left-to-right order, and the lower sequence similarly in the right-to-

left order.

Step 2: Sort the columns.

Step 3: Sort the rows.

Step 1 can be performed in time $\mathcal{O}(n)$, as an instance of off-line routing. Step 2 simply merges two sequences, since each column is a bitonic sequence.

Lemma 5. *After Step* 2, *the sequences stored in the rows are bitonic.*

Proof. Take a key x. Let 0s denote the keys y such that $y \leqslant x$, and 1s keys y such that $y > x$. After Step 1 there are at most two *dirty* rows (containing both 0s and 1s), one having 0s in a block on the left side (if any), the other on the right side (if any). After Step 2, there is exactly one dirty row, of one of two possible forms: 0s1s0s or 1s0s1s. The lemma follows. □

Hence in Step 3 it is sufficient to merge the columns. This proves the following:

Theorem 11. *The algorithm* PADAM MERGE *merges two sorted sequences of* $n^2/2$ *keys in time* $\mathcal{O}(n)$ *on the n-processor* PADAM. □

5.3. Sorting

The sorting problem is to rearrange the elements of an $n \times n$ array in such a way that they appear in a nondecreasing order with respect to some indexing of array entries. Sorting on the PADAM can be accomplished in the optimal time $\mathcal{O}(n \log n)$ by a mergesort, we use the left-to-right row-major indexing:

PADAM MERGESORT

Step 1: Sort each row.

Step 2: Merge repeatedly blocks of sorted rows.

Step 1 can be accomplished in time $\mathcal{O}(n \log n)$, and there are $\mathcal{O}(\log n)$ iterations of Step 2, each taking time $\mathcal{O}(n)$.

Theorem 12. *The algorithm* PADAM MERGESORT *sorts* $\mathcal{O}(n^2)$ *keys in time* $\mathcal{O}(n \log n)$ *on the* PADAM *with n processors.* □

If the keys are integers of polynomial magnitude then they can be sorted faster. We start with a result that is used in the deterministic simulation of PRAM on PADAM.

Theorem 13. *Suppose that at most* n^2 *keys are distributed among the* n^2 *MUs of the* PADAM *with n processors, each key is a natural number between* 1 *and* n^a, *for a constant a. Then the keys can be sorted in time* $\mathcal{O}(n)$.

Proof. The keys are sorted in the column-major order. It is sufficient to show how to sort in a stable way a set of keys in the range $[1 \ldots n]$. If we refer to an array of numbers $[x_{k,l}]$ then it is understood that the entry $x_{k,l}$ is stored in $M[k,l]$.

First each column is sorted in a stable way, similarly as in PADAM ROUTING. Then the location of the beginning of the block of keys equal to i in column j, denoted $b_{i,j}$, is found. The numbers $c_{i,j}$, equal to the number of occurrences of keys i in column j, are computed, and then the prefixes $d_{i,j} = \sum_{k=1}^{j} c_{i,k}$. Let us consider the key s which is now stored in $M[i,j]$. It is to be moved to the $a_{i,j}$th MU (we refer to the numbering according to the column-major order), where $a_{i,j}$ is equal to the number of occurrences of keys smaller than s or those equal to s in the columns $<j$ or in the column j but in the rows $\leqslant i$. The number of keys equal to s above s in column j can be found from the value of $b_{s,j}$. This is combined with $d_{s,j}$ to yield $a_{i,j}$. Finally, the packets are moved to their positions by invoking PADAM ROUTING. □

We can improve the above theorem in the case when the number of keys is strictly less than n^2. This result is also used in simulation s2.

Theorem 14. *Suppose that $m \leqslant n^2$ keys are distributed evenly among the n processors of the* PADAM, *each key is a natural number between* 1 *and r. Then the keys can be sorted in time $\mathcal{O}((m/n + \log n)\lceil(\log r)/\log(m/n + \log n)\rceil)$.*

The above sorting algorithm is optimal for $m \geqslant n^{1+\varepsilon}$ and $r = m^{O(1)}$.

Proof. The algorithm employs an adaptation of the idea of the bucket sorting algorithm of Wagner and Han [31]. Assume first that $r \leqslant m/n + \log n$.

Step 1: Each processor computes sequentially the number of its occurrences of each key and the rank of each amongst these occurrences. The set of keys of value j stored by processor $P(i)$ is referred to as the *bucket* $B(j,i)$. The size of bucket $B(j,i)$ is denoted as $S(j,i)$. A label $C(x)$ is attached to each key x, which is the rank of its occurrence in bucket $B(j,i)$, where j is the value of key x.

Step 2: The buckets are divided into n groups of size r. We order the buckets lexicographically with respect to their identification numbers. The buckets in the jth group, which are assigned to $P(j)$, precede those in the $(j+1)$st group. The sizes $S(k,i)$ of the buckets $B(k,i)$ in group j are routed to processor $P(j)$.

Step 3: Each $P(i)$ computes sequentially prefix sums $E(i,j)$, $0 \leqslant j \leqslant r$, of the sizes of the first j buckets in group i. Let $E(i) = E(i,r)$ be the number of keys in the buckets in group i.

Step 4: All processors compute in parallel the prefix sums $F(i) = \sum_{k=1}^{i-1} E(k)$, $1 \leqslant i \leqslant n$, which is the number of keys in the buckets in groups $1, 2, \ldots, i-1$.

Step 5: Each $P(i)$ computes sequentially $G(i,j) = F(i) + E(i,j-1)$ for $1 \leqslant j \leqslant r$, which is the number of keys in the buckets smaller than the jth bucket in group i.

Step 6: Observe, that if x was in bucket $B(i,j)$, which is the kth bucket in group l, then its rank is $G(l,k) + C(x)$. Hence, to finish the algorithm, we only need to route the values $G(i,j)$ similarly as in Step 2, but in the reverse order.

Next we estimate the performance of the algorithm. Step 1 can be performed in time $\mathcal{O}(m/n)$ and Steps 3 and 5 in time $\mathcal{O}(r)$. Step 4 can be performed in time $\mathcal{O}(\log n)$ on an emulated binary tree by the standard prefix sums algorithm. Finally, Steps 2 and 6 can be done in $\mathcal{O}(r)$ time because our routing pattern is fixed. The total time is $\mathcal{O}(m/n + \log n)$, because we have assumed that $r \leqslant m/n + \log n$.

The algorithm can be modified to handle the case $r > m/n + \log n$ using the idea of radix sort. Divide $\log r$ bits representing the numbers into parts of size $\log(m/n + \log n)$, and, starting from the least significant part, apply the algorithm as described above to all the parts. In order to sort all the numbers, this procedure is repeated $(\log r)/\log(m/n + \log n)$ times. □

References

[1] A. Aggarwal, A.K. Chandra, M. Snir, On communication latency in PRAM computations, in: Proc. ACM Symp. on Parallel Algorithms and Architectures, 1989, pp. 11–21.
[2] M. Blum, R.W. Floyd, V.R. Pratt, R.L. Rivest, R.E. Tarjan, Time bounds for selection, J. Comput. System Sci. 7 (1972) 448–461.
[3] B.S. Chlebus, A. Czumaj, L. Gąsieniec, M. Kowaluk, W. Plandowski, Parallel alternating-direction access machine, in: Proc. 21st Internat. Symp. on Mathematical Foundations of Computer Science, 1996, Lecture Notes in Computer Science, Vol. 1113, pp. 267–278.
[4] B.S. Chlebus, A. Czumaj, J.F. Sibeyn, Routing on the PADAM: degrees of optimality, in: Proc. Euro-Par'97 Parallel Processing, 1997, Lecture Notes in Computer Science, Vol. 1300, pp. 272–279.
[5] M. Dietzfelbinger, F. Meyer auf der Heide, Dynamic hashing in real time, in: J. Buchmann, H. Ganzinger, W.J. Paul (Eds.), Informatik: Festschrift zum 60. Geburtstag von Günter Hotz, B.G. Teuber Verlagsgesellschaft, Leipzig, 1992, pp. 95–119. A preliminary version appeared as: M. Dietzfelbinger, F. Meyer auf der Heide, A new universal class of hash functions and dynamic hashing in real time, in: Proc. 17th Internat. Colloq. on Automata, Languages and Programming, 1990, pp. 6–19.
[6] K. Hwang, Advanced Computer Architecture: Parallelism, Scalability, Programmability, McGraw-Hill, New York, 1993.
[7] K. Hwang, C.-M. Cheng, Simulated performance of a RISC-based multi-processor using orthogonal-access memory, J. Parallel Distrib. Comput. 13 (1991) 43–57.
[8] K. Hwang, M. Dubois, D.K. Panda, S. Rao, S. Shang, A. Uresin, W. Mao, H. Nair, M. Lytwyn, F. Hsieh, J. Liu, S. Mehrotra, C.M. Cheng, OMP: a RISC-based multiprocessor using orthogonal-access memories and multiple spanning buses, in: Proc. 1990 Internat. Conf. on Supercomputing, 1990, pp. 7–22. ACM SIGARCH Computer Architecture News 18 (3) (1990) 7–22.
[9] K. Hwang, P.-S. Tseng, D. Kim, On orthogonal multiprocessor for parallel scientific computations, IEEE Trans. Comput. 38 (1989) 47–61.
[10] J. JáJá, An Introduction to Parallel Algorithms, Addison-Wesley, Reading, MA, 1992.
[11] H. Kadota, K. Kaneko, I. Okabayashi, T. Okamoto, T. Mimura, Y. Nakakura, A. Wakatani, M. Nakajima, J. Nishikawa, K. Zaiki, T. Nogi, Parallel computer ADENART — its architecture and application, in: Proc. Fifth ACM Internat. Conf. on Supercomputing, 1991, pp. 1–8.
[12] H. Kadota, K. Kaneko, Y. Tanikawa, T. Nogi, VLSI parallel computer with data transfer network: ADENA, in: Proc. Internat. Conf. on Parallel Processing, Vol. I, 1989, pp. 319–322.
[13] R.M. Karp, V. Ramachandran, Parallel algorithms for shared-memory machines, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, Vol. A: Algorithms and Complexity, Elsevier Science Publishers, Amsterdam, 1990, pp. 870–941.
[14] C.P. Kruskal, L. Rudolph, M. Snir, A complexity theory of efficient parallel algorithms, Theoret. Comput. Sci. 71 (1990) 95–132.
[15] F.T. Leighton, Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes, Morgan Kaufman Publishers, San Mateo, California, 1991.
[16] T. Leighton, Methods for message routing in parallel machines, in: Proc. 24th ACM Symp. on Theory of Computing, 1992, pp. 77–96.

[17] T. Leighton, F. Makedon, Y. Tollis, A $2n-2$ step algorithm for routing in an $n \times n$ array with constant size queues, in: Proc. ACM Symp. on Parallel Algorithms and Architectures, 1989, pp. 328–335.
[18] T. Leighton, C.G. Plaxton, A (fairly) simple circuit that (usually) sorts, in: Proc. 31st IEEE Symp. on Foundations of Computer Science, 1990, pp. 264–274.
[19] K. Mehlhorn, U. Vishkin, Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories, Acta Inform. 21 (1984) 339–374.
[20] F. Meyer auf der Heide, Hashing strategies for simulating shared memory on distributed memory machines, in: F. Meyer auf der Heide, B. Monien, A.L. Rosenberg (Eds.), Proc. First Heinz Nixdorf Symposium Parallel Architectures and their Efficient Use, Lecture Notes on Computer Science, Vol. 678, 1992, pp. 20–29.
[21] T. Nogi, Parallel computation on ADENA, in: D.J. Evans, G.R. Joubert, H. Liddell (Eds.), Parallel Computing'91, Elsevier Science Publishers, Amsterdam, 1992.
[22] T. Nogi, T. Imamura, K. Sakai, Y. Obayashi, NAS parallel benchmark results on ADENART, Parallel CFD'94, Kyoto, 1994.
[23] S. Rajasekaran, Mesh connected computers with fixed and reconfigurable buses: packet routing, sorting and selection, in: Proc. European Symp. on Algorithms, Lecture Notes in Computer Science, Vol. 726, 1993, pp. 309–320.
[24] S. Saini, D.H. Bailey, NAS parallel benchmark results 12-95, Technical Report NAS-95-021, NASA Ames Research Center, Moffett Field, CA 94035-1000, 1995. This report is available electronically at http://www.nax.nasa.gov/NAS/TechReports/NASreports/NAS-95-021/NAS-95-021.htm.
[25] I.D. Sherson, Y. Ma, Analysis and applications of the orthogonal access multiprocessor, J. Parallel Distrib. Comput. 7 (1989) 232–255.
[26] J.F. Sibeyn, B.S. Chlebus, M. Kaufmann, Deterministic permutation routing on meshes, J. Algorithms 22 (1997) 111–141.
[27] J.F. Sibeyn, M. Kaufmann, R. Raman, Randomized routing on meshes with buses, in: Proc. European Symp. on Algorithms, Lecture Notes in Computer Science, Vol. 726, 1993, pp. 333–344.
[28] Q. Stout, Mesh connected computers with broadcasting, IEEE Trans. Comput. 32 (1983) 826–830.
[29] L.G. Valiant, General purpose parallel architectures, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, Vol. A: Algorithms and Complexity, Elsevier Science Publishers, Amsterdam, 1990, pp. 943–972.
[30] A.J. van der Steen, J.J. Dongarra, Overview of Recent Supercomputres, NHSA Review, 1996 Volume, 1st issue, 1996. This report is available electronically at http://www.crpc.rice.edu/NHSEreview/ORS/.
[31] R.A. Wagner, Y. Han, Parallel algorithms for bucket sorting and the data dependent prefix problem, in: Proc. 15th Internat. Conf. on Parallel Processing, 1986, pp. 924–930.
[32] A.C. Yao, Probabilistic computations: towards a unified measure of complexity, in: Proc 18th Annual Symp. on Foundations of Computer Science, 1977, pp. 222–227.