1985

On Multidimensional Arrays of Processors

Mikhail J. Atallah
Purdue University, mja@cs.purdue.edu

Report Number:
85-528

Atallah, Mikhail J., "On Multidimensional Arrays of Processors" (1985). *Department of Computer Science Technical Reports*. Paper 447.
<https://docs.lib.purdue.edu/cstech/447>

ON MULTIDIMENSIONAL ARRAYS OF PROCESSORS

Mikhail J. Atallah

CSD-TR-528
July 1985

ON MULTIDIMENSIONAL ARRAYS OF PROCESSORS

Mikhail J. Atallah

Department of Computer Sciences
Purdue University
West Lafayette, Indiana 47907.

Abstract

We investigate the relationship between an arbitrary d -dimensional mesh of n processors, and one all of whose dimensions have equal length. We give optimal simulation algorithms between these two models.

keywords: Analysis of Algorithms, Mesh-Connected Processor Arrays, Parallel Computation

1. Introduction

The d -dimensional mesh of processors is one of the most popular models of parallel computation, and researchers have designed an impressive number of algorithms for solving various problems on this model (e.g. [AK,AH,K1,K2,NS1,NS2,TK], see [U] for a more complete bibliography). Most of these algorithms were designed for a mesh all of whose d dimensions have equal length, i.e. a d -dimensional cube (for convenience, we henceforth say that such a mesh is *square*, even when $d > 2$). Note that every side of such a square mesh has length $n^{1/d}$. (The known algorithms for a square mesh typically run in $O(n^{1/d})$ time.) A d -dimensional mesh which is not square is said to be *rectangular*. Rectangular meshes occur quite naturally in a number of settings: In reference [AH] we ended up working with rectangular meshes even though we started out initially with a square one (this happened because we could not fit our subproblems into subsquares of the original square, and we had to settle for "packing" them into rectangular submeshes of the original square mesh). Also note that a $(d-k)$ -dimensional square mesh is just a special case of a d -dimensional rectangular mesh, one where k of the dimensions have length 1 and the remaining $d-k$ dimensions have equal length.

The purpose of this paper is to investigate the relationship between a rectangular mesh and a square one. In the rest of this section we introduce some terminology, state our results and discuss their significance, and (at the end of the section) briefly review the definition of the d -dimensional mesh. We leave the simulation algorithms that prove our results for sections 2 and 3. Section 4 concludes.

Definition 1.1 Throughout this paper, when a mesh U can simulate every step of another mesh W with $O(h)$ of its own steps, then we say that U *h -simulates* W .

Note that if U h -simulates W then any problem that W solves in time T can be solved by U in time $O(hT)$.

Definition 1.2 Throughout this paper, S will denote a square d -dimensional mesh of n processors, i.e. S is an $n^{1/d} \times \dots \times n^{1/d}$ mesh. R will denote a rectangular d -dimensional mesh of n processors, i.e. R is an $l_1 \times l_2 \times \dots \times l_d$ mesh where $\prod_{i=1}^d l_i = n$.

Theorem 1. Mesh R can $(\max_i l_i)/n^{1/d}$ -simulate mesh S .

(The proof of the above theorem is given in Section 2 of this paper.)

Corollary 1.1 Any problem which S solves in time T can be solved by R in time $O(T \cdot (\max_i l_i)/n^{1/d})$.

Note that if S solves a certain problem in time $O(n^{1/d})$, then the above result implies that R can solve that same problem in time $O(\max_i l_i)$. This shows that Theorem 1 is essentially optimal, since any nontrivial computation on R requires $\Omega(\max_i l_i)$ time.

Except for the trivial case of $d=2$, we find it quite surprising that only the largest of the l_i 's is relevant to how well R simulates S .

The next corollary is obtained from Theorem 1 by setting $l_1 = l_2 = \dots = l_{d-k} = n^{1/(d-k)}$ and $l_{d-k+1} = \dots = l_d = 1$.

Corollary 1.2 An n -processor $(d-k)$ -dimensional square mesh can $n^{k/d(d-k)}$ -simulate an n -processor d -dimensional square mesh.

The simulation result of Theorem 1 is quite useful because it allows us to avoid designing algorithms for rectangular meshes and concentrate on designing algorithms for square meshes instead, a much more pleasant task (designing algorithms for rectangular meshes can be quite awkward, especially recursive algorithms where we want each of the "quadrants" to recursively solve a smaller subproblem). Probably the best way to design an $O(\max_i l_i)$ time algorithm running on R is to actually first design an $O(n^{1/d})$ time algorithm running on S and then use Theorem 1. (In

general, algorithm designers prefer square meshes where $n^{1/d}$ is a power of two.)

Of course our result also implies that all the known algorithms for a square mesh immediately imply corresponding algorithms for rectangular meshes of various shapes. We therefore automatically have $O(\max_i l_i)$ time algorithms for solving a large number of problems on rectangular mesh R .

Finally, it is natural to consider the simulation of rectangular mesh R by square mesh S . The next theorem settles this issue.

Theorem 2 Mesh S can 1-simulate mesh R .

(The proof of the above theorem is given in Section 3 of this paper.)

In other words, whatever the problem being solved, S is at least as good as R . The optimality of Theorem 2 can best be seen by noting that there are problems whose time complexity is $\Theta(n)$ on both mesh S and mesh R . One such problem is that of computing $a_1 \oplus (a_2 \oplus (\dots \oplus (a_n))) \dots$ where $\oplus$ is not associative. This last example shows that it is impossible to guarantee that S will be faster than R for *all* problems, however it does not rule out that S will be faster for *some* problems.

To make the paper self-contained, we end this section by reviewing the definition of a d -dimensional mesh of processors (the reader familiar with this model should skip this and go directly to Section 2). The processors, which operate synchronously, are positioned on an $l_1 \times \dots \times l_d$ grid, one processor per grid point. A processor is denoted by its position in the grid, e.g. processor $(i_1, \dots, i_d)$ where $1 \leq i_k \leq l_k$ for every $k \in \{1, \dots, d\}$. Processor $(i_1, \dots, i_d)$ is a *neighbour* of processor $(j_1, \dots, j_d)$ if and only if, for some $k \in \{1, \dots, d\}$, we have $|i_k - j_k| = 1$, and $i_s = j_s$ for every $s \neq k$. We then say that these two processors are *neighbours along dimension k* . Note that a processor can have no more than $2d$ neighbours (processors at the boundary have less). A *step* of such a mesh consists either of each processor communicating with a neighbour by sending/receiving the contents of a register (we call this a

data movement step), or of each processor performing a computation within its own registers. A processor has a fixed (i.e. $O(1)$) number of storage registers. Some papers in the literature assume that a register can store up to $\log n$ bits, while other papers limit the size of a register to $O(1)$ bits: our results hold for either model.

2. Proof of Theorem 1

Recall that we want to prove that R can $(\max_i l_i)/n^{1/d}$ -simulate S , where S and R are as in Definition 1.2. Before giving the proof of this, we need the following two rather straightforward lemmas.

Lemma 2.1 Let U be a $u_1 \times \dots \times u_d$ mesh of n processors, and let W be a $(d-1)$ -dimensional n -processor mesh which is obtained from U by replacing two of the dimensions of U by a dimension whose length is the product of their two lengths, while leaving the other $d-2$ dimensions of U unchanged. In other words W is a $u_1 \times \dots \times u_{d-2} \times (u_{d-1}u_d)$ mesh (we chose to multiply u_{d-1} and u_d purely for notational convenience, and we could have chosen any u_i and u_j instead). Then U can 1-simulate W .

Proof: Figure 2.1 illustrates, for the case $d=3$, how the new dimension in W (Fig 2.1a) is 1-simulated by the two old ones in U (Fig. 2.1b): we just imbed linear chains of length $l_2 l_3$ (in W) into $l_2 \times l_3$ rectangles (in U), by "snaking" them as depicted in Fig. 2.1 (in that figure there are actually l_1 such chains, even though only one is shown). In general, there would be $l_1 l_2 \dots l_{d-2}$ such chains, each of which is of length $l_{d-1} l_d$ and is snaked in an $l_{d-1} \times l_d$ rectangle. $\square$

The previous lemma will be used in the inductive step of the proof of Theorem 1.

1. The next lemma provides the basis for the induction.

Lemma 2.2 Let U be an $l_1 \times l_2$ mesh of n processors. Let W be a $\sqrt{n} \times \sqrt{n}$ mesh.

Figure 2.1. Illustrating how a column of W is imbedded into U .

Then U can $\max(l_1, l_2)/\sqrt{n}$ -simulate W .

Proof: Without loss of generality, assume that $l_1 \geq \sqrt{n} \geq l_2$. Think of W as consisting of $\sqrt{n}$ adjacent columns of length $\sqrt{n}$ each (see Fig. 2.2a).

Figure 2.2. Illustrating how the columns of W are imbedded into U .

Now, snake these columns one after the other in U , as depicted in Fig. 2.2b. Note that each such snaked column occupies a width of $\sqrt{n}/l_2 (=l_1/\sqrt{n})$ along the l_1 direction of U (see the note following this proof). A data movement between

adjacent processors in the same column of W can clearly be simulated in $O(1)$ steps on U . It is trivial to design a data movement taking $O(l_1/\sqrt{n})$ on U and which simulates a data movement between adjacent processors on the same row of W . $\square$

Note. In the above proof, it may seem at first glance that we might run out of space in U before having imbedded all of W , because $l_1/\sqrt{n}$ is generally not an integer and therefore some columns of U are only partially used (see Fig. 2.2b). This poses no problem since we can then "bounce back" at the last column of U and start imbedding backward (i.e. right to left) until we have imbedded all of W . This may result in some processors of U having to simulate two processors of W , but this is acceptable. Actually, Definition 1.1 implies that we can "stretch" the dimensions of a mesh by a constant factor without changing its computational power (in the O sense), i.e. an $l_1 \times l_2$ mesh and a $(c.l_1) \times l_2$ mesh can 1-simulate each other if c is a constant. In the rest of the paper we avoid elaborating on such fine details, and concentrate on conveying to the reader the main ideas.

The above lemma (2.2) was so easy because choosing l_1 automatically determines l_2 . This is no longer the case for $d \geq 3$, so the above proof does not directly generalize to higher dimensions. Instead we have to use induction on d , with Lemma 2.2 providing us with the basis of our induction. The rest of this section gives the inductive step of the proof of Theorem 1.

Inductive Step: Without loss of generality, assume that R is such that $l_1 \geq l_2 \geq \dots \geq l_d$. Note that $l_1 \geq n^{1/d} \geq l_d$. Now, divide R along its first dimension into $n^{1/d}$ consecutive submeshes (which we call *R-chunks*) each of which is an $(l_1/n^{1/d}) \times l_2 \times \dots \times l_d$ mesh of $n^{1-1/d}$ processors. Similarly divide S along (say) its first dimension into $n^{1/d}$ submeshes (which we call *S-chunks*) each of which is a $1 \times n^{1/d} \times \dots \times n^{1/d}$ mesh of $n^{1-1/d}$ processors. From here on we ignore the first dimension (of length 1) of an *S-chunk*, i.e. we consider it to be a $(d-1)$ -dimensional

square mesh of $n^{1-1/d}$ processors. Now, use each R -chunk to simulate an S -chunk. Of course, for this simulation, the S -chunks are assigned to the R -chunks in consecutive order. Before continuing with how an R -chunk simulates an S -chunk, let us pause to observe that two processors of S that are neighbours along S 's first dimension are in two consecutive S -chunks, and that one data movement in S between such processors can be simulated in R by a data movement taking time $O(\text{width of an } R\text{-chunk along its first dimension})$, i.e. $O(l_1/n^{1/d})$ (as in Lemma 2.2, we omit the detailed specification of this easy data movement). We still need to show that a data movement step of S along its second, or third, ..., or d th dimension, can also be simulated by $O(l_1/n^{1/d})$ steps of R . Since each such data movement is between processors in the same S -chunk, it suffices to show that an R -chunk can $l_1/n^{1/d}$ -simulate an S -chunk. We cannot yet use the induction hypothesis because S -chunks are $(d-1)$ -dimensional while R -chunks are d -dimensional. This is where Lemma 2.1 comes in: Let C be an $(l_d l_1/n^{1/d}) \times l_2 \times \dots \times l_{d-1}$ mesh, and observe that (by Lemma 2.1) an R -chunk can 1-simulate C . Therefore it suffices to show that C can $l_1/n^{1/d}$ -simulate an S -chunk. But C and an S -chunk satisfy the induction hypothesis, and therefore C can $\max(l_1 l_d/n^{2/d}, l_2/n^{1/d})$ -simulate an S -chunk. If $l_2/n^{1/d} \geq l_1 l_d/n^{2/d}$ then C can $l_2/n^{1/d}$ -simulate an S -chunk, and since $l_1 \geq l_2$ it follows that C can $l_1/n^{1/d}$ -simulate an S -chunk. If, on the other hand, $l_2/n^{1/d} < l_1 l_d/n^{2/d}$, then C can $l_1 l_d/n^{2/d}$ -simulate an S -chunk. Since $l_d/n^{1/d} \leq 1$, this implies that C can $l_1/n^{1/d}$ -simulate an S -chunk. $\square$

This completes the proof of Theorem 1.

3. Proof of Theorem 2.

Recall that we want to prove that S can 1-simulate R , where S and R are as in Definition 1.2. The proof is by induction on d . Throughout the proof we assume,

without loss of generality, that $l_1 \geq l_2 \geq \dots \geq l_d$.

Basis of induction: If $d=2$ then imbed R in S as follows. Partition S into $\sqrt{n}/l_2$ ($=l_1/\sqrt{n}$) rectangular slabs of dimensions $l_2 \times \sqrt{n}$ each, and then snake R through these slabs in the manner depicted in Fig. 3.1. Note that because of the way R moves from one slab of S to the next, some of the processors of S are idle (those in the empty triangular regions), while other processors of S are each simulating two of R 's processors. It is obvious that this imbedding enables S to simulate one step of R with $O(1)$ of its own steps.

Figure 3.1. Illustrating the basis of the induction.

Inductive Step: Think of l_1 as being the *depth* of R , and $l_2 \times \dots \times l_d$ as being its *base*.

Let θ be the number of processors of R 's base, i.e. $\theta = \prod_{i=2}^d l_i$ ($=n/l_1$). Observe that $\theta \leq n^{1-1/d}$. Partition the $(d-1)$ -dimensional, $n^{1-1/d}$ -processor square determined by the last $d-1$ dimensions of S into $n^{1-1/d}/\theta$ $(d-1)$ -dimensional squares of θ processors each. Fig. 3.2 illustrates for the case $d=3$. This partition induces a partition of S itself into $n^{1-1/d}/\theta$ rectangular *slabs* each having a depth of $n^{1/d}$ along S 's first dimension, and a $(d-1)$ -dimensional square base of θ processors (i.e. each slab is a

Figure 3.2. The small $\sqrt{l_2 l_3} \times \sqrt{l_2 l_3}$ squares in (b) are the bases of the slabs.

$n^{1/d} \times \theta^{1/(d-1)} \times \dots \times \theta^{1/(d-1)}$ rectangle). Now, imbed the base of R into the base of a "corner" slab in S (e.g. into the lowest-leftmost small square in Fig. 3.2b). By the induction hypothesis, this can be done such that the base of the slab 1-simulates the base of R . Next, with its own base imbedded in that of a slab, snake R back and forth through the slabs of S until it is completely imbedded in S . We do not elaborate on this since it is an obvious generalization of the snaking done in Fig. 3.1b.

However there is one point worth mentioning about the above imbedding process: It is crucial that the depth of a slab ($=n^{1/d}$) is no smaller than the length of a base's side ($=\theta^{1/(d-1)}$) in order for R to shift smoothly from one slab to another (this condition was satisfied in Fig. 3.1b since we had $l_2 \leq \sqrt{n}$). A data movement in R along its first dimension can obviously be simulated in $O(1)$ time in S . A data movement along any of the remaining $d-1$ dimensions of R can also be simulated in $O(1)$ time in S , because (by the induction hypothesis) the base of a slab can 1-simulate the base of R . $\square$

4. Concluding Remarks

We gave essentially optimal simulation results between an n -processor, d -

dimensional mesh which is square and one which is rectangular. As corollaries to our results, we obtained simulations between d -dimensional square meshes and $(d - k)$ -dimensional square meshes.

In general, simulation results between various networks of processors are not only interesting but also quite useful, since they enable us to design algorithms on the network we feel more comfortable with (e.g. the square mesh) in spite of the fact that the actual machine on which these algorithms will run is different (e.g. a rectangular mesh).

References

- [AK] M.J. Atallah, S.R. Kosaraju, 'Graph Problems on a Mesh-Connected Processor Array', *JACM*, July 1984, pp 649-667.
- [AH] M.J. Atallah and S.E. Hambrusch, 'Solving Tree Problems on Mesh-Connected Processor Arrays', to appear in *Proc. 26th Annual Symp. on Foundations of Computer Science*, 1985.
- [K1] S.R. Kosaraju, 'Speed of Recognition of Context-Free Languages by Array Automata', *SIAM J. on Computing*, Sept. 1975, pp. 331-340.
- [K2] S.R. Kosaraju, 'Fast Parallel Processing Array Algorithms for Some Graph Problems', *Proc. 11th Annual ACM Symp. on Theory of Computing*, 1979, pp. 231-236.
- [NS1] D. Nassimi, S. Sahni, 'Finding Connected Components and Connected ones on a Mesh-connected Parallel Computer', *SIAM J. on Comp.*, 1980, pp. 744-757.
- [NS2] D. Nassimi, S. Sahni, 'Data Broadcasting in SIMD Computers', *IEEE Transactions on Computers*, 1981, pp 101-106.
- [TK] C. Thompson, H. Kung, 'Sorting on a Mesh-Connected Parallel Computer', *CACM*, 1977, pp 263-271.
- [U] J.D. Ullman, *Computational Aspects of VLSI*, Computer Science Press, 1984.