STRING EDITING ON AN SIMD HYPERCUBE MULTICOMPUTER [+]

Sanjay Ranka and Sartaj Sahni

University of Minnesota

[+] This research was supported in part by the National Science Foundation under grants DCR84-20935 and MIP 86-17374

Abstract

SIMD hypercube algorithms to determine a minimum cost edit sequence to transform one string into another are developed. If the two strings are of length n, our algorithms take $O\left(\sqrt{\frac{nlogn}{p}} + \log^2 n\right)$ time when $n^2 p,\ 1 \le p \le n$, processors are available. When $p^2,\ nlogn \le p^2 < n^2$ processors are available, the complexity of our algorithm is $O\left(\frac{n^{1.5}}{p}\sqrt{\log n}\right)$.

Keywords and Phrases

SIMD hypercube multicomputers, string editing.

1 INTRODUCTION

The input to the string editing problem consists of two strings $A = a_1 a_2 a \cdots a_{n-1}$ and $B = b_1 b_2 b \cdots b_{m-1}$; and three cost functions C, D, and I where:

$C(a_i, b_j)$ = cost of changing a_i to b_j

$D(a_i)$ = cost of deleting a_i from A

$I(a_i)$ = cost of inserting a_i into A

Three edit functions: *change*, *delete* and *insert* are available. C, D, and I give the cost of one application of each of these functions. The *cost* of a sequence of edit functions is the sum of the costs of the individual functions in the sequence. In the *string edit* problem, we are required to find a minimum cost editing sequence that transforms string A into string B.

Wagner and Fischer [WAGN74] obtained an $O(nm)$ time dynamic programming algorithm for the problem. The dynamic programming formulation is in terms of a function *cost* where

$cost(i, j)$ = cost of a minimum cost edit sequence to transform $a_1 a_2 \cdots a_i$ into $b_1 b_2 \cdots b_j$

The following recurrence for *cost* is obtained in [WAGN74]:

$$(1)\quad cost(i, j) = \begin{cases} 0 & i = j = 0 \\ cost(i-1, 0) + D(a_i) & i > 0,\ j = 0 \\ cost(0, j-1) + I(b_j) & i = 0,\ j > 0 \\ cost'(i, j) & i > 0,\ j > 0 \end{cases}$$

$$cost'(i, j) = \min\{cost(i-1, j) + D(a_i),$$
$$cost(i-1, j-1) + C(a_i, b_j),$$
$$cost(i, j-1) + I(b_j)\}$$

Once $cost(i, j),\ 0 \le i < n,\ 0 \le j < m$ has been computed a minimum cost edit sequence may be found by a simple backward trace from $cost(n-1, m-1)$. This

backward trace is facilitated by recording which of the three options for $i > 0, j > 0$ yielded the minimum for each i and j.

The string edit problem is identical to the weighted Levenshtein distance problem [LIU85]. The longest common subsequence problem [WAGN74] and the time warping distance problem [LIU85] are special cases of the string edit problem.

Many researchers have studied parallel solutions to the string edit and related problems. All such attempts begin with the dynamic programming recurrence (1). Cheng and Fu [CHEN87] propose an $n \times m$ VLSI array to compute $cost$ in $O(n + m)$ time. Liu and Fu [LIU85] propose $n \times m$ arrays to solve minimum distance classification, time warping, and weighted Levenshtein distance problems in $O(n + m)$ time. An $O(m)$ processor pipelined architecture for string editing has been proposed by Edminston and Wagner [EDMI87]. This takes $O(n + m)$ time. Champion and Rothstein [CHAM87] solve the longest common subsequence problem in $O(1)$ time using an n^2m processor bus automaton. Ibarra, Pong and Sohn [IBAR88] develop an SIMD hypercube algorithm for the case $n = m$. Their algorithm has complexity $O(n/p + \log^2 n)$ on an SIMD hypercube with $O(n^2p), 1 \le p \le n$ processors. The algorithm is easily modified to work for the case $n \neq m$.

In this paper, we develop an SIMD hypercube algorithm for the string edit problem. This algorithm has time complexity $O(\sqrt{(n \log n)/p} + \log^2 n)$ on an SIMD hypercube with $n^2p, 1 \le p \le n$ processors. Like other previous algorithms for this problem, ours require $O(1)$ memory per processor. It is easily adapted to the case $n \neq m$. Comparing our algorithm to that of [IBAR88], we see that ours is asymptotically superior for $i \le p < n/\log^2 n$ and asymptotically the same for $n/\log^2 n \le p \le n$. It is worth noting that when $p = 1$, our algorithm has complexity $O(\sqrt{n \log n})$ while that of [IBAR88] has complexity $O(n)$. In fact, all pre-

vious parallel algorithms that use $O(n^2)$ processors have complexity $O(n)$. So, our algorithm is the first to be able to solve the string editing problem in less than $O(n)$ time while using only $O(n^2)$ processors.

Using the same strategy as used in our first algorithm, we can obtain another SIMD hypercube algorithm for the case when p^2, $nlogn \le p^2 \le n^2$ processors are available.

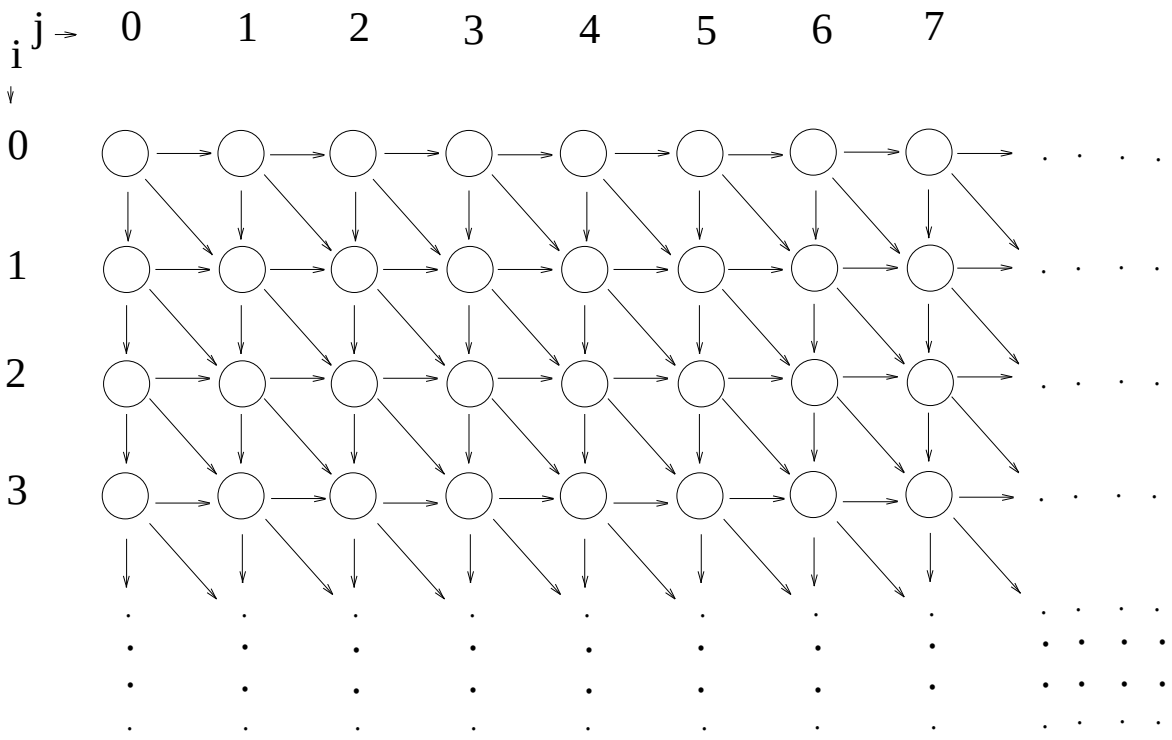


Figure 1 : Lattice graph

2 ALGORITHM OVERVIEW

The dependencies in the dynamic programming formula (1) may be represented by a lattice graph (Figure 1). The vertex in position (i, j) of the lattice graph represents entry (i, j) of the cost matrix of (1). Each edge of the lattice graph is assigned a weight equal to the cost of the corresponding edit

operation. The weights are obtained as follows:

1) The weight of an edge of type $<(e, f), (e, f+1)>$ is $I(b_{f+1})$

2) The weight of an edge of type $<(e, f), (e+1, f)>$ is $D(a_{e+1})$

3) An edge of type $<(e, f), (e+1, f+1)>$ has weight $C(a_{e+1}, b_{f+1})$

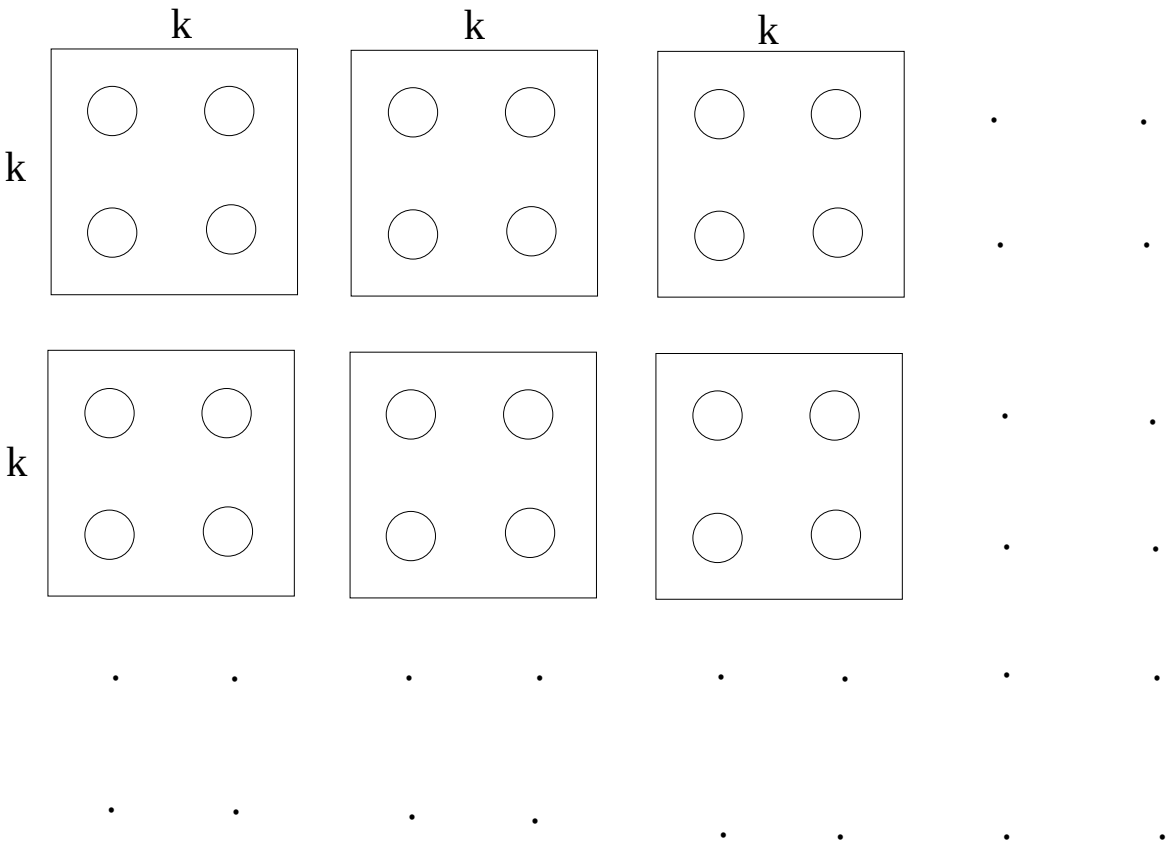


Figure 2 : Decomposition of lattice graph into $k \times k$ subgraphs

It is easy to see that with this weight assignment, $cost(i, j)$ is the length of the shortest path from vertex $(0, 0)$ to vertex (i, j), $0 \le i < n$, $0 \le j < m$. So, the string edit problem is really that of finding a shortest path from vertex $(0, 0)$ to vertex $(n-1, m-1)$. As a result, the string editing problem may be solved in parallel using parallel algorithms previously developed for the shortest path

problem. This will be quite expensive in terms of processor requirement. For example, when $n = m$ we can use the SIMD hypercube shortest path algorithm of [DEKE82] and find a shortest path in $O(log^2 n)$ time. The number of processors required is $O(n^6/logn)$. Our concern here is to find a shortest path using far fewer processors. For convenience, we assume $n = m = 2^q$ for some natural number q in the sequel. The development is easily extended to the case $n \neq m$ and also to the case when n and m are not powers of 2.

Our strategy to find a shortest path from $(0, 0)$ to $(n-1, n-1)$ consists of two phases:

Phase 1: Compute $cost(n-1, n-1)$

Phase 2: Trace back to obtain the path

2.1 Computing $cost(n-1, n-1)$

The first phase itself is accomplished in two stages:

Phase 1, Stage 1: The lattice graph is decomposed into $k \times k$ sublattice graphs for some k that is a power of 2 (Figure 2). The optimal value of k will be determined later. For each $k \times k$ sublattice graph the shortest distance from each vertex on the top and left boundaries to each vertex on the bottom and right boundaries is found.

Phase 1, Stage 2: The boundary to boundary distances computed in Stage 1 are combined to obtain $cost(n, n)$.

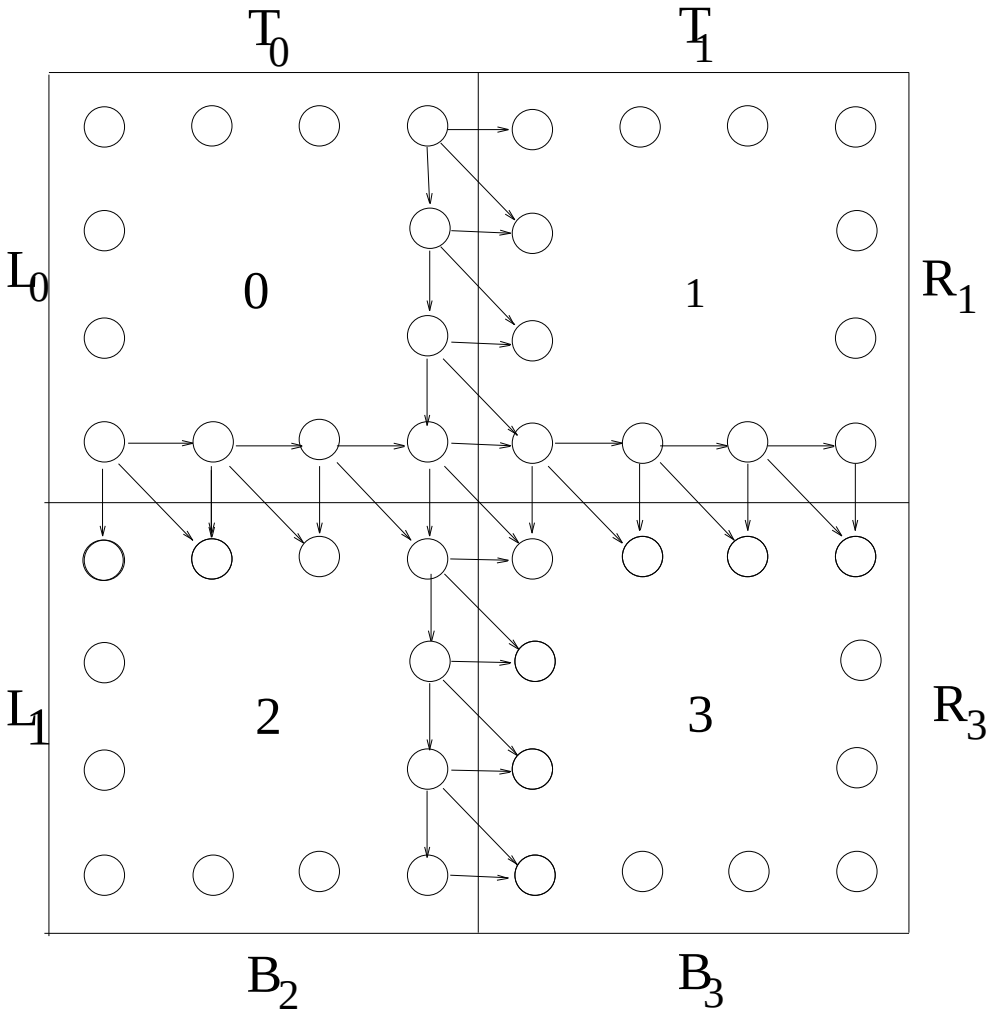


Figure 3 : A 8×8 lattice graph decomposed into 4 4×4 subgraphs

---

2.1.1 Computing Boundary Distances

Boundary to boundary distances may be computed recursively. Figure 3 shows a $2a\times2a$ lattice graph made up of four $a\times a$ lattice graphs. The smaller lattice graphs have been labeled 0–3.

Let T_i, B_i, L_i, and R_i , respectively, denote the top, bottom, left, and right boundary vertices in the smaller lattice graph i, $0 \le i \le 3$. We shall use the notation $XY(i, j)$ to refer to the shortest distance from the i'th vertex in boun-

dary X to the j'th vertex in boundary Y, $X, Y \in \{T_i, B_i, L_i, R_i\}$. Vertices are numbered $0, \ldots, a-1$ left to right for top and bottom boundaries and bottom to top for left and right boundaries. Note that vertex 0 of a top boundary is also the vertex 0 of a left boundary. Similarly, vertex $a-1$ of a top boundary is vertex 0 of a right boundary, etc. $T_0R_0(i, j)$ is the length of a shortest path from the i'th vertex of the top boundary of the lattice graph 0 to the j'th vertex of the right boundary of lattice graph 0.

The boundary distances we are to compute for the $2a \times 2a$ lattice graph are:

$T_0R_1, T_0R_3, T_1R_1, T_1R_3, T_0B_2, T_0B_3, T_1B_2, T_1B_3, L_0R_1, L_0R_3, L_2R_1, L_2R_3, L_0B_2, L_0B_3, L_2B_2,$

and L_2B_3. Because of the edge structure of our lattice graph (Figure 1) we know that all distances in L_2R_1 and T_1B_0 are ∞. From Figure 1, we see that $T_0R_1(i, j)$ is given by:

(2) $$T_0R_1(i, j) = \min\{\min_{0 \le s < a}\{T_0R_0(i, s) + R_0L_1(s, s) + L_1R_1(s, j)\},$$

$$\min_{0 \le s < a-1}\{T_0R_0(i, s) + R_0L_1(s, s + 1) + L_1R_1(s + 1, j)\}\}$$

Since $R_0L_1(s, s)$ and $R_0L_1(s, s + 1)$ are simply edge costs, $T_0R_1(i, j), 0 \le i, j < a$ can be computed if T_0R_0 and L_1R_1 are known. T_0R_0 and L_1R_1 are boundary distances for $a \times a$ lattice subgraphs. The computation of T_0R_3 from $a \times a$ boundary distances is more complex. The equations needed are:

(3) $$T_0R_2(i, j) = \min\{\min_{0 \le s < a}\{T_0B_0(i, s) + B_0T_2(s, s) + T_2R_2(s, j)\},$$

$$\min_{0 \le s < a-1}\{T_0B_0(i, s) + B_0T_2(s, s + 1) + T_2R_2(s + 1, j)\}\}$$

(4) $$T_0R'_3(i, j) = \min\{\min_{0 \le s < a}\{T_0R_2(i, s) + R_2L_3(s, s) + L_3R_3(s, j)\},$$

$$\min_{0 \le s < a-1}\{T_0R_2(i, s) + R_2L_3(s, s + 1) + L_3R_3(s + 1, j)\}\}$$

(5) $$T_0B_1(i, j) = \min\{\min_{0 \le s < a}\{T_0R_0(i, s) + R_0L_1(s, s) + L_1B_1(s, j)\},$$

$$\min_{0 \le s < a-1}\{T_0R_0(i, s) + R_0L_1(s, s + 1) + L_1B_1(s + 1, j)\}\}$$

(6) $T_0R''_3(i, j) = \min\{\min_{0 \le s < a} \{T_0B_1(i, s) + B_1T_3(s, s) + T_3R_3(s, j)\},$

$$\min_{0 \le s < a-1} \{T_0B_1(i, s) + B_1T_3(s, s + 1) + T_3R_3(s + 1, j)\}\}$$

(7) $T_0R_3(i, j) = \min\{T_0R_3'(i, j), T_0R''_3(i, j), T_0B_0(i, a - 1) + B_0T_3(a - 1, 0) + T_3R_3(0, j)\}$

T_1R_1 and L_2B_2 for the $2a \times 2a$ graph are the same as for the corresponding $a \times a$ graphs. The equations for $T_1R_3, T_0B_2, T_1B_2, T_1B_3, L_0R_1, L_2R_3, L_0B_2,$ and L_0B_3 the equations are similar to those for $Tsub0R_3$. For a 1×1 graph,

$$TR(0, 0) = TB(0, 0) = LR(0, 0) = LB(0, 0) = 0$$

Hence the boundary distances for any $k \times k$ lattice subgraph may be computed by computing these distances for 2×2 subgraphs, then for 4×4 subgraphs, then for 8×8 subgraphs, $\cdots$.

Complexity on a CREW PRAM

If the boundary distances for the four $a \times a$ graphs is known, then those for a $2a \times 2a$ graph may be computed in $O(loga)$ time using an $O(a^3/loga)$ processor CREW PRAM. We simply use $a/loga$ processors to compute the distances for each pair (i, j). Hence, to compute the distances for a $k \times k$ graph beginning with those for the 1×1 subgraphs takes:

$$O(\sum_{i=1}^{logk} i) = O(\log^2 k)$$

time.

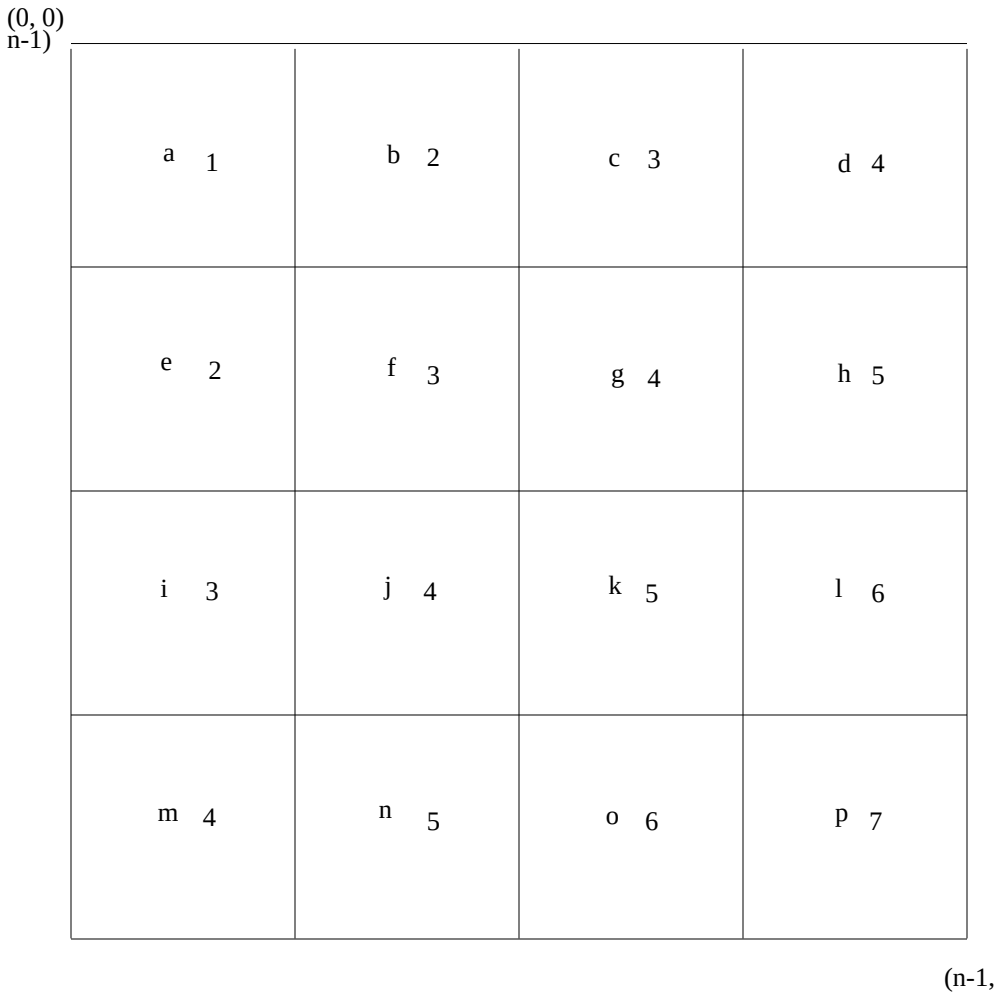


Figure 4 : An $n \times n$ subgaph and its composite $k \times k$ subgraphs $n = 4k$

2.1.2 Obtaining $cost\,(n-1,\, n-1)$

After the boundary distances for each $k \times k$ subgraph have been computed, we compute for each $k \times k$ subgraph the shortest distance from vertex $(0, 0)$ (of the whole graph) to each of the vertices on the top, bottom, left and right boundaries of the $k \times k$ subgraph. Figure 4 shows an $n \times n$ graph and its composite $k \times k$ subgraphs. The figure is for the case $n = 4k$. The $k \times k$ subgraphs are labeled $a - p$.

The shortest distances from $(0, 0)$ to the boundary vertices of $(k \times k)$ subgraphs is computed in several iterations. In iteration i the distances to the boundary vertices of all subgraphs assigned the number i in Figure 4 are computed.

Let $T_iT_i(l, j)$ be the length of the shortest path from the l'th vertex of the top boundary of the $k \times k$ subgraph i to the j'th vertex of its top boundary, $0 \le l \le j < k$. Clearly, $T_iT_i(l, j) = \sum_{r=l}^{j-1} T_iT_i(r, r+1)$ where $T_iT_i(r, r+1)$ is the cost of the directed edge between the top boundary vertices r and $r + 1$. Let $L_iL_i(l, j)$ be the length of the shortest path from the l'th vertex of the left boundary of the $k \times k$ subgraph i to the j'th vertex of its left boundary, $0 \le l \le j < k$. We see that $L_iL_i(l, j) = \sum_{r=l}^{j-1} L_iL_i(r, r+1)$ where $L_iL_i(r, r+1)$ is the cost of the directed edge between the left boundary vertices r and $r + 1$. Let $T_i(j), B_i(j), L_i(j),$ and $R_i(j)$, respectively, denote the shortest distance from vertex $(0, 0)$ to the j'th vertex of the top, bottom, left, and right boundaries of the $k \times k$ subgraph i. For subgraph a of Figure 4, we have:

$$T_a(j) = T_aT_a(0, j)$$

$$L_a(j) = L_aL_a(0, j)$$

$$R_a(j) = T_aR_a(0, j)$$

$$B_a(j) = T_aB_a(0, j)$$

where T_aR_a and T_aB_a are boundary distances computed in Phase 1 , Stage 1 (Section 2.1.1). For subgraphs b and e of Figure 4, we have:

$$L_b(j) = \min\{\min_{0 \le s \le j} \{R_a(s) + R_aL_b(s, s) + L_bL_b(s, j)\},$$

$$\min_{0 \le s < j} \{R_a(s) + R_aL_b(s, s+1) + L_bL_b(s+1, j)\}\}$$

$$T_b(j) = L_b(0) + T_bT_b(0, j)$$

$$R_b(j) = \min_{0 \le s \le j} \{L_b(s) + L_bR_b(s, j)\}$$

$$B_b(j) = \min_{0 \le s \le j} \{L_b(s) + L_b B_b(s,\ j)\}$$

$$T_e(j) = \min\{ \min_{0 \le s \le j} \{B_a(s) + B_a T_e(s,\ s) + T_e T_e(s,\ j)\},$$

$$\min_{0 \le s < j} \{B_a(s) + B_a T_e(s,\ s+1) + T_e T_e(s+1,\ j)\}\ \}$$

$$L_e(j) = T_e(0) + L_e L_e(0,\ j)$$

$$R_e(j) = \min_{0 \le s \le j} \{T_e(s) + T_e R_e(s,\ j)\}$$

$$B_e(j) = \min_{0 \le s \le j} \{T_e(s) + T_e B_e(s,\ j)\}$$

where $R_a L_b$ and $B_a T_e$ are edge costs and $L_b R_b$, $L_b B_b$, $T_e R_e$, and $T_e B_e$ are boundary distances for the respective $k \times k$ subgraphs. The last case to consider is that of subgraph f of Figure 4. For this, we obtain:

$$T_f(0) = L_f(0) = \min\{B_a(k-1) + B_a T_f(k-1,0),$$

$$B_b(0) + B_b T_f(0,\ 0),$$

$$R_e(0) + R_e T_f(0,\ 0)\}$$

$$T_f(j) = \min\{T_f(0) + T_f T_f(0,\ j),$$

$$\min_{0 \le s \le j} \{B_b(s) + B_b T_f(s,\ s) + T_f T_f(s,\ j)\},$$

$$\min_{0 \le s < j} \{B_b(s) + B_b T_f(s,\ s+1) + T_f T_f(s+1,\ j)\}\} \quad , \quad j > 0$$

$$L_f(j) = \min\{L_f(0) + L_f L_f(0,\ j),$$

$$\min_{0 \le s \le j} \{R_e(s) + R_e L_f(s,\ s) + L_f L_f(s,\ j)\},$$

$$\min_{0 \le s < j} \{R_e(s) + R_e L_f(s,\ s+1) + L_f L_f(s+1,\ j)\}\} \quad , \quad j > 0$$

$$R_f(j) = \min\{ \min_{0 \le s \le j} \{T_f(s) + T_f R_f(s,\ j)\},$$

$$\min_{0 \le s \le j} \{L_f(s) + L_f R_f(s,\ j)\}\}$$

$$B_f(j) = \min\{ \min_{0 \le s \le j} \{T_f(s) + T_f B_f(s,\ j)\},$$

$$\min_{0 \le s \le j} \{L_f(s) + L_f B_f(s,\ j)\}\}$$

The $k \times k$ subgraphs of an $n \times n$ lattice graph (Figure 4) may be partitioned into four classes:

(1) Top left corner subgraph (subgraph a of Figure 4)

(2) Remaining top boundary subgraphs (subgraphs b, c, and d of Figure 4)

(3) Remaining left boundary subgraphs (e, i, and m of Figure 4)

(4) All other subgraphs

We see that the formulas obtained above for subgraphs a, b, e, and f can be easily adapted to cover all subgraphs. A close examination of the formulas for a, b, e, and f reveals the following computations are not required.

(1) T and L values of top corner subgraph

(2) T values of the remaining top boundary subgraphs

(3) L values of the remaining left boundary subgraphs

(4) B values of the bottom boundary subgraphs

(5) R values of the right boundary subgraphs (excluding $R_{(k-1)}$ of the bottom right corner subgraph).

Note that $cost(n-1, n-1) = R(k-1)$ of the bottom right corner subgraph.

Complexity on a CREW PRAM

The T_iT_i and L_iL_i values for a $k \times k$ subgraph can be computed in $O(logk)$ time as this is just a prefix sum computation [DEKE83]. The T_i, L_i, B_i, and R_i values of subgraph i can also be computed in $O(logk)$ time as each value requires the computation of the min of $O(k)$ other values. Since all subgraphs with the same numeric label (Figure 4) can be handled in parallel, the total time needed to compute the T_i, L_i, B_i, and R_i values of all the $k \times k$ subgraphs of an $n \times n$ lattice graph is $O(\frac{n}{k}logk)$. The number of processors needed is $O((nk)/logk)$.

The boundary distances of all (n^2/k^2) $k \times k$ subgraph can be computed in $O(\log^2 k)$ time using $O(n^2k/\log k)$ processors (cf. Section 2.1.1.). Hence, we can compute $cost(n-1, n-1)$ in $O(log^2k + \frac{n}{k}logk)$ time on a CREW PRAM with $O(n^2k/\log k)$ processors. When $k = n/\log n$ we get an $O(\log^2 n)$ complexity and an

$O(n^3/\log^2 n)$ processor requirement.

2.2 Traceback

The shortest path from $(0, 0)$ to $(n-1, n-1)$ (i.e., the least cost edit sequence) can be obtained in two stages:

Stage1:Each $k \times k$ subgraph determines the vertex (if any) at which this path enters the subgraph and the vertex (if any) from which it leaves the subgraph.

Stage2:The subgraphs that have an entry and exit vertex determine a shortest path in the subgraph from entry to exit.

2.2.1 Subgraph entry/exit vertices

These can be determined easily if with each $L_i(j)$, $T_i(j)$, $B_i(j)$, and $R_i(j)$ computed in Section 2.1.2 we record 'how' the minimum of the quantities on the right hand side of the respective equation was achieved. So, when computing $B_f(j)$ we will also record a value $(X, u), X \in \{L, T\}, 0 \le u \le j$ such that $B_f(j) = X_f(u) + X_f B_f(u, j)$. Using this information, we begin at $R_z(k-1)$ where z is the bottom right corner subgraph and work our way back to $T_a(0)$ where a is the top left subgraph. For the graph of Figure 4, beginning at $R_p(k-1)$ we obtain the entry vertex for subgraph p. >From this entry vertex and the recorded information, we obtain the exit vertex from subgraphs k, o, or l that was used to get to the entry vertex of p. Suppose this exit vertex is in subgraph l. From B_l we obtain the entry vertex for l and so on.

Because of the edge structure of the graph, exactly one of the subgraphs with any given numeric label (cf. Figure 4) will have an entry and exit vertex.

Complexity on a CREW PRAM

The entry and exit vertices can be computed sequentially by a single

processor in $O(n/k)$ time.

2.2.2 Shortest path in a subgraph

This can be computed if during the computation of boundary distances (Section 2.1.1), we record 'how' each decision is made. Since $O(k^2 \log k)$ decisions are made, this much memory is needed to record the decision information. The actual path computation follows a process similar to that of Section 2.2.1. Entry/exit points in $k/2 \times k/2$ blocks are found; then in $k/4 \times k/4$ blocks; etc.

Complexity on a CREW PRAM

The shortest path in each $k \times k$ subgraph can be found in $O(\log k)$ time using $O(n^2)$ processors. Hence, we can find the shortest path from $(0, 0)$ to $(n-1, n-1)$ in $O(n/k + \log k)$ time. The additional memory requirement is $O(k^2 \log k)$. When $k = n/\log n$, the time is $O(\log n)$ and the additional memory is $O(n^2/\log n)$. Since $O(n^2)$ memory is needed for the $n \times n$ graph the overall memory requirement remains $O(n^2)$.

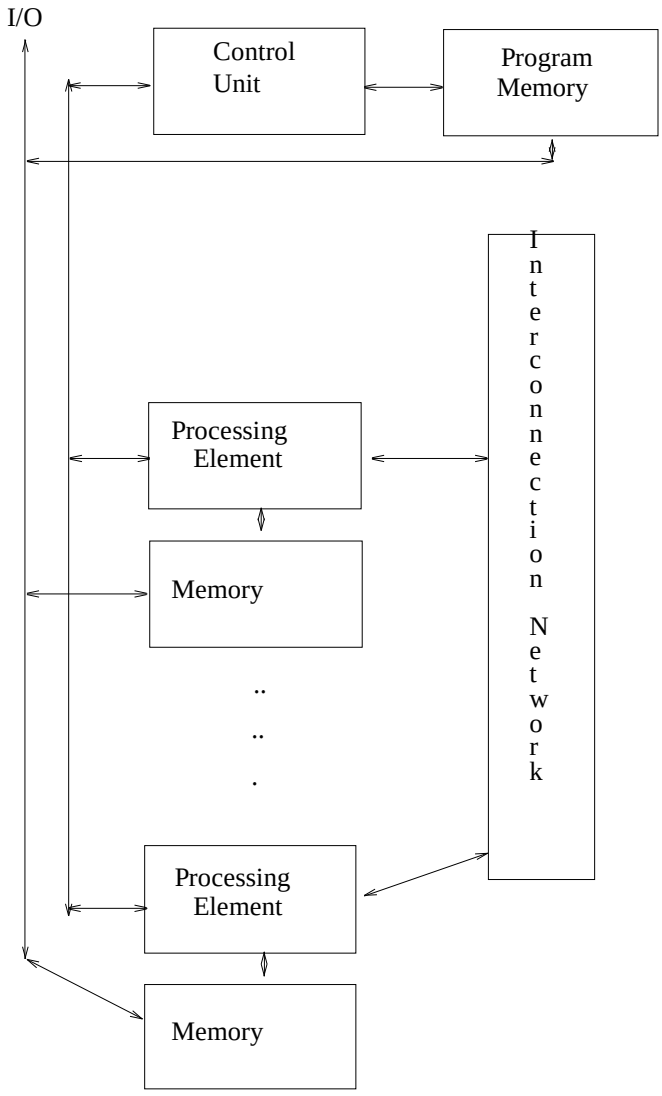


Figure 5 : SIMD hypercube

---

3 HYPERCUBE PRELIMINARIES

3.1 SIMD Hypercube Model

A block diagram of an SIMD hypercube multicomputer is given in Figure 5 . The important features of an SIMD hypercube and the programming notation we use are:

1. There are $P = 2^p$ processing elements connected together via a

hypercube

interconnection network (to be described later). Each PE has the unique index in the range $[0, 2^p - 1]$. We shall use brackets([ ]) to index an array and parentheses('()') to index PEs. Thus A[i] refers to the i'th element of array A and A(i) refers to the A register of PE i. Also, A[j](i) refers to the j'th element of array A in PE i. The local memory in each PE holds data only (i.e., no executable instructions). Hence PEs need to be able to perform only the basic arithmetic operations (i.e., no instruction fetch or decode is needed).

2. There is a separate program memory and control unit. The control unit performs instruction sequencing, fetching, and decoding. In addition, instructions and masks are broadcast by the control unit to the PEs for execution. An *instruction mask* is a boolean function used to select certain PEs to execute an instruction. For example, in the instruction

   $A(i) := A(i) + 1, \quad (i_0 = 1)$

   $(i_0 = 1)$ is a mask that selects only those PEs whose index has bit 0 equal to 1. I.e., odd indexed PEs increment their A registers by 1. Sometimes, we shall omit the PE indexing of registers. So, the above statement is equivalent to the statement:

   $A := A + 1, \quad (i_0 = 1)$

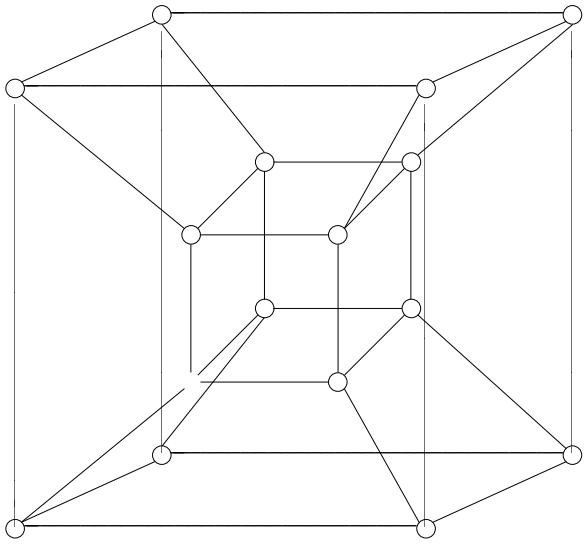


Figure 6 : A 4 dimensional hypercube (16 PEs)

---

3. The topology of a 16 node hypercube interconnection network is shown in Figure 6. A p dimensional hypercube network connects 2^p PEs. Let $i_{p-1}i_{p-2}....i_0$ be the binary representation of the PE index i. Let $\bar{i_k}$ be the complement of bit i_k. A hypercube network directly connects pairs of processors whose indices differ in exactly one bit. I.e., processor $i_{p-1}i_{p-2}...i_0$ is connected to processors $i_{p-1}\cdots\bar{i_k}....i_0$, $0 \le k \le p-1$. We use the notation $i^{(b)}$ to represent the number that differs from i in exactly bit b.

4. Interprocessor assignments are denoted using the symbol $\leftarrow$, while intraprocessor assignments are denoted using the symbol $:=$. Thus the assignment statement:

   $B(i^{(2)}) \leftarrow B(i), \quad (i_2 = 0)$

   is executed only by the processors with bit 2 equal to 0. These processors transmit their B register data to the corresponding processors with bit 2 equal to 1.

5. In a *unit route*, data may be transmitted from one processor to another if it is directly connected. We assume that the links in the interconnection network are unidirectional. Hence at any given time, data can be transferred either from PE i $(i_b = 0)$ to PE $i^{(b)}$ or from PE i $(i_b = 1)$ to PE $i^{(b)}$. Hence the instruction.

   $B(i^{(2)}) \leftarrow B(i), (i_2 = 0)$

   takes one unit route, while the instruction:

   $B(i^{(2)}) \leftarrow B(i)$

   takes two unit routes.

6. Since the asymptotic complexity of all our algorithms is determined by the number of unit routes, our complexity analysis will count only

these.

3.2 Hypercube Embedding of a Grid

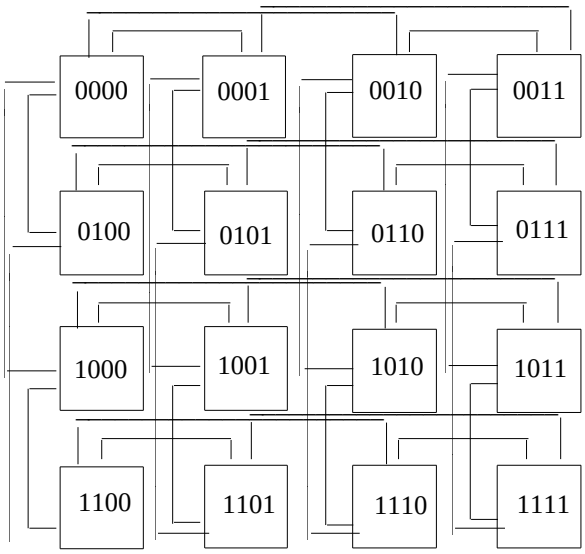


Figure 7: Embedding of a 4×4 mesh in a hypercube of dimension 4

Figure 7 gives a two dimensional grid interpretation of a 4 dimensional hypercube. The index of the PE at position (i, j) of the grid is obtained using the standard row major mapping of a two dimensional array onto a one dimensional array [HORO85]. I.e, for an $N \times N$ grid, the PE at position (i, j) has index $iN + j$. Using this mapping, a two dimensional image grid $I(0..N-1, 0..N-1)$ is easily mapped onto an N^2 hypercube (provided N is a power of 2) with one element of I per PE. Notice that in this mapping, image elements that are neighbors in I (i.e., to the north, south, east, or west of one another) may not be neighbors (i.e., may not be directly connected) in the hypercube. This does not lead to any difficulties in the SIMD algorithms we develop.

3.3 Basic Data Manipulation Operations

3.3.1 SHIFT

SHIFT(A,i,W) shifts the A register data circularly counter-clockwise by i in windows of size W. I.e, $A(qW+j)$ is replaced by $A(qW + (j-i) \bmod W)$, $0 \le q < (P/W)$. *SHIFT*(A, i, W) on an SIMD computer can be performed in $2\log W$ unit routes [PRAS87]. A minor modification of the algorithm given in [PRAS87] performs $i = 2^m$ shifts in $2\log(W/i)$ unit routes [RANK87]. The wraparound feature of this shift operation is easily replaced by an end off *zero* fill feature. In this case, $A(qw + j)$ is replaced by $A(qW + j - i)$, so long as $0 \le j - i < W$ and by 0 otherwise. This change does not increase the number of unit routes.

3.3.2 Column and Row Shift

ColumnShift (A, i, W) shifts the data in the columns of a hypercube down by i. For this purpose, the columns are divided into windows of size W. There is no wraparound and the fill is done using ∞'s. *RowShift* is analogous to *ColumnShift* except that it works on rows of a hypercube and does a leftward shift of i. Both of these procedures are simple adaptations of *SHIFT* and run in the same time.

3.3.3 Prefix Sum

PrefixSum (A, B, W) works on rows of the hypercube. Within each row it considers windows of size W. It computes $A(i) = \sum_{j=0}^{i} B(j)$, $0 \le i < W$ for each such row window. The complexity of this procedure is $O(\log W)$ [DEKE83].

3.3.4 Data Broadcast

Data in one processor of a subhypercube can be broadcast to all processors in that subhypercube in $O(\log S)$ time, where S is the number of processors in the subhypercube. We shall use the operator '$\Leftarrow$' to signify a data broadcast.

4 HYPERCUBE MAPPING

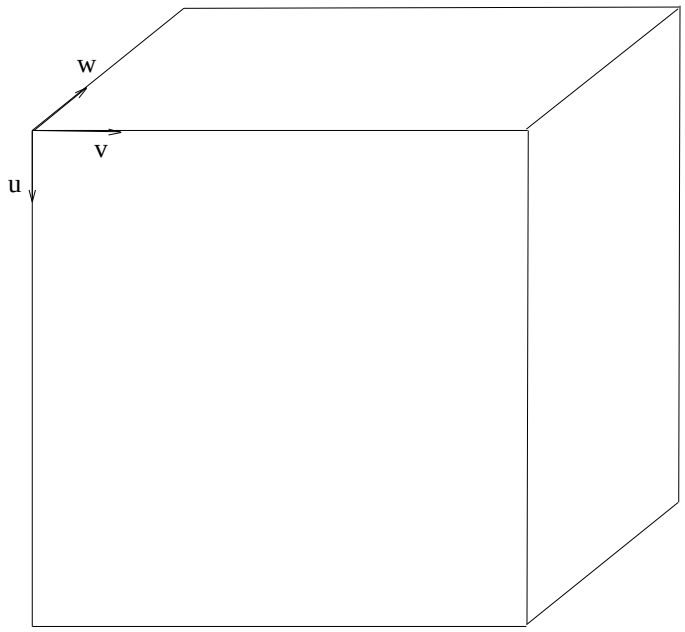


Figure 8: A n^2p hypercube viewed as an $n \times n \times p$ array

4.1 $n^2p,\ 1 \le p \le n$ Processors

First, consider a hypercube with $n^2p,\ 1 \le p \le n$ processors. Such a hypercube can be viewed as an $n \times n \times p$ array (Figure 8). Let $PE(u, v, w)$ denote the processor in position $(u, v, w),\ 0 \le u < n,\ 0 \le v < n,\ 0 \le w \le p$ of this array.

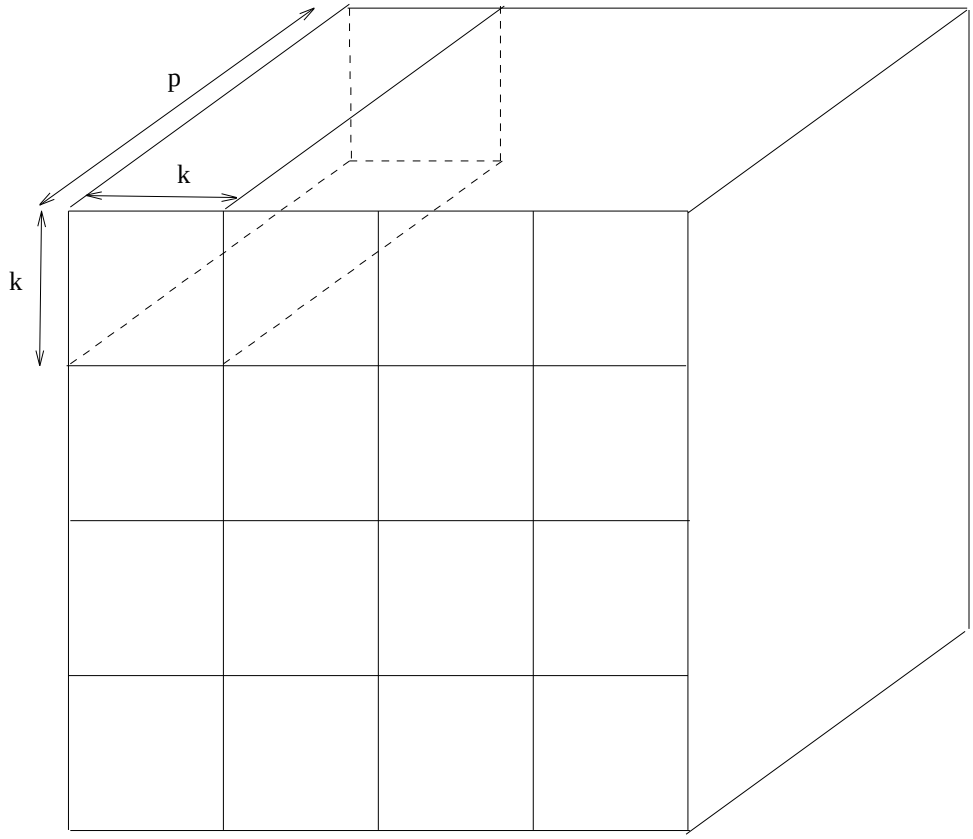


Figure 9: Decomposition into $k \times k \times p$ subhypercubes

---

The $n \times n$ lattice graph is initially mapped onto the face $(u, v, 0)$ of the hypercube. This face is called face 0. $PE(u, v, 0)$ contains the weight of the (at most) three edges coming into vertex (u, v) of the lattice graph, $0 \le u, v < n$. Three registers: *left*, *diagonal*, and *up* are used for this purpose.

$$left(u, v, 0) = \begin{cases} 0 & v = 0 \\ weight((<u, v-1>, <u, v>) & v > 0 \end{cases}$$

$$diagonal(u, v, 0) = \begin{cases} 0 & u = 0 \text{ or } v = 0 \\ weight((<u-1, v-1>, <u, v>) & u > 0 \text{ and } v > 0 \end{cases}$$

$$up(u, v, 0) = \begin{cases} 0 & u = 0 \\ weight((<u-1, v>, <u, v>) & u > 0 \end{cases}$$

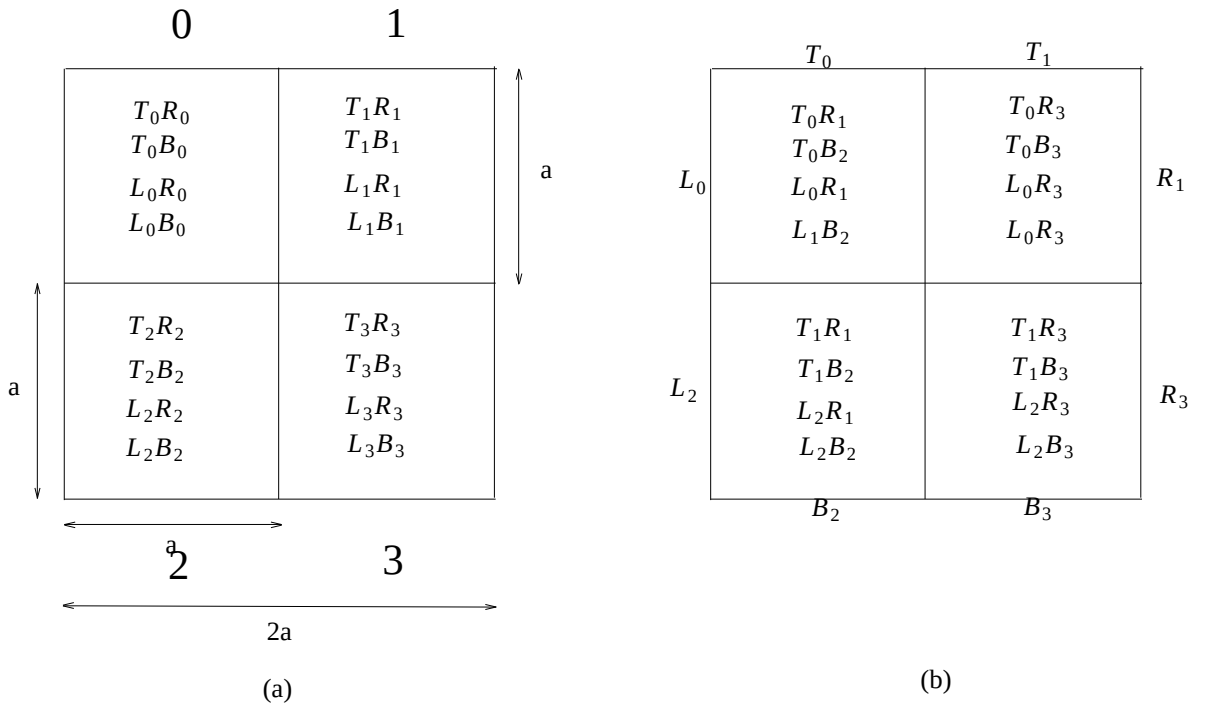
We shall say that processor $(u, v, 0)$ *represents* vertex (u, v) of the lattice graph.

4.1.1 Computing Boundary Distances

Since the computation of the boundary distances for all $k \times k$ subgraphs is done in parallel, we need consider only one of these subgraphs. Under the assumption that k is a power of 2, the processors

$\{(u, v, w) \mid (u, v, 0)$ represents a vertex (u, v) in the $k \times k$ subgraph, $0 \leq w < p \}$

form a $k \times k \times p$ subhypercube (Figure 9).



(a) Initial Configuration

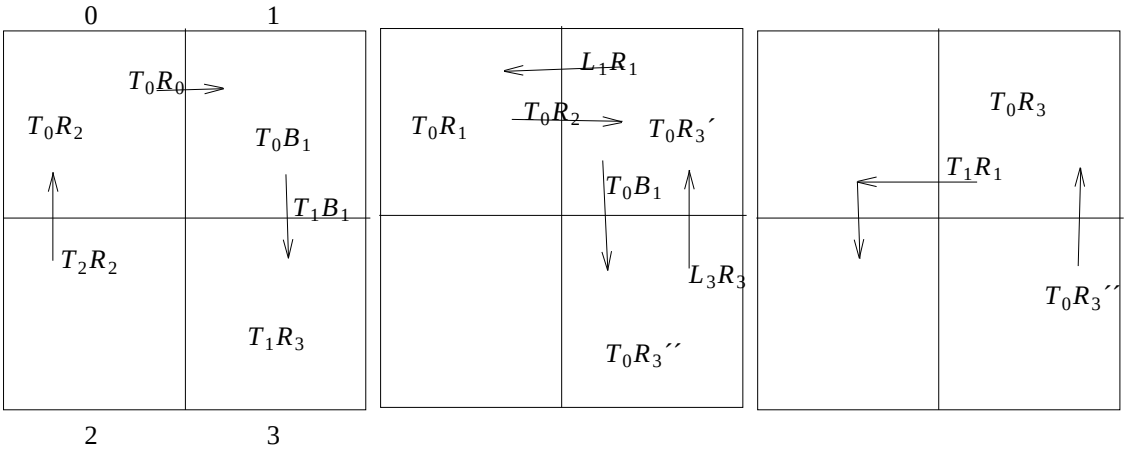
(b) Final Configuration

Figure 10: Initial/Final Coniguration for $2a \times 2a$ subgraphs

The computation of the boundary distances for each $k \times k$ subgraph will be done by the corresponding $k \times k \times p$ subhypercube. To compute the boundary distances for any $a \times a$ subgraph of a $k \times k$ subgraph, the corresponding $a \times a \times p$ subhypercube will be used. Following this computation, the processors on face 0 of these $a \times a \times p$ subhypercubes will contain the boundary distances in

register TR, TB, LR, and LB. Specifically, $XY(i, j, 0)$ will be the shortest distance from the i'th vertex on boundary X to the j'th vertex on boundary Y where $X \in \{T, L\}, Y \in \{R, B\}$, and i and j are relative to the respective $a \times a \times p$ subhypercube (i.e., the top left corner vertex in each such hypercube has $i = j = 0$). When computing for a $2a \times 2a$ subgraph, the initial configuration for face 0 of a $2a \times 2a$ subhypercube is shown in Figure 10(a). I.e., the TR, TB, LR, and LB registers of face 0 of each $a \times a \times p$ subhypercube contain the corresponding boundary distances. Following the computation for the $2a \times 2a$ subgraph the boundary distances are to be distributed as in Figure 10. This will result in the correct initial condition for the computation of boundary distances for $4a \times 4a$ subgraphs.

We explicitly consider only the computation of the new TR values. The computation of the new TB, LR, and LB values is similar. The computation of the TR values is done in three stages. In the first stage the processors on face 0 of the top left $a \times a \times p$ subhypercube compute T_0R_2; on the top right subhypercube T_0B_1; and on the bottom right subhypercube T_1R_3 (Figure 11(a)).



(a) Stage 1 (b) Stage 2 (c) Stage 3

Figure 11: Stages of boundary distance calculation of TR values

The distances computed in the remaining two stages are shown in Figures 11(b) and (c).

Stage 1 Computation

Equations (3) and (5) will be used to compute T_0R_2 and T_0B_1 respectively, The equation for T_1R_3 is:

(8) $$T_1R_3(i, j) = \min\{\min_{0 \le s < a}\{T_1B_1(i, s) + B_1T_3(s, s) + T_3R_3(s, j)\},$$
$$\min_{0 \le s < a-1}\{T_1B_1(i, s) + B_1T_3(s, s+1) + T_3R_3(s+1, j)\}\}$$

Equations (3), (5), and (8) may be rewritten into the form:

$$result(i, j) = \min\{E(i, j), F(i, j)\}$$

where $E(i, j)$ is the result of the $\min_{0 \le s < a}$ part and $F(i, j)$ that of the $\min_{0 \le s < a-1}$ part. The computation of $E(i, j)$ and $F(i, j)$ is very similar to the computation of the product, C, of two $a \times a$ matrices A and B. $C(i, j)$ is given by:

$$C(i, j) = \sum_{0 \le s < a} A(i, s) * B(s, j)$$

Replacing $+$ by min and $*$ by $+$, we get

(9) $$D(i, j) = \min_{0 \le s < a}\{A(i, s) + B(s, i)\}$$

If in (8), we set $A(i, s) = T_1B_1(i, s)$ and $B(s, j) = B_1T_3(s, s) + T_3R_3(s, j)$ or $A(i, s) = T_1B_1(i, s) + B_1T_3(s, s)$ and $B(s, j) = T_3R_3(s, j)$, then

$$D(i, j) = E(i, j).$$

Let $MinSum(A, B, D, a)$ be a hypercube procedure to compute D as in (9) in subhypercubes of size $a \times a \times p$. Such a procedure is easily obtained from a matrix multiplication procedure by using the above transformation. We assume that *MinSum* begins with $A(i, j)$ and $B(i, j)$ in processor (i, j) and leaves $D(i, j)$ in this processor when done.

The algorithm for the stage 1 computation is given in Figure 12. In *Step* 1, we set up the A and B registers of the processors in squares 0, 1, and 3 (cf Figure 11(a)) so that a $MinSum(A, B, E, a)$ will result in $E(i, j)$ as above. For this,

we need:

{square 0}$A(i, j, 0) = T_0B_0(i, j) + B_0T_2(j, j)$

$B(i, j, 0) = T_2R_2(i, j)$

{square 1}$A(i, j, 0) = T_0R_0(i, j)$

$B(i, j, 0) = R_0L_1(i, i) + L_1B_1(i, j)$

{square 3}$A(i, j, 0) = T_1B_1(i, j) + B_1T_3(j, j)$

$B(i, j, 0) = T_3R_3(i, j)$

Step1: [Inititalize to compute $E(i, j)$ in register $E(i, j, 0)$]

{square 0}

$C^0(0, j, 0) \leftarrow up^2(0, j, 0), 0 \le j < a$

$C^0(i, j, 0)$ $fP(sm\ C^0(0, j, 0), 0 \le i, j < a$

$A^0(i, j, 0) := TB^0(i, j, 0) + C^0(i, j, 0), 0 \le i, j < a$

$B^0(i, j, 0) \leftarrow TR^2(i, j, 0), 0 \le i, j < a$

{square 1}

$A^1(i, j, 0) \leftarrow TR^0(i, j, 0), 0 \le i, j < a$

$C^1(i, 0, 0) := left^1(i, 0, 0), 0 \le i < a$

$C^1(i, j, 0)$ $fP(sm\ C^1(i, 0, 0), 0 \le i, j < a$

$B^1(i, j, 0) := C^1(i, j, 0) + LB^1(i, j, 0), 0 \le i, j < a$

{square 3}

$A^3(i, j, 0) \leftarrow TB^1(i, j, 0), 0 \le i, j < a$

$C^3(0, j, 0) := up^3(0, j, 0), 0 \le j < a$

$C^3(i, j, 0)$ $fP(sm\ C^3(0, j, 0), 0 \le i, j < a$

$A^3(i, j, 0) := A^3(i, j, 0) + C^3(i, j, 0), 0 \le i, j < a$

$B^3(i, j, 0) := TR^3(i, j, 0), 0 \le i, j < a$

Step2: [Compute E]

$MinSum(A, B, E, a)$

Step3: [Initialize for F]

{square 0}

$C^0(0, j, 0) \leftarrow diagonal^2(0, j, 0)$

$RowShift(C^0, 1, a)$ {use ∞ fill}

$C^0(i, j, 0) := C^0(0, j, 0), 0 \le i, j < a$

$A^0(i, j, 0) := TB^0(i, j, 0) + C^0(i, j, 0), 0 \le i, j < a$

$ColumnShift(B^0, -1, a)$ {use ∞ fill}

{square 1}

$C^1(i, 0, 0) \leftarrow diagonal^1(i, 0, 0), 0 \le i < a$

$C^1(i, j, 0)$ $fP(sm\ C^1(i, 0, 0), 0 \le i, j < a$

$B^1(i, j, 0) := C^1(i, j, 0) + LB^1(i, j, 0), 0 \le i, j < a$

$ColumnShift(B^0, -1, a)$ *{use ∞ fill}*

{square 3}

$A^3(i, j, 0) := A^3(i, j, 0) - C^3(i, j, 0), 0 \le i, j, < a$

$C^3(0, j, 0) \leftarrow diagonal^3(0, j, 0), 0 \le j < a$

$RowShift(C^3, -1, a)$ *{use ∞ fill}*

$A^3(i, j, 0) := A^3(i, j, 0) + C^3(i, j, 0), 0 \le i, j, < a$

$ColumnShift(B^0, -1, a)$ *{use ∞ fill}*

Step4: [Compute F]
MinSum (*A, B, F, a*)

Step5: [Stage 1 Result]
$S1(i, j, 0) := \min\{E(i, j, 0), F(i, j, 0)\}, 0 \le i, j < a$

Figure 12 : Stage 1 Computation

The notation

$$X^q(i, j, 0)$$

refers to register X of the processor in position $(i, j, 0)$ of square q, $0 \le q \le 3$. So,

$$C^0(0, j, 0) \leftarrow up^2(0, j, 0)$$

denotes a data transfer from the *up* register of the processor in position $(0, j, 0)$ of square 2 to the C register of the processor in position $(0, j, 0)$ of square 0.

In Step2, the E values are computed using *MinSum*. Step3 sets up the A and B registers for the computation of F. This results in:

{square 0}

$A(i, j, 0) = T_0B_0(i, j) + B_0T_2(j, j + 1)$

$B(i, j, 0) = T_2R_2(i + 1, j)$

{square 1}

$A(i, j, 0) = T_0R_0(i, j)$

$B(i, j, 0) = R_0L_1(i, i + 1) + L_1B_1(i + 1, j)$

{square 3}

$A(i, j, 0) = T_1B_1(i, j) + B_1T_3(j, j + 1)$

$B(i, j, 0) = T_3R_3(i + 1, j)$

Note that the *RowShifts* of C^0 and C^3 can be done in parallel and the *ColumnShifts* of B^0, B^1, and B^3 can also be done in parallel. Steps 4 and 5 complete the Stage 1 computation.

Complexity of stage 1

Steps 1 and 3 each take $O(loga)$ time. This is due to the $\Leftarrow$operations and the shifts. The time for Steps 2 and 4 is $O(\frac{a}{z_a} + \log z_a)$, $z_a = \min\{a,p\}$ [DEKE81]. So, the total Stage 1 time is $(\frac{a}{z_a} + \log a)$, $z_a = \min\{a, p\}$.

Stage 2 Computation

This is very similar to the stage 1 computation and can be completed in $O(\frac{a}{z_a} + \log a)$ time.

Stage 3 Computation

T_1R_1 can be moved from square 1 to square 2 using two unit routes by following the path shown in Figure 11(c). The data movements for the computation of T_0R_3 take $O(loga)$ time.

Overall Complexity

The overall time needed to complete boundary distances for $a\times a$ subgraphs is $O(\frac{a}{z_a} + \log a)$, $z_a = \min\{a,p\}$. The time to compute the boundary distances for all $k\times k$ subgraphs is therefore

$$O(\sum_{a = 2, 4, .., k} (\frac{a}{z_a} + \log a)) = O(\frac{k}{p} + \log^2 k).$$

4.1.2 Computing $cost(n-1, n-1)$

The computations of Section 2.1.2 can be performed in $O(\frac{n}{k}\log k)$ time using only those processors that are on face 0 of the hypercube. First, the $k\times k$ subhypercubes of face 0 compute the LL and TT values. The processor in position $(l, j, 0)$ of a $k\times k$ hypercube computes $LL(l, j)$ and $TT(l, j)$. We describe the computation for TT only. The computation of LL is similar.

Step1 [Broadcast $left(0, r, 0)$ over columns]
$T_1 := left(0, r, 0), 0 \le r < k$
T_1 $fP(sm\ T_1, 0 \le s < k$

Step2 [Zero out values not needed]
$T_1 := 0, 0 \le l \le r < k$

Step3 [Compute $TT(l, j, 0)$]
$PrefixSum(TT, T1, k)$ {Prefix sum on rows}

Step4 [Clean up]
$TT(l, j, 0) := \infty, \ 0 \le j < l < k$

Figure 13 : Computing $TT(l, j, 0)$

Computing TT

>From Section 2.1.2 and the definition of $left(i, j, 0)$, we see that $TT(l, j, 0) = \sum_{r=l}^{j-1} left(0, r + 1, 0) = \sum_{r=l+1}^{j} left(0, r, 0)$. Hence, the TT values may be computed in face 0 using the strategy of Figure 13. The computation of TT is viewed as several prefix sum computations; one for each value of l. Steps 1 and 2 set up each row of the $k \times k$ subhypercube so that a prefix sum of the T_1 values on that row will result in the correct TT value. Steps 1 and 3 take $O(logk)$ time and Step 2 takes $O(1)$ time.

Once the TT and LL values have been computed, we proceed to compute the $L, T, B,$ and R values defined in Section 2.1.2. While all the face 0 processors of a $k \times k \times p$ subhypercube are involved in the computation of the $L, T, B,$ and R values for that subhypercube, the final values are stored only in certain processors, The assignment is

$$\left.\begin{array}{l} L(j, 0, 0) = L_i(j) \\ T(0, j, 0) = T_i(j) \\ R(j, k-1, 0) = R_i(j) \\ B(k-1, j, 0) = B_i(j) \end{array}\right\} \ 0 \le j < k$$

Step1 [Bring in R_a values]
Shift in the R values in the $1 \times k$ column of processors to the left of this $k \times k$ subhypercube right by 1. Put the values in the R registers of these processors.

Step2 [Add with *left*() and row broadcast]
$R(s, 0, 0) := R(s, 0, 0) + left(s, 0, 0), 0 \le s < k$
$R(s, j, 0)\ fP(sm\ R(s, 0, 0)\ , 0 \le s, j < k$

Step3 [Compute $L'_b(j)$ in processor $(0, j, 0)$]
$LX(0, j, 0)\ fP(sm \min_{0 \le s < k} \{R(s, j, 0) + LL(s, j, 0)\}$

Step4 [Route to correct processors]
$LX(j, 0, 0)\ fP(sm\ LX(0, j, 0),\ 0 \le j < k$

Figure 14 : Computing LX

---

where i is the label of the corresponding $k \times k$ subgraph (Figure 4). Since the ideas used in the computation of $L, T, R,$ and B are quite similar, we describe only the computation of one part of $L_b(j)$. Specifically, we consider computing:

$$L'_b(j) = \min_{0 \le s \le j} \{R_a(s) + R_a L_b(s, s) + L_b L_b(s, j)\}$$

The value $L'_b(j)$ will be left in register $LX(j, 0, 0)$ of the $k \times k$ subhypercube that represents subgraph b. The strategy to compute L'_b is given in Figure 14. Following Step 1, we have $R(s, 0, 0) = R_a(s), 0 \le s < k$. Following Step 2, $R(s, j, 0) = R_a(s) + R_a L_b(s, s)$. Note that $R(s, j, 0) + LL(s, j, 0)$ is a term in the *min* for $L'_b(j)$. To compute $L'_b(j)$ we need simply find the minimum of all the $R(s, j, 0) + LL(s, j, 0)$ values in column j. This is true as $LL(s, j) = \infty$ for $s > j$. Step 3 computes this minimum in $LX(0, j, 0)$. Step 4 routes the LX values to the proper processors.

Step 1 is a shift of 1 in a window of size $2k$. This takes $O(logk)$ time. Steps 2 and 3 are easily seen to take $O(logk)$ time. Step 4 is a BPC permutation [NASS82] of LX values and can also be done in $O(logk)$ time. Since the L, B, T, R values of all $k \times k$ subgraphs are computed in $O(n/k)$ steps with each step computing these values for some of the $k \times k$ subgraphs, the total time taken to compute the L, B, T, R values for all $k \times k$ subgraphs is $O(\frac{n}{k} logk)$.

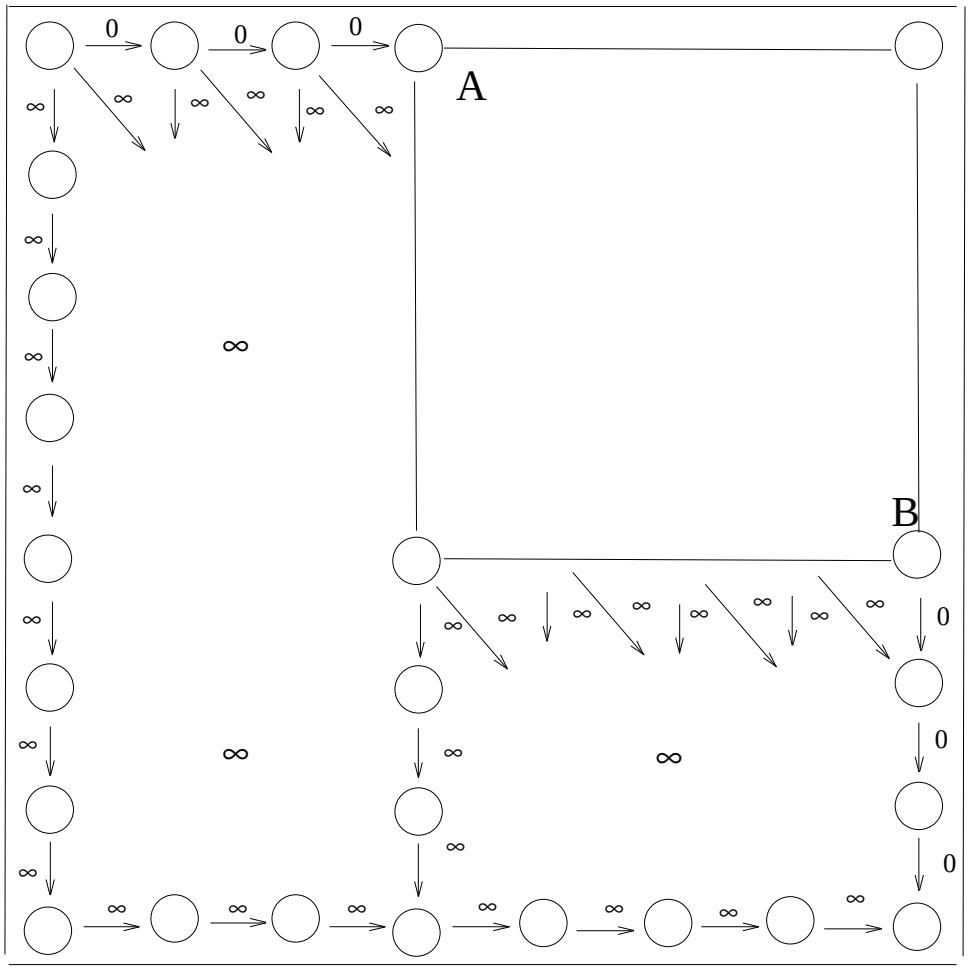
Combining this with the time required to compute the boundary distances, we get $O(\frac{k}{p} + \log^2 k + \frac{n}{k}\log k)$ as the time taken to compute $cost(n-1, n-1)$ beginning with the initial input data. This is minimum when k is approximately $\sqrt{nplogn}$. Substituting this for k, we get $O(\sqrt{\frac{nlogn}{p}} + \log^2 n)$ as the complexity of our algorithm to compute $cost(n-1, n-1)$, The number of processors used is $n^2 p,\ 1 \le p \le n$.

4.1.3 Traceback

When $p \ge logk = log(\sqrt{nplogn})$, $O(1)$ memory per processor is enough to remember the $O(n^2 \log k)$ decisions made during the computation of the boundary distances and the further $O(n^2/k)$ decisions made to compute $cost(n-1, n-1)$ from the boundary distances. The entry/exit points for all the subgraphs can be computed in $O(\frac{n}{k}\log k)$ time using the latter information. The paths within each $k \times k$ subgraph can be constructed in $O(log^2 k)$ time. So, the traceback takes $O(\frac{n}{k}\log k + \log^2 k) = O(\sqrt{\frac{nlogn}{p}} + \log^2 n)$ when $k = \sqrt{nplogn}$.

When $p < logk$, we do not have enough memory to save the decisions made during the computation of the boundary distance. However, we do have enough memory to save the later $O(n^2/k)$ decisions. Hence, the entry and exit vertices of the $k \times k$ subgraphs can still be determined in $O(\frac{n}{k}\log k)$ time. The shortest path from an entry vertex to an exit vertex of a $k \times k$ subgraph may be found by first modifying the edge costs of each $k \times k$ subgraph so that the shortest $(0, 0)$ to $(k-1, k-1)$ path will contain the shortest entry to exit path. Figure 15 shows one of the possible cases for the entry and exit vertices. Edges on the path from $(0, 0)$ to the entry vertex are given a cost of 0. Also those on the path from the exit vertex to

$(k-1, k-1)$ are given a cost of 0. Remaining edges not contained in the rectangle defined by the entry and exit vertices are given a cost of ∞.



A = entry vertex

B = exit vertex

Figure 15 : Modifying a $k \times k$ subgraph

As a result of this transformation, finding a shortest entry to exit path is equivalent to finding a shortest $(0, 0)$ to $(k-1, k-1)$ path. This can be done by recursive application of the algorithm on the $k \times k$ subgraphs. So, we

compute new boundary distances, new $cost(k-1, k-1)$ values, etc.

The run time $T(n, p)$ of the trace back when $p < \log k = \log(\sqrt{nplogn})$ is

$$T(n, p) = O(\sqrt{\frac{nlogn}{p}} + \log^2 n) + T(\sqrt{nplogn}, p)$$

$$= O\left(\sum_i \left(\frac{nlogn}{p}\right)^{\frac{1}{2^i}}\right) = O\left(\sqrt{\frac{nlogn}{p}}\right)$$

So, regardless of whether $p \le logk$ or $p \ge \log k$, the traceback can be completed in $O\left(\sqrt{\frac{nlogn}{p}} + \log^2 n\right)$ time using $n^2 p,\ 1 \le p \le n$ processors and $O(1)$ memory per processor.

4.2 $p^2,\ nlogn \le p^2 < n^2$ Processors

Step 1: Each processor computes the boundary distances for its $\frac{n}{p} \times \frac{n}{p}$ subgraph.

Step 2: *for* $a := 1, 2, 4, ..., k/2$ *do* Each $2a \times 2a$ subhypercube computes the boundary distances for its vertices from those for its constituent $a \times a$ subhypercubes.

Step 3: Combine the boundary distances of $k \times k$ subhypercubes.

Figure 16 : Algorithm for $p^2,\ n\log n \le p^2 \le n^2$ processors

---

The algorithm for this case may be obtained from that for the case of $n^2 p,\ 1 \le p \le n$ processors by first setting p to 1 to get the algorithm for the case of n^2 processors. This simply requires eliminating the third dimension in the earlier algorithm and replacing the *MinSum* procedure with an equivalent procedure for n^2 processors [DEKE81]. Next, we use the ideas of [DEKE81] to go from an n^2 processor algorithm to a p^2 processor algorithm ($1 \le p \le n$). This requires us to map each $\frac{n}{p} \times \frac{n}{p}$ subgraph of the $n \times n$ lattice graph onto a single processor. Hence, each processor of

the $p \times p$ hypercube contains the information corresponding to an $\frac{n}{p} \times \frac{n}{p}$ subgraph.

The algorithm to compute $cost(n-1, n-1)$ takes the form given in Figure 16.

Step1 is done using the serial dynamic programming algorithm for string editing in each processor. The dynamic programming algorithm is used $2n/p - 1$ times; once for each of the $2n/p - 1$ vertices in the top and left boundaries of the $n/p \times n/p$ subgraph. So, in each application the shortest distances from one of these $2n/p - 1$ vertices to all $2n/p - 1$ vertices in the right and bottom boundaries are found. Since each application of the dynamic programming algorithm takes $O(n^2/p^2)$ time, the total time for Step1 is $O(n^3/p^3)$.

Step2 is done using a modified version of the n^2 processor algorithm. This modification is similar to that described in [DEKE81] for matrix multiplication. The time taken is $O\left(\left(\frac{n}{p}\right)^3 \Sigma a\right) = O\left(k\left(\frac{n}{p}\right)^3\right)$.

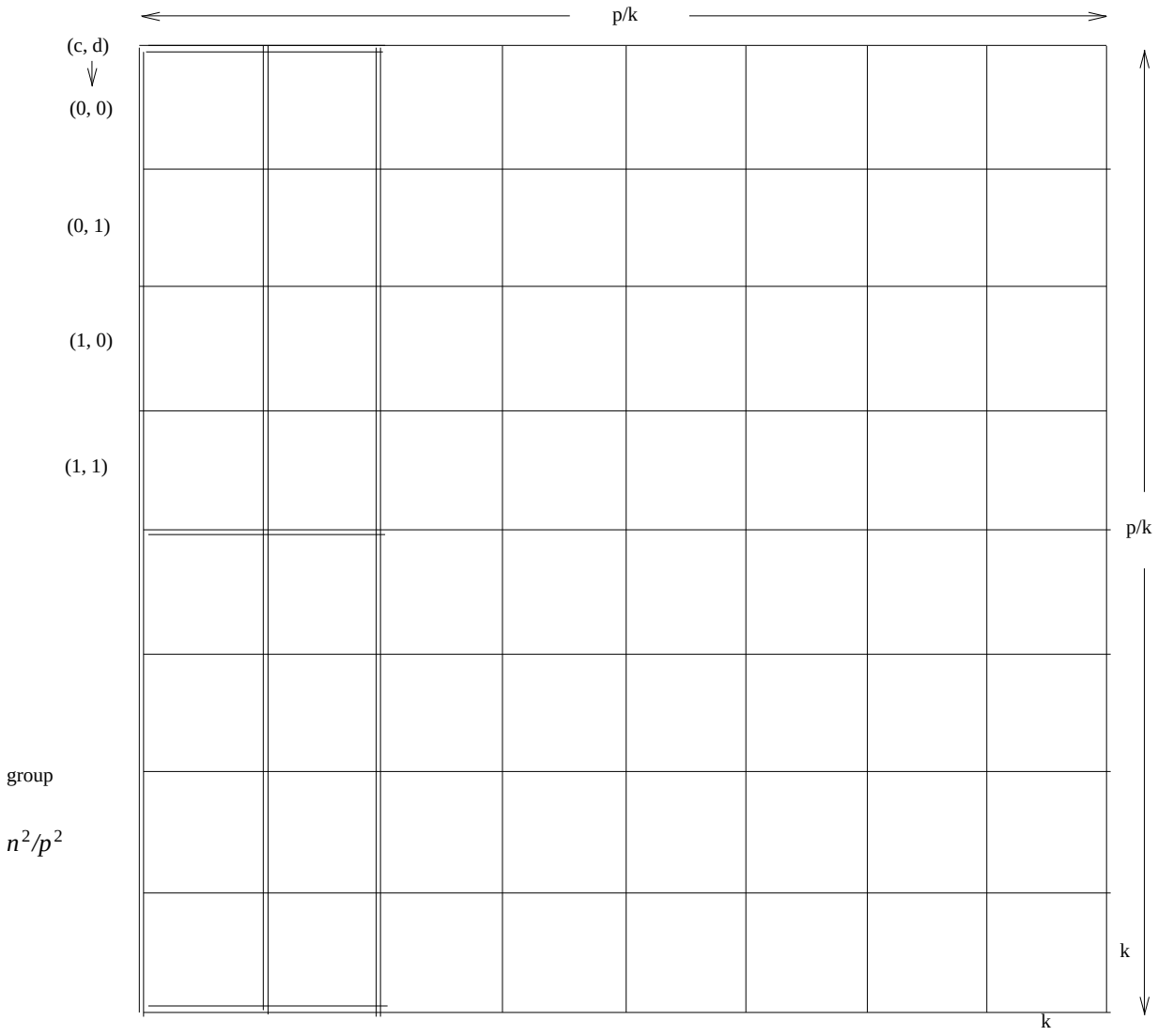


Figure 17 : p^2 processors, $p/k = 8$, $n^2/p^2 = 4$

Following the discussion of Sections 2.1.2 and 4.1.2, the combining of the boundary distances of the $k \times k$ subgraphs is done in $O(p/k)$ iterations. A direct extension of the discussion of Section 4.1.2 will require $O\left(\frac{p}{k}\left(\frac{n^2}{p^2} + \frac{n}{p}\log k\right)\right)$ time be spent on Step 3. However, we note that the $k \times k$ subhypercubes form a $p/k \times p/k$ array (Figure 17). In each iteration of the $O(p/k)$ iterations needed to compute $cost(n-1, n-1)$ at most one $k \times k$

subhypercube of each column of Figure 17 is active. When $p^2 < n^2$, we can put the remaining processors in each column to use. If $\frac{n^2}{p^2} \leq \frac{p}{k}$, then we may group the processors in each column such that each group contains $\frac{n^2}{p^2}$ $k \times k$ subhypercubes from the same column. In Figure 17, $n^2/p^2 = 4$ and $p/k = 8$. The n^2/p^2 $k \times k$ subhypercubes in each group may themselves be viewed as an $n/p \times n/p$ array (2×2 array in Figure 17) which is drawn as a one dimensional array of size n^2/p^2. The pairs (i, j) outside Figure 17 give such an interpretation for the top left group. Hence, we may refer to a processor using the tuple

$[a, b, c, d, e, f]$

where (a, b) indexes the processor group, $0 \leq a < p^3/n^2$, $0 \leq b < p/k$; (c, d) indexes a $k \times k$ subhypercube within a group, $0 \leq c, d < n/p$; and (e, f) indexes a processor within a $k \times k$ subhypercube, $0 \leq e, f < k$. For processors in the top left $k \times k$ subhypercube of Figure 17, $a = b = c = d = 0$. For those in the $k \times k$ subhypercube below this one, $a = b = c = 0, d = 1$. For the bottom left $k \times k$ subhypercube of Figure 17, $a = 1, b = 0, c = d = 1$. Each processor in a $k \times k$ subhypercube represents $\frac{n^2}{p^2}$ (or an $n/p \times n/p$ subgraph) vertices of the original $n \times n$ lattice graph. Let (g, h), $0 \leq g, h < n/p$ index these vertices. We may use the tuple

$[a, b, c, d, e, f, g, h]$

to refer to vertex (g, h) of the (e, f) processor in the $k \times k$ subhypercube (c, d) of group (a, b). In the algorithms of Section 1.1 each vertex was represented by a face 0 processor. Now, each processor contains distances for n^2/p^2 of these face 0 processors. Let $dist[a, b, c, d, e, f, g, h]$ denote these distances for processor (g, h) of the (e, f) processor in the $k \times k$ subhypercube (c, d) of group (a, b) of face 0. Following step 2 of Figure

16, $dist[a, b, c, d, e, f, g, h]$ is in processor $[a, b, c, d, e, f]$. Using the BPC routing algorithm of [NASS82] we can, in $O\left(\frac{n^2}{p^2}\log\frac{n^2k^2}{p^2}\right)$ time rearrange $dist$ such that $dist'[a, b, g, h, e, f, c, d] = dist[a, b, c, d, e, f, g, h]$

Now, the n^2/p^2 $dist$ values formerly in a single processor have been distributed to the $\frac{n^2}{p^2}$ corresponding processors in the same group. As a result of this distribution, we can essentailly use the algorithm of Section 4.1.2 to combine boundary distances in $O(logn)$ time per iteration. So the toatal $Step\,3$ time becomes $O\left(\frac{n^2}{p^2}\log\frac{n^2k^2}{p^2} + \frac{p}{k}logn\right) = O\left(\frac{n^2}{p^2}logn + \frac{p}{k}\log n\right)$ (note that $n^2k^2/p^2 < p^2 < n^2$). The overall time for Figure 16 is therefore $O\left(k\left(\frac{n}{p}\right)^3 + \frac{n^2}{p^2}\log n + \frac{p}{k}logn\right)$. If we set $k = (p^2\sqrt{logn})/n^{3/2}$, then since $\sqrt{nlogn} \le p$, $n^2/p^2 \le p/k$. Further, with this k the run time becomes $O\left(\frac{n^{3/2}}{p}\sqrt{logn}\right)$.

The traceback needed to obtain the shortest $(0, 0)$ to $(n-1, n-1)$ path can be done in the above time by either using $O\left(\frac{n^2}{p^2}\log\frac{p^2}{n}\right)$ memory per processor to store the decisions made during steps 1, 2, and 3 of Figure 16 or by using $O\left(\frac{n^2}{p^2}\right)$ memory per processor to save only the decisions made in Steps 1 and 3 and recomputing those for Step 2 as done in Section 4.1.3.

5 CONCLUSIONS

We have developed efficient SIMD hypercube algorithms for the string edit problem. Our algorithm for n^2p, $1 \le p \le n$ processors is asymptotically superior to that of [IBAR88] for $1 \le p < n/\log^2 n$ and has the same

complexity as theirs for $n/\log^2 n \le p \le n$. In fact, our algorithms achieves an $O(\log^2 n)$ time complexity with $n^3/\log^3 n$ processors. The algorithm in [IBAR88] needs $n^2/\log^2 n$ processors to achieve this time complexity. Further, we have developed the first $O(n^2)$ processor algorithm to solve the string edit problem in less than $\Omega(n)$ time. With n^2 processors, we can solve this problem in $O(\sqrt{n\log n})$ time. The complexities of our algorithms for $n^2 p,\ 1 \le p \le n$ and $p^2,\ n\log n \le p^2 < n^2$ processors may be restated as $O\left(n\sqrt{\frac{\log n}{l}} + \log^2 n\right)$ for $nl,\ \log n \le l \le n^2$ processors.

6 REFERENCES

[CHAM87] D. M. Champion and J. Rothstein, "Immediate parallel solution to the Longest Common Subsequence problem", *Proceedings of 1987 International Conference on Parallel Processing*, **Aug 1987**, pp. 70-77.

[CHEN87] H. D. Cheng and K.S. Fu, " VLSI Architectures for String Matching and Pattern Matching", *Pattern Recognition*, Vol. 20, No. 1, **1987**, pp. 125-141.

[DEKE81] E. Dekel, D. Nassimi and S. Sahni, " Parallel matrix and graph algorithms", *SIAM Journal on computing*, **1981**, pp. 657-675.

[DEKE83] E. Dekel and S. Sahni, "Binary trees and parallel scheduling algorithms", *IEEE Transactions on Computers*, Vol. C-31, No. 3, **March 1983**, pp. 307-315.

[HORO85] E. Horowitz and S. Sahni, "Fundamentals of Data Structures in Pascal", *Computer Science Press*, **1985**.

[NASS81] D. Nassimi and S. Sahni, "Data Broadcasting in SIMD

computers", *IEEE Transactions on Computers*, No. 2, Vol. C-30, **Feb 1981**, pp. 101-107.

[NASS82] D. Nassimi and S. Sahni, "Optimal BPC permutations on a cube connected computer", *IEEE Transactions on Computers*, No. 4, Vol. C-31, **April 1982**, pp. 338-341.

[IBAR88] O. H. Ibarra, T. C. Pong and S. Sohn, "Hypercube algorithms for some string comparision problems", *University of Minnesota Technical Report*, **Jan 1988**.

[LIU85] H. Liu and K. S. Fu, "VLSI Arrays for Minimum-Distance Classifiactions", *VLSI for Pattern Recognition and Image Processing*, **1985**

[WAGN74] R. A. Wagner and M. J. Fischer, "The String-to-string correction problem", *JACM*, Vol. 21, No. 1, **Jan 74**, pp. 168-173.