Optimal Evaluation of Array Expressions on Massively Parallel Machines*

Siddhartha Chatterjee
Research Institute for Advanced Computer Science†

John R. Gilbert
Xerox Palo Alto Research Center‡

Robert Schreiber
Research Institute for Advanced Computer Science†

Shang-Hua Teng
Xerox Palo Alto Research Center§

Abstract

We investigate the problem of evaluating Fortran 90 style array expressions on massively parallel distributed-memory machines. On such machines, an elementwise operation can be performed in constant time for arrays whose corresponding elements are in the same processor. If the arrays are not aligned in this manner, the cost of aligning them is part of the cost of expression evaluation. The choice of where to perform the operation then affects this cost. We present algorithms based on dynamic programming to solve this problem efficiently for a wide variety of interconnection schemes, including multidimensional grids and rings, hypercubes, and fat-trees. We also consider expressions containing operations that change the shape of the arrays, and show that our approach extends naturally to handle this case.

Categories and Subject Descriptors: C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors) — *interconnection architectures, MIMD*; D.3.4 [**Programming Languages**]: Processors — *compilers, optimization*; E.1 [**Data structures**] — *arrays*; G.2.2 [**Discrete Mathematics**]: Graph Theory — *trees*.

General terms: Algorithms, Languages, Theory.

Additional Key Words and Phrases: distributed-memory parallel processors, data-parallel programming, Fortran 90, compact dynamic programming, array alignment, fixed-topology Steiner tree.

---

*An extended abstract of this paper was presented at the *Second Workshop on Languages, Compilers, and Runtime Environments for Distributed Memory Multiprocessors*, Boulder, Colorado, September 30–October 2, 1992.

†The work of these authors was supported by the NAS Systems Division via Cooperative Agreement NCC 2-387 between NASA and the Universities Space Research Organization (USRA).

§Author's current address: Department of Mathematics, Massachusetts Institute of Technology, Cambridge, MA 02139. This work was done while this author was a postdoctoral scientist at Xerox PARC.

1 Introduction

Data parallelism has emerged as a model of parallel computation appropriate for a wide variety of problems in many application domains. As parallel computer architectures proliferate, the need for a stable and portable software base becomes more critical. Data parallelism has many excellent characteristics that make it suitable as the implementation medium for portable parallel software. Languages such as Fortran 90 [1], Fortran D [8], Vienna Fortran [3], and High Performance Fortran [11] are aimed at establishing common ground in array-based data-parallel programming. Languages such as C* [19] and NESL [2] extend the data-parallel model to nested and irregular collections of data.

The two major issues in an efficient implementation of a data-parallel language on a massively parallel machine are the alignment and layout of data on the machine, and the mapping and scheduling of tasks. This is particularly critical for distributed-memory machines, where the cost of accessing nonlocal memory is high. The current state of the art is that data mappings are either supplied by the user, or default canonical data mappings are used; and task mappings are derived from data mappings using the "owner-computes" rule [12]. This has the following disadvantages:

- The best mapping for a problem changes with the interconnection structure of the machine, the size of the machine, and the size of the problem [21]. This means that the user must either write new mappings for each new machine, or be satisfied with poor performance.
- Even if the programmer maps named variables, the compiler must still map temporaries. Mapping according to the owner-computes rule can cause unnecessary communication [14].
- The best data and task mappings may change in different portions of a program. None of the mechanisms in use today deal adequately with the notion of the temporal extent of a data mapping. Having to deal with this explicitly would complicate the programmer's job still further.

We believe that compilers should address the problem of data and computation mapping. Some early work in this direction is the following.

- Knobe, Lucas, and Steele [15] developed a heuristic for data distribution based on alignment preferences for arrays. Their work addresses issues of automatic distribution, but is restricted by the use of the owner-computes rule.
- Wholey [21] demonstrated that optimal mappings are often non-obvious even for a simple array-based programming model. He further showed that a compiler can derive good mappings without programmer intervention. His algorithm used a combination of clustering and hill-climbing techniques.
- Gilbert and Schreiber [9] developed an optimal algorithm for the alignment of temporaries in an expression tree involving array sections by characterizing interconnection networks as metric spaces. Their algorithm applies to a class of so-called "robust" metrics, including multidimensional grids, hypercubes, and the discrete metric. They showed that the presence of common subexpressions makes the problem NP-complete; this makes heuristic solutions necessary for basic blocks.

In this paper, we extend the work of Gilbert and Schreiber [9] in two different directions. First, by reformulating the problem, we show how to handle two additional interconnection structures: multidimensional rings and fat-trees [16]. These networks are not robust and therefore cannot be handled by their algorithm. We also show that their algorithms for grids, hypercubes, and the discrete metric can be recreated in our formulation. Second, we relax the restriction that all arrays must be conformable, modeling this by allowing different weights on the edges of the expression tree. We show how to extend our algorithms to this case as well.

1.1 Basic assumptions

Here we set the ground rules for our work by describing the setting to which it applies.

- We are evaluating a single data-parallel expression or statement as given in the program; commutative or associative rearrangement is not allowed, and there are no common subexpressions. Thus the data flow of the computation can be described by a tree. In a forthcoming paper [4], we extend these algorithms (heuristically) to basic blocks, which can be described as directed acyclic graphs. The algorithms in this paper can be used if the basic block is a forest.
- The only form of parallelism in the evaluation is data parallelism.
- Only one (parallel) array can be communicated at a time. Thus communication along different edges of the expression tree happens at different times, and these costs are additive.
- For the first part of this paper, we ignore considerations of virtualization and stride, and we assume that one data item is allocated to each processor. Section 8 shows how to remove this assumption, and use our techniques for strided array sections distributed in a general block cyclic fashion over the processors. In a general setting, we intend these algorithms to be used for the *alignment* phase, in which all the arrays of a program are mapped to a *template* of virtual processors; then a second *distribution* phase maps the template to the real processors.
- We assume that P, the number of possible positions, is much larger than N, the number of operands in the expression. Thus, for example, any placement algorithm whose running time grows linearly in P is too slow for our purposes; algorithms with running times of $O(N^2)$ or even $O(N^3)$ are preferable.

1.2 Definitions

Throughout the paper, we will let T be a binary expression tree with root node ROOT. The tree T represents a data-parallel expression: the leaves correspond to arrays or array sections (possibly of different shapes), and the internal nodes correspond to binary array operations. Figure 1 contains an example. Let N be the number of leaves of T (operands of the expression); then T has $2N-1$ nodes in all. For any node x, let T_x be the subtree of T rooted at x, and let N_x be the number of leaves in that subtree.

A *position space* is a pair $(\mathbb{P}, d)$, where $\mathbb{P}$ is the set of possible positions, and d is a *distance function* from $\mathbb{P} \times \mathbb{P}$ to the nonnegative real numbers. Let $P = |\mathbb{P}|$. The distance function abstracts the interconnection scheme of the parallel machine: $d(p, q)$ is the cost of moving one data element from position p to position q. We assume that the distance function is a metric:

- **(Symmetry)** $d(p, q) = d(q, p)$.
- **(Nonnegativity)** $d(p, q) \geq 0$; $d(p, q) = 0$ if and only if $p = q$.
- **(Triangle inequality)** $d(p, q) + d(q, r) \geq d(p, r)$.

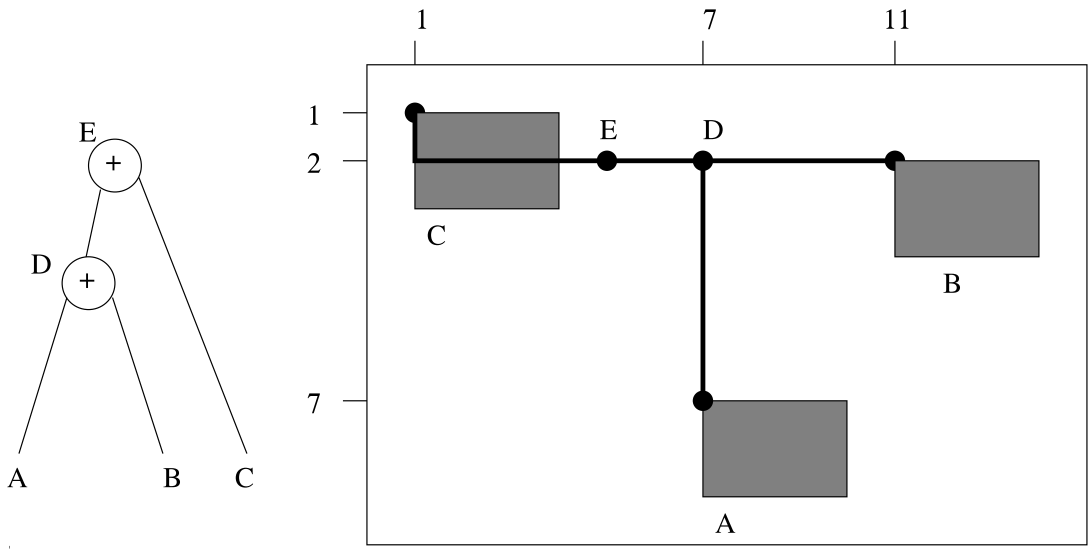


Figure 1: Optimal evaluation of an expression tree. The tree T for the expression $(A + B) + C$ is on the left. The positions of the leaf arrays A, B, and C are the shaded rectangles. The function π encodes the position as the location of the upper left-hand corner of the array. Thus, $\pi(A) = (7, 7)$, $\pi(B) = (2, 11)$, and $\pi(C) = (1, 1)$. With the grid metric, an optimal placement of intermediate nodes is $\pi(D) = (2, 7)$ and $\pi(E) = (2, 5)$, as shown by the heavy dots. The total cost of the operation with these optimal placements of temporaries is 16, as indicated by the heavy lines.

In this framework, the expression mapping problem is to assign a position in $\mathbb{P}$ to each operation (that is, each internal node of T), assuming that positions have been specified for the operands (that is, for the leaves of T). Figure 1 gives an example of an embedding of a tree into a position space that is a two-dimensional grid, where d is the grid or "Manhattan" metric. Let $\pi : \text{NODES}(T) \rightarrow \mathbb{P}$ be a *mapping function* from tree nodes to positions. The value of $\pi(x)$ encodes the position at which to compute the data-parallel operation represented by node x. For example, in Figure 1, $\pi(x)$ is the position of the upper left-hand corner of the array.

We define $C(T, \mathbb{P}, \pi)$ to be the total cost of the communication required to evaluate expression T in position space $\mathbb{P}$ with node placements π.

$$C(T, \mathbb{P}, \pi) = \sum_{(x,y) \in \text{EDGES}(T)} d(\pi(x), \pi(y)).$$

The total cost in Figure 1 is $C(T, \mathbb{P}, \pi) = 16$. Given the values of π at the leaves of the tree, we wish to extend π to the internal nodes so as to minimize $C(T, \mathbb{P}, \pi)$. We call such an embedding a *minimum-cost* embedding. The embedding in Figure 1 is minimum-cost.

Finding a minimum-cost embedding is an instance of the *fixed-topology Steiner tree* problem [22]. Efficient solutions to this problem are known for some metrics, including grids [6, 9], Euclidean distance in the plane [13], and the discrete metric [7, 10, 9]. Our algorithms subsume all these results except the Euclidean, and extend them to metrics based on rings, multidimensional tori, and fat-trees.

1.3 Explicit dynamic programming

All our algorithms originate from a simple dynamic programming approach first used by Mace [18]. Mace's explicit dynamic programming algorithm exploits the structure of the tree but not of the metric; it can find the best embedding of any tree, for any metric, but it is far too expensive in the general case if the number P of possible positions is large. We will show that for each of the metrics we are interested in—grids, rings, tori, fat-trees—we can exploit the structure of the metric to reduce the complexity of Mace's algorithm. The running times of our algorithms depend only on the size of the expression and not on the size of the position space.

The following definitions are fundamental and will be used throughout the paper. Consider an expression tree T and a position space $\mathbb{P}$. We define the *subtree cost function* $C_x(p)$ to be the minimum cost of evaluating the subtree T_x rooted at an internal node x, subject to the constraint that the subtree's root x be placed at position p.

$$C_x(p) = \min_{\pi} \{C(T_x, \mathbb{P}, \pi) | \pi(x) = p\}.$$

We can express the subtree cost function at a node x in terms of the subtree cost functions at its children y and z. It is just a minimum over all possible placements of y and z.

$$C_x(p) = \min_r \Big(C_y(r) + d(p,r) \Big) + \min_r \Big(C_z(r) + d(p,r) \Big). \tag{1}$$

The key observation is that $C_x(p)$ is the sum of two independent terms, one for each subtree below x. To emphasize this, we rewrite the recurrence in terms of the *contribution function* $g_{xy}(p)$, which is the contribution to $C_x(p)$ from the subtree T_y and the edge (x, y). Function g_{xy} is just the $(\text{min}, +)$ convolution of C_y with the distance function. This gives us the following master recurrence.

$$\begin{aligned} C_x(p) &= g_{xy}(p) + g_{xz}(p) \\ g_{xy}(p) &= \min_r (C_y(r) + d(p,r)) \\ g_{xz}(p) &= \min_r (C_z(r) + d(p,r)) \end{aligned} \tag{2}$$

for an internal node x with children y and z, and

$$C_x(p) = d(p, \pi(x)) \tag{3}$$

for a leaf node x. Equation (3) reflects the fact that supplying the value of the fixed leaf x at a given position p requires communicating the value from $\pi(x)$ to p.

Finally, we define a position for y that realizes the minimum contribution $g_{xy}(p)$.

$$r^*_{xy}(p) = \text{some } r \text{ at which } C_y(r) + d(p,r) \text{ is minimized.} \tag{4}$$

Mace's explicit dynamic programming algorithm (shown as Algorithm 1) simply tabulates $C_x(p)$ and $r^*_{xy}(p)$ for all positions p and all nodes x, working from the leaves up to the root; it then assigns positions to the nodes, working from the root down to the leaves. Algorithm 1 works for any distance function. It takes $O(NP^2)$ time and uses $O(NP)$ space.

We are interested in massively parallel machines, for which P is large (much larger than the expression tree size N). Thus the explicit approach is too expensive. In the rest of this paper, we show how to use the properties of some specific distance functions to generate compact representations of subtree cost functions and reduce the cost of dynamic programming to $O(N)$ for grids, $O(Nh)$ for rings and for the discrete metric, and $O(N^3)$ for fat-trees. We call such algorithms *compact dynamic programming* algorithms.

Algorithm 1 (Optimal embedding using explicit dynamic programming.)
Input: An expression tree T, a position space $(\mathbb{P}, d)$, leaf positions $\pi(x)$.
Output: A minimum-cost embedding π of T in $\mathbb{P}$.
Method:

1. Traverse T in postorder, using equations (2) and (4) to tabulate $C_x(p)$ and $r^*_{xy}(p)$ at every non-leaf node x, for all $p \in \mathbb{P}$.

2. Let $\pi(\text{ROOT}) =$ some position at which C_{ROOT} is minimized.

3. Traverse T in preorder, setting $\pi(y) = r^*_{xy}(\pi(x))$ for each internal node y, where $x = \text{PARENT}(y)$.

2 Preliminaries

This section discusses three topics that arise in both the grid and ring metrics. First, we show that multidimensional grids and tori can be treated as simple combinations of independent one-dimensional grids and rings. Second, we show that for one-dimensional grids and rings there is always an optimal embedding in which every internal node is placed at the same position as some leaf. Third, we develop a theory of piecewise-linear, continuous functions on one-dimensional grids and rings.

2.1 Direct sums of distance functions

Complex distance functions can sometimes be decomposed into direct sums of simpler functions. For example, a k-dimensional grid is the direct sum of 1-dimensional grids (that is, linear arrays); a toroidal mesh is the direct sum of two rings; and a k-dimensional hypercube is the direct sum of k two-point discrete metrics. We will see in Section 7 that the direct sum of a discrete metric and a grid metric is also useful.

The formal definition of direct sum metrics is as follows. Let d_i be a real-valued distance function on $\mathbb{P}_i$, for each i from 1 to k. The *direct sum* of the d_i is the distance function d on $\mathbb{P}_1 \times \cdots \times \mathbb{P}_k$ defined by

$$d((p_1, \ldots, p_k), (q_1, \ldots, q_k)) = \sum_{i=1}^{k} d_i(p_i, q_i).$$

We write $d = d_1 \oplus \cdots \oplus d_k = \oplus_{i=1}^{k} d_i$.

The significance of direct sum metrics is that the mapping problem can be decomposed into an independent problem in each summand metric. Thus, multidimensional grids are no harder than linear arrays, tori are no harder than rings, and so forth.

Lemma 1 *Let $(\mathbb{P}_i, d_i)$ be a position space for each i from 1 to k. Let $(\mathbb{P}, d)$ be the position space in which $\mathbb{P} = \times_{i=1}^{k} \mathbb{P}_i$ and $d = \oplus_{i=1}^{k} d_i$. Let T be an expression tree, π an embedding of T in $\mathbb{P}$, and $\pi_i : \text{NODES}(T) \to \mathbb{P}_i$ the projection of π in dimension i. Then π is a minimum-cost embedding of T in $\mathbb{P}$ if and only if every π_i is a minimum-cost embedding of T in $\mathbb{P}_i$.*

Proof: From the definitions, it is immediate that

$$C(T, \mathbb{P}, \pi) = \sum_{i=1}^{k} C(T, \mathbb{P}_i, \pi_i).$$

Since the terms in the sum are independent, the sum is minimized exactly when every term is minimized individually. □

2.2 Continuous grids and rings

For grids and rings, it is convenient to take the set of positions not to be the integers 0 through $P - 1$, but rather the closed real interval $[0, P - 1]$ (for one-dimensional grids) or the half-open real interval $[0, P)$ (for rings). The continuous one-dimensional grid metric is $d(p, q) = |p - q|$; the continuous one-dimensional ring metric is $d(p, q) = \min(|p - q|, P - |p - q|)$. Equations (2), (3), and (4) still hold, but are interpreted in real numbers. The basic theory is the same for rings as for grids with one exception: for rings, we will think of the half-open interval $[0, P)$ as being the real numbers modulo P, so that P is identified with 0. Thus, in the context of rings, addition or subtraction of positions is to be interpreted modulo P.

Of course, for our results to be meaningful on a discrete grid or ring, we need to ensure that all the positions we compute are in fact integers. In this section we prove that any expression tree (even a weighted one as defined in Section 7) can be embedded optimally in a continuous one-dimensional grid or ring with every internal node placed at the same position as a leaf. The significance of this is twofold. First, if leaves are placed at integer locations, we are guaranteed that we can find an optimal position all of whose locations are integers. Second, this restriction reduces the time complexity of Algorithm 1 from $O(NP^2)$ to $O(N^3)$.

We will see in Section 7 that the same result holds for the discrete and fat-tree metrics, and the reader may be tempted to conclude that every metric admits optimal embeddings with all nodes at leaf positions. This is not true, however, even for a two-dimensional grid. In Figure 1 node D cannot be optimally placed at any leaf position, although all projections of node positions are projections of leaf positions.

Theorem 1 *Let T be an expression tree, d be either the one-dimensional continuous grid metric on $[0, P-1]$ or the one-dimensional continuous ring metric on $[0, P)$, and π be an embedding of the leaves of T. Then there is a minimum-cost embedding of the internal nodes of T that agrees with π on the leaves, and that embeds every internal node at the position of some leaf.*

Proof: Extend π to an arbitrary embedding of all the nodes of T. We will show how to modify this extended π so that every internal node is at a leaf position, without moving any leaves or increasing the cost of the embedding.

We ignore the parent-child relationship and consider T to be an undirected tree; two nodes are neighbors if one is the other's parent in the directed tree. A node is a leaf if it has one neighbor, and an internal node if it has more than one neighbor. For reasons that will shortly become clear, we will prove the result for undirected trees whose internal nodes are not restricted to three neighbors.

We proceed by induction on the number of internal nodes of T. The base case is a tree with one leaf, in which case there is nothing to prove. For the induction step, suppose T has some internal node x that is not embedded at a leaf position. There are two cases.

- **Case 1.** Suppose x has some neighbor y with $\pi(x) = \pi(y)$. Since $\pi(x)$ is not a leaf position, y is not a leaf. Form tree T' by merging nodes x and y into a single node x', which inherits all the other

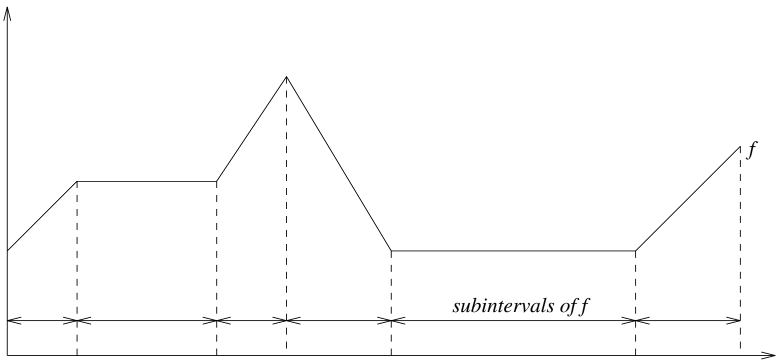


Figure 2: A piecewise linear continuous (PLC) function.

neighbors of both x and y. This tree has the same leaves as T, and one fewer internal node. The embedding of T' that agrees with π everywhere, with node x' placed at $\pi(x) = \pi(y)$, costs the same as the embedding π of T. By the induction hypothesis, there is an embedding π' that agrees with π on the leaves, places all internal nodes of T' at leaf positions, and costs no more than π. We lift π' back to an embedding of T by placing both x and y at position $\pi'(x')$. The result is an embedding that agrees with π on the leaves, places all internal nodes of T at leaf positions, and costs no more than π.

- **Case 2.** Suppose x has no neighbor embedded at $\pi(x)$. Consider moving x from $\pi(x)$ to the left until it first encounters a neighbor y_ℓ, and to the right until it first encounters a neighbor y_r. At most one of those moves increases the cost of the embedding, depending on whether x has more neighbors embedded to the left or to the right of $\pi(x)$. Move x to whichever of $\pi(y_\ell)$ and $\pi(y_r)$ does not increase the cost of the embedding. Now x is embedded at the same position as a neighbor; use the construction in Case 1 to finish the induction step.

□

Most of the algorithms in this paper are guaranteed to find an optimal solution with all nodes as leaf positions, which we assume are integral. Where this is not the case, the construction presented above can be applied in $O(N)$ time to convert the solution the algorithm produces into a solution of the same cost that places all internal nodes at leaf (hence, integer) positions. We will not explicitly refer to this cleanup phase in the rest of the paper.

We remark that Theorem 1 holds (with the same proof) even if we label the edges of T with nonnegative real weights, provided we consider an embedding to be optimal if it minimizes the weighted sum of the edge lengths. We will use this version of the theorem in Section 7.

2.3 Piecewise linear continuous functions

Let f be a *piecewise-linear, continuous function* (or PLC function for short), defined on either $[0, P-1]$ or $[0, P)$. Then f is linear except possibly at a sequence $p_1 \leq \cdots \leq p_k$ of *breakpoints*, which we take to include 0 and $P-1$ in the grid case. We define the *subintervals* of f as the intervals $S_i = [p_i, p_{i+1}]$, for $1 \leq i < k$. In the ring case, there is an additional subinterval $S_k = [p_k, p_1]$. The *slope* m_i of f is constant on the interior of each subinterval. We define $\mathcal{B}(f) = (p_1, \ldots, p_k)$ as the ordered sequence of breakpoints, $\mathcal{S}(f)$ as the ordered sequence of subintervals, and $\mathcal{M}(f)$ as the ordered sequence of slopes. We include the

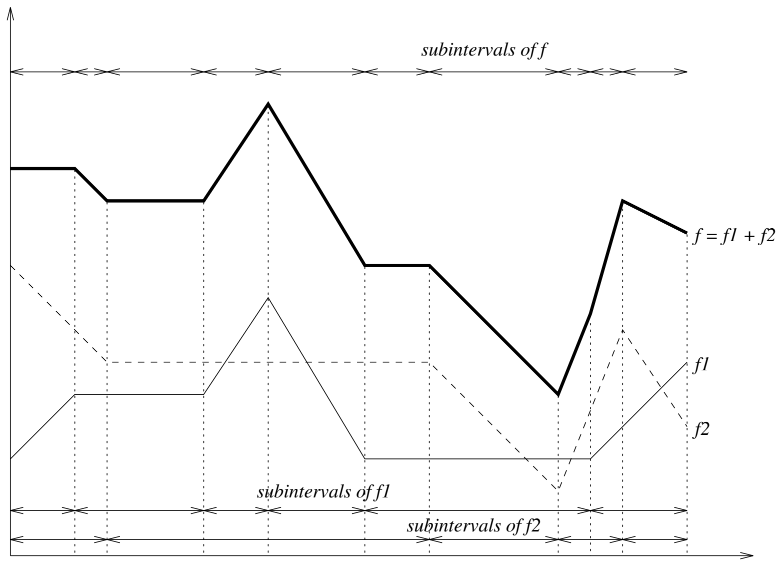


Figure 3: The sum of two PLC functions.

possibility that $p_i = p_{i+1}$, in which case we call the subinterval S_i *degenerate*. We define the slope of a degenerate subinterval as any convenient value. Figure 2 is an example of a PLC function.

The following lemma states some elementary facts about sums of PLC functions; see Figure 3.

Lemma 2 *Let $f = f_1 + f_2$, where f_1 and f_2 are PLC functions defined on the same interval. Then breakpoints can be chosen for f so that:*

1. *f is PLC.*
2. $|\mathcal{S}(f)| \leq |\mathcal{S}(f_1)| + |\mathcal{S}(f_2)|$.
3. $\mathcal{B}(f) \subseteq \mathcal{B}(f_1) \cup \mathcal{B}(f_2)$.
4. *If $m \in \mathcal{M}(f)$ is the slope of f on subinterval $S \in \mathcal{S}(f)$, then $m = m_1 + m_2$ where m_i is the slope of f_i on the unique subinterval $S_i \in \mathcal{S}(f_i)$ such that $S \subseteq S_i, (i = 1, 2)$.*
5. $\max(\mathcal{M}(f)) \leq \max(\mathcal{M}(f_1)) + \max(\mathcal{M}(f_2))$.
6. $\min(\mathcal{M}(f)) \geq \min(\mathcal{M}(f_1)) + \min(\mathcal{M}(f_2))$. □

For each fixed position q, the grid or ring distance function $d(p, q)$ is a real-valued PLC function of one variable p, defined on $[0, P - 1]$ for a grid or $[0, P)$ for a ring. Its slopes are $+1$ and -1. For a grid, its breakpoints are 0, q, and $P-1$; for a ring, it has one breakpoint at q and another at $\Phi(q) = (q + P/2) \bmod P$, which we call the *antipode* of q. Figure 4 shows the two distance functions.

If the distance function has slopes of magnitude at most one, then the subtree cost function has slopes of magnitude at most two.

Lemma 3 *Let d be a PLC distance function with slopes between -1 and $+1$. Let a binary expression tree be given. Then the subtree cost function C_x has slopes of magnitude at most* 2.

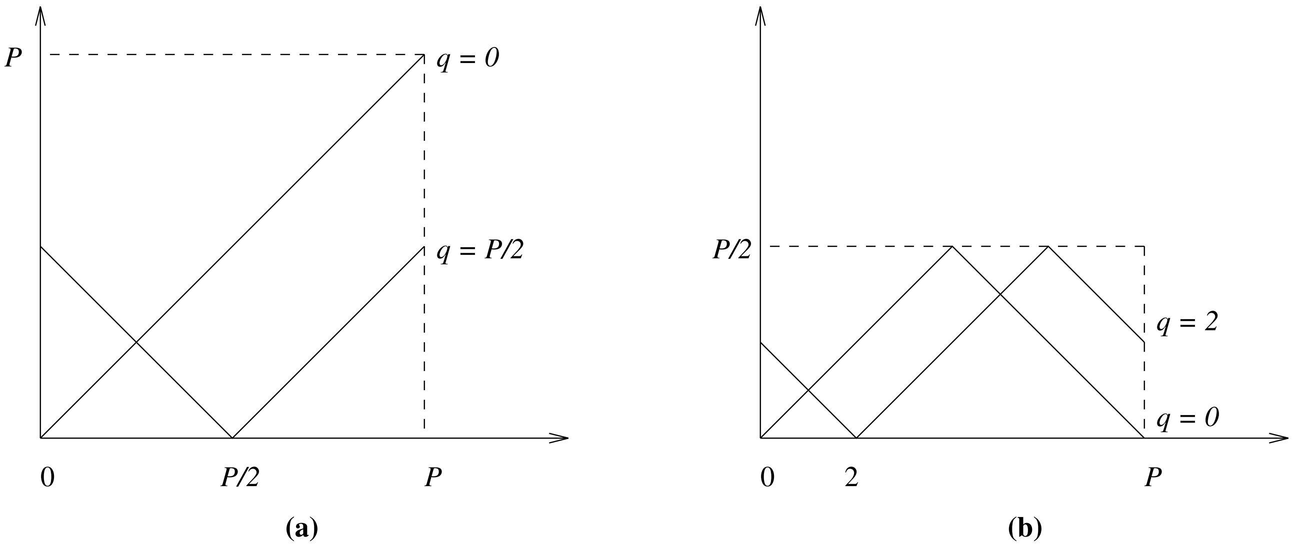


Figure 4: (a) Grid distance $d(p, 0)$ and $d(p, P/2)$. (b) Ring distance $d(p, 0)$ and $d(p, 2)$.

Proof: We show that $C_x(p \pm 1) \leq C_x(p) + 2$. Let the children of node x be y and z. Embed T_x optimally subject to $\pi(x) = p$, with cost $C_x(p)$. Move x one place to the left or right without moving any other node. This is a (possibly suboptimal) placement of T_x subject to $\pi(x) = p \pm 1$, with cost no less than $C_x(p \pm 1)$. Its cost clearly exceeds $C_x(p)$ by at most 2. $\square$

Finally, we show that $r^*_{xy}(p)$, the best position for node y given that its parent node x is placed at position p, can always be chosen to be either p or a breakpoint of y's subtree cost function C_y.

Lemma 4 *Let d be the distance function of a one-dimensional grid or ring. Let p be a position, and y an internal node of an expression tree. Then there exists an $r^*_{xy}(p) \in \mathcal{B}(C_y) \cup \{p\}$.*

Proof: Lemma 2 implies that $C_y(r) + d(p, r)$ is PLC as a function of r, with breakpoints in $\mathcal{B}(C_y) \cup \mathcal{B}(d(p, \cdot))$. Thus a minimum of $C_y(r) + d(p, r)$ occurs at the endpoint of a subinterval, that is, at a breakpoint. In the case of a ring, the breakpoint at $\Phi(p)$ is a maximum of $d(p, \cdot)$ and can be a minimum of the sum only if it is a minimum of C_y. Thus a minimum of the sum occurs either somewhere in $\mathcal{B}(C_y)$ or at p. $\square$

3 Grids and hypercubes

For a k-dimensional grid, positions are k-tuples of integers, and the distance function d is defined as

$$d((p_1, \ldots, p_k), (q_1, \ldots, q_k)) = \sum_{i=1}^{k} |p_i - q_i|. \tag{5}$$

Since the distance is the direct sum of the distances along the k directions, Lemma 1 applies, and we can treat each dimension independently. A hypercube is a special case of a grid (with $k = \log_2 P$), so everything in this section applies to hypercubes as well.

We will simplify Mace's algorithm for a one-dimensional grid by showing that every C_x is a PLC function of position p, with at most five pieces. Thus, for each x, we can represent C_x by only a constant

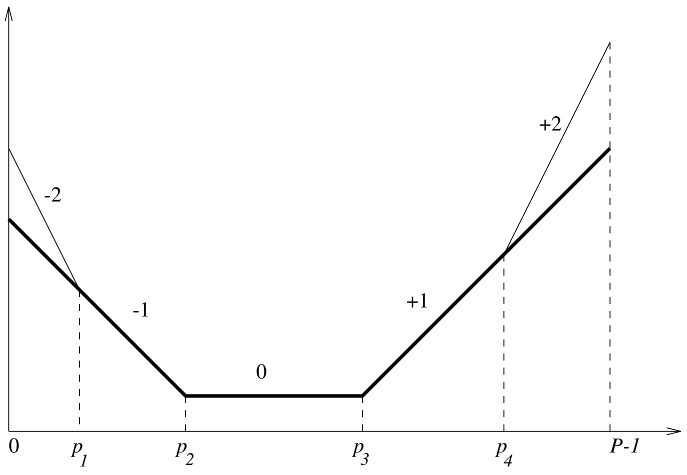


Figure 5: The contribution function for the grid metric. The light lines show the subtree cost function C_y, and the heavy lines show the contribution function g_{xy}.

amount of information, instead of by tabulating all P of its values. We will actually show that all we need to compute are the endpoints of the interval on which C_x is minimum. Our algorithm requires $O(N)$ time and space, which is the same complexity as Gilbert and Schreiber's algorithm [9] that used the robustness of the grid metric; indeed, the compact dynamic programming algorithm turns out to be the same as their algorithm for this case.

3.1 Compact dynamic programming for grids

Throughout this section we fix a one-dimensional grid $\mathbb{P}$ and a binary expression tree T, and define the functions C_x, g_{xy}, and r^*_{xy} as in Section 1.3.

We first show by induction that the subtree cost functions C_x have a simple form: they are PLC with slopes $(-2, -1, 0, 1, 2)$. The induction step uses the following lemma to get from the C's to the g's.

Lemma 5 *Let C_y be PLC with slopes $(-2, -1, 0, 1, 2)$ and breakpoints $(0, p_1, p_2, p_3, p_4, P-1)$. Then g_{xy} is PLC with slopes $(-1, 0, 1)$ and breakpoints $(0, p_2, p_3, P-1)$. Furthermore, $r^*_{xy}(p)$ may always be chosen in $[p_2, p_3]$.*

Proof: A tedious case analysis (helped by Lemma 4) yields

$$g_{xy}(p) = \begin{cases} C_y(p_2) + p_2 - p, & \text{if } 0 \le p \le p_2 \\ C_y(p_2), & \text{if } p_2 \le p \le p_3 \\ C_y(p_2) + p - p_3, & \text{if } p_3 \le p \le P-1. \end{cases}$$

Figure 5 is an illustration. This proves the result for g_{xy}. The result for $r^*_{xy}(p)$ follows from a similar case analysis:

$$r^*_{xy}(p) = \begin{cases} p_2, & \text{if } 0 \le p \le p_2 \\ p, & \text{if } p_2 \le p \le p_3 \\ p_3, & \text{if } p_3 \le p \le P-1. \end{cases}$$

□

The other half of the induction is the step from the g's to the C's:

Lemma 6 *Let g_{xy} and g_{xz} be PLC functions, both with slopes $(-1, 0, 1)$, and with breakpoints $\mathcal{B}(g_{xy})$ and $\mathcal{B}(g_{xz})$. Then $C_x = g_{xy} + g_{xz}$ is also PLC, with slopes $(-2, -1, 0, 1, 2)$ and breakpoints $\mathcal{B}(g_{xy}) \cup \mathcal{B}(g_{xz})$.*

Proof: This follows from points 3 and 4 of Lemma 2. □

This is all we need for the induction.

Theorem 2 *The subtree cost function C_x for every node x of the expression tree T has exactly five subintervals (some of which may be degenerate), with slopes $\mathcal{M}(C_x) = (-2, -1, 0, 1, 2)$.*

Proof: We use induction on the height of T_x.

The base case is a tree of height 0, whose single node is a leaf. By equation (3), the subtree cost function is $C_x(p) = d(\pi(x), p)$, which is PLC with three subintervals of slopes $(-1, 0, 1)$, the middle one being degenerate. We consider this to have two more degenerate subintervals, of slopes -2 and 2, at positions 0 and $P - 1$.

The induction step is then just Lemmas 5 and 6. □

The efficient algorithm for grids (Algorithm 2) follows the explicit dynamic programming outline, but instead of enumerating all the values of C_x at each node x it just computes the breakpoints. Indeed, we see from Lemma 5 that only the two middle breakpoints of C_y contribute to g_{xy} or r^*_{xy}. Thus the algorithm just keeps track of the two middle breakpoints, which are the endpoints of the central flat subinterval of C_y. This turns out to be the same computation as in Gilbert and Schreiber's algorithm; the flat subinterval is a "generalized intersection" in their terminology. The algorithm uses only a constant amount of time and space at each node.

Theorem 3 *For an expression tree T with N leaves, Algorithm 2 correctly solves the optimal placement problem for a k-dimensional grid of processors in $O(kN)$ time.* □

4 Rings

For a $P_1 \times \cdots \times P_k$-processor k-dimensional ring, the distance function d is defined as

$$d((p_1, \ldots, p_k), (q_1, \ldots, q_k)) = \sum_{i=1}^{k} \min(|p_i - q_i|, P_i - |p_i - q_i|). \tag{6}$$

This distance function is the direct sum of the distances in the k orthogonal directions. Lemma 1 says that we can treat each dimension independently. We will therefore take the position space $\mathbb{P}$ to be a one-dimensional ring of size P for the rest of this section.

The ring metric is not robust in the sense of Gilbert and Schreiber [9]; thus their methods do not apply. Our approach will be similar to that for the grid metric: we will show that the function C_x has a simple form that can be computed much more easily than by tabulating all P values. As in the grid case, C_x will turn out to be PLC for each node x. Now, however, the number of linear pieces is proportional to the number of leaves in subtree T_x, not constant as in the grid case. Thus the algorithm will take $O(Nh)$ time in all, where h is the height of the expression tree. Figure 6 is an example of an expression tree embedded optimally in a 64-processor ring, with a graph of the function C_{ROOT}.

Algorithm 2 (Optimal embedding in a grid of processors.)
Input: A position space $(\mathbb{P}, d)$ and an expression tree T with an embedding π for its leaves.
Output: A minimum-cost embedding π of T in $\mathbb{P}$.
Method: For each dimension of the grid, carry out the following to compute one dimension of the embedding.

1. Traverse T in postorder, computing two numbers L and R for each node. $L(x) = R(x) = \pi(x)$ at a leaf node x. At a nonleaf node x with children y and z, $L(x)$ is the second-smallest and $R(x)$ the second-largest of the four positions $L(y)$, $R(y)$, $L(z)$, and $R(z)$.

2. Set $\pi(\text{ROOT})$ to any integer in $[L(\text{ROOT}), R(\text{ROOT})]$.

3. Traverse T in preorder. For each internal node y with parent x, set $\pi(y)$ to the point $p \in [L(y), R(y)]$ that is closest to $\pi(x)$. Thus,

$$\pi(y) = \begin{cases} L(y), & \text{if } \pi(x) \le L(y) \\ \pi(x), & \text{if } L(y) \le \pi(x) \le R(y) \\ R(y), & \text{if } R(y) \le \pi(x). \end{cases}$$

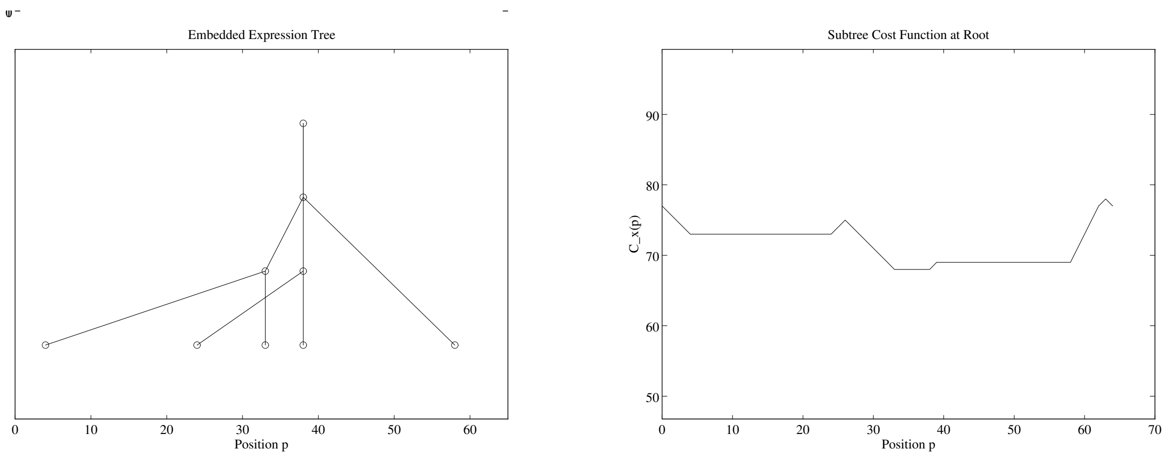


Figure 6: (a) An expression tree embedded optimally in a 64-processor ring. The vertical coordinate is height in the tree. (b) The cost function at the root of the tree.

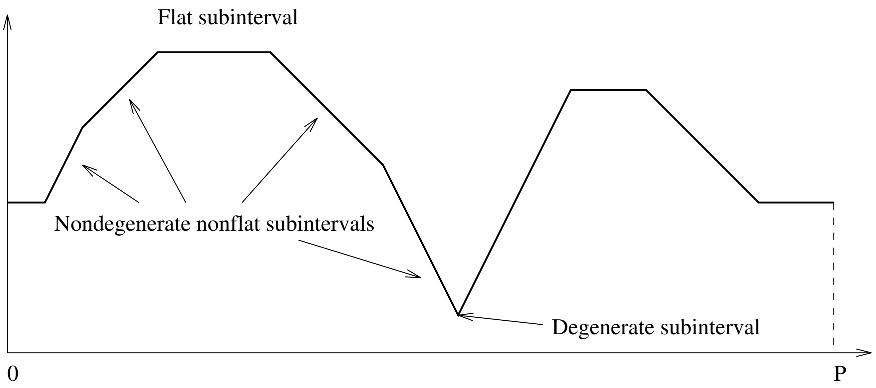


Figure 7: Various kinds of intervals in the cost function for the ring metric.

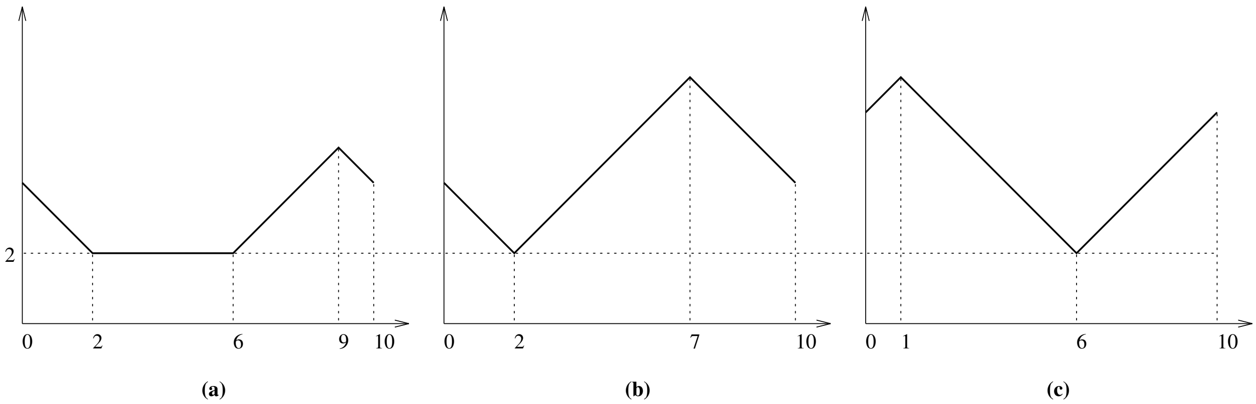


Figure 8: The possible forms of the function g^i_{xy} for the ring metric. In all cases, S_i is the interval $(2, 6)$, the number of processors is 10, and the smallest value of C_y on the interval is 2. (a) S_i has zero slope. (b) S_i has positive slope. (c) S_i has negative slope.

4.1 Compact dynamic programming for rings: Theory

We extend the subtree cost function C_x and the contribution function g_{xy} to the half-open real interval $[0, P)$. As described in Section 2.3, we consider functions on the real numbers modulo P, so position P is identified with 0. The optimal placements that our algorithm derives will be integers.

Our goal is to prove that the subtree cost function C_x is PLC, with a number of subintervals linear in the size of the subtree. We will show that C_x is PLC with slopes 0, ± 1, and ± 2, and with at most $2N$ subintervals if the subtree has N leaves. The proof is by induction on the height of the tree. The induction hypothesis also describes the contribution function g_{xy} for a child y of x. We will show that g_{xy} is PLC with slopes 0 and ± 1.

For the induction step, we consider a child y of x and prove a series of lemmas about g_{xy} and C_x, starting from the hypothesis that C_y is PLC with integral slopes. The details of the induction are messy but not difficult; the squeamish reader may wish to skip directly to Section 4.2.

If p, q, and r are positions, an inequality like $p \leq q$ is meaningless for rings, since one may proceed from p to q in either direction around the ring. However, we define a double inequality like $p \leq q \leq r$ to mean that $(q - p) \bmod P \leq (r - p) \bmod P$; in other words, as one proceeds around the ring in the increasing direction from p, one encounters q before r. For example, if $P = 8$, then $2 \leq 4 \leq 7$ and $4 \leq 7 \leq 2$, but not $4 \leq 2 \leq 7$. Double inequalities on ring positions using $\geq$, $<$, and $>$ are defined similarly. Recall also that the *antipode* of a position p on the ring is $\Phi(p) = (p + P/2) \bmod P$.

Let y be a child of x in the expression tree. Suppose, for the purposes of induction, that C_y is PLC with integral slopes. We represent C_y by a sequence $\mathcal{B}(C_y)$ of breakpoints, with associated subintervals $\mathcal{S}(C_y)$ having slopes $\mathcal{M}(C_y)$. A *flat subinterval* of C_y is one on which C_y is constant. For purposes of this proof only, we will posit that no two adjacent subintervals of C_y have slopes of opposite sign; if necessary, we introduce a *degenerate flat subinterval* of zero length between any two such subintervals. We take the PLC representation of C_y to have no zero-length subintervals other than these, and to have no two adjacent intervals of the same slope. Figure 7 is an example of a subtree cost function for the ring metric, showing the various kinds of subintervals.

Now consider the contribution function g_{xy}. It is defined (in equation (2)) as a minimum over positions r for node y. If we break that minimum up according to the subintervals of C_y, we get

$$g_{xy}(p) = \min_{S_i \in \mathcal{S}(C_y)} g^i_{xy}(p), \tag{7}$$

where

$$g^i_{xy}(p) = \min_{r \in S_i}(C_y(r) + d(p, r)) \tag{8}$$

is the “best” g_{xy} among those with y positioned in the i’th subinterval of C_y.

We can find a closed form for g^i_{xy} as follows. Let $S_i = [p_i, p_{i+1}]$ be the i’th subinterval of C_y. If S_i is flat, then $g^i_{xy}(p) = C_y(p_i) + \min_{r \in S_i} d(p, r)$. If p is within S_i, then the smallest $d(p, r)$ is 0 at $r = p$. If p is outside S_i, then the distance $d(p, r)$ is smallest when r is the endpoint of S_i closer to p. The closer endpoint changes at the antipode of the midpoint of the subinterval. Thus, for a flat subinterval S_i,

$$g^i_{xy}(p) = \begin{cases} C_y(p_i) + (p_i - p), & \text{if } \Phi((p_i + p_{i+1})/2) \leq p \leq p_i \\ C_y(p_i), & \text{if } p_i \leq p \leq p_{i+1} \\ C_y(p_i) + (p - p_{i+1}), & \text{if } p_{i+1} \leq p \leq \Phi((p_i + p_{i+1})/2). \end{cases} \tag{9}$$

Remember that arithmetic on positions, like $(p - p_{i+1})$, is done modulo P, yielding a number in $[0, P)$. Now suppose that C_y has positive slope on S_i. The slope is an integer, so it is at least 1. This implies that, for any position p, the minimum value of $C_y(r) + d(p, r)$ over all positions r in S_j occurs when r is the left endpoint p_i of S_i; moving r away from p_i increases $C_y(r)$ at least as fast as it decreases $d(p, r)$. Substituting $r = p_i$ into the definition of g^i_{xy} and using the definition of the distance d, we see that, in the case that C_y has positive slope on subinterval S_i,

$$g^i_{xy}(p) = \begin{cases} C_y(p_i) + (p_i - p), & \text{if } \Phi(p_i) \leq p \leq p_i \\ C_y(p_i) + (p - p_i), & \text{if } p_i \leq p \leq \Phi(p_i). \end{cases} \tag{10}$$

If C_y has negative slope, $g^i_{xy}(p)$ is as above but with p_{i+1} substituted for p_i.

Therefore, in every case (still under the inductive assumption that C_y is PLC with integral slopes), g^i_{xy} is PLC with either two or three subintervals, with slopes -1, 1, and possibly 0. Figure 8 shows the three possible forms of g^i_{xy}.

Now g_{xy} is the lower envelope of the g^i_{xy} functions, and therefore g_{xy} is also PLC with slopes -1, 1, and 0. We proceed to bound the number of subintervals of g_{xy}. First we show that the minimization in equation (7) can omit those g^j_{xy} that correspond to nonflat subintervals S_j of C_y, because they are dominated by g^i_{xy} corresponding to flat subintervals.

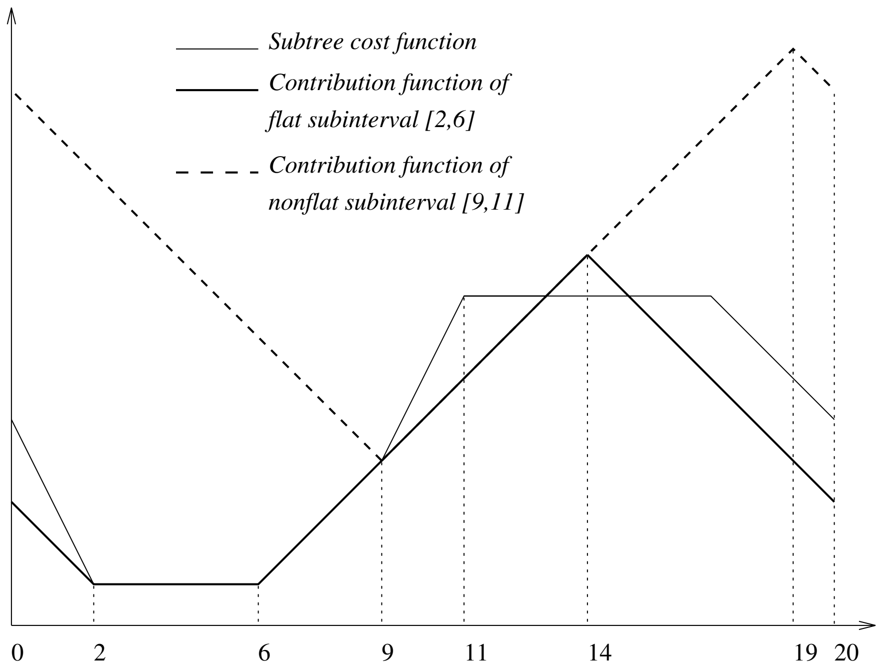


Figure 9: Nonflat intervals are dominated by flat intervals (Lemma 7). The ring has 20 processors. The subtree cost function C_y is shown in light solid lines. The contribution function g^j_{xy} of the nonflat interval $S_j = [9, 11]$ is shown by the dashed lines, and the contribution function g^i_{xy} of the flat interval $S^i = [2, 6]$ by heavy lines. The heavy lines are never higher than the dashed lines, illustrating the dominance property.

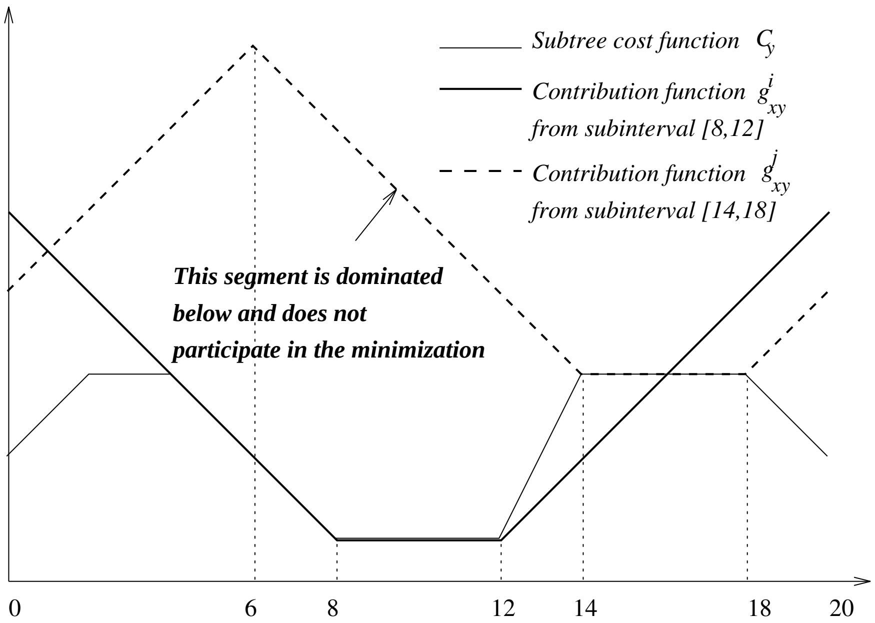


Figure 10: Dominance relations between the contribution functions from flat intervals (Lemma 8). The flat subintervals of interest are $S_i = [8, 12]$ and $S_j = [14, 18]$. The ring has 20 processors.

Lemma 7 *Let C_y be the subtree cost function for node y, and suppose C_y is PLC as above. Let S_j be a subinterval of C_y with nonzero slope. Then there is some flat subinterval S_i such that $g^i_{xy}(p) \leq g^j_{xy}(p)$ for all p.*

Proof: Figure 9 illustrates the result. Suppose C_y has positive slope on subinterval $S_j = [p_j, p_{j+1}]$. Let $S_i = [p_i, p_{i+1}]$ be the first flat subinterval of C_y to the left of S_j. (S_i may be degenerate.) Since C_y does not change sign without going through a flat interval, the slope of C_y is at least 1 between p_{i+1} and p_j. Therefore $C_y(p_j) \geq C_y(p_{i+1}) + (p_j - p_{i+1})$.

Plugging this into equation (9) for g^i_{xy} and equation (10) for g^j_{xy}, we get

$$\begin{aligned} g^i_{xy}(p_j) &= C_y(p_i) + (p_j - p_{i+1}) \\ &= C_y(p_{i+1}) + (p_j - p_{i+1}) \\ &\leq C_y(p_j) \\ &= g^j_{xy}(p_j). \end{aligned}$$

This is the inequality we wanted, but only at the point $p = p_j$. However, since $g^j_{xy}(p)$ increases with absolute slope 1 as p moves away from p_j in either direction, and the absolute slope of g^i_{xy} never exceeds 1, the inequality $g^i_{xy}(p) \leq g^j_{xy}(p)$ actually holds for all p.

The argument if C_y has negative slope on S_j is symmetric, using the first flat subinterval to the right of S_j. $\square$

The remaining terms in the minimization (equation (7)) are the g^i_{xy} that correspond to flat subintervals S_i of C_y. Equation (9) gives a closed form for those functions. The next lemma says that their minimum g_{xy} is itself PLC with at most two subintervals per term in the minimization.

Lemma 8 *Let C_y be the subtree cost function for node y, and suppose C_y is PLC with a representation as described above. Let the PLC representation for C_y have f nondegenerate flat subintervals and d degenerate flat intervals. Then g_{xy} is PLC with slopes 0 and ± 1, with at most $d + 2f$ nondegenerate subintervals.*

Proof: First a trivial case: if C_y is constant then $d = 0$, $f = 1$, and g_{xy} is also constant, so the lemma holds.

If C_y is not constant, then it has subintervals of both positive and negative slope, since it is periodic with period P. Two subintervals of C_y with slopes of opposite sign are separated by two (possibly degenerate) flat subintervals, one in each direction around the ring. Therefore we assume $d + f \geq 2$ below.

Now g_{xy} is the minimum g^i_{xy} over all i such that C_y is flat on S_i. We see from equation (9) that each such g^i_{xy} is PLC with slopes 0 and ± 1. Therefore g_{xy} is also PLC with slopes 0 and ± 1.

It remains to bound the number of subintervals of g_{xy}. Every linear segment of the graph of g_{xy} is a piece of a linear segment of some g^i_{xy}. We will first show that each segment of a g^i_{xy} contributes at most one segment to g_{xy}, concluding that g_{xy} has at most $2d + 3f$ nondegenerate subintervals. Then we will show that there are at least $d + f$ segments of g^i_{xy}'s that do not contribute to g_{xy} at all, so that g_{xy} has at most $d + 2f$ subintervals.

The zero-slope "basin" subinterval of g^i_{xy} is the same as the zero-slope subinterval S_i of C_y; it may be degenerate. Then g^i_{xy} has slope -1 to the left of S_i and has slope 1 to the right of S_i. No two S_i's overlap except possibly at a common endpoint. From this it is straightforward to see that the set $\{p : g^i_{xy}(p) \leq g^j_{xy}(p)\}$ is an interval on the ring $[0, P)$; in other words, the graphs of g^i_{xy} and g^j_{xy} cross over each other at most twice going around the ring. Thus, for each i, the set of positions p for which $g_{xy}(p) = g^i_{xy}(p)$ is the intersection of these sets over all j, and is itself a connected interval. Therefore any one linear segment of the graph of g^i_{xy} contributes at most one connected piece to g_{xy}. Each of the d functions g^i_{xy} for which S_i is flat and degenerate has two linear segments (the zero-slope middle segment of g^i_{xy} has zero length and can be discarded), and each of the f functions g^i_{xy} for which S_i is flat and nondegenerate has three linear segments. Thus g_{xy} has at most $2d + 3f$ subintervals.

Now consider any i and j that are consecutive in the sense that S_i and S_j are both flat subintervals of C_y (possibly degenerate), and there is no flat subinterval S_k of C_y with $S_i < S_k < S_j$ around the ring. Suppose that $g^i_{xy}(p_i) \leq g^j_{xy}(p_j)$; the opposite case is symmetric. Note that $g^i_{xy}(p_i) = g^i_{xy}(p_{i+1})$.

Since the nonflat slope of C_y has magnitude at least one, and because there is no flat subinterval of C_y between S_i and S_j, we must have $g^j_{xy}(p_j) \geq g^i_{xy}(p_{i+1}) + (p_j - p_{i+1})$. Therefore, since the absolute slope of g^i_{xy} is at most one, $g^j_{xy}(p_j) \geq g^i_{xy}(p_j)$. Now g^j_{xy} has a linear segment to the left of p_j with slope -1, so it grows (going to the left) at least as fast as g^i_{xy}. Therefore, throughout that segment (which includes all positions p with $\Phi((p_j + p_{j+1})/2) \leq p \leq p_j$) we have $g^j_{xy}(p) \geq g^i_{xy}(p)$. This means that g^j_{xy} does not contribute to the minimum in equation (7) in that interval, and thus the linear segment of the graph of g^j_{xy} with slope -1 does not contribute to the graph of g_{xy}. Figure 10 is an illustration.

In the symmetric case that $g^i_{xy}(p_i) \geq g^j_{xy}(p_j)$, we conclude that the linear segment of the graph of g^i_{xy} with slope $+1$ does not contribute to the graph of g_{xy}. Thus for every consecutive pair of flat subintervals of C_y there is one linear segment that does not contribute to g_{xy}. Therefore $d + f$ of the at most $2d + 3f$ possible segments in g_{xy} are eliminated, and g_{xy} has at most $d + 2f$ subintervals. □

The argument above takes the induction step from the subtree cost function C_y for the child node y to the contribution function g_{xy}. To complete the induction, we go from g_{xy} to C_x. This is the main theorem of this section.

Theorem 4 *For a binary expression tree T with N leaves, the subtree cost function C_x for any node x of T has a PLC representation with slopes 0, ± 1, and ± 2, and with at most $2N$ nondegenerate subintervals.*

Proof: Let N_x be the number of leaves in the subtree T_x rooted at the internal node x. We prove by induction on the height of T_x that C_x is PLC with slopes 0, ± 1, and ± 2, no two adjacent subintervals of C_x are of opposite sign, and C_x has at most $2N_x$ subintervals.

The base case is a tree of height 0, consisting of the single leaf x. For such a tree, $C_x(p) = d(p, \pi(x))$ is the distance function from the position of x, which is PLC with one interval of slope 1 and one interval of slope -1.

For the induction step, suppose that x has children y and z, with subtree T_y having N_y leaves, T_z having N_z leaves, and $N_x = N_y + N_z$. Consider the PLC representation of C_y with d_y degenerate flat subintervals, f_y nondegenerate flat intervals, and n_y nondegenerate nonflat intervals. Lemma 8 says that g_{xy} is PLC with slopes 0 and ± 1, with at most $d_y + 2f_y$ nondegenerate subintervals. A key observation is that the PLC representation of C_y must have at least as many nonflat as flat subintervals, since two flat subintervals cannot be adjacent. Therefore $n_y \geq d_y + f_y$, and $d_y + 2f_y \leq n_y + f_y$; thus g_{xy} has at most $n_y + f_y$ nondegenerate subintervals.

Define d_z, f_z, and n_z similarly for subtree T_z. Then g_{xz} has at most $n_z + f_z$ nondegenerate subintervals, and is also PLC with slopes 0 and ± 1. Therefore the sum $C_x = g_{xy} + g_{xz}$ is PLC with slopes 0, ± 1, and ± 2, and has at most $n_y + f_y + n_z + f_z$ nondegenerate subintervals.

By the induction hypothesis we have $f_y + n_y \leq 2N_y$ and $f_z + n_z \leq 2N_z$. Therefore C_x has at most $2N_y + 2N_z = 2N_x$ subintervals. This finishes the induction step and the proof. □

4.2 Compact dynamic programming for rings: Algorithm

We now present an $O(Nh)$ time algorithm (Algorithm 3) for computing an optimal placement of an expression tree on a ring. The algorithm hinges on a compressed representation of the subtree cost function $C_x(p)$. Each function in the algorithm is PLC and is represented as a set of breakpoints, values, and slopes, sorted around the ring.

If node y has N_y leaves and node z has N_z leaves, then their parent x has $N_x = N_y + N_z$ leaves. By Theorem 4, C_y and C_z have at most $2N_y$ and $2N_z$ nondegenerate subintervals. Furthermore, by Lemma 8, g_{xy} and g_{xz} also have at most $2N_y$ and $2N_z$ nondegenerate subintervals.

Lemma 9 *Suppose g_{xy} and g_{xz} are PLC and have at most $2N_y$ and $2N_z$ subintervals, respectively, given in order around the ring. Then Sum(g_{xy}, g_{xz}) runs in $O(N_y + N_z)$ time.*

Proof: Merging the lists of breakpoints takes time proportional to the sum of their lengths; only a constant amount of work is done at each subinterval. □

Lemma 10 *If C_y is PLC with at most $2N_y$ subintervals given in a sorted order along the ring, then Convolve(C_y) computes g_{xy} in $O(N_y)$ time.*

Proof: Equation (7) says that the output g_{xy} can be written as the minimum of one function g^i_{xy} for each subinterval S_i of C_y. Lemma 7 says we need only consider those g^i_{xy} for which C_y is flat on S_i. Thus Convolve computes the correct result.

Each g^i_{xy} has at most three subintervals, by equation (9). Thus, computing all the g^i_{xy} takes $O(N_y)$ total time. The total time for all the merge steps is also $O(N_y)$: each step takes constant time, and each step either computes a new breakpoint for g_{xy} or consumes a breakpoint of one of the g^i_{xy}'s. Since the total number of breakpoints of g_{xy} and C_x is at most $8N_y$, the total number of steps is $O(N_y)$. □

Algorithm 3 (Optimal embedding in a ring of processors.)
Input: A position space $(\mathbb{P}, d)$ and an expression tree T with positions π for its leaves.
Output: A minimal-cost embedding π of T in $\mathbb{P}$.
Method:

1. Traverse T in postorder. If x is a leaf, do nothing. If x is an internal node with children y and z:
   (a) Call Convolve(C_y) to compute g_{xy};
   (b) Call Convolve(C_z) to compute g_{xz};
   (c) Call Sum(g_{xy}, g_{xz}) to compute $C_x = g_{xy} + g_{xz}$.
2. Let $\pi(\text{ROOT})$ be some integer r at which $C_{\text{ROOT}}(r)$ is minimized.
3. Traverse T in preorder. For each internal node y with parent x, set $\pi(y) = r^*_{xy}(\pi(x))$.

Subroutine: (Convolve)
Input: A PLC function C_y and an integer number P of processors.
Output: A PLC representation of $g_{xy}(p) = \min_{0 \le r < P}(C_x(r) + d(r, p))$.
Method:

1. Let $S_1, S_2, \ldots, S_k, S_{k+1} = S_1$ be the sequence of intervals (both degenerate and nondegenerate) of C_y; let $x = \text{PARENT}(y)$.
2. Initialize $g_{xy} = +\infty$ as a constant function on $[0, P)$.
3. For $i = 1$ to $k + 1$
   (a) If S_i is nonflat, then do nothing.
   (b) If S_i is flat, first compute g^i_{xy} from equation (9).
   Then compute $g_{xy} = \min(g_{xy}, g^i_{xy})$ by simultaneously traversing the lists of breakpoints for g^i_{xy} and g_{xy}. A breakpoint for the new g_{xy} may come either at a breakpoint of g^i_{xy} or g_{xy}, or at an intersection of the graphs of g^i_{xy} and g_{xy}. Between breakpoints, the new g_{xy} takes its slope and value from the smaller of g^i_{xy} and the old g_{xy}.

Subroutine: (Sum)
Input: Two PLC functions g_{xy} and g_{xz} and an integer number P of processors.
Output: A PLC representation of $C_x = g_{xy} + g_{xz}$.
Method: Simultaneously traverse the lists of breakpoints for g_{xy} and g_{xz}. A breakpoint for C_x may come only at a breakpoint for g_{xy} or g_{xz}. Between breakpoints, the slope and value of C_x are the sum of those for g_{xy} and g_{xz}.

Theorem 5 *For an expression tree T with N leaves and height h, Algorithm 3 correctly solves the optimal placement problem for a k-dimensional ring of processors in $O(kNh)$ time.*

Proof: For a node x with children y and z, the previous two lemmas show that it takes $O(N_y)$ time to compute g_{xy}, $O(N_z)$ time to compute g_{xz}, and $O(N_x)$ time to compute C_x from g_{xy} and g_{xz}. Thus the running time for the complete algorithm is on the order of

$$\sum_{x \in \text{NODES}} |\text{LEAVES}(T_x)|. \tag{11}$$

This sum counts every leaf in T once for each of its ancestors. Thus, each of the N leaves is counted at most h times, giving $O(Nh)$ in each of the k dimensions, for a total running time of $O(kNh)$. □

The ring algorithm runs in $O(N^2)$ time for any expression tree; if the height of T is $O(\log N)$ (for example, if T is a complete binary tree), then the running time is $O(N \log N)$.

5 The discrete metric

The discrete metric d is defined as follows:

$$d(p,q) = \begin{cases} 0, & \text{if } p = q \\ 1, & \text{if } p \neq q. \end{cases} \tag{12}$$

This is a reasonable model for the communication cost of converting a data structure from one representation to another, for example, transposing arrays or operating on array sections with different strides. The discrete metric is also a good approximation to communication cost for message-passing architectures with large message startup times, as well as a good approximation for architectures with wormhole routing [5], where a single message is pipelined so that the distance it has to travel does not contribute significantly to its cost.

Gilbert and Schreiber [9] showed that the discrete metric is robust; therefore, for each node x, we need only record $\text{OPT}(x)$, the set of positions p for x that minimize $C_x(p)$, instead of C_x itself. Rephrasing their Theorem 2 in our terminology yields:

- $C_x(p) \geq \text{mincost}(T_x) + d(p, \text{OPT}(x))$.
- For every $p \in \text{OPT}(x)$, we can choose $r^*_{xy}(p) \in \text{OPT}(y)$ and $r^*_{xz}(p) \in \text{OPT}(z)$.

They show that the OPT set of a parent is either the union or the intersection of the OPT sets of its children. We reproduce their algorithm here in our notation, and show that it runs in $O(Nh)$ time.

Theorem 6 *Algorithm 4 computes an optimal placement of an expression tree with N leaves and height h for the discrete metric in $O(Nh)$ time.*

Proof: We omit the proof of correctness; see Gilbert and Schreiber [9]. To get the time bound, we represent sets of positions as sorted lists. Computing a union or intersection in step 2 takes time proportional to the sum of the lengths of the lists, as does the membership test in step 3. Since the set $\text{OPT}(x)$ contains only leaves of the subtree T_x, the total time is on the order of

$$\sum_{x \in \text{NODES}} |\text{LEAVES}(T_x)|. \tag{13}$$

As in Theorem 5, this sum is $O(Nh)$. □

Like the ring algorithm in Section 4.2, this algorithm runs in $O(N^2)$ time for any expression tree, and $O(N \log N)$ if the tree is balanced.

Algorithm 4 (Optimal embedding in the discrete metric.)
Input: A position space $(\mathbb{P}, d)$ and an expression tree T with positions π for its leaves.
Output: A minimum-cost embedding π of T in $\mathbb{P}$.
Method:

1. Traverse T in postorder. For each internal node x with children y and z, compute the set $\text{OPT}(x)$ of possible placements of x that will result in a minimum-cost embedding for T_x. Thus
$$\text{OPT}(x) = \begin{cases} \text{OPT}(y) \cup \text{OPT}(z), & \text{if } \text{OPT}(y) \cap \text{OPT}(z) = \varnothing \\ \text{OPT}(y) \cap \text{OPT}(z), & \text{otherwise.} \end{cases}$$

2. Let $\pi(\text{ROOT}) =$ some p in $\text{OPT}(\text{ROOT})$.

3. Traverse T in preorder. For each internal node x, set $\pi(x)$ as follows:
$$\pi(x) = \begin{cases} \pi(\text{PARENT}(x)), & \text{if } \pi(\text{PARENT}(x)) \in \text{OPT}(x) \\ \text{some } p \in \text{OPT}(x), & \text{otherwise.} \end{cases}$$

6 Fat-trees

The fat-tree machine architecture is an attempt to provide communication that behaves according to the discrete metric, with all communication actions having the same cost [16]. The processors are located at the leaves of a binary tree of height h, so that $P = 2^h$. (This connection tree is not to be confused with the expression tree T.) The bandwidth of the links doubles at every level as we move upwards in the tree, which prevents link congestion at any level. This argues that the discrete-metric algorithm from Section 5 is the best one to use for fat-trees, which is what we advocate in practice.

However, the discrete metric is not a completely accurate model for two reasons. First, the point-to-point communication time depends on how many levels of the tree the route traverses. This is important only when communicating small data aggregates, since moving a large enough distributed array always involves at least one route through the root. Second, real implementations of fat-trees (such as the network architecture of the CM-5[17]) do not increase link bandwidths at the higher levels of the tree fast enough to meet the theoretical bounds, thereby making congestion an issue.

A complete analysis of this nearly-fat-tree metric is an open question. We have analyzed the two extremes:

- The "large-aggregate" case, where data objects are distributed uniformly over all the processors, and communication cost depends on link congestion rather than path length. In this case, which we do not discuss in detail here, the cost of moving an object by distance d in the machine is $O(d^{1-\log_2 f})$, where f is the rate at which "fatness" grows up the tree. For $f = 2$, a true fat-tree, this reduces to the discrete metric.

- The "small-aggregate" case, where all communication is point-to-point, and cost is proportional to the number of tree levels traversed by the route. We discuss this case in the remainder of this section.

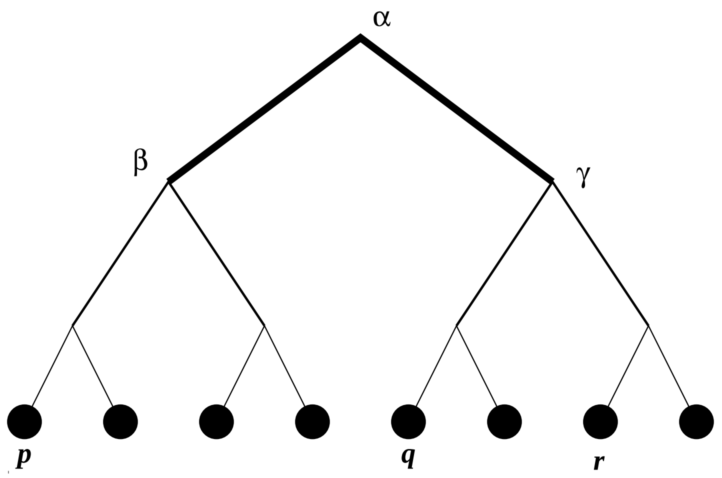


Figure 11: An 8-node fat-tree. The bandwidth of the links doubles at every level of the tree.

We emphasize that a realistic complete model would need to combine elements of both extremes; the discrete metric is probably the best simple model for this architecture.

6.1 The small-aggregate case

For any two processors p and q (that is, leaves of the fat-tree), we let $\text{LCA}(p, q)$ denote the least common ancestor of p and q in the tree. The distance between positions—that is, processors or leaves—p and q is defined to be the distance in the fat-tree, so

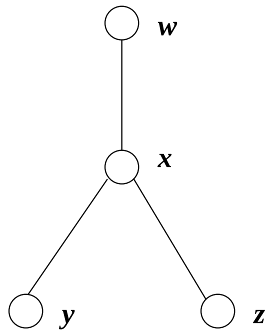
$$d(p, q) = 2 \cdot \text{HEIGHT}(\text{LCA}(p, q)), \tag{14}$$

where $\text{HEIGHT}(t)$ is the height of the subtree of the connection tree rooted at t. We call this the *fat-tree metric*. (If p and q are integers from 0 to $P - 1$, this distance is equal to twice the index of the highest bit in which their binary representations differ.)

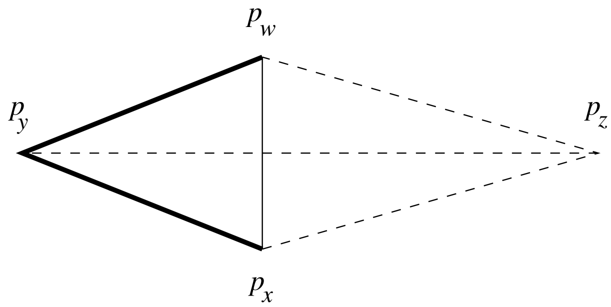
The triangle inequality relates the distances among any three points in an arbitrary metric. A curious fact about the fat-tree metric is that every triangle is isosceles, in the sense that its two longest sides are equal.

Lemma 11 (Isosceles Triangle Inequality) *Let p, q, and r be any three distinct positions in the fat-tree metric, that is, any three distinct processors or leaves of the connection tree. Then exactly two of the distances $d(p, q)$, $d(q, r)$, and $d(p, r)$ are equal, and the third distance is smaller than the other two.*

Proof: Let $\alpha = \text{LCA}(p, q, r)$ be the least common ancestor in the binary connection tree of p, q, and r. Let the two children of α be β and γ. Since α is an ancestor of the leaves p, q, and r, each of p, q, and r is a descendant either of β or of γ. Since α is the least common ancestor, at least one of the leaves p, q, and r is a descendant of β and at least one is a descendant of γ. Thus one of β and γ is an ancestor of one of the leaves, while the other is an ancestor of the other two leaves. Without loss of generality, suppose that p is a descendant of β and that q and r are descendants of γ. Then α is the least common ancestor of p and q, and also of p and r. Thus $d(p, q) = d(p, r) = 2 \cdot \text{HEIGHT}(\alpha)$. The least common ancestor of q and r is either γ or a descendant of γ, which is a child of α and has height less than $\text{HEIGHT}(\alpha)$. Therefore $d(q, r) < d(p, q) = d(p, r)$. $\square$



Relevant portion of computation tree



Relations between distances

Figure 12: Relations between distances in the fat-tree metric used in the proof of Lemma 12.

The labeled nodes in Figure 11 are an example of Lemma 11, with $d(q, r) = 4$ and $d(p, q) = d(p, r) = 6$.

The key fact that gives us an $O(N^3)$ time algorithm for the fat-tree metric is that there is always an optimal embedding that places every internal node is at the same position as one of its children. This fact follows from the isosceles triangle inequality.

Lemma 12 *Let T be an expression tree with fixed leaf positions. There is a minimum-cost embedding of T in the fat-tree metric in which every internal node of T is placed at the same position as one of its children.*

Proof: Let π be a minimum-cost embedding of T. For brevity, we write $p_x = \pi(x)$ and $d_{xy} = d(\pi(x), \pi(y))$. Suppose there is a node x with children y and z, such that $p_x \neq p_y$ and $p_x \neq p_z$. We will show that we can move x either to p_y or to p_z without increasing the total cost of the embedding. Therefore we can traverse the tree in postorder, moving each node to the same position as one of its children, never increasing the cost of the embedding, and eventually achieving an embedding that satisfies the statement of the lemma.

If x is the root of T, we move x to position p_y. This replaces two edges of cost d_{xy} and d_{xz} by two edges of cost 0 and d_{yz}; the triangle inequality says that the total cost does not increase. (We could as well have moved x to p_z.)

If x is not the root, let w be the parent of x in T, as shown in Figure 12. Moving x either to p_y or to p_z does not increase the sum of the costs of edges (x, y) and (x, z), by the argument in the last paragraph. If one of these moves does not increase the cost of edge (w, x), we make it, and we are done.

The only remaining possibility is that both of those moves would increase the cost of edge (w, x), that is, that $d_{wx} < d_{wy}$ and also $d_{wx} < d_{wz}$. Suppose this is so. Suppose in addition that $d_{xy} \leq d_{xz}$, as in Figure 12 (the opposite case is symmetric). We move x to p_y. The three edges (w, x), (x, y) and (x, z) change from total cost $d_{wx} + d_{xy} + d_{xz}$ to total cost $d_{wy} + 0 + d_{yz}$.

Now we apply the isosceles triangle inequality twice. First, $d_{wx} < d_{wy}$ implies $d_{xy} = d_{wy}$. Second, $d_{xy} \leq d_{xz}$ implies $d_{xz} \geq d_{yz}$. Therefore the total cost of the three edges (w, x), (x, y) and (x, z) does not increase when we move x to p_y.

The opposite case $d_{xy} \geq d_{xz}$ is symmetric; we move x to p_z. □

Theorem 7 *There is a minimum-cost embedding of the expression tree T for the fat-tree metric in which each internal node x is placed at one of the leaf positions of T_x.*

Proof: This is an immediate consequence of Lemma 12. □

Algorithm 5 embeds an expression tree in a fat-tree of processors. The algorithm is the explicit dynamic programming algorithm of Section 1.3, but instead of tabulating the various cost functions for all positions in the position space it only tabulates them for the positions of the leaves of the current subtree.

Algorithm 5 (Optimal embedding in a fat-tree of processors.)
Input: A position space $(\mathbb{P}, d)$ and an expression tree T with positions π for its leaves.
Output: A minimal-cost embedding π of T in $\mathbb{P}$.
Method:

1. Traverse T in postorder. At each nonleaf node x, use equations (2) and (4) to explicitly tabulate $C_x(p)$ and $r^*_{xy}(p)$ for all positions p ranging over the positions of the leaves of T_x.
2. Let $\pi(\text{ROOT})$ be some leaf position that minimizes $C_{\text{ROOT}}(p)$.
3. Traverse T in preorder. For each internal node y with parent x, set $\pi(y) = r^*_{xy}(\pi(x))$.

Theorem 8 *For an expression tree T with N leaves, Algorithm 5 correctly solves the optimal placement problem on a fat-tree of processors in $O(N^3)$ time.*

Proof: Correctness follows from Theorem 7, which says that it is sufficient to consider only leaf positions of the current subtree.

For an internal node x whose subtree T_x has N_x leaves, the functions r^*_{xy} and C_x are tabulations of N_x values. Computing each value by Equations (2) and (4) takes $O(N_x)$ time. Thus computing the functions r^*_{xy} and C_x for one node x takes $O(N_x^2)$ time, and the running time for the complete algorithm is on the order of

$$\sum_{x \in \text{NODES}} |\text{LEAVES}(T_x)|^2. \tag{15}$$

This is a sum of N terms each of size at most N^2. □

If T is well-balanced (for example, if T is a complete binary tree) then the sum in equation (15) and the running time are $O(N^2)$.

7 Weighted trees

Up to now we have assumed that the cost of communication between positions p and q depended only on the distance $d(p, q)$ and not on the size of the data aggregate being communicated. This is valid in data-parallel expressions all of whose variables and intermediates have the same size.

We now consider expressions that include data-parallel operations whose outputs can have different shapes or sizes than their inputs. For example, a fundamental building block of numerical linear algebra is the *rank-one update*

$$A := A + ab^T, \tag{16}$$

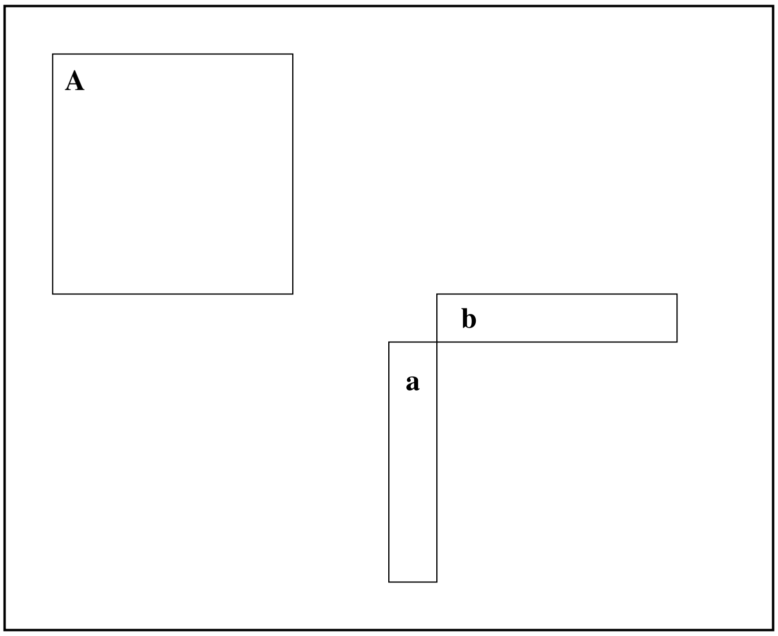


Figure 13: Alignment alternatives for a rank-1 update operation. One choice is to compute the product ab^T in place, align the resulting matrix with A, and perform the addition. The other choice is to move a and b before multiplying, so that the product matrix ab^T is aligned with A. The first choice communicates mn elements; the second choice only communicates $m + n$ elements.

where A is an $m \times n$ matrix, a is an $m \times 1$ column vector, and b is an $n \times 1$ column vector. Figure 13 shows an example in a metric which is the direct sum of a two-dimensional grid and a discrete metric whose two positions represent "row-wise" and "column-wise." If communication cost depends only on distance, the cheapest solution is to form the product ab^T without moving a or b, and then to move that intermediate result to the position of A for the addition. However, a better solution is to move a and b^T to the position of A before forming ab^T; this performs two moves instead of one, but moves only $m + n$ elements instead of mn.

We model different-sized operands by labeling the edges of the expression tree with nonnegative real-valued *weights*. The weight of edge (x, y) is written w_{xy}, and represents the size of the object that is computed at node y. Now the total communication cost of evaluating an expression tree is

$$\sum_{(x,y)\in \text{EDGES(T)}} w_{xy} \cdot d(\pi(x), \pi(y)),$$

and the master recurrences (2), (3), and (4) become

$$\begin{aligned} C_x(p) &= g_{xy}(p) + g_{xz}(p) \\ g_{xy}(p) &= \min_r(C_y(r) + w_{xy} \cdot d(p, r)) \\ g_{xz}(p) &= \min_r(C_z(r) + w_{xz} \cdot d(p, r)) \end{aligned} \tag{17}$$

for an internal node x with children y and z,

$$C_x(p) = w_{\text{PARENT}(x),x} \cdot d(p, \pi(x)) \tag{18}$$

for a leaf node x, and

$$r^*_{xy}(p) = \text{some } r \text{ at which } C_y(r) + w_{xy} \cdot d(p, r) \text{ is minimized.} \tag{19}$$

Explicit dynamic programming (Algorithm 1) once again produces a minimum-cost embedding of a weighted tree in an arbitrary metric in $O(NP^2)$ time, just as in the unweighted case. In this section, we show how to use compact dynamic programming to remove the dependence on P for the grid, ring, discrete, and fat-tree metrics.

7.1 Grids and hypercubes

Lemma 1 on decomposing direct sums of metrics still holds in the weighted case, so we need only consider one-dimensional grids. As in the unweighted case, the key idea is to compute PLC representations of the functions C_x and g_{xy}. However, the number of subintervals will turn out to be linear (it was constant in the unweighted case), and hence the running time of the weighted algorithm will be $O(N \log^2 N)$ (instead of $O(N)$ in the unweighted case).

Here are the details of the analysis. A PLC function is *convex* if the sequence $\mathcal{M} = (m_1, \ldots, m_k)$ of slopes of its subintervals is strictly increasing.

Theorem 9 *Let T be a weighted expression tree, and let d be the one-dimensional grid metric. Then for every node x, the function C_x is PLC and convex with at most $2N_x$ subintervals, where the subtree T_x rooted at x has N_x leaves.*

Proof: We use induction on the height of the tree. The base case is a leaf y; here C_y is PLC with two subintervals of slopes $\mathcal{M} = (-w_{xy}, w_{xy})$, where $x = \text{PARENT}(y)$.

The induction hypothesis is that C_y is convex and PLC with at most $2N_y$ subintervals, and x is the parent of y. We first show that g_{xy} is convex and PLC with at most $2N_y$ subintervals. Let the subintervals of C_y be $(S_1, \ldots, S_k)$, with slopes $(m_1, \ldots, m_k)$, and let $S_i = [p_i, p_{i+1}]$. Since C_y is convex, the sequence of slopes is monotone. We partition this sequence into three sections around the two values $-w_{xy}$ and w_{xy}:

$$m_1 < \cdots < m_{s-1} < -w_{xy} \le m_s < \cdots < m_{t-1} \le w_{xy} < m_t < \cdots < m_k. \tag{20}$$

(Any of the three sections may be empty, in which case $s = 1$, $s = t$, or $t = k + 1$.)

Now it is straightforward to compute $g_{xy}(p)$, with one case corresponding to each of the three sections. If p is one of the subintervals in the middle section, so that $p_s \le p \le p_t$, then the best choice of r in the recurrence (17) for $g_{xy}(p)$ is $r = p$; moving r left or right incurs more cost from the $w_{xy} \cdot d(p, r)$ term than it can save from the $C_y(r)$ term. If p is in the left section, $p < p_s$, then the best choice of r is $r = p_s$; moving r left costs more in C_y than it saves in $w_{xy} \cdot d(p, r)$, and moving r right costs more in $w_{xy} \cdot d(p, r)$ than it saves in C_y. Similarly, if $p > p_t$, the best choice is $r = p_t$. Therefore,

$$g_{xy}(p) = \begin{cases} C_y(p_s) + w_{xy} \cdot (p_s - p), & \text{if } 0 \le p \le p_s \\ C_y(p), & \text{if } p_s \le p \le p_t \\ C_y(p_t) + w_{xy} \cdot (p - p_t), & \text{if } p_t \le p \le P - 1. \end{cases}$$

This says that g_{xy} is identical to C_y, except that any leftmost subintervals of slope less than $-w_{xy}$ are replaced by a single subinterval of slope $-w_{xy}$, and any rightmost subintervals of slope more than w_{xy} are replaced by a single subinterval of slope w_{xy}. Therefore g_{xy} has at most $2N_y$ subintervals, like C_y; also, g_{xy} inherits C_y's convexity. See Figure 14 for an illustration.

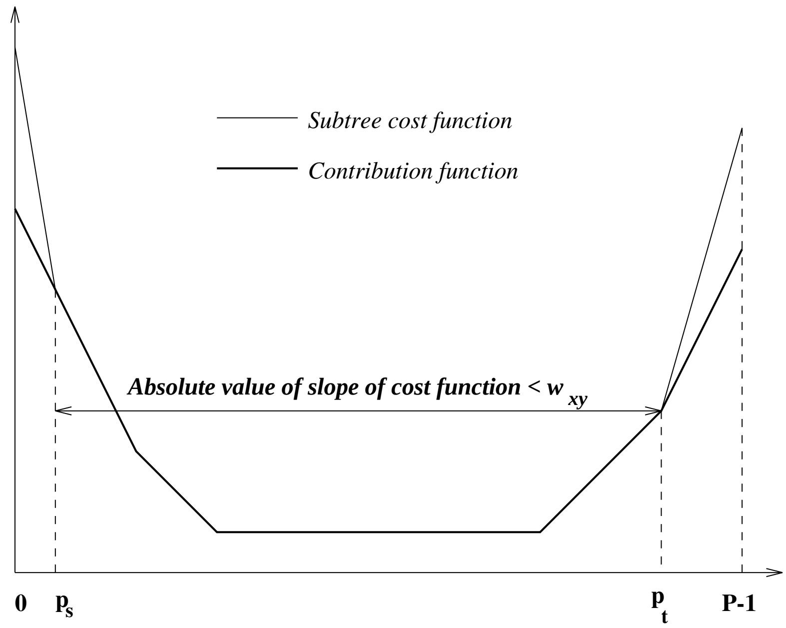


Figure 14: The subtree cost function C_y and the contribution function g_{xy} for a weighted tree in the grid metric.

The rest of the induction step is from g_{xy} to C_x. Let y and z be the children of x. We have $C_x = g_{xy} + g_{xz}$. Both g_{xy} and g_{xz} are convex by the induction hypothesis, so their sum is convex. Since g_{xy} is PLC with at most $2N_y$ subintervals and g_{xz} is PLC with at most $2N_z$ subintervals, their sum C_x is PLC with at most $2N_y + 2N_z = 2N_x$ subintervals. □

A straightforward implementation of the dynamic programming algorithm, using PLC representations of all the functions with $O(N)$ intervals, would use $O(N)$ time at each node and $O(N^2)$ time in all. We now show how to reduce this to $O(N \log^2 N)$. The idea is to compute $g_{xy} + g_{xz}$ not by merging the two lists of breakpoints, but by using binary search to insert breakpoints from the smaller list into the larger one. Algorithm 6 gives details.

It remains to analyze the running time of this algorithm. Each search tree operation on a PLC function takes time proportional to the logarithm of the number of subintervals of the function (in an amortized sense).

Lemma 13 *Convolve(C_y,w_{xy}) computes g_{xy} in $O(\log N_y)$ amortized time.* □

Lemma 14 *Sum(g_{xy},g_{xz}) computes C_x in $O(N_z \log N_y)$ amortized time, if $N_y \geq N_z$.* □

Lemma 15 *The solution to the recurrence*

$$f(m+n) = n \log m + f(m) + f(n), \quad m > n$$

is $f(n) = O(n \log^2 n)$.

Proof: We show that the suggested solution satisfies the recurrence. Let $m + n = p$. Then we need to show that

$$p \log^2 p \geq n \log m + m \log^2 m + n \log^2 n.$$

Algorithm 6 (Weighted optimal embedding in a grid of processors.)
Input: A position space $(\mathbb{P}, d)$ and a weighted expression tree T with an embedding π for its leaves.
Output: A minimum-cost embedding π of T in $\mathbb{P}$.
Method: This algorithm represents a PLC function as a search tree of subintervals; splay trees [20] are a suitable choice of search tree. The breakpoint positions are stored with the subintervals in the search tree. Each internal node of the search tree stores a "partial slope." The slope of a subinterval is represented as the sum of the partial slopes along the search path leading to the subinterval. This allows the algorithm to modify the slopes of an entire range of subintervals at once by changing one partial slope.

1. Form the search trees representing C_y for all $y \in \text{LEAVES}(T)$.
2. Traverse T in postorder. If x is a leaf, do nothing. If x is an internal node with children y and z:
   (a) Call Convolve(C_y, w_{xy}) to compute g_{xy};
   (b) Call Convolve(C_z, w_{xz}) to compute g_{xz};
   (c) Call Sum(g_{xy}, g_{xz}) to compute $C_x = g_{xy} + g_{xz}$.
3. Let $\pi(\text{ROOT})$ be some integer r at which $C_{\text{ROOT}}(r)$ is minimized.
4. Traverse T in preorder. For each internal node y with parent x, set $\pi(y) = r^*_{xy}(\pi(x))$.

Subroutine: (Convolve)
Input: A PLC function C_y and a slope w_{xy}.
Output: A PLC representation of $g_{xy}(p) = \min_r(C_x(r) + w_{xy} \cdot d(p, r))$.
Method:

1. Use binary search with $-w_{xy}$ and w_{xy} in the monotone list of slopes for C_y to find the two positions p_s and p_t defined in equation (20).
2. Convert the PLC representation of C_y into one for g_{xy} by replacing all subintervals to the left of p_s (if any) with a single subinterval of slope $-w_{xy}$, and replacing all subintervals to the right of p_t (if any) with a single subinterval of slope w_{xy}.

Subroutine: (Sum)
Input: Two PLC functions g_{xy} and g_{xz}.
Output: A PLC representation of $C_x = g_{xy} + g_{xz}$.
Method: Suppose g_{xy} has at least as many breakpoints as g_{xz}; if not, reverse their roles in the following. Convert the PLC representation of g_{xy} into one for C_x by traversing the breakpoints for g_{xz}, inserting each one into the list of breakpoints for g_{xy}, and updating the slopes.

Note that $n < p/2$. Therefore,

$$\begin{aligned} n\log m + m\log^2 m + n\log^2 n &\leq m\log m + m\log^2 m + m\log^2(p/2) \\ &\leq m\log m + m\log^2 p + m(\log p - 1)^2 \\ &= m\log m + m\log^2 p + m\log^2 p - 2m\log p + m \\ &\leq p\log^2 p. \end{aligned}$$

□

Theorem 10 *For a weighted expression tree T with N leaves, Algorithm 6 solves the optimal placement problem for the grid metric in $O(N\log^2 N)$ time.*

Proof: The running time at node x is dominated by the $O(N_z \log N_y)$ time to compute $C_x = g_{xy} + g_{xz}$. Thus the recurrence in Lemma 15 describes the total running time. □

7.2 The ring metric

Lemma 1 on decomposing direct sums of metrics still holds in the weighted case, so we need only consider one-dimensional rings.

In the unweighted case, we showed that the cost function at a node x could be represented as a PLC function with $O(N_x)$ subintervals, and thus derived an $O(Nh)$ optimal embedding algorithm. This is no longer true for the weighted case. We can still get a compact dynamic programming algorithm whose running time does not depend on P, however, by using the fact that every internal node can be embedded at the position of some leaf of T. The algorithm (Algorithm 7) is much simpler than the unweighted ring algorithm; it just traverses the nodes of T, tabulating the subtree cost functions for every leaf position at every node. This takes $O(N^3)$ time in the worst case, as compared to the $O(N^2)$ unweighted ring algorithm.

Theorem 11 *Let T be a weighted expression tree, and let d be the ring metric. Then any minimum-cost embedding of T places each node at the position of a leaf of T.*

Proof: The proof is a minor variation of the proof of Theorem 1 and is left to the reader. □

Theorem 12 *For a weighted expression tree T with N leaves, Algorithm 7 solves the optimal placement problem for the ring metric in $O(N^3)$ time.*

Proof: For correctness, Theorem 11 says that it suffices to consider the leaf positions as possible positions for the internal nodes. For the time bound, there are $O(N^2)$ values $C_x(p)$, each of which is computed by a minimization over $O(N)$ terms. □

7.3 The discrete metric

In the unweighted case, the robustness of the discrete metric implies the existence of an optimal embedding of a tree T that is also an optimal embedding of every subtree T_x. Therefore in Section 5 we could omit the computation of the subtree cost functions C_x, and only compute the minimum-cost sets $\text{OPT}(x)$ for each

Algorithm 7 (Weighted optimal embedding in a ring, the discrete metric, or a fat-tree.)
Input: A position space $(\mathbb{P}, d)$ and a weighted expression tree T with an embedding π for its leaves.
Output: A minimum-cost embedding π of T in $\mathbb{P}$.
Method:

1. Traverse T in postorder. At each nonleaf node x, use Equations (17) and (19) to explicitly tabulate $C_x(p)$ and $r^*_{xy}(p)$ for all positions p ranging over the positions of the leaves of T.
2. Let $\pi(\text{ROOT})$ be some leaf position that minimizes $C_{\text{ROOT}}(p)$.
3. Traverse T in preorder. For each internal node y with parent x, set $\pi(y) = r^*_{xy}(\pi(x))$.

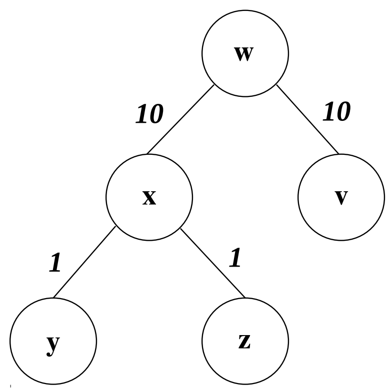


Figure 15: A weighted tree in which the internal node x cannot be embedded at the position of any leaf of its subtree.

node. All the elements of $\text{OPT}(x)$ were leaves of T_x, and the computation was a simple matter of unions and intersections of OPT sets.

None of this works in the weighted case. Indeed, there may be no optimal embedding of T that places node x at the position of any of the leaves of the subtree T_x. For example, in the tree of Figure 15, the only minimum-cost embedding is to place both internal nodes x and w at $\pi(v)$.

We can still get an efficient compact dynamic programming algorithm, however. First we show that every internal node can be embedded at the position of some leaf of T, so that we need only tabulate $C_x(p)$ for the N leaf positions. Then we observe that $C_x(p)$ takes on only one value for all p that are not positions of leaves of T_x, which simplifies the tabulation further. Finally, we show that each entry can be computed in constant time. The resulting algorithm runs in $O(Nh)$ time. This is the same asymptotic complexity as the unweighted algorithm, though the weighted algorithm is slightly more complicated. The details follow.

Theorem 13 *Let T be a weighted expression tree, and let d be the discrete metric. Then any minimum-cost embedding of T places every node at the position of a leaf of T.*

Proof: Suppose that embedding π does not place every node at a leaf position. Then there is an internal node x such that π does not place x at a leaf position, but π does place both children y and z of x at leaf positions. Let π' be the embedding

$$\pi'(v) = \begin{cases} \pi(y), & \text{if } \pi(v) = \pi(x) \\ \pi(v), & \text{otherwise.} \end{cases}$$

Notice that $\pi'(v)$ is the same as $\pi(v)$ if v is a leaf. Changing π to π' does not increase the cost of any edge of T, and it decreases the cost of edge (x, y) from w_{xy} to 0. Therefore π was not minimum-cost. □

Lemma 16 *Let T be a weighted expression tree, let d be the discrete metric, let π be a mapping of the leaves of T, and let C_x be as defined in equations (17) through (19). For each node x, the subtree cost function $C_x(p)$ takes only one value for all $p \notin \pi(\text{LEAVES}(T_x))$.*

Proof: Let p and q be positions, neither of which is in $\pi(\text{LEAVES}(T_x))$. Extend π to an embedding of the entire subtree T_x that embeds node x at position p and that has cost $C_x(p)$. Now modify this extended embedding by moving every node of T_x that is embedded at p, including node x, to be embedded at q. This modification does not move any leaves of T_x. It does not increase the cost of any edge of T_x. We conclude that $C_x(q) \leq C_x(p)$. Reversing the roles of p and q in the argument gives $C_x(p) \leq C_x(q)$. Therefore $C_x(q) = C_x(p)$. □

Lemma 17 *Let T be a weighted expression tree, let d be the discrete metric, let π be a mapping of the leaves of T, and let C_x be as defined in equations (17) through (19). For each node y with parent x, the contribution function $g_{xy}(p)$ is equal either to $C_y(p)$ or to $\min_r(C_y(r)) + w_{xy}$.*

Proof: Equation (17) says that $g_{xy}(p) = \min_r(C_y(r) + w_{xy} \cdot d(p, r))$. If the minimum occurs at $r = p$, this is equal to $C_y(p)$. Otherwise, since the metric is discrete, this is equal to $C_y(r) + w_{xy}$ for some r; and the same r minimizes $C_y(r)$ and $C_y(r) + w_{xy}$. □

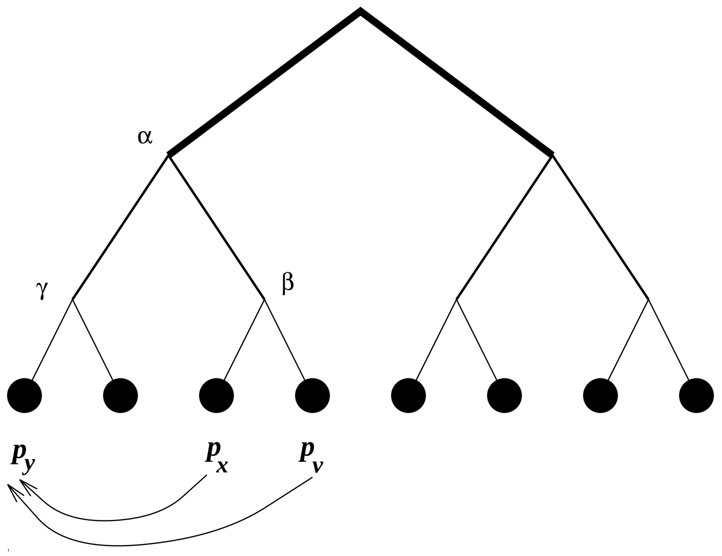


Figure 16: Moving internal tree nodes x and v to leaf position p_y.

The compact dynamic programming algorithm for the weighted discrete case tabulates the subtree cost function C_x for each node x, just like the weighted ring algorithm. However, the lemmas above imply that only $O(|\text{LEAVES}(T_x)|)$ information needs to be tabulated at each node x, and that each entry can be computed in constant time.

Theorem 14 *For a weighted expression tree T with N leaves and height h, in the discrete metric, Algorithm 7 solves the optimal placement problem and can be implemented to run in $O(Nh)$ time.*

Proof: We tabulate the subtree cost function C_x for each node x. Lemma 16 says that we need only tabulate one value for each leaf position of T_x plus one value for all the other positions. Thus the total space for the table is on the order of $\sum_x(|\text{LEAVES}(T_x)| + 1)$, which is $O(Nh)$.

To fill in one entry $C_x(p)$ of the table, we evaluate $g_{xy}(p)$ and $g_{xz}(p)$ by equation (17). Lemma 17 says that evaluating $g_{xy}(p)$ only requires choosing the smaller of $C_y(p)$ and $\min_r(C_y(r)) + w_{xy}$. We can compute $\min_r(C_y(r))$ at no extra cost while tabulating C_y; thus, it takes only a constant amount of work to compute the three values $g_{xy}(p)$, $g_{xz}(p)$, and $C_x(p)$. Therefore we compute the entire table in time linear in its size, that is, in $O(Nh)$ time. □

7.4 Fat-trees

In the unweighted case, we showed that some optimal embedding of T places every node at the position of one of the leaves of its subtree. The example in Figure 15, if embedded in a fat-tree with 4 processors, shows that this is not true in the weighted case. As with the weighted discrete metric, however, there is always an optimal embedding with every node placed at the position of *some* leaf of T.

Theorem 15 *Let T be a weighted expression tree, and let d be the fat-tree metric. Then some minimum-cost embedding of T places every node at the position of a leaf of T.*

Proof: Let π be an embedding of T. For brevity, we write $p_x = \pi(x)$ and $d_{xy} = d(\pi(x), \pi(y))$.

Suppose that embedding π does not place the internal node x at a leaf position. The position p_x is a leaf of the fat-tree (not to be confused with a leaf of the expression tree). Let α be the minimum-height fat-tree vertex that is an ancestor in the fat-tree both of p_x and of p_y for some leaf y of T. In other words, walk up the fat-tree from p_x, and stop at the first vertex α that is an ancestor of some leaf position p_y. Then in fact $\alpha = \text{LCA}(p_x, p_y)$. Figure 16 is an illustration.

Suppose the two children of α in the fat-tree are β and γ, where β is an ancestor of p_x and γ is an ancestor of p_y. Now let π' be such that

$$\pi'(v) = \begin{cases} p_y, & \text{if } p_v \text{ is a fat-tree descendant of } \beta \\ p_v, & \text{otherwise.} \end{cases}$$

Notice that $\pi'(v)$ is the same as $\pi(v)$ if v is a leaf of the expression tree, since by choice of α, no expression tree leaf position is a descendant of β. We claim that changing π to π' does not increase the cost of any edge of T. Consider an arbitrary edge (a, b) of T. If neither p_a nor p_b is a fat-tree descendant of β, then the cost of (a, b) does not change. If both p_a and p_b are fat-tree descendants of β, then $\pi'(a) = \pi'(b) = p_y$, and the cost of (a, b) becomes zero. Finally, suppose that p_a is a fat-tree descendant of β and p_b is not. Then the fat-tree path from p_a to p_b goes through α, and therefore $d_{ab} \geq 2 \cdot \text{HEIGHT}(\alpha) = d_{ay}$. The isosceles triangle inequality for p_a, p_b, and p_y then implies that $d_{ab} \geq d_{by}$. But d_{ab} is the cost of edge (a, b) under π and d_{by} is the cost of edge (a, b) under π', so the cost of (a, b) does not increase in changing π to π'.

Therefore changing π to π' preserves the leaf positions, does not increase the total cost, and decreases the number of internal nodes at non-leaf positions. We can repeat this transformation until every internal node is at a leaf position. $\square$

The compact dynamic programming algorithm for this case is identical to the weighted ring algorithm (except for the definition of $d(p, q)$). It tabulates the subtree cost functions for every leaf position at every node. Its $O(N^3)$ running time is the same as the unweighted fat-tree algorithm (Algorithm 5) in the worst case. However, this algorithm is never faster than $O(N^3)$, whereas the unweighted fat-tree algorithm runs in $O(N^2)$ time on well-balanced expression trees and in $O(N^2 h)$ time on trees of height h because it can always embed nodes at leaf positions chosen from their own subtrees.

Theorem 16 *For a weighted expression tree T with N leaves, Algorithm 7 solves the optimal placement problem for the fat-tree metric in $O(N^3)$ time.* $\square$

8 Virtualization and data partitioning

The discussion above has assumed that arrays must be aligned to avoid communication, and that the cost of residual communication is proportional to the difference in the positions involved. These assumptions are realistic when there are as many processors as there are array elements.

The number of array elements will ordinarily exceed the number of physical processors. In this case, multiple array elements are mapped to a single processor. That processor also computes multiple elements in all array computations, as if it were doing the work of many virtual processors. Hence the term *virtualization*.

We shall concern ourselves here with so-called *cyclic(b) virtualizations*. For an m-element, one-dimensional array A mapped to a one-dimensional P-processor ring or grid, element $A(i)$ resides on processor $(i \div b)$ mod P. When $b = 1$ the mapping is *cyclic*, and when $b = \lceil m/P \rceil$ it is a *block* mapping.

Multidimensional cyclic mappings are just Cartesian product mappings, and positions may continue to be determined in each dimension independently.

In this section we show howto use compact dynamic programming to obtain optimal alignments of arrays with cyclic(b) mappings to processor grids or rings. What is needed is a correct redefinition of the position space, reaffirmation that the grid and ring distance functions are appropriate as cost estimates, and the proper interpretation of the positions of the leaf nodes in that space.

Consider the cost of aligning the array sections $A(l:u:s)$ and $A(l+d:u+d:s)$. Let $m = u - l + 1$. We propose some realistic estimates of the cost of the communication as a function of the parameters d, m, s, b, and P. We assume for now that P, b, and m are fixed, and investigate how cost varies as a function of stride s and shift d.

The placement of elements is periodic with period bP. For example, if $d = bP$, then no interprocessor communication is required. In the case of an operation on two such "misaligned" arrays, like

```
A(1:100) = A(1:100) + A(1+b*P:100+b*P)
```

no communication is required. Furthermore, an optimizing compiler should not make a copy of either section in memory, but perform the arithmetic with the data "in place". Thus, the communication cost is zero.

Two subarrays with identical stride are perfectly aligned if their first indices differ by a multiple of bP. We can therefore take as the position space $\mathbb{P}$ the set of integers $[0, \ldots, bP-1]$. The position of an array section is given by its first index, reduced modulo bP. In the instance above, the leaf positions are l mod bP and $(l+d)$ mod bP.

Now consider the cost of communication due to an operation on nonaligned sections of cyclic(b) mapped arrays. For the unit of cost, we take the time to transfer one array element from a processor to one of its neighboring processors, assuming there is no other traffic on the channel. The transfer of several elements in parallel over separate channels costs just one unit. Communication cost may be dominated by one of several factors:

- **(Path length)** the hardware latency imposed on the message travelling the longest path;
- **(Congestion)** the number of bits traversing the most heavily loaded channel; or
- **(Startup)** the cost of message startup.

In our discussion of the grid and ring metrics, we are ignoring the startup effect. This is certainly reasonable for SIMD machines. For MIMD message passing machines, these metrics do not fully capture the real communication cost; rather, a sum of the discrete metric and one of the grid or ring metrics is better. However, the analysis of such a metric is an open problem.

It turns out that, for `SHIFT` in grids and rings, path length and congestion predict the same dependence of cost on d: cost is linear in $d' = (d \bmod bP)$. Indeed, the maximum distance travelled by a communicated datum is clearly $\lceil d'/b \rceil$ in a grid. Using congestion to model cost has the advantage of accounting for the effects of both virtualization (the larger the data array, the more traffic per channel) and stride (the larger the stride, the smaller the set of values to communicate). Thus, we shall use a realistic estimate of the congestion on the most heavily loaded channel as our "distance function". Congestion turns out to be proportional to the grid and ring distance on the position space $[0, \ldots, bP-1]$.

When aligning array sections, the total *flux* is the sum over all array elements of the number of "hops" (channels) the element travels. The ratio of the flux to the number of channels is clearly a lower bound on the cost of the communication, achievable if the communication load is balanced across the channels. In a one-dimensional grid or ring, the number of channels is $O(P)$.

With unit stride, the load on channels is uniform, consisting of $d'\lceil m/(bP)\rceil$ array elements moving in each direction. So, for an array A mapped cyclic(b) on a P-processor grid, the communication cost for aligning the sections $A(p:p+m-1)$ and $A(q:q+m-1)$ is

$$\begin{aligned} d(p,q) &= 2(\lceil m/(bP)\rceil)d', \\ \text{where } d' &= |p-q| \bmod bP. \end{aligned}$$

The factor of two may be removed if the interprocessor connection supports bidirectional communication at full bandwidth.

For rings, the distance must be adjusted to take into account wrap-around:

$$\begin{aligned} d(p,q) &= (\lceil m/(bP)\rceil)\min(d',(bP-d')), \\ \text{where } d' &= |p-q| \bmod bP. \end{aligned}$$

Bidirectional channels are not advantageous for shifting in rings.

If the block size b is not fixed but is to be chosen to minimize alignment cost, then the problem becomes more complicated. As a practical matter, we think that b can be determined on the following grounds. Increasing b reduces the cost of shift (by reducing $\lceil m/(bP)\rceil$ in the distance functions above), except for certain fortuitous choices (that reduce d'). If the shift distances in the program are all small, however, then b has no effect on the second term in the distances above, since then $d \bmod bP = d$ regardless of b. So if shift is the dominant theme in a computation, large b (*i.e.*, block distribution) is good. On the other hand, there may be reasons of load balance (frequent operation on contiguous subarrays, which is the central paradigm of matrix computation, for example) for preferring small b.

Stride adds some unfortunate complications. The communication volume is inversely proportional to stride s. Usually, the maximum channel load is reduced in proportion to the total volume. But there can be uneven distribution of the communication (and the computation, for that matter): take $s = bP$ for example, so that all the active elements in $A(l : u : s)$ fall in the same processor. We give realistic cost estimates for the grid case, without proofs, here.

Consider the pattern of active elements $\{l + ks\}$. This pattern repeats every $\bar{s} = s/\gamma$ blocks, where $\gamma = \gcd(s, b)$. The flux from a single period is given by

$$\bar{d} = \left\lfloor \frac{d' + (l \bmod \gamma)}{\gamma} \right\rfloor.$$

The number of periods is $N/(b\bar{s})$; therefore the total flux is

$$F = \frac{N\bar{d}}{b\bar{s}}.$$

Now, to get a better estimate of the communication cost, we consider the load on the most heavily loaded link. The processors fall into $\delta = \gcd(\bar{s}, p)$ classes, with a different load on the output channels of the processors of each class. This load is given by

$$L = \left(\frac{N\bar{d}}{b\bar{s}} \cdot \frac{1}{P}\right) \cdot \frac{\lceil \bar{d}/\delta \rceil}{\bar{d}/\delta}.$$

The first term is the average flux per processor (number of periods $\times$ flux per period / number of processors). The correction factor $\lceil \bar{d}/\delta \rceil / \bar{d}/\delta$ accounts for the uneven distribution of the communication to the channels; it cannot exceed δ, independent of the value of $\bar{d}$.

In summary, the grid and ring metrics are still the appropriate models for residual communication in the case of arrays mapped in block-cyclic fashion.

9 Conclusions

We have investigated the problem of optimally evaluating Fortran 90 array expressions on massively parallel distributed-memory machines. Algorithms based on dynamic programming solve this problem efficiently for a wide variety of interconnection schemes, including multidimensional grids and rings, hypercubes, the discrete metric, and fat-trees. Expressions containing operands of different sizes are aligned optimally using algorithms that extend the dynamic programming technique to edge-weighted trees. The running times of these algorithms range from linear to cubic in the size of the expression, and are independent of the size of the position space. Consequently, heuristics are not necessary for compiling optimal code from expression trees for these interesting distance functions. In a forthcoming paper [4], we present a unified optimization framework incorporating the ideas of this paper that extends the approach heuristically to basic blocks that are represented as directed acyclic graphs.

It might be possible to improve the asymptotic running times of some of the slower algorithms in this paper; we feel that they are already sufficiently fast to be practical. Other open problems in this area include a complete analysis of the nearly-fat-tree metric and a metric modeling startup costs in communication, a comprehensive treatment of array replication and collapsing (aligning some array axes with memory), and an extension of the analysis to handle control flow.

References

[1] AMERICAN NATIONAL STANDARDS INSTITUTE. *Fortran 90: X3J3 internal document S8.118 Submitted as Text for ANSI X3.198-1991, and ISO/IEC JTC1/SC22/WG5 internal document N692 Submitted as Text for ISO/IEC 1539:1991*, May 1991.

[2] BLELLOCH, G. E. NESL: A nested data-parallel language. Tech. Rep. CMU-CS-92-103, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, Jan. 1992.

[3] CHAPMAN, B., MEHROTRA, P., AND ZIMA, H. Programming in Vienna Fortran. *Scientific Programming 1*, 1 (Fall 1992), 31–50.

[4] CHATTERJEE, S., GILBERT, J. R., SCHREIBER, R., AND TENG, S.-H. Automatic array alignment in data-parallel programs. In *Proceedings of the Twentieth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages* (Charleston, SC, Jan. 1993). To appear.

[5] DALLY, W. J., AND SEITZ, C. L. The torus routing chip. *Distributed Computing 1* (1986), 187–196.

[6] FARRIS, J. S. Methods for computing Wagner trees. *Systematic Zoology 19* (1970), 83–92.

[7] FITCH, W. M. Toward defining the course of evolution: Minimum change for a specific tree topology. *Systematic Zoology 20* (1971), 406–416.

[8] FOX, G. C., HIRANANDANI, S., KENNEDY, K., KOELBEL, C., KREMER, U., TSENG, C.-W., AND WU, M.-Y. Fortran D language specification. Tech. Rep. Rice COMP TR90-141, Department of Computer Science, Rice University, Houston, TX, Dec. 1990.

[9] GILBERT, J. R., AND SCHREIBER, R. Optimal expression evaluation for data parallel architectures. *Journal of Parallel and Distributed Computing 13*, 1 (Sept. 1991), 58–64.

[10] HARTIGAN, J. A. Minimum mutation fits to a given tree. *Biometrics 29* (1973), 53–65.

[11] HIGH PERFORMANCE FORTRAN FORUM. High Performance Fortran language specification version 0.4. Draft, Nov. 1992. Also available as technical report CRPC-TR 92225 from the Center for Research on Parallel Computation, Rice University.

[12] HIRANANDANI, S., KENNEDY, K., AND TSENG, C.-W. Compiler support for machine-independent parallel programming in Fortran D. Tech. Rep. Rice COMP TR90-149, Department of Computer Science, Rice University, Houston, TX, Feb. 1991.

[13] HWANG, F. K. A linear time algorithm for full Steiner trees. *Operations Research Letters 4* (1986), 235–237.

[14] KNOBE, K., LUKAS, J. D., AND DALLY, W. J. Dynamic alignment on distributed memory systems. In *Proceedings of the Third Workshop on Compilers for Parallel Computers* (Vienna, Austria, July 1992), Austrian Center for Parallel Computation, pp. 394–404.

[15] KNOBE, K., LUKAS, J. D., AND STEELE JR, G. L. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing 8*, 2 (Feb. 1990), 102–118.

[16] LEISERSON, C. E. Fat-Trees: Universal networks for hardware-efficient supercomputing. *IEEE Trans. Comput. C-34*, 10 (Oct. 1985), 892–901.

[17] LEISERSON, C. E., ABUHAMDEH, Z. S., DOUGLAS, D. C., FEYNMAN, C. R., GANMUKHI, M. N., HILL, J. V., HILLIS, W. D., KUSZMAUL, B. C., ST. PIERRE, M. A., WELLS, D. S., WONG, M. C., YANG, S.-W., AND ZAK, R. The network architecture of the Connection Machine CM-5. In *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures* (San Diego, CA, June 1992), pp. 272–285.

[18] MACE, M. E. *Memory Storage Patterns in Parallel Processing*. Kluwer international series in engineering and computer science. Kluwer Academic Press, Norwell, MA, 1987.

[19] ROSE, J. R., AND STEELE JR, G. L. C*: An extended C language for data parallel programming. In *Proceedings of the Second International Conference on Supercomputing, Vol. 2* (San Francisco, CA, May 1987), pp. 2–16.

[20] SLEATOR, D. D., AND TARJAN, R. E. A data structure for dynamic trees. *J. Comput. Syst. Sci. 26* (1983), 362–391.

[21] WHOLEY, S. *Automatic Data Mapping for Distributed-Memory Parallel Computers*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, May 1991. Available as Technical Report CMU-CS-91-121.

[22] WINTER, P. Steiner problem in networks: A survey. *Networks 17* (1987), 129–167.