**Programming Solutions to the Algorithm Contraction Problem**

**Authors:** Philip A. Nelson and Lawrence Snyder

**Performing Organization:** University of Washington, Department of Computer Science, Seattle, Washington 98195

**Contract or Grant Number:** N00014-85-K-0328

**Report Date:** April 1986

**Number of Pages:** 4

**Distribution Statement:** Distribution of this report is unlimited.

**Abstract:**

Algorithms for the parallel solution of problems are usually designed assuming an unlimited number of processors. Physical parallel machines have a fixed number of processors. The algorithm contraction problem arises when an algorithm requires more processors than are available on the physical machine. We present tools for comparing algorithm contractions based on bottleneck communication paths. We apply these tools to minimum, matrix product and sorting.

**Keywords:** Algorithm contraction problem: parallel programming
Abstract

Algorithms for the parallel solution of problems are usually designed assuming an unlimited number of processors. Physical parallel machines have a fixed number of processors. The algorithm contraction problem arises when an algorithm requires more processors than are available on the physical machine. We present tools for comparing algorithm contractions based on bottlenecks in communication paths. We apply these tools to minimum, matrix product and sorting.

Introduction

Algorithms for parallel computers are usually designed assuming an unlimited number of processors. For non-shared memory parallel algorithms, this assumption generally manifests itself by the algorithm utilizing one processor "per point", or some other input size-dependent processor allocation. The physical machine has only a fixed number of processors, of course, which will almost certainly be less than the number required by the algorithm. In order to make the logical processes of the algorithm conform to the physical processors of the machine, we must group processes together into a module to be executed on a single physical machine. This activity is called contraction[13]. The way this contraction is performed can have a significant affect on performance.

Consider two examples based on an n x n grid of processes, i.e. the processes communicate with their four nearest neighbors:

1. There is much process-to-process communication and approximately equal computation required of each process.
2. There is little process-to-process communication and the amount of computation per process is proportional to its |j| index, e.g. process i,j iterates |j| times.

Suppose we have only one fourth the required number of processors and now compare two ways of forming contractions of four processes per processor[4]: Coalescing groups of adjacent 2x2 subarrays; folding groups as if the grid is folded in half and then in half again, i.e. (i,j) (1/2i,1/2j) is associated with i,j, n-i-j, i, j, n-i,j, n-j-1. Clearly, algorithm (1) should be coalesced by coalescing because the process-to-process communication for the processes sharing the same processor will become interprocessor communications (i.e. fast memory references) rather than slow interprocessor communication; folding would not be as attractive because no communication is saved by locality. Alternatively, algorithm (2) should be contracted by folding because the work is balanced since each processor will perform a matching amount of long and short computations; coalescing would not be as attractive because the processors receiving processes with large indexes will become a bottleneck.

Using the results of Berman and her colleagues[3], an algorithm can be automatically contracted, and this seems to be the best approach when nothing is known about the algorithm. At the other end of the spectrum, however, the programmer has "complete" knowledge about the algorithm. How should be be guided when performing his own contraction? In this paper we develop some apparatus to guide the programmer who must contract an algorithm. We provide some case studies of contraction that show an unexpected diversity and we offer some general contraction strategies that can find application in other algorithms. Contraction is an enrichment problem for parallel programmers[13], and so a secondary goal here is to expose it as an important topic for study and a subject suitable for rigorous analysis.

Definitions

The generic parallel architecture under consideration in this paper is a non-shared memory model. It is a collection of homogeneous sequential computers operating asynchronously and connected in a communication network that is a bounded degree graph[13]. A single "edge" in the graph provides bidirectional communication between two processors. The CHIP[11] architecture is an example of this generic architecture.

The method used for programming this model consists of defining a sequential program for each processor and a communication graph. We are assuming a configurable architecture. (The problem of mapping a communications graph onto a different processor connection graph is discussed by Berman and Snyder[4] and Bokhari[5].) Communication is explicitly shown in the sequential code by specifying a data value to be sent to the processor connected by a given edge.

The algorithm contraction problem arises when an algorithm that is designed for use on n processors must be mapped to a physical parallel computer with only p < n processors. The programmer must decide which logical processes are to be mapped to the physical processors. Assuming that the logical processes have balanced loads (they run for the same length of time), we would like the physical processors to have balanced loads. This is done by mapping the same number of logical processes to each physical processor. The number of logical processes assigned to two arbitrary physical processors should differ by at most an additive constant c. For most contractions, it would be best to have c = 1.

The contraction induces a communication graph for the p physical processors. This new graph is defined by logical processes needing to communicate with other logical processes not mapped to the same physical processor. We assume that if a logical process in processor i needs to communicate with a logical process in processor j, there is a physical edge connecting the two processors in the new graph. The contraction may map many of these logical edges to one physical edge in the new graph. This is, we are allowing only one edge between physical processors. Under the assumption of a bounded degree graph for the generic architecture, this induced graph must also be of bounded degree.

As an example of contraction, let us assume we have an algorithm with a tree graph. Consider the contraction to 5 processors shown in Figure 1. This contraction caused an increase in the degree; for example, the new root vertex has four descendents. Using this kind of a contraction, it can be shown that given p processors, contracting an algorithm with at least p^3 logical processes requires degree p^2. Figure 2a shows a contraction of the tree to 4 processors. An extension of this method yields a binary tree in the p processors.

Figure 2b gives another contraction to 4 processors. This contraction is derived by the recursive tree contraction given by Leiserson[9]. Given two instances of a tree each with an associated free node, we can build a new tree and an associated free tree. This produces a linear area layout in the plane with several desirable properties, one of which is the constant number of external edges.

In this paper, we will be considering the contribution of the communication time to the performance of the contracted algorithm. Unless otherwise stated, we assume that a communication between processing elements costs a fixed time t_c. During this time, no other communication is possible.
the same direction may take place. We are specifically allowing all edges to have simultaneous communication. Communication internal to a processor costs the fixed time $t_i$. We also assume that $t_i > t_j$.

We would like to develop tools for reasoning about the relative merits of different contractions. This includes their communication costs and their execution times. To aid in this objective we give the following definitions.

Let $A = (V, E)$ be an algorithm where $V$ is a set of logical processors (vertices and associated programs) and $|V| = n$, $E$ is a set of edges $(V_i, V_j)$. The contraction $M$ maps elements of $V_n$ onto $V_m$ such that the number of elements of $V_n$ mapped to an arbitrary element of $V_m$ differs by no more than one from the number of elements of $V_n$ mapped to any other element of $V_m$.

Let $w(e)$, the weight of $e$, for $e = (V_i, V_j)$, be the larger of the number of messages from $V_i$ to $V_j$ and the number of messages from $V_j$ to $V_i$.

Let $K(A) = \max w(e)$, for $e \in E$, be the communication "cost" of $A$. This cost is an estimate of the minimum communication time required for the algorithm. Due to dependencies, the actual communication cost may be more.

Let $T(A)$ be the execution time for $A$.

PROPOSITION 1: For a given $A$, $p$, $M_1$, and $M_2$, and $t_i > t_j$ if $K(M(A_p)) \leq K(M(A_{p'}))$, then $T(M(A_p)) \leq T(M(A_{p'}))$.

This proposition is formalizing the notion that the bottleneck edge will be a lower bound on the time required for the execution of the mapped algorithm. If the processors have a small amount of computation relative to the communication, the execution time will depend on the communication time. The bottleneck edge of the contraction $M_1$ will require a minimum of $t_i K(M_1(A_p))$ time, which is less than $t_i K(M_2(A_p))$. If we choose processors that can map between $p$ processors for each physical processor in the same direction, it is possible to have a smaller cost than the one original algorithm. Therefore, $K(M_1(A_p)) \leq K(A)$, the cost of the original algorithm.

LEMMA 2: For complete binary tree algorithms with balanced processor loads, equal edge weights, and unidirectional communication, the cost of an algorithm contraction based on Leiserson's binary tree layout technique yields optimum results.

PROOF: For the mapping $M(A_p)$, each processor contains all the subtrees of an "extra" node. The extra nodes are used in the tree above the subtrees contained in the processors. Therefore, there are at most 4 external connections. Of these four, two edges are used to receive/send data from/to the children of the extra node, and two edges are used to send/receive data to/from the subtree's and the extra node's parents. Since the root of the subtree and the extra node are not at the same level in the tree, edges with data flowing in the same direction can not be connected to the same physical processor. Therefore, $K(M(A_p)) \leq K(A)$, which is the lower bound.

Notice that this layout technique will place two logical edges in the same physical edge for some physical edge. For tree algorithms with bidirectional communication, we then get $K(M(A_p)) = 2K(A)$.

To help verify these results, the minimum algorithm was programmed using the Pacer parallel programming environment[12]. Both $M_1$ and $M_2$ were programmed. Each contraction was run using 64 data items per processor with 4 and 16 processors. The results of these timings are given in Table 1. Each "tick" represents a microsecond on the 64 processor Pringle.

| Tree algorithms |

There are several algorithms that run on complete binary trees (Figure 1) having similar characteristics, like the aggregation operations of minimum and maximum. All processors have data, and we want to compute a global value that depends on all these values. Leaf processors send their value to their parent. Internal processors take the minimum (sum) of their own value and their child's value and then send the result to their parent. The final value will be computed at the root processor in $O(\log n)$ time. The communication in these algorithms requires one message over each edge for each global minimum (sum).

For a single minimum we have $K(M_{\text{minimum}}) = 1$.

Consider the contraction in Figure 2a. Let us call this contraction $M_{\text{minimum}}(p)$. Each edge in the original algorithm requires one message. Each edge in the smaller graph has 4 edges from the original graph. Since we have only one connection between the physical processors, we have 4 messages for each edge. For an arbitrary processor, the number of processors we have $K(M_{\text{minimum}}(p)) = \frac{p}{4}$.

A similar contraction to Figure 2a is touched on by Berman and Snyder[4]. Figure 3 shows that contraction. This is achieved by "folding" the tree. As Berman and Snyder notice, this contraction, $M_2$, has $K(M_{2}(\text{minimum } p)) = \frac{n}{p}$.

Consider the contraction in Figure 2b. Let us call this contraction $M_{\text{minimum}}(p)$. We note that each edge in the smaller graph has at most one edge from the original graph in each direction. For an arbitrary processor, the number of processors we have $K(M_{2}(\text{minimum } p)) = 1$.

Proposition 1 tells us that since $M_2$ has a smaller $K$, it is the preferable contraction. Both $M_1$ and $M_2$ depend on $n$ and $p$ for their cost. But, $M_1$ has a constant cost, regardless of $n$ and $p$. In fact, this contraction is optimum for all tree algorithms that have identical edge weights and unidirectional communication (all toward the root or all toward the leaves).

We first look for a lower bound. Since the tree is connected, the physical processors must be connected. This requires at least one incident edge for each physical processor. The smallest cost $K(M(A_p))$ would be where a maximum of one logical edge was mapped to a physical edge. Therefore, $K(M(A_p)) \geq K(A)$, the cost of the original algorithm.

Table 1: Timings of the minimum algorithm.

<table>
<thead>
<tr>
<th>Connection</th>
<th>16 on 4</th>
<th>64 on 16</th>
<th>64 on 16</th>
<th>128 on 16</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M_1$</td>
<td>11056</td>
<td>205856</td>
<td>35496</td>
<td>103201</td>
</tr>
<tr>
<td>$M_2$</td>
<td>1356</td>
<td>7682</td>
<td>8878</td>
<td>12005</td>
</tr>
</tbody>
</table>
Major orders in the binary-cube computes their product algorithm(2).

The Jacobi iterative method(1) and has been commonly conjectured that can be improved.

For instance for the Jacobi iterative method(1) and has been commonly conjectured that can be improved.

we have where

Figure 4: WAP organization

Figure 5: A contraction of 16 logical processes to 4 processors physical connection graph, shown in Figure 5, is a grid with end around (i.e. toroidal) connections. For each logical process in a physical processor, there are horizontal and vertical communication paths. Since we have logical processes in a processor, the number of logical edges using one processor-to-processor connection is \( \frac{n^2}{p} \). Since all horizontal and vertical edges have the same number of messages, \( n \), we have

\[ K(M_1(WAP_p)) = \frac{n^2}{p} \]

Consider the contraction in Figure 6. Let us call this contraction \( M_2(WAP_p) \). This is the contraction done by cutting the graph into \( p \) equal size connected subgraphs and assigning an entire subgraph to a processor. We see that only the perimeter processes have edges that go from processor-to-processor. Also, notice that no end around connections are needed. The number of communication paths over one processor-to-processor connection is \( \frac{n^2}{p} \). Each communication path requires \( n \) messages giving \( K(M_2(WAP_p)) = \frac{n^2}{p} \).

Comparing the two contractions, we see that \( K(M_2(WAP_p)) \) is smaller than \( K(M_1(WAP_p)) \) by a factor of \( \frac{1}{2} \). Proposition 1 tells us that \( M_2 \) is the better contraction. We conjecture that \( M_3 \) is the best contraction that can be achieved for grid algorithms. The basis for this conjecture is that this contraction has the smallest perimeter for a given area, and has been commonly used for contractions in published algorithms, for example for the Jacobi iterative method(1) and for the conjugate gradient method(6).

Both \( M_1 \) and \( M_2 \) were programmed using Fortran. Table 2 summarizes the results of the timings. As predicted, \( M_2 \) was the faster contraction, but because the communication time is not the only time consuming part in algorithms the difference is perhaps not as dramatic as might be seen on a larger problem.

**Binary n-cube algorithms**

We now look at two algorithms for the binary n-cube. The first algorithm is the divide-and-conquer algorithm for matrix product given by Nelson(10). The other algorithm is Backer's bison sorting algorithm(2).

The matrix product algorithm takes two \( n \times n \) matrices, \( A \) and \( B \), and computes their product \( C = AB \). \( A \) and \( B \) are assumed to be in row major order in the binary n-cube of order \( 2^k \), where \( k = \log n \). The algorithm views \( A \) and \( B \) as a \( 2 \times 2 \) matrix of \( \frac{n}{2} \times \frac{n}{2} \) matrices. The \( 2 \times 2 \) matrix algorithm is then used to multiply the submatrices. Figure 7 shows a order 4 cube laid out inLee plane using the CHIP architecture. The numbers in the boxes show the index of the matrix elements initially contained in that processor. We are assuming that the processors are numbered in row major order. The dotted boxes show cubes of order 2. These cubes, which generally have order \( 2(k-1) \) contain an \( \frac{n}{2} \times \frac{n}{2} \) submatrix of both \( A \) and \( B \). Note that these cubes are constructed by "removing" the edges of order \( k \) and \( 2k \), where and edge of order \( k \) connects processors that are \( 2^{k-1} \) distance apart.

To compute the \( 2 \times 2 \) matrix product, all processors exchange values of \( B \) on the order \( 2k \) edge and values of \( A \) on the order \( k \) edge. After the exchange, each cube of order \( 2(k-1) \) contains 4 submatrices of size \( \frac{n}{2} \times \frac{n}{2} \). This is all the data that is required for each cube of order \( 2(k-1) \) to compute its part of the \( 2 \times 2 \) matrix product independently. If the submatrix is not a single element, two matrix products of \( \frac{n}{2} \times \frac{n}{2} \) matrices are required. These matrix products are done using the same algorithm. Matrix addition is done element by element. Because corresponding elements of the matrices are contained in the same processor, no communication is required.

To find the cost of this cube matrix multiply, \( K(CM(M)) \), we need to find the edge with the most messages. At the first level of recursion, the order \( k \) and \( 2k \) edges were used to send a message each way. This is the only use of these edges in the algorithm. Therefore, \( w(e) = 1 \), where \( e \) is a order \( k \) or \( 2k \) edge. At the second level of recursion, two matrix products are computed using the order \( k-1 \) and \( 2k-1 \) edges. Each matrix product sends one message each way on each edge giving \( w(e) = 2 \). where \( e \) is a order \( k-1 \) or \( 2k-1 \) edge. At level 1 of the recursion, \( w(e) = 2^k \) messages over the order \( k-(k-1) \) and \( 2k-(k-1) \) edges. The recursion stops when we have order 2 cubes. This is at the log \( n \) level of recursion. There are \( \frac{n}{2} \) matrix multiplies done by order 2 cubes. These

![Figure 6: Another contraction of 16 logical processes to 4 processors.](image)

**Table 2: Timings of the WAP matrix multiply algorithm.**

<table>
<thead>
<tr>
<th>WAP matrix multiply: ticks for ( n ) (items) on ( p ) (processors)</th>
<th>16 on 4</th>
<th>64 on 4</th>
<th>16 on 4</th>
<th>64 on 4</th>
<th>256 on 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>( M_1 )</td>
<td>44654</td>
<td>111478</td>
<td>400452</td>
<td>901543</td>
<td></td>
</tr>
<tr>
<td>( M_2 )</td>
<td>31113</td>
<td>73088</td>
<td>221545</td>
<td>7 7646</td>
<td></td>
</tr>
</tbody>
</table>

![Figure 7: An order 4 binary n-cube.](image)
order 2 cubes use the order 1 and \( k+1 \) edges. Each matrix multiply sends 1 message each way giving \( w(e) = \frac{n}{2} \), where \( e \) is a order 1 or \( k+1 \) edge.

Since this is the largest value, \( K(CMM) = \frac{n}{2} \).

Consider any contraction, \( M(CMM, p) \) where \( p = 2^m \) for some \( m \leq k \). \( M(CMM, p) \) will map \( \frac{n}{p} \) logical processes to every processor.

This allows us to put a cube of order \( \left\lceil \frac{2k-m}{2} \right\rceil \) and order \( k+1 \) through \( \left\lceil \frac{2k-m}{2} \right\rceil \) into internal edges. This makes the edge of order \( k + \left\lceil \frac{2k-m}{2} \right\rceil + 1 \) the edge with the most messages. This edge is used by level \( k + \left\lfloor \frac{2k-m}{2} \right\rfloor \) of the recursion. From before we know that \( w(e) = \frac{2^{m+k}}{2^k} \).

Therefore, \( K(M(CMM, p)) = \frac{n}{2^k} \). Clearly, this contraction is better in terms of the number of messages over the busiest physical edge than any contraction that does not keep the high traffic logical edges internal to a processor.

By contrast, let us consider the Batcher bisonic merge sort. This sort runs on a order \( k \) cube to sort \( n = 2^k \) elements. The final sorting will have the smallest element in the first processor and the largest element in the last processor. Figure 8 shows a graphical representation of the algorithm. The arrows represent a data exchange and a compare, leaving the larger number at the end with the arrow and the smaller at the other end. It is obvious from the figure that the order 1 edge has the most messages. Therefore, \( K(SORT) = \log n \).

Again, to contract this algorithm, we see that we want to assign a sub-cube into a processor. Consider the contraction \( M(SORT, p) \) where the edges of order 1 through order \( \log p \) are mapped to internal edges. We are assuming that \( p = 2^m \), for some \( m \leq \log n \). This contraction assigns the biggest logical edges to internal edges. These edges carry \( n - \log p \) messages. Since each processor contains \( \frac{n}{p} \) logical processors, \( K(M(SORT, p)) = \frac{n(n - \log p)}{p} \). Any contraction that does not map these first \( \log p \) edges to internal edges will have a higher communication cost. These results agree with and explain the results of Hsiao [7], even though his final algorithm was embedded in a grid instead of another cube.

In comparing the contractions for matrix multiply and Batcher's sort we see that the same size cube is mapped in a different way when mapped to the same number of processors. The busiest edges are different for the two algorithms, thus, the contractions are different.

Conclusion

The algorithm contraction problem is an important problem for parallel programmers. The way in which an algorithm is contracted can have a significant affect on performance. Processor-to-processor communication can be used as a lower bound on the execution time of an algorithm. It is the processor-to-processor communication that is affected by different contractions.

We have looked at algorithms for the tree, grid, and binary n-cube interconnections. For each algorithm we have compared possible contractions of these algorithms. For trees, we proved that Leshner's layout technique was the best for contracting tree algorithms such as minimum and sums. For grid algorithms, we conjectured that coalescing by maximizing the area for a given perimeter is optimal for the algorithms with balanced edge loadings. Finally, we showed two algorithms for binary n-cubes that required different contractions to produce the optimal results for the algorithm.

References


Figure 8: Batcher's bisonic merge sort.
END
DATE
FILMED
DTIC
July 88