# Programming Solutions to the Algorithm Contraction Problem

## 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 bottle neck communication paths. We apply these tools to minimum, matrix product and sorting.
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 grid n x n 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≤i,j≤n/2) is associated with i-n,j+1, n-i+1,j and n-i+1,n-j+1.

Clearly, algorithm (1) should be contracted 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. As the other end of the spectrum, however, the programmer has "complete" knowledge about the algorithm. How should he be guided when performing his own contraction? In this paper we develop some apparatus to guide the programmer who must contract a algorithm. We will 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 a nontrivial 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 CHIPS[11] architecture is an example of this generic architecture.

---

**Abstract**

Algorithms for the partial 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 the graph. We apply these tools to minimum, matrix product and sorting. The communication graph for the processes sharing the same processor will become a bottleneck. Clearly, algorithm (1) should be contracted 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. As the other end of the spectrum, however, the programmer has "complete" knowledge about the algorithm. How should he be guided when performing his own contraction? In this paper we develop some apparatus to guide the programmer who must contract a algorithm. We will 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 a nontrivial 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 CHIPS[11] architecture is an example of this generic architecture.

---

**Figure 1:** A 5 processor contraction.

![Figure 1](https://example.com/figure1.png)

**Figure 2:** Two 4 processor contractions.

![Figure 2](https://example.com/figure2.png)
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 \((V_i,V_j) \forall i,j \in V\).

Let \( M(A,p) = B \) be a contraction of algorithm \( A \) into algorithm \( B \) where \( B \) uses \( p \) processors and \( p < |V| \). The contraction \( M \) maps elements of \( V_1 \) onto \( V_2 \) such that the number of elements of \( V_2 \) mapped to an arbitrary element of \( V_2 \) differs by no more than one from the number of elements of \( V_1 \) mapped to any other element of \( V_2 \).

Let \( w(e) \), the weight of \( e \), \( e = (V_i,V_j) \), 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 if \( T(M_1) < T(M_2) \) for all \( p \), if \( K(M_1(A,p)) < K(M_2(A,p)) \), then \( T(M_1(A,p)) < T(M_2(A,p)) \).

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 \( K(M_1(A,p)) \) time, which is less than \( K(M_2(A,p)) \). With a higher minimum communication time, we can not expect \( M_1 \) to execute in less time than \( M_2 \). If the processors have a large amount of computation in ratio to the communication, the computation time will dominate, yielding near equal times. Even in this case, \( M_1 \) uses less time for communication than \( M_2 \). This proposition then motivates us to map the busiest edges of an algorithm to minimal edges.

**Case Studies**

We now look at several parallel algorithms and some contractions. We approach these by considering algorithms with similar communication graphs. The three graphs considered are the tree, grid, and binary n-cube.

**Tree Algorithms**

There are several algorithms that run on complete binary trees (Figure 1) having similar characteristics, like the aggregation operations of minimum and global sum. All processors have a value 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 children's values and then send the results 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 \( K'(\text{minimum}) = 1 \).

Consider the contraction in Figure 2a. Let us call this contraction \( M_1(\text{minimum} \cdot p) \). 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 one connection between the physical processors, we have 4 messages for each edge. For an arbitrary \( n \) (size of original algorithm) and \( p \) (the number of processors) we have \( K(M_1(\text{minimum} \cdot p)) \).

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

Consider the contraction in Figure 2b. Let us call this contraction \( M_3(\text{minimum} \cdot p) \). We note that each edge in the smaller graph has at most one edge from the original graph in each direction. For an arbitrary \( n \) and \( p \) we have \( K(M_3(\text{minimum} \cdot p)) \).

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

**Proof:** For the mapping \( M_3(\text{Ap}) \), each processor contains a complete subtree and an "extra" node. The extra nodes are used in the tree above the subtrees contained in the processors. Therefore, there is at most four external connections. Of these, 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 cannot be connected to the same physical processor. (It is possible to have two of these edges over the same physical edge, but the data moves in opposite directions.) This gives the same weight to the physical edges as the original edges. Therefore, \( K(M(A,p)) \).

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 Parallel Prolog programming environment[12]. Both \( M_1 \) and \( M_3 \) were programmed. Each contraction was tested running 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.

**Grid Algorithms**

We next look at algorithms that run on a grid interconnection. Consider the matrix product algorithm for the Wavefront Array Processor [WAP][8]. It uses \( n^2 \) processors for the \( n \times n \) matrix product \( AB = C \). The data is fed in along the top \( n \) processes and from the left \( n \) processors. The matrix \( A \) is arranged to enter column by column, starting with the first column. The matrix \( B \) is arranged to enter row by row, starting with the first row. (See Figure 4.) All processors execute identical procedures. The results, \( C_{ij} \), is initialized to zero. A loop is executed \( n \) times that reads an \( A \) value from the left and a \( B \) value from above, multiplies them together, and adds the result to \( C_{ij} \). The \( A \) and \( B \) values are sent to the right and down, respectively. This causes the upper left processor to be the first processor to start execution. As the data moves into the array, there is a wavefront of executing processors on the cross diagonal. Each edge is used to send all of one row of \( A \) or one column of \( B \). For the WAP algorithm we have \( K(WAP) = n \).

Consider the contraction in Figure 3. Let us call this contraction \( M_4(\text{WAP} \cdot p) \). This is the contraction done by cutting the graph into \( p \) equal size connected subgraphs and assigning one process from each subgraph selected from corresponding positions to a single processor.

<table>
<thead>
<tr>
<th>Minimum ticks for n (terms) on p processors</th>
<th>Connection</th>
<th>16 on 4</th>
<th>64 on 16</th>
<th>64 on 4</th>
<th>256 on 16</th>
</tr>
</thead>
<tbody>
<tr>
<td>( N )</td>
<td>111.650</td>
<td>205.585</td>
<td>354.985</td>
<td>1035.01</td>
<td>1205.67</td>
</tr>
<tr>
<td>( M )</td>
<td>43.566</td>
<td>75.622</td>
<td>88.786</td>
<td>1205.67</td>
<td>1205.67</td>
</tr>
</tbody>
</table>

Table 1: Timings of the minimum algorithm.
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 \( 2k \), 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 in a 4x4 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 \( \frac{n}{2} \times \frac{n}{2} \) submatrices of both \( A \) and \( B \). Note that these cubes are constructed by removing the edges of order \( k \) and \( 2k \), where \( k \) 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, \( K(CMM) \), 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 uses 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, \( w(e) = 2 \) messages over the order \( k-1 \) and \( 2k-1 \) edges. The recursion stops when we have order 2 cubes. This is at the level \( \log n \) of recursion. There are \( \frac{n}{2} \) matrix multiplies done by order 2 cubes. These

![Figure 4: WAP organization](image1)

![Figure 5: A contraction of 16 logical processes to 4 processors](image2)

![Figure 6: Another contraction of 16 logical processes to 4 processors](image3)

![Figure 7: An order 4 binary n-cube](image4)
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 \( \frac{n^2}{p} \) logical processes to every processor.

This allows us to put a cube of order \( \log \frac{n}{\rho} = 2k-m \) into each processor. The processor-to-processor connection graph is also a cube and is of order \( m \). Each processor-to-processor connection supports \( \frac{n+1}{p} \) communication paths in the original graph. The real question is which sub-cube do we map to each processor. The cost of the contraction, \( K(M(CMM_p)) \) will be \( \frac{n^2}{p} \) times the maximum \( w(e) \), where \( e \) is mapped to a physical edge. If \( e \) is order 1 or 2k+1 from the original cube, \( K(M(CMM_p)) = \frac{n^2}{2p} \).

Consider the contraction that maps the edges of order 1 through \( \left[ \frac{2k-m}{2} \right] \) and order k+1 through k+1 into internal edges. This makes the edge of order \( k+1 \left[ \frac{2k-m}{2} \right] + 1 \) the edge with the most messages. This edge is used by level \( k+1 \left[ \frac{2k-m}{2} \right] + 1 \) of the recursion. From before we know that \( w(e) = 2 \). Therefore \( K(M(CMM_p)) = \frac{n^2}{2p} \). 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 bitonic 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 wish 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 contrast 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 busiest logical edges to the internal edges. These edges carry \( \log n - \log p \) messages. Since each processor contains \( \frac{n}{p} \) logical processors, \( K(M(SORT_p)) = \frac{n(\log n - \log p)}{p} \). 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 for 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 tree, we proved that Leiserson'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**

END
DATE
FILMED
DTIC
July 88