A Review of Data Placement Optimisation for Data-Parallel Component Composition

Olav Beckmann and Paul H J Kelly

Department of Computing, Imperial College,
180 Queen's Gate, London SW7 2BZ, United Kingdom
{ob3,phjk}@doc.ic.ac.uk
www.doc.ic.ac.uk/{~ob3,~phjk}

Abstract. Constructive methods for parallel programming are characterised by the composition of optimised, parallel software components. This paper concerns data placement, a key cross-component optimisation for regular data-parallel programs.

This article is a survey of data placement optimisation techniques. The main contributions are (1) a uniform terminology, which identifies analyses of the problem which have proven fruitful, (2) a taxonomy of versions of the problem, distinguished by the efficiency with which they can be solved, (3) a discussion of open problems, challenges and opportunities for further progress in the area, and (4) a discussion of the significance of these results for constructive methods in parallel programming. We observe, in particular, the role of skeletons in restricting program graph structure to ensure that optimisation is tractable.

1 Introduction

This is a review paper on parallel data placement optimisation for data-parallel programs. The key challenge in constructing efficient parallel programs for distributed memory multiprocessors has shifted from automatically generating parallel components to optimising the composition of existing components. In the domain of purely data-parallel programs, this problem is somewhat more tractable than in more general domains: the performance of a data-parallel program constructed from existing parallel components depends largely on the choice of data placements.

1.1 Contributions of this Paper

– We propose a taxonomy of different instances of the parallel data placement problem. Our classification clarifies the complexity of different instances of the problem.
– We review and summarise existing work in this problem domain, illustrating how the complexity and accuracy of the proposed solutions depends on the method adopted.
– We outline outstanding issues within the domain.
– We discuss how the current state of the art in parallel data placement optimisation can best be synthesised with existing constructive methods for parallel programming, and point out areas for future research.

1
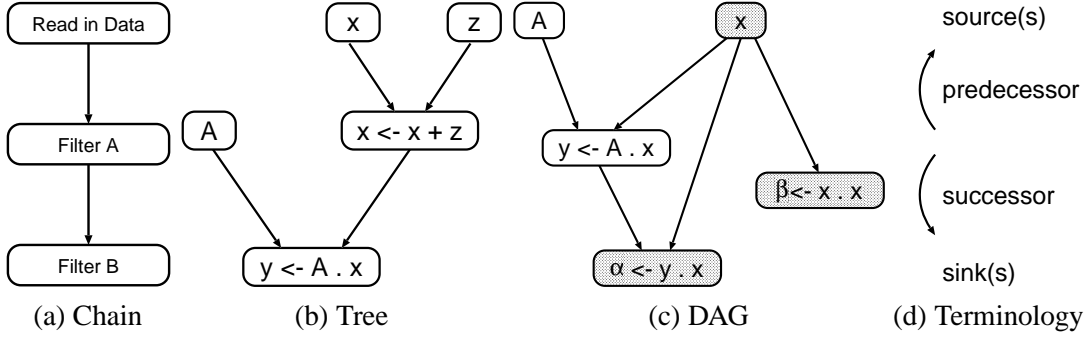
Fig. 1. Three classes of program graph. The DAG is distinguished by the presence of a shared node and by the fact that there may be more than one sink. The right hand side of the diagram (d) shows the terms we will use to refer to different nodes in program graphs in this discussion.

1.2 Terminology

Mace [22] defines the parallel data placement problem in one very general form: the objective is to minimise the overall execution time of a program graph in which the nodes represent parallel operations and data is communicated along the edges. Operations may accept inputs and generate output in a number of different parallel data placements. Each operation is assigned a cost, which is a function of the placement of its input and output data. Data may be redistributed in between operations, and there is a cost function associated with redistributions.

– *Data-Parallel Operations*
  Data-parallel operations are elementwise array operations, such as provided by Fortran 90. In the context of our own work [7, 8], array operations are calls to our library of parallel numerical routines. We assume that all operations produce exactly *one* result.
– *Parallel Data Placements*
  We define parallel data placement in general as a relation that specifies which subarray(s) of an array are stored on the individual nodes of a distributed memory multiprocessor.
  It is common practice to represent parallel data placement as a mapping of multiple stages: *alignment*, *distribution*, and possibly *replication*. Alignment maps array index vectors onto an auxiliary Cartesian grid, known as a template or virtual processor grid. Distribution, also known as folding, maps virtual processors onto physical ones. Replication is sometimes handled implicitly as part of the alignment stage, for example in HPF [18], or, according to our own proposal [8], as an explicit third stage preceding alignment.
  Mace [22] calls parallel data placements *shapes*. We will therefore use $\mathcal{S}$ to denote the set of all possible placements and s to denote the size of $\mathcal{S}$ (if the set is countable).
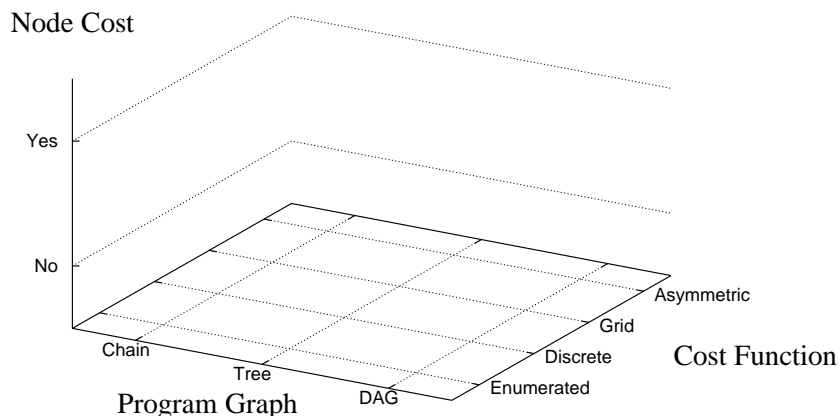– *Implementations of Data-Parallel Operations*
  Data-parallel operations may have a number of different *implementations*, which often have different associated computation costs. We give two alternative, more precise definitions for implementations in Section 2.3.
  Mace [22] uses the term *methods*. We will use $\mathcal{M}$ to refer to the set of all possible methods and m to denote the size of $\mathcal{M}$.
– *Program Graphs*
  In the context of this article, we will use the term program graph for the data flow graph

Fig. 2. Illustration of the problem domain of parallel data placement in data-parallel programs.

of a parallel program, where the nodes represent data-parallel operations, and data flow is represented by edges. The direction of all edges will be in the direction of data flow.[1] We do not impose any restrictions on the arity of nodes or on the structure of the graph, except that we do not consider control flow edges at this stage, hence the graphs are not cyclic. Note that since each operation has exactly one result, we can uniquely identify nodes by the result of the operation they represent.

– *Array Redistributions*
  In our program graphs, the placement of arrays need not match at the source and sink of edges. If that is the case, the placement of the array needs to be changed at runtime. We will term this a redistribution.

– *Cost Functions*
  In any optimisation, candidate solutions are evaluated according to some cost function. In our problem domain, the purpose of the cost function is to assess the quality of a proposed set of parallel data placements for a particular program graph. The cost will in general depend on both the cost of executing the individual nodes and on the cost of any redistributions. In Section 2.1, we explain why the cost functions for modelling redistribution cost are generally required to be *metrics*.

1.3 Scope of this Review

In this paper, we review techniques for minimising the overall execution time of a program graph as defined above. In purely data-parallel programs, the nodes of the graph are executed *sequentially* to each other, i.e. we do not consider task parallelism between nodes. This means that parallel data placement becomes the key determinant of program performance. This observation is confirmed for example by Chatterjee *et al.* [13]. The problem we have to solve is therefore to select a set of parallel data placements for all arrays in a program graph, both at the sink and at the source of each edge, such that the overall execution time of the program graph is minimised.

---

[1] This corresponds to the direction in which data dependence arrows are drawn in a dependence graph. The edges in Mace's program graphs [22] have the opposite orientation.

1.4 Overview of the Parallel Data Placement Problem

In this article, we will argue that the complexity of the parallel data placement problem is determined by three key parameters. These are illustrated in Figure 2 and are as follows:

1. *The structure of the program graph*
   We distinguish between three different classes of program graph: chains, trees and DAGs. These are illustrated in Figure 1. DAGs are substantially harder to optimise than trees. We explain why that is so in Section 2.4.
2. *The cost function used as the objective when optimising*
   The properties of the cost function used in optimisation have a strong effect on the complexity of the problem to be solved. Some types of parallel data placement may be modelled adequately with a very simple cost function (such as the discrete metric, which assigns every redistribution either a cost of 1 or 0). For other aspects of data placement, this would be insufficient. The precise meaning of the different cost functions shown in Figure 1 will be addressed in subsequent sections of the paper.
3. *Whether or not node costs have to be taken into account*
   In some situations, the computation cost at the nodes in a graph may be unaffected by data placements so that we are only having to minimise the cost of redistributions. In other cases the computation cost is affected by data placement. We show examples for this case in Section 5.

In the following sections, we will always indicate which part of the problem domain that we have mapped out in Figure 2 we are currently addressing.

1.5 Structure of this Paper

In the next section (2), we review solutions to the parallel data placement problem for the important case where the cost function is formulated as a lookup table. This variant of the problem is the easiest to understand and exposes well the complexity of the problem. However, the solution here actually also has the worst time complexity. Following that, in Section 3, we review solutions for optimising only the alignment part of parallel data placement. Proposed algorithms in that domain rely on using cost functions that are expressed in a more compact form. Next, in Section 4, we consider optimisation of data replication. The issue with replication is that it has no natural representation as a function; we discuss the implications of that and possible solutions. Section 5 shows an example where the computation cost of nodes is affected by data placement: distribution. Finally, Section 6 reviews the relationship of the problem domain of this paper to other research activities, and Section 7 concludes.

2 Solutions for Explicitly Enumerated Placements

We begin by reviewing solutions to the parallel data placement problem for the case where the cost functions, both for node costs and for redistributions, are expressed in the form of a lookup table. Specifically, this means that

– There is a single set $\mathcal{S} = \{S_1, \ldots, S_s\}$ of s parallel data placements, and all arrays in the program under consideration can adopt all s placements. The algorithm easily generalises to the case where different arrays can adopt placements from different sets. The crucial point is that we have a finite set of explicitly enumerated placements, and that we do not make use of any specific properties of this set when optimising.

– The cost function for redistributions is represented as a pre-computed table of size s^2:

$$C_{\text{Red}}[S_{\text{from}}][S_{\text{to}}], \qquad S_{\text{from}}, S_{\text{to}} \in \mathcal{S} . \tag{1}$$

2.1 Dynamic Programming Solution for Chains

We begin with the simplest class of program graph, chains. The algorithm we propose below is based on the algorithm proposed by Mace [22, 23] for trees (see Sections 2.2 and 2.3). Observe that for chains, each operation op has exactly one input and one result. Leaf nodes represent evaluated data or input operations; in either case their data placement is *fixed*. We will denote the required input data placement of op by $op.S_{\text{in}}$, the result placement by $op.S_{\text{out}}$ and the cost of computing op with these placements by $op.C_{\text{Node}}$. Finally, we will denote the operation to be performed at some node x by $x.op$ and the predecessor of node x by $x.pred$. In addition to assuming that the program graph is a chain, we will initially also assume that each operation has only one implementation, which for now we take to mean that there is *one* specific pair $(S_{\text{in}}, S_{\text{out}})$ of input and result placements. We address the issue of multiple implementations in Section 2.3.
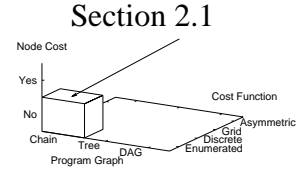
Algorithm 1 (Solution for Chains, One Implementation). We can now give the following recursive formulation for the minimum cost $C_{\min}(x, S_i)$ of evaluating a chain of data-parallel operations with sink x into placement S_i.

$$
\begin{aligned}
C_{\text{tmp}}(x, S_i) &= x.op.C_{\text{Node}} + C_{\text{Red}}[x.op.S_{\text{out}}][S_i] \\
C_{\min}(x, S_i) &= C_{\text{tmp}}(x, S_i) && \text{if } x \text{ is the leaf} && (2) \\
&= C_{\text{tmp}}(x, S_i) + C_{\min}(x.pred, x.op.S_{\text{in}}) && \text{otherwise} && (3)
\end{aligned}
$$

This recursive algorithm to find $C_{\min}(x, S_i)$ takes $\Theta(n)$ steps[2] for a chain of n nodes. The overall minimum cost for calculating node x is now theoretically

$$C_{\min}(x) = \min_{S_i \in \mathcal{S}}(C_{\min}(x, S_i)) , \tag{4}$$

which could be calculated in $\Theta(n+s)$ steps. However, with one implementation, we will for all practical cases (if C_{Red} is a metric — explained shortly), have

$$C_{\min}(x) = C_{\min}(x, x.op.S_{\text{out}}) \quad , \tag{5}$$

which means that the problem may be solved simply in $\Theta(n)$ steps. The only remaining task is to read off the actual shapes S_i which resulted in the minimum cost.

Metric Properties of Redistribution Cost. Chatterjee *et al.* [12–14, 17] state that all their cost functions for redistributions, which they call *distance functions*, ought to be metrics. A distance function $d : \mathcal{S} \times \mathcal{S} \longrightarrow \mathbb{R}$ is a metric if for all $S_i, S_j, S_k \in \mathcal{S}$,

$$
\begin{aligned}
& d(S_i, S_j) \geqslant 0 && \text{and} \quad d(S_i, S_j) = 0 \Leftrightarrow S_i = S_j && \text{Nonnegativity} && (6) \\
& d(S_i, S_j) = d(S_j, S_i) && && \text{Symmetry} && (7) \\
& d(S_i, S_j) \leqslant d(S_i, S_k) + d(S_k, S_j) && && \text{Triangle inequality .} && (8)
\end{aligned}
$$

---

[2] We will use the complexity notations proposed by Knuth [20].

Nonnegativity. There is a strong case that in real-world computer systems, costs are non-negative. If that is the case, we can use Equation (5), rather than (4) in Algorithm 1.

Triangle inequality. Real systems do not necessarily satisfy the triangle inequality. Two-phase random routing [28], for example, makes use of the fact that performing two communications via a random intermediary, rather than sending one direct message, can avoid contention in networks. However, if we do allow cost functions which do not satisfy this property, it is very difficult to bound the search space for the above algorithm: We have to check (recursively!) whether any redistribution can be replaced with two cheaper redistributions via an intermediate placement.

Symmetry. It appears that the parallel data placement algorithms proposed by Chatterjee *et al.* do not actually rely on the symmetry property of the distance function. There are some interesting models of redistribution cost that are not symmetric. Replication is one such example, which we discuss further in Section 4. Another example are some instances of the BSP cost model.

Weighted vs. Unweighted Optimisation Problem. Strictly speaking, the entries in the redistribution cost table ought to be functions that take array size as an argument and return a cost. Chatterjee *et al.* [12, 14] approximate this by defining redistribution cost as the cost of moving one unit data and then assigning to all edges E in the program graph a weight w_E which is the number of data units communicated along that edge. The cost contribution of the edge is set to $w_E \cdot C_{\text{Red}}[S_{\text{from}}][S_{\text{to}}]$. The resulting optimisation problem is called the *weighted* parallel data placement problem and is in some instances harder to solve than the unweighted problem [14].

In this paper, we concentrate on the unweighted problem, i.e. we will assume that we are working with one fixed array size, which allows the entries in table C_{Red} to be scalars. This corresponds to the approach of Mace [22] and Gilbert and Schreiber [17].

2.2 Dynamic Programming Solution for Trees

We now consider the case where the program graph is a tree rather than a chain. For an a-ary tree, we will denote the ith predecessor of a node x by $x.\text{pred}[i]$, and the placement in which the operator at node x requires its ith operand by $x.\text{op}.S_{\text{in}}[i]$.

Algorithm 2 (Solution for Trees, One Implementation). The minimum cost of evaluating an a-ary tree with sink x into placement S_i is

$$
\begin{aligned}
C_{\text{tmp}}(x, S_i) &= x.\text{op}.C_{\text{Node}} + C_{\text{Red}}[x.\text{op}.S_{\text{out}}][S_i] \\
C_{\min}(x, S_i) &= C_{\text{tmp}}(x, S_i) && \text{if } x \text{ is a leaf} && (9) \\
&= C_{\text{tmp}}(x, S_i) + \sum_{0 \leqslant j \leqslant a} C_{\min}(x.\text{pred}[j], x.\text{op}.S_{\text{in}}[j]) && \text{otherwise} && (10)
\end{aligned}
$$

If C_{Red} is a metric, the overall minimum cost is given by Equation (5), as in the case for chains. The time complexity of this algorithm is $\Theta(n)$.
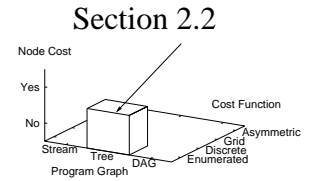
Chains and Trees. It is interesting to note that the time complexity for solving the optimal parallel data placement problem is independent of the arity of the program tree, with chains simply being a 1-ary tree. The reason for this is that the choice of optimal placements for all input-branches to a node is independent, as long as the program graph is a tree.

2.3 Solution with Different Implementations for Nodes

We now move on to the case where operations can have m different implementations. We consider two definitions of implementations.

Section 2.3

First Definition (Mace). Mace [22] characterises an implementation[3] of a data-parallel operation as a distinct tuple of data placements for the operands and the result of the operation. For example, let operation op1 take 2 operands. One implementation of op1 might be characterised by the fact that its operands need to be in placements S_1 and S_2 respectively, and that the result will have placement S_3. We write this as a tuple (S_1, S_2, S_3). If op1 can also be performed leaving its result in placement S_4 instead of S_1, op1 is said to have two implementations. We will denote the required input data placement of implementation M_k of op by op.M_k.S_{in}, the result placement by op.M_k.S_{out} and the cost of computing op with these placements by op.M_k.C_{Node}. If there are any implementations which have the *same* input and output placements, but different costs, we can eliminate the more expensive methods in a preprocessing stage.
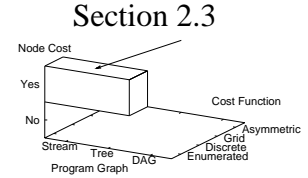
Algorithm 3 (Solution for Trees, Implementations as Tuples). We have:

$$C_{\text{tmp}}(x, M_k, S_i) = x.\text{op}.M_k.C_{\text{Node}} + C_{\text{Red}}[x.\text{op}.M_k.S_{\text{out}}][S_i]$$

$$C_{\text{min}}(x, S_i) = \min_{M_k \in \mathcal{M}} \left(C_{\text{tmp}}(x, M_k, S_i) \right) \qquad (11)$$

$$\text{if } x \text{ is a leaf}$$

$$= \min_{M_k \in \mathcal{M}} \left(C_{\text{tmp}}(x, M_k, S_i) + \sum_{0 \leqslant j \leqslant a} C_{\text{min}}(x.\text{pred}[j], x.\text{op}.M_k.S_{\text{in}}[j]) \right) \quad (12)$$

$$\text{otherwise}$$

The complexity of calculating $C_{\text{min}}(x, S_i)$ now is $\Theta(mn)$. Furthermore, the presence of multiple methods with different result placements means that we can now not immediately see which placement for the sink will result in overall minimum cost. We have:

$$C_{\text{min}}(x) = \min_{S_i \in \mathcal{S}} \left(C_{\text{min}}(x, S_i) \right) \qquad . \qquad (13)$$

The complexity of Algorithm 3 is $O(msn)$. We could have up to s placements with distinct costs for the sink x since candidate solutions that result in the same placement for x are reduced to the cheapest solution for that placement by Equations (12) and (11). If there is some $S_i \in \mathcal{S}$ such that no $M_k \in \mathcal{M}$ exists with $S_i = x.\text{op}.M_k.S_{\text{out}}$, then that placement S_i could be eliminated from the search. Therefore, the complexity is $O(msn)$ rather than $\Theta(msn)$.

---

[3] Mace uses the term *method*.

Discussion. Note that given the above definition of implementations and the assumption that implementations with identical input and output placements but higher costs are eliminated, m has to be a number between 1 and s^{a+1} where a is the arity of the operators in the tree. Therefore, if we assume that the maximum possible number of implementations does indeed exist, the complexity of the algorithm is $O(s^{a+2}n)$, i.e. $O(s^3 n)$ for chains.

The algorithm we have described in this section is similar to the original proposal for a dynamic programming solution to the parallel data placement problem for trees by Mace [22, 23]. Versions of the algorithm for chains have been proposed by To [27, page 223] and Skillicorn *et al.* [26].

The first definition treats as separate implementations *all* possible combinations of input and output placements. Thus, if we might calculate a vector addition with either both operands and the result blocked over the rows of a processor mesh, or blocked over the columns, these would be counted as two distinct implementations. The proposal we describe in the second definition aims to capture the fact that there exists a common pattern between such implementations.

Second definition (Implementations as Placement Relations). Assume that the data placement we are optimising takes the form of an invertible mapping. This includes, for example, *alignment*, which may be expressed in the form of an invertible affine mapping $f(i) = A.i + t$ from array index vectors i onto virtual processor indices [7]. Given invertibility, we may always, for any pair of placement functions f and g, calculate a *relationship function* $r = g \circ f^{-1}$ with the property that $g = r \circ f$.

We now define an implementation M_k of an a-ary data-parallel operation as a tuple $M_k = (r_1, \ldots r_a)$ of relationship functions which describe the relationship between the placements of *result* and *operands*, as follows: $\mathrm{op}.S_{\mathrm{in}}[i] = r_i \circ \mathrm{op}.S_{\mathrm{out}}$. We will denote the relationship function for operand j under implementation M_k by $M_k.r_j$.

Example. A very important class of data-parallel operations are those that can be expressed as binary array operations in Fortran 90. A compiler can generate code for these in any placement, as long as both operands and the result are aligned. Under the definition we have outlined above, we may represent this by saying that these operations have *one implementation*, with $r_1, r_2 = id$, where *id* is the identity affine function which maps every placement to itself.

Algorithm 4 (Solution for Trees, Implementations as Placement Relations). Notice that under this definition, the result of each operation may take *any* placement from within $\mathcal{S}$, it is the relationship to the operands that is defined by the implementation. The placement of leaf nodes is *fixed*. The minimum cost $C_{\min}(x, S_i)$ of evaluating node x into placement S_i may be calculated as
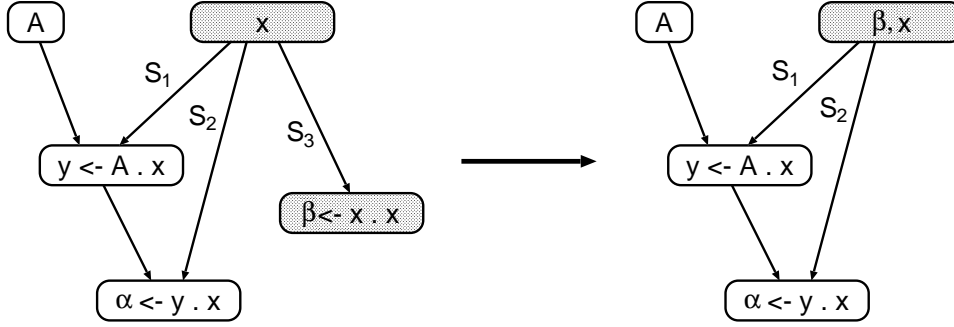
$$C_{\mathrm{tmp}}(x, M_k, S_j, S_i) = x.\mathrm{op}.M_k.C_{\mathrm{Node}} + C_{\mathrm{Red}}[S_j][S_i]$$

$$C_{\min}(x, S_i) = C_{\mathrm{tmp}}(x, M_1, x.\mathrm{op}.M_1.S_{\mathrm{out}}, S_i) \tag{14}$$

$$\text{if } x \text{ is a leaf}$$

$$= \min_{\substack{S_j \in \mathcal{S} \\ M_k \in \mathcal{M}}} \left(C_{\mathrm{tmp}}(x, M_k, S_j, S_i) + \sum_{0 \leqslant l \leqslant a} C_{\min}(x.\mathrm{pred}[k], x.\mathrm{op}.M_k.r_l \circ S_j) \right) \tag{15}$$

$$\text{otherwise} \quad .$$

Fig. 3. Example of DAG transformation, collapsing nodes x and β into one.

The complexity of calculating $C_{\min}(x, S_i)$ is $\Theta(msn)$. The overall minimum cost $C_{\min}(x)$ for evaluating node x may again be calculated with Equation (13), resulting in an overall complexity of $\Theta(ms^2n)$.

Discussion. This algorithm is based on the assumption that implementations which are defined as above have constant parallel cost for node operations, independent of the actual placements chosen for the result and operands.[4] We argue, therefore, that the above definition of implementations captures the interestingly different cases of 'implementations' as viewed under the first definition.

The algorithm we have outlined in this section provides a solution for the large class of Fortran 90 array operations in $\Theta(s^2n)$ time. This corresponds exactly to the algorithm outlined by Chatterjee *et al.* in [13]. The definition for implementations used in this chapter has been previously stated in [7, 8].

2.4 Solution for DAGs

Mace [22] shows the parallel data placement problem to be NP-complete when the program graph is a general DAG. An intuition as to why the problem is so much harder for DAGs than for trees may be had from the following observations:

– The contribution towards the costs of a node from that come from calculating the operands in the required placement (Equation 10) are no longer independent.
– A DAG may have multiple sinks. The minimum cost of computing the DAG need *not* place the sinks in minimum cost positions, as we have assumed for trees in Equation (13).

However, Mace also demonstrates that a polynomial algorithm may be found for a subclass of DAGs known as *collapsible DAGs*, and Mace argues [22, page 10] that a very significant proportion of real program DAGs fall into this subclass.

---

[4] The reasoning behind this assumption is that any differences in parallel completion time for the node operations are most likely due to intrinsic communication which is necessary as part of the operation; purely sequential computation time should be unaffected by data placement. The only situation where we might expect to observe a change in the placement relation of result and operands is if there is a change in intrinsic communication, which in turn would give a different implementation and therefore a different cost.
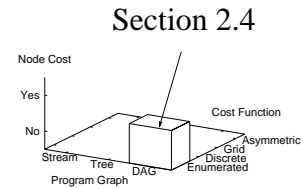
This result has parallels in the field of automatic code generation. Efficient algorithms are known for the case where the program graph is a tree [1], but the problem is known to be NP-complete for general DAGs [2]. Collapsible graphs were introduced by Prabhala and Sethi [24], who showed that the optimal code generation problem may be solved in polynomial time for this subclass of DAGs.

Mace presents an algorithm which solves the parallel data placement problem for collapsible DAGs. The algorithm relies on a series of graph reductions, each of which collapses two nodes into one and deletes all edges connecting those nodes. All nodes in a DAG are required to have in-degree *or* out-degree of 1 or less. DAGs which do not meet this property can be transformed into ones that do; however, that may result in an $O(n)$ increase in the number of nodes the algorithm has to run over.

We illustrate one of the transformations which may be applied to a DAG in Figure 3. The idea behind each transformation is to isolate the contribution to the overall cost of the DAG of those edges that connect the two nodes being collapsed, and to minimise their cost. In the example in Figure 3, the cost of node x is a function $\text{Cost}(x.\text{op}, S_1, S_2, S_3)$ of the operator at node x and the data placements assigned to the edges which touch it. Similarly for node β: $\text{Cost}(\beta.\text{op}, S_3)$. The cost for the combined node (β, x) is

$$\text{Cost}((\beta, x).\text{op}, S_1, S_2) = \min_{S_j \in S} \big(\text{Cost}(x.\text{op}, S_1, S_2, S_j) + \text{Cost}(\beta.\text{op}, S_j) \big) \qquad (16)$$

The algorithm has complexity $O(s^{d+1}(n-1))$ where d is the out-degree of the most highly-shared node.

2.5 Conclusion for Enumerated Cost Functions

The advantage of the algorithms for explicitly enumerated cost functions is that efficient algorithms are known for solving the parallel data placement problem optimally for a large number of program graphs. The major drawback of enumerated cost functions and enumerated placements is that for many real applications, the number of feasible placements is infinite (shift-offsets in alignments, skewings, block-cyclic distributions).

3 Compact Dynamic Programming Solution for Alignment

The solution to the problem of having a large, or infinite, number of possible placements is to represent the cost function in a more compact way than a lookup table, and then make use of the *structure* of the cost function in optimisation. Chatterjee *et al.* [13, 14] describe a series of *compact dynamic programming* algorithms which solve the parallel data placement problem for axis, stride and offset alignment for trees in a way that makes use of the structure of the cost function.
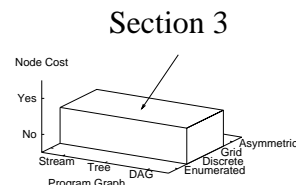
Discrete metric for axis and stride alignment. Axis re-alignments are a generalisation of transpose: permutations of the mapping of array axes onto processor axes. The discrete metric assigns a cost of 1 to every change in either stride or axis alignment. These redistributions are general data exchanges, which might be implemented through all-to-all communication. The assertion is that they have a high, fixed cost.

In terms of optimisation, the discrete metric intuitively has the property that each node can be either placed in the same placement as its parents, or, if that is not possible, all alternatives have the same cost penalty.

The algorithm which Chatterjee *et al.* propose that uses the discrete metric for axis and stride alignment solves the same problem for which we described a $\Theta(s^2 n)$ solution in Section 2.3 in only $O(nh)$ time, where h is the height of the tree.

Note, however, that Chatterjee *et al.* do not allow skewings in their alignments. These are cheaper communications than, say, a transpose, which means that the discrete metric is inadequate in comparing their cost to that of a transpose.

Grid metric for offset alignment. For offset alignment, the discrete metric is clearly unsatisfactory, since it would assign the same cost to any offset / shift of data, independent of the size of the offset.

Chatterjee *et al.* propose to use a *grid* metric for offset alignment: the cost of a redistribution is modelled as the "distance" between the placements. The resulting compact dynamic programming algorithm has complexity $O(kn)$, where k is the number of dimensions of the virtual processor grid. Note that since the number of possible offset alignments is generally very large (in fact, n^k), the general algorithm from Section 2.3 is completely infeasible. If the distance metric is to take account of wrap-around, the solution becomes slightly more complex: $O(knh)$. This is known as the *ring* metric.

Heuristic Solution for DAGs. The above algorithms work for trees. In [9, 13], Chatterjee *et al.* present heuristic algorithms which solve the problem for DAGs. The algorithm for axis and stride alignment, based on the discrete metric, is a modified tree algorithm, while an integer linear programming algorithm is used to minimise the cost due to offset realignments. The latter is heuristic in that Chatterjee *et al.* use the real solution as an approximation to the integer solution.

Discussion. The aim of this section has been to show how, in cases where the enumerated algorithms are infeasible because the number of placements is very large, solutions may be found that make use of the structure of the cost function to direct the search.

4   Solution for Replication

Replication is a difficult problem in terms of data placement optimisation because the natural representation for replication is not even a function: one location is mapped onto many. Several authors [8, 9, 29] have therefore proposed representing the *inverse* of copying, i.e. a function which maps multiple locations onto one. Notice that this representation is now not invertible: such a function is not injective and hence cannot be inverted. This means that according to this representation, the only "*replication redistributions*", i.e. changes in replication that can be represented, are broadcasts. Once an array is replicated, it remains replicated and all nodes "downstream" in the DAG from the broadcast, except for the results of reductions, have to be replicated aswell. A further observation is that the natural replication cost function is not a metric because it is *asymmetric*.
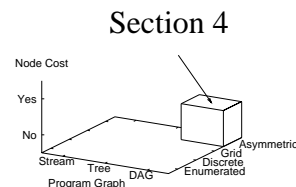
Chatterjee, Gilbert and Schreiber [10] make use of the rather unique properties of replication by proposing to solve the parallel data placement problem for replication using network flow: The fact that only those replication redistributions can be represented which allow an increase in replication means that the max flow / min cut theorem may be used to find a set of placements that minimises the total number of broadcast operations. Notice that the network flow algorithm is naturally an algorithm that works for general DAGs and therefore subsumes the case for trees.

We have proposed a new representation for replication [8] which solves the problem of replication not being invertible. We can therefore represent both increases in replication, to be implemented by broadcasts, *and* reductions in replication, which may be done by simply "dropping" data. This means that we can represent a wider set of possible solutions. However, it also means that the problem can no longer be solved in the same way as proposed by Chatterjee *et al.*. We have outlined a possible heuristic algorithm in [8].

The trade-off between the two proposals is a wider set of possible placements with a harder optimisation problem (our proposal) *vs.* a more restrictive set of available solutions with a precise algorithm (Chatterjee *et al.*).

5   Solution for Distribution

Distribution determines how many data elements of each array, in each dimension, get allocated to each physical processor. Therefore, distribution clearly has an impact on the parallel *computation* time of the operations at the nodes in the DAG. This makes distribution a hard problem to optimise. There exists a large body of research on this topic, some key references are listed by Sheffler *et al.* in [25]. The approaches may be separated into solving either the *static* or the *dynamic* distribution problem. Static in this case means that each array has only one distribution. If redistributions are permitted, the problem is termed *dynamic*. In the context of component composition, we are interested in the dynamic instance of the problem, which has been shown to be NP-complete by Kremer [21].

Chatterjee *et al.* [11] propose a heuristic divide-and-conquer algorithm to address the dynamic distribution problem for data parallel programs. The "divide" pass of the algorithm recursively subdivides the program into regions, within which distributions are assigned independently. In the "conquer" pass, regions are merged if the redistribution cost between them exceeds the disadvantage of choosing a sub-optimal distribution for one of them.

Redistribution is a very complex communication, especially if the blocksize of a block-cyclic distribution is changed. Accordingly, Chatterjee *et al.* use the discrete metric (see Section 3) as the communication cost function in [11].

Sheffler *et al.* [25] show how the size of the distribution problem may be substantially reduced by *graph contraction*.

6   Related Work

We now briefly review the relationship between the work discussed in this paper and two somewhat different approaches to solving a similar problem: automatic data placement in automatic parallelisation and interprocedural analysis of user programs.
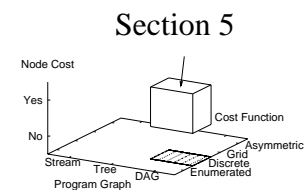
6.1   Automatic Data Placement in Automatic Parallelisation

The aim of automatic parallelisation is to translate an existing sequential program into an equivalent parallel program. Most early work in this field was done in the field of automatic vectorisation [3]. However, the emphasis in supercomputer architecture has shifted somewhat from vector processors to distributed memory multiprocessors, for a number of reasons, as outlined in [16]. The key challenges in automatic parallelisation have accordingly also shifted. Distributed memory machines have a high penalty for non-local memory access. Accordingly, data placement optimisation has become one of the key tasks an automatic parallelisation system has to address.

Survey paper on automatic parallelisation are [6] by Banerjee *et al.* and [5] by Bacon *et al.*. A survey which focuses more closely on the automatic data placement techniques in such systems is [4] by Ayguadé *et al.*.

There are a number of reasons why solving the parallel data placement problem from an automatic parallelisation approach is harder than from a component-composition approach.

Source code analysis. Automatic parallelisation is facilitated by a number of program analysis techniques. In fact, the success of automatic parallelisation systems often depends on the degree of sophistication of the analysis techniques used. In the problem domain which we discuss in this paper, the difficulties and limitations of source code analysis are avoided because our starting point are data-parallel *components* for which the relevant information is either supplied by the user (in the case of hand-written components) or can be derived with much greater ease from the source code because the code is written in a programming language that is designed to expose such information.

Source code transformation. Automatic parallelisation generally relies on restructuring user code in order to extract parallelism or maximise locality. It is clear that a conservative approach needs to be taken here to ensure that program semantics are not changed. In the problem domain which we discuss here, the idea is that the external characterisation of a component (more precisely, the metadata describing different available implementations) describes a set of possible different behaviours which can be selected from by the optimiser. Note, however, that *legality tests* might still need to be performed.

Global Optimisation vs. Component Composition. In our approach, we optimise the composition of components that have been optimised in isolation. In contrast, an automatic paralleliser would in general analyse and attempt to optimise the same program on one level. Although it is possible that some optimisation opportunities are missed by optimising components separately, it is also clear that not doing so will result in re-optimising smaller units of code every time they occur in a different context. For some purposes it is possible to fully characterise the behaviour of a component through some metadata such that no opportunities are lost by not re-optimising, see Section 2.3.

6.2   Interprocedural Analyses

An alternative approach to solving a very similar problem to the one we have addressed in this paper is outlined by Creusillet and Irgoin [15]. For the case where the components whose composition we are optimising are subroutines in a language such as Fortran, Creusillet and Irigoin show how to analyse such programs interprocedurally, such that the normal techniques of automatic parallelisation are not blocked by procedure boundaries.

It is clear that this approach may result in re-optimising components, which, under the component-composition approach, would have been optimised in isolation and not re-optimised at composition time. The question is whether component-composition stands to lose optimisation opportunities. The key to not loosing optimisation opportunities has to be that enough information is carried over into the component interface (metadata) when each component (procedure, loop nest etc.) is optimised. Carrying only the information about the best possible solution is insufficient. However, if the space of feasible solutions is bounded, representing those as part of component metadata means that we can avoid having to re-optimise components without losing any optimisation opportunities.

7 Summary and Conclusions

We have outlined current approaches to solving the parallel data placement problem when optimising the composition of data-parallel components.

– The parallel data placement problem has been solved for those cases where the set of possible placements can be enumerated. The algorithms are linear in program size but polynomial in the number of different placements.
– Heuristic algorithms exist for solving the problem for axis, stride and offset alignment that avoid the problem of large or possibly infinite sets of possible placements.
– We have outlined two alternative proposals for optimising replication. No practical comparisons on their effectiveness have been carried out.
– There are a number of communication patterns (e.g. skewings) that have not been covered by the algorithms we have outlined in this paper. In order to find an algorithm that allows optimising general alignment, including skewings, a cost model has to be found that correctly compares the different types of alignment costs. Once that is available, its structure can be studied with a view to finding a suitable algorithm.
– All DAG algorithms, except for the case of enumerated placements, are heuristic. We are not aware of much practical experience with the effectiveness of these heuristics.

7.1 Relationship with Constructive Methods for Parallel Programming

Our discussion has highlighted a number of key issues that affect the feasibility of applying parallel data placement optimisation in practice.

Component Metadata. We argued in Section 6.2 that it is very important that we do not have to re-optimise components when seeking to optimise their composition. In order for that to be possible without losing optimisation opportunities, the metadata which describe the properties of our components have to be designed very carefully. We outlined one of the issues, the compact representation for different implementations of a component, in Section 2.3.

Enumerated vs. Compact Placements The algorithms for enumerated placements are attractive because of their clarity and precise nature. Unfortunately, they also have high complexity in the number of different possible placements. Enumerating possible placements also has some advantages, especially in a context where programs are re-written symbolically. However, if a more more compact, non-symbolic representation for placements does exist (such as using $\left(\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right)$, $\left(\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right)$, ... instead of names such as identity, transpose, ... this may actually allow us to use a more compact algorithm, such as those described in Section 3. The

drawback of symbolic names for placements is that they might hide relationships between them that the optimiser might exploit.

Nature of the Program Graph. It is clear that the nature of the program graph, specifically, whether it is a tree or a DAG, has a very substantial impact on the complexity of the optimisation problem to be solved. If a graph is in fact a DAG, applying a tree algorithm may have unpredictable effects.

The question may be asked whether using BSP as a programming model ensures that the program graphs we obtain are chains (unary trees). Unfortunately, this is not the case. BSP describes and constrains the parallel scheduling of components (sequential with synchronisation), it does not restrict the *data flow* of a program. As long as the BSP program contains operators (or supersteps) that require more than one input, we cannot be sure that the data flow graph is not a tree or a DAG.

However, *skeletons* do have the potential to provide us with the type of information about the data flow graph of a program which is much harder to obtain in a less constrained programming environment. Skeletons might give us the information that a component either only has one operand, or, they might directly constrain the data flow (such as in a pipe). It seems advantageous to make use of this information in determining whether a tree algorithm can be used.

References

[1] A. V. Aho and S. C. Johnson. Optimal code generation for expression trees. *Journal of the ACM*, 23(3):488–501, July 1976.

[2] A. V. Aho, S. C. Johnson, and J. D. Ullman. Code generation for expressions with common subexpressions. *Journal of the ACM*, 24(1), Jan. 1977.

[3] R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, Oct. 1987.

[4] E. Ayguadé, J. Garcia, and U. Kremer. Tools and techniques for automatic data layout: A case study. *Parallel Computing*, 24(3–4):557–578, May 1998.

[5] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, Dec. 1994.

[6] U. Banerjee, R. Eigenmann, A. Nicolau, and D. A. Padua. Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211–243, Feb. 1993.

[7] O. Beckmann and P. H. J. Kelly. Efficient interprocedural data placement optimisation in a parallel library. In D. O'Hallaron, editor, *LCR98: Fourth International Workshop on Languages, Compilers and Run-time Systems for Scalable Computers*, volume 1511 of *LNCS*, pages 123–138. Springer-Verlag, May 1998.

[8] O. Beckmann and P. H. J. Kelly. A linear algebra formulation for optimising replication in data parallel programs. In J. Ferrante and L. Carter, editors, *LCPC99: Twelfth International Workshop on Languages and Compilers for Parallel Computing*, LNCS, Aug. 1999. To appear.

[9] S. Chatterjee, J. R. Gilbert, L. Oliker, R. Schreiber, and T. J. Sheffler. Algorithms for automatic alignment of arrays. *Journal of Parallel and Distributed Computing*, 38:145–157, 1996.

[10] S. Chatterjee, J. R. Gilbert, and R. Schreiber. Mobile and replicated alignment of arrays in data-parallel programs. In *Proceedings of Supercomputing '93*, pages 420–429, Nov. 1993.

[11] S. Chatterjee, J. R. Gilbert, R. Schreiber, and T. J. Sheffler. Array distribution in data-parallel programs. In *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, volume 892 of *LNCS*, pages 76–91, Ithaca, New York, Aug. 8–10, 1994. Springer-Verlag.

[12] S. Chatterjee, J. R. Gilbert, R. Schreiber, and S.-H. Teng. Optimal evaluation of array expressions on massively parallel machines. In *Proceedings of the Second Workshop on Languages, Compilers and Run-Time Environments for Distributed Memory Multiprocessors*, Oct. 1992. Published in SIGPLAN Notices 28(1):68–71, January 1993.

[13] S. Chatterjee, J. R. Gilbert, R. Schreiber, and S.-H. Teng. Automatic array alignment in data-parallel programs. In *Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, January 10–13, 1992*, pages 16–28. ACM Press, 1993.

[14] S. Chatterjee, J. R. Gilbert, R. Schreiber, and S.-H. Teng. Optimal evaluation of array expressions on massively parallel machines. *ACM Transactions on Programming Languages and Systems*, 17(1):123–156, Jan. 1995.

[15] B. Creusillet and F. Irigoin. Interprocedural analyses of Fortran programs. *Parallel Computing*, 24(3–4):629–648, May 1998.

[16] P. Feautrier. Toward automatic distribution. *Parallel Processing Letters*, 4(3):233–244, 1994.

[17] J. R. Gilbert and R. Schreiber. Optimal expression evaluation for data parallel architectures. *Journal of Parallel and Distributed Computing*, 13(1):58–64, Sept. 1991.

[18] High Performance Fortran Forum. High Performance Fortran language specification, version 1.1. TR CRPC-TR92225, Center for Research on Parallel Computation, Rice University, Houston, TX, Nov. 1994.

[19] *ICS '94, International Conference on Supercomputing 1994 (Manchester, U.K.)*. ACM Press, July 1994.

[20] D. E. Knuth. Big Omikron and Big Omega and Big Theta. *SIGACT News*, pages 18–24, Apr.–June 1976.

[21] U. Kremer. NP-Completeness of dynamic remapping. In *Proceedings of the Fourth Workshop on Compilers for Parallel Computers*, Delft, The Netherlands, 1993. Also available as CRPC-TR93330-S.

[22] M. E. Mace. *Storage Patterns in Parallel Processing*. Kluwer Academic Press, 1987.

[23] M. E. Mace and R. A. Wagner. Globally optimum selection of memory storage patterns. In *International Conference on Parallel Processing*, pages 264–271, Los Alamitos, Ca., USA, Aug. 1985. IEEE Computer Society Press.

[24] B. Prabhala and R. Sethi. Efficient computation of expressions with common subexpressions. *Journal of the ACM*, 27(1):146–163, Jan. 1980.

[25] T. J. Sheffler, R. Schreiber, W. Pugh, J. R. Gilbert, and S. Chatterjee. Efficient distribution analysis via graph contraction. *International Journal of Parallel Programming*, 24(6):599–620, Dec. 1996.

[26] D. B. Skillicorn, M. Danelutto, S. Pelagatti, and A. Zavanella. Optimising data-parallel programs using the BSP cost model. In D. Pritchard and J. Reeve, editors, *Euro-Par'98 Parallel Processing, Proceedings of the Fourth International Euro-Par Conference*, volume 1470 of *LNCS*, pages 698–703, Southampton, UK, Sept. 1998. Springer-Verlag.

[27] H. W. To. *Optimising the Parallel Behaviour of Combinations of Program Components*. PhD thesis, Department of Computing, Imperial College, London, U.K., Sept. 1995.

[28] L. G. Valiant. A scheme for fast parallel communication. *SIAM Journal on Computing*, 11(2):350–361, May 1982.

[29] D. Wilde and S. Rajopadhye. Memory reuse analysis in the polyhedral model. *Parallel Processing Letters*, 7(2):203–215, June 1997.