

A REVIEW OF DATA PLACEMENT OPTIMISATION FOR DATA-PARALLEL COMPONENT COMPOSITION*

OLAV BECKMANN AND PAUL H J KELLY

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 use a uniform terminology, which identifies analyses of the problem that have proven fruitful.
- 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 applied in practice, and point out areas for future research.

1.2. Terminology. Mace [26] 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, which results in a redistribution cost described by a cost function.

- *Data-Parallel Operations*

Data-parallel operations are elementwise array operations, such as provided by Fortran 90. In the context of our own work [9], array operations are calls

* *Authors' Affiliation:* Department of Computing, Imperial College, 180 Queen's Gate, London SW7 2BZ, United Kingdom. *Email:* {ob3,phjk}@doc.ic.ac.uk. This work was partially supported by a United Kingdom Engineering and Physical Sciences Research Council (EPSRC) Studentship for Olav Beckmann.

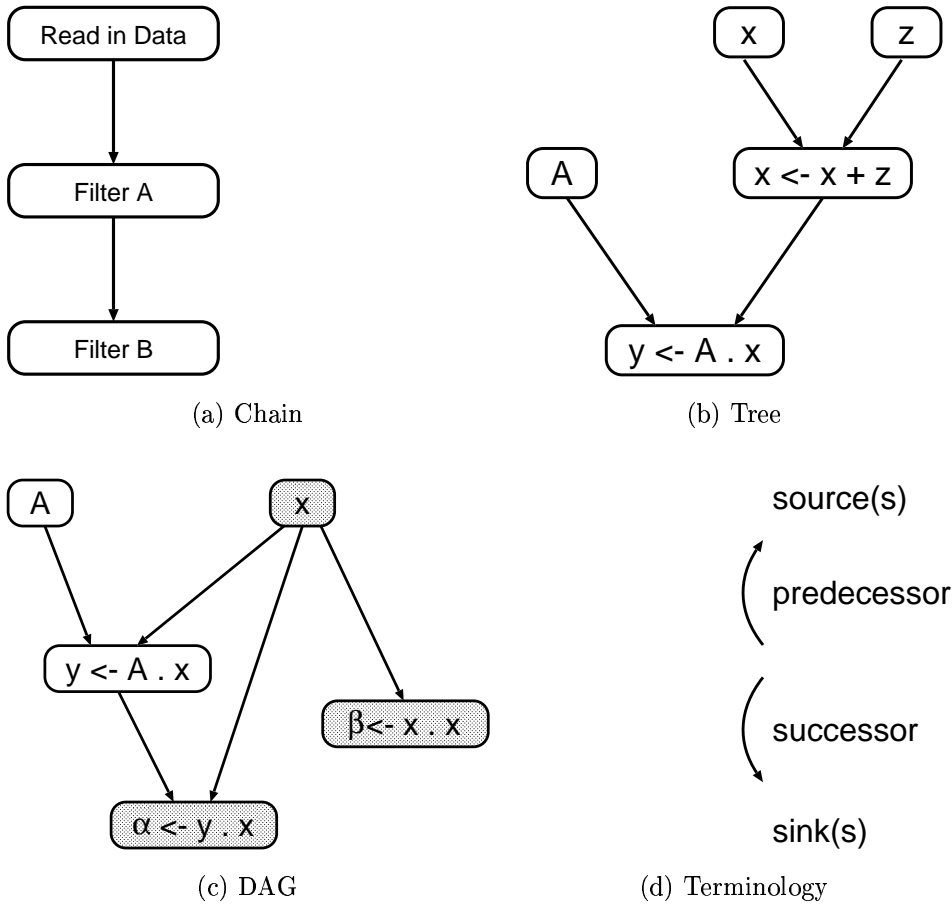


FIG. 1.1. Three classes of program graph. The DAG is distinguished by the presence of a shared node and by the fact that there may be more than one sink. Next to the DAG, in part (d), we also show the terms we will use to refer to different nodes in program graphs and to their relationships in this discussion.

to a 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 [22], or, according to our own proposal [10], as an explicit third stage preceding alignment.

Mace [26] calls parallel data placements *shapes*. We will therefore use S to denote the set of all possible placements and s to denote the size of 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 2.5.

Mace [26] 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*

We will use the term *program graph* for the data flow graph 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 2.2, we explain why the cost functions for modeling redistribution cost are generally required to be *metrics*.

Note that the term generally used for cost functions in the skeletons literature is *performance models*.

1.3. Scope of this Review. 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.* [16]. 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.

1.4. Overview of the Parallel Data Placement Problem. We argue that the complexity of the parallel data placement problem is determined by three key parameters. These are illustrated in Figure 1.2 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 1.1. DAGs are substantially harder to optimise than trees. We explain why that is so in Section 2.6.

2. *The cost function used as the objective when optimising*

¹This corresponds to the direction in which data dependence arrows are drawn in a dependence graph. The edges in Mace's program graphs [26] have the opposite orientation.

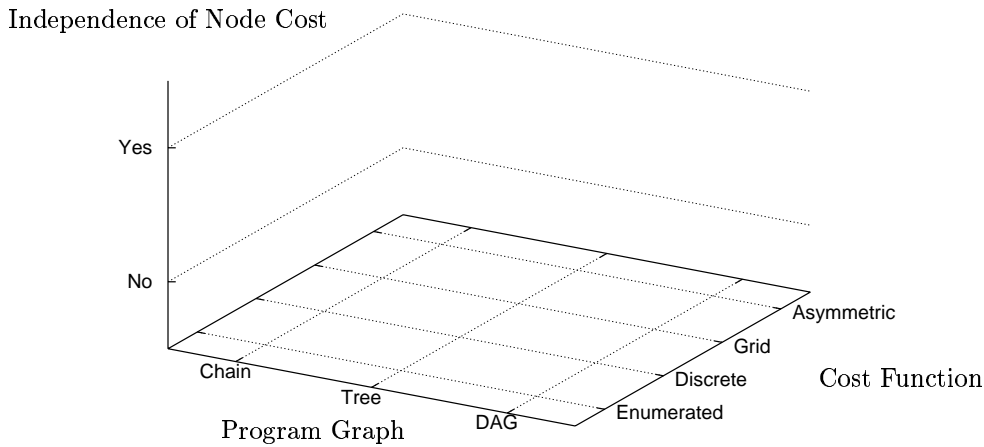


FIG. 1.2. *Illustration of the problem domain of parallel data placement optimisation in data-parallel programs.*

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 modeled 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 1.1 will be addressed in subsequent sections of this 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 1.2 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 is the easiest to understand and exposes the complexity of the problem well. However, the solution here 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 review to other research activities, and Section 7 concludes the paper.

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 $\mathcal{S} = \{S_1, \dots, S_s\}$ 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 :

$$C_{\text{Red}}[S_{\text{from}}][S_{\text{to}}], \quad S_{\text{from}}, S_{\text{to}} \in \mathcal{S} . \quad (2.1)$$

In Sections 2.2 and 2.3 we comment in more detail on the nature and properties of these cost functions, and we discuss the question of how the cost functions are obtained in Section 7.2.

2.1. Dynamic Programming Solution for

Chains. We begin with the simplest class of program graph, chains. The algorithm described below is based on the algorithm proposed by Mace [26,27] for trees (see Sections 2.4 and 2.5). 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 $\text{op}.S_{\text{in}}$, the result placement by $\text{op}.S_{\text{out}}$ and the cost of computing op with these placements by $\text{op}.C_{\text{Node}}$. Finally, we will denote the operation to be performed at some node x by $x.\text{op}$ and the predecessor of node x by $x.\text{pred}$.

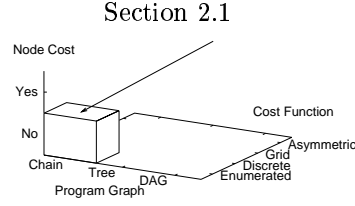
In addition to assuming that the program graph is a chain, we will initially also assume that each operation has only one implementation, which for now we take to mean that there is *one* specific pair $(S_{\text{in}}, S_{\text{out}})$ of input and result placements. We address the issue of multiple implementations in Section 2.5.

2.1.1. Algorithm: Solution for Chains, One Implementation. We can now give the following recursive formulation for the minimum cost $C_{\text{min}}(x, S_i)$ of evaluating a chain of data-parallel operations with sink x into placement S_i .

$$\begin{aligned} C_{\text{min}}(x, S_i) &= C_{\text{this}}(x, S_i) && \text{if } x \text{ is the leaf} && (2.2) \\ &= C_{\text{this}}(x, S_i) + C_{\text{min}}(x.\text{pred}, x.\text{op}.S_{\text{in}}) && \text{otherwise} && (2.3) \end{aligned}$$

where

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



This recursive algorithm to find $C_{\min}(x, S_i)$ takes $\Theta(n)$ steps² for a chain of n nodes. The overall minimum cost for calculating node x is now theoretically

$$C_{\min}(x) = \min_{S_i \in \mathcal{S}} (C_{\min}(x, S_i)) , \quad (2.4)$$

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

$$C_{\min}(x) = C_{\min}(x, x.\text{op}.S_{\text{out}}) , \quad (2.5)$$

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

2.2. Metric Properties of Redistribution Cost. Chatterjee *et al.* [15–17] and Gilbert and Schreiber [20] state that all their cost functions for redistributions, which they call *distance functions*, ought to be metrics. A distance function $d : \mathcal{S} \times \mathcal{S} \rightarrow \mathbb{R}$ is a metric if the following conditions hold for all $S_i, S_j, S_k \in \mathcal{S}$.

$$d(S_i, S_j) \geq 0 \quad \text{and} \quad d(S_i, S_j) = 0 \Leftrightarrow S_i = S_j \quad \text{Nonnegativity} \quad (2.6)$$

$$d(S_i, S_j) = d(S_j, S_i) \quad \text{Symmetry} \quad (2.7)$$

$$d(S_i, S_j) \leq d(S_i, S_k) + d(S_k, S_j) \quad \text{Triangle inequality} \quad (2.8)$$

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 (2.5), rather than (2.4) in Algorithm 2.1.1.

Triangle Inequality. Real systems do not necessarily satisfy the triangle inequality. Two-phase random routing [35], 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.

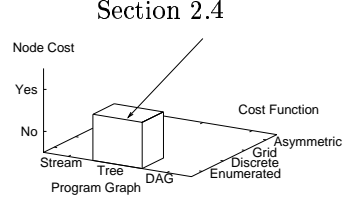
Symmetry. It appears that the parallel data placement algorithms proposed by Chatterjee *et al.* do not actually rely on the symmetry property of the distance function. There are some interesting models of redistribution cost that are not symmetric. Replication is one such example, which we discuss further in Section 4. Another example are some instances of the BSP cost model [11].

2.3. 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.* [15, 17] 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 [17].

In this review, 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 [26] and Gilbert and Schreiber [20].

²We will use the complexity notations proposed by Knuth [24].

2.4. 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 i^{th} predecessor of a node x by $x.\text{pred}[i]$, and the placement in which the operator at node x requires its i^{th} operand by $x.\text{op}.S_{\text{in}}[i]$.



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

$$C_{\min}(x, S_i) = C_{\text{this}}(x, S_i) \quad \text{if } x \text{ is a leaf} \quad (2.9)$$

$$= C_{\text{this}}(x, S_i) + \sum_{0 \leq j < a} C_{\min}(x.\text{pred}[j], x.\text{op}.S_{\text{in}}[j]) \quad \text{otherwise} \quad (2.10)$$

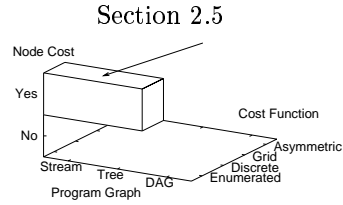
where

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

If C_{Red} is a metric, the overall minimum cost is given by Equation (2.5), as in the case for chains. The time complexity of this algorithm is $\Theta(n)$.

2.4.2. Remark: 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.

2.5. 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.



2.5.1. Definition: Implementations as Tuples (Mace). Mace [26] characterises an implementation³ 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_3 , op1 is said to have two implementations. We will denote the required input data placement of implementation M_k of op by $\text{op}.M_k.S_{\text{in}}$, the result placement by $\text{op}.M_k.S_{\text{out}}$ and the cost of computing op with these placements by $\text{op}.M_k.C_{\text{Node}}$. 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.

³Mace uses the term *method*.

2.5.2. Algorithm: Solution for Trees, Implementations as Tuples. We have:

$$C_{\min}(x, S_i) = \min_{M_k \in \mathcal{M}} (C_{\text{this}}(x, M_k, S_i)) \quad (2.11)$$

if x is a leaf

$$= \min_{M_k \in \mathcal{M}} \left(C_{\text{this}}(x, M_k, S_i) + \sum_{0 \leq j \leq a} C_{\min}(x.\text{pred}[j], x.\text{op}.M_k.S_{\text{in}}[j]) \right) \quad (2.12)$$

otherwise,

where

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

The complexity of calculating $C_{\min}(x, S_i)$ 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:

$$C_{\min}(x) = \min_{S_i \in \mathcal{S}} (C_{\min}(x, S_i)) . \quad (2.13)$$

The complexity of Algorithm 2.5.2 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 (2.12) and (2.11). 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 in Equation (2.13). Therefore, the complexity is $O(msn)$ rather than $\Theta(msn)$.

2.5.3. Remarks. 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.* $O(s^3n)$ 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 [26,27]. Versions of the algorithm for chains have been proposed by To [33, page 223] and Skillicorn *et al.* [32].

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.

2.5.4. Definition: Implementations as Placement Relations. Assume 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 $f(\vec{i}) = A.\vec{i} + \vec{t}$ from array index vectors \vec{i} onto virtual processor

indices [9]. Given invertibility, we may always, for any pair of defined placement functions f and g , calculate a *relationship function* $r = g \circ f^{-1}$ with the property that $g = r \circ f$.

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

2.5.5. 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 $r_1, r_2 = \text{id}$, where id is the identity affine function which maps every placement to itself.

2.5.6. Algorithm: 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

$$C_{\min}(x, S_i) = C_{\text{this}}(x, M_1, x.\text{op}.M_1.S_{\text{out}}, S_i) \quad (2.14)$$

x is a leaf

$$= \min_{\substack{S_j \in \mathcal{S} \\ M_k \in \mathcal{M}}} \left(C_{\text{this}}(x, M_k, S_j, S_i) + \sum_{0 \leq l \leq a} C_{\min}(x.\text{pred}[k], x.\text{op}.M_k.r_l \circ S_j) \right) \quad (2.15)$$

otherwise,

where

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

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 (2.13), resulting in an overall complexity of $O(ms^2n)$.

2.5.7. Remarks. 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. 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. 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 described in this section provides a solution for the large class of Fortran 90 array operations in $O(s^2n)$ time. This corresponds exactly to the algorithm outlined by Chatterjee *et al.* in [16]. The definition for implementations used in this section has been stated in our previous publications [9, 10].

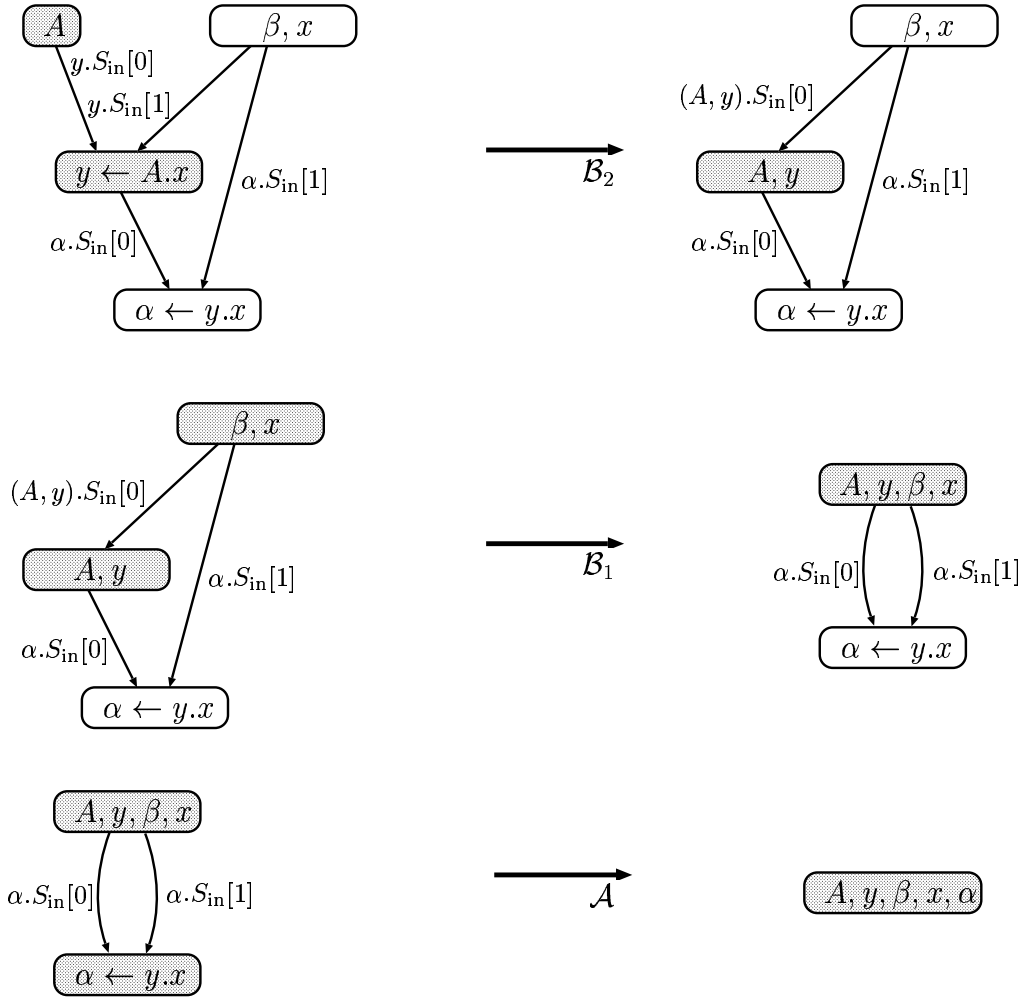
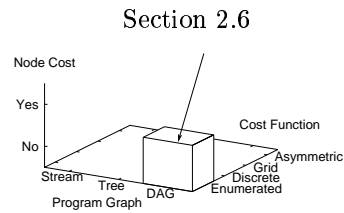


FIG. 2.1. Example of parallel data placement optimisation for collapsible DAGs: A collapsible DAG is reduced to a single node through a series of three transformations. The transformations are labelled with the transformation names used by Mace (A , B_1 , B_2).

2.6. Solution for DAGs. Mace [26] shows the parallel data placement problem to be NP-complete when the program graph is a general DAG. The following observations provide an intuition as to why the problem is so much harder to solve for DAGs than for trees:

- The contributions towards the cost of a node that come from calculating the operands in the required placement (Equation 2.10) 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 (2.13).

However, Mace also demonstrates that a polynomial algorithm may be found for a subclass of DAGs known as *collapsible DAGs*, and Mace argues [26, page 10] that a very significant proportion of real program DAGs fall into this subclass.

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 [29], who showed that the optimal code generation problem may be solved in polynomial time for this subclass of DAGs. A program graph is collapsible if it is series-parallel with tree-like subgraphs. This means in particular that graphs with more than one sink are *not* collapsible. Linear-time algorithms exist for detecting whether a graph is series-parallel [34].

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 the collapsible graph algorithm in Figure 2.1. 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. Let d denote the *degree* of a node, *i.e.* the number of edges leaving the node, and a the arity of the operator at a node. Then, the cost of a general node i is a function $\text{Cost}(i.\text{op}, i.S_{\text{out}}[0], \dots, i.S_{\text{out}}[d], i.S_{\text{in}}[0], \dots, i.S_{\text{in}}[a])$ of the operator at node i and the data placements assigned to the edges which touch it. In this model, redistribution costs are counted as part of the cost of the source node of an edge; this means that the cost of calculating node i with different output placements $i.S_{\text{out}}[0]$ and $i.S_{\text{out}}[1]$ may *include* the cost of a redistribution. Placements at the source and sink of edges are then required to be identical.

The cost for the combined node (β, x) in Figure 2.1 is

$$\text{Cost}((A, y).\text{op}, (A, y).S_{\text{in}}[0], \alpha.S_{\text{in}}[0]) = \min_{S_j \in S} \left(\begin{array}{l} \text{Cost}(y.\text{op}, S_j, y.S_{\text{in}}[1], \alpha.S_{\text{in}}[0]) \\ + \text{Cost}(A.\text{op}, S_j) \end{array} \right) \quad (2.16)$$

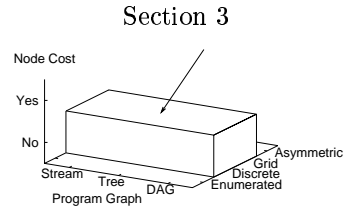
2.6.1. Remarks. The algorithm has complexity $O(s^{d+1}(n-1))$ where d is the out-degree of the most highly-shared node. This complexity does not yet include any graph dilation which might arise because some nodes originally have in- and out-degree of 1 or more. The problem of finding the smallest cost for generating the result of a shared node in a number of different placements is a Steiner problem [16, 37].

2.7. Conclusion for Enumerated Cost Functions. The defining features of the algorithms for explicitly enumerated cost functions are that they do not use the structure of cost functions to direct the search and that the set of possible placements must be small for the algorithms to be feasible.

The advantage of these algorithms 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. Gilbert and Schreiber [20] and Chatterjee *et al.* [16, 17] 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.



3.1. 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 and a cost of 0 otherwise. 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 predecessors, 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 makes use of this intuitive property: For each node, a set of *level sets* is defined. Each level set contains the placement solutions which result in the same cost for that node. The cheapest solution for each node in a tree is then the intersection of the cheapest level sets of the node’s parents, or, if that intersection is empty, the union. For a tree of height h , this algorithm solves the same problem for which we described an $O(s^2n)$ solution in Section 2.5.4 in only $O(nh)$ time under the discrete metric.

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.

3.2. Grid metric for offset alignment. For offset alignment, the discrete metric is clearly unsatisfactory, since it would assign the same cost to any offset or shift of data, independent of the size of the offset. Although the distance by which an object is moved is not very important in most modern networks, the size of affine offset realignment is still significant, because the folding of several virtual processors onto one physical processor during distribution means that distance of movement on the virtual processor grid will often correspond to actual amount of data movement after distribution.

Chatterjee *et al.* propose to use a *grid* metric for offset alignment: the cost of a redistribution is modeled 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 2.5.4 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.

3.3. Heuristic Solution for DAGs. The above algorithms work for trees. In [12, 16], 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.

3.4. Discussion. The aim of Section 3 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.

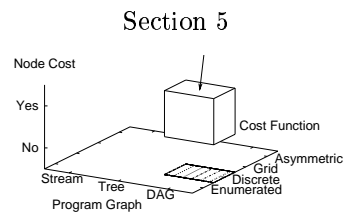
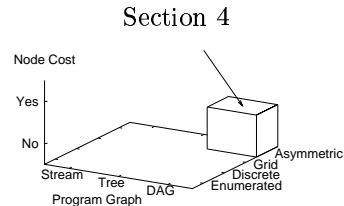
4. Solution for Replication. Replication is a difficult problem in terms of data placement optimisation because the natural representation for replication is not even a function: one location is mapped onto many. Several authors [12,36] have therefore proposed representing the *inverse* of copying, *i.e.* 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 “down-stream” in the DAG from the broadcast, except for the results of reductions, have to be replicated as well. A further observation is that the natural replication cost function is not a metric because it is *asymmetric*.

Chatterjee, Gilbert and Schreiber [13] 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 [28] 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 [10] 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 outline a possible heuristic algorithm in [10].

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 [31]. 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 [25].

Chatterjee *et al.* [14] 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 3.1) as the communication cost function in [14].

Sheffler *et al.* [31] show how the size of the distribution problem may be substantially reduced by *graph contraction*.

6. Related Research Areas. We now briefly review the relationship between the work discussed in this paper and three somewhat different approaches to solving a similar problem: explicit parallel data placement, optimisation based on conventional interprocedural analyses and automatic data placement in automatic parallelisation.

6.1. Explicit Parallel Data Placement. A number of data parallel programming systems require the user to specify parallel data placement, *e.g.* HPF [22]. XDP [6] is a language which allows representing data ownership explicitly within a compiler intermediate language. The work we have discussed in this paper can be regarded as seeking to generate such data placement annotations automatically.

6.2. Optimisation Based on Interprocedural Analyses. An alternative approach to solving the cross-component data placement optimisation problem is to perform interprocedural analyses on the program we are optimising and then apply parallel data placement techniques from the field of automatic parallelisation (see Section 6.3) on the entire program. Some of the analysis techniques that would be required for such a solution are outlined by Creusillet and Irigoien [18]. For the case where the components whose composition we are optimising are subroutines written in a language such as Fortran, Creusillet and Irigoien show how to analyse such programs interprocedurally, such that automatic parallelisation techniques are not blocked by procedure boundaries.

Creusillet and Irigoien do not discuss the application of their analyses to optimisation of parallel programs. From the components-perspective, the challenge is to re-use optimisation results on individual components in order to do less optimisation work than the conventional interprocedural approach, and at the same time not suffer performance losses because of missed optimisation opportunities.

6.3. 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 by Feautrier in [19]. The key challenges in automatic parallelisation have therefore 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 papers on automatic parallelisation are [8] 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. Examples of such techniques are dependence testing [3, 21], detection of “task parallelism” between loop nests [7] and privatisation tests [30]. 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 (hand-written) data-parallel *components* for which the relevant information is supplied by the user.

- *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 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 (the metadata) describes a set of possible different implementations which can be selected from by the optimiser without having to perform the transformation. 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 may be possible to characterise fully the behaviour of a component through metadata such that no opportunities are lost by not re-optimising. We return to this issue in our conclusion, Section 7.

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

7.1. Solved and Unsolved Problems. We begin our conclusions with a summary of solved and unsolved problems.

- The parallel data placement problem has been solved for those cases where the set of possible placements can be enumerated, and where the program graph is either a tree or a collapsible DAG. The algorithms (Algorithm 2.1.1, 2.4.1, 2.5.2, 2.5.6 and Section 2.6) are linear in program size but polynomial in the number of different placements.
- Accurate dynamic programming algorithms exist for solving the problem for axis, stride and offset alignment in trees. These avoid the obstacle of large or

infinite sets of possible placements by exploiting structure in the cost functions for these metrics (Sections 3.1 and 3.2). Section 3.3 mentions a heuristic proposal for DAGs.

- We have outlined two alternative proposals for optimising replication in Section 4, one of them being our own proposal, presented in [10]. Neither approach is always optimal; no experimental comparisons on their effectiveness have yet 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 collapsible DAGs with enumerated placements, are heuristic. We are not aware of much practical experience with the effectiveness of these heuristics.

7.2. Component Metadata. We have argued that carefully constructed component metadata play a crucial role in efficient cross-component data placement optimisation for data-parallel programs.

- *Obtaining the Metadata*

Thus far we have not addressed the question of how the metadata that are used in optimisation, such as the description of available implementations, are obtained. In the context of our own work [9], where data-parallel operations are functions from a library of parallel numerical routines, the metadata have been provided by the library implementor. However, this information could also be computed by a compiler for a data-parallel language, or even by an automatic paralleliser.

- *Avoiding Re-optimisation of Components*

It is very important that we do not have to re-optimize components at runtime 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.

The question is whether component-composition stands to lose optimisation opportunities. The key to not losing optimisation opportunities has to be that enough information is carried over into the component metadata when each component is optimised. Carrying only information about one good or optimal solution is insufficient; instead, we have to preserve the feasible space of implementations from the component's compilation time, such that a cross-component optimiser can later select any one of the feasible implementations. We believe that representing the set of possible implementations as placement relations (see Definition 2.5.4) is one example: we have a compact representation for the feasible space of placements for any binary Fortran 90 array operation.

- *Symbolic 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 optimised by symbolic transformation. However, if a more compact, non-

symbolic representation for placements does exist (such as using $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$, $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$ instead of *identity*, *transpose*) this may actually allow us to use more compact algorithms, such as those described in Section 3.

Optimisation based on symbolic transformation does not expose the underlying mathematical optimisation problem being solved. The drawback of this is that symbolic names for placements might *hide* relationships that the optimiser could exploit.

7.3. 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.

BSP. 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.

Skeletons. However, there are situations where we may have information about the structure of the program graph: Skeletons for example might give us the information that components either only have 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.

7.4. Cost Functions. In this paper, we have so far not addressed the question of how the cost functions are obtained. In the case of enumerated placements, it seems likely that the cost table has to be obtained by calibration. Given that the optimisation algorithms for enumerated placements are only really feasible for small numbers of possible placements, this seems not unreasonable. Otherwise, such as in the case of compact metrics, the cost functions are chosen as models that are thought to be a reasonable representation for the particular placement being optimised.

Accuracy of Cost Function vs. Structure of Cost Function. When a cost function is chosen that is in some way an approximation to the actual cost (such as the discrete metric for axis alignment), an interesting trade-off arises: Choosing an approximate cost model may facilitate using a non-heuristic optimisation algorithm. Using a more accurate, calibrated cost representation may remove structure in the cost model in such a way that the only feasible algorithm is a heuristic one.

If we know a bound (such as 10%) on the maximum misprediction of our cost model, it seems preferable to use a simpler model with an accurate algorithm, since this means that we can carry the information about maximum misprediction forward onto the result of our optimisation.

7.5. Irregular Data. So far in this paper we have always assumed that data are regular arrays. This has meant that the size of all data placement metadata has been independent of the problem size. With irregular data, this may no longer be true. Tasks such as calculating the virtual processor which owns a particular array element, partitioning an array, or comparing two placements for equality would then have $O(N)$ complexity rather than $\Theta(1)$. For some data it may, however, be possible to use cyclic or block-cyclic distributions to adequately “mask” an underlying irregular structure without sacrificing fixed-size data placement descriptors.

7.6. Mixed Task- and Data-Parallelism. The set of problems that can be efficiently parallelised using a purely data-parallel programming model is restricted. A programming model where sets of data-parallel tasks can be composed in parallel is suited for a larger class of applications. Optimising such programs is much harder because data placement is no longer the primary determinant of parallel performance; instead, complex scheduling problems have to be addressed. However, the technology we have discussed in this paper is still of fundamental importance: In a mixed task- and data-parallel programming model, we have to solve the parallel data placement problem for each individual task before composing them in parallel.

Acknowledgements. We are grateful to the anonymous referees and to Susanna Pelagatti of the University of Pisa for helpful comments on the paper. Further, we are grateful to Julian Webster of Imperial College for providing insight into network flow problems.

REFERENCES

- [1] A. V. AHO AND S. C. JOHNSON, *Optimal code generation for expression trees*, Journal of the ACM, 23 (1976), pp. 488–501.
- [2] A. V. AHO, S. C. JOHNSON, AND J. D. ULLMAN, *Code generation for expressions with common subexpressions*, Journal of the ACM, 24 (1977).
- [3] R. ALLEN AND K. KENNEDY, *Automatic translation of FORTRAN programs to vector form*, ACM Transactions on Programming Languages and Systems, 9 (1987), pp. 491–542.
- [4] E. AYGUADÉ, J. GARCIA, AND U. KREMER, *Tools and techniques for automatic data layout: A case study*, Parallel Computing, 24 (1998), pp. 557–578.
- [5] D. F. BACON, S. L. GRAHAM, AND O. J. SHARP, *Compiler transformations for high-performance computing*, ACM Computing Surveys, 26 (1994), pp. 345–420.
- [6] V. BALA, J. FERRANTE, AND L. CARTER, *Explicit data placement (XDP): a methodology for explicit compile-time representation and optimization of data movement*, ACM SIGPLAN Notices, 28 (1993), pp. 139–148.
- [7] V. BALASUNDARAM AND K. KENNEDY, *A technique for summarizing data access and its use in parallelism enhancing transformations*, ACM SIGPLAN Notices, 24 (1989), pp. 41–53.
- [8] U. BANERJEE, R. EIGENMANN, A. NICOLAU, AND D. A. PADUA, *Automatic program parallelization*, Proceedings of the IEEE, 81 (1993), pp. 211–243.
- [9] O. BECKMANN AND P. H. J. KELLY, *Efficient interprocedural data placement optimisation in a parallel library*, in LCR98: Languages, Compilers and Run-time Systems for Scalable Computers, no. 1511 in LNCS, Springer-Verlag, May 1998, pp. 123–138.
- [10] ———, *A linear algebra formulation for optimising replication in data parallel programs*, in LCPC99: Languages and Compilers for Parallel Computing, no. 1863 in LNCS, Springer-Verlag, Aug. 1999, pp. 100–116.
- [11] M. BIANCO AND G. PUCCI, *On the predictive quality of BSP-like cost functions for NOWs*, in Proceedings of Euro-Par'2000: Parallel Processing, LNCS, Munich, Germany, 2000, Springer Verlag. To appear.
- [12] 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 (1996), pp. 145–157.
- [13] S. CHATTERJEE, J. R. GILBERT, AND R. SCHREIBER, *Mobile and replicated alignment of arrays in data-parallel programs*, in Proceedings of Supercomputing '93, Nov. 1993, pp. 420–429.
- [14] 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, vol. 892 of LNCS, Ithaca, New York, Aug. 8–10, 1994, Springer-Verlag, pp. 76–91.
- [15] 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.
- [16] S. CHATTERJEE, J. R. GILBERT, R. SCHREIBER, AND S.-H. TENG, *Automatic array alignment in data-parallel programs*, in POPL'93: ACM Symposium on Principles of Programming Languages, ACM Press, 1993, pp. 16–28.

- [17] ———, *Optimal evaluation of array expressions on massively parallel machines*, ACM Transactions on Programming Languages and Systems, 17 (1995), pp. 123–156.
- [18] B. CREUSILLET AND F. IRIGOIN, *Interprocedural analyses of Fortran programs*, Parallel Computing, 24 (1998), pp. 629–648.
- [19] P. FEAUTRIER, *Toward automatic distribution*, Parallel Processing Letters, 4 (1994), pp. 233–244.
- [20] J. R. GILBERT AND R. SCHREIBER, *Optimal expression evaluation for data parallel architectures*, Journal of Parallel and Distributed Computing, 13 (1991), pp. 58–64.
- [21] G. GOFF, K. KENNEDY, AND C.-W. TSENG, *Practical dependence testing*, ACM SIGPLAN Notices, 26 (1991), pp. 15–29.
- [22] 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.
- [23] *ICS '94, International Conference on Supercomputing 1994 (Manchester, U.K.)*, ACM Press, July 1994.
- [24] D. E. KNUTH, *Big Omikron and Big Omega and Big Theta*, SIGACT News, (1976), pp. 18–24.
- [25] 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.
- [26] M. E. MACE, *Storage Patterns in Parallel Processing*, Kluwer Academic Press, 1987.
- [27] M. E. MACE AND R. A. WAGNER, *Globally optimum selection of memory storage patterns*, in International Conference on Parallel Processing, Los Alamitos, Ca., USA, Aug. 1985, IEEE Computer Society Press, pp. 264–271.
- [28] C. H. PAPADIMTRIOU AND K. STEIGLITZ, *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, NJ, 1982.
- [29] B. PRABHALA AND R. SETHI, *Efficient computation of expressions with common subexpressions*, Journal of the ACM, 27 (1980), pp. 146–163.
- [30] L. RAUCHWERGER AND D. PADUA, *The privatizing DOALL test: A run-time technique for DOALL loop identification and array privatization*, in ICS '94 [23], pp. 33–43.
- [31] 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 (1996), pp. 599–620.
- [32] D. B. SKILLICORN, M. DANELUTTO, S. PELAGATTI, AND A. ZAVANELLA, *Optimising data-parallel programs using the BSP cost model*, in Euro-Par'98 Parallel Processing, Proceedings of the Fourth International Euro-Par Conference, D. Pritchard and J. Reeve, eds., vol. 1470 of LNCS, Southampton, UK, Sept. 1998, Springer-Verlag, pp. 698–703.
- [33] H. W. TO, *Optimising the Parallel Behaviour of Combinations of Program Components*, PhD thesis, Department of Computing, Imperial College, London, U.K., Sept. 1995.
- [34] J. VALDES, R. E. TARJAN, AND E. L. LAWLER, *The recognition of series parallel digraphs*, in Conference Record of the Eleventh Annual ACM Symposium on Theory of Computing, Atlanta, Georgia, 30 Apr.–2 May 1979, pp. 1–12.
- [35] L. G. VALIANT, *A scheme for fast parallel communication*, SIAM Journal on Computing, 11 (1982), pp. 350–361.
- [36] D. WILDE AND S. RAJOPADHYE, *Memory reuse analysis in the polyhedral model*, Parallel Processing Letters, 7 (1997), pp. 203–215.
- [37] P. WINTER, *Steiner problem in networks: A survey*, Networks, 17 (1987), pp. 129–167.