# A Linear Algebra Formulation for Optimising Replication in Data Parallel Programs

*Olav Beckmann and Paul H J Kelly*⋆

Department of Computing, Imperial College,
180 Queen's Gate, London SW7 2BZ, U.K.
{ob3,phjk}@doc.ic.ac.uk

**Abstract.** In this paper, we present an efficient technique for optimising data replication under the data parallel programming model. We propose a precise mathematical representation for data replication which allows handling replication as an explicit, separate stage in the parallel data placement problem. This representation takes the form of an invertible mapping. We argue that this property is key to making data replication amenable to good mathematical optimisation algorithms. We further outline an algorithm for optimising data replication, based on this representation, which performs interprocedural data placement optimisation over a sequence of loop nests. We have implemented the algorithm and show performance figures.

## 1 Introduction

Choosing parallel data placements which minimise communication is key to generating efficient data parallel programs. Under the data parallel programming model, parallel data placement is typically represented by a two-stage mapping. In the first stage, an affine alignment function maps array elements onto virtual processors. In the second stage, a distribution function then maps virtual processors onto physical ones. Examples for such a two-stage approach are listed in [5]. This decomposition fails to account properly for data replication: rather than using an explicit replication stage, replication is often handled implicitly as part of the alignment stage through *replicated alignments*. In this paper, we propose an efficient mathematical representation for data replication which expresses replication as an independent third stage of parallel data placement.

While a good range of optimisation techniques has been described for the alignment stage, distribution and replication have received less attention. We argue that the representation which we propose in this paper is a step towards making data replication amenable to the same type of mathematical optimisation algorithms which have previously been used for alignment optimisation. We demonstrate this assertion by describing and implementing an algorithm which performs interprocedural data replication optimisation across a sequence of loop nests. The objective function which our optimisation algorithm attempts to minimise is communication volume.

The practical importance of maintaining multiple copies of certain data is evident:

⋆ While this work was carried out, Paul Kelly was a visiting research scientist at the Department of Computer Science and Engineering, University of California at San Diego, USA.

- The programmer may specify a replicated placement, e.g. through `spread` operations [5], replicated alignments in HPF [8] or flooding operators in ZPL [12].
- If a read-only array is accessed by several processors in a particular subroutine or loop, it is more efficient to use a broadcast operation, generally $O(\log P)$, to replicate that array than to let processors access the array by remote reads, which would most likely be $\Theta(P)$, unless a special scheme, such as proxies [13] is used.
- Arrays may be replicated in order to facilitate parallelisation of a loop nest which would otherwise be blocked by anti- or output-dependencies on those arrays. This is a highly effective technique, known as *array privatisation* [11].
- In certain circumstances, runtime re-alignments of arrays can be avoided by replication. Specifically, mobile offset alignments can be realised through replication [4].

**Default strategies.** Implementations have typically made the assumption that scalars are replicated everywhere (e.g. HPF [8]), or sometimes more generally, that when mapped onto a higher-dimensional processor grid, lower-dimensional arrays are replicated in those dimensions where their extent is 1 (e.g. our own work [2]). There are other possible default strategies for choosing which arrays and scalars to replicate. However, while such a uniform default layout might conceivably be optimal in some circumstances, it is commonly much more efficient (as we show in an example shortly) to choose whether and how to replicate each value according to context.

**Motivation.** The key performance benefits from optimising data replication arise from:

1. *Replacing All-Reduce Operations with Simple Reductions.*
   A reduction which leaves its result in replicated form on all participating processors is known as an all-reduce. On most platforms, all-reduce operations are significantly more expensive than simple reductions; their complexity generally is that of a simple reduction followed by a broadcast. This is illustrated in Figure 1. Kumar et al. [9] show that theoretically-better implementations do exist, but they require at least the connectivity of a hypercube with two-way communication links. We therefore frequently have the opportunity of a significant performance gain by making an optimal choice about whether the result of a reduction is replicated or not.
2. *Reducing the Cost of Residual Affine Communications.*
   In many programs, it is not possible to find a set of affine alignments which eliminate all redistributions. However, the cost of these residual affine communications will be less if the data being communicated is not replicated.

Concrete instances of these two optimisations are illustrated in the working example introduced at the end of this section.

**Background: DESO BLAS Library.** Although this work is applicable to compile-time optimisation, it has been developed and implemented in the context of our delayed evaluation, self-optimising (DESO) library [2] of parallel numerical routines (mainly level 1 and 2 BLAS [10]). This library uses delayed evaluation — more details will be given in Section 3 — to capture the calling program's data flow and then performs run-time interprocedural data placement optimisation. Finding algorithms efficient enough for use at run-time is a key motivation for our work.

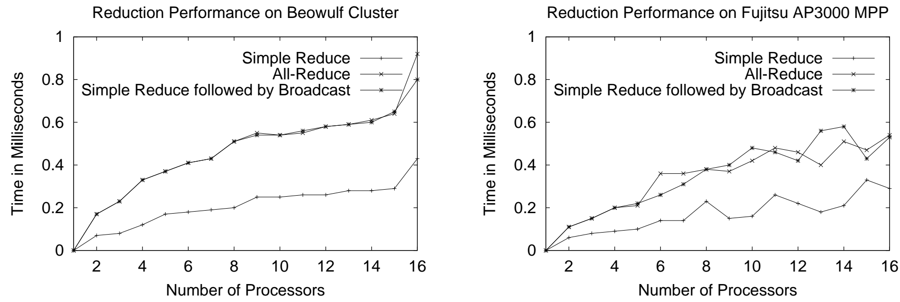


**Fig. 1.** Performance comparison of simple reduction and all-reduce (reduction over addition, 1 scalar per processor). *Left:* Cluster of 350MHz Pentium II workstations, running Linux 2.0.36 (TCP patched), connected by 100Mb/s ethernet and using `mpich-1.1.1`. *Right:* Fujitsu AP3000 MPP: Nodes are 300 MHz UltraSPARC, connected by 200 MB/s AP-Net. Averages of 100 runs; for both platforms, 5% of peak values were left out. In both cases, all-reduce takes roughly twice as long as reduce. Further, the figures illustrate that the performance of all-reduce is on these two platforms the same as that of a simple reduction followed by a broadcast.

**Contributions of this paper.** We propose a technique for optimising data replication:

– We describe a mathematical representation for replication which takes the form of an invertible mapping. We argue that this property is key to making data replication amenable to good mathematical optimisation algorithms.
– We describe an optimisation algorithm, based on this representation, which performs interprocedural data placement optimisation over a sequence of loop nests.
– We argue that our optimisation algorithm is efficient enough to be used in a runtime system.

Our optimisation algorithm works from aggregate loop nests which have been parallelised in isolation. We do not address any parallelism vs. replication trade-offs; we assume that decisions about which arrays have to be privatised for parallelisation have been taken separately.

The paper builds on related work in the field of optimising affine alignment, such as [5, 6]. Chatterjee et al. [4] provides a key point of reference for our work and we evaluate our work in comparison to their approach at the end of the paper in Section 5.

**Overview of this Paper.** After the final section of this introduction, which presents an example to illustrate the potential performance benefits of optimising replication, Section 2 describes our proposed representation for data replication. In Section 3, we describe an algorithm for interprocedural optimisation of data replication, which is based on the representation from the previous section. Section 4 discusses evaluation of our work using our DESO library of parallel numerical routines. Finally, Section 5 reviews related work and Section 6 concludes.

**Example: Conjugate Gradient.** Consider the sequence of operations from the first iteration of the conjugate gradient algorithm which is shown in Figure 2. By far the most

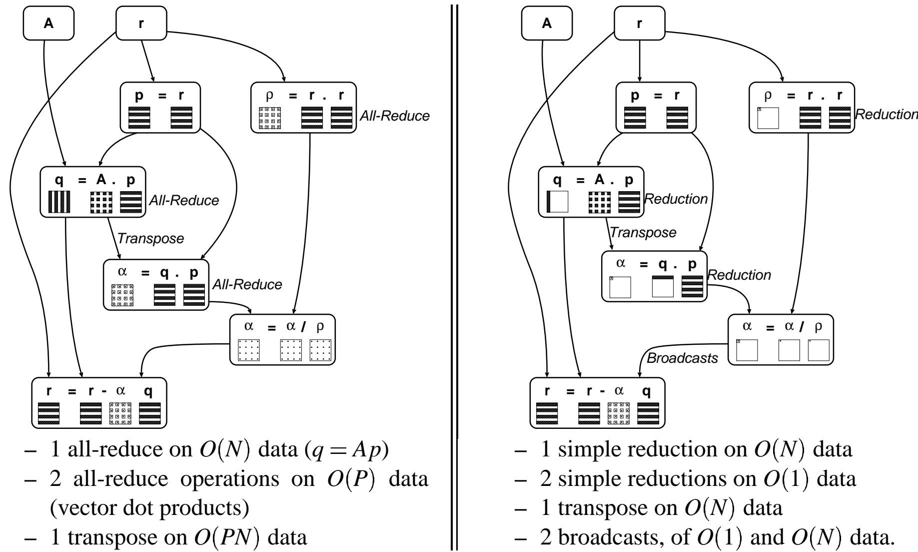


**Fig. 2.** Sequence of operations from the conjugate gradient iterative algorithm for solving linear systems $Ax = b$, showing data layout on a mesh of processors. Affine alignment has already been optimised in both examples.

compute-intensive part of this algorithm is the vector-matrix product $q = Ap$. On a mesh of processors, this can be parallelised in two dimensions as long as every row of the processor mesh has a private copy of vector $p$. It would not be profitable to reconfigure the processor mesh as a ring for the remaining vector-vector operations; rather, the easiest unoptimised solution is to block all vectors, in replicated form, over the rows (or columns, in the case of $q$) of the mesh. Similarly, scalars are replicated on all processors. As illustrated in the left-hand part of Figure 2, the resulting communications are 3 all-reduce operations and one transpose of $O(NP)$ data. However, this can be optimised: the solution in the right-hand part of Figure 2 keeps replicated only those scalars and vectors which are involved in updating vector $p$. This leads to the following optimisations:

– We replace all-reduce operations with simple reductions and then choose optimum points to broadcast data which is required for updating replicated vectors. On many platforms, this will save a broadcast operation.
– Further, the transpose operation which is necessary to align $q$ and $p$ for the dot-product $\alpha = q.p$ now only has to communicate $O(N)$ data and involves $2P - 1$ rather than $P^2 - P$ processors.

Our choice of unoptimised implementation is arguably somewhat arbitrary; the point is that unless replication is incorporated into the data placement optimisation process, optimisations like these will be missed. Note also that we can improve further still on the solution shown here by choosing a skewed affine placement for $q$. Detecting this automatically requires our optimiser to take account of both affine alignment and replication. We will address this issue again when we discuss future work in Section 6.

## 2 Representing Data Replication

In this section, we introduce our representation for data replication. Our objective has been to develop a representation which is both efficient to implement and which facilitates efficient optimisation.

### 2.1 Overview of Data Placement for Data Parallel Programs

Our starting point is the typical data parallel two-stage approach of a mapping onto virtual processors followed by a distribution of virtual processors onto physical ones. The notion of a virtual processor grid is equivalent to that of a template, as described by Chatterjee et al. [5].

We *augment* the dimensionality of all arrays in an optimisation problem to the highest number of dimensions occurring in that problem. This is a purely conceptual step which does not imply any data movement and it is equivalent to the concept that a template is a Cartesian grid of "sufficiently high dimension" into which all arrays can be mapped [5]. If we wish to map an $N$-vector over a two-dimensional processor grid, we conceptually treat this vector as a $(1,N)$ matrix[1]. Scalars are handled in the same way, so a scalar would be treated as a $(1,1)$-array when mapped onto the same grid. Following augmentation, our representation for data placement consists of three stages:

1. *Replication descriptors* allow us to represent the replication of arrays in any dimension where their extent is 1. We describe these descriptors in detail later in this section.
2. *Affine alignment functions* act on augmented, replicated array index vectors $i$ and map them onto virtual processor indices. They take the form

$$f(i) = Ai + t \quad . \tag{1}$$

   The alignment function for mapping a row vector over the rows of a processor mesh is $f(i) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} i + \begin{pmatrix} 0 \\ 0 \end{pmatrix}$. Note that this representation allows us to capture axis, stride and offset alignment as defined in [5]. Some approaches [5, 8] limit the nature of the matrix $A$, such as to require exactly one non-zero entry per column and no more than one non-zero entry per row. The only restriction we impose on these alignment functions is that they be invertible. Thus, we can represent skewings, as well as simple permutations for axis alignments.
3. *Distribution* or *folding* functions map virtual processor index vectors onto pairs of physical processor and local indices. We currently use the well-known symbolic representations `block`, collapsed (`*`) and `cyclic(N)` as distribution functions. Notice that folding allows us to "serialise" some of the replication we may have introduced in step 1, leaving no more than one copy of each array per physical processor.

[1] We always add dimensions in initial positions. Thus, when augmenting an $N$-vector to 2 dimensions, we always treat it as a $(1,N)$ matrix, never as a $(N,1)$ matrix.

**Properties of Affine Alignment Functions.** The affine alignment functions we have described above have two properties which facilitate efficient implementation and optimisation: *invertibility* and *closure under composition*.

- Invertibility means that, given an affine alignment function, we can always calculate both which virtual processor holds a particular array element, and also, which array element is held by a particular virtual processor. This property facilitates sender-initiated communication, an important optimisation on distributed-memory architectures.
- Further, given the above properties and two affine alignment functions $f$ and $g$ for an array, we may always calculate a *redistribution* function $r = f^{-1} \circ g$, which is itself an affine function (invertibility gives us the existence of $f^{-1}$ and closure under composition that $r$ is affine). As we will discuss in more detail in Section 3, this property facilitates efficient optimisation. We define a weight function $w$, which returns an estimate of the amount of data movement generated by $r$. The optimisation problem we need to solve is then to minimise, over all edges in a DAG, the sum of weights $w(r)$ associated with the redistributions $r$ along the edges, subject to placement constraints. Examples for this approach are [2, 6].

### 2.2 Replication Descriptors

Our aim in designing descriptors for data replication has been to re-use as much previous work on optimising affine alignment as possible; we have therefore required that our descriptors have both the above-mentioned properties of invertibility and closure under composition. The advantages become apparent in Section 3 where we outline our optimisation algorithm.

Let $d_v$ be the number of dimensions of an array after augmentation (i.e. the number of dimensions of the virtual processor grid). Let $V$ be the index space of the array after augmentation and let $V_n$ be the set of all possible index values in dimension $n$ of $V$. Let $i$ be an array index vector. We define $(\ ,\ )$ to be a constructor function which takes two $d_v \times d_v$ matrices $D_1$, $D_2$ and returns a function $(D_1, D_2)$, where

$$(D_1, D_2)\ i = D_1 \cdot \mathrm{Solve}(D_2, i\,)\ . \tag{2}$$

$\mathrm{Solve}(M, v\,)$, where $M$ is a matrix and $v$ a vector, is the set of solutions to the equation $Mx = v$, i.e. $\mathrm{Solve}(M, v\,) \stackrel{\mathrm{def}}{=} \{x \mid Mx = v\,\}$. This is also known as the pre-image of $M$.

**Definition 1 (Copy Function).** *We now define a replication or copy function $c$ to be $(D_1, D_2)$, where $D_1$, $D_2$ are $d_v \times d_v$ matrices, and we have*

$$\begin{aligned} c\ i &\stackrel{\mathrm{def}}{=} (D_1, D_2)\ i \\ &= D_1 \cdot \mathit{Solve}(D_2, i\,) \\ &= \{D_1 x \mid D_2 x = i \text{ and } x \in V\}\ . \end{aligned} \tag{3}$$

Matrix $D_2$ is used to generate sets of locations to copy data to; $D_1$ is used to collapse sets. We first give one preliminary example and then prove that this definition does indeed meet the properties which we require. Further examples and rationale follow.

*Example 1.* The copy function for replicating a vector down the columns of a processor mesh is $\left(\begin{pmatrix}1&0\\0&1\end{pmatrix}, \begin{pmatrix}0&0\\0&1\end{pmatrix}\right)$. Note that the vector x is a row-vector, i.e. a $(1,n)$ array. Its first index value is therefore always 0. Thus, we have:

$$\begin{aligned}\begin{pmatrix}1&0\\0&1\end{pmatrix} \cdot \text{Solve}(\begin{pmatrix}0&0\\0&1\end{pmatrix}, i) &= \{x \mid \begin{pmatrix}0&0\\0&1\end{pmatrix} x = \begin{pmatrix}0\\i\end{pmatrix}, x \in V\} \\ &\overset{H}{=} \{x \mid \begin{pmatrix}0&0\\0&1\end{pmatrix} x = 0, x \in V\} + \begin{pmatrix}0\\j\end{pmatrix} \\ &= \{\begin{pmatrix}x_1\\0\end{pmatrix} \mid x_1 \in V_1\} + \begin{pmatrix}0\\i\end{pmatrix} \\ &= \{\begin{pmatrix}x_1\\i\end{pmatrix} \mid x_1 \in V_1\} \ .\end{aligned}$$

The second equality, marked H, is due to the homomorphism theorem [7]. We will expand shortly. Each vector element $i$, which after augmentation corresponds to $\begin{pmatrix}0\\i\end{pmatrix}$, therefore gets mapped to all virtual processor indices in its column.

*Remark 1.* The only formal restriction which we have imposed on the matrices $D_1$ and $D_2$ in a replication descriptor is that their dimensions are $d_v \times d_v$. However, we do not lose any expressive power in practice by only using *diagonal* matrices: A skewed replication such as $\left(\begin{pmatrix}1&0\\0&1\end{pmatrix}, \begin{pmatrix}1&-1\\0&0\end{pmatrix}\right)$ can always be achieved by using a replication descriptor consisting of diagonal matrices $\left(\begin{pmatrix}1&0\\0&1\end{pmatrix}, \begin{pmatrix}0&0\\0&1\end{pmatrix}\right)$ together with a skewed affine alignment function $f(i) = \begin{pmatrix}1&0\\1&1\end{pmatrix} i + \begin{pmatrix}0\\0\end{pmatrix}$.

**Proposition 1.** *The composition of two copy functions* $c_1 = (D_1, D_2)$, $c_2 = (E_1, E_2)$ *is*

$$c_1 \circ c_2 = (D_1 \cdot E_1,\ E_2 \cdot D_2) \ . \tag{4}$$

*Proof.* We have

$$\begin{aligned}(c_1 \circ c_2)(i) &= c_1(\ \{E_1 x \mid E_2 x = i, x \in V\}\ ) \\ &= \{D_1 y \mid D_2 y = E_1 x,\ E_2 x = i,\ y, x \in V\} \\ &= \{D_1 E_1 y \mid D_2 y = x,\ E_2 x = i,\ y, x \in V\} \\ &= \{D_1 E_1 y \mid E_2 D_2 y = i,\ y \in V\} \ .\end{aligned}$$

□

**Proposition 2.** *The composition of two copy functions is again a copy function.*

*Proof.* Follows from the fact that the product of two $d_v \times d_v$ matrices is a $d_v \times d_v$ matrix.

**Proposition 3.** *If the matrices $D_1$ and $D_2$ contain identical entries in corresponding locations, we may "cancel" those entries by replacing them with 1 in both matrices.*

*Proof.* We examine the one-dimensional case. Let $d_1$, $d_2$ be arbitrary scalars. Thus, we have $(d_1, d_2)\ i = d_1 \cdot \text{Solve}(d_2, i)$. If we now assume that $d_1 = d_2$, we have

$$\begin{aligned}(d_1, d_2)\ i &= d_1 \cdot \text{Solve}(d_1, i) \\ &= \{d_1 x \mid d_1 x = i\} \\ &= \{i\} \\ &= (1,1)\ i \ .\end{aligned} \tag{5}$$

The multidimensional case easily follows. Note that this type of "cancellation" even applies if the identical corresponding entries are zeros. □

**Proposition 4.** *The inverse of a copy function* $c = (D_1, D_2)$ *is*

$$c^{-1} = (D_2, D_1) \ . \tag{6}$$

*Proof.* $c \circ c^{-1} = (D_1 D_2, D_1 D_2)$. Therefore, the two matrices in $c \circ c^{-1}$ are identical, which means that according to Proposition 3, we can replace all entries with 1, so we have $c \circ c^{-1} = (I, I)$. □

**Rationale.** The first problem that had to be addressed when trying to represent replication is that a one-to-many "mapping" is not a function. The first idea in trying to work around this problem was to represent the "inverse replication" function instead, i.e., a many-to-one mapping. Given such an inverse function $f$, we have to solve equations of the form $f(x) = i$ in order to establish which virtual processors the data element with index vector $i$ is replicated on.

Since we wish to optimise at runtime, the second challenge was to ensure that these equations can be solved very efficiently; in particular, their solution complexity should not depend on either array data size or processor numbers. We make use of the homomorphism theorem [7]: Formally, an equation is a pair $(f, y)$ of a function $f : D \longrightarrow R$ and an element $y$ of $R$. The *solution* of $(f, y)$ is the set $\{x \in D \mid f(x) = y\}$. The *kernel* of $f$ is the solution to $(f, 0_R)$: $\mathrm{Kern}\, f = \{x \in D \mid f(x) = 0_R\}$. If a function $f$ is a homomorphism, it may not be invertible, but, we can make a very useful statement about the nature of the solution to all equations of the form $(f, y)$ with $y \in R$: the homomorphism theorem states that for all $y \in R$,

$$\mathrm{Solve}(f, y) = \{x \in D \mid f(x) = y\} = \mathrm{Kern}\, f + y \ . \tag{7}$$

This means that although we may not be able to formulate an inverse for such a function, we need only solve *one* equation, the kernel, in order to be able to state the solutions to *all* possible equations involving this function: they may then be calculated by simple addition. The requirement that the inverse copy function be a homomorphism meant choosing a vector-matrix product, i.e. multiplication by the matrix $D_2$ in our replication descriptor.

Finally, since the inverse copying homomorphisms $D_2$ are not invertible, we cannot use them to represent collapsing, i.e. a change from replicated to non-replicated placement. We therefore use a pair $(D_1, D_2)$ of matrices. Multiplying the solutions to the equation $D_2 x = i$ by $D_1$ allows us to represent collapsing.

**Intuition.** Our construction of an invertible representation for data replication is in many aspects analogous to the construction of rational numbers from integers, which is prompted by the lack multiplicative inverses in $\mathbb{Z}$. In both cases, the answer is to use a pair (fraction) of elements. Note also the parallel nature of the definitions for composition (multiplication) and inverse, and the notion of 'cancel and replace with 1'. One important difference, though, is that since we are dealing with finite sets of numbers (index vector spaces), having zeros in the right hand component ('denominator') does not cause problems.

### 2.3 Examples

1. The copy function for replicating a scalar on column 0 of a processor mesh is $\left(\left(\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right),\left(\begin{smallmatrix}0&0\\0&1\end{smallmatrix}\right)\right)$ (see Example 1).
2. The copy function for replicating a scalar on row 0 of a mesh of processors is $\left(\left(\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right),\left(\begin{smallmatrix}1&0\\0&0\end{smallmatrix}\right)\right)$.
3. The *redistribution* function for changing the placement of a scalar from replicated on column 0 to replicated on row 0 is $\left(\left(\begin{smallmatrix}0&0\\0&1\end{smallmatrix}\right),\left(\begin{smallmatrix}1&0\\0&0\end{smallmatrix}\right)\right)$.
4. We conclude with a more complicated example: Suppose we have an $n \times n$ matrix distributed `(block, block)` over a $p \times p$ processor mesh, and that we wish to replicate an $m$-element vector on every processor, i.e., 'align' the entire vector with every $\frac{n}{p} \times \frac{n}{p}$ block of the matrix. We can represent such a placement. We augment the virtual processor space dimensions to 3, treating the matrix as $1 \times n \times n$, and then choose the following placement descriptors:

   <u>Matrix</u>:

   Replication: $\left(\begin{pmatrix}1&0&0\\0&1&0\\0&0&1\end{pmatrix},\begin{pmatrix}1&0&0\\0&1&0\\0&0&1\end{pmatrix}\right)$

   Affine: $f(i) = \begin{pmatrix}1&0&0\\0&1&0\\0&0&1\end{pmatrix} i + \begin{pmatrix}0\\0\\0\end{pmatrix}$

   Folding: `(*, block, block)`

   <u>Vector</u>:

   Replication: $\left(\begin{pmatrix}1&0&0\\0&1&0\\0&0&1\end{pmatrix},\begin{pmatrix}1&0&0\\0&0&0\\0&0&0\end{pmatrix}\right)$

   Affine: $f(i) = \begin{pmatrix}0&1&0\\0&0&1\\1&0&0\end{pmatrix} i + \begin{pmatrix}0\\0\\0\end{pmatrix}$

   Folding: `(*, *, *)`

   The point here is that although we cannot replicate the vector along a dimension where its data extent is more than 1, we can use a combination of augmentation, affine permutation of axes, replication along those axes which after permutation have an extent of 1 and collapsed distribution to represent the same effect.

**Summary.** We have presented a powerful and efficient, invertible mathematical representation for data replication in data parallel programs. We have illustrated that we can represent a wide range of replicated data placements. We will discuss related work, in particular by Chatterjee et al. [4] in Section 5.

## 3 Optimisation

We have developed the techniques described in this paper in the context of a delayed evaluation, self-optimising (DESO) library of parallel numerical routines. The library uses delayed evaluation of library operators to capture the control flow of a user program at runtime. This is achieved through wrappers round the actual parallel numerical routines. When we encounter a *force point* (a point in the user program where evaluation of our library calls can be delayed no longer, such as when output is required), we call our interprocedural data placement optimiser on the DAG of library calls that has been accumulated.

We have previously described [2] and implemented an affine alignment optimisation algorithm, loosely based on that of Feautrier [6]. In this following section, we outline an algorithm for optimising replication. We make use of the invertibility and closure

properties of our replication descriptors so that this algorithm follows a very similar pattern to our affine alignment optimisation algorithm.

It is not possible within the confines of this paper to give an exhaustive description of our optimisation algorithm; we will therefore focus on describing key enabling techniques for our algorithm which rely on the replication representation described in Section 2: metadata for operators, redistributions and redistribution cost.

### 3.1 Library Operator Placement Constraints

Our library operators have one or more parallel implementations. Each of these implementations is characterised by a set of placement constraints (metadata) that constrain our search during optimisation. In our case, these have been provided by the library implementor; however, they could also have been computed by a compiler. Note that each call to a library operator forms a node in the DAG we are optimising. Our library operators therefore precisely correspond to single statements in the compile-time alignment optimisation approach of Chatterjee [5], where nodes in the graph represent array operations. In this paper, we will concentrate on those placement constraints which describe replication.

- The replication placement constraints for library operators describe the *placement relationship* between the result and the operands. For a library operator which defines an array y, reading array x, we denote the replication descriptor for the result y by $c_{\mathtt{y}}$ and the descriptor for the operand x by $c_{\mathtt{y}_\mathtt{x}}$. For example, for the `daxpy` loop $\mathtt{y} \leftarrow \alpha\mathtt{x} + \mathtt{y}$, we have

$$c_{\mathtt{y}_\mathtt{x}} = \left(\left(\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right), \left(\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right)\right) \circ c_{\mathtt{y}} \qquad c_{\mathtt{y}_\alpha} = \left(\left(\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right), \left(\begin{smallmatrix}1&0\\0&0\end{smallmatrix}\right)\right) \circ c_{\mathtt{y}} \ . \tag{8}$$

  This means that the input vector x always has the same degree of replication as the result y, while $\alpha$ has replication along dimension 1 added to the placement of y.
- Thus, when doing a vector update the result of which is required in non-replicated form, the chosen replication placements will be that the input vector x is not replicated, while $\alpha$ is replicated along row 0 of a processor mesh.
  However, when the required placement for the result is replicated on all rows of a processor mesh, i.e, $c_{\mathtt{y}} = \left(\left(\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right), \left(\begin{smallmatrix}0&0\\0&1\end{smallmatrix}\right)\right)$, then we can work out the resulting placements for the operands x and $\alpha$ as follows:

$$\begin{aligned} c_{\mathtt{y}_\mathtt{x}} &= \left(\left(\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right), \left(\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right)\right) \circ \left(\left(\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right), \left(\begin{smallmatrix}0&0\\0&1\end{smallmatrix}\right)\right) \\ &= \left(\left(\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right), \left(\begin{smallmatrix}0&0\\0&1\end{smallmatrix}\right)\right) \end{aligned} \qquad \begin{aligned} c_{\mathtt{y}_\alpha} &= \left(\left(\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right), \left(\begin{smallmatrix}1&0\\0&0\end{smallmatrix}\right)\right) \circ \left(\left(\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right), \left(\begin{smallmatrix}0&0\\0&1\end{smallmatrix}\right)\right) \\ &= \left(\left(\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right), \left(\begin{smallmatrix}0&0\\0&0\end{smallmatrix}\right)\right) \ . \end{aligned}$$

  Thus, x will now be replicated on every row of the processor grid, while $\alpha$ is replicated on every processor.
- When our optimiser changes the placement of one of the operands or of the result of any node in a DAG, it can use these placement constraints to re-calculate the placements for the other arrays involved in the computation. Library operators also have to adapt dynamically their behaviour so as to always comply with their placement constraints.

### 3.2 Calculating Required Redistributions

While accumulating a DAG, our library assigns placements to library operands according to the default scheme mentioned in Section 1: when aligned with higher-dimensional arrays, lower-dimensional arrays are replicated in dimensions where, after augmentation, their extent is 1. In particular, this means that scalars are by default replicated on all processors.

Once a DAG has been accumulated and is available for optimisation, our algorithm begins by calculating the required *replication redistributions* between the placements of arrays at the source and sink of all edges in the DAG. We denote *nodes* in a DAG by the values they calculate. For an edge $a \longrightarrow b$, we denote the replication descriptor (copy function) of a at the source by $c_{\mathtt{a}}$ and the copy function at the sink by $c_{\mathtt{a_b}}$. The replication redistribution function $r_{\mathtt{a}\to\mathtt{b}}$ for this edge is defined by $c_{\mathtt{a}} = r_{\mathtt{a}\to\mathtt{b}} \circ c_{\mathtt{a_b}}$ and may be calculated as $r_{\mathtt{a}\to\mathtt{b}} = c_{\mathtt{a}} \circ c_{\mathtt{a_b}}^{-1}$ .

### 3.3 Cost Model for Redistributions

We define the *size vector* $N_{\mathtt{a}}$ of an array a to be the vector consisting of the array's data size in all dimensions, so for an $n \times m$ matrix $M$, we have $N_{\mathtt{M}} = \binom{n}{m}$. We define the data volume $\mathcal{V}_{\mathtt{a}}$ of a as $\mathcal{V}_{\mathtt{a}} = \prod_{0 \leqslant i \leqslant d_v} N_{\mathtt{a}}[i]$, in other words, $\mathcal{V}_{\mathtt{a}}$ is the total data size of a. Let $P$ be the vector consisting of the physical processor grid's size in all dimensions. Given these definitions, we may build a reasonably accurate model of communication cost for a *replication redistribution* function $r_{\mathtt{a}\to\mathtt{b}} = (D_1, D_2)$ as follows: We first calculate which dimensions $i$, $0 \leqslant i \leqslant d_v$ are replicated by $r_{\mathtt{a}\to\mathtt{b}}$. We then define the cost, or weight, of the edge $\mathtt{a} \longrightarrow \mathtt{b}$ as

$$W_{\mathtt{a}\to\mathtt{b}} = \sum_{\substack{0 \leqslant i \leqslant d_v \\ \text{dimension } i \text{ replicated}}} C_{\text{bcast}}(P_i, \mathcal{V}_{\mathtt{a}}) \quad , \tag{9}$$

where $C_{\text{bcast}}(p, m)$ is the cost of broadcasting an array of size $m$ over $p$ processors. On typical platforms, we have $C_{\text{bcast}}(p, m) \approx (t_s + t_w m) \log p$, with $t_s$ being the message startup time and $t_w$ per-word transfer time.

The key aspect of this cost model is that it takes account of both the data size and the number of processors involved in broadcast operations that may result from replication redistributions.

### 3.4 The Algorithm

Given that our replication descriptors now have the same essential properties as our affine alignment descriptors, the same algorithm which we have previously described for affine alignment optimisation [2] applies. It is originally based on the algorithm proposed by Feautrier in [6].

1. We select the edge with the highest weight. Suppose this is an edge $\mathtt{a} \to \mathtt{b}$.
2. We change the distribution at the *sink* of the edge such that the redistribution $r_{\mathtt{a}\to\mathtt{b}}$ is avoided, i.e., we substitute $c_{\mathtt{a_b}} \leftarrow c_{\mathtt{a}}$. We then use the constraint equations at node b for calculating the resulting placement of b and any other operands and *forward-propagate* this change through the DAG.

3. We check the weight of the DAG following the change. If the weight has gone up, we abandon the change and proceed to step 4. If the weight has gone down, we jump to step 6.
4. We change the distribution at the *source* of the edge by substituting $c_{\mathtt{b}} \leftarrow c_{\mathtt{a}_{\mathtt{b}}}$. We update the placements of the operands at node `a` and *backwards-propagate* the change through the DAG.
5. We check the weight of the DAG. If it has gone up, we abandon the change and mark the edge $\mathtt{a} \rightarrow \mathtt{b}$ as "attempted". Otherwise, we accept the change.

We stop optimising if the weight of the DAG becomes zero. This, however, is rare. Otherwise, we iterate the algorithm a fixed, small number of times each time we encounter a particular context, attempting to eliminate the costliest remaining residual communication. This is particularly suitable for runtime systems where we wish to only spend a strictly limited time optimising whenever the algorithm is invoked.

Once we have begun optimising, we use our value numbering scheme [2] for recognising previously encountered contexts and no longer use our default placement strategy for such nodes, but rather use the results of the last optimisation. Thus, we have the chance of improving on a placement scheme every time a particular context is encountered.

**Summary.** The fact that our replication descriptors have two key properties means that we have been able to propose an algorithm for optimising data replication which is exactly analogous to our previous affine alignment optimisation algorithm. The algorithm aims to minimise the overall communication cost arising from data replication. It works incrementally, attempting to eliminate the costliest communications at each stage. We review related work in Section 5.

## 4 Evaluation

We have implemented the techniques described in this paper in our DESO library of parallel linear algebra routines. In this section, we show performance results for an implementation of the Conjugate Gradient iterative algorithm [1] which uses our library (see [2] for a source code sample). Table 1 splits the overall time spent by our benchmark into different categories; in particular, *point-to-point* communication accounts for transpose operations and *collective* communication for reductions and broadcasts.

- We achieve very encouraging parallelisation speedup: 13.03 for 16 processors.
- Affine alignment optimisation alone achieves a reduction by a factor of about 2 in point-to-point communication.
- Performing the replication optimisation algorithm from this paper in addition to affine alignment optimisation results in a further factor 2.0–2.8 reduction in point-to-point communication. In addition, collective communication is decreased by about 10%. The two key motivations for this work were that handling replication correctly results in cheaper affine realignments and in fewer broadcasts.

| | *P* | *Compu-tation* | *Runtime Overhead* | Communication *Pt-to-Pt* | Communication *Collective* | *Optimi-sation* | *Total* Σ | *O-Speedup* | *P-Speedup* |
|---|---|---|---|---|---|---|---|---|---|
| *N* | 1 | 4351.92 | 7.10 | 0.00 | 0.24 | 0.00 | 4359.26 | 1.00 | 1.00 |
| *A* | 1 | 4359.11 | 7.22 | 0.00 | 0.25 | 6.43 | 4372.99 | 1.00 | 1.00 |
| *R* | 1 | 4340.46 | 7.39 | 0.00 | 0.25 | 11.01 | 4359.11 | 1.00 | 1.00 |
| *N* | 4 | 1108.62 | 12.00 | 57.18 | 95.85 | 0.00 | 1273.66 | 1.00 | 3.42 |
| *A* | 4 | 1114.36 | 10.77 | 28.78 | 80.89 | 6.85 | 1241.64 | 1.03 | 3.52 |
| *R* | 4 | 1095.75 | 10.21 | 12.63 | 60.12 | 16.41 | 1195.12 | 1.07 | 3.65 |
| *N* | 9 | 467.35 | 11.77 | 51.28 | 83.90 | 0.00 | 614.30 | 1.00 | 7.10 |
| *A* | 9 | 464.51 | 11.52 | 27.08 | 72.49 | 7.12 | 582.71 | 1.05 | 7.50 |
| *R* | 9 | 463.53 | 10.57 | 14.32 | 65.29 | 16.64 | 570.34 | 1.08 | 7.64 |
| *N* | 16 | 238.28 | 12.50 | 41.22 | 72.82 | 0.00 | 364.81 | 1.00 | 11.95 |
| *A* | 16 | 237.94 | 12.01 | 25.00 | 62.88 | 7.17 | 345.00 | 1.06 | 12.68 |
| *R* | 16 | 235.07 | 10.09 | 8.92 | 64.22 | 16.18 | 334.48 | 1.09 | 13.03 |

**Table 1.** Time in milliseconds for 10 iterations of conjugate gradient, with a $3600 \times 3600$ parameter matrix (about 100 MB) on varying numbers of processors. *N* denotes timings without any optimisation, *A* timings with affine alignment optimisation only, and *R* timings with affine alignment and replication optimisation. *O-Speedup* shows the speedup due to our optimisations, and *P-Speedup* the speedup due to parallelisation. The platform is a cluster of 350MHz Pentium II workstations with 128MB RAM, running Linux 2.0.36 (TCP patched), connected by two channel-bonded 100Mb/s ethernet cards per machine through a Gigabit switch and using `mpich-1.1.1`. Averages of 10 runs; the standard deviation is about 1% of the reported figures.

– The data in Table 1 were obtained with optimisation running on every iteration of the CG loop. The optimisation times we achieve show that firstly, our replication algorithm takes roughly the same time as affine alignment optimisation, and, secondly, that it is feasible to execute both at runtime in this way. However, we have previously described a technique [2] that allows us to *re-use* the results of previous optimisations at runtime. Applying this technique here will cause the overall optimisation time to become insignificant. We plan to implement this shortly.
– Conjugate Gradient has $O(N^2)$ computation complexity, but only $O(N)$ communication complexity. This means that for relatively small numbers of processors with a fairly large problem size, such as in Table 1, the overall speedups that can be achieved by optimising communication are small. We expect our optimisations to have greater overall benefit on more fine-grain problems and problems with a higher communication complexity.

## 5 Related Work

*Affine Alignment Optimisation.* Feautrier [6] proposes a compile-time method for automatic distribution which works for static-control programs. This method minimises communication between *virtual* processors, i.e. it deals with the affine alignment stage of parallel data placement only. The method does not address replication; for lower-dimensional loops, some processors are assumed to remain idle. Further, the placement

functions in [6] are static. In contrast, our method allows for dynamic realignments and dynamic changes in replication, and will attempt to schedule such operations in an optimal way.

Chatterjee et al. [5] give a comprehensive theoretic treatment of the alignment problem, including axis-, stride- and offset-alignment. Our affine alignment descriptors are very similar to those of Chatterjee et al., though we impose slightly fewer restrictions.

*Optimising Replicated Alignments.* To our knowledge, the only previous work on optimising replicated alignments is by Chatterjee et al. [4]. They use a representation which permits replicating data not just on entire template axes, but also on subsets of template axes. However, this refinement is not taken into account in their optimisation algorithm. On the other hand, it appears that our use of augmentation, together with carefully chosen alignment, permits us to handle a range of replication patterns, as illustrated in Section 2.3 which the representation in [4] was not intended for. We consider the strongest point of our representation to be the two properties of closure under composition and invertibility.

Chatterjee et al. propose to use replication labelling, where data is labelled either replicated or non-replicated, and network flow is used to find an optimal labelling. In comparison, we use a more finely differentiated representation and cost model for replication. While Chatterjee et al. therefore solve a slightly simpler problem than we do, their proposed algorithm finds the optimum solution to the problem as they formulate it. Our algorithm, solving a harder problem, is heuristic and incremental, seeking to eliminate the costliest communications as quickly as possible. This makes our algorithm particularly suitable to runtime optimisation, without restricting its potential of finding the optimum solution with a larger investment in time.

## 6 Conclusion

We have presented an efficient technique for optimising data replication:

- We propose a mathematical representation for replication which satisfies the properties of closure under composition and invertibility.
- These two properties of our replication descriptors allow us to propose an optimisation algorithm for data replication which is exactly analogous to previously published algorithms for optimising affine alignment.
- Our optimisation algorithm is efficient enough to be used in a runtime system, but we believe that its simplicity should also make it attractive for compile-time optimisers.

*Future Work.* This work can be extended in a number of ways. By taking account of affine placements while optimising replication, and vice-versa, we should be able to detect overall placement strategies which are more efficient still than what we can obtain by optimising both separately. For example, using skewed placements for the results of non-replicated reductions may allow us to eliminate some affine re-alignments which appear inevitable when the result of the reduction is replicated.

Through most of this paper we have assumed a two-dimensional processor array. This works well for BLAS, but we should evaluate our techniques for the one-dimensional and higher-dimensional cases. A more difficult issue is how to mix different processor arrangements within a single computation.

*Acknowledgements.* This work is partially supported by a United Kingdom Engineering and Physical Sciences Research Council (EPSRC) Studentship for Olav Beckmann. Special thanks to colleagues at UCSD for their hospitality during Paul Kelly's sabbatical. We thank Steven Newhouse, Keith Sephton and the Imperial College Parallel Computing Centre for the use of their Beowulf cluster and AP3000, and Scott Baden and NPACI for SP2 access. We thank Ariel Burton from Imperial College for supplying a lightweight, accurate timing mechanism under Linux [3] and for pointing out some 'performance bugs' in the TCP implementation of 2.0.x Linux kernels.

## References

1. R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, USA, 1994.
2. 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.
3. A. N. Burton and P. H. J. Kelly. Tracing and reexecuting operating system calls for reproducible performance experiments. *Journal of Computers and Electrical Engineering—Special Issue on Performance Evaluation of High Performance Computing and Computers*, 1999. To appear.
4. 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.
5. 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.
6. P. Feautrier. Toward automatic distribution. *Parallel Processing Letters*, 4(3):233–244, 1994.
7. J. A. Green. *Sets and Groups.* Routledge & Kegan Paul, second edition, 1988.
8. C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele Jr., and M. E. Zosel. *The High Performance Fortran Handbook.* MIT Press, Cambridge, MA, USA, Jan. 1994.
9. V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing.* Benjamin/Cummings, 1993.
10. C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5(3):308–323, Sept. 1979.
11. Z. Li. Array privatization for parallel execution of loops. In *1992 International Conference on Supercomputing, Washington, DC*, pages 313–322. ACM Press, 1992.
12. L. Snyder. *A Programmer's Guide to ZPL.* Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195, Jan. 1999. Verion 6.3.
13. S. A. M. Talbot. *Shared-Memory Multiprocessors with Stable Performance.* PhD thesis, Department of Computing, Imperial College London, U.K., 1999.