Efficient Interprocedural Data Placement Optimisation in a Parallel Library

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. This paper describes a combination of methods which make interprocedural data placement optimisation available to parallel libraries. We propose a delayed-evaluation, self-optimising (DESO) numerical library for a distributed-memory multicomputer. Delayed evaluation allows us to capture the control-flow of a user program from within the library at runtime, and to construct an optimised execution plan by propagating data placement constraints backwards through the DAG representing the computation to be performed.
Our strategy for optimising data placements at runtime consists of an efficient representation for data distributions, a greedy optimisation algorithm, which because of delayed evaluation can take account of the full context of operations, and of re-using the results of previous runtime optimisations on contexts we have encountered before. We show performance figures for our library on a cluster of Pentium II Linux workstations, which demonstrate that the overhead of our delayed evaluation method is very small, and which show both the parallel speedup we obtain and the benefit of the optimisations we describe.

1 Introduction

Parallelising applications by using parallel libraries is an attractive proposition because it allows users to use any top-level calling language convenient to them, such as Fortran, C, C++ or spreadsheets. Further, it has been argued [10, 11] that, at least for the time being, library-oriented parallelism, i.e. the use of carefully tuned routines for core operations, is often the only way to achieve satisfactory performance. A disadvantage of accessing parallelism through libraries, however, is that we seem set to miss opportunities for optimisation across library calls.

We propose a delayed evaluation, self-optimising (DESO) parallel linear algebra library as a way of avoiding this drawback: the actual execution of function calls is delayed for as long as possible. This provides the opportunity to capture the control flow of a user program at runtime. We refer to those points in a program where execution cannot be delayed anymore as *force points*. The most common reasons for this are output or conditional tests which depend on the result of a sequence of delayed operations. On encountering a force-point,

we can construct an optimised execution plan for the DAG representing the computation to be performed.

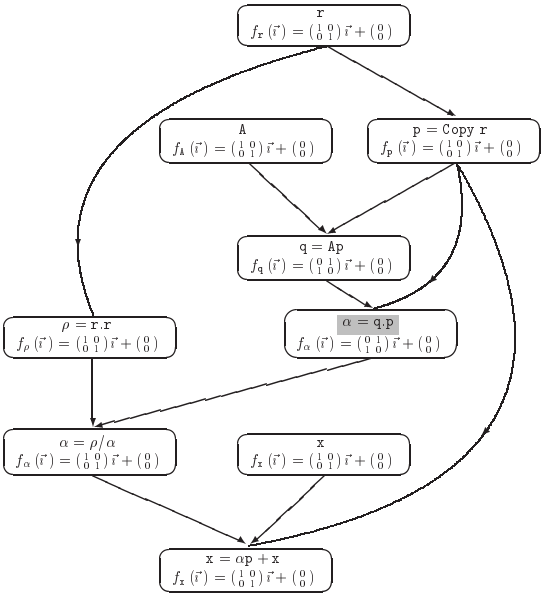

Fig. 1. DAG for the first iteration of the conjugate gradient algorithm. The "open ends" are: `A`, the parameter matrix, `x` the initial guess for a solution vector and `r`, the initial remainder-vector, $\mathtt{r} = \mathtt{b} - \mathtt{Ax}$. The DAG has been annotated with the affine placement functions for the results of all operations. We refer to these in Section 2.1.

Example. Consider the DAG shown in Figure 1. The highlighted dot-product, $\alpha = \mathtt{q.p}$, gives rise to a data placement conflict: If `p` is blocked over the rows of a mesh of processors, then the natural parallel implementation of the vector-matrix product $\mathtt{q} = \mathtt{Ap}$ means that `q` is blocked down the columns. Therefore, we need to redistribute either `p` or `q` in order to calculate the dot-product $\alpha = \mathtt{q.p}$. If we perform the computation immediately, we have no information available about the future use of `p` and `q` to guide our choice. Delayed evaluation allows us to make a better decision by taking account of the use of `p` in the vector update $\mathtt{x} = \alpha\mathtt{p} + \mathtt{x}$.

Key issues. The main challenge in optimising at run-time is that optimisation itself has to be very efficient. We achieve this with a combination of the following techniques:

- Working from aggregate loop nests, which have been optimised in isolation and which are not re-optimised at run-time.

- Using a purely mathematical formulation for data distributions, which allows us to calculate, rather than search for optimal distributions.
- Re-using optimised execution plans for previously encountered DAGs. A value-numbering scheme is used to recognise cases where this may be possible. The value numbers are used to index a cache of optimisation results, and we use a technique adapted from hardware dynamic branch prediction for deciding whether to further optimise DAGs we have previously encountered.

Delayed evaluation introduces runtime anti-dependence hazards which we overcome using a mechanism analogous to register renaming [18].

Related Work, Contribution of this Paper. This paper builds on related work in the field of automatic data placement [9, 15], runtime parallelisation [6, 17], automatic, compile-time parallelisation [3, 7], interprocedural optimisation [12] and conventional compiler and architecture technology [1, 8]. In our earlier paper [5] we described our method for re-using runtime-optimised execution plans in more fundamental terms. In this current paper, we add to this by describing our actual optimisation algorithm and by illustrating the benefits of our re-use strategy when combined with this particular algorithm.

Structure of this paper. We begin in Section 2 by outlining the fundamentals of our approach towards fast, runtime optimisation. Section 3 describes the optimisation algorithm we use and Section 4 presents our techniques for avoiding re-optimisation where appropriate. Finally, Section 5 shows performance results for our library. This is followed by a concluding discussion in Section 6, which includes a review of related and future work.

2 Basic Approach

2.1 Data Distributions.

Our representation for data distributions is based on a combination of affine transformation functions ("alignment" in HPF [14]) and non-affine folding functions ("distribution" in HPF), together with a copying function which represents data replication.

- We *augment* the dimensionality of all arrays in an optimisation problem to the highest number of dimensions occurring in that problem: for the example shown in Figure 1, the matrix `A` is the array with the highest number of dimensions. We therefore augment scalars and vectors to two dimensions, treating them as 1×1 and $1 \times N$ matrices respectively. Thus we refer to the i^{th} element of a vector as element $\binom{0}{i}$.
- Affine transformation functions act on array index vectors $\boldsymbol{i}$ and map them onto *virtual processor* indices. They take the form

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

Example. The affine transformation function for mapping an N-element vector down column 0 of a virtual processor mesh is $f(\boldsymbol{i}) = \left(\begin{smallmatrix}0 & 1\\ 1 & 0\end{smallmatrix}\right)\boldsymbol{i} + \left(\begin{smallmatrix}0\\ 0\end{smallmatrix}\right)$, that is, vector element $\left(\begin{smallmatrix}0\\ i\end{smallmatrix}\right)$ is mapped to virtual processor $\left(\begin{smallmatrix}i\\ 0\end{smallmatrix}\right)$.

- The replication of scalars and vectors, such as whether or not a vector which is mapped down the first column of a virtual processor mesh is replicated on all columns, is represented by a special copying function which we will not describe further here.
- Between any two affine placement functions f and g, we can calculate a *redistribution* function r such that $g = r \circ f$, which is itself an affine function.

We optimise with respect to affine placement functions, aiming to minimise the cost of affine redistributions.

2.2 Library Operator Placement Constraints

Each of our library operators has one or more parallel implementations. For each implementation, we formulate placement constraints as follows.

- Operators are implemented for one arbitrarily but reasonably chosen set of placements for the result and the operands. Naturally, when these placements are obeyed, the loop nest will execute correctly. As an example, for vector-matrix products, such as `q = A.p` in Figure 1, the chosen placements are

$$f_{\mathtt{A}}(\boldsymbol{i}) = f_{\mathtt{p}}(\boldsymbol{i}) = \left(\begin{smallmatrix}1 & 0\\ 0 & 1\end{smallmatrix}\right)\boldsymbol{i} + \left(\begin{smallmatrix}0\\ 0\end{smallmatrix}\right) \text{ and}$$
$$f_{\mathtt{q}}(\boldsymbol{i}) = \left(\begin{smallmatrix}0 & 1\\ 1 & 0\end{smallmatrix}\right)\boldsymbol{i} + \left(\begin{smallmatrix}0\\ 0\end{smallmatrix}\right) \quad .$$

  We also use the notation $A_{\mathtt{q}} = \left(\begin{smallmatrix}0 & 1\\ 1 & 0\end{smallmatrix}\right)$ and $\boldsymbol{t}_{\mathtt{q}} = \left(\begin{smallmatrix}0\\ 0\end{smallmatrix}\right)$.
- From that, we can calculate *constraint equations* which characterise each operator implementation by describing the relationship between the placements of the result and of the operands. In our example, we have

$$\begin{aligned} A_{\mathtt{A}} &= \left(\begin{smallmatrix}0 & 1\\ 1 & 0\end{smallmatrix}\right) A_{\mathtt{q}} & A_{\mathtt{p}} &= \left(\begin{smallmatrix}0 & 1\\ 1 & 0\end{smallmatrix}\right) A_{\mathtt{q}} \\ t_{\mathtt{A}} &= t_{\mathtt{q}} & t_{\mathtt{p}} &= t_{\mathtt{q}} \quad . \end{aligned} \tag{2}$$

- For "open ends" in a DAG, i.e. for nodes representing either arrays which have already been evaluated or the result of a force, the placement is fixed and we obtain constraint equations such as (for Figure 1)

$$A_{\mathtt{r}} = \left(\begin{smallmatrix}1 & 0\\ 0 & 1\end{smallmatrix}\right) \qquad t_{\mathtt{r}} = \left(\begin{smallmatrix}0\\ 0\end{smallmatrix}\right) \quad . \tag{3}$$

- Our optimiser is permitted to change the placement of the result or of one of the operands of a node in the DAG at optimisation time in order to avoid data redistributions. If that happens, we re-calculate the placements for the other arrays involved in the computation at that node by means of the constraint equations described above. Further, we dynamically transform the operator's loop nest, loop bounds and communication pattern. See [3] for a basic introduction to the techniques required.

When a value is forced, the optimisation problem we need to solve consists of the union of of the equations as shown in (2) and (3) for all nodes in a DAG.

3 Optimisation

3.1 Calculating Required Redistributions

Once we have a DAG available for optimisation, our algorithm begins by calculating the affine *redistribution* functions (see Section 2.1) between the placements of arrays at the source and sink of all edges in the DAG. Let nodes in a DAG be denoted by the values they calculate. For an edge $\mathtt{a} \to \mathtt{b}$, we denote the placement of $\mathtt{a}$ at the source by $f_{\mathtt{a}}$ and the placement at the sink by $f_{\mathtt{a_b}}$. We then define the redistribution function for this edge to be the affine function $r_{\mathtt{a}\to\mathtt{b}}$ such that

$$f_{\mathtt{a}} = r_{\mathtt{a}\to\mathtt{b}} \circ f_{\mathtt{a_b}} \quad . \tag{4}$$

For our example in Figure 1, $\mathtt{p}$ is generated with distribution $f_{\mathtt{p}} = \left(\begin{smallmatrix}1 & 0\\ 0 & 1\end{smallmatrix}\right)\boldsymbol{i} + \left(\begin{smallmatrix}0\\ 0\end{smallmatrix}\right)$ and used in the dot-product $\alpha = \mathtt{q}.\mathtt{p}$ with distribution $f_{\mathtt{p}_\alpha} = \left(\begin{smallmatrix}0 & 1\\ 1 & 0\end{smallmatrix}\right)\boldsymbol{i} + \left(\begin{smallmatrix}0\\ 0\end{smallmatrix}\right)$, i.e. aligned with $\mathtt{q}$. The redistribution function for the edge $\mathtt{p} \to \alpha$ therefore is $r_{\mathtt{p}\to\alpha}(\boldsymbol{i}) = \left(\begin{smallmatrix}0 & 1\\ 1 & 0\end{smallmatrix}\right)\boldsymbol{i} + \left(\begin{smallmatrix}0\\ 0\end{smallmatrix}\right)$.

3.2 A Cost Model for Redistributions

We define the *size vector* $N_{\mathtt{a}}$ of an array $\mathtt{a}$ to be the vector consisting of the array's data size in all dimensions, so for an $n \times m$ matrix $\mathtt{M}$, we have $N_{\mathtt{M}} = \left(\begin{smallmatrix}n\\ m\end{smallmatrix}\right)$, and for an n-element vector $\mathtt{v}$, $N_{\mathtt{v}} = \left(\begin{smallmatrix}1\\ n\end{smallmatrix}\right)$. Next, we define $\bar{r}$ to be the function obtained from a redistribution function r by substituting all diagonal elements in the matrix A with 0. For identity functions, we obtain a matrix $A = \left(\begin{smallmatrix}0 & 0\\ 0 & 0\end{smallmatrix}\right)$ and for the transpose in our example, $\bar{r}_{\mathtt{p}\to\alpha} = r_{\mathtt{p}\to\alpha}$. Then, we define the weight $W_{\mathtt{a}\to\mathtt{b}}$ of an edge $\mathtt{a} \to \mathtt{b}$ as

$$W_{\mathtt{a}\to\mathtt{b}} = \|\bar{r}_{\mathtt{a}\to\mathtt{b}}(N_{\mathtt{a}})\|_2 \quad . \tag{5}$$

In our example, assuming that $\mathtt{p}$ is an n-element vector, $W_{\mathtt{p}\to\alpha} \approx n$. This model captures the amount of data movement involved in a redistribution and we have so far found it to give a sufficiently accurate reflection of which redistributions in a DAG are the most costly and should therefore be eliminated first.

3.3 The Algorithm.

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 $f_{\mathtt{a_b}} \leftarrow f_{\mathtt{a}}$. We then use the constraint equations at node $\mathtt{b}$ for calculating the resulting placement of $\mathtt{b}$ 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 $f_{\mathtt{b}} \leftarrow f_{\mathtt{a}_\mathtt{b}}$. 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.
6. If the weight of the DAG has become zero, or, if the remaining weight is entirely due to edges which have already been attempted, we stop optimising. Otherwise, we have the option of optimising further. We describe in Section 4 how we decide whether or not to do this.

3.4 Related Work.

Our optimisation algorithm resembles that of Feautrier [9] in that we optimise with respect to affine placement functions disregarding the non-affine mapping of virtual processors onto physical ones, and use a greedy algorithm, attempting to resolve edges with the highest weight first. However, our approach differs in that we work from aggregate data structures and use a different cost model for communication. The fact that we work at runtime and capture the control flow of a program by delayed evaluation means that we do not have to impose any restrictions on the user's source code, as is the case in [9], where programs are required to be of "static control".

4 Re-Using Execution Plans

The previous section has described our algorithm for finding an optimal execution plan for any one DAG. In real programs, essentially identical DAGs often recur. In such situations, our runtime approach is set to suffer a significant performance disadvantage over compile-time techniques unless we can reuse the results of previous optimisations we have performed. This section shows how we can ensure that our optimiser does not have to carry out any more work than an optimising compiler would have to do, unless there is the prospect of a performance benefit by doing more optimisation than would be possible with static information.

4.1 Run-Time Optimisation Strategies

We begin by discussing and comparing two different basic strategies for performing run-time optimisation. We will refer to them as "Forward Propagation Only" and "Forward And Backward Propagation". We use the following terminology: n is the number of operator calls in a sequence, a the maximum arity of operators, m is the maximum number of different methods per operator. If we work with a fixed set of data placements, s is the number of different placements, and in DAGs, d refers to the degree of the shared node (see [15]) with maximum degree.

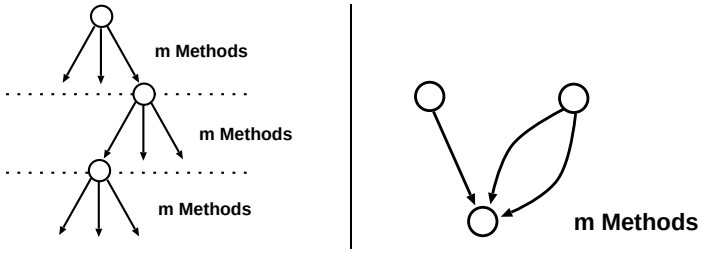

Fig. 2. Left. Optimisation of a linear stream of operators under *Forward Propagation Only*. Complexity is $\Theta(mn)$. **Right.**The only type of DAG we can recognise as such under *Forward Propagation Only*.

Forward Propagation Only. This is the only strategy open to us if we perform run-time optimisation of a sequence of library operators under *strict evaluation*: We optimise the placements for each new operator based purely on information about its ancestors.

- In Figure 2, we illustrate that the total optimisation time for a linear sequence of n operators with m different available methods per operator under this strategy has complexity $\Theta(mn)$.
- For our library (where, as stated in Section 2.2, each method captures a number of possible placements for operands), we would be able to optimise the placement of any a-ary operator in $O(a)$ time, i.e., the time for optimising any call graph would be $O(an)$.
- Linear complexity in the number of operators is probably all we can afford in a runtime system.
- However, as we already illustrated in Figure 1, the price we pay for using such an algorithm is that it may give a significantly suboptimal answer. This problem is present even for trees, but it is much worse for DAGs: Figure 2 shows the only type of DAG we can recognise as a DAG under *Forward Propagation Only*. All other DAGs can not be handled in an optimal way.

Note 1. If we choose to use *Forward Propagation Only*, there is no benefit in delaying optimisation decisions, since we already have all optimisation information available at the time when operators are called.

Forward And Backward Propagation. Delayed evaluation gives us the opportunity to propagate placement constraint information backwards through a DAG since we accumulate a full DAG before we begin to optimise.

- We illustrate in Figure 3 that this type of optimisation is much more complex than *Forward Propagation Only*.
- Mace [15] has shown it to be NP-complete for general DAGs, but presents algorithms with complexity $O((m + s^2)n)$ for trees and with complexity $O(s^{d+1}n)$ for a restricted class of DAG.

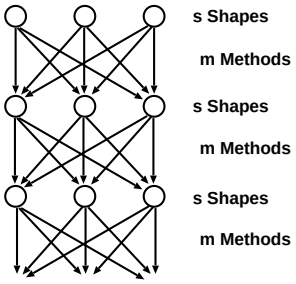

Fig. 3. Optimisation of a linear stream of operators under *Forward And Backward Propagation*. The additional complexity derives from the fact that it may be necessary to choose sub-optimal placements for one step in the calculation in order to gain optimal performance overall: we need to calculate the cost of generating each result in every possible shape, not just the cheapest for that step.

- Our own greedy optimisation algorithm does not enumerate different possible placements, but captures all distributions with one representation (see Section 2.2). This means that the complexity of our optimisation algorithm is independent of the number of possible placements we allow. Its complexity is similar to that of Gaussian elimination, i.e. $O((kn)^2)$ for some constant k. Note that this is an upper bound, the algorithm will often run in linear time.
- The point to note here is that *Forward and Backward Propagation* does give us the opportunity to find the optimal solution to a problem, provided we are prepared to spend the time required.

What we would like to do therefore is to use the full context information to derive correct, optimal placements, but to then re-use the results of such a full optimisation whenever the same optimisation problem occurs.

4.2 Recognising Opportunities for Reuse

We now deal with the problem of how to recognise a DAG, i.e. optimisation problem, which we have encountered before. The full optimisation problem, characterised by the full DAG or the resulting system of equations, is a large structure. To avoid having to traverse it to check for cache hits, we derive a hashed "value number" [1, 8] for each node and use that value number to determine whether we have encountered a DAG before.

- Our value numbers have to encode data placements and placement constraints, not actual data values. For nodes which are already evaluated, we simply apply a hash function to the placement descriptor of that node. For nodes which are not yet evaluated, we have to apply a hash function to the *placement constraints* on that node.

- The key observation is that by seeking to encode all placement constraints on a node in our value numbers, we are in danger of deriving an algorithm for calculating value numbers which has the same O-complexity as *Forward and Backward Propagation* optimisation algorithms: each node in a DAG can potentially exert a placement constraint over every other node.
- Our algorithm for calculating value numbers is therefore based on *Forward Propagation Only*: we calculate value numbers for unevaluated nodes by applying a hash function to those placement constraints deriving from their immediate ancestor nodes only.
- According to Note 1, there is therefore no point in delaying the calculation of value numbers; they are generated on-the-fly, as library operators are called.
- The only way to detect hash conflicts would be to traverse the full DAGs which the value numbers represent. Apart from the fact that such a validation would have $O(n)$ complexity, it would also mean having to store full DAGs which we have previously encountered and optimised. We return to this point shortly.

4.3 When to Re-Use and When to Optimise.

We now discuss when we re-use the results of previous optimisations and when we re-optimise a problem. Because our value numbers are calculated on *Forward Propagation Only* information, we have to address the problem of how to handle those cases where nodes which have identical value numbers are used in a different context later on; in other words, how to to avoid the drawbacks of *Forward Propagation Only* optimisation. This is a branch-prediction problem, and we use a technique adapted from hardware dynamic branch prediction (see [13]) for predicting heuristically whether identical value numbers will result in identical future use of the corresponding node and hence identical optimisation problems.

We store a bit, `OPTIMISE`, to implement a strategy of re-optimising in the following three situations:

1. A DAG has not been encountered before.
2. We have encountered a DAG before and successfully performed some optimisation when we last saw it. We "predict" that more optimisation might lead to further improvements.
3. We re-used a previously stored execution plan when we last encountered a DAG, but found this execution plan to give sub-optimal performance.

- In all other cases, we have seen the encountered DAG before, and we re-use a cached execution plan (see below).
- Point 3 deals with the problem of false cache hits, which we cannot detect directly. The run-time system automatically introduces any necessary redistributions, so the effect of using a wrong cached plan is that the number of redistributions may be larger than expected.
- The metrics which are required for deciding whether or not to invoke the optimiser therefore are (a) the success or otherwise of the optimiser in reducing

the "weight" of a DAG when we last encountered it, (b) the communication-cost of evaluating the DAG when we last encountered it, and (c) the communication cost of the 'optimal' execution plan for a DAG. We have instrumented our system such that these metrics are available.

Caching Execution Plans. Value numbers and 'dynamic branch prediction' together provide us with a fairly reliable mechanism for recognising the fact that we have encountered a node in the same context before. Assuming that we optimised the placement of that node when we first encountered it, our task is then simply to re-use the placement which the optimiser derived. We do this by using a "cache" of optimised placements, which is indexed by value numbers. Each cache entry has a valid-tag which is set by our branch prediction mechanism.

Competitive Optimisation. As we showed in Section 4.1, full optimisation based on *Forward And Backward Propagation* can be very expensive. Each time we invoke the optimiser on a DAG, we therefore only spend a limited time optimising that DAG. For a DAG which we encounter only once, this means that we only spend very little time trying to eliminate the worst redistributions. For DAGs which recur, our strategy is to gradually improve the execution plan used until our optimisation algorithm can find no further improvements.

Finally, it should be pointed out that although our value numbers are calculated on-the-fly, under *Forward Propagation Only*, we still delay the execution of DAGs we have encountered before. This is to allow new, yet unseen contexts to trigger a re-optimisation of DAGs which we have already optimised for other contexts.

Summary. We use the full optimisation information, i.e. *Forward and Backward Propagation*, to optimise. We obtain access to this information by delayed evaluation. We use a scheme based on *Forward Propagation Only*, with linear complexity in program length, to ensure that we re-use the results of previous optimisations.

5 Implementation and Performance

The implementation of our library is based on MPI. This offers portability and, MPI allows parallel libraries to be designed in such a way that they can safely be used together with other communication libraries and in user programs which themselves carry out communication [11]. In this Section, we show performance figures for our library on a cluster of desktop Pentium II Linux workstations here at Imperial College. As a benchmark we used the non-preconditioned conjugate gradient iterative algorithm. The pseudo-code for the algorithm (adapted from [4]) and the source code when implemented using our library are shown in Figure 4.

$r^{(0)} = b - Ax^{(0)}$
for $i = 1, \ldots, i_{max}$
  $\rho_{i-1} = r^{(i-1)^T} r^{(i-1)}$
  if $i = 1$
    $p^{(1)} = r^{(0)}$
  else $\beta_{i-1} = \rho_{i-1}/\rho_{i-2}$
    $p^{(i)} = r^{(i-1)} + \beta_{(i-1)} p^{(i-1)}$
  endif
  $q^{(i)} = Ap^{(i)}$
  $\alpha_i = \rho_{i-1}/p^{(i)^T} q^{(i)}$
  $x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)}$
  $r^{(i)} = r^{(i-1)} - \alpha_i q^{(i)}$
  check convergence
end

```
for(i = 1; i <= max_iter; i++) {
  if (i != 1)
    L_Dcopy(rho, &rho_o);
  L_Ddot(r, r, &rho);
  if (i == 1)
    L_Dcopy(r, &p);
  else {
    L_Ddiv(rho_o, rho, &beta);
    L_Daxpy(beta, p, r, &p);
  };
  L_Dgemv(one, A, p, zero, q, &q);
  L_Ddot(q, p, &alpha);
  L_Ddiv(alpha, rho, &alpha);
  L_Daxpy(alpha, p, x, &x);
  L_Dscal(alpha, minusone, &alpha);
  L_Daxpy(alpha, q, r, &r);
  /* Check convergence. */
};
```

Fig. 4. Pseudo-code for the conjugate gradient algorithm (left) and source code (slightly cut down) when implemented using our library.

5.1 Comparison with Sequential, Compiled Model

Figure 5 compares the performance of the single processor version of our parallel code with two different purely sequential implementations of the same algorithm. One of these uses the same BLAS kernels which our parallel code calls; these are routines which we have manually optimised for the Pentium II processor. The other version links with the Intel / ASCI Red [2] BLAS kernels.

There is virtually no distinction between the performance of our parallel code running on 1 processor and the "best effort" purely sequential version which calls our own optimised BLAS implementation. For the smallest data size in our experiment (512×512 parameter matrix), the code using our parallel library has about 10% less performance in terms of MFLOP/s, but this becomes almost indistinguishable for larger data sizes. Both the codes using our BLAS kernels do, however consistently perform somewhat better than the purely sequential code which links with the Intel / ASCI Red BLAS kernels. This demonstrates that the overhead of our delayed evaluation scheme is very small and that the baseline sequential performance of our code is very respectable.

5.2 Parallel Performance

Table 1 shows the parallel speedup we obtain with our library on a cluster of Pentium II Linux workstations. As noted in the caption of the table, these figures underestimate the actual parallel speedup we obtain. Nevertheless, a parallel speedup of 2.65 for 4 processors is encouraging, given the nature of the platform. On our current configuration, and with the problem size we used (this

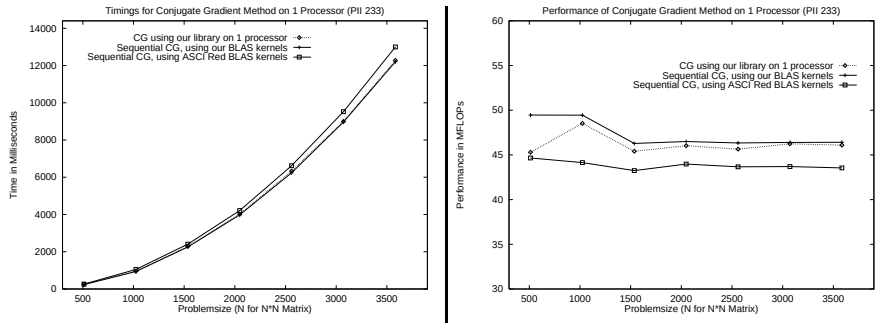

Fig. 5. Performance comparison of the single processor version of our parallel code with two different purely sequential implementations of the same algorithm on a Pentium II 233 with 128MB RAM running Linux 2.0.30. Both purely sequential codes are a direct transcription of the conjugate gradient template which is distributed with [4]; the only difference is in the underlying BLAS implementation they call.

is limited by the machine with least free RAM), our benchmark does not scale well to 9 processors.

5.3 Performance of Our Optimisations

Tables 1 and 2 show detailed measurements which break up the overall time spent by our benchmark into different categories. Note that the CG algorithm has $O(N^2)$ computation complexity, but only $O(N)$ communication complexity. This can be seen in Table 2.

- Our optimiser avoids two out of three vector transposes. This was determined by analysis and by tracing communication.
- Optimisation achieves a reduction in communication time of between 30% and 40%. We do not achieve more because a significant proportion of the communication in this algorithm is due to reduce-operations.
- Runtime overhead and optimisation time are virtually independent of the number of processors and of the problem size. We suspect that the slight differences are due to cache effects.
- With caching of optimisation results, the overall time spent by our optimiser is very small. On platforms with less powerful nodes than PII 300–233s, this aspect or our optimisation strategy is crucial. On such platforms, optimisation without re-use can easily result in overall slowdown.
- The optimisations we have described in this paper achieve speedups over the non-optimised code using our library of between 9% and 29% for reasonably large problem sizes. The overall benefit of our optimisations decreases with increasing problem size on the same number of processors. This is due to the CG algorithm's $O(N^2)$ computation complexity with only only $O(N)$ communication complexity. However, the benefit of our optimisations increases with the amount of parallelism we use to solve a problem.

	P	*Comp.*	*Memory*	*Overh.*	*Comms.*	*Opt.*	*Total*	*O-Speedup*	*P-Speedup*
N	1	21276.13	5.83	11.69	0.00	0.00	21293.65	1.00	1.00
O	1	21280.05	5.95	9.93	0.00	8.69	21304.62	1.00	1.00
C	1	21276.56	5.84	9.54	0.00	0.63	21292.57	1.00	1.00
N	4	5286.77	5.08	9.39	4047.85	0.00	9349.09	1.00	2.28
O	4	5264.42	4.39	8.15	2771.15	8.38	8056.51	1.16	2.64
C	4	5251.73	4.48	8.57	2778.93	0.62	8044.33	1.16	2.65
N	9	2625.52	5.43	10.10	3992.16	0.00	6633.23	1.00	3.21
O	9	2608.84	4.69	8.70	2516.95	9.18	5148.36	1.29	4.14
C	9	2610.86	4.63	8.49	2516.98	0.66	5141.62	1.29	4.14

Table 1. Time in milliseconds for 20 iterations of conjugate gradient, with a 4992 × 4992 parameter matrix (about 190 MB) on varying numbers of processors. *N* denotes timings without any optimisation, *O* timings with optimisation but no caching, and *C* timings with optimisation and caching of optimisation results. *O-Speedup* shows the speedup due to our optimisations, and *P-Speedup* the speedup due to parallelisation. All processors are Pentium IIs running Linux 2.0.32. Note however, that their specification decreases from processor 0 (dual 300 MHz, 512 MB RAM) to processor 8 (233 MHz, 64 MB). Further, processors 0–3 are connected by fast ethernet (100 Mb/s), whereas some of the remaining ones are connected by 10 Mb/s ethernet. Hence, the above numbers for parallel speedup do not actually show the full potential of our library.

6 Conclusions

We have presented an an approach to interprocedural data placement optimisation which exploits run-time control-flow information and is applicable in contexts where the calling program cannot easily be analysed statically. We present preliminary experimental evidence that the benefits can easily outweigh the run-time costs.

6.1 Run-Time vs. Compile-Time Optimisation.

To see some of the relative advantages of our technique and of compile-time methods, consider the following loop, assuming that there are no force-points inside the loop and that we encounter the loop a number of times and each time force evaluation after the loop.

```
for(i = 0; i < N; ++i) {
  if <unknown conditional>
    <do A>
  else
    <do B>
}
```

This loop can have up to 2^N control-paths. A compile-time optimiser would have to find one compromise execution plan for all invocations of this loop. With our approach, we would optimise the actual DAG which has been generated on

	N	*Comp.*	*Memory*	*Overh.*	*Comms.*	*Opt.*	*Total*	*O-Speedup*	$MFLOP/s$
N	1024	218.81	5.00	9.28	783.81	0.00	1016.90	1.00	45.64
O	1024	219.13	4.27	8.09	395.25	8.24	634.99	1.60	73.09
C	1024	217.79	4.35	7.99	394.27	0.61	625.01	1.63	74.26
N	2048	848.32	5.02	9.12	1522.06	0.00	2384.53	1.00	77.63
O	2048	851.87	4.26	8.24	1123.02	8.25	1995.64	1.19	92.75
C	2048	839.00	4.26	7.88	1058.75	0.61	1910.50	1.25	96.89
N	3072	2411.17	5.21	9.63	2910.47	0.00	5336.47	1.00	77.97
O	3072	2343.11	4.27	8.16	2083.26	8.11	4446.89	1.20	93.56
C	3072	2337.84	4.32	8.02	2025.89	0.62	4376.68	1.22	95.06
N	4096	4236.84	5.11	9.23	5354.08	0.00	9605.26	1.00	76.97
O	4096	4267.31	4.32	8.15	4500.90	8.34	8789.02	1.09	84.12
C	4096	4273.14	4.36	8.02	4502.32	0.62	8788.47	1.09	84.12

Table 2. Time in milliseconds for 20 iterations of conjugate gradient on four Pentium II Linux workstations, with varying problem sizes. N, O and C are as in Table 1.

each occasion. If the number of different DAGs is high, compile-time methods would probably have the edge over ours, since we would optimise each time and could not reuse execution plans. If, however, the number of different DAGs generated is small, our execution plans for the *actual* DAGs will be superior to compile-time compromise solutions, and by reusing them, we limit the time we spend optimising.

6.2 Related Work.

Most successful work on parallelising compilers for distributed-memory systems has relied on explicit control over data placement, using e.g. Fortran-D or HPF [14]. The problem for a compiler is then reduced to optimising the communications which are required.

There is a large amount of work on the problem of automatically parallelising an affine nested loop [7, 9]. The first stage of that process is to map each array element and each operation onto a virtual processor, in such a way that as many non-local data accesses as possible are avoided.

We have already mentioned how our optimisation algorithm is related to that of Feautrier [9] in Section 3.4. Our approach has similarities with that of Mace [15] in that both work on aggregate arrays, rather than individual data elements. Mace gives a precise formulation of our optimisation problem in its fullest sense and shows it to be NP-complete.

Hall *et al.* [12] describe a one-pass, optimising interprocedural compilation system for Fortran D and also demonstrate the vital importance of interprocedural analysis for optimising parallelisation.

Saltz *et al.* [17] address the basic problem of how to parallelise loops where the dependence structure is not known statically. Loops are translated into an *inspector* loop which determines the dependence structure at runtime and constructs a

schedule, and an *executor* loop which carries out the calculations planned by the inspector. Saltz *et al.* discuss the possibility of reusing a previously constructed schedule, but rely on user annotations for doing so. Ponnusamy *et al.* [16] propose a simple conservative model which avoids the user having to indicate to the compiler when a schedule may be reused.

Benkner *et al.* [6] describe the reuse of parallel schedules via explicit directives in HPF+: `REUSE` directives for indicating that the schedule computed for a certain loop can be reused and `SCHEDULE` variables which allow a schedule to be saved and reused in other code sections.

Value numbering schemes were pioneered by Ershov [8], who proposed the use of "convention numbers" for denoting the results of computations and avoid having to recompute them. More recent work on this subject is described by Aho *et al.* [1].

6.3 Future work.

- The most direct next step is to store cached execution plans persistently, so that they can be reused subsequently for this or similar applications.
- Although we can derive some benefit from exploiting run-time control-flow information, we also have the opportunity to make run-time optimisation decisions based on run-time properties of data; we plan to extend this work to address sparse matrices shortly.
- The run-time system has to make on-the-fly data placement decisions. An intriguing question raised by this work is to compare this with an optimal off-line schedule.

Acknowledgments. This work was partially supported by the EPSRC, under the Futurespace and CRAMP projects (references GR/J 87015 and GR/J 99117). We extend special thanks to Fujitsu and the Imperial College / Fujitsu Parallel Computing Research Centre for providing access to their AP1000 multicomputer, and to the Imperial College Parallel Computing Centre for the use of their AP3000 machine.

References

1. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers.* Addison-Wesley, Reading, Massachusetts, 1986.
2. ASCI Red Pentium Pro BLAS 1.1e. See `http://www.cs.utk.edu/~ghenry/distrib/` and `http://developer.intel.com/design/perftool/perflibst/`.
3. Uptal Banerjee. Unimodular transformations of double loops. Technical Report TR–1036, Center for Supercomputing Research and Development (CSRD), University of Illinois at Urbana-Champaign, 1990.
4. Richard Barrett, Mike Berry, Tony Chan, Jim Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Chuck Romine, and Henk 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.

5. Olav Beckmann and Paul H J Kelly. Data distribution at run-time; re-using execution plans. To appear in Euro-Par '98, Southampton, U.K., September 1st - 4th, 1998. Proceedings will be published by Springer Verlag in the LNCS Series.
6. Siegfried Benkner, Piyush Mehrotra, John Van Rosendale, and Hans Zima. High-level management of communication schedules in HPF-like languages. Technical Report TR-97-46, Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, Hampton, VA 23681, USA, September 1997.
7. Michele Dion, Cyril Randriamaro, and Yves Robert. Compiling affine nested loops: How to optimize the residual communications after the alignment phase. *Journal of Parallel and Distributed Computing*, 38(2):176–187, November 1996.
8. Andrei P. Ershov. On programming of arithmetic operations. *Communications of the ACM*, 1(8):3–6, 1958. Three figures from this article are in CACM 1(9):16.
9. Paul Feautrier. Toward automatic distribution. *Parallel Processing Letters*, 4(3):233–244, 1994.
10. William D. Gropp. Performance driven programming models. In *MPPM'97, Proceedings of the* 3^{rd} *International Working Conference on Massively Parallel Programming Models*, London, U.K., November 1997. To appear.
11. William D Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, Cambridge, MA, USA, 1994.
12. Mary W. Hall, Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Interprocedural compilation of Fortran D. *Journal of Parallel and Distributed Computing*, 38:114–129, 1996.
13. John L. Hennessy and David A. Patterson. *Computer Architecture A Quantative Approach*. Morgan Kaufman, San Mateo, California, 1^{st} edition, 1990.
14. 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, November 1994.
15. Mary E. Mace. *Storage Patterns in Parallel Processing*. Kluwer Academic Press, 1987.
16. Ravi Ponnusamy, Joel Saltz, and Alok Choudhary. Runtime compilation techniques for data partitioning and communication schedule reuse. In *Proceedings of Supercomputing '93: Portland, Oregon, November 15–19, 1993*, pages 361–370, New York, NY 10036, USA, November 1993. ACM Press.
17. Joel H. Saltz, Ravi Mirchandaney, and Kay Crowley. Run-time parallelization and scheduling of loops. *IEEE Transactions on Computers*, 40(5):603–612, May 1991.
18. R. M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1):25–33, January 1967.