Experiments with Parallelising Numerical Applications via DESOLibraries (extended abstract)

Olav Beckmann and Paul H J Kelly

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

August 1997

Abstract

DESOLibraries are "delayed evaluation, self-optimising" parallel libraries of numerical routines. The aim is to allow users to parallelise computationally expensive parts of numerical programs by simply linking with a parallel rather than sequential library of subroutines. The library performs interprocedural data placement optimisation at runtime, which requires the optimiser itself to be very efficient. This paper outlines the techniques we use to achieve this and describes the current state of our implementation. We show performance results for an implementation of the conjugate gradient iterative solver on the AP1000 that uses our library.

1 Introduction

This short paper outlines some aspects of our approach to runtime interprocedural data placement optimisation in the context of a DESOLibrary of parallel vector-matrix routines which we are currently developing. The fundamental ideas behind our approach have been outlined in previous publications [5, 2, 3]. Briefly, we provide a library of parallel implementations for common numerical problems which can be called from any convenient top-level calling language (plain C, spreadsheets, Mathematica etc.). The problem with a naive implementation of such a library is that we do not know the sequence in which library operators will be called and hence have no scope for interprocedural data placement optimisation to avoid unnecessary redistributions. We use *delayed evaluation* to capture the dataflow at runtime and optimise the resulting dataflow graph (DFG) when a value is actually forced.

1.1 Structure of this Paper

Section 2 describes the basis of our strategy for making the optimiser sufficiently efficient to work at runtime. Next, Section 3 illustrates the optimisation problem we have to solve and looks at two issues that can complicate it. Section 4 shows performance figures for an implementation of the conjugate gradient algorithm using our library. Finally, Section 5 looks at future work, Section 6 at related work, and Section 7 concludes.

2 Basic Approach

Since we are optimising at runtime, the optimiser itself has to be very efficient. We achieve this by

- working from aggregate loop nests which have been optimised in isolation and by
- using a carefully constructed mathematical formulation for data distributions and the distribution requirements of library operators.

2.1 Data Distributions

Our formulation for data distributions is based on sing a combination of affine transformation functions (known as "alignment" directives in

HPF [6]), folding functions ("distribution" directives in HPF) and a new type of function for representing data copying.

- Affine transformation (alignment) functions act on array-index vectors $\vec{\imath}$ and take the form

$$f(\vec{\imath}) = A \cdot \vec{\imath} + \vec{t} \quad . \tag{1}$$

  If $\vec{t} = \vec{0}$, the function is fully described by the matrix A.

- *Example.*
  The affine transformation function for distributing a $(1 \times N)$ row vector `v` over the columns (i.e. the first dimension) of a mesh is

$$\begin{aligned} f\left(\begin{smallmatrix} i \\ 0 \end{smallmatrix}\right) &= A_v \cdot \left(\begin{smallmatrix} i \\ 0 \end{smallmatrix}\right) + \vec{t} \\ &= \left(\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right)\left(\begin{smallmatrix} i \\ 0 \end{smallmatrix}\right) + \left(\begin{smallmatrix} 0 \\ 0 \end{smallmatrix}\right) \\ &= \left(\begin{smallmatrix} 0 \\ i \end{smallmatrix}\right) \end{aligned} \tag{2}$$

- We can formulate affine *placement constraints* for library operators by stating relationships between the placement matrices A[1] of their operands.

- We optimise with respect to affine transformation functions.

3 Optimisation

In this section, we illustrate the optimisation problem we have to solve and discuss two issues that can complicate it.

Consider the example shown in Figure 1. As indicated in the diagram, we can write down placement constraints for operands of all library operators called: For the dot-product α `= v.y`, `v` and `y` need to be aligned in the same way, hence we have $A_v = A_y$[2]. Similarly, we have $A_B = A_x$ for `y = B * x`. However, the result `y` of the vector-matrix product is generated in transposed alignment to the input vector `x`, hence $A_x = A_y^T$. We can in analogous fashion write down placement constraints for all library operators called in a DFG.

[1] If $t \neq \vec{0}$, we also state relationships for the translation vectors t.

[2] We leave out the offset vectors $\vec{t}$ and assume they are all zero.

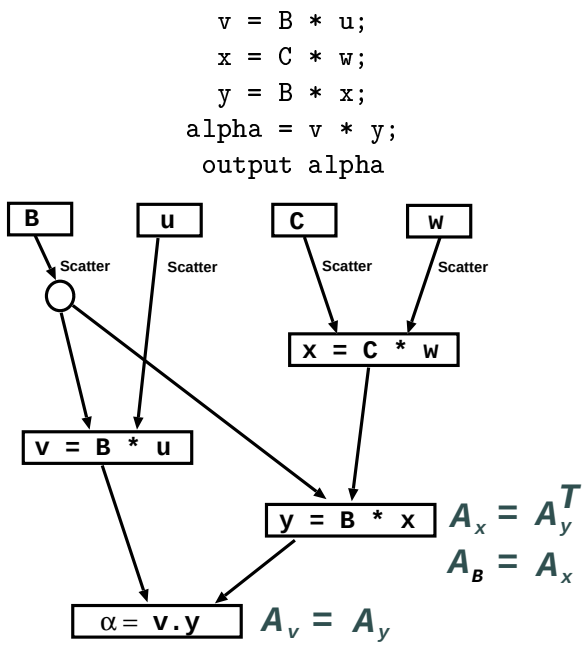

Figure 1: DFG for example optimisation problem, showing library operators' data-placement constraints.

3.1 Placement Constraints and Equation Systems

The placement constraints we have shown in Figure 1 correspond to systems of linear equations, as can be seen from the following derivation[3].

$$A_v = A_y \tag{3}$$

$$A_v - A_y = \mathbf{0} \tag{4}$$

$$\left|\begin{array}{l} \alpha_v^{00} - \alpha_y^{00} = 0 \\ \alpha_v^{01} - \alpha_y^{01} = 0 \\ \alpha_v^{10} - \alpha_y^{10} = 0 \\ \alpha_v^{11} - \alpha_y^{11} = 0 \end{array}\right| \tag{5}$$

$$\begin{pmatrix} 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} \alpha_v^{00} \\ \alpha_y^{00} \\ \alpha_v^{01} \\ \alpha_y^{01} \\ \alpha_v^{10} \\ \alpha_y^{10} \\ \alpha_v^{11} \\ \alpha_y^{11} \end{pmatrix} = \mathbf{0} \tag{6}$$

This particular equation system is obviously under-constrained.

[3] Let α_v^{ij} denote element (i, j) of matrix A_v.

The optimisation problem we need to solve is precisely equivalent to the union of all such systems of equations derived from the placement constraints. That union can be

- *under-constrained* — in which case we can avoid all redistributions and have some freedom to choose the distributions of operands,
- *over-constrained* — in which case we cannot eliminate all placement conflicts and have to perform some redistributions, or
- *have a unique solution* — in which case there is precisely one placement for all operands that will avoid having to perform redistributions.

The most interesting cases are those where the equation system is over-constrained. Here, we have to find a set of placements that minimise the cost of the redistributions we have to perform. This can no longer be solved by considering the placement constraints alone. Instead, we have to calculate what redistributions have to be performed and we require a function that will estimate the cost of such redistributions.

3.2 Shared Nodes

One situation that can result in a more complicated optimisation problem is the existence of shared nodes in a DFG. This can be illustrated from the DFG in Figure 1 if we consider what happens when the value α is forced into an identity-distribution ($A_\alpha = \left(\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right)$). We have:

$$A_\alpha = \left(\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right) \tag{7}$$

$$\left(\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right) = A_v = A_y \tag{8}$$

$$A_u = A_v^T \qquad A_x = A_y^T \tag{9}$$

$$A_B = A_u \qquad A_w = A_x^T \tag{10}$$

$$A_C = A_w \tag{11}$$

$$A_B = A_x \quad . \tag{12}$$

- `C` is not a shared node, and we can easily derive a unique placement for it:

$$A_C = A_w = A_x^T = (A_y^T)^T = A_y = \left(\begin{smallmatrix}1&0\\0&1\end{smallmatrix}\right) \tag{13}$$

- `B` is a shared node, and we have two different ways of deriving a placement for it:

$$A_B = A_x = A_y^T = \left(\begin{smallmatrix}0&1\\1&0\end{smallmatrix}\right) \tag{14}$$

or

$$A_B = A_u = A_v^T = \left(\begin{smallmatrix}0&1\\1&0\end{smallmatrix}\right) \tag{15}$$

In this case, we obtain the same result for both derivations, but this need not be the case.

The problem of shared nodes in DFGs is well understood in principle [7]. The three commonly suggested solutions to a situation where conflicting placements are obtained are as follows.

1. Choose a placement that satisfies one set of constraints. The other operators have to use the data item in a layout that does not satisfy their constraints.
2. Choose a compromise layout. This means that no operator uses that data item in its preferred placement, but the resulting "remote access" costs may be smaller than if one set of constraints had been entirely satisfied.
3. Duplicate the data item in different placements.

It seems that solutions 1 and 2 are unlikely to be efficient for distributed-memory multiprocessors because of the hight cost of "remote reads". This leaves solution 3. Mace [7] only considers the cost of generating the duplicate copy in assessing this option. It seems likely, however, that available memory will often be the true constraining factor for this option.

We will have to develop a solution that takes the amount of memory available into account.

3.3 Alternative Operator Implementations

So far, we have assumed that each operator has one implementation with one set of placement constraints, such that for example

$$\texttt{x} = \texttt{B * w} \tag{16}$$

$$\Longrightarrow \quad A_x = A_w^T$$

$$\Longrightarrow \quad \mathbf{0} = A_x - A_w^T \tag{17}$$

If, however, we have alternative implementations for operators with alternative constraints, we might obtain the following.

$$A_x = A_w^T \qquad \text{or} \qquad A_x = A_w \tag{18}$$

and

$$\left(A_x - A_w^T\right)\left(A_x - A_w\right) = \mathbf{0} \quad . \tag{19}$$

This is a non-linear system. Therefore, the presence of alternative operator implementations to choose from will significantly increase the complexity of the optimisation problem, even though it might offer redistribution-free solutions where that would not have been possible with just one implementation per operator.

4 Performance

The implementation of our library is based on MPI and runs both on the Fujitsu AP1000 at Imperial College and on workstation networks under `mpich`. We have implemented a simple (non-preconditioned) conjugate gradient [1] iterative solver for large systems of sparse linear equations using our library. The pseudo-code for the conjugate gradient algorithm can be found in Figure 2 and a slightly cut-down version of the source code of our implementation, showing our library operators, is in Figure 3. The convergence test aside, it consists of one matrix-vector product, three vector updates and 2 dot products.

Figure 4 shows performance results we have obtained for the conjugate gradient solver on the AP1000.

- Two out of three vector transpose operations are eliminated by the optimiser.
- There is a reduction of about 20% in communication time and of about 10% in overall runtime.
- The runtime overhead appears quite high, and we are working on reducing it. However, it is independent of the size of data objects and will therefore become less significant with larger problem sizes.
- The optimisation time is small and is only incurred on the first iteration.

$r^{(0)} = b - Ax^{(0)}$
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

Figure 2: Pseudo-code for the Conjugate Gradient Algorithm.

```
for(i = 1; i <= max_iter; i++) {
  if (i != 1)
    L_S_Copy(rho, &rho_pre);
  L_VV_Dot(r, r, &rho);

  L_SS_Divide(rho, rho_pre, &beta);
  L_DAXPY(beta, p, 1, r, &p);

  L_DGEMV(A, p, 0, q, &q);
  L_VV_Dot(q, p, &alpha);
  L_SS_Divide(rho, alpha, &alpha);

  L_DAXPY(alpha, p,  1, x, &x);
  L_DAXPY(alpha, q, -1, r, &r);

  L_DGEMV(A, x, -1, b, &resid);
  L_V_Norm(resid, &norm_res);
  err = L_S_Return_Value(&norm_res);
};
```

Figure 3: Source code for conjugate gradient solver using our libraries (slightly cut down).

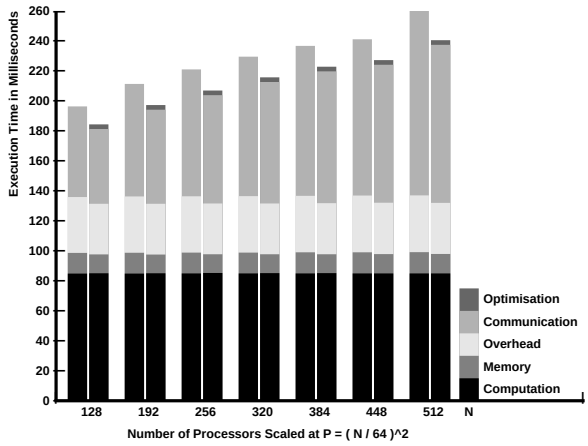

Figure 4: Performance of the conjugate gradient iterative solver on the AP1000. Performance figures are based on 10 iterations. Notice that the number of processors used is scaled with the problem size.

5 Future Work

- "Caching" of optimisation results so that we don't re-optimise in loops.
- Parallel as well as sequential composition of library operators. That means we need to worry about scheduling.
- Explore application to adaptive multigrids.

6 Related work

Our optimisation algorithm is most similar to that of Feautrier [4] in that we perform optimisation with respect to affine placement functions. The key differences are that by working with aggregate data structures and operators, we greatly reduce the complexity of the problem to be solved.

Mace [7] gives a precise formulation of our optimisation problem in its fullest sense and shows it to be NP-complete. However, a key difference with our approach is that due to our mathematical representation for data distributions and costs, we can calculate the information which Mace enumerates, thus reducing the complexity to proportions that can be handled in a runtime optimiser.

7 Conclusion

Our approach makes interprocedural data placement optimisation available for use at runtime and with any convenient top-level calling language for numerical operators, such as spreadsheets.

We capture the runtime dataflow by using lazy evaluation.

We achieve the necessary efficiency for working at runtime by

- working from aggregate loop nests and
- using an efficient mathematical representation for data distributions.

Acknowledgements

This work was partially supported by the EPSRC, under the Futurespace and CRAMP projects (refs. GR/J 87015 and GR/J 99117). We extend special thanks to Fujitsu and the Imperial/Fujitsu Parallel Computing Research Centre for providing access to their AP1000 and AP3000 multicomputers. Thanks also to Steven Newhouse of Imperial's Department of Aeronautics who very kindly allowed us access to their C++ matrix library.

References

[1] 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.

[2] Olav Beckmann and Paul H J Kelly. Automatic data distribution optimisation in a lazy, self-optimising parallel matrix library (extended abstract). In *PCW '96, Proceedings of the Sixth Parallel Computing Workshop, Kawasaki, Japan*, November 12–13 1996.

[3] Olav Beckmann and Paul H. J. Kelly. Runtime interprocedural data placement optimisation for lazy parallel libraries (extended ab-

stract), August 1997. To appear in *Euro-Par '97*, Springer-Verlag Lecture Notes in Computer Science.

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

[5] Simon Govier and Paul H. J. Kelly. A lazy, self-optimising parallel matrix library. In David N. Turner et al., editor, *Glasgow Functional Programming Workshop*, Ullapool, July 1995. Springer-Verlag.

[6] High Performance Fortran Forum. High Performance Fortran language specification, version 1.1. Technical Report CRPC-TR92225, Center for Research on Parallel Computation, Rice University, Houston, TX, November 1994.

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