Runtime Interprocedural Data Placement Optimisation for Lazy Parallel Libraries (extended abstract)

Olav Beckmann and Paul H J Kelly

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

Abstract. We are developing a lazy, self-optimising parallel library of vector-matrix routines. The aim is to allow users to parallelise certain 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. We achieve this firstly by working from aggregate loop nests which have been optimised in isolation, and secondly by using a carefully constructed mathematical formulation for data distributions and the distribution requirements of library operators, which allows us largely to replace searching with calculation in our algorithm.

1 Introduction

This paper describes an approach to interprocedural data placement optimisation in the context of a parallel numerical library. The idea for such a library, as described in our previous paper [4], is to make it easy for users to parallelise a program incrementally using parallel versions of numerical subroutines. Since, from the library implementor's point of view, we cannot analyse the user's source code, interprocedural optimisation of data distributions cannot be done at compile-time. *Lazy evaluation* is proposed as a way to do the optimisation at run-time. Rather than executing each library operation immediately, we return and store a recipe for the result that it defines. In that way, we build up a data-flow graph (DFG) for a sequence of operations. When we can delay evaluation no further, the accumulated DFG is available for devising an optimised execution plan.

An Example. Consider the simple sequence of operations shown in Figure 1. It demonstrates how in a straightforward implementation, where we scatter A and u, calculate v, then scatter B and w and calculate x, we then have to redistribute x for the third library call. If, however, we capture the information about how x is used before we execute the first two operations, we can choose a transposed layout for B and do not have to perform any redistributions. This illustrative example is simple enough that well-known compile-time analyses could achieve

```
v = A * u;
x = B * w;
y = A * x;
```

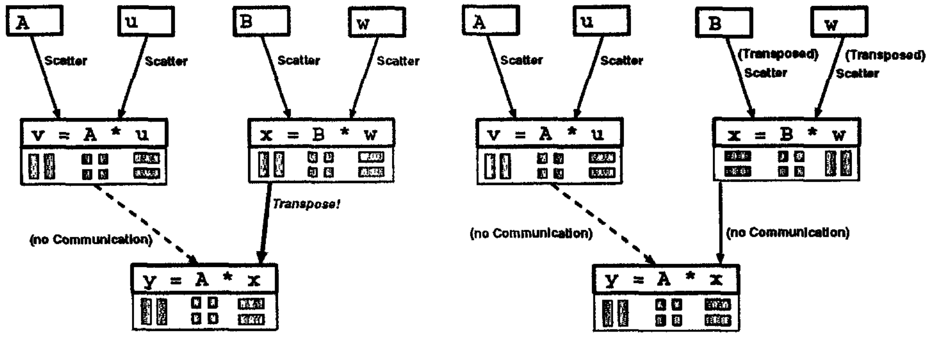

Fig. 1. Unoptimised (left) and optimised DAG for our example

this optimisation; by doing the optimisation at run-time, we are able to use any convenient calling language (such as a spreadsheet or computer algebra system), and we can optimise the actual data flow exercised for a particular problem instance.

Work described in this paper. Since we are performing optimisation at runtime, the performance of the optimiser itself is crucial. Hence, our approach has been to seek to *calculate* optimal distributions, rather than search for them. Space here does not allow a full discussion of this approach; instead, we briefly state its main theoretical components and then show some performance figures obtained with our library.

2 Theoretical Components of Our Approach

Representing Data Distributions. Our representation for data distributions allows us to calculate both any redistributions required between given distributions and new distributions that result from changes to data placements made by our optimisation algorithm.

Estimating redistribution costs. We derive a cost model from the mathematical representations for redistributions that are required in a DFG. We use this model as a 'weight' function for guiding our optimisation algorithm.

Optimising using library function distribution requirements. Our optimisation algorithm seeks to optimise the execution of a DFG by changing the distribution of some data structures in order to obtain a more compatible overall set of distributions. We follow the approach of Feautrier [3], but do this using placement constraints expressed by means of our model for data distributions. Essentially, assembling the constraints expressed by the DFG leads to a system of linear equations.

3 Performance

Our implementation is based on MPI, and has been calibrated for the 128-processor Fujitsu AP1000 at Imperial College. Figure 2 shows performance results from a simple conjugate gradient solver (transcribed from [1]). The histogram shows execution time for 10 iterations using an $n \times n$ matrix, where n ranges from 128 to 512 and the number of processors in use is scaled proportionately from 4 to 64. These preliminary results show a promising reduction in the redistribution costs due to avoiding two of the three vector transpose operations involved per iteration. The time spent by the optimiser is small, and is only incurred on the first iteration of the loop.

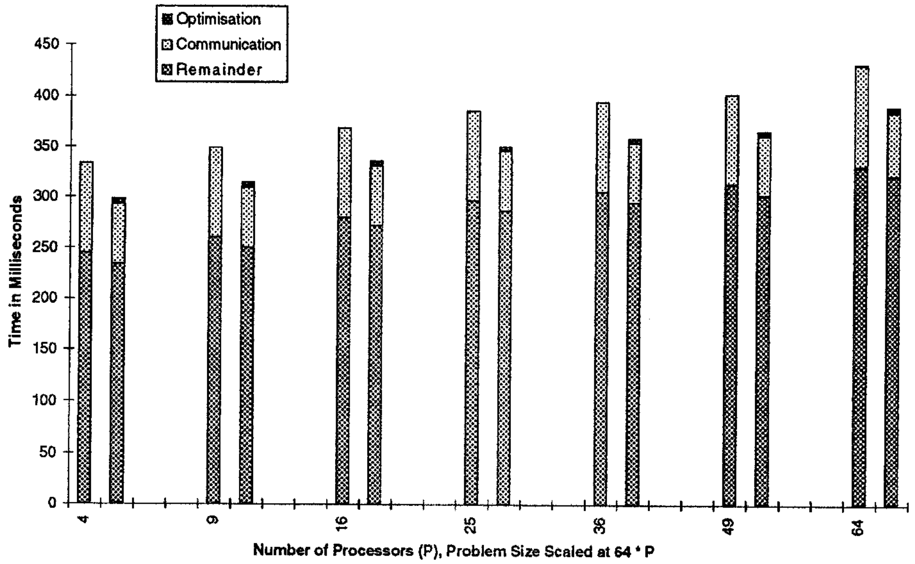

Fig. 2. Unoptimised (Left) and Optimised (Right) Performance of Conjugate Gradient Method using our Library.

4 Related work

Our optimisation algorithm is most similar to that of Feautrier [3] 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 [5] 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.

5 Conclusions

This extended abstract has introduced our approach to interprocedural data placement optimisation in a parallel numerical library: We have demonstrated that lazy evaluation can be used to expose opportunities for data distribution optimisation at run-time. For applications for which our library is suitable, there is promising evidence that much of the benefit of compile-time optimisation can be achieved without compile-time analysis of the calling program.

Although developed for optimisation of calls to manually-constructed library routines, the techniques we have developed should also be applicable to interprocedural optimisation of compiler-generated procedures in for example an HPF implementation, whether at compile-time or at run-time.

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 multicomputer. Thanks also to Steven Newhouse of Imperial's Department of Aeronautics who very kindly allowed us access to their C++ matrix library and furnished us with examples and advice.

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, Philadelphia, PA, USA, 1994.
2. Olav Beckmann. A lazy, self-optimising parallel matrix library. Master's thesis, Department of Computing, Imperial College, London SW7 2AZ, U.K., 1996.
3. Paul Feautrier. Toward automatic distribution. *Parallel Processing Letters*, 4(3):233–244, 1994.
4. 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.
5. Mary E. Mace. *Storage Patterns in Parallel Processing.* Kluwer Academic Press, 1987.