ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

David N. Turner (Ed)

Functional Programming, Glasgow 1995

Proceedings of the 1995 Glasgow Workshop on Functional
Programming,
Ullapool, Scotland, 10-12 July 1995

Paper:

A Lazy, Self-optimising Parallel Matrix Library

Simon Govier and Paul H.J. Kellyill

Published in collaboration with the
British Computer Society

©Copyright in this paper belongs to the author(s)

A Lazy, Self-optimising Parallel Matrix Library

Simon Govier and Paul H J Kelly
Department of Computing, Imperial College
180 Queen's Gate, London SW7 2BZ, UK
Email: phjk@doc.ic.ac.uk

Abstract

This paper describes a parallel implementation of a matrix/vector library for C++ for a large distributed-memory multicomputer. The library is “self-optimising” by exploiting lazy evaluation: execution of matrix operations is delayed as much as possible. This exposes the context in which each intermediate result is used. The run-time system extracts a functional representation of the values being computed and optimises data distribution, grain size and scheduling prior to execution. This exploits results in the theory of program transformation for optimising parallel functional programs, while presenting an entirely conventional interface to the programmer. We present details of some of the simple optimisations we have implemented so far and illustrate their effect using a small example.

Conventionally, optimisation is confined to compile-time, and compilation is completed before run-time. Many exciting opportunities are lost by this convenient divide. This paper presents one example of such a possibility. We do optimisation at run-time for three important reasons:

- We wish to deliver a library which uses parallelism to implement ADTs efficiently, callable from any client program (in any sensible language) without special parallel programming expertise.
This means we cannot perform compile-time analysis of the caller's source code.
- We wish to perform optimisations which take advantage of how the client program uses the intermediate values.
This would be straightforward at compile-time, but not for a library called at run-time.
- We wish to take advantage of information available only at run-time, such as the way operations are composed, and the size and characteristics of intermediary data structures.

We aim to get much of the performance of compile-time optimisation, possibly more by using run-time information, while retaining the ease with which a library can be installed and used. There is some run-time overhead involved, which limits the scope of the approach.

1 Background: parallelism via libraries

Software engineers in computational science and engineering are not keen to invest in substantial restructuring of their applications in order to improve performance on parallel computers. Fully-automatic parallelisation is obviously an ideal, but an attractive alternative is to use a parallel library. This way, the complexity of using parallelism can be avoided completely. Most of the application need not be changed at all, and, assuming proper inter-language working standards, the user can use any language and any compiler.

Host-cell data movement

A parallel program built this way has a characteristic structure: there is a single, master processor (“host”) which runs the user's program, and a set of worker processors (“cells”) which are involved only in parallel operations.

Interesting parallel library functions generally manipulate large data structures as operands and/or results, such as matrices, graphs, relations etc. When a parallel operation on such an ADT is called, the operand data must be partitioned and distributed to the cells. When the operation has finished, the host can assemble the result by combining fragments from each cell.

Redundant data movement

If the host needs to assemble the result matrix in order to print it, or to perform some test which will determine control flow, then perhaps this assembly stage is absolutely necessary. Very often, though, the result of a library operation is soon passed to another operator belonging to the same library.

If this, too, is a parallel operation, then it would have been better to delay assembling the result, in case it turns out to be needed in distributed form by the operation which uses it.

The composition problem

The central problem with parallel libraries is that there is no opportunity to eliminate this redundant data movement at compile-time.

In fact, further optimisation could reduce data movement even more, for example by selecting parallel implementations of the operations used so that data is generated where it will be needed by the operation which follows.

The skeleton approach

One approach to this problem, advocated for example in [Darlington et al., 1993a] and [Darlington et al., 1993b], is to parameterise parallel library functions with user code. Higher-order parallel functions, called “skeletons”, capture general algorithmic structures. In some cases at least, the library implementor can guarantee good parallel performance whatever the user code’s characteristics (this is claimed, for example, in [Cole, 1989]).

In general, it doesn’t make sense to parameterise a skeleton with client code which itself employs a skeleton. A poor schedule results if both skeletons run in ignorance of one another. However, skeletons can be combined explicitly using higher-level skeleton composition operators. These combinators can be used to capture precisely how intermediate values are passed from skeleton to skeleton.

In practice, what this means is that the outermost control structure of the application has to be rewritten in a functional “skeleton composition language” (SCL):

- Most of user’s code is unchanged, encapsulated in (i.e. called from) SCL.
- SCL provides high-level control structure + distributed data structures.
- SCL controls interfaces between fragments of reused user code.

This idea of building applications from reused components has software engineering advantages of course (as observed, for example, in [Magee and Dulay, 1991]). It also offers the opportunity to do inter-skeleton optimisations, for example to minimise data movement and improve schedules. This has been the goal of a number of recent efforts, including [Harrison, 1992], [Lucco and Sharp, 1991], [Peiro, 1994] and perhaps most notably the P³L compiler [Danelutto et al., 1992]. A compiler for a program built this way

- uses dependence information from SCL *only*: no need to analyse encapsulated code.
- uses performance models for target architecture and skeleton implementation schemes to derive optimal parallel implementation.

The optimisation is taken care of by the SCL compiler, and performance tuning is handled automatically.

The objective of this work

The problem is that skeletons don't do what users want. Although the promised automatic performance tuning would be very valuable, it remains unattractive for some users if it can only be achieved by rewriting the outer control structure of the application in a functional language. In this paper we outline a scheme which uses essentially the same optimisation and program transformation techniques, but the library functions are *called by* the user code. This makes it easy to parallelise an application in an incremental fashion. We aim to gain the same performance advantage.

2 A run-time approach to the composition problem

We illustrate this idea using a parallel matrix library as an example. Our work has been based on C++, which is convenient because of its overloading. In principle any client language could have been used, since there is no compile-time processing.

Lazy data movement

The first inefficiency to dispose of is to avoid unnecessary movement of matrices to and from the host. To do this, we represent each matrix by an opaque “handle”. The run-time system keeps track of whether the matrix is stored on the host or in the cells (or both), and if it is stored on the cells it also records how it is distributed.

This way, a matrix result need never be assembled on the host. If the host program does refer to matrix elements individually (for example to print the matrix), communication will be required, and it may be more efficient to collect blocks of the matrix to reduce the number of message exchanges needed.

If the distribution of a matrix happens to coincide with the way the following operation needs it, then data movement is avoided. If the data is held in the cells, but in some other distribution, the cells communicate among themselves to get the components they need. We return to this issue in section 4.

Lazy evaluation

We can often avoid unnecessary data movement by deciding the distribution of operands on the basis of how the results of an operation are to be used. To gain the opportunity to do this, we delay execution of library calls as long as possible.

This is not apparent to the calling program, since the representation of the matrix is hidden behind the opaque “handle”: we choose to represent the value in terms of the operations involved in computing it, that is as a symbolic expression DAG.

Evaluation cannot be delayed indefinitely:

- The matrix may be used in an operation which produces a non-opaque result, such as a scalar (e.g. the matrix is required on the host for printing or testing).

We refer to both these cases as “strict” contexts.

- The expression DAG representing the value may become too large.

Dependences and name management

There is a problem with delaying evaluation: we are working with an imperative language with assignment. Isn't there a danger that the operands will have changed by the time we actually do the calculation?

This problem, essentially that of respecting anti- and output-dependences, i.e. write-after-read (WAR) and write-after-write (WAW) hazards, is solved using a simple technique analogous to the register renaming found in [Tomasulo, 1967]. Each operation returns a new handle, referring to a distinct matrix result.

Consider the example shown in Fig. 1. We have used overloading and templates in C++ to provide a concise and flexible syntax which is hopefully self-explanatory:

```

#include <matrix.h>
#include <complex.h>

void main()
{
    matrix<complex<double>> a(500,500);
    matrix<complex<double>> b(500,500);
    matrix<complex<double>> c(500,500);

S1:    infile >> a;        // read a from file
S2:    infile >> b;        // read b from file
S3:    c = a * b;
S4:    a = a * 2;
S5:    outfile << a;       // this forces evaluation of matrix a
S6:    outfile << c;       // this forces evaluation of matrix c
}

```

Figure 1: Evaluation of c can be delayed even though a is redefined.

- In statement S1, the host reads the matrix and stores it on its heap. Client variable a contains a handle, α , which refers to it.
Statement 2 is similar, storing handle β in b .
- In S3, the matrix-matrix multiply operator returns a handle, say γ , in unevaluated form, and it is γ which is stored in c .
The handle γ points to a DAG node indicating that the value is to be computed by multiplying the matrices referred to by handles α and β .
- In S4 the matrix-scalar add operator returns another handle, δ , representing the addition of 2 to each element of the matrix referred to by handle α . Variable a holds the new handle δ , but the matrix referred to by α is still available.
- In S5, evaluation of handle δ is forced.
- in S6, evaluation of handle γ is forced, picking up α and β despite a 's new value.

Garbage collection

The renaming idea relies on keeping handles in case the client program should refer to them again. In general we have no control over what the client does with handles, which can be stored and copied at will. We therefore have a problem reclaiming the space occupied by the matrices or recipes they refer to.

We do not have a general solution to this problem: we need a garbage collector. In C++, however, it is not hard to keep reference counts by exploiting the constructor/destructor mechanism, and this is adequate since there will be no cycles (the expression graphs are guaranteed to be acyclic because each expression node built can refer only to pre-existing nodes).

3 Implementation

Using a pre-existing matrix library for C++ as a model, we added templates and overloading, and modified it to return handles referring to values represented in evaluated or unevaluated form. We implement a number of trivial simplifications, such as exploiting distributivity, associativity, identities and eliminating common subexpressions. Lazy

evaluation automatically eliminates work arising from dead code. Conveniently, doing common-subexpression elimination at run-time also catches loop-invariant expressions - since the same expression is built at each loop iteration.

Data placement optimisation

If two matrices are to be added, they should have the same distribution, but there are no restrictions on what that should be. If they are to be multiplied, there are various alternative implementations (e.g. a twisted layout, [Tatsuya Shindo et al., 1994]). For our preliminary implementation we took a simple scheme: consider

```
void main()
{
    matrix<complex<double>> a(500,500);
    matrix<complex<double>> b(500,500);
    matrix<complex<double>> c(500,500);

    c = a * b;
}
```

We require the matrix *a* to be distributed across the *N* processors row-wise, in blocks, and *b* to be distributed column-wise, in blocks. We can then form the product in *N* steps in one of two ways. We can either compute the result with a row-wise distribution (by shifting *b* right *N* times), or columnwise (by shifting *a* down *N* times). Which of the two implementations we choose depends on our preferred result distribution. Similarly, if the operands are distributed columnwise and row-wise instead of row-wise and columnwise, we can select a variant of the matrix multiply algorithm which accepts its operands in transposed form and produces a transposed result.

The placement requirements of each assignment can be assembled into a system of constraints of essentially the following form:

```
fits(var(C,DistC),add_mm,var(A,DistA),var(B,DistB),Implementation) :-
    Implementation = add_mm_aligned,
    DistC = DistA = DistB.
```

```
fits(var(C,DistC),mul_mm,var(A,DistA),var(B,DistB),Implementation) :-
    Implementation = mul_mm_colwise,
    DistC = colwise,
    DistA = rowwise,
    DistB = colwise.
```

```
fits(var(C,DistC),mul_mm,var(A,DistA),var(B,DistB),Implementation) :-
    Implementation = mul_mm_rowwise,
    DistC = rowwise,
    DistA = rowwise,
    DistB = colwise.
```

The predicate

```
fits(var(c,distc),mul_mm,var(a,dista),var(b,distb),implementation)
```

is satisfied if, in the triple $c = a \text{ op } b$, the function *implementation*, which performs the operation *op*, can accept its operands *a* and *b* with distributions *dista* and *distb* respectively, and produces its result with distribution *distc*.

Note that values loaded into the cells from the host (e.g. because they have been read from a file) can be distributed in any convenient way, as can matrices which will be passed only to the host.

If the placement constraints cannot be satisfied, some data redistribution will have to be introduced. To do this properly, the time taken to do redistribution should be weighed against the performance of operator implementations which work with non-optimal data distribution.

4 The “cache”

On the cell side, we have implemented a run-time system which enables any operator implementation to be used, even if its operands are not distributed as it requires. We allow matrices to be distributed only in strictly limited ways. Each matrix is distributed over the processor mesh according to a simple rule characterised by a five-tuple, which constitutes a “map” indicating where each element of each handle’s matrix can be found.

This is to be contrasted with the directory mechanism in a conventional multiprocessor’s cache, which provides placement information on a cacheline basis.

Furthermore, thanks to the renaming trick, matrices are never updated in place, so no coherency problem arises.

5 Example: Biconjugate Stabilised Iterative Solver

To evaluate the potential of the approach we have been studying some existing applications, and we present one such example here. Fig. 2 shows a C++ implementation of a matrix solver described in [Van der Vorst, 1992] which is essentially unmodified (some type declarations have been changed), and is used for acoustic analysis of underwater structures [Jeans, 1994].

Naive data distribution

If we evaluate each intermediate immediately, instead of using lazy evaluation, we can only use information about the distribution of the operands to decide the preferred operator implementation.

Optimised data distribution

The scope for optimisation in this example is limited unless lazy evaluation is used for vector/matrix operators which return a scalar result. If this is done:

- At line S1, a suspended expression representation of the expression $\mathbf{r} \cdot \mathbf{h}$ is built.
- At line S2, a suspended expression representation of the expression $(\mathbf{r} \cdot \mathbf{h} / \mathbf{old_r} \cdot \mathbf{h}) * (\alpha / \omega)$ is built. This is potentially very inefficient, since with scalar arithmetic, building a suspended expression takes longer than doing the calculation. The advantage is that it allows us to delay deciding on where to place the vectors $\mathbf{r} \cdot \mathbf{h}$ and $\mathbf{r}$ until later.
- Like S1 and S2, statements S3–S12 are delayed.
- At statement S13, the host processor must evaluate $\mathbf{err}$ to decide whether another iteration is needed. By delaying evaluation to this point, we can see how each intermediate value is to be used so we can optimise its placement.
- Note that nowhere is evaluation of the vector $\mathbf{x}$ forced. With each iteration, a larger suspended representation is built up. This need not become a problem: the run-time system must simply force evaluation before it runs out of free memory.

By delaying evaluation, we can choose initial distributions for the arguments of the procedure, choose compatible distributions for intermediate values, and we can select efficient implementations of the broadcast and reduction operations.

6 Evaluation

This paper reports preliminary work. We have yet to run substantial example applications in parallel. We have, though, experimented with the lazy evaluation mechanism using some fairly realistic examples.

```

Vector BICGSTAB::solve(const matrix<complex<double>>& A,
                      vector<complex<double>>& start,
                      vector<complex<double>>& b)
{
    vector<complex<double>> x(start.size()); // initialise elements to 0 + 0i
    vector<complex<double>> p(x.size());     // initialise elements to 0 + 0i
    vector<complex<double>> v(x.size());     // initialise elements to 0 + 0i
    vector<complex<double>> temp, rh, s, t, r;
    complex<double> err, old_err, rho, old_rho, omega, alpha, beta;

    for (i=0;i<start.size();i++) {
        x(i) = start(i);
    }

    i      = 0;
    r      = b - (A)*x;
    rh     = duplicate(r);
    rho    = complex(1.0, 0.0);
    alpha  = complex(1.0, 0.0);
    omega  = complex(1.0, 0.0);
    err    = r*r;

    do {
        i++;
        old_err = err;
        old_rho = rho;
S1:  rho      = rh % r; // NB scalar result
S2:  beta    = (rho/old_rho)*(alpha/omega); // NB scalar calculation
S3:  temp    = p - (omega*v); // lazy
S4:  p       = r + (beta * temp); // lazy
S5:  v       = A * p; // lazy
S6:  alpha   = rho / (rh % v); // NB scalar result
        // (eval'n would force v, p, temp)
S7:  s       = r - (alpha * v); // lazy
S8:  t       = A * s; // lazy
S9:  omega   = (t % s)/(t % t); // NB scalar result
        // (eval'n would force t, s and r)
S10: x      += (alpha * p)+(omega * s); // lazy
S11: r      = s - (omega * t); // lazy
S12: err    = r % r; // NB scalar result
        // (eval'n would force r)
S13: if (abs(err)<tolerance) // value of err actually used
        break; // (forces r,s,t,alpha,rho,rh,v,p,temp,beta,omega).
    } while (i!=_mit); // (maximum number of iterations to complete)

    return x;
}

```

Figure 2: Biconjugate stabilised iterative solver

As expected, there are real optimisation opportunities, but they are limited by the presence of strict operators. This effect can be reduced by using lazy evaluation for library operators which return scalars: a common example of this is computing a convergence test.

It is not uncommon for some matrix variables to be updated at each iteration of a loop, but not inspected by a strict operator until loop exit. In this case, we have to force evaluation just to manage space properly. Ideally we would observe this effect before a huge unevaluated expression is accumulated.

We have not been able to evaluate how much the time spent optimising at run-time influences overall performance. This may impact applications involving small matrices and warrants attention.

Although we found cases where data placement optimisation is important, we did not find opportunities for standard optimisations like common-subexpression elimination. This is not surprising, since we started with fairly carefully-written code.

7 Discussion

Further optimisations

We have not yet experimented with “stencil”-type computations, as arise for example in computational fluid dynamics. This will further exercise our data placement optimisation techniques.

The range of possible matrix distributions is somewhat restrictive; we should investigate alternatives. This has a subtle interaction with the tractability of the placement optimisation process. An important technique for reducing the time spent optimising is to cache the results from previous optimisations of the same expression. This would be easy to add. Of course, having executed a particular implementation plan once, it should be possible to use the results from monitoring to construct an even better implementation. This is not obviously interesting with highly-regular structures like our dense matrices but should be more so when we look at richer data structures.

Other data types

The obvious next step is to investigate other bulk data types. Natural examples include various species of sparse matrix, perhaps encompassing irregular meshes. Here the characteristics of the mesh (and especially load balance) are only known at run-time, so we can hope to do more powerful optimisations using this information.

Related work

The performance advantages of using high-level libraries such as BLAS3 for matrix operations has long been recognised, and have become more important with more sophisticated computer architectures [Gallivan et al., 1988]. Compile-time approaches to minimising data movement have also been studied in some depth, both in the parallelising compiler community (see, for example, [Torrellas et al., 1990]), and in higher-level languages (e.g. [Blelloch, 1994], [Danelutto et al., 1992]). As is observed in [Knobe et al., 1990], the problem of finding a good placement and schedule for a given program is computationally very hard, and further research is needed to identify good heuristics. As indicated in the introduction, there has been a great deal of interest in functional “skeletons” for which automatic optimisation of data placement ought to be possible; further examples include [Südholz, 1994] and [Kuchen et al., 1994], who propose interesting ideas for avoiding the need for multiple copies of data. Run-time optimisation has been used in languages which are traditionally interpretive, and in particular, optimisation of compositions of vector/matrix operators has been used in compilers for APL [Guibas and Wyatt, 1978].

8 Conclusions

We have presented preliminary results of our investigation of the idea of self-optimising lazy libraries. There is evidence of promise: the overheads are manageable when the matrices are large, the optimisations can deal with tricky placement issues without user intervention. Careless use of strict operators can destroy the advantage, and this may have to be explained to users with some care.

We are actively seeking application areas where run-time information is important to good data distribution, and we have identified some problems in geographical databases.

We have failed, so far, however, to find a use for the more powerful program transformation techniques, as described, for example, in [Harrison, 1992] and [Skillicorn and Cai, 1993].

Acknowledgements

This work was partially supported by the EPSRC, under the Compaqt and Futurespace projects (refs. GR/J 14448 and GR/J 87015). 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

- [ICS, 1994] (1994). *International Conference on Supercomputing*. ACM Press.
- [Blelloch, 1994] Blelloch, G. E. (1994). NESL: A nested data-parallel language (version 3.0). Technical Report CMU-CS-94 (Draft), Carnegie Mellon University.
- [Bode et al., 1993] Bode, A., Reeve, M., and Wolf, G. (1993). *PARLE'93: Parallel Architectures and Languages Europe*. Springer Verlag. LNCS 694.
- [Cole, 1989] Cole, M. (1989). *Algorithmic Skeletons: Structured Management of Parallel Computation*. Pitman/MIT Press.
- [Danelutto et al., 1992] Danelutto, M., Di Meglio, R., Orlando, S., Pelagatti, S., and Vanneschi, M. (1992). A methodology for the development and the support of massively parallel programs. *Future Generation Computer Systems*, 8:205–220.
- [Darlington et al., 1993a] Darlington, J., Field, A., Harrison, P., Kelly, P., Sharp, D., Wu, Q., and While, R. (1993a). Parallel programming using skeleton functions. In [Bode et al., 1993, 146–160].
- [Darlington et al., 1993b] Darlington, J., Ghanem, M., and To, H. W. (1993b). Structured parallel programming. In *Programming Models for Massively Parallel Computers*, pages 160–169. IEEE Computer Society Press.
- [Gallivan et al., 1988] Gallivan, K., Jalby, W., and Sameh, U. M. A. (1988). The impact of hierarchical memory systems on linear algebra algorithm design. *International Journal of Supercomputing Applications*, 2(1).
- [Guibas and Wyatt, 1978] Guibas, L. J. and Wyatt, D. K. (1978). Compilation and delayed evaluation in APL. In *Conference record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 1–8. ACM Press.
- [Harrison, 1992] Harrison, P. (1992). A higher-order approach to parallel algorithms. *Computer Journal*, 35(6):555–566.
- [Jeans, 1994] Jeans, R. (1994). C-SCAT: A robust package for large scale underwater acoustic analysis. Technical report, Dept. Aeronautics, Imperial College, London.
- [Knobe et al., 1990] Knobe, K., Lukas, J. D., and Steele, G. L. (1990). Data optimization: Allocation of arrays to reduce communication on simd machines. *Journal of Parallel and Distributed Computing*, 8:102–118.
- [Kuchen et al., 1994] Kuchen, H., Plasmeijer, R., and Stoltze, H. (1994). Distributed implementation of a data parallel functional language. In Halatsis, C., Maritsas, D., Philokyprou, G., and Theodoridis, S., editors, *PARLE 94 Parallel Architectures and Languages Europe, Athens, July 1994*, volume 817 of *Lecture Notes in Computer Science*, pages 464–477, Berlin. Springer-Verlag.

- [Lucco and Sharp, 1991] Lucco, S. and Sharp, O. (1991). Parallel programming with coordination structures. In *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages*, pages 197–208, Orlando, FL.
- [Magee and Dulay, 1991] Magee, J. and Dulay, N. (1991). A configuration approach to parallel programming. In *PARLE 91 : Parallel Architectures and Languages Europe, vol 2: Parallel Languages*, pages 313–330. Springer Verlag. LNCS 506.
- [Peiro, 1994] Peiro, A. (1994). Parallel programming using composition of skeleton functions. Technical report, Department of Computer Science, University of Manchester.
- [Skillicorn and Cai, 1993] Skillicorn, D. B. and Cai, W. (1993). A cost calculus for parallel functional programming. *J. Parallel and Distributed Computing*. Also appears as Department of Computer Science Technical Report 93-348.
- [Südholt, 1994] Südholt, M. (1994). Data distribution algebras — a formal basis for programming using skeletons. In Olderog, E.-R., editor, *Programming Concepts, Methods and Calculi*, pages 19–38. North-Holland.
- [Tatsuya Shindo et al., 1994] Tatsuya Shindo et al. (1994). Twisted data layout. In [ICS, 1994].
- [Tomasulo, 1967] Tomasulo, R. (1967). An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11:25–33.
- [Torrellas et al., 1990] Torrellas, J., Lam, M. S., and Hennessy, J. (1990). Shared data placement optimizations to reduce multiprocessor cache miss rates. In *International Conference on Parallel Processing, Pennsylvania State University, August*, pages 266–270.
- [Van der Vorst, 1992] Van der Vorst, H. A. (1992). A fast and smoothly converging variant of VI-CG for the solution of non-symmetric linear systems. *SIAM Journal of Science and Statistical Computing*, 13:631–644.