Delayed Evaluation, Self-Optimising Software Components as a Programming Model

Peter Liniker, Olav Beckmann, and Paul H J Kelly

Department of Computing, Imperial College,
180 Queen's Gate, London SW7 2BZ, United Kingdom
{pl198,ob3,phjk}@doc.ic.ac.uk

Abstract. We argue that delayed-evaluation, self-optimising scientific software components, which dynamically change their behaviour according to their calling context at runtime offer a possible way of bridging the apparent conflict between the quality of scientific software and its performance. Rather than equipping scientific software components with a *performance interface* which allows the caller to supply the context information that is lost when building abstract software components, we propose to recapture this lost context information at runtime. This paper is accompanied by a public release of a parallel linear algebra library with both C and C++ language interfaces which implements this proposal. We demonstrate the usability of this library by showing that it can be used to supply linear algebra component functionality to an existing external software package. We give preliminary performance figures and discuss avenues for future work.

1 Component-based Application Construction

There is often an apparent conflict between the quality of scientific software and its performance. High quality scientific software has to be easy to re-use, easy to re-engineer, easy to maintain and easy to port to new platforms, as well as suited to the kind of thorough testing that is required for instilling confidence in application users. Modern software engineering achieves these aims by using abstraction: We should only have to code one version of each operation [16], independently of the context in which it is called or the storage representation of participating data. The problem with this kind of abstract, component-based[1] software is that abstraction very often blocks optimisation: the fact that we engineer software components in isolation means that we have no context information available for performing certain types of optimisation.

Performance Interfaces. One common solution to this problem is to equip software components with a *performance interface* that allows a calling program to tune not only those parameters that affect the semantics of a component, but also those that affect performance. One example for this might be PBLAS [5]: The P_DGEMV parallel matrix-vector product routine takes 19 parameters, 3 of which are themselves arrays of 9 integers. This compares with 11 parameters for the equivalent sequential routine

---

[1] In this paper we use the term *component* to refer to separately deployable units of software reuse, including *e.g.* subroutines from libraries like the BLAS [4].

from BLAS-2 [4]. The additional parameters in PBLAS are used to select parallel data placement. Thus, when a calling program contains a series of PBLAS routines, these parameters can be used to choose a set of data placements that minimise the need for redistributions between calls. Assuming that the application programmer knows what the optimal data layout is, the performance interface solution is of course "optimal". However, calling routines with such large numbers of parameters is very tedious and highly likely to induce programming errors. Furthermore, selecting optimal data placements is often an NP-hard problem [14], so expecting application programmers to make the right choice without access to suitable optimisation algorithms is unrealistic.

1.1 Background: Related Work

Code-Generating Systems. Several systems have been described that automatically adapt numerical routines to new computer architectures: PHiPAC [3] uses parameterised code generators and search scripts that find optimal parameters for a given architecture to generate matrix multiply routines that are competitive with vendor libraries. ATLAS (automatically tuned linear algebra software) [19] uses code generators to automatically adapt the performance-critical BLAS library [4] to new architectures.

Telescoping Languages. The telescoping languages work [13] is in some aspects similar to code-generating systems discussed above; however, the aim is not to optimise individual routines to exploit machine architectures, but rather to optimise library routines according to the context in which they are called. The strategy is to exhaustively analyse a library off-line, generating specialised instances of library routines for different calling contexts. This is combined with a language processor that recognises library calls in user programs and selects optimised implementations according to context. This work is currently still very much in-progress.

Template Meta-programming. Generic Programming techniques in C++ have been used for example in MTL [16]: each algorithm is implemented only once as an abstract template, independently of the underlying representation of the data being accessed. Optimisation in this framework is achieved by using C++ effectively as a two-level language, with the template mechanism being used for partial evaluation and code generation [18]. However, as pointed out by Quinlan *et al.* [15], a serial C++ compiler cannot find scalable parallel optimisations. A further possible problem with this technique is that templates make heavy demands of C++ compilers which on at least some high-performance architectures are much less developed than C or Fortran compilers.

Incorporating Application Semantics into Compilation. Many library-based programming systems effectively provide programmers with a semantically rich meta-language. However, this meta-language is generally not understood by compilers, which means that both syntactic checking and optimisation of the meta-language are impossible. MAGIK [9] is a system that allows programmers to incorporate application-specific semantics into the compilation process. This can be used for example in specialising remote procedure calls or in enforcing rules such that application programs should check

the return code of system calls. A related system, ROSE [15], is a tool for generating library-specific optimising source-to-source preprocessors. ROSE is demonstrated through an optimising pre-processor for the P++ parallel array class library.

1.2 Delayed Evaluation, Self-Optimising (DESO) Libraries

Our approach is to use delayed evaluation of software components in order to re-capture lost context information from within the component library at runtime. While execution is being delayed, we can build up a DAG (directed acyclic graph) representing the *data flow* of the computation to be performed [1]. Evaluation is eventually forced, either because we have to output result data, or because the control-flow of the program becomes data dependent (in conditional expressions).[2] Once execution is forced, we can construct an optimised execution plan at runtime, automatically and transparently changing the behaviour of components according to calling context.

We have implemented a library of delayed evaluation, self-optimising routines from the widely used set of BLAS kernels. The library performs cross-component data placement optimisation at runtime, aiming to minimise the cost of data redistributions between library calls. Our library has both a C language interface, which is virtually identical to the recently proposed C bindings for BLAS [4], and a C++ interface. The C++ interface uses operator overloading to facilitate high-level, generic coding of algorithms. This paper is accompanied by a public release of this library [7].

Contributions of this Paper. We have previously described the basic idea behind this library [1, 2]. The distinct contributions of this paper are as follows:

1. We demonstrate the *usability* of our approach by showing how a number of common iterative numerical solvers can be implemented in a high-level, intuitive manner using this approach.
2. We show that the C++ interface, which we have not previously described, implements the API required for instantiating the algorithm templates in the IML++ package by Dongarra *et al.* [8].
3. We give performance figures for four iterative solver algorithms from the IML++ package, which show that fairly good parallel performance can be obtained by simply using our library together with an existing generic algorithm.
4. We discuss the techniques used in implementing the C++ interface to our library.

2 Usability and Software Quality

One of the main requirements for a high-level parallel programming model is that it should be easy for application programmers to implement scientific algorithms in parallel. IML++ by Dongarra *et al.* [8] provides generic C++ algorithms for solving linear systems using a variety of iterative methods. Figure 1 (left) shows the generic IML++ code for the preconditioned biconjugate gradient algorithm. Note that this C++ code

---

[2] We show examples of both kinds of force points in Section 2.

```
 1  template < class Matrix, class Vector,          1  #include <ParDeso.h++>
 2          class Preconditioner, class Real >      2  #include "include/bicg.h" // IML++ BiCG template
 3  int BiCG( const Matrix &A, Vector &x,            3
 4          const Vector &b, const Precond &M,       4  int main( int argc, char * argv[] ) {
 5          int &max_iter, Real &tol) {              5    int SZ, max_iter;
 6    Vector rho_1(1), rho_2(1), alpha(1), beta(1);  6    int result = -1;       // CG return code
 7    Vector z( x.size() ), ztilde( x.size() );      7
 8    Vector p( x.size() ), ptilde( x.size() );      8    deso::initialise (&argc, &argv);
 9    Vector q( x.size() ), qtilde( x.size() );      9
10    Vector r( x.size() ); r = b - A * x;          10    SZ       = atoi(*(++argv));
11    Vector rtilde( x.size() ); rtilde = r;        11    max_iter = atoi(*(++argv));
12    Real resid, normb; normb = norm( b );         12
13    // Omitted check whether already converged    13    // Create and read in matrix
14                                                  14    Matrix<double> A( SZ, SZ );
15    for( int i = 1; i <= max_iter; i++ ) {        15    deso::fileRead ( A, "filename_A" );
16      z = M.solve(r);                             16
17      ztilde = M.trans_solve(rtilde);             17    // Create rhs and solution vectors
18      rho_1(0) = dot(z, rtilde);                  18    Vector<double> b( A.xsize() );
19      // Omitted check for breakdown              19    Vector<double> x( A.ysize() );
20      if (i == 1) {                               20    deso::fileRead ( b, "filename_b" );
21        p = z;                                    21    deso::fileRead ( x, "filename_x" );
22        ptilde = ztilde;                          22
23      }                                           23    // Create identity preconditioner
24      else {                                      24    DiagPreconditioner<double> I( SZ, "I");
25        beta(0) = rho_1(0) / rho_2(0);            25
26        p = z + beta(0) * p;                      26    // Convergence tolerance
27        ptilde = ztilde + beta(0) * ptilde;       27    double tol = (50.0 * DBL_EPSILON );
28      }                                           28    Scalar<double> err( tol );
29      q = A * p;                                  29
30      qtilde = A.trans_mult(ptilde);             30    deso::startTimer ();
31      alpha(0) = rho_1(0) / dot(ptilde, q);      31
32      x += alpha(0) * p;                         32    result = BiCG( A, x, b, I, max_iter, err );
33      r -= alpha(0) * q;                         33
34      rtilde -= alpha(0) * qtilde;               34    tol = deso::returnValue ( err );
35                                                 35
36      // DESO++: Need to force evaluation of x   36    deso::stopTimer ();
37      deso::evaluate ( x );                      37
38                                                 38    if( deso::isController ( ) ) {
39      rho_2(0) = rho_1(0);                       39      printf ( "\nFinal tolerance: %.10f.\n", tol );
40      if ((resid = norm(r) / normb) < tol) {     40    }
41        tol = resid; max_iter = i; return 0;     41
42      }                                          42    deso::printTime (SZ);
43    }                                            43
44                                                 44    deso::finalise ();
45    tol = resid; return 1;                       45
46  }                                              46    return (result == 1 ? 0 : -1);
                                                   47  }
```

Fig. 1. IML++ Preconditioned BiConjugate Gradient template function (left), together with calling program (right).

is almost as high-level as pseudocode, the only likely difference being various type declarations. We believe that the API defined by IML++ satisfies the requirements of being easy-to-use, high-level and abstract. Since the C++ `BiCG` function is a templated (generic) function, it has to be instantiated with a `Matrix`, `Vector`, `Precond` and `Real` class in order to be called. The template function implicitly defines the API these classes need to implement, such as overloaded operators for vector-matrix computations.

DESO++, the C++ interface for our delayed evaluation, self-optimising linear algebra library, provides parallel matrix, vector, scalar and preconditioner types that implement the API required for instantiating IML++ template algorithms. Figure 1 (right) shows an executable parallel program which is obtained by instantiating the `BiCG` template. This demonstrates:

– A parallel BiCG solver can be implemented simply by creating DESO++ objects for initial matrices and vectors, choosing a DESO++ preconditioner and then calling the IML++ template.
– Note that each operator in the BiCG template will call a delayed evaluation parallel function, building up a DAG representing the computation to be performed. Execution is *forced* either transparently on conditionals, such as the convergence test
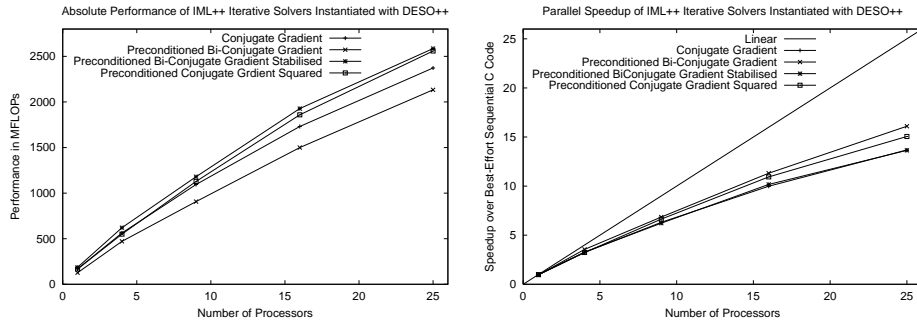
Fig. 2. Performance of four different parallel iterative algorithms implemented using IML++ templates with DESO++. The platform is a (heterogeneous) cluster of AMD Athlon processors with 1.0 or 1.4 GHz clockspeed, 256 KB L2 cache, 256 or 512 MB RAM, running Linux 2.4.17 and connected via a switched 100Mbit/s ethernet LAN. The problem size in each case is a dense matrix of size 7200×7200. The left graph shows absolute performance in MFLOP/s, the right graph shows speedup over a handwritten sequential C-language version of the same algorithm.

in line 40, or explicitly by using the `deso::evaluate` function. The latter can be seen in line 37. The reason why we have to manually force evaluation of the solution vector x here is because the control flow of the program never directly depends on x. Alternatively, we could wait until function exit when x would normally be written to disk, which would also force evaluation.

IML++ was written with the aim of being usable with a diverse range of vector and matrix classes. Since the code we instantiated required virtually no changes, we believe that our parallel library should be suitable for transparently parallelising a range of existing applications that currently rely on sequential vector and matrix classes written in C++ to implement an API similar to IML++.

3   Performance

We have implemented four different iterative solvers in parallel in the manner shown in Section 2: Conjugate Gradient, preconditioned Bi-Conjugate Gradient, preconditioned Bi-Conjugate Gradient Stabilised and Conjugate Gradient Squared. The performance we obtain is shown in Figure 2.

- Note that all these algorithms have $O(N^2)$ computation complexity on $O(N^2)$ data, which means that there is only limited scope for getting good sequential performance because of memory re-use.
- The measurements we show in Figure 2 are obtained without performing data placement optimisation at runtime [1]. We believe that the performance can be improved by optimising data placement to eliminate unnecessary communication.
- Even without data placement optimisation, a speedup of about 15 on a 25-processor commodity cluster platform is encouraging, given how easy it was to obtain.

4 C++ Interface

In this section we discuss some of the design decisions and C++ programming techniques that were used in implementing the DESO++ interface. The DESO++ interface is built fully on top of the C interface, *i.e.* it calls the functions and uses the datatypes from our C language library API. In the C language interface, the results of delayed operations are represented by handles (which ultimately are integer indices into the data structure storing the DAG for the computation being performed). The application programmer has to force evaluation of such handles explicitly before being able to access the data. In C++, we can do better by using operator overloading: For example, the *force* that happens on the conditional statements in line 40 of Figure 1 is entirely transparent.

Reference-Counting Smart Pointers. The following example illustrates a potential problem that could arise due to our use of delayed evaluation:

```
1  Vector &fun ( const Vector &x, const Scalar &beta ) {
2    Vector a;
3    a = beta * x;
4    return (x + a);
5  }
```

In our system, this function would return a handle for a delayed expression, to be evaluated when the return value of the function is eventually forced. The problem is that on function exit, a would normally be destructed, leaving the return value of the function having an indirect reference to an invalid handle. We resolve this issue by using reference-counting smart pointers, via an extra level of indirection, for accessing delayed handles.

Expression Templates. We use expression templates similar to those in Blitz++ [17] and POOMA II [12] for parsing array expressions such as `r = b - A * x`. Construction of such expressions is fully in-lined. Execution of the assignment operator = triggers the actual construction of the DAG of delayed operations representing the expression.

Careful Separation of Copy Constructors and Assignment Operators. Non-basic types such as our handles for the results of delayed operations trigger copy constructors in C++ even for the purpose of parameter passing. We initially defined copy constructors as making delayed calls to the BLAS copying routine _copy. This resulted in vast numbers of superfluous data copies. We therefore took the design decision to define copy constructors as making aliases, whilst the assignment operator actually copies data.

Traits. The traits technique [18] allows programmers to write functions that operate on and return *types*. This technique is very useful when implementing generic functions, in particular generic operators such as `*`. We could envisage writing a generic interface for `*` as follows:

```
1  template< typename T1, typename T2 >
2  inline Return_Type operator* ( const T1 &m1, const T2 &m2 ) {
3    // ...
4  }
```

What should `Return_Type` be? Traits allow us to define a function that gives the correct type:

```
 1  template< typename T1, typename T2 >
 2  class _promote_product {
 3    // General case: type of product is type of first operand.
 4    typedef T1 Value_Type;
 5  };
 6
 7  template< typename T2 >
 8  class _promote_product< Scalar<double>, T2 > {
 9    // But Scalar * any T2 is always T2
10    typedef T2 Value_Type;
11  };
12
13  template< >
14  class _promote_product< Vector<double>, Vector<double> > {
15    // Special case for dot product: Vector * Vector = Scalar
16    typedef Scalar<double> Value_Type;
17  };
```

The return type for * would then be `_promote_product<T1, T2>::Value_Type`. Note that this example has been very much simplified in order to illustrate the programming technique used.

5   Conclusion

We have described delayed evaluation, self-optimising software components as a possible way of bridging the apparent conflict between the quality of scientific software and its performance. We have presented a library which implements this proposal and have shown that this can be used to write parallel numerical algorithms in a very high-level intuitive manner as well as to transparently parallelise some existing sequential codes.

Skeletons without a Language. It is interesting to consider how our work compares with the Skeletons approach to parallel programming [6, 10]. Typically, skeletons provide a language for expressing the composition of computational components. The benefit of this is that we have very precise high-level structural information about application programs available for the purpose of optimisation. This information can be hard to capture automatically when using compilers for common imperative languages. In our approach, the information which is provided through high-level constructs in skeleton programs is instead captured at runtime by using delayed evaluation.

References

1. O. Beckmann. *Interprocedural Optimisation of Regular Parallel Computations at Runtime.* PhD thesis, Imperial College of Science, Technology and Medicine, University of London, Jan. 2001.

2. O. Beckmann and P. H. J. Kelly. Runtime interprocedural data placement optimisation for lazy parallel libraries (extended abstract). In *Proceedings of Euro-Par '97*, number 1300 in LNCS, pages 306–309. Springer Verlag, Aug. 1997.

3. J. Bilmes, K. Asanovic, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using PhiPAC: A portable, high performance, ANSI C coding methodology. In ICS '97 [11], pages 340–347.

4. BLAST Forum. Basic linear algebra subprograms technical BLAST forum standard, Aug. 2001. Available via `www.netlib.org/blas/blas-forum`.

5. J. Choi, J. J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and R. C. Whaley. LAPACK working note 100: a proposal for a set of parallel basic linear algebra subprograms. Technical Report CS–95–292, Computer Science Department, University of Tennessee, Knoxville, July 1995.

6. J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, D. W. N. Sharp, Q. Wu, and R. L. While. Parallel programming using skeleton functions. In *PARLE '93: Parallel Architectures and Languages Europe*, number 694 in LNCS. Springer-Verlag, 1993.

7. Release of DESO library. `http://www.doc.ic.ac.uk/~ob3/deso`.

8. J. Dongarra, A. Lumsdaine, R. Pozo, and K. A. Remington. LAPACK working note 102: IML++ v. 1.2: Iterative methods library reference guide. Technical Report UT-CS-95-303, Department of Computer Science, University of Tennessee, Aug. 1995.

9. D. R. Engler. Incorporating application semantics and control into compilation. In *DSL '97: Proceedings of the Conference on Domain-Specific Languages*, pages 103–118. USENIX, Oct. 15–17 1997.

10. N. Furmento, A. Mayer, S. McGough, S. Newhouse, T. Field, and J. Darlington. Optimisation of component-based applications within a grid environment. In *Supercomputing 2001*, 2001.

11. *Proceedings of the 11th International Conference on Supercomputing (ICS-97)*, New York, July 7–11 1997. ACM Press.

12. S. Karmesin, J. Crotinger, J. Cummings, S. Haney, W. J. Humphrey, J. Reynders, S. Smith, and T. Williams. Array design and expression evaluation in POOMA II. In *ISCOPE'98: Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments*, number 231–238 in LNCS, page 223 ff. Springer-Verlag, 1998.

13. K. Kennedy. Telescoping languages: A compiler strategy for implementation of high-level domain-specific programming systems. In *IPDPS '00: Proceedings of the 14th International Conference on Parallel and Distributed Processing Symposium*, pages 297–306. IEEE, May 1–5 2000.

14. M. E. Mace. *Memory Storage Patterns in Parallel Processing*. Kluwer Academic Press, 1987.

15. D. J. Quinlan, M. Schordan, B. Philip, and M. Kowarschik. Compile-time support for the optimization of user-defined object-oriented abstractions. In *POOSC '00: Parallel/High-Performance Object-Oriented Scientific Computing*, Oct. 2001.

16. J. G. Siek and A. Lumsdaine. The matrix template library: A generic programming approach to high performance numerical linear algebra. In *ISCOPE '98: International Symposium on Computing in Object-Oriented Parallel Environments*, number 1505 in LNCS, pages 59–71, 1998.

17. T. L. Veldhuizen. Arrays in Blitz++. In *ISCOPE'98: Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments*, number 1505 in LNCS, page 223 ff. Springer-Verlag, 1998.

18. T. L. Veldhuizen. C++ templates as partial evaluation. In *PEPM '99: Partial Evaluation and Semantic-Based Program Manipulation*, pages 13–18, 1999.

19. R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, Jan. 2001.