Numerical Partial Differential Equations in Scheme*

Bradley J. Lucier
Department of Mathematics
Purdue University
West Lafayette, IN 47907-1395

1 Introduction

I worked with the students in my one-semester graduate course CS615 "Numerical methods for partial differential equations" at Purdue University to write a set of routines to use the finite element method to solve elliptic and parabolic partial differential equations (PDEs). We used hacked versions of gcc-2.95.1, the Gambit-C 3.0 Scheme system by Marc Feeley, and the Meroon object system by Christian Queinnec as our software tools. We developed and ran the code on a Compaq DS20 clone with two 500 MHz Alpha 21264 processors and two GB of memory running RedHat 6.0. Our system performance is competitive with similar systems written in C or Fortran. The URL is `http://www.math.purdue.edu/~lucier/615`. We discuss the process of developing the software in this paper.

2 Scope of the system

We defined objects and methods for two-dimensional points in the plane; vertices, edges, and triangles of planar triangulations; refinement of those triangulations; code to triangulate an arbitrary polygon; general vectors and operators (at the level of linear spaces, not at the level of $\mathbb{R}^n$), together with generic functions for the usual vector operations plus Operator-apply and Operator-compose; sparse vectors and matrices in $\mathbb{R}^n$; various orderings of the geometric objects to attempt to be more cache friendly; linear finite elements on triangulations, spaces of such elements, vectors in those spaces, and operators on those vectors; construction of various operators from the coefficients of the problem PDE; and general conjugate-gradient and multigrid iterative methods for solving the resulting linear systems.

3 Good things about the tools

Generic functions plus classes seem to be a good fit for this problem domain. Complex algorithms that have a brief high-level description (e.g., multigrid) can be implemented in a relatively straightforward way by mixing the functional and object-oriented approaches.

Homogeneous vectors of flonums are essential, as are declarations for fixnum and flonum arithmetic. The debugging support in Gambit-C for interpreted code is excellent. Meroon objects are just specially tagged structure in Gambit-C, so object access, etc., is very fast. The beta-reduction pass of the Gambit-C compiler, which is user-controllable via declarations, is very useful for inlining routines and loop unrolling.

The initialize! methods in Meroon are very useful for error checking, complex initialization of some fields based on values in other fields, etc. The with-co-instantiation macro allows one to easily create co-dependent objects (e.g., triangles, edges, and vertices in a refine method).

4 Things that could be improved about the tools

Keeping flonums unboxed as long as possible is important for iterative algorithms. Gambit-C currently boxes flonums across all jumps; this can be a performance issue (e.g., the code for Gauss-Seidel preconditioning is not competitive with C). Gambit-C uses an on-the-fly register allocation algorithm that behaves poorly for heavily nested, lambda-lifted, iterations. While many small routines that return Points (two-vectors) are inlined by Gambit-C, a structure analysis that eliminates the allocation of temporary vectors that are not referenced outside a function would improve performance somewhat.

When Meroon creates an object, and not all the fields of that object have been explicitly initialized, it calls fill-other-fields! to check that each uninitialized field has an initializer thunk or can remain uninitialized. This information is available at compile time, and should be used at that time by the macro expanders. Ensuring that all fields are initialized cut the time to refine a triangulation in half.

Although Meroon objects are given special treatment by the Gambit-C compiler backend, the front end knows nothing about Meroon, so redundant type-checks, say, must be elided by hand with the with-access Meroon macro, or they remain. Also, the Gambit-C printer does not know about Meroon objects–there are hooks in Gambit-C's runtime library that can be used to print Meroon objects, but that is a difficult procedure, since most of the objects we created in this application are recursive. Meroon's unveil function deals properly with this, but it is quite verbose.

5 Changes made to the tools

Queinnec used Meroon's MOP to allow us to use field setters of the form FIELD-NAME-set! rather than set-FIELD-NAME!. This reduced the number of typing and cut-and-paste error with names like Polygon-vertex-forward-edge-set!.

*This work was supported in part by the Office of Naval Research, Contract N00014-91-J-1152.

Meroon does a lot of dynamic error checking to ensure that one does not, say, apply vector-ref to a pair. This allows us to compile Meroon in unsafe mode in Gambit-C. On the other hand, Meroon makes heavy use of macros and code-walkers, and if that code is handed misformed Scheme code, it can crash if compiled in unsafe mode. So we compiled this code in safe mode. Nonetheless, we sometimes needed to run the Meroon system and our own code interpreted to avoid system crashes when debugging.

Queinnec changed the simple accessors and setters to expose more code to the possibility of inlining.

We use IEEE arithmetic, and gcc's register allocator had problems with some of the hardware scheduling constraints of the Alpha 21264. Feeley modified Gambit-C's floating-point code generator to generate Static-Single-Assignment (SSA) type code for the floating-point variables; this led to near-optimal code with twice the performance in some routines.

In gcc, we added a flag -fno-math-errno to not set errno when sqrt is given a negative number, since we rely on IEEE arithmetic error checking, not errno-based error checking. This sped up some routines by 20%. We submitted a patch to remove the trapb instructions for correct IEEE arithmetic implementation on the Alpha 21264, which are needed for correct implementation on earlier Alpha processors. A similar patch is contained in gcc-2.95.1 and the development version of gcc (but not gcc-2.95.2).

6 What we learned

Many small objects (vertices, edges, etc.) are long-lived; many large objects (Finite-element-vectors, etc.) are short-lived.

The Scheme code for about a dozen low-level routines (vector addition, scalar multiplication, dot products, sparse-matrix–vector multiplication, etc.) is hand optimized and is as fast as equivalent code in C. That is not an issue.

The generic function dispatch overhead for points in the plane was too high. Writing special code for points cut the execution time for one routine to fill the nonzero entries of a matrix from 45 seconds to 17 seconds. On the other hand, a better algorithm cut the constant in the complexity bound so much that the new routine for the same purpose ran in 0.5 seconds. Overall, Meroon's performance as an object system, even with its dynamic properties, was good enough.

We are working with a layered system—Meroon on top of Gambit-C on top of gcc. Unfortunately, there is little or no upward knowledge between these systems—gcc does not know about Gambit-C, and Gambit-C does not know about Meroon. And it is difficult to link information about routines in a downward direction—Meroon generic functions and methods are not compiled to named, global, Scheme routines, but to anonymous lambdas, so their names do not appear directly in the C code that Gambit-C generates. And Gambit-C generates a single C routine for all the Scheme routines in a single file, so the names of the Scheme routines do not appear at all in the assembly code generated by gcc.

Last summer, gcc's routines for register allocation, jump analysis, flow analysis, calculation of dominators, and global common subexpression elimination were all quadratic or worse in the number of blocks/jumps/pseudo-registers/etc. This is not very good when trying to compile a single, two-megabyte, routine with 20,000 blocks and similar numbers of jumps and pseudo-registers. Gcc's development team has responded to my notes and come up with new routines that are faster for the first three problems; a fix exists but has not yet been integrated for the dominators calculation, and I believe that the GCSE calculation will be streamlined, too. That doesn't mean challenges to gcc's algorithms will end—the next version of Gambit-C uses the gcc extension of computed goto's and label addresses to increase performance by 50% in many applications. A naive analysis leads to millions of edges in the flow graph, but I believe that this can be avoided.

Performance profiling is essential, and it is difficult in this environment. To find and "fix" the register allocation problem on the 21264 required reading assembly code, trying early (buggy) SSA transformations in gcc and staring at the debugging table of scheduled instructions (of the nonfunctional code) to get enough information to change Gambit-C's floating-point expression generation. Finding the problem with fill-other-fields! required instrumenting the C output of Gambit-C with gcov and finding, not the most-executed statements, but the ones with the largest product of number-of-executions and execution-path length, and then associating that C code with the Scheme code that it came from. Gcc's profiling code operates at the function level, so with all Scheme functions compiled to one large C routine, it was useless.

Finally, in this and similar problem domains, high-level transformations (beta reduction, partial evaluation, temporary structure elimination, etc.) can provide profound performance improvements. This was recognized years ago by Jim Boyle with his TAMPR system, and others. Many current researchers in these areas concentrate on only one aspect of the performance problem—parallelization, fast (e.g., multipole) algorithms, time-stepping methods, etc. Providing an expressive, high-performance programming system like Gambit-C+Meroon (or Common Lisp+CLOS, or ...) offers researchers an environment where the entire problem can be attacked, and the most promising avenues to improve global performance and/or functionality can be revealed.

7 Acknowledgements

Marc Feeley, Christian Queinnec, and (without knowing it) the gcc developers, especially Richard Henderson and Michael Matz, were effectively partners in this effort.

8 Appendix I—Comparison with C

Over 85% of the floating-point operations needed to solve a typical multigrid problem occur in sparse matrix–vector multiply. Today (August 13, 2000) the sparse matrix–vector multiply code runs at a rate of 63 Mflops (8.25 cycles/flop) in our Scheme system and *at exactly the same rate* in C. Our Scheme code for complete solution of a multigrid problem runs at 40 Mflops.

9 Appendix II—Essential language and implementation features

For high performance, I would say that the most important implementation features are: (0) Compilation; (1) Specialized fixnum and (unboxed) flonum arithmetic; (2) Uniform (homogeneous) vectors of floating-point numbers; (3) Compilation of the object system; (4) Efficient β-reduction (inlining) and λ-lifting in the compiler; and (5) Fast method dispatch.

For functionality, The combination of Meroon/CLOS object programming (with classes, objects, generic functions, and methods) plus the functional programming approach can't be topped. As for specific language features, initialization methods are very helpful. In places, it would have been nice to have mixin classes if not full multiple inheritance.

10 Selected Bibliography

10.1 Basic texts

H. Abelson and G. J. Sussman, with J. Sussman, *Structure and Interpretation of Computer Programs* (SICP), 2nd edition, McGraw-Hill, New York, 1996.

V. S. Manis and J. J. Little, *The Schematics of Computation*, Prentice Hall, Englewood Cliffs, N.J., 1995. SICP for mortals.

10.2 Advanced texts

A. W. Appel, *Modern Compiler Implementation in Java*, Cambridge University Press, New York, 1998.

S. C. Brenner and L. R. Scott, *The Mathematical Theory of Finite Element Methods*, Springer-Verlag, New York, 1994.

D. P. Friedman, M. Wand, C. T. Haynes, *Essentials of Programming Languages*, McGraw-Hill, New York, 1992.

C. Queinnec, *Lisp in Small Pieces*, Cambridge University Press, New York, 1996 (translated by K. Callaway).

10.3 Papers on program transformation, programming, and Lisp for Scientific Computing

A. Berlin, "Partial evaluation applied to numerical computation", ACM Conference on Lisp and Functional Programming, June 1990.

A. A. Berlin, R. J. Surati, "Partial evaluation for Scientific Computing: The Supercomputer Toolkit experience", PEPM 1994 - Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Program Transformation Techniques, Orlando, FL, 1994, pp. 133–141.

A. Berlin and D. Weise, "Compiling scientific code using partial evaluation", IEEE Computer, vol. 23, no. 12, Dec. 1990.

J. M. Boyle, T. J. Harmer and V. L. Winter, "The TAMPR program transforming system: Simplifying the development of numerical software" in *Modern Software Tools in Scientific Computing*, E. Arge, A.M. Bruaset and H.P. Langtangen (eds.) pp 353-72 (Birkhuser, 1997).

J. Cuny, R. Dunn, S. T. Hackstadt, C. Harrop, H. H. Hersey, A. D. Malony, and D. Toomey, "Building domain-specific environments for Computational Science: A case study in seismic tomography", International Journal of Supercomputing Applications and High Performance Computing, vol. 11, no. 3.

R. J. Fateman, K. A. Broughan, D. K. Willcock and D. Rettig, "Fast floating-point processing in Common Lisp", ACM Transactions on Mathematical Software, 1995, pp. 26-62.

M. Frigo and S. G. Johnson, "FFTW: An adaptive software architecture for the FFT", Proceedings of ICASSP 1998, vol. 3, p. 1381.

P. Norvig, "Design patterns in dynamic programming," http://www.norvig.com/vita.html