An Efficient Parallel Linear Solver with a Cascadic Conjugate Gradient Method: Experience with Reality

Peter Gottschling[1]* and Wolfgang E. Nagel[2]

[1] GMD FIRST, Berlin, Germany,
pg@first.gmd.de
[2] Center for High Performance Computing (ZHR),
Dresden University of Technology, Germany,
nagel@zhr.tu-dresden.de

Abstract. To solve large systems of linear equations with sparse matrices in parallel, there are three factors that contribute to the computing time: the numerical efficiency, the floating point performance, and the scalability. In this paper, we mainly consider the floating point performance. For large linear systems, multi-level techniques, like the cascadic conjugate gradient method (CCG), require significantly less operations than single-level methods. On the other hand, they are considered less efficient with regard to performance and limited in parallelization. Therefore, to achieve an efficient, massively parallel multi-level solver, we used the fastest available communication and revised the whole computation. The performance improvements led to a parallel solver which is able to solve a linear system with more than 16 million unknowns in 0.77 seconds on 256 PEs of Cray T3E. This corresponds to an overall performance of 10.34 GFLOPS.

Keywords. Floating Point Performance, RISC Processors, Matrix Sparsity Pattern, Cascadic Conjugate Gradient Method

1 Introduction

The solution of large systems of linear equations with sparse matrices plays an important role in many simulation applications and in some of the so-called 'grand challenge' problems. Three components can be identified which determine the necessary time to solve a linear system in parallel: the numerical efficiency, the floating point performance, and the scalability.

First of all, one should examine whether it is possible to apply a multi-level solver – like multigrid or cascadic conjugate gradient method – to the investigated problem. Single-level solvers – like Gauss-Seidel – most often converge poorly for large linear systems. Though the floating point performance and the

* Part of the work was done while the author was a research associate at ZHR in 1999.

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 784–794, 2000.
© Springer-Verlag Berlin Heidelberg 2000

parallel efficiency is sometimes better, this cannot compensate for the numerical deficiencies. The usually better parallelism is explained by the larger ratio between the number of operations and the size of transferred data between the processors because multi-level methods work on several linear systems with different dimensions.

Multi-level techniques use single-level methods on the individual levels. Therefore, the floating point performance on several levels is similar to that of the single-level techniques used. On the smaller linear systems, the performance is sometimes even higher because of better cache reuse. On the other hand, the additional operations on multi-level techniques – the transfer operations between the grids – perform poorly due to indirect and irregular memory accesses. Nevertheless, the computing time of the transfer operations is usually quite short. For that reason, a multi-level technique can sometimes perform similarly as the containing single-level method.

The systems of linear equations, which we considered in our investigations, originate from the discretization of the ground-water flow equation. The strong variation of the parameters of this partial differential equation causes strongly varying coefficients in the matrix of the linear system. Despite the variation of the coefficients and the largeness of the linear system, the cascadic conjugate gradient method with an algebraically generated hierarchy of linear systems enables good convergence [5].

The paper is organized as follows. In section 2, we consider different types of sparse matrices. For these matrix types, the counter-movement of the applicability of discretization schemes and the possibilities of performance tuning is shown. The communication expense is covered in the subsequent section. Section 4 presents the optimization targets for the arithmetic part used in the parallel solver. Different implementations of the matrix vector multiplication are compared in section 5. The last section describes the optimization of the conjugate gradient method.

2 Sparsity Patterns of Matrices

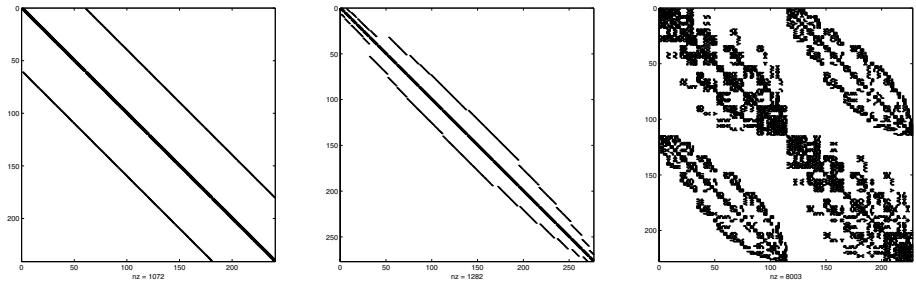
Often, systems of linear equations with sparse matrices originate from discretized partial differential equations. The type of discretization determines the sparsity pattern of the matrix. In this paper, we distinguish three types of matrices.

Structured Matrices: These matrices (fig. 1a) are characterized by a set of constants $C = \{c_1, c_2, \ldots c_m\}$ so that $j - i \notin C \rightarrow a_{ij} = 0$. This means that only matrix elements with a certain distance from the diagonal can be non-zero. Matrices of this form arise in equidistant discretizations of rectangular or cuboid domains. Matrix vector products with this type can be programmed with simple loops using constant offsets. Therefore, different optimizations like loop unrolling and blocking (cf. [4]), are applicable.

Locally Structured Matrices: For the second type of sparsity, the expression 'locally structured' matrices (fig. 1b) is introduced. Here, several sets

of differences $C_1, C_2, \ldots$ can be defined where each set is valid for a certain interval of lines. The sparsity pattern can be expressed in an implementation by structural specifications that correspond with these intervals of matrix lines. This representation at least allows floating point optimizations within the intervals. Locally structured matrices originate, for instance, from the equidistant discretization of domains with irregular borders.

Unstructured Matrices: The most general kind of sparse matrices are unstructured matrices (fig. 1c, from [6]). No assumptions about the sparsity pattern are made here. Thus, arbitrary discretizations are permitted. On the other hand, each matrix element must be treated separately in a matrix vector multiplication and the performance is lower for that reason. Nevertheless, discretizations that are adapted to the problem are often necessary and the lower speed is justified by a significant reduction of the equation size.



(a) structured (b) locally struct. (c) unstructured

Fig. 1. Types of sparsity patterns

In our work, we consider locally structured matrices because of their importance in ground-water flow simulations. The two other matrix types are used in numerous other projects (structured matrices e.g. in [1] and unstructured ones e.g. in [10]).

3 Communication Expense

The computation of one step of the conjugate gradient method involves one exchange of the inner borders in the matrix vector multiplication and two reductions in dot products. Further exchanges of the inner borders are necessary in some preconditionings (e.g. incomplete CHOLESKY factorization). Moreover, the implementation of the termination criterion requires an additional reduction unless the values of the conjugate gradient method (CG) are used (cf. [2]).

The data dependencies in the conjugate gradient method allow the simultaneous reduction of the termination criterion and of the first dot product in the next iteration step of the CG method. Since the communication latency is

rather large compared to the bandwidth on every parallel computer and computer network the global reduction of two values takes roughly the same time as the reduction of one value.

To minimize the expense of the partitioning, the domain (on the fine grid) was decomposed by coordinate section in $P_x \times P_y$ subdomains on $P = P_x \cdot P_y$ processors. The decomposition was passed to the coarser grid where the boundaries were slightly adapted to conserve the load balance (cf. [5]).

A significant decrease of the communication time can sometimes be established by replacing portable communication procedures with proprietary ones. Figure 2 shows the time line, visualized by VAMPIR [7], of two iterations of the CG-method on a linear system with about 16,000 unknowns on Cray T3E. In this implementation based on the MPI library, the interprocessor communication, represented by lines, is the dominant part. Implementing the communication with equivalent shmem functions (shmem_double_get and shmem_double_sum_to_all) clearly shortens the communication time so that the execution time of one iteration is reduced from 1.437 ms to 0.632 ms (figure 3).

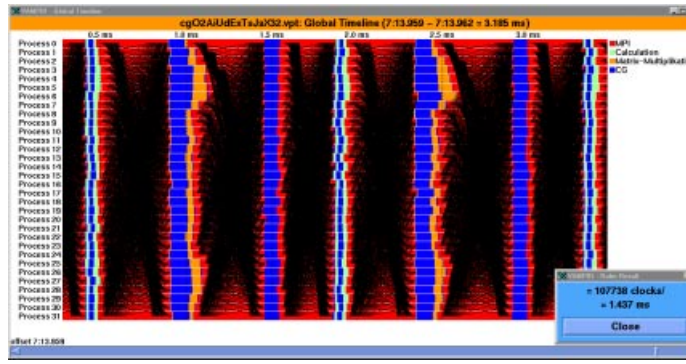


Fig. 2. Two iteration steps with MPI communication

Although solving a linear system with 16,000 unknowns is not very interesting to parallel computing, there are two reasons for accelerating the communication. Firstly, the communication becomes more important for increasing numbers of processors, even for large problems, and secondly, multilevel solvers spend most of their time on communication (delay mainly due to latency issues) while solving the coarse grid equations. Since we are interested in multilevel solvers on many processors, an optimized communication is very important.

4 Optimization Targets to Improve the Floating Point Performance on RISC Processors

To increase the floating point performance of a RISC processor for our application, we followed four goals. We focused on the DEC Alpha 21164 of Cray T3E,

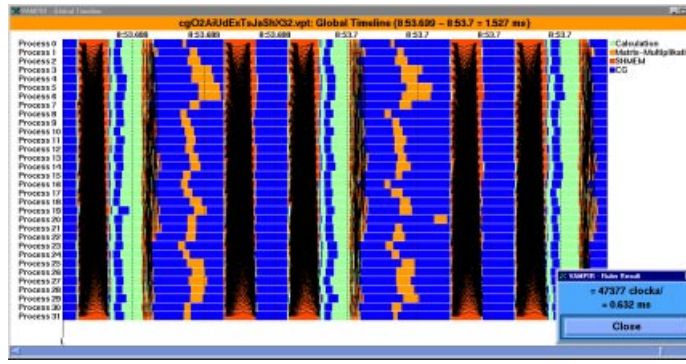


Fig. 3. Two corresponding iterations with shmem communication

nevertheless, the optimization steps should be helpful on other superscalar RISC processors, too.

Decrease of the Memory Accesses by Cache Reuse: The main bottleneck of fast RISC processors is the rather slow main memory access. On Cray T3E, for instance, processors with 300-600 MHz and 600-1200 MFLOPS face main memories with 75 MHz. To yield floating point performance near peak performance, it is necessary (but not sufficient) to reuse data in registers or in the primary cache as often as possible so that memory accesses are not relevant to the execution time. Seidl [9] has shown by examples that the acceleration of algorithms can be predetermined from the reduced probability of memory accesses.

Consecutive Memory Accesses in Increasing Order: Loading a data item into the cache is realized by loading a certain amount of memory, called cache line, which is typically larger than the data item itself. If referred subsequently, the next data items are already in the cache with some probability. On Cray T3E, additional benefit can result from the stream buffers. This is a mechanism that looks for increasing memory accesses and, if recognized, loads successive cache lines into a special register where they can be rapidly loaded into the second level cache (cf. [8]). Operations on large vectors are, from the authors' experience, typically computed twice as fast with stream buffers.

Independent Operations: Implementations with many independent operations allow the efficient use of multiple pipelines. The number of independent operations is very often increased by loop unrolling.

Reduction of Divide Operations: The divisions of floating point numbers are not as fast as additions and multiplications on most processors. In addition, they cause pipeline blocking on the DEC Alpha 21164.

5 Matrix Vector Multiplication

The importance of the matrix vector multiplication is based on the fact that in our solver half of the floating point operations are performed in this section.

In the original implementation, the multiplication was calculated in two steps. At first, the result vector was initialized with the product of the input vector and the diagonal of the matrix. Then, the components of the result vector were incremented by non-zero matrix entries outside the diagonal multiplied with the elements of the input vector. In this way, sub-vectors of maximal length were used and the loop overhead was minimized.

As an example, we considered a multiplication with a vector containing about 126,000 elements and an appropriate matrix, so that about 1,134,000 floating point operations had to be executed. The original implementation required 34.9 ms to compute the multiplication on the 600 MHz processor applying stream buffers. This corresponds to 32.5 MFLOPS.

While in the second section approximately 8 floating operations per unknown were computed in relation to 4 to 6 memory accesses (depending on the extension and form of the domain and the size of the second level cache), there are three memory accesses per unknown with only one operation performed in the first section. This unfavorable ratio between operations and memory accesses resulted in a very slow computation. Once more looking at Cray T3E, on the 600 MHz processor the initialization step performed with 8.4 MFLOPS without stream buffers where the performance was only augmented to 18.3 MFLOPS by using the stream buffers. To emphasize the importance of the memory bandwidth, the same operation was performed with 7.5 MFLOPS and 15 MFLOPS respectively on a 300 MHz processor.

To avoid this slow computation, the initialization of the output vector was included in the incrementing step. The product of the diagonal and the input vector was then only computed for those elements where the output vector was incremented in the next moment. This modified implementation saved 2 memory accesses per unknown for loading the vectors (unless the sub-vectors involved in the incrementing section were too large for the cache) but additional overhead was produced to control which components of the result vector must be initialized and due to dividing the initialization into several sections. On the example problem, this implementation took 26.9 ms for the multiplication (42.2 MFLOPS).

Experiments with structured matrices have shown that matrix vector multiplications were significantly faster if the elements of the result vector were computed explicitly (`v[i]= a[i][i]*q[i] + a[i][j]*q[j] + ···`) instead of by incrementing as described above. For locally structured matrices, it is more complicated to apply the explicit calculation. In comparison with the former implementation, the loops are shorter and they need more preparations. Altogether, the loop overhead is more than doubled, although the computing time is decreased by 59.5 per cent to 14.1 ms on the example problem (80.2 MFLOPS).

Another performance optimization is applicable if the multiplication is part of the conjugate gradient method. There, the dot product of the input and the output vector is used. Since two memory accesses are necessary per floating point

operation this calculation is slow (there is a bottleneck on the scalar value, too). For the considered vector size, the dot product requires 5.1 ms.

Including this dot product in the matrix vector multiplication, as proposed in [3], saves the memory accesses. Consequently, most of the computing time for the dot product is saved (the scalar value is less critical here because several floating point operations are performed between two increments).

In a parallel implementation, attention has to be paid to the inner boundaries. To enable a multiplication with a symmetric matrix, vector components `q[j]` that are assigned to other processors are considered in an extra computation (`v[i]+= a[i][j]*q[j]`). In this section, the dot product is calculated as `dot+= a[i][j]*q[j]*q[i]` while the computation with the symmetric part of the matrix is `dot+= v[i]*q[i]`. Altogether, the execution time of the combined computation was 14.7 ms which corresponds to 93.9 MFLOPS.

6 Iteration Steps of the Conjugate Gradient Method

The conjugate gradient method itself only consists of vector operations. Since vector operations are characterized by a poor ratio between the number of floating point operations and the number of memory accesses, the performance is rather low. To improve this ratio, the vector operations have to be combined to compute as many floating point operations as possible on a vector component while it resides in the cache. Whereas calculations depending on a vector can start as soon as a part of the vector is available, calculations depending on a scalar value resulting from a vector reduction must wait until the reduction is finished.

Although the conjugate gradient method is well known, we present the iteration step as a C++ program in table 1 for a better illustration. The startup phase is omitted because it does not provide further optimization opportunities. The extension of the CG method to the cascadic conjugate gradient method is quite simple. Starting on the coarsest grid, the CG method is computed on every grid. When the termination criterion is fulfilled on a certain grid the vector x is interpolated to the next finest grid and is used as an initial guess for the CG method on that grid.

The program can be implemented in this form by using templates in C++. The use of templates is critical because each operator is computed separately. For this reason, temporary vectors are required, which produce overhead for additional memory allocations and memory accesses, unless advanced numerical template libraries like Blitz++ [11] are used.

In the calculation of dot products, the scalar value represents a bottleneck for superscalar processors. Since the addition is associative (in exact arithmetic), the products of the vector components can be summed into several values within the loop and added at the end. For the vector size considered in the previous section, the computing time was reduced from 5.1 ms to 3.1 ms, which corresponds to an increase of the performance from 49.2 MFLOPS to 77.8 MFLOPS. In our

```
vector<double>      x, v, q, r, w;
double              alpha, delta, gamma, gamma_old;

int cg_iteration ()
{ double     delta_local, gamma_local;

  q= (gamma/gamma_old) * q + w;
  exchange_inner_borders (q);          // requires communication
  v= a_mult (q);
  delta_local= dot (v, q);
  delta= reduce (delta_local);         // requires communication
  alpha= gamma / delta;
  x+= alpha * q;
  r-= alpha * v;
  w= thepc→f (r);                      // chosen preconditioning (possibly requires comm.)
  gamma_local= dot (w, r);
  gamma= reduce (gamma_local);         // requires communication
  return thepc→f ();                   // chosen termination criterion (usually requires comm.)
}
```

Table 1. Iteration of the conjugate gradient method

investigations, changing the order of the summation did not noticeably influence the exactness of the floating point arithmetic.

Inlining a particular preconditioning and a particular termination criterion saves the function calls and allows further optimization. For the investigated systems of linear equations, the diagonal preconditioning $w = D^{-1}r$ represented the best compromise between the numerical properties and the computational and communicational expense. Among different termination criteria, it has been shown that for equations with strongly varying coefficients, the diagonally preconditioned residual $D^{-1}r$ enables the best error estimation. So, the commitment to this combination permits the elimination of redundant calculations.

Since the element-wise division is rather slow and the vector components are divided by constant values, it is worthwhile to store the inverse of the diagonal matrix. An element-wise multiplication then replaces the element-wise division at the price of an extra vector and some additional computation before starting the linear solver.

To reduce the number of memory accesses, the calculations of `r-= alpha * v`, `w= adinv * r`, `dot (w, r)` and `dot (w, w)` can be combined in one loop, where `adinv` is a vector with $\texttt{adinv}[i] = 1/a_{ii}$. Of course, this loop can be unrolled, too. With the simultaneous reduction of the local computations of `dot (w, r)` and `dot (w, w)` and the modifications described above, the iteration of the conjugate gradient method looks as shown in table 2.

As an example, a linear system with more than 16 million unknowns was solved on 32 processors. Here, the execution time of a single iteration step on the finest level was decreased from 432 ms in the original version to 206 ms in the accelerated one. Furthermore, it has been shown that the variations of the computing time between the different processors gain in significance due to the performance tuning. Although the number of operations are equally distributed among the processors, the computing time varies noticeably. As a consequence,

```
int     nupo;     // number of points corresponds to vector size

int cg_iteration_jacobi ()
{ double     delta_s, delta_a, delta_local, stop_gamma_local [2], stop_gamma [2],
             s0, s1, s2, s3, g0, g1, g2, g3, tmp0, tmp1, tmp2, tmp3;
  int        i, nupo4= nupo ≫ 2 ≪ 2;

  scadd (gamma/gamma_old, q, w);
  exchange_inner_borders (q);                                  // requires communication
  delta_local= a_mult (v, q);
  delta= reduce (delta_local);                                 // requires communication
  alpha= gamma / delta; scadd (x, alpha, q);
  s0= s1= s2= s3= g0= g1= g2= g3= 0.0;
  for (i= 0; i < nupo4; i+= 4)
    {tmp0= (r [i]-= alpha * v [i]) * adinv [i]; s0+= tmp0 * tmp0; g0+= r [i] * tmp0;
    tmp1= (r [i+1]-= alpha * v [i+1]) * adinv [i+1]; s1+= tmp1 * tmp1; g1+= r [i+1] * tmp1;
    tmp2= (r [i+2]-= alpha * v [i+2]) * adinv [i+2]; s2+= tmp2 * tmp2; g2+= r [i+2] * tmp2;
    tmp3= (r [i+3]-= alpha * v [i+3]) * adinv [i+3]; s3+= tmp3 * tmp3; g3+= r [i+3] * tmp3;
    w [i]= tmp0; w [i+1]= tmp1; w [i+2]= tmp2; w [i+3]= tmp3;}
  for (i= nupo4; i < nupo; i++)
    {w [i]= tmp0= (r [i]-= alpha * v [i]) * adinv [i]; s0+= tmp0 * tmp0; g0+= r [i] * tmp0;}
  stop_gamma_local [0]= s0 + s1 + s2 + s3; stop_gamma_local [1]= g0 + g1 + g2 + g3;
  reduce2 (stop_gamma_local, stop_gamma);                      // requires communication
  gamma_old= gamma; gamma= stop_gamma [1];
  return sqrt (stop_gamma [0]) < thetc→epsilon;
}
```

Table 2. Iteration of the specialized and accelerated version

the parallel execution time was affected more by the loss of synchronism than by the communication expense on Cray T3E.

On two processors, where the balance of the computing time and the communication are less significant, the accelerated implementation achieved a performance of 66.2 MFLOPS per processor. On 256 processors,[1] the performance per processor was more than 40 MFLOPS, leading to an overall performance of 10.34 GFLOPS. To solve the linear system with 16 million unknowns, the solver based on the cascadic conjugate gradient method and on algebraically generated coarse grid equations required 9 iterations of the CG method on the finest grid and 15 iterations on the other grids [5]. So, the linear system was solved in 0.77 seconds. Since many simulations of physical processes – like ground-water flow – are based on the solution of many large linear systems, a fast parallel multi-level solver can save a lot of computing time. On the other hand, the acceleration of the solver may also be used to increase the simulation complexity in order to improve the simulation results.

7 Conclusion

Although on modern computer architectures the memory bandwidth is still by far too low with regard to the processor speed, this relation is expected to get worse in the near future (cf. [4, p. 34]). For this reason, the primary performance optimization target is reducing the number of main memory accesses. Therefore,

[1] The authors would like to thank Prof. Hoßfeld from John von Neumann-Institute for Computing (NIC-ZAM) for providing capacity on Cray T3E.

the computations have to be reordered so that as many operations as possible are performed on a data item while it resides in cache.

Fortunately, there is a relatively small kernel in many numerical applications where most of the computing time is spent. In this case, the performance tuning efforts can be restricted to this kernel so that the expense to improve the performance is usually low compared with the expense for the program development.

In scientific applications described by partial differential equations most of the execution time is usually spent on the solution of linear systems. Therefore, the acceleration of the linear solver can yield a great profit. First of all, it should be examined whether a multi-level method can be applied to the respective type of the linear system. In this case, the number of operations can often be decreased by several orders of magnitude by using a different algorithm. For the considered linear systems, which originate from the ground-water flow equation, the difficulty lies in the strong variation of the coefficients (up to 10^8). Nevertheless, the cascadic conjugate gradient method with algebraically generated coarse grid equations converged well for the examined linear systems.

To implement the parallel CCG solver efficiently, the whole iteration step of the CG method was optimized with regard to the performance, including the preconditioning and the termination criterion. A special matrix type is introduced which allows the demanded applicability of discretization schemes and provides more performance optimization possibilities than unstructured matrices. In addition, the communication time was significantly shortened by changing from the portable MPI library to the proprietary shmem library. So, even multilevel solvers can work efficiently on up to several hundred PEs on Cray T3E. Altogether, the fast communication, the high floating point performance and the good convergence enabled the solution of a linear system with over 16 million unknowns in less than a second.

References

[1] Manfred Alef. Implementation of a multigrid algorithm on suprenum and other systems. *Parallel Computing*, 20:1547–1557, 1994.

[2] Peter Deuflhard. Cascadic conjugate gradient methods for elliptic partial differential equations. algorithm and numerical results. In D. Keyes and J. Xu, editors, *Proc. of the 7th Int. Conf. on Domain Decomp. Methods 1993*, pages 29–42, 1994.

[3] Jack J. Dongarra, Iain S. Duff, Danny C. Sorensen, and Henk A. van der Vorst. *Numerical Linear Algebra for High-Performance Computers*. SIAM, Philadelphia, 1998.

[4] Kevin Dowd and Charles R. Severance. *High Performance Computing*. O'Reilly, Sebastopol, second edition, 1998.

[5] Peter Gottschling. Efficient parallel computation of the discretized ground-water flow equation with algebraic multigrid methods (in german). Phd thesis, TU Dresden, in preparation.

[6] Uwe Lehmann. Festigkeitsuntersuchung von Eisenbahnfahrwegen auf Parallelrechnern. Diplom, TU Dresden, 1999.

[7] W.E. Nagel and A. Arnold. Performance visualization of environment. Technical report, Forschungszentrum Jülich, 1995.
http://www.fz-juelich.de/zam/PT/ReDec/SoftTools/PARtools/PARvis.html.

[8] Wolfgang E. Nagel. The new massively parallel computer cray t3e in spring 1996: Experience with virginity (in german). In Hans-Werner Meuer, editor, *Supercomputer 1996*, pages 92–107. K. G. Saur, 1996.
[9] Stephan Seidl. Code crumpling: A straight technique to improve loop performance on cache based systems. In *Proc. 5th Eur. SGI/Cray MPP Workshop*, 1999.
[10] J. Stiller, K. Boryczko, and W.E. Nagel. A new approach for parallel multigrid adaption. In *Proc. 9th SIAM Conf. on Par. Proc. for Sci. Comp.*, 1999.
[11] Todd Veldhuizen et al. *Blitz++*. www://oonumerics.org/blitz/.