Improving the Performance of Tensor Matrix Vector Multiplication in Quantum Chemistry Codes

W. D. Gropp, D. K. Kaushik, M. Minkoff, and B. F. Smith

Argonne National Laboratory {gropp,kaushik,minkoff,bsmith}@mcs.anl.gov

Abstract

Cumulative reaction probability (CRP) calculations provide a viable computational approach to estimate reaction rate coefficients. However, in order to give meaningful results these calculations should be done in many dimensions (ten to fifteen). This makes CRP codes memory intensive. For this reason, these codes use iterative methods to solve the linear systems, where a good fraction of the execution time is spent on matrix-vector multiplication. In this paper, we discuss the tensor product form of applying the system operator on a vector. This approach shows much better performance and provides huge savings in memory as compared to the explicit sparse representation of the system matrix.

1 Introduction and Motivation

The prevalence of parallel processors makes many areas of simulation accessible that was only possible in the recent past on specialized facilities. One area of application is the use of computational methods to calculate reaction rate coefficients. These coefficients are often estimated experimentally. However, the simulations approaches [4] provide a reasonable alternative. Typically the ab initio approach is only applicable to small atomic systems. In these models the dimensionality of the problem is the number of degrees of freedom in the molecular system. If we consider torsion, stretching, etc., the maximum number of DOF for a molecule is proportional to N, the number of atoms. Thus dealing with problems of only three to five degrees of freedom is quite restrictive. The alternative to ab initio methods is the use of statistical studies of reaction paths and thus obtain the reaction rate coefficients statistically. This approach is founded however on a less solid theoretical basis.

Reaction rate calculation involves a dimensional effect based upon degrees of freedom (DOF). That is we consider the reactions that involve molecules

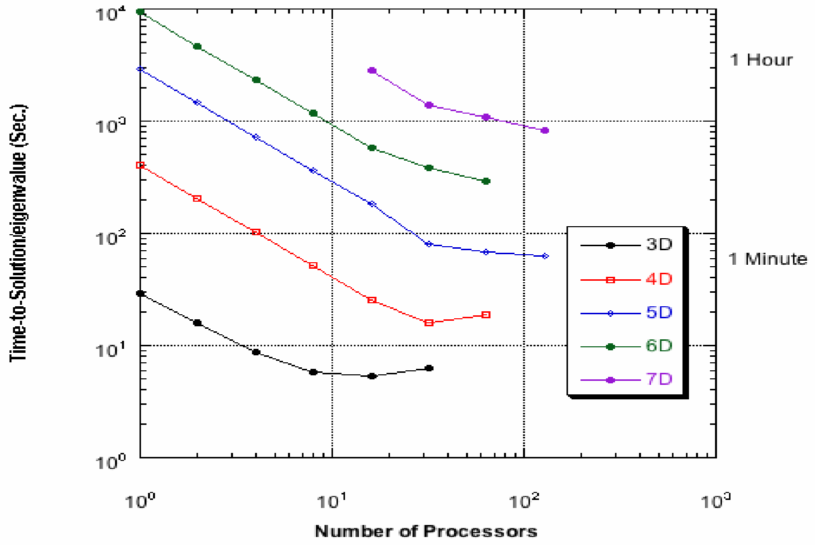

Fig. 1. Sample parallel performance of the CRP code on up to 128 processor of IBM SP3 at NERSC.

having various independent coordinates. For a simple two atom molecule in which we only consider one dimension and a variable representing the distance between the two atoms, we would have one DOF. However, if we add the angle between the atoms in two dimensional space and also add the torsion effect we would have three DOF. We are interested in problems of up to ten or more DOF. This leads to large-scale problems in which parallel computation is a central aspect of the algorithmic approach. For such problems the Greens function solutions (see Section 2) cannot be done by direct linear solvers. A standard approach even applied to lower DOF is to use iterative methods such as GMRES for solving the linear systems. The solution of these linear systems is the fundamental computational cost in the method as we and others have observed. In some of our computational experiments (see Figure 1) we have obtained an accurate eigenvalue in only two to three iterations, however we require from five hundred to a thousand GMRES iterations for each of the Greens function solves. Thus the principal focus of this paper is the effective solution of the matrix-vector multiply.

Normally the matrix vector multiplication is done by first building up the large sparse matrix from the tensor products of one dimensional operators with the identity matrix. The sparse matrix vector product is well known to give poor performance since it is memory bandwidth limited computation with poor data reuse [3, 7]. Since this kernel is responsible for a large fraction (over 80 %) of overall execution time, addressing its performance issues is crucial to obtain a reasonable percentage of machine peak. In this paper, we suggest an alternative approach (in Section 4) that transforms the memory bandwidth limited sparse matrix vector products to matrix-matrix multiplications with high level of data locality. This approach holds the potential to improve the performance of the overall code by a large factor.

The rest of the paper is organized as follows. We discuss the background of the CRP approach in Section 2.1. Next we analyze the performance characteristics of the sparse matrix vector multiplication approach in Section 3. We present the details of the tensor matrix vector multiplication approach in Section 4. Then we compare the performance of these two choices for matrix vector multiplication on Intel Madison processor in Section 5.

2 Background of the CRP Approach

The Cumulative Reaction Probability function is:

$$k(T) = [2\pi\hbar Q_r(T)]^{-1} \int_{-\infty}^{\infty} dE e^{-E/kT} N(E) \tag{1}$$

where Q_r is the reactant partition function. The rate constant is given as

$$k(E) = [2\pi\hbar\rho_r(E)]^{-1} N(E) \tag{2}$$

Therefore the CRP is key in calculating the rate constant. In fact, $N(E)$ can be expressed in terms of the trace of the reaction probability operator, $\hat{P}$

$$N(E) = tr[\hat{P}(E)] \equiv \sum kp_k(E) \tag{3}$$

and

$$\hat{P}(E) = 4\hat{\epsilon}_r^{1/2}\hat{G}(E)\hat{\epsilon}_p\hat{G}(E)\hat{\epsilon}_r^{1/2} \tag{4}$$

The Green's function is

$$\hat{G}(E) = (E + i\hat{\epsilon} - \hat{H})^{-1} \tag{5}$$

$\hat{H}$ is the Hamiltonian and $\hat{\epsilon} = \hat{\epsilon}_r + \hat{\epsilon}_p$ where $\hat{\epsilon}$ is a given absorbing potential, and $\hat{\epsilon}_r$ and $\hat{\epsilon}_p$ are, respectively are the part of $\hat{\epsilon}$ in the reactant and product regions.

In summary, we seek to obtain the major components of the trace of $\hat{P}(E)$. Thus we seek the largest few eigenvalues of this operator. This can be accomplished by means of a Lanczos iteration of (4). For each Lanczos iteration we require the solution of two linear systems (5):

$$(E + i\hat{\epsilon} - \hat{H})\mathbf{y} = \mathbf{x} \tag{6}$$

and its adjoint when $\mathbf{x}$ is known. The matrix on the left hand side of Equation 6 is obtained from one dimensional operators as described next.

2.1 Matrix Vector Multiplication in CRP

For simplicity, let us consider a three dimensional system with n mesh points in each dimension. Then, we need to multiply matrix A ($n^3 \times n^3$) with a vector $\mathbf{v}$ of size n^3.

$$\mathbf{w} = A\mathbf{v} \tag{7}$$

with $\mathbf{w}$ being the output vectors of size n^3. The system matrix A is sparse with the following components:

$$A = B_z \otimes I \otimes I + I \otimes B_y \otimes I + I \otimes I \otimes B_x \tag{8}$$

Where, $\otimes$ denotes the tensor (Kronecker) product of one dimensional operators (B_x, B_y, B_z) with the identity matrix(I). The operators B_x, B_y, and B_z are dense matrices of size $n \times n$.

For d dimensions, we will have d terms in Equation 8 involving d tensor products of dense matrices of size $n \times n$ with the identity matrices of order n. As stated earlier, doing the matrix vector multiplication (Equation 7) is a key operation in the CRP algorithm. Next we discuss the sparse representation of matrix A.

3 Sparse Matrix Vector Product

The sparse matrix-vector product is an important part of many iterative solvers used in scientific computing. While a detailed performance modeling of this operation can be complex, particularly when data reference patterns are included [6, 7, 8], a simplified analysis can still yield upper bounds on the achievable performance of this operation. To illustrate the effect of memory system performance, we consider a generalized sparse matrix-vector multiply that multiplies a matrix by N vectors. This code, along with operation counts, is shown in Figure 2.

3.1 Estimating the Memory Bandwidth Bound

To estimate the memory bandwidth required by this code, we make some simplifying assumptions. We assume that there are no conflict misses, meaning that each matrix and vector element is loaded into cache only once. We also assume that the processor never waits on a memory reference, that is, any number of loads and stores can be issued in a single cycle.

For the algorithm presented in Figure 2, the matrix is stored in compressed row storage format (similar to PETSc's AIJ format [1]). For each iteration of the inner loop in Figure 2, we transfer one integer (`ja` array) and $N+1$ doubles (one matrix element and N vector elements), and we do N floating-point multiply-add (`fmadd`) operations or $2N$ flops. Finally, we store the N output vector elements. If we just consider the inner loop and further assume that

```
for (i = 0, i < m; i++) {
  jrow = ia(i+1)                           // 1 Of, AT, Ld
  ncol = ia(i+1) - ia(i)                   // 1 Iop
  Initialize, sum1, ..., sumN              // N Ld
  for (j = 0; j < ncol; j++) {             // 1 Ld
    fetch ja(jrow), a(jrow),
          x1(ja(jrow)), ..., xN(ja(jrow))  // 1 Of, N+2 AT, N+2 Ld
    do N fmadd (floating multiply add)     // 2N Fop
    jrow++
  }                                        // 1 Iop, 1 Br
  Store sum1, ..., sumN in
        y1(i), ..., yN(i)                  // 1 Of, N AT, N St
}                                          // 1 Iop, 1 Br
```

Fig. 2. General form of sparse matrix-vector product algorithm: storage format is AIJ or compressed row storage; the matrix has `m` rows and `nz` non-zero elements and gets multiplied with `N` vectors; the comments at the end of each line show the assembly level instructions the current statement generates, where `AT` is address translation, `Br` is branch, `Iop` is integer operation, `Fop` is floating-point operation, `Of` is offset calculation, `LD` is load, and `St` is store.

vectors are in cache (and not loaded from memory), we load one double and one integer for $2N$ flops or 6 bytes/flop for one vector and 1.5 bytes/flop for four vectors (see [3] for more detailed treatment). The STREAM [5] benchmark bandwidth on Intel Madison processor is about 4,125 MB/s. This gives us the maximum achievable performance of 687 Mflops/s for one vector and 2,750 Mflops/s for four vectors while the corresponding observed numbers are 627 Mflops/s and 1,315 Mflops/s (out of the machine peak of 6 Gflops/s).

Following a similar procedure, we show the memory bandwidth bound, the actual performance and the peak performance for IBM Power 4, Intel Xeon, IBM BlueGene, and Intel Madison processors (assuming only one vector) in Figure 3. It is clear that the performance of sparse matrix vector multiplication is memory bandwidth limited and the peak processor performance is pretty much irrelevant for this computation. We next discuss the tensor product form of the system operator that does not suffer from this limitation.

4 Tensor Matrix Vector Product

The system matrix in the CRP code comes from the tensor products of one directional dense operators with the identity matrix. This allows us to do the matrix vector multiplication without ever forming the large sparse matrix. Though a cheap approximation to the system matrix is usually needed for preconditioning purpose, we assume that it can be obtained in some other suitable way.

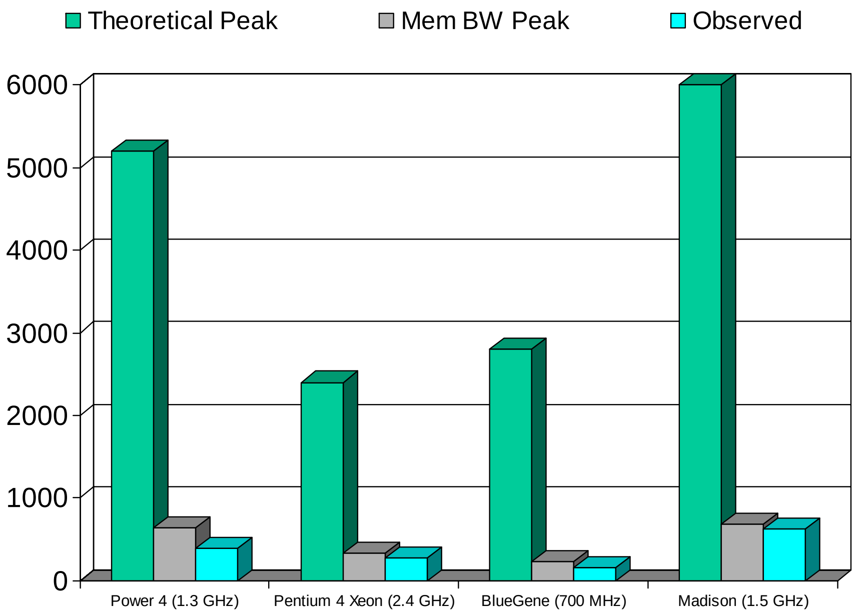

Fig. 3. Memory bandwidth bound for sparse matrix-vector product. Only one vector ($N = 1$) is considered here. The matrix size has `m` = 90,708 rows and `nz` = 5,047,120 nonzero entries. The processors are 1.3 GHz IBM Power 4, 2.4 GHz Intel Xeon, 700 MHz IBM BlueGene, and 1.5 GHz Intel Madison. The memory bandwidth values are measured using the STREAM benchmark.

We can combine the identity matrix tensor products in Equation 8 (and its higher dimensional counterparts). In general, the matrix vector product of Equation 7 will be the sum of the terms made from the three types of operations: $(I \otimes B)\mathbf{v}$, $(B \otimes I)\mathbf{v}$, and $(I \otimes B \otimes I)\mathbf{v}$. We describe how to carry out each of these operations efficiently next. The three dimensional case is described in detail in [2]

Type A: $(\boldsymbol{I} \otimes \boldsymbol{B})\mathbf{v}$

We need to evaluate

$$(I_{p\times p} \otimes B_{m\times m})\mathbf{v}$$

with $\mathbf{v} = (v_1, v_2, ..., v_{pm})^T$. We can view the vector $\mathbf{v}$ as a matrix (V) of size $m \times p$ and then

$$(I_{p\times p} \otimes B_{m\times m})\mathbf{v} = B_{m\times m} \times V_{m\times p}$$

.

It should be noted that the memory layout of the vectors **v** and **w** does not change in this operation. Since the matrix V is stored columnwise, its data access pattern in the above matrix-matrix multiplication is ideal (unit stride). As the number of dimensions increases, the order (p) of the identity matrix gets larger and larger. Therefore, the above algorithm multiplies a small square matrix (B) with a highly rectangular matrix (V) for large dimensions. We will see in Section 5 that many matrix-matrix multiplication implementations do not perform well under this situation.

Type B: $(\boldsymbol{B} \otimes \boldsymbol{I})\mathbf{v}$

Here we need to evaluate

$$(B_{m\times m} \otimes I_{p\times p})\mathbf{v}$$

We can view the vector **v** as a matrix (V) of size $p \times m$ and then

$$(B_{m\times m} \otimes I_{p\times p})\mathbf{v} = V_{p\times m} \times {B^T}_{m\times m}$$

where ${B^T}_{m\times m}$ is the transpose of $B_{m\times m}$ ([2]). Again, the memory layout of the vector **v** and **w** does not change with this operation and this is also a matrix-matrix multiplication. The data access pattern for the matrix V is not unit stride here (with the normal triply nested loop implementation) and transposing this matrix may bring significant performance gains.

Type C: $(\boldsymbol{I} \otimes \boldsymbol{B} \otimes \boldsymbol{I})\mathbf{v}$

Here we need to evaluate

$$(I_{p\times p} \otimes B_{m\times m} \otimes I_{r\times r})\mathbf{v}$$

with $\mathbf{v} = (v_1, v_2, ..., v_{pmr})^T$.

This can be evaluated by looping over Type B term algorithm p times [2]. Each iteration of this loop will evaluate the Type B term $V_{r\times m} \times {B^T}_{m\times m}$. Again this can be done without changing the memory layout of the vectors **v** and **w**.

5 Results and Discussion

In the previous section, we saw that all terms of the generalized form (for d dimensions) of Equation 8 can be evaluated as dense matrix-matrix multiplication, which inherently has very high data reuse and usually performs at a large fraction of machine peak (if implemented properly). We present here some sample performance results on Intel Madison processor (1.5 GHz, 4 MB L2 cache, and 4GB memory). We discuss three implementations:

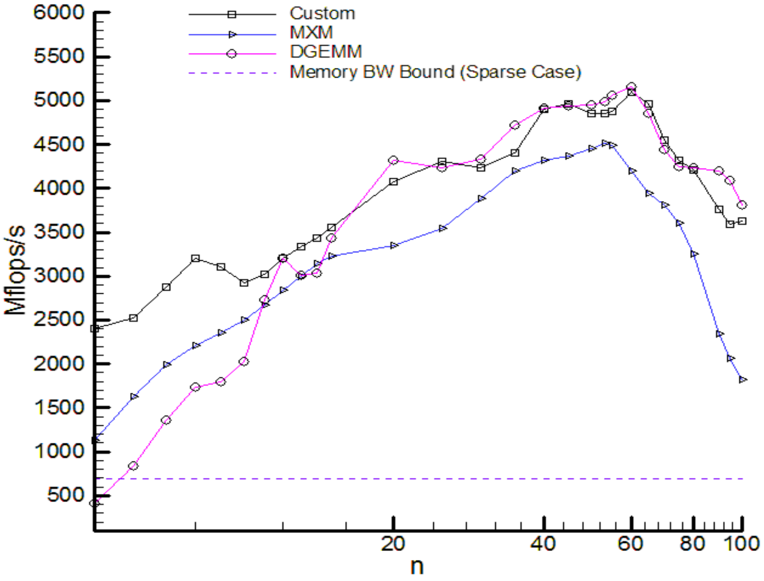

Fig. 4. Performance of the tensor matrix vector multiplication for three dimensions on Intel Madison (1.5 GHz) processor. The custom code is manually optimized code, MXM code is from [2] and DGEMM() routine is from Intel's MKL library. Note that the sparse matrix vector multiplication will only do at most about 687 Mflops/s based on the memory bandwidth bound on this processor.

- **Custom code**: this is the hand optimized code specifically written for evaluating the Type A, B, and C terms.
- **MXM code**: this is taken from Deville, et al. [2].
- **DGEMM**: this is from a vendor library (Intel MKL).

We show the performance advantage of the tensor matrix vector multiplication in three dimensions for n = 5 to 100 in Figure 4. If we had done the matrix vector multiplication by explicitly building the sparse matrix, the performance would have been limited to about 687 Mflops/s (see the dotted line in Figure 4, which is based on the memory bandwidth bound) on this processor. All the three variants give good performance for reasonably large n (≥ 15).

While vendors have invested considerable effort in optimizing the matrix-matrix multiplication, it is usually done for large and balanced matrix sizes. The CRP code involves matrix-matrix multiplications between small square matrices (typically 7×7 to 10×10) and highly rectangular matrices (arising from the matrix view of the input vector $\mathbf{v}$). We show this situation in Figure 5

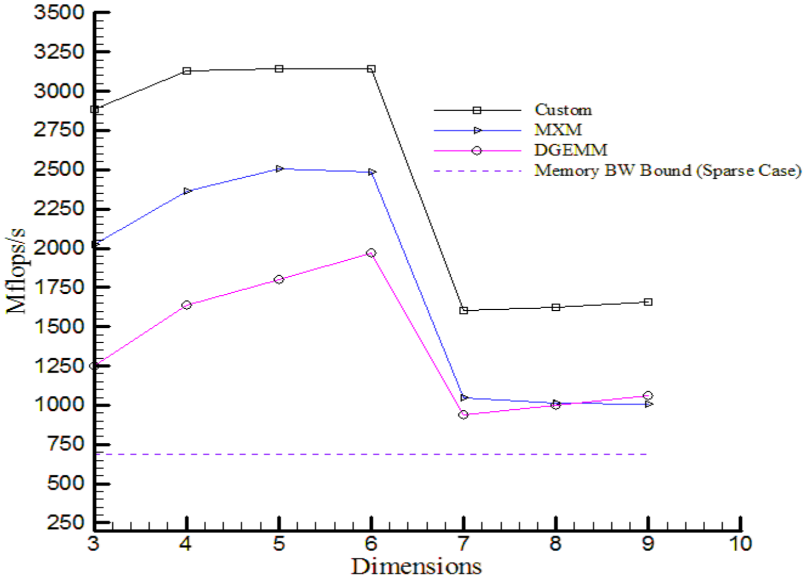

Fig. 5. Performance of the tensor matrix vector multiplication for n = 7 in all dimensions. The sharp drop in performance is due to the working set of the problem going out of the L2 cache (4 MB) of the Intel Madison processor. We are trying to contain this drop (to some extent) with better implementation (with extra blocking). Notice that the DGEMM() does not perform well for small values of matrix sizes and especially when the two matrix sizes are vastly different (large dimension case).

for $n = 7$. The DGEMM gives the worst performance of all for this case, especially for higher dimensions (when the matrix coming from the input vector becomes very elongated, e.g., 7×7^7 for eight dimensional problem). The custom code also shows sharp drop in performance (typically characteristic of the working set getting out of a fast memory level). We are trying some other implementations to reduce this performance drop.

Figure 6 shows the same scenario as in Figure 5 except that there are more mesh points (51) along the reaction coordinate than in the other directions (7). This is more consistent with the linear systems being solved in the CRP code (Figure 1). Again the performance is much better with the custom code than is possible with the corresponding sparse matrix-vector multiplication code (the dotted line in Figure 6).

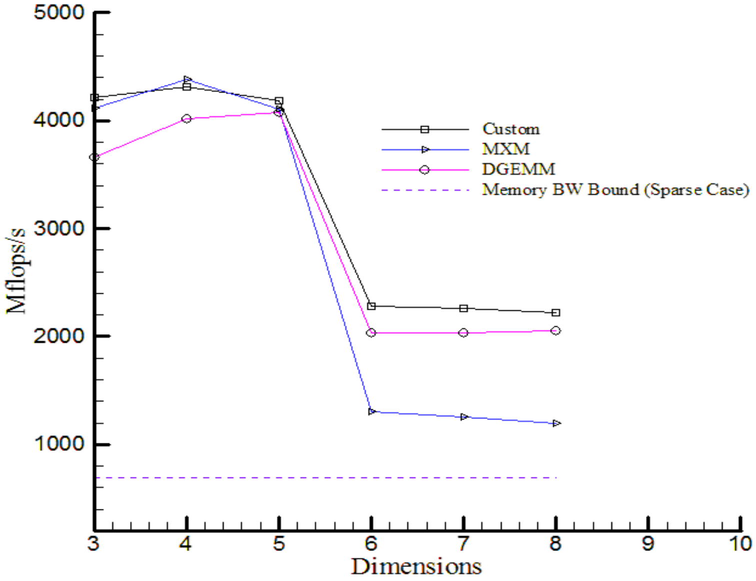

Fig. 6. This case has 51 points along the reaction path and 7 points in other dimensions. This represents the CRP code more closely. The performance advantage of the tensor matrix vector multiplication over the sparse approach is still maintained.

Storage Advantage

The chemistry codes work with many dimensions and are memory intensive for that reason. If we never form the large sparse system matrix, there is huge saving in memory. The memory needed for tensor representation of the operator in d dimensions is $O(dn^2)$ while it will be $O(n^d)$ we explicitly store it as sparse matrix. Therefore, the tensor product form of the operator will allow larger problems to be solved for the same amount of available memory.

6 Conclusions and Future Work

We have demonstrated the performance advantages of applying the system operator in the tensor product form (rather than as a sparse matrix). Since matrix-vector multiplication takes a large chunk of the overall execution time, we hope to see a big improvement in the overall performance of the CRP code if we can exploit the tensor product form of the operator. Our final paper will include the results of using this alternative approach. We are also evaluating some more competing implementations (like transposing the input vector for a more efficient evaluation of Type B terms, doing more blocking to contain the performance drops when the computation goes out of L2 cache, etc). Additionally, we will include the results of parallel implementation of the tensor matrix vector multiplication in our final paper.

Acknowledgments

We thank Paul Fischer, Ron Shepard, and Al Wagner of Argonne National Laboratory for many helpful discussions. The computer time was supplied by DOE (through Argonne, NERSC, and ORNL) and NSF (through Teragrid at SDSC).

References

1. S. Balay, K. R. Buschelman, W. D. Gropp, D. K. Kaushik, M. G. Knepley, L. C. McInnes, and B. F. Smith. PETSc home page. `http://www.mcs.anl.gov/petsc`, 2002.
2. M. O. Deville, P. F. Fischer, and E. H. Mund. *High-Order Methods for Incompressible Fluid Flow.* Cambridge University Press, 2002.
3. W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith. Toward realistic performance bounds for implicit CFD codes. In D. Keyes, A. Ecer, J. Periaux, N. Satofuka, and P. Fox, editors, *Proceedings of Parallel CFD'99*, pages 233–240. Elsevier, 1999.
4. U. Manthe and W. H. Miller. The cumulative reactions probability as eigenvalue problem. *J. Chem. Phys*, pages 3411–3419, 1999.
5. J. D. McCalpin. STREAM: Sustainable memory bandwidth in high performance computers. Technical report, University of Virginia, 1995. `http://www.cs.virginia.edu/stream`.
6. O. Temam and W. Jalby. Characterizing the behavior of sparse algorithms on caches. In *Proceedings of Supercomputing 1992*, pages 578–587. IEEE Computer Society, 1992.
7. S. Toledo. Improving the memory-system performance of sparse-matrix vector multiplication. *IBM Journal of Research and Development*, 41:711–725, 1997.
8. J. White and P. Sadayappan. On improving the performance of sparse matrix-vector multiplication. In *Proceedings of the 4th International Conference on High Performance Computing (HiPC '97)*, pages 578–587. IEEE Computer Society, 1997.