Toward Scalable Matrix Multiply on Multithreaded Architectures

Bryan Marker[1], Field G. Van Zee[2], Kazushige Goto[2], Gregorio Quintana-Ortí[3], and Robert A. van de Geijn[2]

[1] National Instruments
[2] The University of Texas at Austin
[3] Universidad Jaume I, Spain

Abstract. We show empirically that some of the issues that affected the design of linear algebra libraries for distributed memory architectures will also likely affect such libraries for shared memory architectures with many simultaneous threads of execution, including SMP architectures and future multicore processors. The always-important matrix-matrix multiplication is used to demonstrate that a simple one-dimensional data partitioning is suboptimal in the context of dense linear algebra operations and hinders scalability. In addition we advocate the publishing of low-level interfaces to supporting operations, such as the copying of data to contiguous memory, so that library developers may further optimize parallel linear algebra implementations. Data collected on a 16 CPU Itanium2 server supports these observations.

1 Introduction

The high-performance computing community is obsessed with efficient matrix-matrix multiplication (Gemm). This obsession is well justified, however; it has been shown that fast Gemm is an essential building block of many other linear algebra operations such as those supported by the Level-3 Basic Linear Algebra Subprograms (BLAS) [1, 2], the Linear Algebra Package (LAPACK) [3], as well as many other libraries and applications. Moreover, and perhaps more importantly, many of the issues that arise when implementing other dense linear algebra operations can be demonstrated in the simpler setting of Gemm.

The high-performance implementation of Gemm on serial and distributed memory architectures is well understood [4–8]. However, the implementation in the context of multithreaded environments, which includes Symmetric MultiProcessor (SMP), Non-Uniform Memory Access (NUMA), and multicore architectures, is relatively understudied. It is conceivable that architectures with hundreds of concurrent threads will be widespread within a decade, since each processor socket of an SMP or NUMA system will have multiple, possibly many, cores. Thus, issues of *programmability as well as scalability* must be addressed for Gemm and related operations.

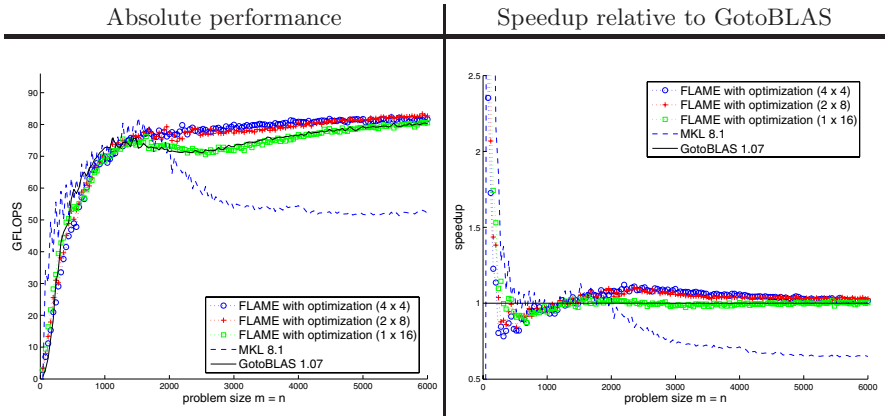

Fig. 1. Performance of our optimized implementations of GEMM when k is fixed at 256 on a 16 CPU Itanium2 server. (For further details, see Section 5.)

Our Formal Linear Algebra Methods Environment (FLAME) project studies these programmability[4] and scalability issues. The latter issue is the subject of this paper, which provides empirical evidence for the following observations:

- As was observed for distributed memory architectures in the late 1980s and early 1990s [11–14], work must be assigned to threads using a two-dimensional partitioning of data [15, 16, 10].
- In order to achieve near-optimal performance, library developers must be given access to routines or *kernels* that provide computational- and utility-related functionality at a lower level than the customary BLAS interface.

The first issue is of contemporary concern since a number of recent projects still use one-dimensional (1D) partitioning when parallelizing dense linear algebra operations for multithreaded environments [17]. The second is similarly important because it is tempting to simply call sequential BLAS within each thread once the work has been partitioned.

Figure 1 offers a preview of the performance observed when 2D partitioning and low-level optimizations are applied to a special case of parallel GEMM.

2 Partitioning for Parallel Panel-Panel Multiply

Consider the prototypical matrix multiplication $C := AB + C$ with $C \in \mathbb{R}^{m \times n}$, $A \in \mathbb{R}^{m \times k}$, and $B \in \mathbb{R}^{k \times n}$. In a recent paper [5], we argue that the case where k is chosen to be relatively small (in the $k = 256$ range) and m and n relatively large is the most important special case of this operation, for a number of reasons: (1) the general case (m, k, n of any size) can be cast in terms of this special case, (2) most operations supported by LAPACK can (and should) be cast in terms of this

[4] How the programmability issue is addressed by FLAME is discussed in [9, 10].

special case, and (3) this special case can achieve near-optimal performance. In the literature this case is often referred to as a rank-k update. We prefer calling it a general panel-times-panel multiply (GEPP). We will restrict the discussion to this special case.

Now we describe basic ideas behind the partitioning of computation among threads for parallel execution.

The usual, and simplest, way to parallelize the GEPP operation using t threads is to partition along the n dimension:

$$C = \left(C_0 \middle| \cdots \middle| C_{t-1} \right) \text{ and } B = \left(B_0 \middle| \cdots \middle| B_{t-1} \right)$$

where C_j and B_j consist of $n_j \approx n/t$ columns. Now, $C_j := AB_j + C_j$ can be independently computed by each thread. We will refer to this as a 1D work partitioning since only one of the three dimensions (m, n, and k) is subdivided. Clearly, there is a mirror 1D algorithm that partitions along m. In this paper we consider k to be too small to warrant further partitioning.

When distributed memory architectures became popular in the 1980s and 1990s, it was observed, both in theory and in practice, that as the number of processors increased, a 2D work and data partitioning is required to attain scalability[5] within dense linear algebra operations like GEMM. It is for this reason that libraries like ScaLAPACK [18] and PLAPACK [19] use a 2D data and work distribution. While the cost of data movement is much lower on multithreaded architectures, we suggest that the same basic principle applies to exploiting very large numbers of simultaneously executing threads.

To achieve a 2D work partitioning, t threads are logically viewed as forming a $t_r \times t_c$ grid, with $t = t_r \times t_c$. Then,

$$C = \left(\begin{array}{c|c|c} C_{0,0} & \cdots & C_{0,t_c-1} \\ \hline \vdots & & \vdots \\ \hline C_{t_r-1,0} & \cdots & C_{t_r-1,t_c-1} \end{array} \right), A = \left(\begin{array}{c} A_0 \\ \hline \vdots \\ \hline A_{t_c-1} \end{array} \right), \text{ and } B = \left(B_0 \middle| \cdots \middle| B_{t_c-1} \right),$$

with $C_{ij} \in \mathbb{R}^{m_i \times n_j}$, $A_i \in \mathbb{R}^{m_i \times k}$, and $B_j \in \mathbb{R}^{k \times n_j}$ ($m_i \approx m/t_r$ and $n_j \approx n/t_c$), so that each task $C_{i,j} := A_i B_j + C_{i,j}$ can be computed by a separate thread.

The purpose of this partitioning is to create parallelism through independent tasks that each perform a sequential GEPP.

3 Anatomy of the Sequential, High-performance GEPP Algorithm

The following is a minimal discussion of key internal mechanisms of the state-of-the-art GEMM implementation present within the GotoBLAS [5]. This discussion will allow us to explain how to reduce redundant memory-to-memory copies in the multithreaded implementation.

[5] Roughly speaking, scalability here should be understood as the ability to retain high performance as the number of processors is increased.

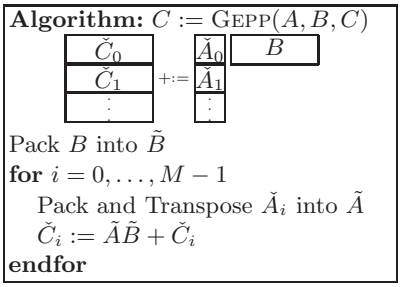

Fig. 2. Outline of optimized implementation of GEPP.

Consider again the computation $C := AB + C$ where the k dimension is relatively small. Assume for simplicity that $m = b_m M$ where b_m and M are integers. Partition

$$C = \left(\begin{array}{c} \check{C}_0 \\ \hline \vdots \\ \hline \check{C}_{M-1} \end{array}\right) \quad \text{and} \quad A = \left(\begin{array}{c} \check{A}_0 \\ \hline \vdots \\ \hline \check{A}_{M-1} \end{array}\right),$$

where $\check{C}_i$ and $\check{A}_i$ have b_m rows. Figure 2 gives a high-performance algorithm for the GEPP operation. This algorithm requires three highly optimized components, or low-level kernels:

- **Pack B:** A routine for packing B into a contiguous buffer[6].
- **Pack and transpose $\check{A}_i$:** A routine for packing $\check{A}_i$ into a contiguous buffer. Often this routine also transposes the matrix to improve the order in which data is accessed by the GEBP kernel routine. This transpose is orchestrated so that the packed and transposed matrix, $\tilde{A}$, is in the L2 cache upon completion.
- GEBP **kernel routine:** This routine computes $\check{C}_i := \tilde{A}\tilde{B} + \check{C}_i$ using the packed buffers. Here, GEBP stands for general block-times-panel multiply.

On current architectures the size of $\check{A}_i$ is chosen to fill about half of the L2 cache (or the memory addressable by the TLB), as explained in [5]. Considerable effort is required to tune each of these kernels, especially GEBP.

Two relevant key insights may be gathered from [5]. First, the cost of packing $\hat{A}_i$ is not much greater than the cost of loading the L2 cache, so that, while significant, it is also an unavoidable cost. This means that the column dimension of B should be large. Second, the cost of packing B is significant and should, therefore, be amortized over as many blocks of A as possible. This calls for a careful look at how packing occurs in a parallel implementation.

[6] Note that B is typically embedded in an array with more than k rows and is thus not contiguous.

4 Avoiding Redundant Packing in the Parallel GEPP

The description of the highly tuned GotoBLAS GEPP implementation brings to light the overhead that is incurred if one naively calls the `dgemm` library routine to perform each task: not only will there be redundant parameter checking, but there will also be redundant packing of B and/or submatrices $\check{A}_i$.

Redundant packing of submatrices $\check{A}_i$ is less of a concern for two reasons. First, redundant packing is cheaper because the copy is not necessarily written back to memory and ends up in the L2 cache. Second, creating redundant copies is unavoidable if each processing core has its own L2 cache; the submatrix has to be loaded into each L2 cache, regardless. Thus, if $t_r \times t_c = 1 \times t$, the naive approach that simply invokes the sequential `dgemm` routine can be expected to be quite effective.

The packing of matrix B into $\tilde{B}$ is a different matter. It is more expensive since typically B is large enough that this operation requires both reading from and writing to memory. Now, if $t_r > 1$, then the separate calls to `dgemm` performed by the jth column of threads would each repack B_j. The problem is potentially compounded by the fact that during this redundant packing there is contention for the limited bandwidth to memory. Thus, it can be argued that redundant packing of submatrices of B should be avoided, and, if possible, all available processor-bound threads should be employed during the packing of B.

5 Experiments

In this section we provide early evidence that the issues discussed in this paper are observed in practice. For this we employed a 16 CPU Itanium2 server. While the effects are somewhat limited when there are only 16 simultaneously executing threads, one would expect the effects to become more pronounced as systems utilize larger numbers of CPU cores, as is expected in the near future.

FLAME/C. The experiments were coded using the FLAME/C API, a programming interface that implements common linear algebra operations in an object-based environment [20]. The programmability issues in the multithreaded arena that FLAME/C solves are discussed in a recent paper [9].

Target platform. Experiments were performed on an SGI Altix ccNUMA system containing eight dual-processor Itanium2 nodes. Each pair of CPUs shares 4GB of local memory with the other CPUs to form a logically contiguous address space of 32GB. The Itanium2 microprocessor executes a maximum of four floating-point operations per clock cycle. All CPUs on this system run at 1.5GHz. This allows a peak performance of 6 GFLOPS (10^9 floating-point operations per second) per processor and an aggregate peak attainable performance of 96 GFLOPS, which is represented by the top range of the y-axes in the left column of graphs in Fig. 3. All computation was performed in double precision (64-bit) arithmetic.

Matrix sizes tested. For all experiments $k = 256$, a value for which the GotoBLAS implementation of `dgemm` is essentially optimal. Dimensions m and n were varied from 40 to 6000 in increments of 40. Matrices A, B, and C were stored in arrays that had a leading dimension equal to the maximum row dimension for our experiments (6000). This was done to ensure that the packing of the matrices captured what would typically occur when a GEPP operation is employed in practice.

Implementations tested. Since the $t_r \times t_c$ mesh of threads includes the 1D special cases of 1×16 and 16×1, our implementation assumed a logical $t_r \times t_c$ mesh and accordingly partitioned the matrices and computation. Two implementations were prepared:

- "FLAME without optimization." Partitions the work so that each thread performs a panel-panel multiply and also performs its own packing of B and $\check{A}_i$ by calling `dgemm`.
- "FLAME with optimization." Prepacks B (utilizing all 16 threads) so that redundant packing of B is avoided.

In addition, we timed the `dgemm` routine from the GotoBLAS (version 1.07), which views the threads as a 1×16 mesh on this architecture[7].

The basic kernels that provide high performance for the GotoBLAS are the exact same kernels used by our implementations.

Results. Figure 3 (left) shows the performance attained, in GFLOPS, by the different implementations. The right column of this figure shows speedup relative to the GotoBLAS `dgemm`. We only report results for 4×4, 2×8, and 1×16 partitionings since the remaining configurations yielded worse performance.

We point out that,

- For small matrices the prepacking of B for a 4×4 partitioning results in attenuated performance relative to a similar implementation that packs the matrix data redundantly. We speculate that this is because the packing operation leaves a part of $\tilde{B}$ in the L3 cache of the processor that packs it. When a thread on a different processor attempts to read this data to perform its calculations, the system's cache coherence protocol first requires the data to be written to main memory from the packing processor's L3 cache. Subsequently, the data may be read into the cache of a different CPU. We suspect that this memory update operation causes additional overhead that cannot be well amortized due to the small matrix dimensions.
- As dimensions m and n become large, it becomes worthwhile to configure the threads logically as a 2D mesh.

[7] Naturally, one can expect changes in future multithreaded implementations of the GotoBLAS consistent with the insights in this paper.

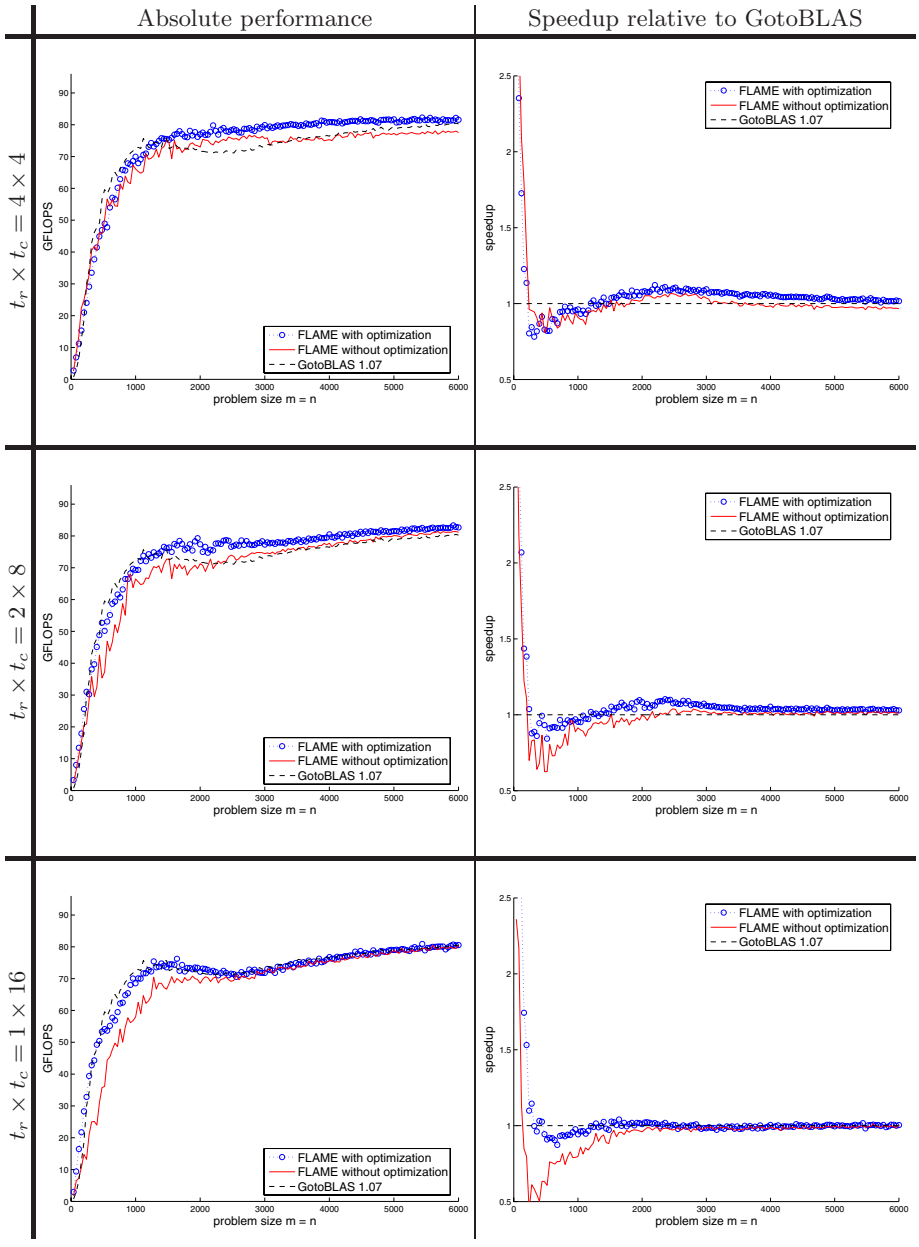

Fig. 3. Performance of conventional and low-level parallelizations (16 threads) of GEPP when m and n are varied and $k = 256$. Absolute performance is shown in the left column for three choices of $t_r \times t_c$ while the corresponding speedup relative to the GotoBLAS `dgemm` is shown on the right. This figure provides evidence that there is merit to (1) using a 2D partitioning of work and (2) avoiding redundant packing of B.

6 Conclusions

We have shown empirically that the parallelization of GEPP on multithreaded architectures with many threads benefits from a 2D decomposition and assignment of work. Moreover, we have demonstrated that it is important to avoid redundant copying of submatrices into contiguous memory.

While we have made similar observations about other linear algebra operations such as symmetric rank-k update [10] and Cholesky factorization [16], the study of GEPP presents the issues in isolation, devoid of tangential complexities such as dependencies (between subpartitions) and load-imbalance (due to matrices with special shape). Moreover, in many ways the Itanium2 server on which the experiments were performed is a much more forgiving architecture than more commonly encountered Pentium- or Opteron-based multithreaded architectures due to its massive memory bandwidth. Finally, given that 16 simultaneously executing threads represents a relatively small level of parallelism, the effects observed here will likely become more dramatic as multicore systems are designed and built with more CPUs.

We envision a range of further studies and development. The GEPP operation is the most important of three frequently used cases of GEMM, which also include GEMP (multiplication of a matrix times a panel of columns) and GEPM (multiplication of a panel of rows times a matrix). Similar experiments on multicore processors are in order given that these systems employ memory architectures different from that of single-core-per-socket SMPs. Interfaces to lower level kernels should be published to help other researchers perform similar experiments more easily. Furthermore, *de facto* standards of such interfaces would allow library developers to realize the performance gains demonstrated in this paper across platforms.

Further information

For additional information regarding the FLAME project, visit
`http://www.cs.utexas.edu/users/flame/`.

Acknowledgements

This research was partially sponsored by NSF grants CCF-0540926, CCF-0342369, and ACI-0305163. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation (NSF). In addition, Dr. James Truchard (National Instruments) provided an unrestricted grant to our research. Furthermore, the research of Bryan Marker was partially funded by the Undergraduate Research Opportunities Program of the Department of Computer Sciences at The University of Texas at Austin.

Access to the 16 CPU Itanium2 (1.5 GHz) system on which the experiments were performed was provided our collaborators at Universidad Jaume I, Spain. Initial experiments (not reported in this paper) were conducted on a 4 CPU

Itanium2 server which was donated to our project by Hewlett-Packard and is administered by UT-Austin's Texas Advanced Computing Center.

As always, we are indebted to our collaborators on the FLAME team.

References

1. Dongarra, J.J., Du Croz, J., Hammarling, S., Duff, I.: A set of level 3 basic linear algebra subprograms. ACM Trans. Math. Soft. **16**(1) (March 1990) 1–17
2. Kågström, B., Ling, P., Loan, C.V.: GEMM-based level 3 BLAS: High performance model implementations and performance evaluation benchmark. ACM Trans. Math. Soft. **24**(3) (1998) 268–302
3. Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J., Croz, J.D., Greenbaum, A., Hammarling, S., McKenney, A., Ostrouchov, S., Sorensen, D.: LAPACK Users' Guide - Release 2.0. SIAM (1994)
4. Gunnels, J.A., Henry, G.M., van de Geijn, R.A.: A family of high-performance matrix multiplication algorithms. In Alexandrov, V.N., Dongarra, J.J., Juliano, B.A., Renner, R.S., Tan, C.K., eds.: Computational Science - ICCS 2001, Part I. Lecture Notes in Computer Science 2073, Springer-Verlag (2001) 51–60
5. Goto, K., van de Geijn, R.: Anatomy of high-performance matrix multiplication. ACM Trans. Math. Soft. to appear.
6. Agarwal, R.C., Gustavson, F., Zubair, M.: A high-performance matrix multiplication algorithm on a distributed memory parallel computer using overlapped communication. IBM Journal of Research and Development **38**(6) (1994)
7. van de Geijn, R., Watts, J.: SUMMA: Scalable universal matrix multiplication algorithm. Concurrency: Practice and Experience **9**(4) (April 1997) 255–274
8. Gunnels, J., Lin, C., Morrow, G., van de Geijn, R.: A flexible class of parallel matrix multiplication algorithms. In: Proceedings of First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing (1998 IPPS/SPDP '98). (1998) 110–116
9. Low, T.M., Milfeld, K., van de Geijn, R., Zee, F.V.: Parallelizing FLAME code with OpenMP task queues. Technical Report TR-04-50, The University of Texas at Austin, Department of Computer Sciences (December 2004)
10. Van Zee, F.G., Bientinesi, P., Low, T.M., van de Geijn, R.A.: Scalable parallelization of FLAME code via the workqueuing model. ACM Trans. Math. Soft. (2007) submitted.
11. Stewart, G.: Communication and matrix computations on large message passing systems. Parallel Computing **16** (1990) 27–40
12. Lichtenstein, W., Johnsson, S.L.: Block-cyclic dense linear algebra. Technical Report TR-04-92, Harvard University, Center for Research in Computing Technology (Jan. 1992)
13. Hendrickson, B.A., Womble, D.E.: The torus-wrap mapping for dense matrix calculations on massively parallel computers. SIAM J. Sci. Stat. Comput. **15**(5) (1994) 1201–1226
14. Dongarra, J., van de Geijn, R., Walker, D.: Scalability issues affecting the design of a dense linear algebra library. J. Parallel Distrib. Comput. **22**(3) (Sept. 1994)
15. Addison, C., Ren, Y.: OpenMP issues arrising in the development of parallel BLAS and LAPACK libraries. In: EWOMP. (2001)
16. Chan, E., Ortí, E.S.Q., Ortí, G.Q., van de Geijn, R.: Supermatrix out-of-order scheduling of matrix operations for smp and multi-core architectures. (2007) submitted to SPAA.

17. Kurzak, J., Dongarra, J.: Implementing linear algebra routines on multi-core processors with pipelining and a look ahead. LAPACK Working Note 178 UT-CS-06-581, University of Tennessee (September 2006)
18. Blackford, L.S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., Whaley, R.C.: ScaLAPACK Users' Guide. SIAM (1997)
19. van de Geijn, R.A.: Using PLAPACK: Parallel Linear Algebra Package. The MIT Press (1997)
20. Bientinesi, P., Quintana-Ortí, E.S., van de Geijn, R.A.: Representing linear algebra algorithms in code: The FLAME APIs. ACM Trans. Math. Soft. **31**(1) (March 2005) 27–59