Automatic Tuning of PDGEMM Towards Optimal Performance

Sascha Hunold and Thomas Rauber

Department of Mathematics and Physics
University of Bayreuth, Germany
{hunold,rauber}@uni-bayreuth.de

Abstract. Sophisticated parallel matrix multiplication algorithms like PDGEMM exhibit a complex structure and can be controlled by a large set of parameters including blocking factors and block sizes used for the serial execution on one of the participating processors. But it requires a deep understanding of both the parallel algorithm and the execution platform to select the parameters such that a minimum execution time results. In this article, we describe a simple mechanism that automatically selects a suitable set of parameters for PDGEMM which leads to a minimum execution time in most cases.

1 Introduction

There is usually a complex dependency between the computations and the memory accesses performed by a computation-intensive program, the required data exchanges between neighboring processors, and the computation and communication characteristics of the execution platform. Parallel numerical libraries like ScaLAPACK (Scalable LAPACK) [1, 2] or PETSc [3] try to cope with these dependencies by providing a set of parameters which allow the user to adjust the execution behavior of the library routines to the characteristics of the execution platform such that the parallel execution time is reduced as far as possible. By selecting appropriate parameter values, the library routines can run very efficiently on most parallel execution platforms. But it is often quite difficult for the user to select suitable parameter values, since this requires a deep understanding of the algorithmic behavior of the library routines. In many situations, the user wants to use the library as black-box and does not have time to learn more about the internals of the algorithm. Moreover, even knowing the algorithmic details of a library routine does not necessarily yield a suitable set of parameters to use. The complex dependency between the algorithm and the characteristics of the execution platform is still present and it is usually necessary to perform runtime experiments with different parameter settings before a suitable set of parameters can be identified. It even might be the case that for different numbers of processors different parameter values lead to the best performance.

In this situation, it would be useful to have a tool that automatically selects a suitable set of parameters, thus relieving the user from this time-consuming

J.C. Cunha and P.D. Medeiros (Eds.): Euro-Par 2005, LNCS 3648, pp. 837–846, 2005.

work. In this article, we address this issue. In particular, we consider the matrix-matrix multiplication routine PDGEMM of ScaLAPACK. PDGEMM is part of PBLAS which is the parallel implementation of BLAS (Basic Linear Algebra Subprograms) for distributed memory machines. It can be used as a building block in a parallel version of Strassen's algorithm [4] as well as in many advanced algorithms from scientific computing. We investigate which parameters have a major impact on the overall performance and should therefore be considered in more detail. In particular, we present an approach to tune the PDGEMM routine by adjusting the parameters gradually to the given execution platform. The Automatically Tuned Linear Algebra Software (ATLAS) [5] provides an approach for the sequential case, but there is no solution for a parallel execution yet. An experimental evaluation shows that the proposed method selects a parameter setting that leads to significant performance gains compared to the default setting for different test platforms.

The rest of the paper is organized as follows. Section 2 gives an overview of the algorithmic details of the PDGEMM routine and shows examples of how to use PDGEMM routines in a *C* environment. Section 3 analyzes the impact of different parameters for PDGEMM. In Section 4 we describe an heuristic method for an automatic selection of suitable PDGEMM parameters to optimize PDGEMM. Section 5 evaluates the experimental results and Section 6 concludes.

2 Algorithmic Details

To improve the performance of the PDGEMM routine efficiently we must first consider some algorithmic details. The PDGEMM routine which is part of ScaLAPACK is derived from the DIMMA algorithm (Distribution-Independent Matrix Multiplication). DIMMA is an enhanced version of SUMMA (Scalable Universal Matrix Multiplication Algorithm), see [6] for a detailed description of SUMMA and [7] for an introduction of DIMMA. We also refer to [8] for an overview of basic parallel matrix-matrix multiplication algorithms such as the algorithms of Cannon or Fox.

In the following, we summarize the basic ideas which make PDGEMM (using DIMMA) a well-performing algorithmic option in many cases. An example of the starting configuration of matrix A for DIMMA and SUMMA is shown in Figure 1. In SUMMA, the processors P_0 and P_3 broadcast the first column of A along their row, i.e., P_0 sends its first column to processors P_1 and P_2. At the same time, the first row of matrix B which is distributed likewise is broadcasted along the processor columns. When the broadcasts are performed on a logical ring, SUMMA takes advantage of a pipelined communication scheme. The authors of DIMMA state that SUMMA contains extra waiting times between two communication procedures [7]. Hence, DIMMA improves the communication scheme and eliminates the extra waiting time by proceeding to send blocks of columns (rows) from the current column (row) of the processor grid. That means, in SUMMA the processors P_1 and P_4 broadcast column 1 directly after receiving column 0 from P_0 and P_3, respectively. In case of DIMMA, P_0 and P_3 continue with broadcasting another column whose distance is LCM blocks where LCM

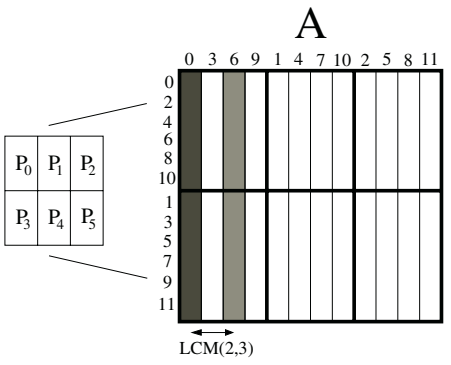

Fig. 1. DIMMA snapshot for a 2×3 processor grid. DIMMA uses a block cyclic distribution of matrix A onto the processors.

is the least common multiple of the grid dimensions p and q. For a further performance improvement, SUMMA as well as DIMMA use blocks of columns (rows) rather than single columns (rows).

Since it is not very straightforward to use ScaLAPACK routines from C we would like to demonstrate the calling conventions from within C. We indicate that this method is compiler specific. Because Fortran 77 uses the call-by-reference paradigm we need to pass the address of each parameter to Fortran functions. The basic problem of calling Fortran routines from C is the conversion of strings. Figure 2 shows a sample call of PDGEMM from C for our compiler collection, see Table 1. The function `c2f_char` copies a character into a character buffer. Since there is no header file for C, an underscore is required for the linker to resolve the PDGEMM function. But primarily we want to emphasize the use of hidden parameters. The last two arguments of the call to PDGEMM are hidden parameters because they need to be passed in order to make the Fortran function work but the arguments are not part of PDGEMM's Fortran interface. The hidden parameters `1L` denote the length of the passed strings, i.e. the length of `cha` and `chb`. If we had a function with three string parameters we would need to pass three hidden parameters as well.

```
c2f_char(cha, 'N');
c2f_char(chb, 'N');
pdgemm_( cha, chb, &m, &k, &n, &scalar, a[0],
         &one, &one, &desca, b[0], &one, &one, &descb,
         &scalar, c[0], &one, &one, &descc, 1L, 1L );
```

Fig. 2. Call to PDGEMM from C.

3 Parameter Analysis and Optimization Strategies

In this section we examine the performance dependencies of PDGEMM from different parameters. The experiments were performed on a cluster with 32 Dual Opteron nodes. An overview of the configuration is given in Table 1.

Table 1. System configuration used for experiments.

System	32 node cluster (each node equipped with 2 AMD Opterons) Linux 2.4.21
Processor	Opteron 244, 1.8 GHz, 128 KB L1-Cache, 1024 KB L2-Cache
C/F77 Compiler	GCC 3.4.0
MPI version	MPICH 1.2.5 + VMI 2.0 (Infiniband)
Infiniband driver	Mellanox HPC Gold Collection (IBHPC) v0.5.0 for Linux Mellanox THCA for Linux 3.2-rc17
ScaLAPACK	1.7
ATLAS	3.6.0

As the sequential computation of PDGEMM is based on BLAS [9], the right choice of the BLAS implementation is crucial for the overall parallel performance of PDGEMM. There is a tremendous performance difference between hardware-optimized BLAS routines and the standard routines. When the user has no access to a vendor-provided BLAS library like ESSL, we recommend using the ATLAS library [5]. All local computations used in the evaluation of the parallel algorithms are performed by ATLAS. Moreover, each experiment reported herein was repeated at least three times.

Since all variables in the parameter list of PDGEMM and also the logical block size defined inside ScaLAPACK may influence the runtime, the search space for optimization is extremely large. To obtain a satisfactory parallel performance, it is necessary to use a local computation kernel which almost achieves the peak performance of the processor.

The parameters with the biggest influence on the performance of the algorithm are the dimensions of the input matrices, the number of processors and their arrangement within the processor grid.

There are also other parameters that strongly influence the MFLOPS rate of the algorithm but are not obvious. These are

1. the three blocking factors mb, nb and kb of the block-cyclic distribution of matrices A, B and C which are of size $m \times k$, $k \times n$ and $m \times n$,
2. and the logical block size lb.

Blocking factor The blocking factor is used to distribute the rows and columns of the matrices onto the processor grid. A blocking factor of b means that blocks of matrix $\mathcal{M}$ of size $b \times b$ are distributed block-cyclicly. It is also possible to have distinct blocking factors for each matrix dimension.

Logical block size The logical block size denotes the size of the sub-matrix of C which is computed by each processor per parallel step of PDGEMM. Let the logical block size be lb. In each parallel step, a processor P_i gathers lb rows of A and lb columns of B and computes a part of the result matrix C of size $lb \times lb$.

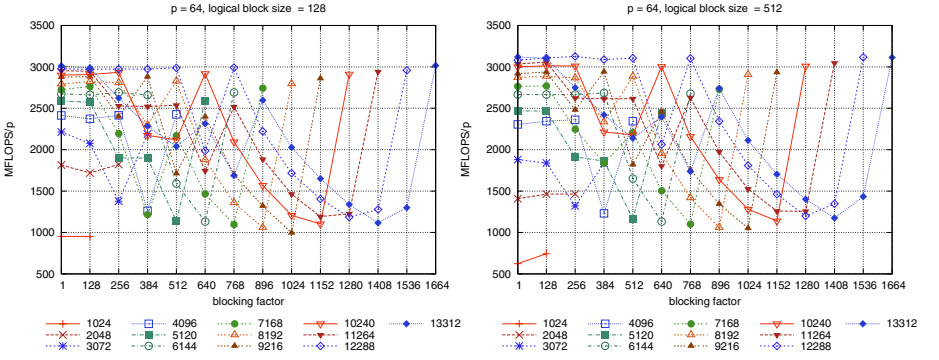

Fig. 3. Performance of PDGEMM, number of processors $p = 64$ for matrix sizes from 1024 to 13312 for different blocking factors.

3.1 Impact of the Blocking Factor

In order to analyze the performance dependency on the blocking factor, we have performed several tests of PDGEMM with varying block sizes. Due to the huge number of degrees of freedom we limited the test cases to square input matrices and square matrix blocks where $mb = nb = kb$. We performed numerous tests with blocking factors ranging from 1 to a logical maximum which is defined by the matrix dimensions and the processor grid. Furthermore, we also examined how the logical block size is reflected in the runtime of PDGEMM for each range. Figure 3 shows the performance of PDGEMM for 64 processors and for different blocking factors using logical block sizes of 128 and 512. The experiment was repeated for 8, 16 and 32 processors as well, using logical block sizes of 32, 64, and 256. We observed that the coarse characteristics of the resulting MFLOPS rate does not depend on the logical block size, but the MFLOPS rate is only slightly increased (decreased) for a smaller (larger) value of the logical block size. Hence, big differences in the MFLOPS like at 1536 and 1664 in Figure 3 can not be compensated by adjusting only the logical block size. But let us have a closer look at Figure 3. The MFLOPS rates for matrix dimension 13312 show peaks for blocking factors 1, 128 and 1664. This behavior can be explained as follows: The 64 processors are arranged in a grid of 8×8 elements. Thus, in each dimension the matrix is distributed evenly among the processors if $\frac{13312}{8} = 1664$ is a multiple of the blocking factor. And indeed, 1664 is a multiple of 1, 128 and 1664. In these cases, the data is uniformly distributed over all processors in the grid which leads to a balanced workload on homogeneous systems. So, when in doubt which blocking factor to use, it is a good choice to use the possible maximum $\mathcal{M}$ which is $\mathcal{M} = \frac{\text{matrix dimension}}{\#\text{ processors in row (colum)}}$.

It is not surprising that high performance is achieved when the matrix dimensions are multiples of the blocking factor. But choosing the blocking factor appropriately does not necessarily lead to best performance. The central parameter to optimize is therefore the logical block size.

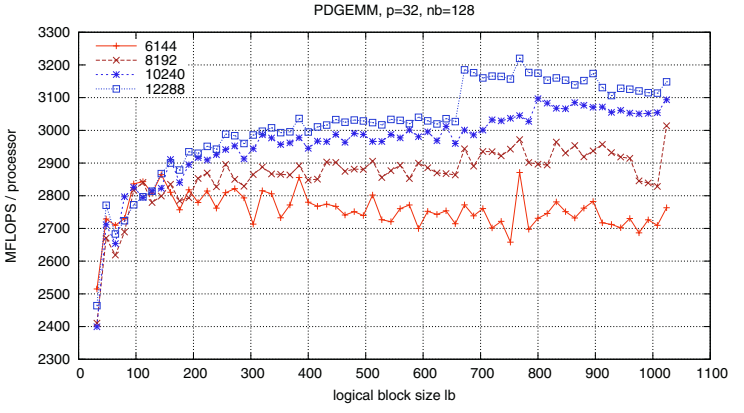

Fig. 4. MFLOPS achieved by PDGEMM with different logical block sizes, number of processors $p = 32$, blocking factor $nb = 128$ (Infiniband).

3.2 Impact of the Logical Block Size

The logical block size directly influences the overall performance of PDGEMM. A small value of the logical block size (lb) will not only cause more communication but worse, the local matrix updates (multiplications) will not reach the processor's peak performance. On the other hand, choosing a very large value may hamper the pipelined communication scheme and so the overlapping of communication and computation as well.

Finding the best value for the logical block size is highly machine-dependent and the impact on the resulting execution time can only be determined experimentally. We ran a series of tests with PDGEMM on the cluster system for varying values of lb. Since the value lb is hard-coded in file `pilaenv.f` of the ScaLAPACK distribution, the value lb in `pilaenv.f` needs to be changed and ScaLAPACK must be recompiled for each test. The results of this experiment is shown in Figure 4 and Figure 5. Figure 4 contains the MFLOPS rate achieved by PDGEMM using the Infiniband network. Figure 5 shows the results for the Gigabit Ethernet. We can observe that the plots in both figures have similar characteristics, i.e., the bandwidth and latency of the interconnection network plays a minor role. As example, we consider the steep increase of the MFLOPS rate at block size 672 for a matrix dimension of 12288. The 32 processors are arranged in a 4×8 grid and each processor stores $\frac{12288}{4} = 3072$ rows and $\frac{12288}{8} = 1536$ columns of the matrices, if a suitable blocking factor nb has been chosen. Let us examine the case where the biggest performance enhancement has been observed. For $lb = 672$, PDGEMM performs a series of local matrix-multiplications using DGEMM where the matrix A is of size 3072×672 and Matrix B has 672×1536 elements. One local matrix update with these parameters achieves about 3508 MFLOPS on a single Opteron processor. In comparison, with a logical block size of 656 the routine DGEMM achieves 3380 MFLOPS only, which is about 4%

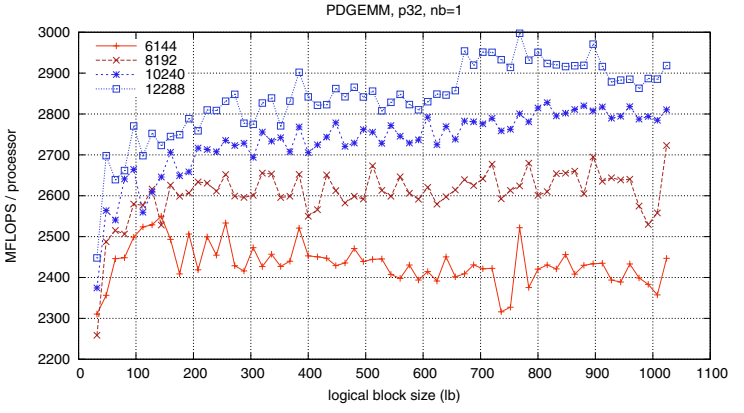

Fig. 5. MFLOPS achieved by PDGEMM with different logical block sizes, number of processors $p = 32$, blocking factor $nb = 1$ (Gigabit Ethernet).

slower. We measured the cache misses generated by DGEMM using PAPI [10]. Figure 6 clearly shows that the weak performance is the result of producing more L2 cache misses which is caused by the fact that DGEMM (ATLAS) generates a different call tree for $lb = 656$ and $lb = 672$.

Instead of ignoring possible drops in the MFLOPS rate, we present an approach to avoid choosing an unfavorable logical block size which is discussed in Section 4.

4 Automatic Parameter Tuning

In this section, we present an approach for selection a favorable logical block size. The basic problem of optimizing the logical block size is the huge search space. The logical block size depends on the network parameters, the dimension of the matrices, the matrix ordering, the processor grid, the number of processors and the BLAS implementation. Additional informations about the hardware, e.g. cache size, will surely decrease the search space but it remains too large to evaluate all possible combinations.

We present an heuristic method based on an evaluation on a single processor of the parallel execution platform. The approach is simple, but it turns out to be fast and provides a suitable logical block size for the parallel case. The algorithm keeps two matrix dimensions fixed and varies the third one. The size of the matrix dimensions which are kept fixed is computed by the information provided by the user, e.g. the number of processors, the typical size of matrices and the preferred processor grid. The third and varying dimension represents the logical block size. After the series of tests on a single processor has been completed, the optimization algorithm selects the smallest logical block size which is only less than a fixed percentage f slower than the best logical block size (which is in general the largest, but, as we said before, for the parallel execution the largest

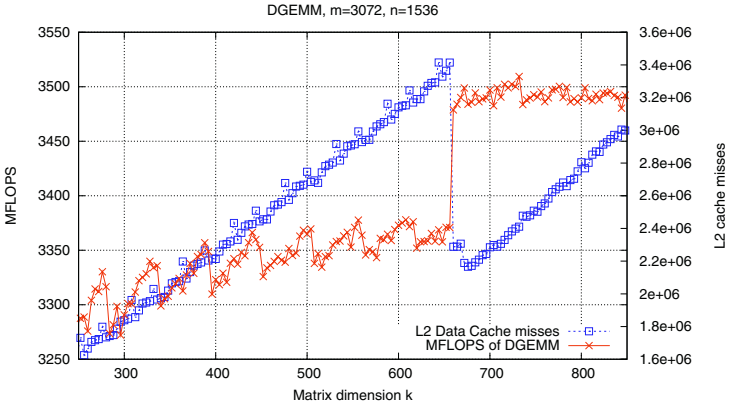

Fig. 6. MFLOPS vs. L2 cache misses by DGEMM (ATLAS).

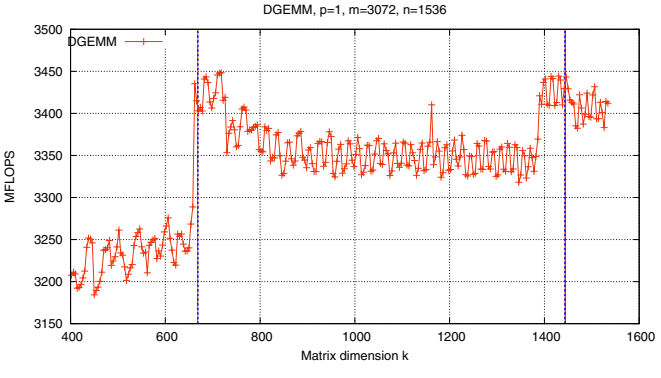

Fig. 7. MFLOPS by DGEMM (ATLAS). Vertical lines at 668 and 1444.

logical block size is not always the fastest). For a specific parallel platform, the algorithm only needs to run once on one processor in a pre-computation phase to determine a suitable value of the logical block size. Then, this block size can be used for all parallel executions. In our experiments, we obtained the best performance for $f = 2\%$.

5 Experimental Evaluation

We now consider the experimental evaluation of the proposed method on the cluster system from Table 1. The cluster consists of 32 processors and we want to use a rectangular grid, e.g. 4 rows and 8 columns. The typical size of matrices is set to 12288 in each dimension. Hence, PDGEMM will deal with submatrices of size $3072 \times lb$ and $lb \times 1536$. The tuning algorithm will test all possible logical block sizes in the range $2 \ldots 1536$ (odd numbers are not considered). The test results are shown in Figure 7. The function line has a maximum at 1444. The

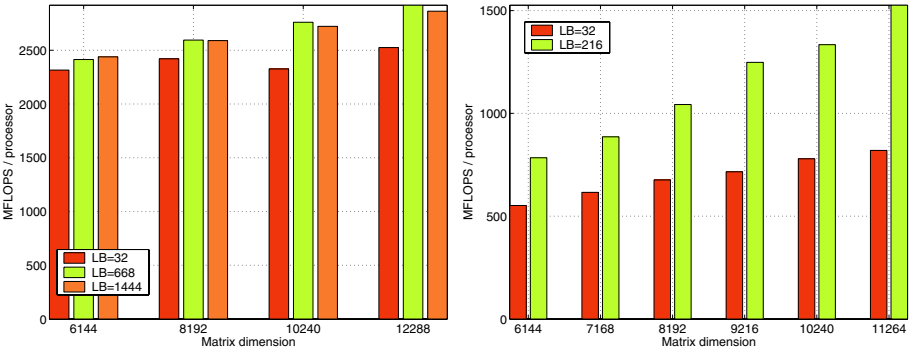

Fig. 8. MFLOPS achieved by PDGEMM on 32 processors. Left: results for Opteron cluster (Infiniband) for logical block size $lb = 32, 668, 1444$; right: performance comparison of PDGEMM for a logical block size of 32 and 216 on Xeon cluster (SCI).

smallest matrix dimension which is less than 2% slower than 1444 is 668. We marked both values in the plot. The value of 668 is the optimized logical block size. Surprisingly, the value that has been found is very close to the crucial value of 656. For a final evaluation of the logical block size, Figure 8 (left) compares the performance of PDGEMM achieved with the default value of the logical block size ($lb = 32$) and with the automatically selected value of 668. The plot also includes the MFLOPS rates for a logical block size of 1444 for comparison reasons. It can be observed that the block size which achieves best results on a single processor is not necessarily gaining the maximum performance in parallel. For matrix dimensions 10240 and 12288 in Figure 8 on the left, the performance gain for the automatically selected logical block size is 18% and 15%, respectively. Additional tests have been performed on a cluster consisting of 16 nodes (Dual Xeon 2 GHz). The nodes are running Linux and are connected via an SCI network. The resulting MFLOPS achieved by PDGEMM for this cluster are shown on the right-hand side of Figure 8. The tuning algorithm selects a logical block size of 216. This block size clearly outperforms the default settings of PDGEMM. In this experiment, the automatically selected value of $lb = 216$ reduces the runtime of PDGEMM by up to 47% for a matrix dimension of 11264.

6 Conclusions

The performance of ScaLAPACK routines strongly depends on the logical block size. In this article we have shown how to use the function PDGEMM and how to improve its performance by selecting a well-suited blocking factor and a logical block size automatically. The experimental results confirm that the heuristic method of selecting a logical block size leads to a significant performance enhancement of PDGEMM.

References

1. 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: A Linear Algebra Library for Message-Passing Computers. In: Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing (Minneapolis, MN, 1997), Philadelphia, PA, USA, Society for Industrial and Applied Mathematics (1997) 15 (electronic)
2. Choi, J., Demmel, J., Dhillon, I., Dongarra, J., Ostrouchov, S., Petitet, A., Stanley, K., Walker, D., Whaley, R.C.: ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory Computers - Design Issues and Performance. Technical report, Knoxville, TN 37996, USA (1995)
3. Balay, S., Gropp, W.D., McInnes, L.C., Smith, B.F.: PETSc 2.0 User Manual. Argonne National Laboratory, http://www.mcs.anl.gov/petsc/. (1997)
4. Hunold, S., Rauber, T., Rünger, G.: Multilevel Hierarchical Matrix Multiplication on Clusters. In: Proceedings of the 18th Annual ACM International Conference on Supercomputing, ICS'04. (2004) 136–145
5. Whaley, R.C., Dongarra, J.J.: Automatically Tuned Linear Algebra Software. Technical Report UT-CS-97-366, University of Tennessee (1997)
6. Geijn, R.A.V.D., Watts, J.: SUMMA: Scalable Universal Matrix Multiplication Algorithm. Concurrency: Practice and Experience **9** (1997) 255–274
7. Choi, J.: A New Parallel Matrix Multiplication Algorithm on Distributed-Memory Concurrent Computers. Concurrency: Practice and Experience **10** (1998) 655–670
8. Golub, G.H., Van Loan, C.F.: Matrix Computations, Third Edition. John Hopkins University Press (1998)
9. Dongarra, J., Croz, J.D., Hammarling, S., Duff, I.: A Set of Level 3 Basis Linear Algebra Subprograms. ACM Transactions on Mathematical Software **16** (1990) 1–17
10. Browne, S., Dongarra, J., Garner, N., London, K., Mucci, P.: A Scalable Cross-Platform Infrastructure for Application Performance Tuning Using Hardware Counters. In: Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM), IEEE Computer Society (2000) 42