Optimizing OpenMP Parallelized DGEMM Calls on SGI Altix 3700

Daniel Hackenberg, Robert Schöne, Wolfgang E. Nagel, and Stefan Pflüger

Technische Universität Dresden,
Center for Information Services and High Performance Computing (ZIH),
01062 Dresden, Germany
{daniel.hackenberg,robert.schoene,wolfgang.nagel,stefan.pflueger}
@zih.tu-dresden.de
http://www.tu-dresden.de/zih

Abstract. Using functions of parallelized mathematical libraries is a common way to accelerate numerical applications. Computer architectures with shared memory characteristics support different approaches for the implementation of such libraries, usually OpenMP or MPI.
This paper's content is based on the performance comparison of DGEMM calls (floating point matrix multiplication, double precision) with different OpenMP parallelized numerical libraries, namely Intel MKL and SGI SCSL, and how they can be optimized. Additionally, we have a look at the memory placement policy and give hints for initializing data. Our attention has been focused on a SGI Altix 3700 Bx2 system using BenchIT [1] as a very convenient performance measurement suite for the examinations.

1 Measurement Environment

For a detailed analysis of a system architecture by parameter studies, the choice of a suitable measuring framework is an important decision. To benchmark the DGEMM calls we use BenchIT. This performance measurement suite helps to compare different algorithms, implementations of algorithms, features of the software stack, and hardware details of whole systems. It has been designed to run many microbenchmarks on every POSIX 1.003 compatible system in a very user-friendly way. BenchIT has been developed at the Center for Information Services and High Performance Computing (ZIH) at the Technische Universität Dresden and was previously mentioned at [2–4]. Sources and results are freely available at [1].

2 The SGI Altix 3700 Bx2 System

2.1 System Architecture

The SGI [5] Altix 3700 Bx2 is a ccNUMA shared memory system based on Intel Itanium 2 processors and SGI's scalable node architecture SN2. In developing this, special attention has been paid to building a highly scalable computer

with large bandwidths on all data paths. It provides cache coherency in one coherent sharing domain (CSD), which runs a Linux kernel and can scale up to 512 processors each. A single processor is operating at 1.5 GHz and therefore a maximum floating point performance of 6 GFLOPS can be reached. More information, especially about the SGI bricks and application benchmarks, can be found at [6].

2.2 First Touch Policy

In contrast to former SGI systems like the Origin 3800, the Altix does not move data near the processor which is using it most. Instead, it uses the so-called first touch policy which means that data is placed next to the processor that writes to it first. This may have no effect if the application was parallelized with MPI, and the data was spread manually. Multithreaded programs usually don't spread data because all addresses can be accessed from every thread.

In the worst case, all data is placed in just one memory module when all other OMP threads want to access it. The remaining bandwidth for each thread would shrink to b/p, where b is the bandwidth of a single memory module and p the number of participating processors and threads respectively.

3 Optimizing the DGEMM Call

The `cblas_dgemm` call is defined as shown in Listing 1.1. A description of the parameters can be found at [7].

```
void cblas_dgemm(
        const enum CBLAS_ORDER Order,
        const enum CBLAS_TRANSPOSE TransA,
        const enum CBLAS_TRANSPOSE TransB,
        const int M, const int N,
        const int K, const double alpha,
        const double *A, const int lda,
        const double *B, const int ldb,
        const double beta,
        double *C, const int ldc);
```

Listing 1.1. `cblas_dgemm` call declaration

A simple `cblas_dgemm` call for a matrix multiplication for two square matrices of order *size* would look like this:

```
cblas_dgemm(CblasRowMajor,CblasNoTrans,CblasNoTrans,
        size, size, size, 1.0, A, size, B,
        size, 1.0, C, size);
```

Listing 1.2. Simple matrix multiplication using `cblas_dgemm`

The following sections will explain how these DGEMM calls can be parallelized and optimized to scale well up to at least 124 processors on the SGI Altix 3700.

3.1 The BenchIT DGEMM Kernel

As previously mentioned, BenchIT is a measurement environment which helps to examine a system with microbenchmarks. Some of these execute a sequential matrix multiplication with different libraries. During each measuring run, which means one performance measurement for one problem size, the matrices are allocated and filled with variables. After the measurement the allocated memory is being released. Therefore, each measurement consists of three steps: initializing the data, recording the duration of processing, and destroying data. The problem sizes that are to be measured can be set as parameters for the microbenchmark.

The original hardware vendor mathematical libraries available on SGI Altix 3700 are the Intel Math Kernel Library - MKL 8.0 [8] and the SGI Scientific Computing Software Library - SCSL 1.6.1.0 [9]. They both nearly reach peak performance on a single processor (Fig. 1).

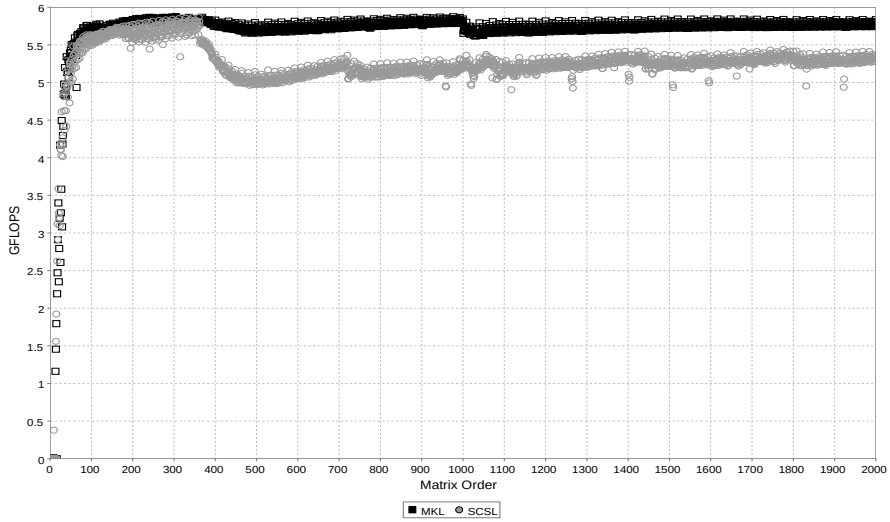


Fig. 1. MKL and SCSL DGEMM performance on Intel Itanium 2

3.2 Library Provided OpenMP Parallelization

The DGEMM calls can easily be parallelized by setting the environment variable OMP_NUM_THREADS. This implicit parallelization is provided by both the MKL and SCSL, using OpenMP to handle different threads in order to calculate the matrix multiplication faster.

Fig. 2 shows a nearly linear speedup for 2, 4, and 8 processors. The graph for 16 CPUs in Fig. 3 also shows acceptable performance, but using 32 or 64

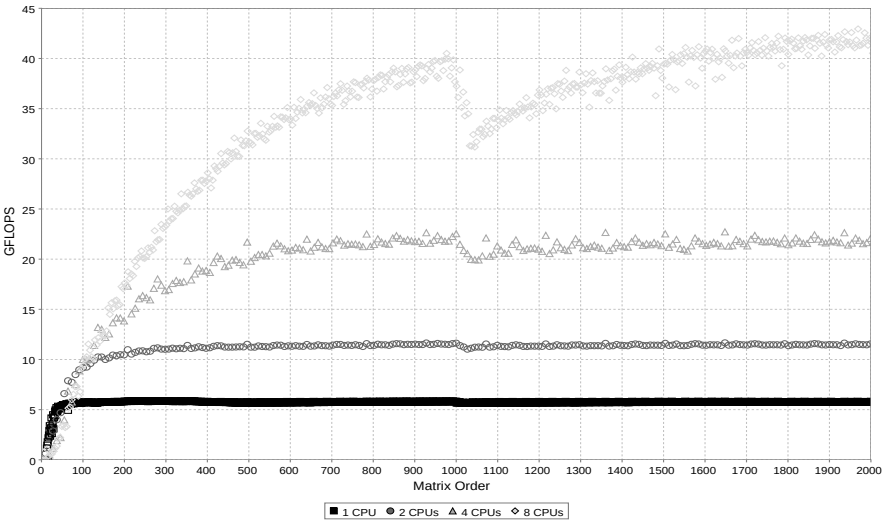


Fig. 2. MKL DGEMM performance on Itanium 2 for 1, 2, 4, and 8 CPUs

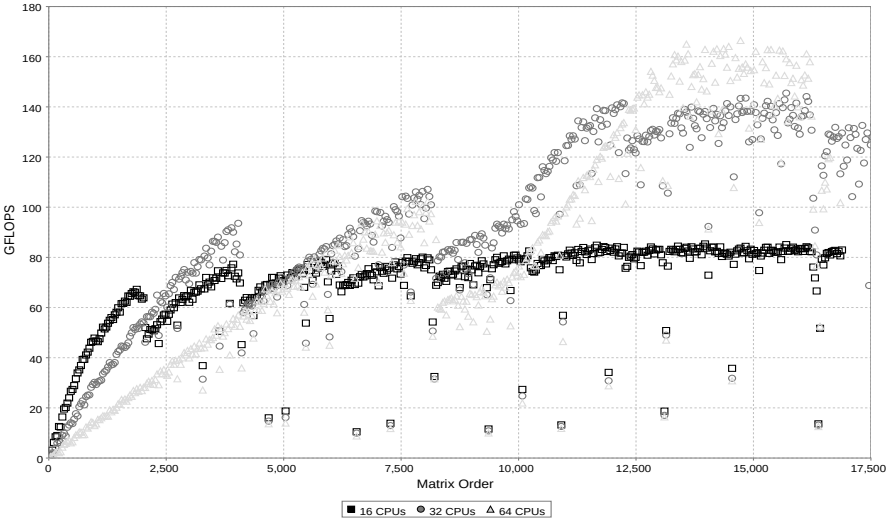


Fig. 3. MKL DGEMM performance on Itanium 2 for 16, 32, and 64 CPUs

processors will need very large matrices to achieve at least a small speedup, and can even be slower for small matrices.

The SCSL shows equally discouraging results as the MKL does in Fig. 3, which is not surprising, as the limiting factor for more than 16 processors is just

the memory bandwidth due to the bad data distribution. The next subsection will describe how this performance degradation can easily be corrected for both libraries.

3.3 Parallelizing the Data Initialization

With the first touch policy (see 2.2) in mind, a better data distribution on the machine should be implemented. This can be achieved by surrounding the initialization loop that fills the matrices with an OpenMP pragma. Listing 1.3 shows the corresponding C code. Please note that the data is distributed on the machine by different threads, each of them running on a different processor. Therefore the matrices are spread row-wise over the memory modules. Special attention should be paid to this parallelization in order not to scatter the cache lines. We did not find a better data distribution by using, for example, different chunksizes or other scheduling strategies such as dynamic scheduling.

Without having extended knowledge of the DGEMM library, it is impossible to know which processor will predominantly use which data to further improve the code in listing 1.3. Thus we can only spread the data over the memory modules in order to use the whole memory bandwidth of the system as efficiently as possible without providing a perfect data distribution. However, when using OpenMP-parallized calculations in a specific program, the data distribution should be adjusted accordingly.

```
#pragma omp parallel for schedule(static,1) \
 private(x,index,max) shared(A,B,C,size)
for(x = 0; x < size; x++) {
        index = x * size;
        max = index + size;
        for(index; index < max; index++) {
                A[index] = 30.0;
                B[index] = 0.01;
                C[index] = 0.0;
        }
}
```

Listing 1.3. Surrounding pragma for initialization loop

As shown in Fig. 4, the improved data distribution speeds up the matrix multiplication for a large number of processors significantly. The 64 processor measurement now peaks at about 330 GFLOPS instead of 170 GFLOPS. As we have noticed for example for `getrf` (LAPACK) measurements, the effect can be much larger for algorithms which are not as cache efficient as DGEMM but depend more on memory bandwidth. As a welcome side effect, the improved data distribution has resolved the issue of the striking variability in the performance of the MKL calls in Fig. 3.

Despite the obvious performance improvement, Fig. 4 also shows that MKL DGEMM calls with 64 and more processors are still not perfect. Instead, they show a very unsteady behavior. A more detailed view on the results (Fig. 5)

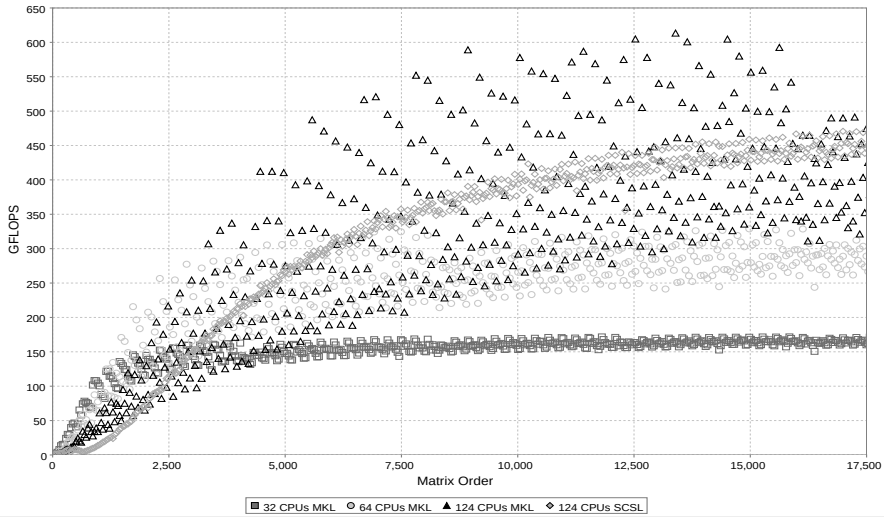


Fig. 4. Optimized MKL and SCSL DGEMM performance on Itanium 2

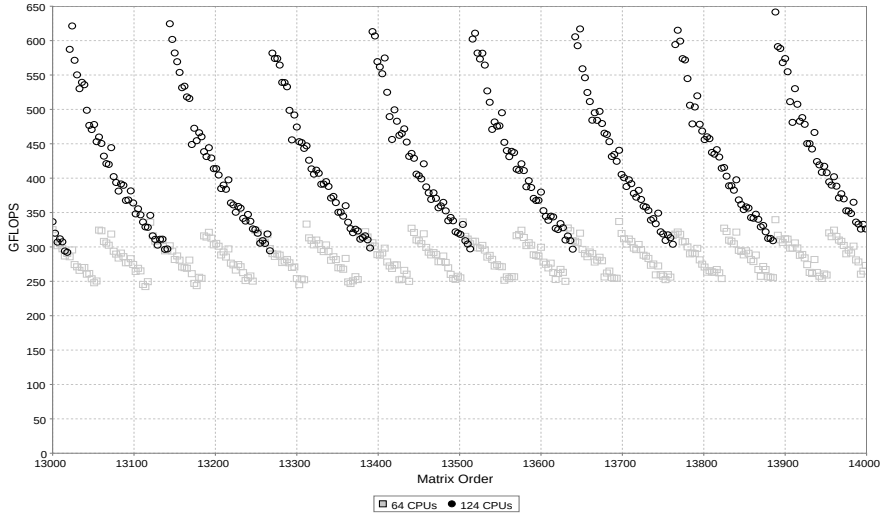


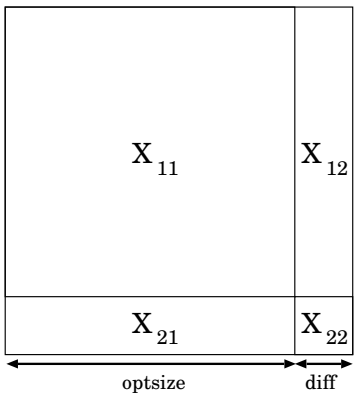
Fig. 5. Zoom on MKL DGEMM performance for 64 and 124 CPUs

shows that MKL scales best when the problem size is a multiple of the number of processors used. For a matrix multiplication running on 124 processors a difference of one as matrix order can decide whether the achieved performance is about 300 or 600 GFLOPS.

The SCSL on the other hand does not show this unsteady behavior as the plot for 124 CPUs represents in Fig. 4. However, the MKL peak performance is significantly higher, which suggests a further improvement of the MKL DGEMM call.

3.4 Partitioning the DGEMM Call

It was shown that a very good but not constant performance for a DGEMM call was reached by the MKL. For further optimization, we split up the DGEMM call to limit the computation of the resulting matrix in the number of columns and rows to a multiple of the number of processors and threads respectively. The remaining parts of the resulting matrix are calculated later on in separate DGEMM calls. Fig. 6 shows how this partitioning is done for a square matrix. There are four separate matrix multiplications and four corresponding DGEMM calls to be calculated. *optsize* equals the original matrix order (*size*) truncated to a multiple of the number of processors (p) and *diff* represents the overlapping part. Or as a formula: $\mathit{diff} = \mathit{size} \bmod p$ and $\mathit{optsize} = \mathit{size} - \mathit{diff}$.



$$\text{1. main}: \; C_{11} = (A_{11}A_{12}) \cdot \begin{pmatrix} B_{11} \\ B_{21} \end{pmatrix}$$

$$\text{2. right}: \; C_{12} = (A_{11}A_{12}) \cdot \begin{pmatrix} B_{12} \\ B_{22} \end{pmatrix}$$

$$\text{3. bottom}: \; C_{21} = (A_{21}A_{22}) \cdot \begin{pmatrix} B_{11} \\ B_{21} \end{pmatrix}$$

$$\text{4. bottom right}: \; C_{22} = (A_{21}A_{22}) \cdot \begin{pmatrix} B_{12} \\ B_{22} \end{pmatrix}$$

Fig. 6. Partitioning the matrix multiplication

By splitting up the DGEMM calls we make sure that the largest part of the calculation (C_{11} in Fig. 6) is done with optimal performance. Considering the cubic complexity of the matrix multiplication, the computing effort for the remaining parts is, especially for larger matrices, very small.

The optimized C code for square matrices is shown in Listing 1.4. The code for non-square matrices is similar but slightly more sophisticated, as there are three optimal sizes that need to be calculated.

With this implementation, a maximal performance of about 600 GFLOPS for 124 processors is reached and stays nearly at the very same level. This means a speedup of about 102 in comparison to a single Itanium 2 processor for DGEMM

calls. The performance of the new implementation is dominating the old one which means that there is nearly no performance loss. A speedup of 2 is reached for problem sizes N, when $N = p * n - 1$, where p is the number of processors and n is a large natural number. The lowest performance improvement is visible for problem sizes $N = p * n$, as they already had a good performance before the optimization. The overhead for these problem sizes is very small as there are only three additional integer calculations to be executed.

```
diff = size % omp_get_max_threads();
optsize = size - diff;
cblas_dgemm(CblasRowMajor,CblasNoTrans,CblasNoTrans,
  optsize, optsize, size, 1.0,
  A, size, B, size, one, C, size);     /* main part */
if (diff > 0)          /* calculate remaining parts */
{
  /* right */
  cblas_dgemm(CblasRowMajor,CblasNoTrans,CblasNoTrans,
    optsize, diff, size, 1.0, A, size, &(B[optsize]),
    size, one, &(C[optsize]), size);
  /* bottom */
  cblas_dgemm(CblasRowMajor,CblasNoTrans,CblasNoTrans,
    diff, optsize, size, 1.0, &(A[size*optsize]),
    size, B, size, one, &(C[size*optsize]), size);
  /* bottom right */
  cblas_dgemm(CblasRowMajor,CblasNoTrans,CblasNoTrans,
    diff, diff, size, 1.0, &(A[size*optsize]),
    size, &(B[optsize]), size, one,
    &(C[size*optsize+optsize]), size);
}
```

Listing 1.4. Optimized DGEMM call for MKL on SGI Altix

The overall speedup for the optimization described in this section is between one and two. Figure 7 compares the DGEMM performance on 124 processors for SCSL and MKL with improved data distribution according to 3.3 and MKL with additional DGEMM partitioning.

However, a direct optimization within the Intel MKL library might deliver even higher performance than the implementation described above.

3.5 Reinitializing the Data

Further examinations have revealed that beyond the DGEMM call a lot of time is used for initializing the matrices, even though this was parallelized. In fact, the parallel initialization of the three matrices with an order of 13000 takes about nine seconds on 124 processors. The corresponding write bandwidth is about 31 MByte/s. According to SGI [10], the glibc `malloc`/`free` calls consume significant system time due to memory management overhead. In order to prevent glibc from using `mmap`, the environment variables `MALLOC_TRIM_THRESHOLD_=-1`

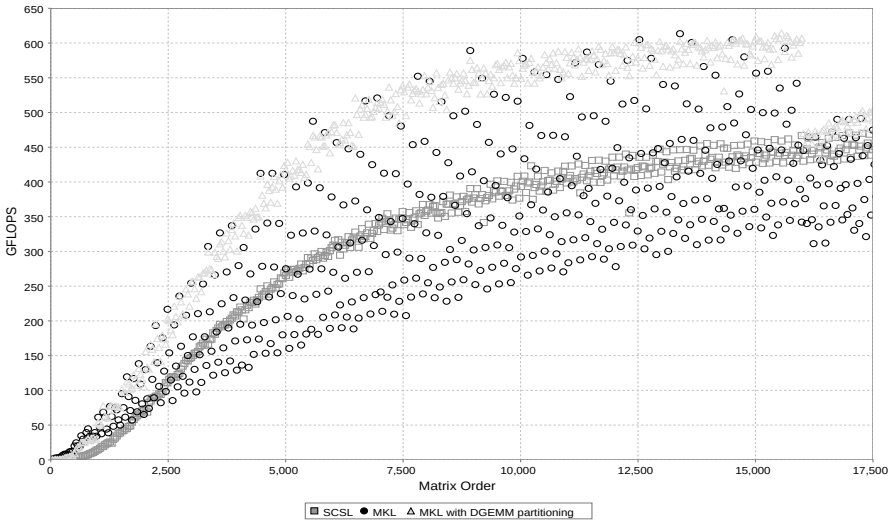


Fig. 7. Optimized MKL and SCSL DGEMM performance for 124 CPUs

and `MALLOC_MMAP_MAX_=0` have to be set. Thereby, repeated matrix reinitializations as in our BenchIT performance measurement runs are about one order of magnitude faster. However, these time values are unsteady and therefore further examination is planned.

4 Conclusion

The Altix 3700 combined with the Intel MKL or SGI SCSL provides a fast computation of DGEMM calls. These calls can easily be parallelized, but special attention should be paid to the first touch policy. For matrices of order 5000 or higher, 64 or more CPUs can be used efficiently but native MKL DGEMM calls show unsteady performance. We have described how to optimize these calls to offer steady performance and clearly outperform SCSL, mostly independent of the matrix order and the number of OpenMP threads used.

References

1. BenchIT: Homepage http://www.benchit.org
2. Juckeland, G., Börner, S.; Kluge, M., Kölling, S., Nagel, W.E., Pflüger, S., Röding, H., Seidl, S., William, T., Wloch, R.: ParCo 2003: BenchIT - Performance Measurement and Comparison for Scientic Applications http://www.benchit.org/DOWNLOAD/DOCUMENTS/parco_paper.pdf
3. Juckeland, G., Kluge, M., Nagel W.E., Pflüger, S.: QEST, IEEE Computer Society, 2004.: Performance Analysis with BenchIT: Portable, Flexible, Easy to Use ISBN 0–7695–2185–1, S. 320–321
4. Schöne, R., Juckeland, G., Nagel W.E., Pflüger, S., Wloch, S.: Parco 2005: Performance comparison and optimization: Case studies using BenchIT http://www.benchit.org/downloads/documents/parco_05_abstract.pdf
5. Silicon Graphics Inc.: Homepage http://www.sgi.com
6. Oak Ridge National Laboratoy: Evaluation of the Altix 3700 at Oak Ridge National Laboratoy http://www.gelato.unsw.edu.au/archives/linux-ia64/0409/10993.html
7. University of Tennessee: Basic Linear Algebra Subprograms Technical (BLAST) Forum http://www.netlib.org/utk/papers/blas-report.ps
8. Intel: Intel Math Kernel Library 8.0 http://www.intel.com/cd/software/products/asmo-na/eng/perflib/mkl/index.htm
9. Silicon Graphics Inc.: Scientific Computing Software Library http://www.sgi.com/products/software/scsl.html
10. Silicon Graphics Inc.: Linux Application Tuning Guide http://techpubs.sgi.com/library/manuals/4000/007-4639-004/pdf/007-4639-004.pdf