PLASMA scaling study

Simon Haendeler, Severin Staudinger

January 31, 2017

Abstract

We are examining the scaling of the PLASMA library in case of the Cholesky Decomposition. We compare the runtime when using a different amount of cores and compare this to the runtime of LAPACK. To verify correctness we computed residuals.

1 Introduction

In this paper we demonstrate how a driver function of PLASMA [2] scales. Therefor we decided to show this by the cholesky factorization, which is an important decomposition of symmetric positive definite matrices in the field of scientific computing.
To achieve this we implemented PLASMA's *DPOTRF* routine and measured the runtime and mflops of its execution on two, four, eight, sixteen, 32 and 48 cores. For time measuring we used the library PAPI [5].
All measurements were done on a system with an x86_64 architecture, 48 CPUs, one thread per core, AMD Opteron™ Processor 6174 with a theoretical peak performance of 2200.00 MHz. Furthermore there are 8 NUMA nodes, which combine six cores per each node. Since it was not easy to found out how many instructions per cycle (IPC) this particular AMD processor can handle, we assumed that it is just as high as in similiar processors of AMD. In [3] it says that the number of operations which the processor can handle in one cycle is four. Additionally we found a similar project, in which the authors compare the the scaling of a LU factorization in LAPACK [6]. This paper confirms our assumption that our CPUs can handle 4 operations per cycle, since this study used exactly the same processors as we did.
To calculate relative speedups we compared the runtimes of the multiple core execution to the sequential computation of PLASMA.
In the end we compared PLASMA's *DPOTRF* runtimes to the sequential cholesky decomposition implemented in LAPACK [1] to calculate the absolute speedup.

Staudinger

2 Implementation

In order to build PLASMA we installed OpenBLAS [7] and LAPACK [4] from the latest stable releases. Since our system has just one thread per core we built OpenBLAS single-threaded with *MAKE USE_THREAD=0.*

To guarantee portability we decided to make a bash-script, which installs every necessary library fully automatically and also auto-generates a Makefile, which compiles our *main.c* programm. Also we coded a second helper bash script run.sh which executes *plasma_lapack_blas.sh* and delivers the necessary $INSTALL_DIRECTORY to the install-bash-script.
If you want to set a specific install directory just modify run.sh and replace ._install with your favoured directory. When you are ready execute the script with *bash run.sh*. After the installation process is done and the Makefile is auto-generated compile the *main.c* programm with the command *make*. To execute the programm simply type into the terminal *"./main n cores"* or *"./main n l"* if you want to measure LAPACK's Cholesky Decomposition.
The parameter *n* stands for the matrix size and *cores* stands for the number of cores which are used for the computation.
To verify a correct execution of the test programm we used valgrind -memcheck, which found no errors.

Since the cholesky decomposition is only possible for symmetric positive definite matrices we generated diagonally dominant ones, because they are always positive definite.

```
void generateMatrix(double *A, int n) {

    double random_number;
    srand(time(NULL));

    for (int j = 0; j < n; j++) {
        for (int i=j; i<n; i++) {
            random_number = ((double)rand() / ( (double)RAND_MAX ));
            A[i*n+j] = random_number;
            A[j*n+i] = random_number;
        }
    }

    // make positive definite
    for (int i=0; i<n; i++) {
        A[i*n+i] += n;
    }

}
```

As you can see we added n to the diagonal entries. This is because all entries of matrix A are never higher than one and through this little trick the sum of the row elements is less than their corresponding diagonal element. This guarantees that matrix A is positive definite.

For time measurement itself we used *PAPI_flops()*.

```
int measure_PLASMA_dpotrf(double* A, double* L, double* work,
  PLASMA_enum uplo, int n, int cores) {

    error = PLASMA_Init(cores);
    if(error != PLASMA_SUCCESS){
        printf("{\"ERROR-TEXT"}\n", error, error);
        exit(1);
    }

    PAPI_flops(&rtime, &ptime, &flpops, &mflops);
    error = PLASMA_dpotrf(uplo, n, L, n);
    PAPI_flops(&rtime, &ptime, &flpops, &mflops);

    error = PLASMA_Finalize();
    if(error != PLASMA_SUCCESS){
        printf("{\"ERROR-TEXT\n", error, error);
        exit(1);
    }

    //these are just shortened code snippets
}
```

To verify correctness of PLASMAs and LAPACKs decompositions we calculated residuals. We used the one-norm. The closer the results are to zero the better they are.

$$\frac{||L * L^T - A||_1}{||A||_1}$$

```
double residual(double* L, double* A, double* work, int n) {

    //make L lower triangular
    for (int j=1; j<n; j++) {
        for (int i=0; i<j; i++) {
            L[j*n+i]=0;
        }
    }
    // L L^t
    PLASMA_dgemm(PlasmaNoTrans, PlasmaTrans, n, n, n, 1, L, n, L, n, 0,
      work, n);

    // LL^t - A
        cblas_daxpy( n*n, -1, A, 1, work, 1 );

    double o_value = PLASMA_dlange(PlasmaOneNorm, n, n, work, n);
    double A_norm = PLASMA_dlange(PlasmaOneNorm, n, n, A, n);

    return o_value/A_norm;

}
```

Since there is no function in PLASMA, LAPACK or BLAS which can compute a matrix multiplication with two triangular matrices as an input we had to set the upper entries of the matrix L to zero.
In the beginning of our implementation we used here BLAS and LAPACK functions, because we don't measure execution time of the residual calculation. When testing we recognized that the residual computation takes a lot of time and so we replaced all functions with the related PLASMA ones.
This resulted in a far quicker execution.

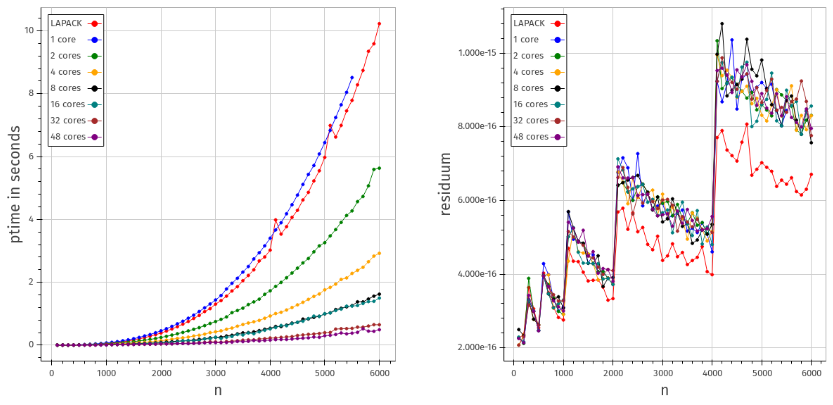
For gathering output data we coded a python-script which encodes the data in JSON. This made further processing e.g. plotting a lot easier.

Staudinger

3 Evaluation

Because the measurement was done on a shared system we could not guarantee that our program was the only one running. Measurement errors through context switches and alike are possible.
Each datapoint represents 5 measurements that were averaged so that small measurement errors cancel out. Systematic error could still be present i.e. long running processes which affect all measurements are still possible. Although we

tried to control for those manually via htop and similar tools, we cannot exclude those with 100% certainty.



(a) Process time (b) Residuum

Figure 1: Runtime and Residuum of PLASMA vs LAPACK

In figure 1a we see the runtime of the LAPACK version and of the PLASMA version with different amount of cores/threads used. As expected the runtime mostly increases like x^3. The single threaded PLASMA version is slightly slower than the sequential LAPACK version. The other PLASMA versions are faster the more cores there are, at least for n large enough. The versions for 8 and 16 threads are very close together. 32 and 48 threads are also close, although we expect that this gap should be relativly narrow. There are bumps in the graph of the LAPACK version for $n = 4100$ and $n = 5100$.

The residuum is shown in figure 1b. All PLASMA versions have the same residuum behavior, but LAPACK is always a bit better. There are certain thresholds after which the residuual gets worse (200, 500, 1000, 2000, 4000). After these sudden increases the residuum gets better with higher n.

In figure 2a we see the relative speedup for each version. For comparison there are black lines at 2, 4, 8, 16, 32, 48 which represent the theoretical possible speedups. We can see that the parallel versions start slower than the sequential version. Then for relatively low n each version archieves a local maxima. For a higher thread number the n is higher when the maxima is reached.

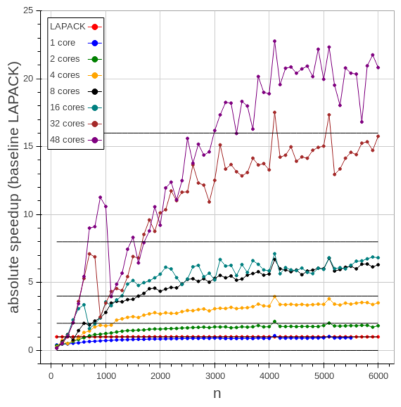
Only 2 and 4 threads nearly reaches its theoretical performance, the higher the amount of threads, the worse the performance relatively to the theoretical performance.

In figure 3 we see the Mflops per second. We see that both LAPACK and PLASMA with one core are very similar.

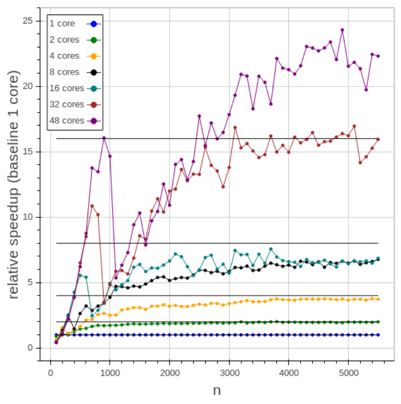
Haendeler

4 Conclusions

When looking at the runtime we can see that PLASMA scales with additional cores. For a lower amount of cores the scaling is very good and nearly multiplies



(a) Absolute speedup (baseline LAPACK)



(b) Relative speedup (baseline 1 core)

Figure 2: Relative and absolute speedups

the performance with the amount of cores. As soon as we go higher than 8 the scaling gets worse. The test server has 4 CPUs with each 12 cores, so when the number of cores is higher than 12 additional overhead should come into play. 32 cores reaches the performance we would have wished for 16 cores. This trend also goes on with 48 cores, the maximum on the system.

LAPACK has consistently a better residuum than PLASMA. This is very surprising, because LAPACK is used internally of PLASMA. Maybe the merging of the subsolutions of the multiple LAPACK invocations affect the accuracy.
In that case we would expect the residuum to grow with the number of cores, which we cannot see here. Maybe we could see this if the residuum was more stable, i.e. when averaging over a higher number of trails. One could also do additional statistic analysis of the result. But because the residuum is good enough in the PLASMA case we will not further discuss it here.

It is also surprising that the residuum gets better with higher n between the jumps. The jumppoints have a very systematic distribution, so it is likely that LAPACK changes some parameters when around these thresholds.

For mflops we roughly get 150 Gflops.
We computed the theoretical peak performance for our system as follows:
2.1 GHz x 12 cores x 4 ops per cycle = 105.6 Gflop/s per socket. [6]
So for all four CPU sockets its ~422 *Gflops/s.* We can clearly see that we reach ~35% of the system's theoretical peak performance. This is similar to other publications, e.g. [6], although in that case GPUs were used. Generally this is a good result for the low additional work we need to integrate PLASMA.

Haendeler

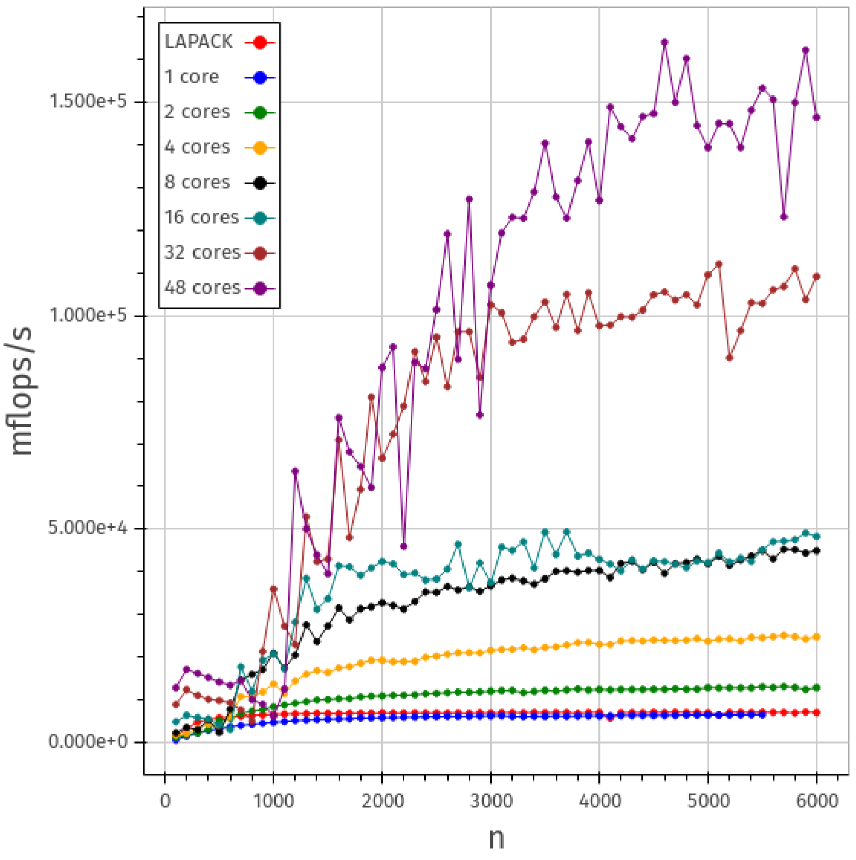


Figure 3: MFlops/s LAPACK vs PLASMA

References

[1] Lapack. `http://www.netlib.org/lapack/`.

[2] Plasma. `http://www.netlib.org/plasma/`.

[3] AMD. Amd opteron™ 6200 series processor quick reference guide, 2012. *http : //tupac.conicet.gov.ar/files/Opteron*_6000_*QRG.pdf*.

[4] Edward Anderson, Zhaojun Bai, Christian Bischof, James W. Demmel, Jack J. Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, Alan McKenney, and Danny C. Sorensen. LAPACK: A portable linear algebra library for high-performance computers. Technical Report 20, LAPACK Working Note, May 1990.

[5] Shirley Browne, Christine Deane, George Ho, and Phil Mucci. Papi: A portable interface to hardware performance counters. 06-1999 1999.

[6] Jakub Kurzak, Piotr Luszczek, Mathieu Faverge, and Jack Dongarra. Lu factorization with partial pivoting for a multi-cpu, multi-gpu shared memory system. Technical Report 266, LAPACK Working Note, April 2012.

[7] Z. Xianyi, W. Qian, and Z. Yunquan. Model-driven level 3 blas performance optimization on loongson 3a processor. In *2012 IEEE 18th International Conference on Parallel and Distributed Systems*, pages 684–691, Dec 2012.