Comput Sci Res Dev (2011) 26: 221–228
DOI 10.1007/s00450-011-0159-z

SPECIAL ISSUE PAPER

Scalable parallel AMG on ccNUMA machines with OpenMP

Malte Förster · Jiri Kraus

Published online: 9 April 2011
© Springer-Verlag 2011

Abstract In many numerical simulation codes the backbone of the application covers the solution of linear systems of equations. Often, being created via a discretization of differential equations, the corresponding matrices are very sparse. One popular way to solve these sparse linear systems are multigrid methods—in particular AMG—because of their numerical scalability. But looking at modern multi-core architectures, also the parallel scalability has to be taken into account. With the memory bandwidth usually being the bottleneck of sparse matrix operations these linear solvers can't always benefit from increasing numbers of cores. To exploit the available aggregated memory bandwidth on larger scale NUMA machines evenly distributed data is often more an issue than load balancing. Additionally, using a threading model like OpenMP, one has to ensure the data locality manually by explicit placement of memory pages. On non uniform data it is always a tradeoff between these three principles, while the ideal strategy is strongly machine- and application dependent. In this paper we want to present some benchmarks of an AMG implementation based on a new performance library. Main focus is on the comparability to state-of-the-art solver packages regarding sequential performance as well as parallel scalability on common NUMA machines. To maximize throughput on standard model problems, several thread and memory configurations have been evaluated. We will show that even on large scale multi-core architectures easy parallel programming models, like OpenMP, can achieve a competitive performance compared to more complex programming models.

Keywords LAMA · AMG · OpenMP · ccNUMA · First Touch · PETSc · hypre

M. Förster (✉) · J. Kraus
Fraunhofer Institute for Algorithms and Scientific Computing SCAI, Schloss Birlinghoven, 53754 Sankt Augustin, Germany
e-mail: malte.foerster@scai.fraunhofer.de

J. Kraus
e-mail: jiri.kraus@scai.fraunhofer.de

1 Introduction

In this paper we show that we obtain competitive parallel performance with an OpenMP based parallelization compared to an MPI based parallelization. The test application is a classical algebraic multigrid (AMG) solver [13, 19] for sparse matrices used as a preconditioner for the conjugate gradient method (CG).

Algorithmically, every iteration of AMG does consist of several sparse matrix vector multiplications (SpMV) on each level of the AMG hierarchy [10]. It is well known that these operations are rather memory bound than CPU bound. Therefore, it is not common to observe linear speedups by just increasing the number of cores. It is more crucial to utilize the available memory bandwidth, especially when expanding threads on multi-socket ccNUMA machines. We show that, via OpenMP worksharing, it is possible to achieve nearly linear speedups in terms of sockets rather than cores, resulting in absolute runtimes comparable to complex MPI implementations.

For the MPI implementation we use the package hypre [1] via the PETSc [5] interface. For the OpenMP implementation we use an AMG based on the new Library for Accelerated Math Applications (LAMA [4]). For the detailed comparison we first verify that—for given sets of parameters—the two implementations are basically equivalent regarding the mathematical operations. Additionally,

we will see that due to algorithmic optimizations and general software design the sequential AMG in LAMA even outperforms the reference solver. After verification we present parallel benchmarks on different ccNUMA machines described in Sect. 3. Here we show that LAMA shows good parallel scalability over the available sockets, resulting in absolute runtimes still comparable to the reference solver programmed with MPI.

2 LAMA

The Library for accelerated math applications, LAMA, is a new open source project which will be available at http://www.libama.org shortly. The first of two main design aims of LAMA is to enable a natural mathematical syntax without sacrificing performance like it is also achieved by the C++ Library Blitz++ [7]. The second main design aim is to allow easy integration of accelerators like CUDA GPGPUs [12]. To achieve both goals LAMA is separated into two parts. A C library which provides BLAS functionality for dense and sparse types and which is used to utilize all types of accelerators and a C++ part which provides the natural mathematical syntax. The C library makes our core algorithms of our library usable by a wide range of applications and allows the integration of existing BLAS Libraries. The C++ part uses simplified expression templates [22] to achieve the second design aim. Utilizing this and by formulating solvers only in terms of simple BLAS operations, like they are printed in text books [11], we achieve very comprehendible solver implementations and it is easy to experiment with new accelerators or data structures, e.g. different sparse matrix formats. To give an example, Fig. 1 shows the main part of our CG implementation. Currently the C++ part of LAMA mainly uses "compile time polymorphism" through templates. This enables aggressive compiler optimizations while sacrificing some runtime flexibility. For the AMG solver examined in this paper we have concentrated on the solver phase. The setup is only partially parallelized and optimized, so the AMG setup times are not comparable to Boomer AMG.

```
template<...>
void CG<Matrix,Vector,Logger>::iterate()
{
    typedef typename VectorType::ValueType ValueType;
    ValueType lastPScalar = m_pScalar;
    ValueType& pScalar = m_pScalar;
    ValueType alpha;
    ValueType beta;
    VectorType& residual = (*m_residual);
    const MatrixType& A = (*m_coefficients);
    VectorType& x = (*m_solution);
    VectorType& p = (*m_p);
    VectorType& q = (*m_q);
    VectorType& z = (*m_z);

    //CG implementation start
    if(m_preconditioner == 0)
        z = residual;
    else
    {
        z=0;
        m_preconditioner->solve(&residual,&z);
    }

    pScalar = residual * z;

    if (this->getIterationCount() == 0)
        p = z;
    else
    {
        if (lastPScalar == 0.0)
            beta = 0.0;
        else
            beta = pScalar / lastPScalar;
        p = z + beta * p;
    }

    q = A * p;

    const ValueType pqProd = p * q;
    if (pqProd == 0.0)
        alpha = 0.0;
    else
        alpha = pScalar / pqProd;

    x = x + alpha * p;
    residual = residual - alpha * q;

    //CG implementation end

    m_solution.setDirty(false);
}
```

Fig. 1 Main part of the LAMA CG implementation

3 Hardware setup

All results presented in this paper were computed on three different machines. Table 1 shows the key specifications for the used hardware.

All machines are running with Scientific Linux SL 5.3. Please note that in case of SCAIPROD84 we work with activated hyperthreading, effectively doubling the number of *logical* cores visible to the system. Still, one has to keep in mind that the number of physical cores remains at 32 when looking at benchmark results for the 'oversubscribed' configurations at 64 threads/processes for this machine.

3.1 ccNUMA

The machines BULL, SCAIPROD83 and SCAIPROD84 are cache coherent NUMA (ccNUMA) machines. Nearly all larger servers today are ccNUMA machines that are build from 2 to 8 CPUs where each CPU has access to a local memory. Figure 2 shows a basic NUMA architecture with 4 sockets or NUMA nodes. As NUMA stands for None Uniform Memory Access a CPU can access the memory which

Table 1 Different architectures used for benchmarks

Name	BULL	SCAIPROD83	SCAIPROD84
Cpu	Xeon X5650	Opteron 8435	Xeon X7560
Core freq.	2.67 GHz	2.6 GHz	2.27 GHz
L3-cache	12 MB	6 MB	24 MB
Cores/cpu	6	6	8
HT	Off	n.a.	on
Sockets	2	8	4
Cores (w.HT)	12(12)	48(48)	32(64)
Memory	12 GB	128 GB	128 GB
Channels	3	2	4
Mem. type	DDR3	DDR2	DDR3
Mem. freq.	1333 MHz	333 MHz	1066 MHz

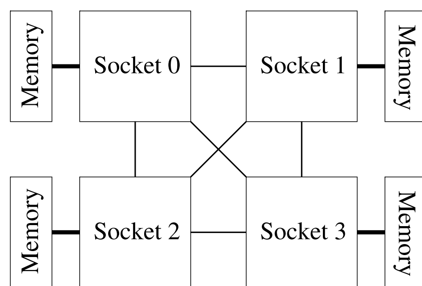


Fig. 2 Basic NUMA architecture

it is directly attached to faster than remote memory which is attached to a different CPU. The remote memory is accessed over a very fast inter socket protocol (QPI for Intel and Hyptertransport for AMD). On the one hand, this makes memory placement a crucial thing, especially for a memory bound algorithms like AMG. But on the other hand, the available memory bandwidth increases with each added socket. To fully utilize the available memory bandwidth it is necessary to distribute the data evenly across the different sockets and to keep the portion of data that a single thread works on local to the CPU it is running on. Because of the rather unspecified access to an input vector in a matrix vector multiplication for sparse matrix formats like CSR it is not possible to fully exploit both, data locality and utilization of available memory bandwidth. The data access pattern of SpMV are comparable to the access patterns of AMG [10].

4 Software setup

In this section we describe our software setup with enough detail that our results should be easily reproducible by others. This includes compiler and configure options used to build PETSc, hypre and LAMA, as well as runtime options. For all compilations we used the GNU Compiler Version 4.5. with the optimization flags `-O3 -ffast-math` and MPICH2-1.2 for the MPI implementation. The used version of PETSc is 3.1p4 with hypre 2.6.0b. We have configured PETSC (including automatic hypre installation) with the following options:

```
--with-c++-support
--COPTFLAGS=-O3 -ffast-math
--CXXOPTFLAGS=-O3 -ffast-math
--with-debugging=0
--download-parmetis=0
--with-log=0
--with-errorchecking=0
--download-superlu_dist=1
--download-mumps=1
--download-f-blas-lapack=1
--download-hypre=1
--download-blacs=1
--download-scalapack=1
```

As parmetis did not compile with `-O3` we have build parmetis with `-O2`.

5 Execution

Because AMG is memory bound [10] the main focus should be to inspect scaling with respect to available memory bandwidth instead of the number of CPU cores. Besides this, we are normally interested in what can we get out of a certain machine and not what we can get if we utilize only a part of it. Therefore, all benchmarks have been executed bound to a subset of the available CPU sockets, utilizing all cores on these sockets to simulate systems with different numbers of sockets. For LAMA we have done this with `numactl` and for PETSc we have bound the mpd daemon of MPICH to a socket with `taskset` to enforce the PETSc processes to only utilize these specific sockets [15, 16]. When Using taskset, it has to be taken into account that the numbering of CPU cores is not always as expected [20]. To start the CG solver with AMG preconditioning described in Sect. 5.2 we have used the following PETSc commandline options [2, 3, 6] (Logging has been disabled for all benchmarks):

```
-ksp_max_it 10 -ksp_type cg -pc_type hypre
-ksp_rtol 1e-20 -ksp_norm_type NATURAL
-pc_hypre_boomeramg_max_levels 7
-pc_hypre_boomeramg_grid_sweeps_all 2
-pc_hypre_boomeramg_relax_type_all jacobi
-pc_hypre_boomeramg_relax_weight_all 0.5
-pc_hypre_boomeramg_relax_type_coarse
Gaussian-elimination
-pc_hypre_boomeramg_coarsen_type HMIS
-pc_hypre_boomeramg_interp_type standard
-pc_hypre_boomeramg_truncfactor 0.1
```

Table 2 Laplacian discretizations used for solver benchmarks

Name	Dimensions	Diags	Entries	CSR mem
1D3P	1,000,000	3	3 Mio.	38 MB
2D5P	1,000 × 1,000	5	5 Mio.	61 MB
3D7P	100 × 100 × 100	7	7 Mio.	83 MB
2D9P	1,000 × 1,000	9	9 Mio.	107 MB
3D27P	100 × 100 × 100	27	27 Mio.	307 MB

The only parameter that changes between the different benchmarks is the max level parameter to synchronize the setups of LAMA and PETSc. The solver phase has been called 5 times for PETSc and LAMA all reported runtimes are the minimum of these 5 executions.

5.1 Model problems

Our set of test matrices are shown in Table 2. They consist of different discretizations of the Laplacian operator on structured grids in up to three dimensions. All matrices have a total of 1 million rows but increase in the number of nonzero entries. Each row corresponds to exactly one grid point and its nonzero values refer to the entries of the differential stencil applied. We have chosen this model problems because they are well known, which makes it more easy to compare our results [12]. Additionally, they are a good measure for real world 1D, 2D and 3D applications because of the basic local access patterns common for matrices based on a wide range of PDE applications.

To exploit the sparsity all matrices are stored in Compressed Sparse Row (CSR) format using double precision.

5.2 Algorithmic comparability of Boomer AMG and LAMA

For the benchmarks we measure 10 iterations of a CG solver preconditioned with AMG. As a coarsening strategy for the AMG we use the classical Ruge-Stüben algorithm [19] (1stage) in combination with standard interpolation. On the coarsest grid Gaussian elimination is used. In the solution phase AMG is running a V-cycle performing two pre- and post-smoothing steps with a weighted Jacobi. Note that for the benchmarks only the time for the solution phase, not the setup time needed to construct the AMG hierarchy, is measured.

One important difference between our LAMA configuration and the hypre counterpart is in the type of coarsening. HMIS [14] is actually a combination of (local) Ruge-Stüben(1stage) and PMIS, introducing additional coarsegrid points at the processor boundaries for better convergence in parallel. Since LAMA does not construct the coarser grids in a distributed parallel process this is not needed here.

Table 3 Galerkin operator stats for 1D3P on 13 levels

	BoomerAMG		LAMA	
Lvl	Rows	Entries	Rows	Entries
0	1,000,000	2,999,998	1,000,000	2,999,998
1	500,000	1,499,998	500,000	1,499,998
2	250,000	749,998	250,000	749,998
3	125,000	374,998	125,000	374,998
4	62,500	187,498	62,500	187,498
5	31,249	93,745	31,250	93,748
6	15,624	46,870	15,625	46,873
7	7,812	23,434	7,812	23,434
8	3,906	11,716	3,906	11,716
9	1,953	5,857	1,953	5,857
10	976	2,926	976	2,926
11	488	1,462	488	1,462
12	244	730	244	730

Table 4 Galerkin operator stats for 2D9P on 7 levels

	BoomerAMG		LAMA	
Lvl	Rows	Entries	Rows	Entries
0	1,000,000	8,988,004	1,000,000	8,988,004
1	250,000	622,0036	250,000	622,0036
2	62,500	277,6608	62,500	277,6594
3	15,624	68,4720	15,625	684,745
4	3,124	120,906	3,126	120,954
5	614	20,110	601	21,161
6	95	2,171	121	3,405

In order to compare benchmarks for the different implementations of AMG in BoomerAMG and LAMA we have to ensure that the approaches basically perform the same work. This work consists of SpMV operations defined by the solution process based on the matrices of all AMG levels. Due to implementation details in the coarsening and interpolation strategy (e.g. truncation techniques or special thresholds) it is not possible to guarantee identical AMG behavior. Therefore, we show that the hierarchies constructed within the setup processes have (nearly) identical matrix properties.

Tables 3 and 4 show the dimensions and the density of the constructed Galerkin matrices. In the one dimensional test case—due to the straightforward coarsening and interpolation process—the hierarchies are almost identical. In the two dimensional test case the operators start to differ slightly starting from the second Galerkin construction. However, these differences are fairly small and most probably negligible while analyzing SpMV sequences over all levels.

Although it has no effect on the performance analysis, we also like to verify the correctness of the entries inside Galerkin and interpolation matrices. Therefore we exem-

Table 5 L2-residual reduction for 2D9P

Iter	BoomerAMG	LAMA
0	$1.89 \times 10^{+2}$	$1.89 \times 10^{+2}$
1	$2.49 \times 10^{+1}$	$2.62 \times 10^{+1}$
2	$1.68 \times 10^{+0}$	$2.39 \times 10^{+0}$
3	1.31×10^{-1}	1.82×10^{-1}
4	3.75×10^{-3}	9.22×10^{-3}
5	2.94×10^{-4}	5.59×10^{-4}
6	2.81×10^{-5}	7.99×10^{-5}
7	1.73×10^{-6}	4.32×10^{-6}
8	1.08×10^{-7}	3.24×10^{-7}
9	9.61×10^{-9}	3.54×10^{-8}
10	6.64×10^{-10}	1.67×10^{-9}

Table 6 Sequential runtimes (s) on BULL

	1D3P	2D5P	2D9P	3D7P	3D27P
BoomerAMG	2.79	3.56	3.89	5.12	7.47
LAMA	1.17	1.87	2.13	2.86	4.70

plary show convergence histories of both approaches applied to 2D9P.

As shown in Table 5 both approaches show convergence rates with similar orders of magnitude. Slightly better convergence of BoomerAMG can be explained by the differences in the coarsening strategy as mentioned above.

6 Single threaded performance

As a first step towards a scaling analysis we look at the sequential performance of both AMG implementations. All timings are minimal runtimes out of 5 repetitions. Table 6 is an overview of the single thread/process run on BULL.

Obviously, there is a huge gap between the runtimes of both packages. Looking closer at the implementations we identify two main optimizations applied to LAMA that could be responsible for this:

– We introduce a solution proxy which is used to only recalculate the residual if it is needed. This way, we are able to reuse the residual calculated for the convergence check for the next CG iteration because the solution is not changed between these two steps.
– We do not copy the solution before each Jacobi iteration. Instead, we simply switch the pointers of old and new solution.

Additionally, due to the design of LAMA using C++ templates, most decisions are made at compile time. Especially, this has an impact on the logging since no additional information is being computed in case of deactivated output.

Table 7 Execution times on BULL in seconds

N	1D3P		2D5P		2D9P		3D7P		3D27P	
N	P	L	P	L	P	L	P	L	P	L
S	2.50	1.17	3.56	1.87	3.89	2.13	5.12	2.86	7.47	4.70
1	1.31	0.58	1.76	0.86	1.79	0.98	2.42	1.25	3.33	1.99
2	0.59	0.32	0.85	0.51	0.88	0.53	1.34	0.67	1.90	1.05

Table 8 Execution times on SCAIPROD83 in seconds

N	1D3P		2D5P		2D9P		3D7P		3D27P	
N	P	L	P	L	P	L	P	L	P	L
S	5.64	2.45	7.92	3.90	8.15	4.56	10.27	5.78	15.91	9.49
1	3.69	1.60	4.66	2.21	4.62	2.47	6.31	3.13	7.89	4.69
2	2.79	0.90	3.08	1.34	3.22	1.34	3.93	1.60	6.82	2.40
3	1.64	0.63	2.37	1.18	2.24	0.96	3.12	1.12	5.40	1.57
4	0.85	0.64	1.19	0.75	1.46	0.92	2.33	0.85	3.41	1.21
5	0.60	0.35	0.90	0.54	1.08	0.53	2.10	0.72	2.76	0.96
6	0.46	0.28	0.66	0.38	0.85	0.47	1.61	0.58	2.40	0.82
7	0.25	0.22	0.46	0.43	0.52	0.37	1.08	0.51	1.37	0.70
8	*0.19*	*0.31*	0.37	0.28	0.43	0.32	1.00	0.47	1.27	0.65

Deactivating the solution proxy and the Jacobi optimization slows down LAMA execution quite a lot to 3.44 s on 2D9P and 1.86 s on 1D3P, moving the execution timings further into the direction of BoomerAMG. Only relevant on the finest grid, the cost of the computation of additional residuals prevented by the solution proxy scales exactly with the matrix size. The cost for copying of the solution vector remains constant throughout all test matrices, being negligible on larger 3D examples with higher bandwidth.

7 Parallel performance

We ran benchmarks on the three machines described in Sect. 3 for all the five model problems from Sect. 5.1. To analyze the scaling behavior of PETSc and LAMA we have always occupied full sockets like it is described in Sect. 5. The results of these measurements are given in Tables 7, 8 and 9. Remember that on SCAIPROD84 hyperthreading was enabled, the benchmarks runs where hyperthreading was utilized are marked with a *, e.g. 4* stands for 4 fully utilized sockets with hyperthreading (64 threads on SCAIPROD84). The first column of these tables is the number of sockets or NUMA nodes (NN) that have been used to execute the benchmark, S stands for serial run and the configuration where PETSc was faster than LAMA are highlighted in red. The reasons why the runtimes of PETSc are in general slower than LAMA are explained in Sect. 6. However, these reasons are also source for the better parallel scaling behav-

Table 9 Execution times on SCAIPROD84 in seconds

N	1D3P		2D5P		2D9P		3D7P		3D27P	
N	P	L	P	L	P	L	P	L	P	L
S	4.92	2.52	7.09	4.58	8.16	5.60	11.35	6.99	16.49	11.87
1	1.75	0.95	2.75	1.49	2.87	1.75	3.97	2.05	5.48	3.39
2	0.89	0.53	1.31	0.93	1.31	0.98	3.11	1.07	4.52	1.82
3	0.74	0.43	0.74	0.69	0.89	0.76	1.63	0.87	3.01	1.43
4	*0.31*	*0.37*	0.47	0.39	*0.57*	*0.60*	1.24	0.61	2.02	0.96
4*	*0.18*	*0.25*	*0.28*	*0.33*	0.42	0.40	1.42	0.56	1.74	0.82

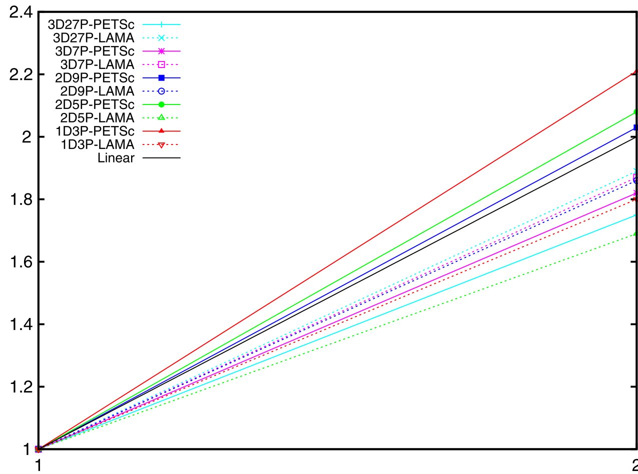


Fig. 3 (Color online) Speedup vs. one socket on BULL

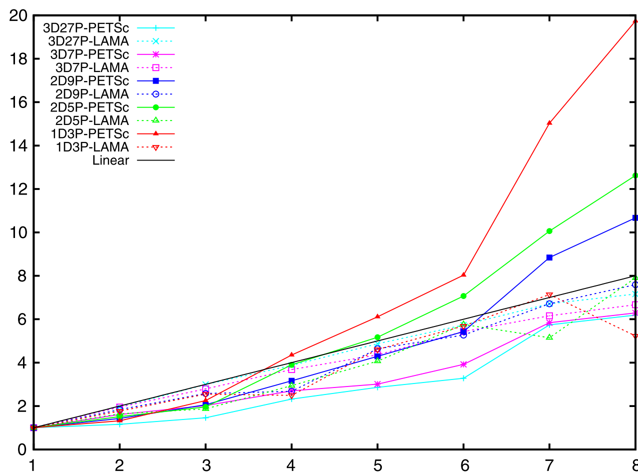


Fig. 4 (Color online) Speedup vs. one socket on SCAIPROD83

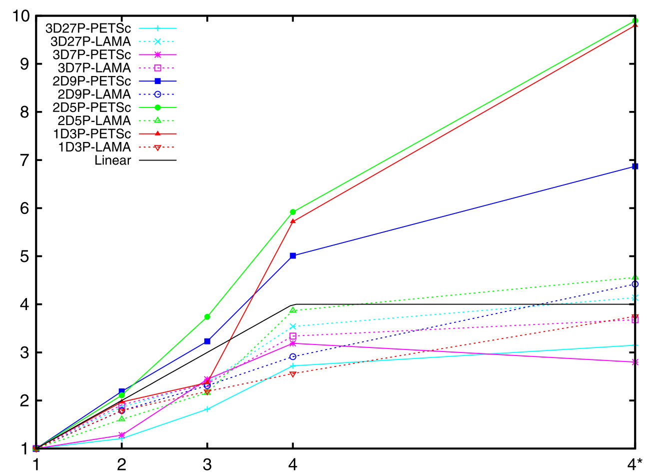


Fig. 5 (Color online) Speedup vs. one socket on SCAIPROD84

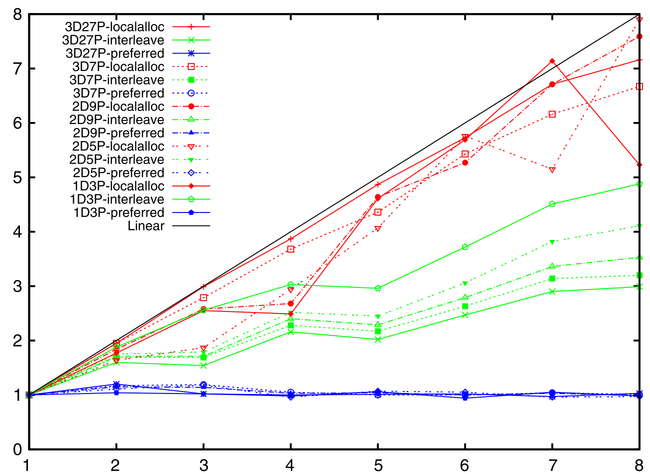


Fig. 6 (Color online) Speedup vs. on socket on SCAIPROD83 for different memory allocation policies

ior of PETSc because especially the residual calculation on the fine grid has good parallel scaling behavior.

As one can see in Figs. 3, 4 and with some limitations also in Fig. 5, LAMA and PETSc scale quite nicely with the available memory bandwidth, i.e. the number of processor sockets. The super linear speedup of PETSc in case of the low dimensional model problems 1D3P, 2D5P and 2D9P are most likely due to cache effects because of the relatively small input sets. Additionally, those examples involve less communication than 3D models. The surprisingly large benefit for PETSc of Hyperthreading on SCAIPROD84 for these model problems have been already reported by Leng et al. [17]. They observed that a MPI applications where the interconnect is the bottleneck benefit from Hyperthreading. Although the interconnect on SCAIPROD84 (QPI) can be considered really fast, it becomes the bottleneck because of the small chunks each process has to process. We assume that the small chunks are also the reason for the suboptimal scalability of LAMA for these small examples, because they lead to false sharing across socket boundaries.

To obtain the scalability for LAMA it was crucial to utilize the first touch policy to achieve good data locality [18, 21] as can be seen in Fig. 6. In this figure the speedups vs. one socket for the memory allocation policy localalloc is shown in red, the policy interleaved is shown in green

and the policy preferred is shown in blue. Localalloc or first touch means to place a memory page in the memory which is local to the socket from which it is first accessed. Interleaved distributes the memory pages in a round robin fashion across the memory of the sockets in use. Preferred places all memory pages in the local memory of socket 0. Figure 6 shows quite nicely the expected behavior. If we use only the memory of a single socket we do not see any scaling because the available memory bandwidth remains constant as shown in blue. If all memory is placed locally we see a near optimal behavior as shown in red. The interleaved policy is somewhere in between because the available memory bandwidth increases but the memory is not always local to the thread that needs access to it.

7.1 Parallel scalability considerations

Looking at the results of Sect. 7 we see a quite good overall scaling for both, the MPI and OpenMP programming model. However, while cutting out cache effects, running plain SpMV operations LAMA seems to scale worse on smaller test cases compared to large ones regarding both, lower dimensions and bandwidth. Of course, this statement also holds for low level matrices inside an AMG hierarchy. There are at least two effects that might explain this to a certain degree. First of all, the matrix data as well as the solution and right hand side vectors are accessed in worksharing constructs. By default, this is done in a *static* manner, causing all loop indizes to be divided into as many equally sized chunks as there are threads to process them. In the same way, we try to keep those chunks of data locally in the memory closest to the thread that need access. Additionally, the size of memory chunks limited by the pagesize, which is typically $4k$. For our data types, e.g. integer, this means that arrays can not be subdivided into smaller chunks than 1024. Looking at smaller matrices this means that data locality just can not be established by first touch anymore. Effectively, this often even leads to slowdowns with increasing numbers of threads. The second effect that is often misleading in interpreting speedups for differently sized matrices is cache effects. The more data a CPU can fit into its level 3 cache the faster data access becomes. This often leads to much better relative performance (e.g. MFLOPS) in sequential mode. That, in combination with problems in data locality, has a negative effect on the speedup on multiple CPU sockets. While we don't have much control over the cache effects, there are some few remedies in OpenMP that might help to overcome the problem of data locality with small chunks of data. One way could be to handle data storage manually by subdividing arrays in a MPI fashion and synchronize explicitly. However, this is certainly not consistent with general OpenMP philosophy.

8 Conclusion

We have shown that the easy to use parallel programming model OpenMP can achieve competitive performance compared to the more complex programming model MPI even on larger ccNUMA machines. For this, the most important point is to achieve good data locality which can be done by utilizing the first touch policy of modern NUMA aware operating systems. There are however, some points that we could not explain in a satisfiable manner. We could only speculate about the facts that the parallel scaling behavior of the lower dimensional model problems is suboptimal and what are the reasons for some outliers. One guess for the outliers is a bad thread placement by the operating system. We have tried to track that down with an explicit pinning of the OpenMP threads, but always degraded performance with more explicit thread pinning, especially with hyper threading enabled. Baker et al. needed to explicitly manage the placement of threads and data by using NUMAlib [10]. Because our code is structured in a way that we basically can utilize the first touch policy for this we have not done this so far. Nevertheless more explicit data and thread placement might be an option for LAMA as well.

9 Future work

The solution phase of our AMG is already competitive with state of the art implementations, but if we take a look at the performance of the whole algorithm the setup phase still has most optimization potential. So this will be one of the next things to do. To overcome the page size and load balancing issues mentioned in Sect. 7.1, we want to evaluate more explicit threading models. As we also plan to support distributed memory machines one option is to use a PGAS language like UPC [9] or a communication library like GPI [8] to resolve these issues. Besides that, we are planning to evaluate the impact of different data structures and mixed precision calculation on AMG and other linear solvers. These are especially interesting on accelerator like GPGPUs which is an additional topic we want to explore. Last mentioned, but definitely our very next step is the open source release of LAMA on http://www.libama.org.

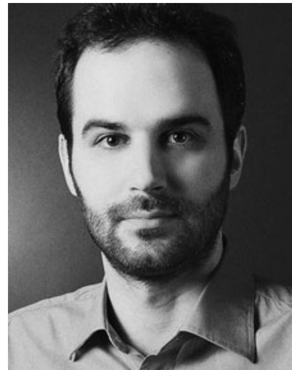
Acknowledgements We want to thank Matthias Makulla who has designed the C++ solver framework within LAMA which we used to implement the analyzed AMG solver.

References

1. hypre homepage. https://computation.llnl.gov/casc/hypre/software.html, last viewed Dec 2010
2. hypre reference manual. https://computation.llnl.gov/casc/hypre/download/hypre-2.6.0b_ref_manual.pdf, last viewed Jan 2011

3. hypre user's manual. https://computation.llnl.gov/casc/hypre/download/hypre-2.6.0b_usr_manual.pdf, last viewed Jan 2011
4. Lama homepage. http://www.libama.org, last viewed Dec 2010
5. Petsc homepage. http://www.mcs.anl.gov/petsc/petsc-as/, last viewed Dec 2010
6. Petsc users manual. http://www.mcs.anl.gov/petsc/petsc-as/snapshots/petsc-current/docs/manual.pdf, last viewed Jan 2011
7. Blitz++ homepage. http://www.oonumerics.org/blitz/, last viewed Jan 2011
8. Gpi homepage. http://www.itwm.fraunhofer.de/abteilungen/competence-center-high-performance-computing/hpc-tools.html, last viewed Jan 2011
9. Unified parallel c homepage. http://upc.gwu.edu/, last viewed Jan 2011
10. Baker A, Schulz M, Yang U (2009) On the performance of an algebraic multigrid solver on multicore clusters. Tech rep, Lawrence Livermore National Laboratory (LLNL), Livermore, CA
11. Barrett R (1994) Templates for the solution of linear systems: building blocks for iterative methods. Society for Industrial Mathematics
12. Bell N, Garland M (2009) Efficient sparse matrix-vector multiplication on CUDA. In: Proc ACM/IEEE conf supercomputing (SC), Portland, OR, USA
13. Brandt A, McCormick S, Ruge J (1984) Algebraic multigrid (AMG) for sparse matrix equations. In: Evans DJ (ed) Sparsity and its applications. Cambridge University Press, Cambridge
14. De Sterck H, Yang U (2006) Reducing complexity in parallel algebraic multigrid preconditioners. SIAM J Matrix Anal Appl 27:1019–1039
15. Kayi A, Kornkven E, El-Ghazawi T, Newby G (2008) Application performance tuning for clusters with ccnuma nodes. In: 2008 11th IEEE international conference on computational science and engineering, pp 245–252. IEEE
16. Kleen A (2005) A numa api for Linux. Novel Inc
17. Lenga T, Ali R, Celebioglu O, Hsieh J, Mashayekhi V, Rooholamini R (2003) The impact of hyper threading on communication performance in HPC clusters. In: Proceedings of the 17th annual international symposium on high performance computing systems and applications and the OSCAR symposium, May 11–14, 2003, Sherbrooke, Quebec, Canada. NRC Research Press, Ottawa, p 173
18. Nikolopoulos D, Artiaga E, Ayguadé E, Labarta J (2001) Exploiting memory affinity in OpenMP through schedule reuse. Comput. Archit. News 29(5):49–55
19. Ruge J, Stüben K (1987) Algebraic multigrid (AMG). In: McCormick SF (ed) Multigrid methods. Frontiers in applied mathematics, vol 3. SIAM, Philadelphia, pp 73–130
20. Terboven C Daily cc-numa craziness. http://terboven.wordpress.com/2009/12/02/daily-cc-numa-craziness/, last viewed Jan 2011
21. Terboven C, et al (2008) Data and thread affinity in openmp programs. In: Proceedings of the 2008 workshop on memory access on future processors: a solved problem? ACM, New York, pp 377–384
22. Vandevoorde D, Josuttis N (2003) C++ templates: the complete guide. Addison-Wesley, Reading

Malte Förster is a senior scientist at the Numerical Software Department of Fraunhofer's Scientific Computing Institute (FhG-SCAI), where he is a member of the numerical solver group lead by Klaus Stüben. In his current work he focusses on the development of efficient parallel software, both MPI- and OpenMP-based. On the application side, his focus is primarily on algebraic multigrid approaches for the solution of Navier-Stokes equations. Malte Förster has obtained his master degree in Mathematics at the University of Cologne.



Jiri Kraus is a senior scientist at the Simulation Engineering Department of Fraunhofer's Scientific Computing Institute (FhG-SCAI), where he is a member of the high performance computing group lead by Thomas Soddemann. His work focuses on parallel algorithms and their efficient implementation on modern parallel hardware. Besides MPI and OpenMP he has a special focus on many core parallelism for hpc accelerators like GPUs. Jiri Kraus has obtained his master degree in Mathematics at the University of Cologne.