Performance modeling of the HPCG benchmark

Vladimir Marjanović, José Gracia, and Colin W. Glass

High Performance Computing Center Stuttgart (HLRS),
University of Stuttgart, Germany

Abstract. The TOP 500 list is the most widely regarded ranking of modern supercomputers, based on Gflop/s measured for High Performance LINPACK (HPL). Ranking the most powerful supercomputers is important: Hardware producers hone their products towards maximum benchmark performance, while nations fund huge installations, aiming at a place on the pedestal. However, the relevance of HPL for real-world applications is declining rapidly, as the available compute cycles are heavily overrated. While relevant comparisons foster healthy competition, skewed comparisons foster developments aimed at distorted goals. Thus, in recent years, discussions on introducing a new benchmark, better aligned with real-world applications and therefore the needs of real users, have increased, culminating in a highly regarded candidate: High Performance Conjugate Gradients (HPCG).
In this paper we present an in-depth analysis of this new benchmark. Furthermore, we present a model, capable of predicting the performance of HPCG on a given architecture, based solely on two inputs: the effective bandwidth between the main memory and the CPU and the highest occuring network latency between two compute units.
Finally, we argue that within the scope of modern supercomputers with a decent network, only the first input is required for a highly accurate prediction, effectively reducing the information content of HPCG results to that of a stream benchmark executed on one single node.
We conclude with a series of suggestions to move HPCG closer to its intended goal: a new benchmark for modern supercomputers, capable of capturing a well-balanced mixture of relevant hardware properties.

1 Introduction

High Performance Computing (HPC) has emerged as a powerful tool in research and industry. Thus, comparing the power of supercomputers has become a central topic. The HPC community selected the High Performance LINPACK benchmark (HPL) as the central metric, reporting the corresponding Gflop/s for the TOP500 ranking. HPL is an example of a highly scalable MPI program, heavily optimized to squeeze the utmost performance out of parallel machines. Combined with a compute heavy kernel, HPL almost reaches the theoretical Gflop/s peak performance of machines. However, the majority of real-world applications feature kernels which are memory-bound on modern hardware and usually reach less than 20% of the theoretical peak performance [19]. Thus, the

current ranking relies purely on a heavily optimized, compute-bound application, arguably very untypical for real-world HPC applications.

The discrepancy in performance between HPL and real-world applications has grown larger over the years, mainly due to a developing trend in computer architectures: the available computing power increases a lot faster than memory speed. The latter therefore limits the data throughput of an increasing number of computational kernels. In HPL this is irrelevant, as the amount of data required per computation (Byte/Flop) is very small. Discussions on HPL's lack of representativity have become more frequent in recent years and a new benchmark was proposed to address the above mentioned issues: High Performance Conjugate Gradients (HPCG). The most expensive kernel of HPCG is dominated by a matrix-vector multiplication. With a Byte/Flop ratio bigger than 4 [10], this kernel shows memory-bound behavior for all current hardware.

In this paper, we present a model capable of predicting the performance of HPCG for a given architecture within 3% of the real value, based on two simple hardware metrics: effective memory bandwidth from main memory and IC latency. The good predictive power of the model indicates, that only these two hardware metrics are relevant for HPCG performance. Furthermore, the model allows us to extrapolate HPCG's performance to future systems.

The paper is organized as follows: In section 2 on overview of related work is presented. Section 3 describes hardware platforms and software enviroments used in this work. Section 4 explains the transition from the HPL to the HPCG benchmark and analyzes the implementation of HPCG. In section 5 we introduce the performance model. Section 6 evaluates the performance model and predicts the performance of HPCG on future systems. Finally, in section 7 we draw conclusions and discuss future work.

2 Related Work

In this section we give a short overview on related work, in particular alternative benchmarks to quantify the performance of HPC systems and performance models for these benchmarks where available.

While HPCG and HPL both consist of a single application which is dominated by a single kernel, the NAS parallel benchmark [5][4] is a collection of small applications or kernels from the fields of computational fluid dynamics, linear algebra, etc. A MPI + OpenMP version has been presented and analyzed in [9]. Similarly, the SPEC MPI2007 benchmark suite [17] is a collection of MPI-parallel compute-intensive applications which has been analyzed in [21]. While the type of applications used in both of these benchmark suites make up a significant part of the workload in most supercomputing centers, the resulting number is in both cases a somewhat arbitrarily weighted total execution time. Further, these benchmarks do not explicitly target scalability of very large machines.

The HPC Challenge benchmark [16] in contrast consists of very low-level kernels (including HPL, Stream, FFT, Random Access, etc), but does not try to produce a single aggregated metric out of these. Instead the individual bench-

marks can be used to measure specific characteristics of a systems. For instance, this paper uses a Stream benchmark to quantify the effective memory bandwidth. Similarly, Random Access can be used to measure the latency of remote memory accesses, and FFT to quantify the quality of all-to-all communication capabilities.

The most widely used benchmark to characterize the performance of supercomputers is still HPL [18]. Discussions on shortcomings of HPL can be found in [11][13].

The proposed new benchmark HPCG investigated here is based on a iterative sparse-matrix Conjugate Gradient kernel. The initial version of HPCG, i.e. v1.0, used a simple additive Schwarz pre-conditioner and symmetric Gauss-Seidel sweep for each sub-domain [11]. The pre-conditioner was replaced in version 2.0 with a multi-grid approach using 3 levels of coarsening. Effectively, this is a kind of tiling of the algorithm and allows to use the caches more effectively. Performance analysis of pre-conditioners is a vast field of research. Here we mention only [3], discussing a multi-grid approach similar to the one used in HPCG, and [6] for a survey of various other techniques. Most pre-conditioners use sparse-matrix vector multiplication which was modelled and optimized for instance in [7] [8]. An other important part of the benchmark is network communication, were related work includes [20] on simulators for network interconnects, [15] [14] on modeling of collective MPI operations and [22] on MPI communication overhead analysis.

3 Platform

In this section we describe the hardware and explain the software stack used in this work. Real performance data on HPCG (version 2.4) was collected on three different platforms, which shall be referred to as A, B and C. Furthermore, as will be described in section 6, large validation runs were performed on a further platform not included in the process of modelling.

- The *platform A* is based on the XC30 architecture. It contains 64 nodes with 2 chips of the Intel SandyBridge 2,6 GHz E5-2670 per node. Each chip has 8 cores (16 HT) with 20 MB of shared L3 cache per chip and 4 memory channels connected with DDR3 1600 MHz, which makes the maximum memory bandwidth 51.2GB/s. The cores use "Turbo Boost" technology that allows to increase the frequency to 3.3 GHz as long as the thermal budget is not exceeded. All of the cores can execute 8 Flop per cycle. Peak performance per socket is 166.4 Gflop/s. The interconnection network is an Aries[1] with a Dragonfly topology, a bandwidth of up to 117GB/s per node and a latency between the closest nodes of less than $2\mu s$.
- The *platform B* is based on AMD Opteron Interlagos 6276 processors (2 chips per node). Each chip has 16 cores and 16MB of shared L3 cache. The Interlagos with DDR3 PC3-12800 gives a memory bandwidth of 51.2 GB/s. Each core runs at 2.3GHz (up to 3.2 GHz with TurboCore) and can

execute 4 Flop per cycle. Peak performance per socket is 147.2 Gflop/s. The interconnect is of the type Gemini [2] with a 3D torus topology, 160 GB/s bandwidth per node and the lowest latency of $2\mu s$.
On platforms A and B, we use the GNU Programming Environment: C/C++ GNU Compiler 4.8, MPICH-6.2 Cray MPI library that uses the MPICH2 distribution from Argonne.

– The *platform C* is based on Intel Xeon X5560 chips (Nehalem), it contains 2 chips per node. The chip contains 4 cores (8 HT) and works at 2.80 GHz (3.20 GHz maximum for Turbo Boost). The L3 cache size is 8MB, 3 channels with 1333 MHz memory interface which delivers 32GB/s. Peak performance per socket is 44.8 Gflop/s.
The Infiniband interconnect of platform C uses Voltaire Grid Director 4036 switches with 36 QDR (40Gbps) ports (6 backbone switches). We use C/C++ GNU Compiler 4.8 and the OpenMPI library 1.6.5.

4 From the HPL to the HPCG benchmark

In this section, we review the HPCG benchmark. First we comment on the importance and relevance of a new benchmark for comparing the most powerful supercomputers in the world, then we describe the structure and routines of the implementation.

4.1 Transition

Two times per year, the TOP500 project publishes a ranking of the worlds most powerful supercomputers. The ranking is based on performance achieved for HPL, measured in Gflop/s. HPL implements a LU decomposition with partial pivoting.

The TOP500 measurement rules allows modifying the internal functions and tuning input parameters for an optimal result. This modification freedom has lead to significant restructuring of the code and redesigns of the input. Meanwhile, HPL is a highly optimized software package. Furthermore, the arbitrary problem size, combined with the compute complexity $\mathcal{O}(N^3)$ of the matrix multiplication, implies selecting the largest problem size that fits to the main memory. This straight forward choice leads to such enormous costs of the main kernel, that everything else becomes effectively irrelevant. Furthermore, due to the nature of the kernel, the Byte/Flop ratio is small and it is therefore purely compute-bound. HPL is a non-trivial code with complex communication patterns. However, for the reasons explained above the performance-optimized HPL runs display an almost identical behaviour to a pure matrix multiplication and reach around 90% of the theoretical peak performance.

Real-world applications usually do not have such an increadibly expensive compute-bound kernel, but a more diverse set of kernels, which do not dwarf communication costs. Furthermore, many kernels have large Byte/Flop ratios, rendering them memory-bound. Thus, with real applications 20% of theoretical

peak is considered very good. Finally, the trend in hardware development points towards decreasing ratios of available Bytes per Flop, which will decrease the achievable percentage of theoretical peak even further.

The lacking representativity of HPL is increasingly a matter of general discussion. HPCG has been developed with these discussions in mind, aiming at a far superior representativity. HPCG is not intended to replace HPL, but rather to complement it.

HPCG features the same freedom regarding modification of routines and problem size as HPL and retains Gflop/s as the unique evaluation metric. The crucial difference between HPL and HPCG is the respective main computational kernel. While in HPL it is a matrix multiplication, in HPCG it is a matrix vector multiplication. As we will show, the higher Byte/Flop of the matrix vector multiplication renders it memory-bound on current and most likely future hardware, leading to a better representativity for real-world applications.

4.2 HPCG structure and routines

In this paper, we used the HPCG version 2.4 without any further modification. The code is written in C/C++ and for parallelization the user can select MPI and/or OpenMP at compile time. The problem size and minimum execution time is specified in the input file. In order to produce official results, the execution time has to be at least 1 hour. The authors of HPCG suggested to increase the minimal problem size to achieve a memory footprint beyond L3 cache sizes [12], but this is currently not contained in the requirements for the benchmark (currently the minimal size is 16,16,16).

Comparing MPI with hybrid MPI/OpenMP on homogeneous clusters, we decide to model only pure MPI execution, as the performance is much higher than for hybrid execution. There are two main reasons for it: First, the HPCG algorithm is well balanced on MPI level, and the OpenMP dynamic balancing feature cannot deliver any performance improvement, and second, OpenMP threads are idle during communication operation due to the fork/join model which lead to performance drop.

The HPCG benchmark (version 2.4) is based on a conjugate gradient solver, where the pre-conditioner is a three-level hierarchical multi-grid method (MG) with Gauss-Seidel relaxation. The number of iterations is fixed at 50 per set, which is sufficient for the residual to drop below 1^{-6}. The structure of the main loop is shown in Figure 1.

The algorithm starts with MG that contains symmetric Gauss-Seidel(SYMGS) and sparse matrix-vector multiplication(SpMV) for each depth level. Data is distributed across nodes, thus SYMGS and SpMV require data from their neighbors. Their predecessor, an exchange halos(ExchangeHalos) routine, provides data for SYMGS and SpMV, therefore performing communication with neighbors. An iteration within the main loop also calls the SpMV/ExchangeHalos pair. Dot product (DDOT) locally computes the residual, while MPI_Allreduce follows DDOT and completes a global dot product operation. WAXPBY updates a vector with the sum of two scaled vectors.

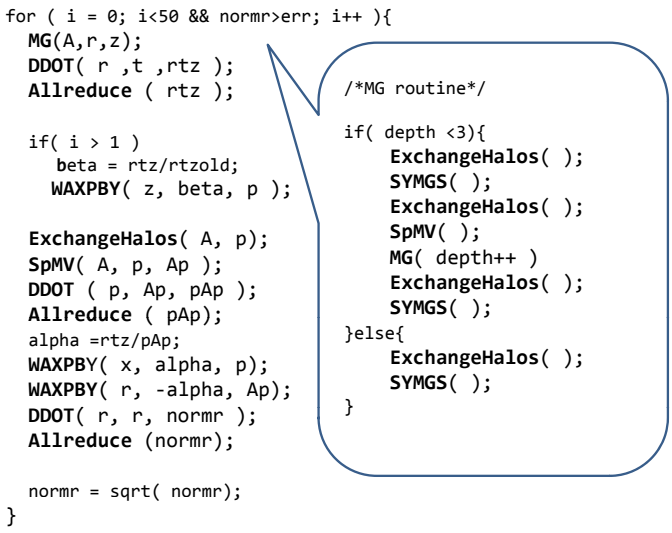

Fig. 1. Pseudocode of the HPCG main loop and the Multi-Grid routine. Modeled routines are basic blocks for the HPCG pseudocode.

The routines SYMGS, SpMV, WAXPBY, DDOT are the basic computational blocks of HPCG, while the routines MPI_Allreduce and ExchangeHalos are the basic communication blocks. In the next section we discuss internals of these routines and how they are modelled.

5 Model

In this section, we describe basic considerations for modelling the performance of HPCG. From these considerations, the executing time of every routine is derived. In combination with an estimation of the number of Flop, this results in a model for predicting the Gflop/s which can be achieved with HPCG on a given hardware.

5.1 Basic Considerations

Estimating execution time of a routine requires the following information: type and number of operations of the routine, the size of memory used by the routine and technical information on the machine (system description).

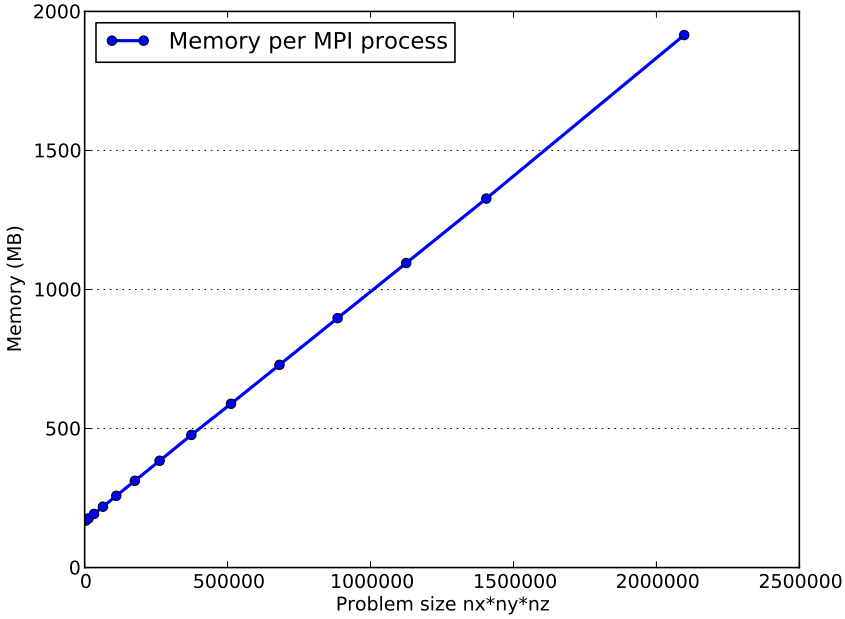

Fig. 2. Required memory per MPI process for different problem sizes of HPCG.

Large HPC machines in essence consist of two parts: shared memory nodes for computation and the interconnection network responsible for communication between nodes.

We discuss and model two distinct classes of routines: computational and communication. Communication depends on the interconnection characteristics, while computation depends on the characteristics of the compute nodes.

Memory vs. compute bound The execution of a computational kernel contains two phases:
– Memory operations, *i.e.* fetching data from memory and writing results back
– Execution of arithmetic and logic operations
The memory speed and the amount of data determine the execution time of memory operations, while the CPU clock speed, Flop per cycle, amount and type of operations define the execution time of computation. The more expensive of the two, memory operations vs. computation, limits the performance and thus renders the overall kernel memory- or compute-bound.

The metric Byte/Flop quantifies the amount of data a kernel requires to perform one Flop. When used for hardware, it quantifies the maximum amount of data which can be delivered per available Flop.

HPCG mainly performs a matrix-vector operation on sparse matrices. The number of Flop to be performed is $2 * nnz$, where nnz is the total number of non-zero elements. The size of one float (double) is 8 Bytes and the required memory is $(nnz+2 * n)$ * 8 Bytes, where n is the matrix dimension. For large problem sizes $nnz \approx 27 * n$. The Byte/Flop requirement of HPCG is therefore $\approx (28/27 * nnz * 8Bytes)/(2 * nnzFlop) > 4$Byte/Flop.

In order to check the limitation of HPCG we compare the Byte/Flop demand of HPCG with the respective values for the used hardware, as shown in table 1.

	HPCG	**Platform A**	**Platform B**	**Platform C**
Byte/Flop	> 4	0.3077	0.3478	0.7142

Table 1. Performance for platforms A-C in terms of Byte/Flop and the Byte/Flop requirement for HPCG.

Clearly, HPCG is memory-bound, as the amount of Flop that can be executed is limited by the maximum data throughput of the respective platform. Furthermore, none of the TOP500 machines offers anywhere near 4 Byte/Flop. A vector processor delivers up to 1 Byte/Flop, while a scalar processor usually delivers less than 0.5 Byte/Flop.

Thus, knowing the effective bandwidth and the required data for a given routine directly allows us to model the execution time. The theoretical peak of memory bandwidth is not reachable due to the internal implementation of the processor architecture and its resources. Further, there is no reliable way to estimate the effective memory bandwidth from the theoretical one. Therefore, measuring the effective bandwidth is an unavoidable step and in this work we use the Triad stream benchmark kernel. Usually one core cannot exploit the whole memory bandwidth of a socket, so we occupy all cores with a stream kernel, measuring the total effective memory bandwidth of the socket. From this, we compute the average effective memory bandwidth per core and use it for the performance model.

Communication In HPCG there is collective communication (MPI_Allreduce) and the routine HaloExchange, performing a set of point-to-point communication.

The amount of data in the MPI_Allreduce is independent of the problem size. However, the cost of the collective routine increases with number of MPI processes (N) and therefore becomes relevant for very large supercomputers.

The amount of data communicated in HaloExchange depends on the problem size. However, the cost is independent of N.

Problem size We evaluated HPCG on the node level for different problem sizes. Figure 3 shows performance results in terms of Gflop/s for different problem

sizes, running on three different platforms. Minimum problem size corresponds to the minimal size allowed for the benchmark, 16x16x16 3D sparse matrix per MPI process. We distinguish three different phases: problem size that fits to the L3 cache, a transition phase and a constant performance regime. The smallest possible problem size requires 2.1 MB per core and only fits to the L3 cache of the platform A (2.5MB/core). In the transition phase the problem is only slightly larger than L3 cache and the performance depends strongly on the cache properties. For larger problem sizes the performance asymptotically approaches a constant value and depends on the effective memory bandwidth to main memory only.

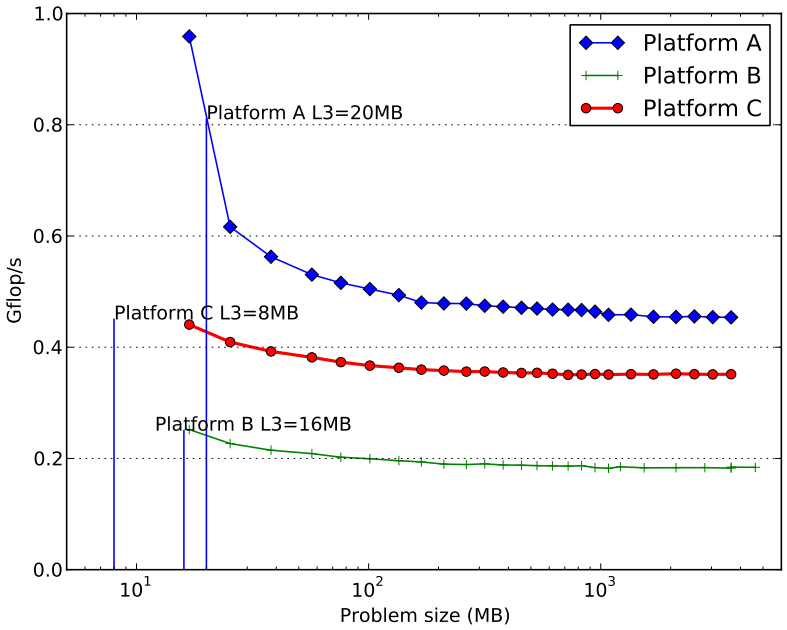

Fig. 3. Gflop/s of computational routines for different problem sizes.

If the memory footprint fits to the L3 cache, the kernels still are memory bound, however, the effective bandwidth of interest is to L3 cache instead of to main memory. The difference in these bandwidths is large. We argue that, while the bandwidth to main memory is a relevant metric regarding the suitability of a given hardware for real-world applications, the size of the L3 cache is less relevant. It therefore makes little sense that the size of the L3 cache has a large impact on the measured performance. We suggest therefore to increase the minimal problem size, in order to avoid this scenario, which is in line with the

opinion of the authors [12]. In the following we will assume this limitation is in place.

5.2 Modelling computational routines

The execution time of all computational kernels depend on the size of the 3D sparse matrix and the number of non-zero elements per local row. The number of local rows is equal to $nx * ny * nz$, while the number of non-zero elements in a row is 27 or fewer. The number 27 is hardcoded within the benchmark. The limit is shown below:

$$\lim_{nx*ny*nz \to \infty} numberOfNonzerosPerRow = 27$$

For a large problem size we consider the number of non-zeros as equal to 27. Instances of computational routines called directly from the main loop work on the whole domain, while the MG routine calls recursively a set of computational routines, reducing the resolution per depth. We will discuss the MG routine further down.

The number of outer iterations (LNI) appears in all computational kernels and sub-kernels:

$$LNI = \frac{nx * ny * nz}{2^{3*d}}$$

SYMGS The Symmetric Gauss-Seidel Method is the most expensive routine in the benchmark (except for MG, which is a combination of routines). It performs two steps: forward and backward sweeps. Regarding the memory footprint and number and type of operations, the two steps are identical. Each step performs a two dimensional loop, the outer number of iterations being LNI and the inner number 27. Pseudo code is shown below:

```
Loop j=1..LNI(depth)
  Loop i=1..27
    c[i]+=a[j]*b[index[j]
  endloop
  c[i]+=d[i]*b[i]
  b[i]=c[i]/d[i]
endloop
```

The kernel is based on a Flop with double precision, where one factor has indirect addressing. The two dimensional loop fetches two doubles and one integer in each iteration which makes 20 Bytes in total (arrays a and b in pseudo code). The process unit also fetches arrays c and d in the outer loop and a number of non-zero elements, which increases the total size of data in memory. We model the execution time by dividing the required data by the effective bandwidth from main memory($BWeff$):

$$executionSYMGS(sec) = 2 * \frac{LNI * (20 + 20 * 27)(Bytes)}{BWeff(Bytes/sec)}$$

SpMV The SpMV routine is very similar to SYMGS, but performs only one step. The pseudo code is:

```
Loop j=1..LNI(depth)
  Loop i=1..27
    c[i]+=a[j]*b[index[j]
  endloop
  d[i]=c[i]
endloop
```

The number of iterations, memory accesses and main computational operations are the same for both routines. The model is therefore analoguous to above:

$$executionSpMV(sec) = \frac{LNI * (20 + 20 * 27)(Bytes)}{BWeff(Bytes/sec)}$$

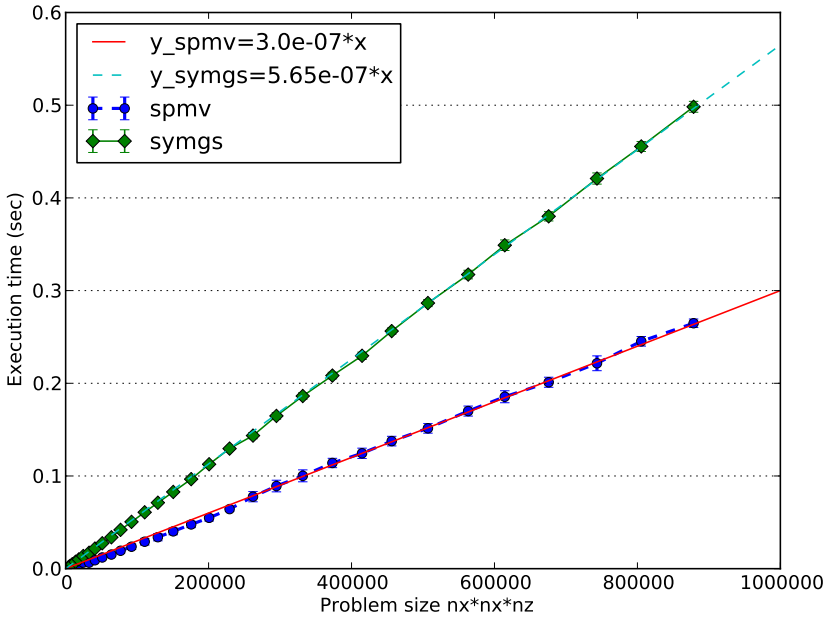

Fig. 4. Modeled and measured execution time of the two most expensive computational routines.

WAXPB The WAXPB routine behaves like a triad vector kernel, which is the most complex scenario of all stream vector kernels. The update of two scaled vectors is shown below.

```
Loop i=1..LNI(depth=0)
  c[i]=alfa*a[i]+beta*b[i]
endloop
```

Vectors a, b and c contain elements of size double. So, 24 Bytes from memory are required for every iteration. The number of iterations is always the same for a given input set as only the main loop calls the WAXPB routine directly. The execution time is modelled as:

$$executionWAXPB(sec) = \frac{LNI * 24(Bytes)}{BWeff(Bytes/sec)}$$

DDOT First the DDOT routine computes locally a dot product before performing a global sum operation across the system. The multiplication of vector elements and accumulation of the results into a single variable in pseudo code:

```
Loop i=1..LNI(depth=0)
  c+=a[i]*b[i]
endloop
```

While computationally WAXPB and DDOT are different, their memory footprint is very similar. However, the DDOT routine only requires 16 Bytes per iteration. The execution time without communication is modelled as:

$$executionDDOT(sec) = \frac{LNI * 16(Bytes)}{BWeff(Bytes/sec)}$$

5.3 Modelling communication

We execute HPCG in parallel using MPI, which requires static data distribution across processes with separated address spaces. Naturally, the data decomposition is 3-dimensional due to the 3D sparse matrix. Each process receives the same input size and the algorithm is almost perfectly load balanced. Communication between processes uses the MPI interface and there are two communication routines: MPI_Allreduce that finalizes the DDOT routine and a halo exchange between neighbouring MPI processes.

Both routines use the MPI_COMM_WORLD communicator. There is no interleaving of communication between different communicators, which makes routing in the IC network easy. Both routines use blocking MPI calls and the nature of the algorithm holds no potential for overlapping communication and computation. The communication behaves as synchronization points for all processes.

Collective The MPI_Allreduce is the only collective communication used in the algorithm. The operation reduces a single variable of size double over all processes. As with all collective operations, the MPI_Allreduce implementation relies on point-to-point communication and the optimal implementation depends

on the topology of the IC network itself. The hypercube algorithm performs reduce among N processes in $log(N)$ steps. The algorithm reduces the information in the least number of steps necessary and shows the highest efficiency for regular topologies. The amount of data per communication step is 8 Bytes, thus we consider the latency between processes as the only relevant IC parameter and disregard the bandwidth in the model.

For N MPI processes and a given latency l between processes, we predict the execution time of a MPI_Allreduce operation as

$$executionAllreduce(sec) = \sum_{i=1}^{M} l_i(log(M_i) - log(M_{i-1}))$$

Each index refers to a group of processes with the same latency. E.g. l_0 refers to the latency between MPIs within a socket, l_1 refers to the latency between MPIs within a node that are located on different sockets, l_2 refers to the latency between MPIs within a blade that are located on different nodes etc. M_i is the maximum number of MPI processes in a group with the same latency time.

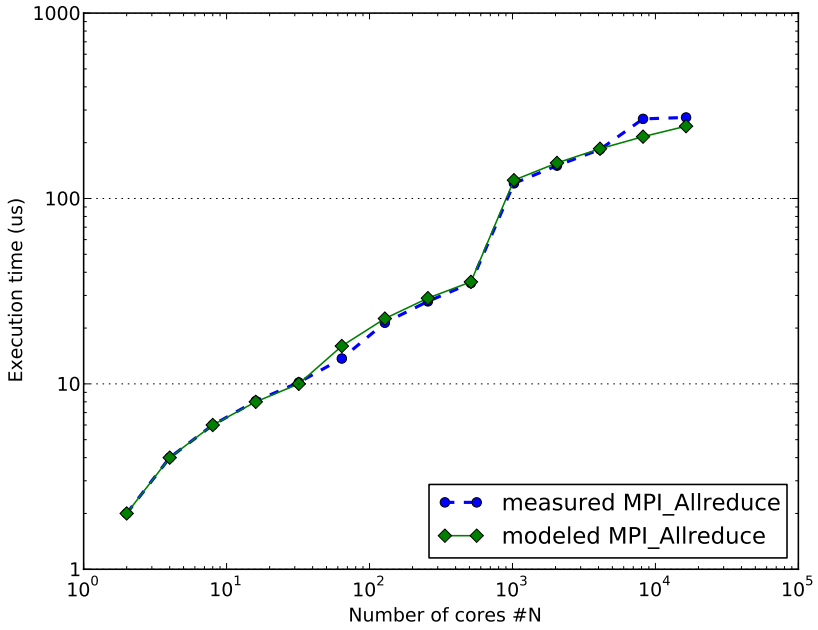

Fig. 5. Modeled and measured execution time of the MPI_Allreduce operation (8 Byte)

Point to point The halo exchange is a nearest-neighbor data exchange. It is a common communication pattern for MPI-HPC applications. The number of neighbors of a given MPI process depends on the location in the decomposition grid, the maximum being 26 neighbors. If HPCG is run on 27 or more MPI processes at least one process has 26 neighbors in the halo exchange phase. The maximum data size that one process receives or sends during a single halo exchange instance is:

$$maxHaloSize(Bytes) = (2(nx * ny + nx * nz + ny * nz) + 4(nx + ny + nz) + 8) * 8Bytes$$

The MG routine calls the halo exchange from different depths, reducing the halo size by a factor of $2^{2*depth}$. Figure 6 shows the execution time of the ExchangeHalos routine for different problem sizes. Rendezvous protocol introduces a significant performance drop which should be part of the model.

We assume the minimal effective bandwidth for data movement across the IC in the halo exchange (IC_BW_{eff}). As even for large workloads maxHaloSize is relatively small for modern IC networks, the overhead of the MPI call plays an important role. Halo exchange is achieved through a sequence of MPI_Irecv, MPI_Send and MPI_Wait. The overall model is:

$$executionHaloEx(sec) = \frac{maxHaloSize}{IC\_BW_{eff}} + 26(overhead(Irecv, Send, Wait)) + overhead(Rendezvousprotocol)$$

The overheads for Irecv, Send and Wait can be directly determined from the latency, the overhead of the rendezvous protocol is determined from the MPI pingpong benchmark. The latter can be easily avoided by adjusting the corresponding parameter, however, as it barely impacts the model, we did not do so.

5.4 Modelling the whole benchmark

MG – a combination of routines The MG routine combines multiple routines and calls them from different depths. The multi-grid level decreases the problem size by $2^{3*depth}$.

Thus, larger depth indicates smaller problem size and shorter execution time. In the forward recursion phase, the MG calls the sequence HaloEx-SYMGS-HaloEx-SpMV up to depth 2, while depth 3 performs only HaloEx-SYMGS. The backward recursion phase calls HaloEx-SYMGS. The following sum gives the execution time of the MG routine.

$$executionMG = HaloEx(depth = 3) + SYMGS(depth = 3) + \sum_{depth=0}^{2} (2 * SYMGS(depth) + SpMV(depth) + 3 * HaloEx(depth))$$

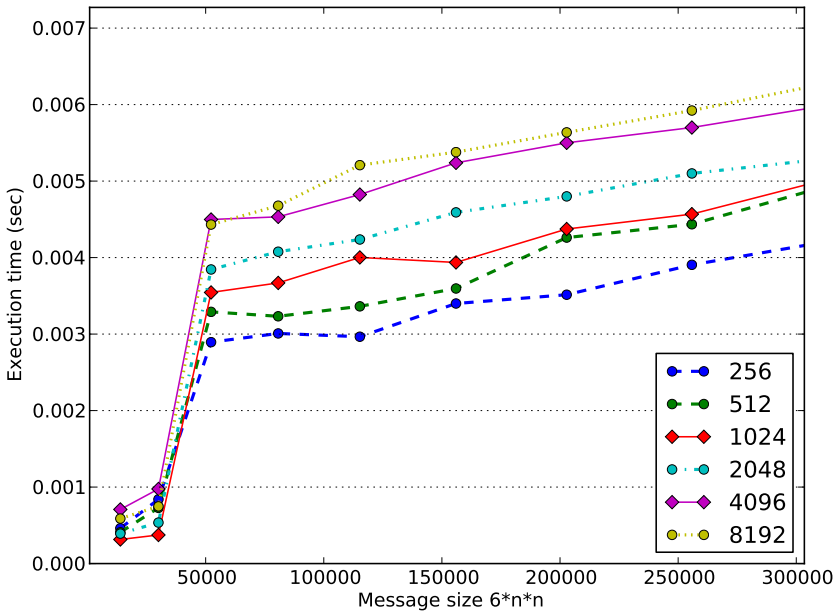

Fig. 6. Measured execution time of halo exchange for platform B with increasing message size. The impact of the rendezvous protocol is clearly visible as a jump in execution time.

Total execution time The main loop does 50 iterations, calling the sequence MG-DDOT-WAXPB-SpMV-DDOT-WAXPB-WAXPB-DDOT. The first iteration calls one instance of WAXPB less, we forgo considering this in the model. Thus, the execution time of one iteration is modelled as:

$$totalTime = MG + SpMV(depth = 0) + 3(DDOT + WAXPB)$$

Predicting Gflop/s In order to calculate the Gflop/s, HPCG predicts the number of floating point operations necessary per routine and measures the execution time. Input data set and the total number of non-zeros define the total number of floating point operations. If we assume 27 non-zero elements per row for a large problem size, the total number of non-zero elements is:

$nnz = N * nx * ny * nz * 27$

The resulting total number of floating point operation is:

$$\begin{aligned} MG_{flop} &= 10 * (nnz + nnz/8 + nnz/64) + 4 * (nnz/128) \\ SMPV_{flop} &= 2 * nnz \\ DDOT_{flop} &= 6 * N * nx * ny * nz \end{aligned}$$

$$WAXPB_{flop} = 6 * N * nx * ny * nz$$

Combined with the prediction of the execution time, this allows us to predict the achieved Gflop/s and thus completes the performance model of HPCG. The model is suitable for large problem sizes (per MPI process) and is viable even for very large systems, which matches the HPCG target as a new benchmark for the TOP500 list.

6 Results

We have validated the proposed performance model by comparing predicted performance values to measured results. The model shows excellent predictability of HPCG performance. Based on the model we then predicted the performance on envisioned future systems.

6.1 Validating the model

The essential part of the HPCG model is a prediction of computational routines which have almost constant execution time for different numbers of cores.

We analyze the HPCG routines for a large problem size and different number of cores. For all data shown, large refers to the size (96x96x96) per core. Figure 7 compares percentages of execution time per routine, measured on platform B. The MG pre-conditioner clearly is the most expensive routine, taking more than 80% of the total execution time and very slowly become less important for larger numbers of cores, while the MPI_Allreduce slowly becomes more important. The computational routines take more than 98% of the total execution time.

In order to determine the effective bandwidth to main memory and the latency of the IC, we obtained results by using the Triad stream benchmark kernel and a MPI pingpong benchmark respectively. Table 2 shows results for the different platforms.

	Platform A		Platform B		Platform C	
BWeff(MB/s) per core	4705		1700		3430	
IC latency(μs) (min, max)	2	4	2	90	4	240

Table 2. Effective memory bandwidth measured by using the Triad stream kernel and minimal/maximal IC latency measured by the osu mpi benchmark.

Figure 8 compares the measured and modeled HPCG computational routines per node for platforms A-C.

Figure 9 shows the measured performance results vs. the predicted performance for the whole benchmark. As can be seen, HPCG scales approximately

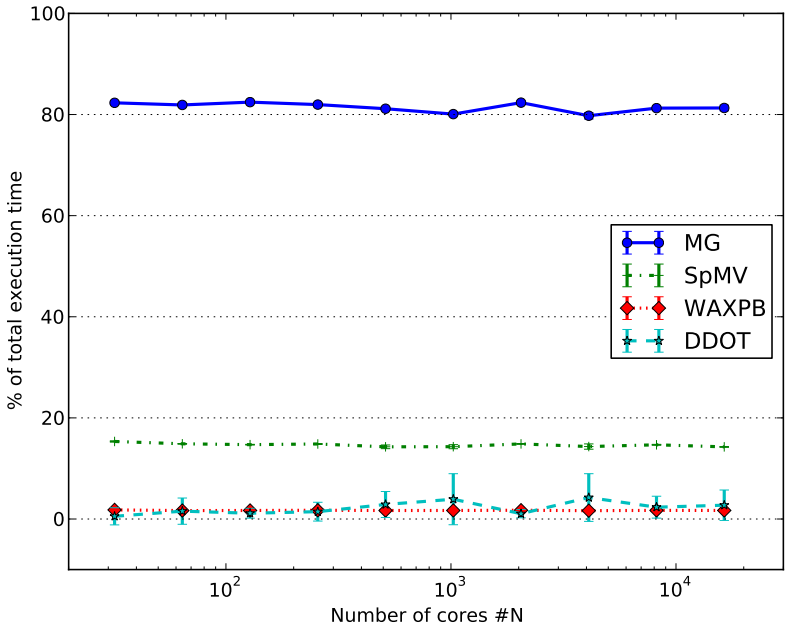

Fig. 7. Percentage of HPCG total execution time per routine. Measured on platform B.

linearly with N and the performance predictions from our model are very accurate (deviations of less than 2%).

Finally, we tested our model by predicting the performance for a full HPC system, of which we had no performance data during model creation, and subsequently comparing to the real performance results. This supercomputer is based on XC40 nodes and an Aries interconnection network. Each node contains two Intel Haswell E5-2680v3 2,5 GHz, 12 cores per socket with 128GB of DDR4-2133 RAM memory. Each socket has 30MB of L3 cache. Having determined the effective memory bandwidth (3740MB/s per core) and the maximum IC latency (3 μs), we predicted the performance and then ran HPCG across all 3900 nodes (93600 cores) with a problem size of (nx,ny,nz)=(144,144,144) per MPI process. Our model predicts the overall performance to within 1% of the real value.

6.2 Extrapolating HPCG performance to future systems

High requirements for Byte/Flops renders the computational kernels of HPCG memory-bound for all modern machines. In order to predict HPCG benchmarking potential for future exascale systems, we consider that all computational kernels will remain memory-bound, which is to be expected.

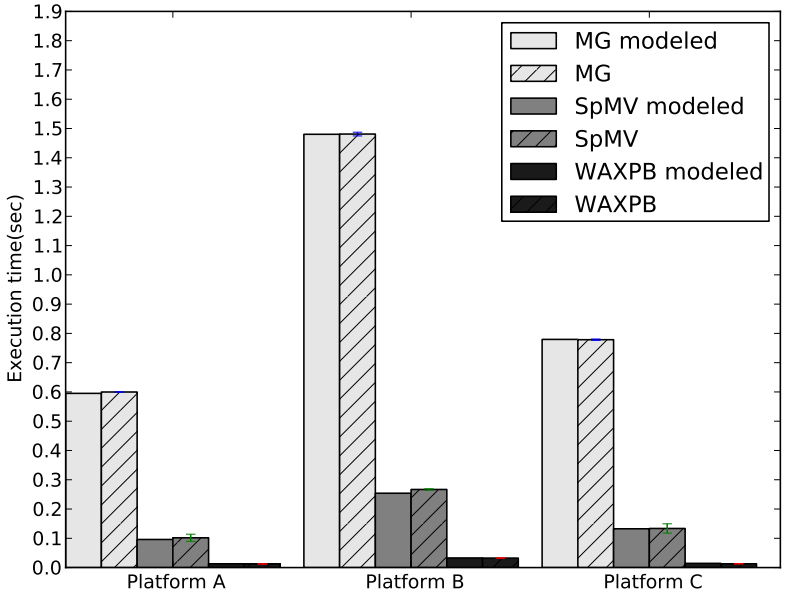

Fig. 8. Predicted vs. measured execution time per computational routine for platforms A-C.

Further, communication obviously costs time without producing Flop/s. Therefore, it is to be expected that the benchmark will be run with the largest problem size which fits to the main memory in order to increase the computation to communication ratio. This is analoguous to what can be observed for the HPL benchmark. Machines featured in the TOP500 currently have 2GB of main memory or more per physical core. According to the memory usage formula from section 4 the largest problem size which fits to main memory is (128,128,128) per MPI process and we assume future systems will have similar amounts of main memory per core.

Communication cost grows with N due to the MPI_Allreduce. In Figure 10 we show an extrapolation to very large numbers of cores for platform B, the problem size is taken as (128,128,128). We have evaluated for the current hardware properties, and furthermore changed one property at a time by one order of magnitude. For the current setting, the communication cost stays below 1,2% of the entire execution time for machines with up to one million cores. Furthermore, for one billion cores, the communication still costs below 3%. As can be seen in Figure 10, unless the available Byte/Flop ratio or the IC latency increase significantly for future systems, the communication cost will remain irrelevant even at the exascale.

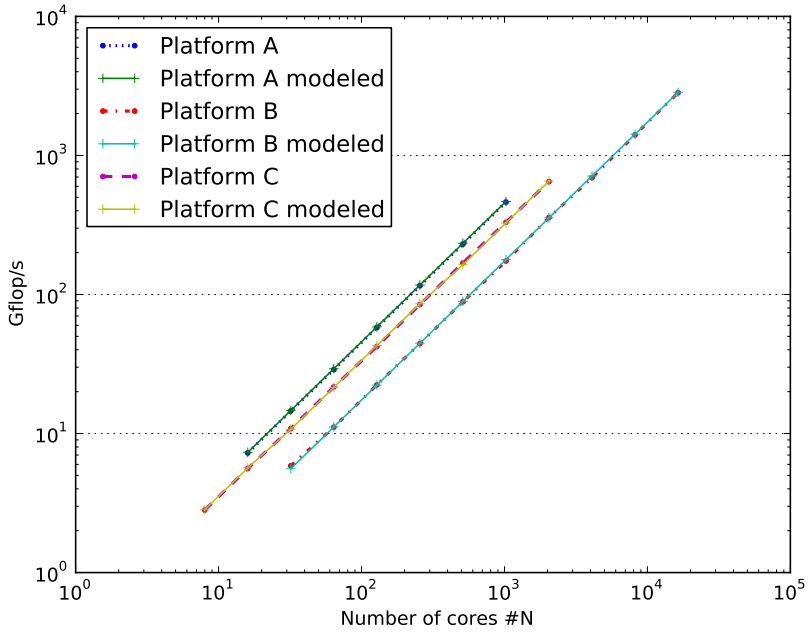

Fig. 9. Measured performance results vs. the prediction for HPCG on platforms A-C.

7 Conclusion

The TOP500 list relies on HPL, a benchmark increasingly unrepresentative for the performance issues real-world applications face today. Thus, as hardware vendour try to boost HPL results, hardware development is subtly steered towards increasing overall compute cycles and frequencies, which cannot be exploited and even introduce overheads in energy consumption. HPCG, a prominent candidate for the next step in supercomputer benchmarks, moves into the right direction: by featuring memory-bound kernels, it reflects the bottlenecks of real-world applications more realistically.

As we have demonstrated in this paper, it is possible to predict the performance of HPCG with accuracy, relying only on two numbers: effective memory bandwidth from main memory and highest occurring IC latency. Obviously, the logical conclusion is, that the performance of HPCG only depends on these two numbers. The effective memory bandwidth determines the necessary time to execute the computational kernels, as the limiting factor is the availability of data to perform computations on. The IC latency determines the time to perform the MPI_Allreduce, as the amount of data (per process) being communicated is only 8 bytes and therefore IC bandwidth is irrelevant. The MPI_Allreduce is the only

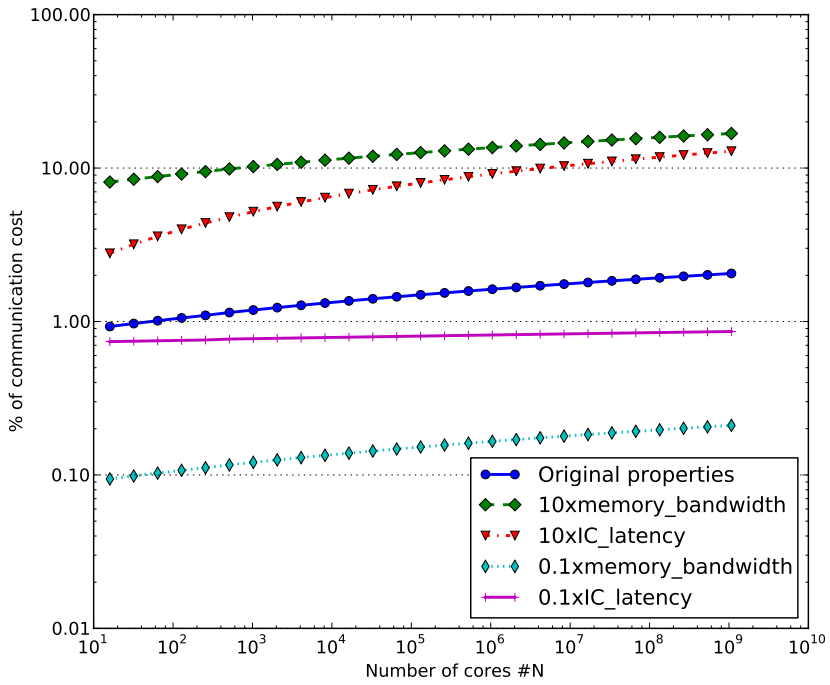

Fig. 10. Extrapolating the relative importance of communication cost in HPCG to huge HPC systems. Original properties reflect the platform B, the other lines are predictions for systems differing in IC latency or effective memory bandwidth, respectively, by one order of magnitude.

relevant communication, as it's time requirement increases with $\log(N)$, while the point-to-point communication is constant in N.

Furthermore, as shown in the paper, the problem size is extremely relevant. Especially, the smallest hitherto allowed problem size can fit into the L3 cache on certain hardware. In this case, the effective bandwidth of interest of course is to the L3 cache, which is a lot higher then to the main memory, speeding up the benchmark accordingly. We argue that this is a problem: Hardware with sufficiently large L3 cache get a huge competitive edge, which is not sensible. We argue the problem size should have a much more restrictive lower limit, to ensure it does not fit to L3 cache, as discussed in [12]. Given that small problem sizes will be restricted, the obvious choice will be to increase the problem size to the limit of main memory, thereby diminishing the relative cost of communication. We have shown that for a representative current supercomputer architecture, this strategy effectively dwarves communication cost. Keeping the hardware specifications and extrapolating to a system with one billion cores, the

communication cost is less than 3%. Thus, even the IC latency can be effectively ignored, rendering the overall performance approximately proportional to the effective memory bandwidth of one single node.

Thus, HPCG is in danger of encountering the same problems as HPL: by allowing arbitrary problem sizes only one system property is relevant for the final result, while the performance of real-world applications depends on a much more diverse set of properties. We suggest this approach be reconsidered. Even simple changes to the execution protocol could drastically improve on this. For example, running a suite of short simulations with varying, predefined system sizes and reporting the (weighted) average performance.

Our model targets performance prediction for the official, unmodified HPCG benchmark (version 2.4) run on homogeneous clusters. In the future, we plan to extend our model to heterogeneous architectures, *e.g.* featuring GPGPU and Xeon Phi accelerators.

Acknowledgements

The authors would like to thank Mandes Schönherr for valuable contributions. This research is partly supported by EU project POLCA (FP7-ICT-2013-10, grant agreement no. 610686).

References

[1] Bob Alverson, Edwin Froese, Larry Kaplan, and Duncan Roweth. Cray xc® series network.

[2] Robert Alverson, Duncan Roweth, and Larry Kaplan. The gemini system interconnect. In *High Performance Interconnects (HOTI), 2010 IEEE 18th Annual Symposium on*, pages 83–87. IEEE, 2010.

[3] Steven F Ashby and Robert D Falgout. A parallel multigrid preconditioned conjugate gradient algorithm for groundwater flow simulations. *Nuclear Science and Engineering*, 124(1):145–159, 1996.

[4] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The nas parallel benchmarks—summary and preliminary results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 158–165, New York, NY, USA, 1991. ACM.

[5] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The nas parallel benchmarks. Technical report, The International Journal of Supercomputer Applications, 1991.

[6] Michele Benzi. Preconditioning techniques for large linear systems: a survey. *Journal of Computational Physics*, 182(2):418–477, 2002.

[7] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröoder. Sparse matrix solvers on the gpu: conjugate gradients and multigrid. In *ACM Transactions on Graphics (TOG)*, volume 22, pages 917–924. ACM, 2003.

[8] Aydin Buluc, Samuel Williams, Leonid Oliker, and James Demmel. Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 721–733. IEEE, 2011.

[9] Franck Cappello and Daniel Etiemble. Mpi versus mpi+ openmp on the ibm sp for the nas benchmarks. In *Supercomputing, ACM/IEEE 2000 Conference*, pages 12–12. IEEE, 2000.

[10] James Demmel, Mark Hoemmen, Marghoob Mohiyuddin, and Katherine Yelick. Avoiding communication in sparse matrix computations. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–12. IEEE, 2008.

[11] Jack Dongarra and Michael A Heroux. Toward a new metric for ranking high performance computing systems. *Sandia Report, SAND2013-4744*, 312, 2013.

[12] Jack Dongarra and Piotr Luszczek. Hpcg: One year later, 2014. ISC 2014.

[13] Michael A. Heroux, Jack Dongarra, and Piotr Luszczek. Hpcg benchmark technical specification. Technical report, Oct 2013.

[14] Torsten Hoefler, William Gropp, Rajeev Thakur, and Jesper Larsson Träff. Toward performance models of mpi implementations for understanding application scaling issues. In *Recent Advances in the Message Passing Interface*, pages 21–30. Springer, 2010.

[15] Torsten Hoefler, Andrew Lumsdaine, and Wolfgang Rehm. Implementation and performance analysis of non-blocking collective operations for mpi. In *Supercomputing, 2007. SC'07. Proceedings of the 2007 ACM/IEEE Conference on*, pages 1–10. IEEE, 2007.

[16] Piotr Luszczek, Jack J Dongarra, David Koester, Rolf Rabenseifner, Bob Lucas, Jeremy Kepner, John McCalpin, David Bailey, and Daisuke Takahashi. Introduction to the hpc challenge benchmark suite. *Lawrence Berkeley National Laboratory*, 2005.

[17] Matthias S Müller, Matthijs van Waveren, Ron Lieberman, Brian Whitney, Hideki Saito, Kalyan Kumaran, John Baron, William C Brantley, Chris Parrott, Tom Elken, et al. Spec mpi2007an application benchmark suite for parallel systems using mpi. *Concurrency and Computation: Practice and Experience*, 22(2):191–205, 2010.

[18] Antoine Petitet. Hpl-a portable implementation of the high-performance linpack benchmark for distributed-memory computers. *http://www. netlib-. org/-benchmark/hpl/*, 2004.

[19] John Shalf, Sudip Dosanjh, and John Morrison. Exascale computing technology challenges. In *High Performance Computing for Computational Science–VECPAR 2010*, pages 1–25. Springer, 2011.

[20] JE Smith and WR Taylor. Accurate modelling of interconnection networks in vector supercomputers. In *Proceedings of the 5th international conference on Supercomputing*, pages 264–273. ACM, 1991.

[21] Zoltán Szebenyi, Brian J. N. Wylie, and Felix Wolf. SCALASCA parallel performance analyses of SPEC MPI2007 applications. In *Proc. of the 1st SPEC International Performance Evaluation Workshop (SIPEW), Darmstadt, Germany*, volume 5119 of *Lecture Notes in Computer Science*, pages 99–123. Springer, June 2008.

[22] Zhiwei Xu and Kai Hwang. Modeling communication overhead: Mpi and mpl performance on the ibm sp2. *Parallel & Distributed Technology: Systems & Applications, IEEE*, 4(1):9–24, 1996.