D4.2.1 – Prediction Model for identifying limiting Hardware Factors

WP4: Algorithms and libraries

Project Acronym	CRESTA
Project Title	Collaborative Research Into Exascale Systemware, Tools and Applications
Project Number	287703
Instrument	Collaborative project
Thematic Priority	ICT-2011.9.13 Exascale computing, software and simulation

Due date:	18
Submission date:	31/03/2013
Project start date:	01/10/2011
Project duration:	36 months
Deliverable lead organization	University of Stuttgart (USTUTT)
Version:	1.0
Status	Final
Author(s):	Stephen P Booth (UEDIN), Uwe Küster (USTUTT), Stephen Sachs (CRAY UK), José Gracia (HLRS), Gregor Matura (DLR), Dmitry Khabi (USTUTT), Mhd. Amer Wafai (USTUTT)
Reviewer(s)	Mark Bull (UEDIN), Jens Doleschal(TUD)

Dissemination level	
PU	*PU – Public*

Version History

Version	Date	Comments, Changes, Status	Authors, contributors, reviewers
0.1	07.01.2013	Definition of document structure	Dmitry Khabi (USTUTT) Uwe Küster (USTUTT)
0.2	12.02.2013	Contributions regarding the read of a sparse matrix	Gregor Matura (DLR)
0.2	15.02.2013	Executive Summary and Introduction	Dmitry Khabi (USTUTT)
0.3	16.02.2013	CPU Performance	Dmitry Khabi (USTUTT)
0.3	22.02.2013	CG Prediction Model	Dmitry Khabi (USTUTT) Uwe Küster (USTUTT)
0.4	28.02.2013	Performance models and limiting factors for FFTs	Stephen P Booth (UEDIN)
0.5	01.03.2013	Stephen Sachs	Stephen Sachs (CRAY UK)
0.6	01.03.2013	Performance of GPU for MvM and CG	Mhd. Amer Wafai (STUTT)
0.7	0.1.03.2013	Network and performance of collective operations	José Gracia (USTUTT)
0.8	14-19.01.2013	Addressing of referee comments	All
0.9	19.03.2013	Changes in Executive Summary and Introduction	Dmitry Khabi (USTUTT)
1.0	20.03.2013	Final version	Dmitry Khabi (USTUTT)

Table of Contents

Index of Figures

Index of Tables

1 Executive Summary

Hardware is one of the main factors to consider for the efficient use of massive parallel systems. It is also important to understand the main limiting factors that influence the efficiency of existing and developing programs. To successfully exploit an exascale system both hardware and software need consideration.

The purpose of this document is to support the further implementation of library “exascale algorithms and solvers” in the CRESTA work package 4 (WP4). We have performed many tests on different platforms to determine their differences and most important limiting factors (A description of the hardware used can be found in the attachment “Platforms” at the end of this document).

In the first part of this deliverable we report performance data for the newest processors and their nearest future development. This allows us to predict what new features in the local computation will appear, and what restrictions will not change. We will show how hardware features impact the performance and power consumption of a set of kernel operations (add, dot product and sparse matrix vector multiplication). These kernels are the basic operations, which will be used in the developing libraries. We will not consider the embedded processors (e.g. ARM) although these are also of great interest. There are many projects that attempt to integrate such processors into a supercomputer. For example the aim of the Green Flash Project (a development at Berkeley Lab) is to design a special-purpose supercomputer to perform climate simulations (1). However, there are still many unresolved questions. One of the most important is that these processors are difficult to integrate into a high performance network (2). The other problem is that compilers, operating systems, applications and libraries must be partially rewritten for such kinds of processor.

On other hand network performance is key to most application loads on HPC systems. That is also true for the numerical library, which is under developing in WP4. In the section “Network and performance of collective operations” of this document we consider not only the current features of the hardware but also it’s nearest future developing. The main question is, what are the main issues that need to be considered in the development of new collective operations within the CRESTA project for its successful usage in the numerical library "exascale algorithms and solvers".

Hardware accelerators are already being used for a while for the numerical calculations. We examined its weak and strong sides in the section “Hardware accelerators”.

The IO System is as important as the other components of an HPC system. However, we use the IO server rather to debug and test purposes. Nerveless we briefly describe the state of the art of the IO Server in this deliverable too.

Hardware failures are becoming a more important issue with increasing levels of parallelism. In this deliverable we consider “fault tolerant MPI” as a possible remedy for this limiting hardware factor and explore its influence on possible exascale systems. In the development of "exascale algorithms and solvers" we must not forget this issue. The suggested model will help us to consider this.

Furthermore we compare a set of results of numerical calculations on modern hardware with their theoretical performance. The performance prediction model of a set of numerical algorithms from the "exascale algorithms and solvers" library is also the part of this document. These and the discussions in the CRESTA Deliverable 2.1.1 “Architectural developments towards exascale” allows us to draw conclusions about limiting hardware factors for current petaflop platforms and their next generations and will allow us to choose the best way for the implementation of the library.

2 Introduction

Parallel computation uses a variety of streams to deliver the data via network, memory channels, cache and registers to the distributed arithmetic components and to store the results of its calculation on different storage units. Streams may consist of single numbers as well as arrays of numbers. Each of these streams has its own qualitative and quantitative properties. It depends not only on the characteristics of the hardware and the algorithm, but also on the programming model, compilers and runtime-system. The developers have to provide enough information to the compiler about dependencies between streams, destination of streams (ex. temporal, non-temporal) and other useful and necessary information. In this case the efficiency of a compiled program is limited generally by the “intelligence” of the compiler and the underlying hardware (which is involved in transport of these streams). In chapter “Hardware performance” we determine the most important performance characteristics of the hardware.

On the other hand, one of the biggest challenges of distributed system is that the performance of different parts is very different: performance of processors, memory, networks, hard disks differ by several times. Therefore it is important for each algorithm to develop a prediction model that considers it. With it, we cannot only choose the best algorithm for the problem, but also the best way of its usage. Several examples of such models can be found in chapter “Performance prediction models”.

At the end of this document in the chapter ”Conclusion” we summarize the key limiting factors an major trends in the hardware developments and its influence on the developing of "exascale algorithms and solvers" library.

2.1 Glossary of Acronyms

AVX	Advanced Vector Extensions
Cores	Hardware cores - In terms of possible hyper-threading, we consider in this document, only the actual number of cores on the chip.
CPU	Central processing unit
FP	Floating point number
CG	Conjugate Gradient – Krylov subspace solver
GPU	Graphics processing unit
GiB	2^30 = 1024*1024*1024 Bytes
KiB	2^10 = 1024 Bytes
MiB	2^20 = 1024*1024 Bytes
MPI_Allreduce	Reduce operation; its result is returned to all processes.
MPI_Alltoall	All processes send the same amount of data to each other
MV	Matrix vector (e.g. multiplication)
RAM	Main memory
MDS	Metadata Server
OSS	Object Storage Server
SpMVM	Sparse Matrix-Vector Multiplication

3 Hardware performance

3.1 Introduction

In this chapter we look at the individual components of a parallel system: processors, hardware accelerators (GPU), network and IO system.

The determined performance and limitation factors of the hardware components will be used by us in the implementation of the library, for example: to determine whether a time consuming function should be executed on CPU or GPU, we have to consider not only the performance but also the transfer time, kernel calls, and so on (see sections “Processors” and “Hardware accelerators”).

It is also important to know how large the optimal number of computational nodes for a particular job is. In addition to the CPU / GPU performance, it is important to know how the network behaves depends on the number of nodes (see sections “Network and performance of collective operations” and “Sparse-matrix vector multiplication (SpMVM) on NEC Nehalem cluster”).

In this chapter we wish also to express our opinion about the nearest future development of the hardware components.

3.2 Processors

Intel processors currently appear to have a lead in the area of high performance computing. This confirms the fact that the latest CRAY XC30 supercomputer uses Sandy Bridge processors. CRAY XC30 is designed to scale high performance computing (HPC) workloads of more than 100 petaflops. (3)

In sections “State of the art” we consider some important features of the Intel Sandy Bridge processor. A detailed study (4) of the most recent processors architectures shows that the Sandy Bridge processor has much in common with his successor Haswell.

In section “Future developing“ we will not explicitly predict limiting factors for the Haswell, but a view of certain features of Sandy Bridge will give a good overview for future processor development, due to their similarity.

In the last two sections we will consider one of the most important and time-consuming operations in the scientific computation, namely the sparse matrix vector multiplication.

The received information will be used in section “Iterative solver for sparse linear systems” of chapter “Performance prediction models”.

3.2.1 State of the art

There is always a difference between theoretical peak performance and the performance, which can be achieved in the applications. But it should be noted that the slowdown of Moore’s Law (5) leads to the architecture of the processor becoming more and more complex and “smarter”, which leads to better utilization of the CPU resources. On the other hand, this increasing complexity requires more effort for compiler designer and also for the programmers.

One of the most important limiting factors of the processors is the bandwidth and the latency of the different hierarchical levels of cache and memory.

We consider the performance of two simple examples:

- Add: *a[i] = b[i] + c[i]*
- Dot product: dot += *b[i] * c[i]*

The intrinsic functions and loop unrolling were used in the implementation of these examples (Intel compiler flag -O1). This gives us the opportunity to better control the hardware. On the other hand the compiler is limited in the optimization of the code. This is clearly visible in the comparison of the results, which will be carried out below.

The length of the arrays is chosen so that 80% percent of the cache or memory was used.

The performance measurements were done on the processor Intel E5-2687W (Sandy Bridge 3.1 GHz, Turbo 3.4-3.8 GHz, 8 cores, 4 memory channels by 1600 MHz, Launch Date Q1'12). In the example, we change the active hierarchy level of cache and memory (L1, L2, L3 and RAM – via different length of arrays) and the number of the cores. We change also the frequency of the processor for each test. For a better overview, we use the metric aggregated CPU frequency for the x-axis. The aggregated CPU frequency depends on the number of active cores and their frequency. For example if two cores are used for the calculation and each runs with 1.2 GHz, the aggregated frequency is equal to 2.4 GHz. The performance of the calculation on 1, 4 and 8 cores is shown on the Figure 1. The y-axis is logarithmic. The achieved bandwidth can be calculated by the multiplication of the performance, size of the data type and number of streams needed for the calculation:

$$\frac{elements}{second} * size\ of\ element * number\ of\ streams$$

Before the data will be stored it must be read. The consequence is that the store access requires two streams. . The theoretical and calculated bandwidth by usage of all cores is shown in Table 1.

.

Mem.	Latency (cycles)	Size	Theoretical Bandwidth (bytes/cycle or GiB/sec)		Bandwidth Add	Bandwidth dot
			read	write		
L1	2	256 KiB	384 (1216 GiB/sec)		1181 GiB/sec	668 GiB/sec
			256	128		
L2	16	2 MiB	256 (811 GiB/s)		508 GiB/sec	313 GiB/sec
L3	30+	20 MiB	256 (811 GiB/s)		289 GiB/sec	230 GiB/sec
RAM	100	<750 GiB	47.7 GiB/s		39 GiB/sec	43 GiB/sec

Table 1 – Aggregated performance of the Intel E5-2687W for two examples: Add and Dot product

The RAM bandwidth of 39GiB/s is approximately 80% of the theoretical one. The bandwidth of memory depends mostly on two factors: the memory frequency and number of memory channels. The maximum bandwidth was achieved in both cases (add and dot) by 12 virtual GHz. That is nearly twice of the bus memory frequency (4 x 1600 MHz) and less than half of the maximum CPU frequency: Technological trends have produced a large gap between CPU speeds and RAM speeds. This trend will most likely continue. One other important fact is that the memory bandwidth cannot be fully exploited with one or two cores. But there is some progress in compare to the older multi-core processors; the performance of one core of older processors was worse relative to the memory bandwidth.

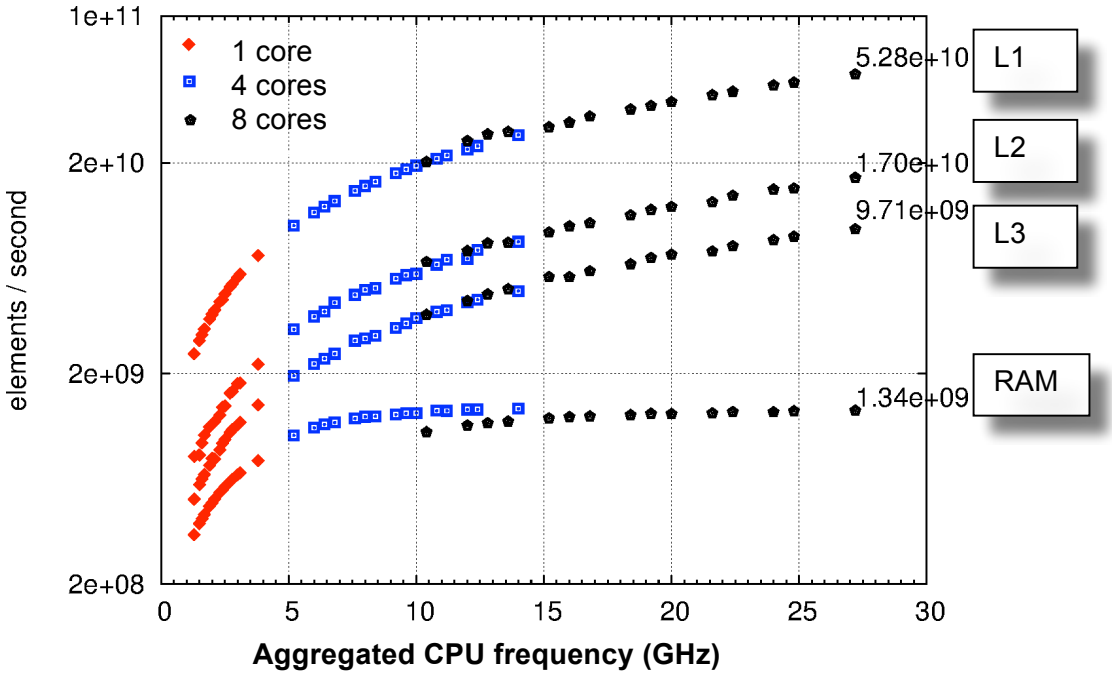


Figure 1 - Performance of a[i]=b[i]+c[i]; data is in L1, L2, L3 and RAM; 1, 4 and 8 cores;

In contrast to the memory bandwidth, the cache bandwidth is directly related to the frequency of the CPU and increases in a linear manner, because L1 and L2 caches run at the same core clock speed. In addition, the bandwidth of L3 is almost three times smaller than the theoretical one. This is explained by the fact that the newest Intel processors provide the rings in L3 cache. Each core is connected to the local segment of the L3 cache. The adjacent segments are connected via different busses called rings. If the data for the calculation is stored in the owned segment of L3, it can be immediately delivered to the core. Otherwise the required data will hop through the ring until it reaches the core. The data in L3 cache is cyclically distributed across all segments. Each core transports 32-bytes of data per cycle and per direction (this will be increased in Haswell to 64 bytes (4)). The benefit of the rings is that the number of cores can be more easily increased as in older processor models. As shown in Table 1, the bandwidth of the operation "dot" is lower than that of the addition. Although the CPU does not store the data by calculation of the dot product, its bandwidth is lower than that of the add operation. This is explained by the fact that there is a dependency in the calculation, resulting in additional waiting times. The waiting times are hidden if the data come from memory and will be prefetched. By using of multiple accumulation registers the performance of the dot product can be further increased. Unfortunately it is not so simple to express this via the intrinsic functions. The alternative version of “dot product” with the compiler optimizations[1] has approximately 20% (L1), 10% (L2) and 5%(L3) higher performance. On other hand the alternative version of "add" with the same compiler optimizations has less performance than the implementation with the intrinsic functions.

It is unrealistic to derive a general formula for the performance of each kind of the operation. It is necessary to consider not only the bandwidth, frequency, latency the size of the pipeline and a lot of other features, but also the dependencies within the computation and compiler features. This is a complex problem that can be only solved with help of the complex hardware simulation.

[1] Intel® C++ Compiler XE 2013; Optimizations flags: -O3 -axAVX -openmp -restrict -ansi-alias –unroll-aggressive

3.2.2 Future developing

Unfortunately, the performance of the CPU cores cannot be enlarged by the increasing of the frequency due to the power constraints: by the frequency increasing the electric power increases disproportionately (6). This is clearly visible on Figure 2. On this figure is shown, the approximated processor's electric power by the intensive calculation (a[i]=b[i]+c[i]) depending on the frequency and number of cores. The "work area" of E5-2687W is marked by thick lines: If eight cores are active the "work area" begins with 8x1.2 GHz = 9.6 GHz and ends with 8x3.4GHz=27.2 GHz.

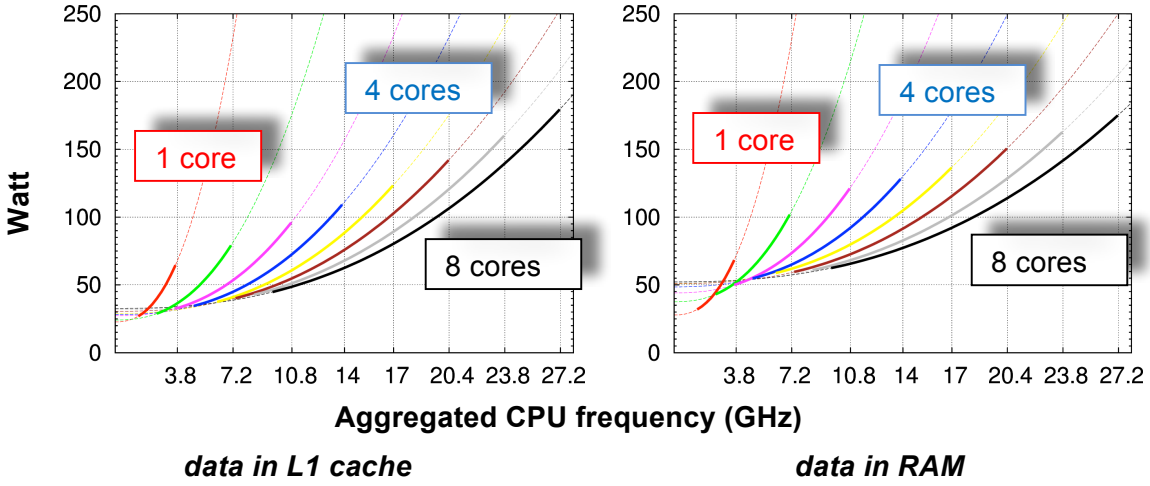


Figure 2 - Electric power of E5-2687W and RAM (4x4 GiB) by computation on 1- 8 cores

The solution of the problem, that the frequency cannot be increased, is to increase the numbers of cores. In this case, not only the performance of the CPU as well as the cache size will be increased. The newest processors with 12 or even 16 cores on a chip are already appearing (e.g. Intel Ivy Bridge and Haswell). Unfortunately, the memory bandwidth is not so "easy" to double. The memory bus frequency can be increased up to 50% (DDR4 SDRAM), but increasing the memory channels requires additional layers on the motherboard. Whether this will be done or not is currently not clear.

3.2.3 Sparse-matrix vector multiplication and Bandwidth cap

As discussed above the electric power increases disproportionately by increasing of the CPU frequency. On other hand the memory bandwidth is the one of the main limiting factor for the performance even if the CPU frequency is stark increased. The dependency of the performance and electric power on the aggregated CPU frequency is shown in Figure 3. As already mentioned above, the aggregated CPU frequency is simply summation of individual core frequency. If the lowest possible frequency is assigned to the CPU (8 x = 1.2GHz 9.6GHz), the bandwidth of the calculation is less than 50 GiB/sec. The speed of calculation is increased by the increasing of the frequency (blue line). However, the electrical power of the CPU and memory increases (red line) faster than the speed of calculation. The electrical power of the whole workstation (black line) increased a little slower. This is explained by the fact that the efficiency of the power supply depends on the load.

In addition, beginning from 21.6 GHz (8 x 2.7 GHz) there is no increasing in the performance. But the power consumption increases rapidly.

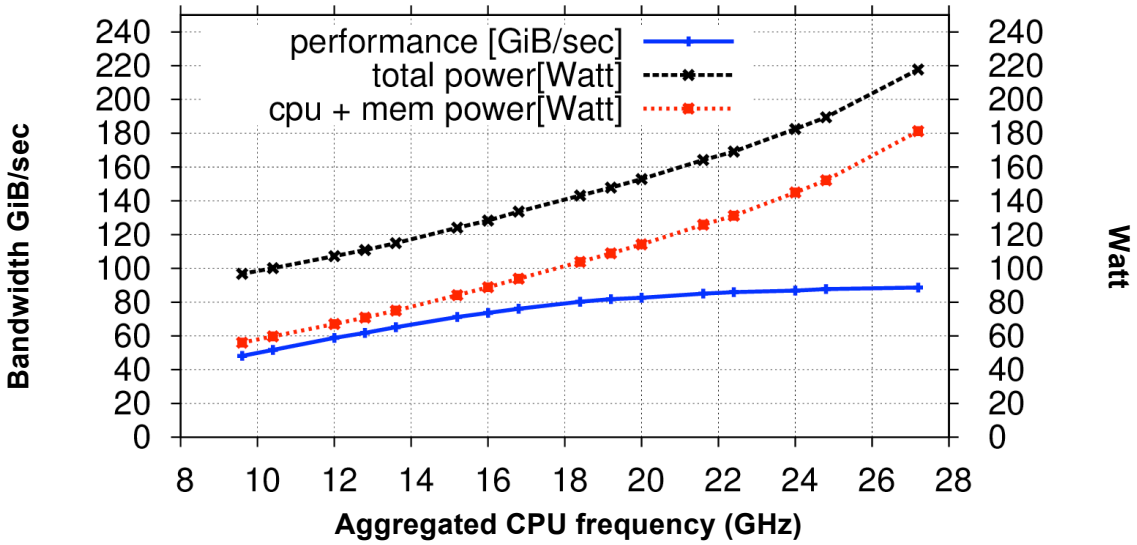


Figure 3 - Dependencies of Power, Performance and CPU Frequency on Intel E5 2687W; Profile of CG for sparse matrix of size (~ 10 GiB data)

One can expect that such cases will be automatically detected in future processors and the frequency will be automatically reduced. However a great deal of work still needs to be done. The switching frequency of the CPU still takes far too long.

3.2.4 Sparse-matrix vector multiplication (SpMVM) on NEC Nehalem cluster

The computation was done on NEC Nehalem cluster with Sandy Bridge E5-2670 @ 2.60GHz processors and Infiniband QDR node-node interconnector. Each node has two processors and memory size of 32 GiB (see section A.1.1. for more details).

The performance of the matrix vector multiplication is shown on the Figure 4. The SpMVM was launched on 1, 2, 4, 8,…, 64 nodes. Each node has 16 cores. On each core one mpi process was started. The connecting lines between the measurement points are only guides for the eye. The aggregated performance increases with the number of used cores. If 256 and more processes are used the data fits in the cache. This is the reason while the performance on each core increases from 12.5 GiB/sec to 22.5 GiB/sec as the reader can see on the right diagram of the Figure 4.

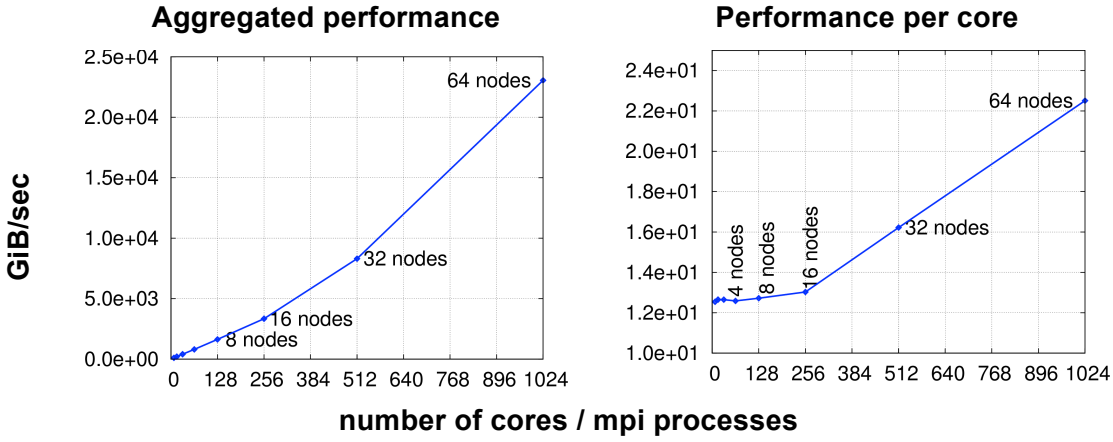


Figure 4 - Aggregated and per core performance of matrix vector multiplication on NEC Nehalem cluster

Figure 5 shows two diagrams. On the left the time of the SpMVM for each core for the launch with different numbers of processes is shown. The time axis is logarithmic. On the right the time of an MPI_Allreduce operation is shown. The computational time of these two operations changes inversely. If the calculation occurs on many cores the time of MV multiplication decreases, but the time of MPI_Allreduce increases. The load imbalance of the MPI_Allreduce is clearly visible on the diagram: The measured time on the mpi process with rank 0 was always less than on the process with the last mpi rank.

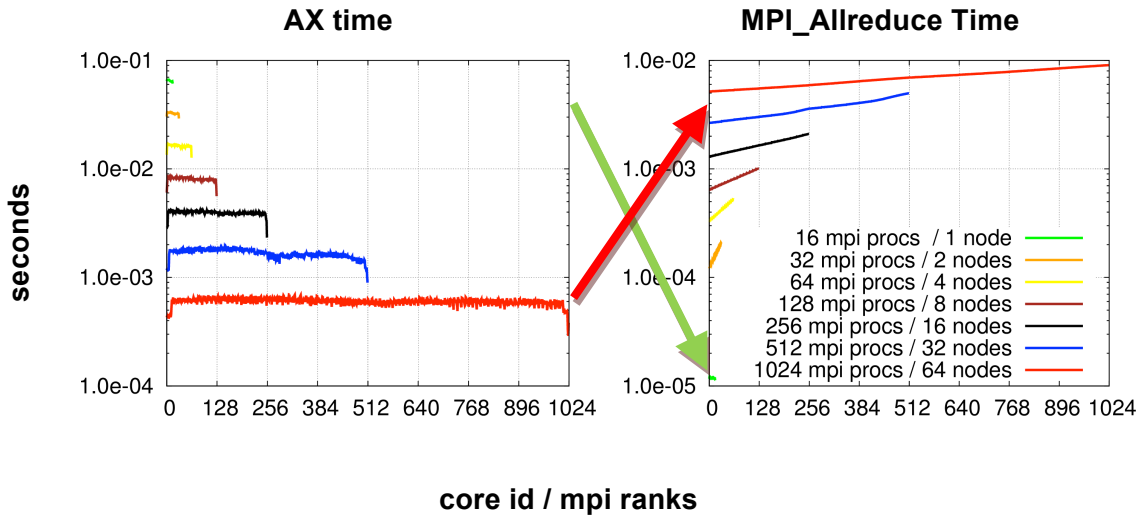


Figure 5 – Calculation and communication time of the matrix vector multiplication and MPI_Allreduce command per core / mpi process by calculation on 16, 32, ..., 1024 cores

The result of poor scaling of the MPI_Allreduce operation is that the improvement in performance when adding additional nodes tails off quickly. This can be seen in the Figure 6.

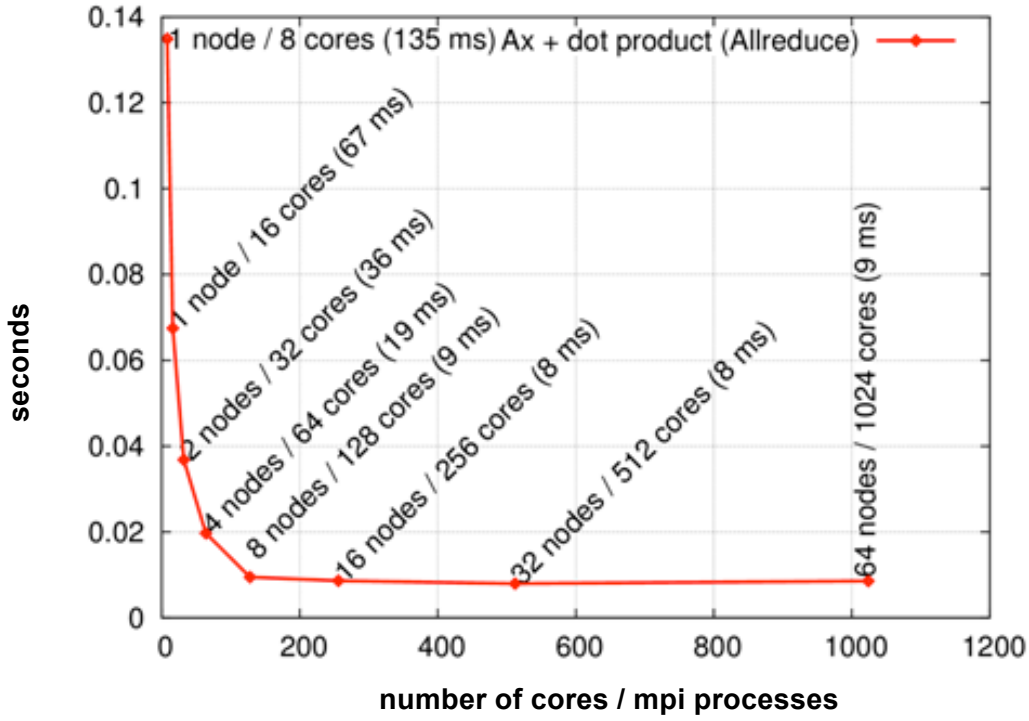


Figure 6- Time of one iteration of CG for different numbers of core / mpi processes

Often this is not considered by the user, which leads to additional unnecessary power consumption.

3.3 Hardware accelerators

To test the limiting factors of hardware accelerators, a Conjugate Gradient (CG) algorithm has been chosen with both a NEC Nehalem Cluster and a Cray XE6.

Our results are based on the GPU K20X (Kepler Architecture) and on both Intel Xeon E5-2670 (Sandy Bridge) and AMD Opteron 6276 (Interlagos) processors.

The table **Error! Reference source not found.** shows a series of properties of the compute node on the NEC Nehalem Cluster, on the Cray XE6 and of the NVIDIA K20X can be found (see section A.1. for more details).

Categories	*2 x Intel E5 2670*	*2 x AMD Opteron 6276*	*K20X*
Peak Perf. (DP)	332.8 GFLOPS	332.8 GFLOPS	1.31 TFLOPS
Peak Perf (SP)	665.6 GFLOPS	665.6 GFLOPS	3.95 TFLOPS
# of Cores	32 (2 x 8 HT)	32 (2x16)	2688 (4x192)
RAM size	32GB	32GB	6GB (GDDR5)
Memory BW	102.4 GB/S	102.4 GB/S	250 GB / S

Table 2 – NEC Nehalem cluster / Cray XE6 nodes and K20X Properties

3.3.1 The Benchmark

From the software side, a Conjugate Gradient (CG) algorithm has been chosen. CG is an iterative algorithm for solving systems of linear equations and is widely used to solve sparse linear systems. The other powerful Krylov subspace methods are very similar in the programming techniques and computational effort.

The heart of this algorithm is the Sparse Matrix-Vector Multiplication (SpMVM), which is one of the most important kernels in scientific computing. In SpMVM many problems arise starting from storage format and ending with data localities. Indirect addressing causes another problem which makes SpMVM a memory bound problem.

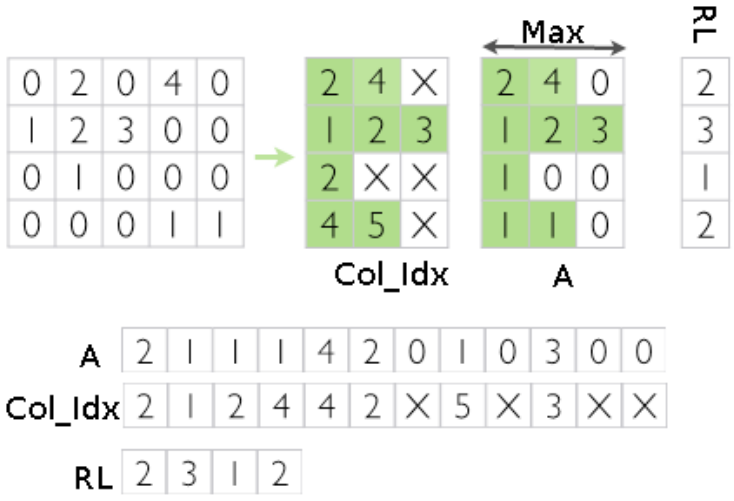


Figure 7 - ELLPACK-R Storage Format

Based on a previous study on SpMVM, an aligned-ELLPACKR storage format has been chosen, as this was shown to be a suitable format on GPGPU accelerators (7). Figure 7 depicts the ELLPACK-R storage format. In this storage format, all non-zero elements are shifted left and another matrix is generated to include all column-indices of all non-zero elements and a vector, which includes row lengths. In both matrices (Col_Idx and A), all the remaining zeros after the maximum row length are truncated

because they hold no important information. Those elements are stored column-wise in memory. The storage needed to store the matrix in this format is:

*Storage = N * (MAX * (Idx_ele_size + Mat_ele_size) + 1);* Where

N is the matrix dimension; *MAX* is the maximum row length; *Idx_ele_size* is the index-matrix element size; *Mat_ele_size* is the matrix element size.

For SpMVM (y = A.x), two more vectors are needed (x and y). It means the required storage becomes

*Storage = N * (MAX * (Idx_ele_size + Mat_ele_size) + 3)*

3.3.2 Performance of sparse matrix vector multiplication in comparison

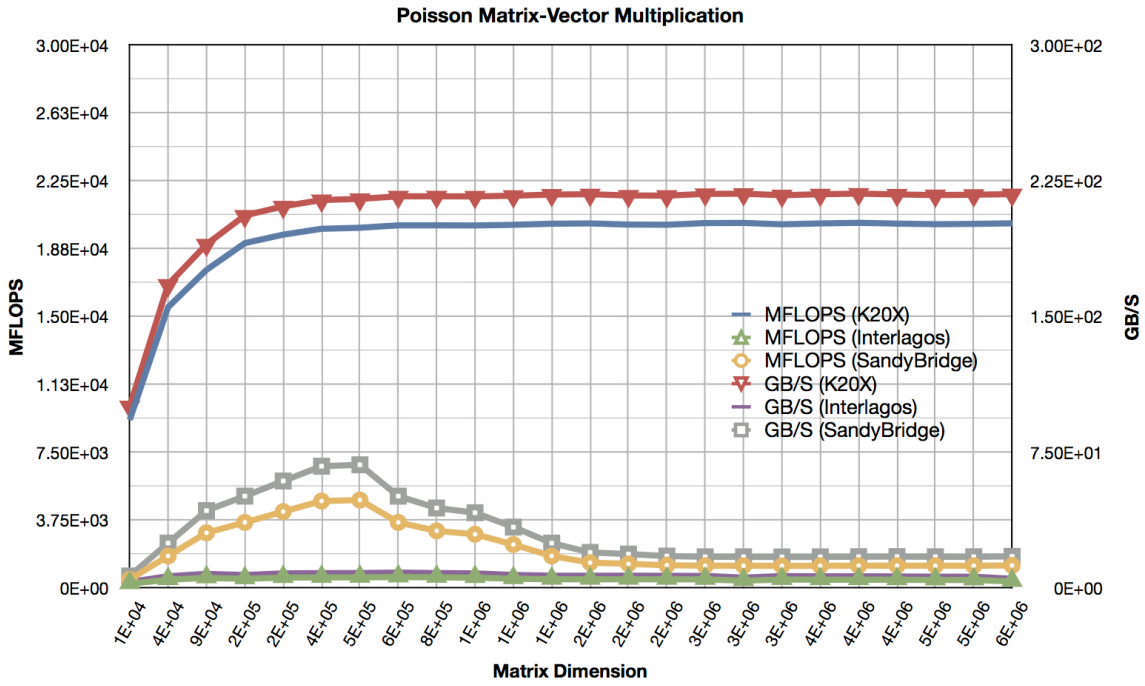


Figure 8 - Sparse Matrix-Vector Multiplication Performance and Memory BW

Figure 8 shows the performance in MFLOPS and the memory BW in GB/s on both CPU and GPU. The GPU results are for the K20X and the CPU results for the Sandy Bride and Interlagos processors. The superiority of GPU over the CPUs is obvious for the SpMVM kernel without data transfer between the GPU and CPU. The GPU is attached to the CPU though PCIex2, which has a transfer BW up to 6GB/S when transferring data to/from pinned memory on the CPU side. Figure 9 depicts the latency of PCIex2 on the NEC Nehalem cluster and the Cray XE6. This test is based on transferring chunks of data in KBytes to and from the GPU. Although the GPU on the NEC Nehalem cluster is not the same as on the Cray XE6, the latency of the PCIe on the Cray XE6 is larger than on the NEC Nehalem cluster. PCIe on the NEC Nehalem cluster requires 64 KBof data to saturate on 6 GB/S but the Cray XE6 needs at least 2 MB to saturate.

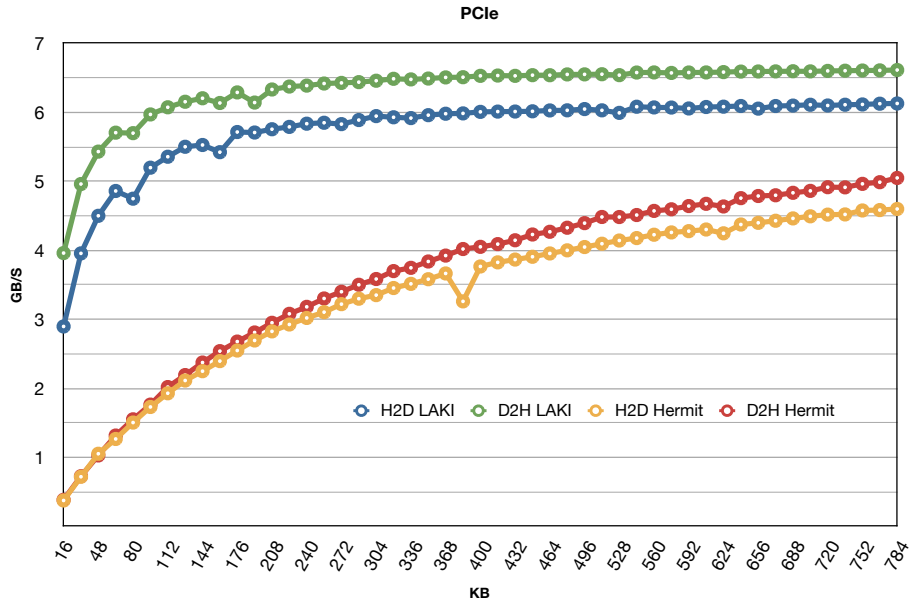


Figure 9 - PCIe x2 bandwidth on NEC Nehalem cluster and Cray XE6

The GPUs offer a very high memory bandwidth, which is larger than the CPU's. In spite of the high performance and memory bandwidth, the GPU is most suitable for applications that are computationally intensive tasks with fine grain parallelism. The GPU has a complex design and many memory architecture features such as on-chip, off-chip, coherent and non-coherent memories. This complexity makes the task for software developers difficult. Moreover, the GPU is attached to the CPU through PCI-Express with 6 GB/S bandwidth and different latency timings as seen in Figure 9. This bandwidth is very low compared to the main memory bus on either the CPU or GPU. Transferring data from and to the GPU often during the same run will degrade the performance dramatically. When the CPU and GPU operate at the same time a very high amount of electrical power is required. This requires very strong power supplies, which will have a low efficiency under low load conditions.

Moreover, GPU kernel startup overhead has been tested on K20X and Figure 10 shows the overhead function with respect to number of thread-blocks.

StartupTime = 4.2 + 0.52 * X

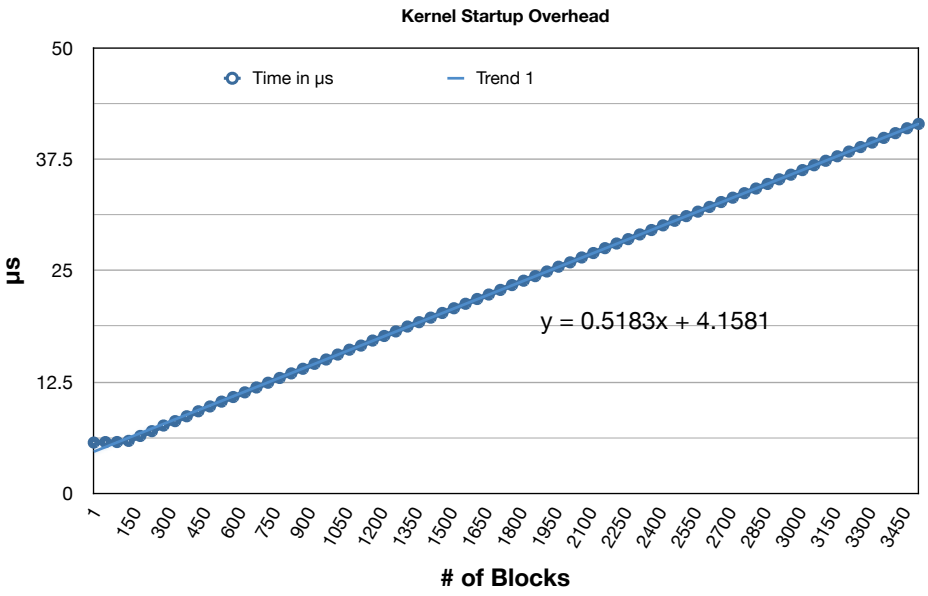


Figure 10 - Kernel Startup Time on NVIDIA Kepler K20X Using CUDA 5.0

It was noticed that the overhead increases 520 nanoseconds each 50 thread-blocks. From one to 150 thread-blocks the overhead is constant and it is around 5.5 µs.

3.3.3 Performance of Conjugate Gradient in comparison

The implementation of CG for solving Poisson matrix problem using double precision on GPU is based on the implementation of SpMVM and is designed to keep the communication as few as possible. The only thing, which has to be transferred to CPU each iteration is *sigma_new* and the result vector at the end once. Figure 11 depicts the Performance and memory BW, which has been reached on CPU and GPU. The CPU version is parallelized only for SpMVM and all others are serial on one core which could be slow for big systems.

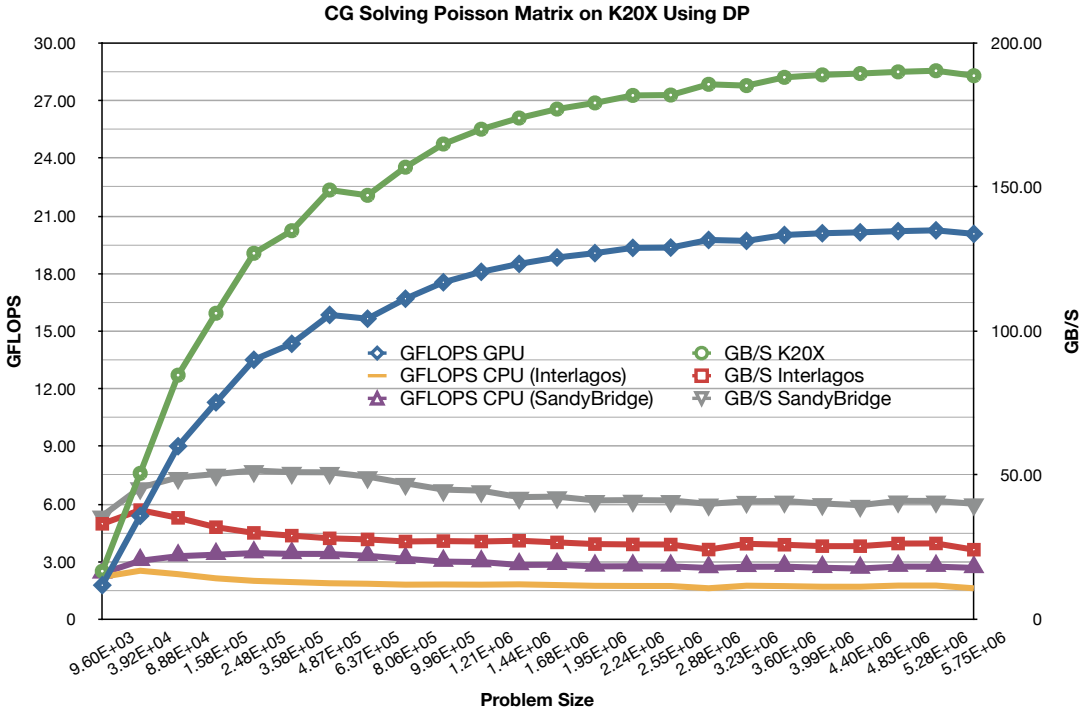


Figure 11 - CG Performance and Memory Bandwidth on Different Platforms

It means that it is recommended to minimize the communication between CPU and GPU and reduce the number of kernel calls. Also the software developers should pay attention to the performance of PCIe, which may differ depending on the vendor and model.

3.4 Network and performance of collective operations

Network performance is key to most application loads on HPC systems. This is particularly true for applications making heavy usage of collective operations. In a very naïve way, networks can be characterizes by two parameters, the network bandwidth (B) and the latency (L). In this picture the total time required to transfer a single message from one node to another is $T_1 = L + m/B$, where m is the size of the message in appropriate units. Other more complex models, as for instance the logGP model (8), have been used to do quantitative analysis of collective operations (9) and (10). However, for the purpose of highlighting future hardware limitations, it is sufficient to illustrate them using the simpler model above.

In MPI, collective operations are operations in which data is sent or received from many nodes simultaneously. Note, that simultaneously refers to the programmer's perspective only. In reality, networks of any type can only send a limited number of messages at the same time because the number of independent network channels is limited to a few, typically. For the remainder of this section we will assume that only a single channel is available, i.e. only one message can be sent or receive the same time.

The most straightforward implementation of single-rooted collective operations - as for instance *barrier*, *broadcast*, *gather*, *reduce* - is to have the root node of the collective sent and receive messages from every participant one after the other. Then, the time to complete the (single-rooted) collective is proportional to $n*T_1$, where n is the number of participants. The next best implementation is to forward messages along a (binary) tree, which results in a completion time proportional to $\log(n)*T_1$. In fact, this seems to be the basic strategy chosen by the Cray MPI implementation as suggested by our experiments below.

More complex collectives, as for instance *all-to-all*, can be modelled as a superposition of multiple single-rooted collectives (here multiple *gathers*). A straightforward implementation of those would be to do the basic single-rooted collectives one after the other which results in a completion time of $n*\log(n)*T_1$. The issues and conclusions presented below apply equally to these complex collectives, even though we have not yet analysed the runtime behaviour of these in great detail.

In addition the network on HPC systems in practise is always a hierarchical one, each level characterized by its own unique latency and bandwidth (and number of channels). The most obvious level transition is from communication on a node (through i.e. shared memory, which here is considered a part of the communication network) to communication across different nodes. In the former case, latency is theoretically in the range of nanoseconds, across nodes it is in the range of microseconds. The bandwidth does not wary as much and is in the range of 100 Mb/s.

In the past we have seen steep increases in the available network bandwidth, but only relatively little improvement in the network latency. The latency is physically limited by the speed of light. For a distance of 10 metres this translates into a ping-pong latency of roughly 0.1 microseconds. There is no such limit for the bandwidth - we might go on adding channels to increase it.

Using the Collectives Microbenchmark Suite(11) we have measured the total completion time for the MPI collective operation *all_reduce*, which typically is the one consuming most resources in the set of CRESTA applications(11). Results taken on Hermit (Cray XE6) are shown in Figure 12.

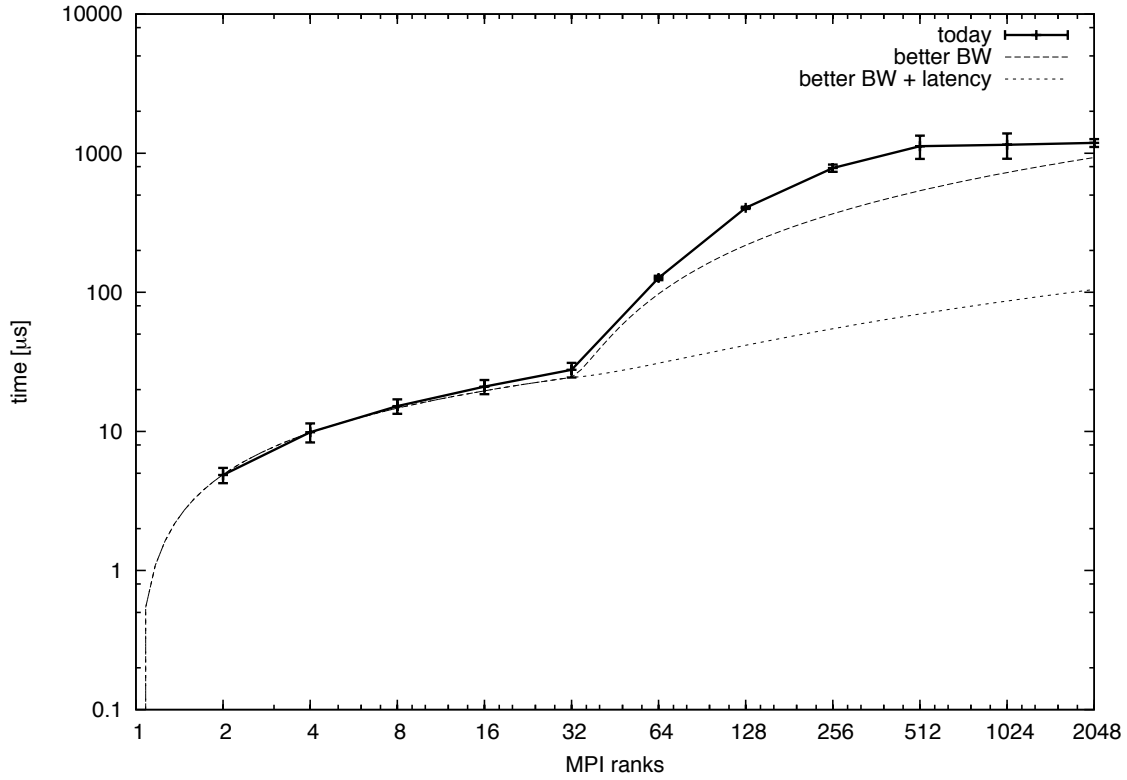


Figure 12 - Experimental measurements of the total completion time of the MPI collective all_reduce with a payload size of 512 floats and varying number of participants taken on Hermit (heavy line). The lighter curves are models with better bandwidth, and a second model with better bandwidth and latency, respectively.

The sudden increase of total completion time at 32 MPI ranks is due to the much larger latency across nodes compared to the latency on the node (Hermit has 32 cores per node). We have fitted this data with a binary tree model as discussed above. In order to estimate the impact of future developments we have dialled up bandwidth and/or latency (see model curves in the figure). Increasing the network bandwidth, as is likely to happen in the future, will results in only modest performance improvements, particularly at large number of communication participants. On the other side, bringing down the latency to its physical limit, will improve the performance by an order of magnitude for the case studied.

This conclusion in principle holds for most HPC workloads, where the amount of data transferred collective operations is relatively small. The expected increase in network bandwidth (and number of communications participants) will only strengthen the trend and push collective operations into a mode where it is latency-dominated.

Part of the communication latency is not caused by the network hardware, but rather by the various software layers, as MPI library or high-level network drivers. These software latencies are particularly severe if the software layers need to be traverse to do only relatively little computation as reducing a vector to a scalar. Hardware capabilities for reduction operations would allow to effectively by-pass the software stack thus potentially improving latency significantly.

Another approach is to overlap communication with calculation. This is particularly promising in situations where load imbalance causes some nodes to arrive at collective communication points earlier than others. Currently the collective communication tree is traversed in a specific order and late arrivals will delay the whole process. A more suitable out-of-order traversal of the tree, as is subject to investigation in this project, could potentially allow to mitigate the late-arrival problem, as caused for instance by load-imbalance.

3.5 IO System

The IO System is as important as the other components of an HPC system. The temporal data as well as the results of the simulation have to be maintained for future analysis and use. The Lustre system is often used as a parallel distributed file system and dominates the TOP 500. An example of Lustre integration is shown in Figure 13. The files with the data will be stripped over the Object Storage Server (OSS). Thus, several independent streams can be read by the processes simultaneously. The bandwidth is generally limited by network connection. The most costly operation is often the “open” and “close” of the files. The bottleneck is the Metadata Server (for example see (12)). When one opens a file, a lot of information is retrieved from a database (e.g which OSS contains the file, how big is the stride and so on). Additionally some locking must also be done in the file system. If several thousands of processes are trying to open one or more files at the same time, the waiting time may be several hours. To escape it one uses a nested approach: only a few processes read the data from the IO system and send it to the rest of the processes (for example see (13)).

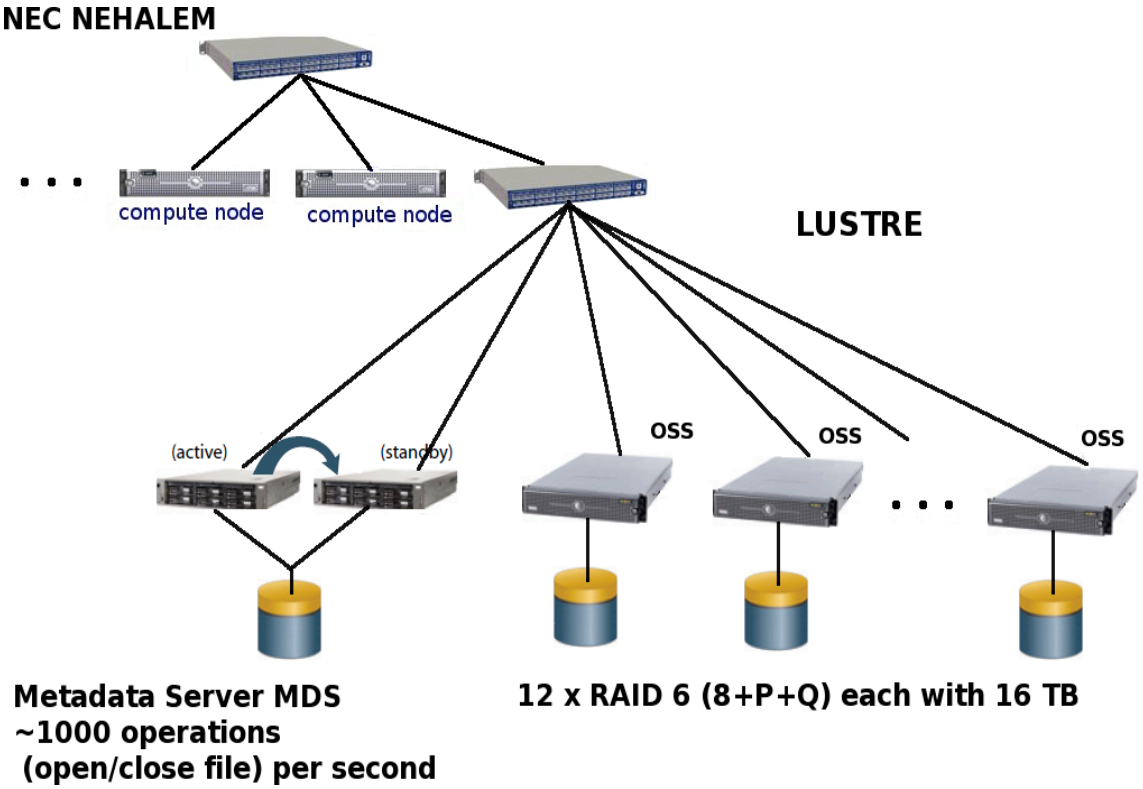


Figure 13 - Distributed file system Lustre on NEC Nehalem

There are some new ideas how to improve the performance of the file storage system (e.g. see http://www.eiow.org/).

As mentioned above we use the IO server rather to debug and test purposes. Therefore, we will not include further details about IO Servers to this document.

4 Fault Tolerance in MPI

4.1 State Of the Art

By raising computational performance through increased parallelism, single component failures in modern supercomputers have become a more important and more expensive issue. This is due to the fact that with core count the number of overall components within a system rises. Thus, as the mean time to failure (MTTF) of every single component does not grow as fast as the number of components in a supercomputer, the overall MTTF of the system shrinks. A study (14) including 22 system at LANL over a time span of 9 years has shown, that the average MTTF of some current systems is less than 8 hours.

A manifold of current applications oppose untreated errors with checkpoint/restart techniques. Regarding the constantly growing parallelism, these techniques continuously turn deprecated as the implicated overhead is growing to a significant portion of the run time with increasing core count. Therefore, in exascale applications the error treatment will become stringently dynamic.

Fault tolerant design has permeated the majority of computer hard and software construction including CPU exception handling, RAID, adaptive routing, and others. This design choice is virtually absent in the predominant standard for parallel communication. The default error behaviour in MPI is to treat it as fatal, which in turn causes the parallel program to abort at any reported failure. The only other predefined error handler returns after an error without any information concerning the immediate fault.

4.2 Proposition

Making efficient use of future hardware implies optimizing some metric to measure computational costs. In the following restart and dynamic fault tolerant techniques are compared with respect to expected CPU time of an exemplary exascale application.

Let T be the overall run time for an application to complete and τ the primary core death, i.e. there exists a probability distribution $\Pr(\tau \leq t) = \int_0^t f(t)\,dt$ with probability density function $f(t)$ describing core death time, with mean value and deviation dependent on the MTTF. For the restart case the CPU time function dependent on τ can be described as

$$c(\tau) = \begin{cases} 0, & \tau < 0 \\ T + \tau, & 0 \leq \tau \leq T \\ T, & \tau > T \end{cases}$$

For the sake of simplicity we assume the second run to be successful in any case. Then, the expected value of the CPU time for an application of size n dependent on core death is:

$$E\big(c(\tau)\big) = \int_0^T nt\, f(t)\,dt + \int_0^\infty nT\, f(t)\,dt$$

In the latter equation the second part of the right hand side T is constant and the rest is equal to one by definition.

For the fault tolerant case it is assumed that the number of spare cores is sufficient for successful application termination. Hence, the expected value of the core death time dependent CPU time is:

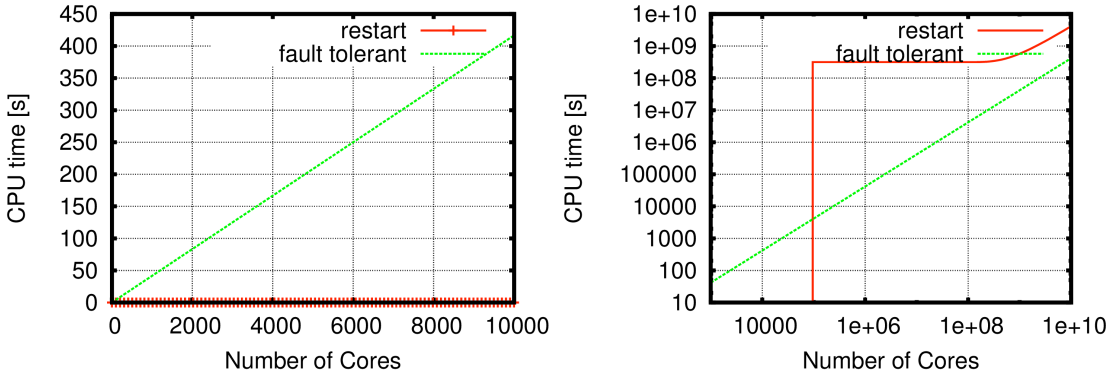
$$E\big(c(\tau)\big) = (1+\varepsilon) n\, T$$

For the sake of simplicity the power savings of an idle core is neglected, thus the expression above is independent of τ.

Subtracting the minimal application run time from the two above statements results in the overheads

$$n \int_0^T t\, f(t)\, dt \qquad \text{and} \qquad \varepsilon\, n\, T$$

In order to illustrate the dependency of the left hand term on n some example assumptions are made. The MTTF of every core is set to ten years, $f(t)$ is a normal distribution with standard deviation of 1, the runtime of our example code is 1 hour and one extra core is allocated in the fault tolerant case for every CPU day.



The figures above show the expected additional CPU seconds for the two methods. For small core numbers the restart technique outperforms the constant number of additional cores, but for large core count the restart technique needs orders of magnitude more CPU seconds. For very large core counts the slopes of the two methods converge, but the absolute values are still separated by more than an order of magnitude.

4.3 Application to Exascale

This shows, that for increasing core count the energy-efficient choice is to incorporate spare cores, which will be used in case of a core failure. Especially in relation to exascale computing we have to revaluate the priorities and take allegedly idle cores in an application into account. For a machine running exascale applications this will be a crucial design decision with regard to improving overall utilization.

5 Performance prediction models

The following descriptions of the numerical algorithms will help us to understand the hardware limitations that will inhibit scalability of them.

5.1 Performance prediction model for FFTs

The parallel performance of the FFT algorithm is best understood by considering the data movement graph for the algorithm. For the FFT this is the “butterfly” pattern (see Figure 14). As can clearly be seen from the graph representation a 2^n FFT has a computational complexity of O(N log(N)). The algorithm has a potential parallelism of O(N) with good load-balance but is also a non-local algorithm requiring a high degree of data movement. The computation performed by each node of the graph is quite small so the time to execute these communication steps will dominate performance at large numbers of nodes.

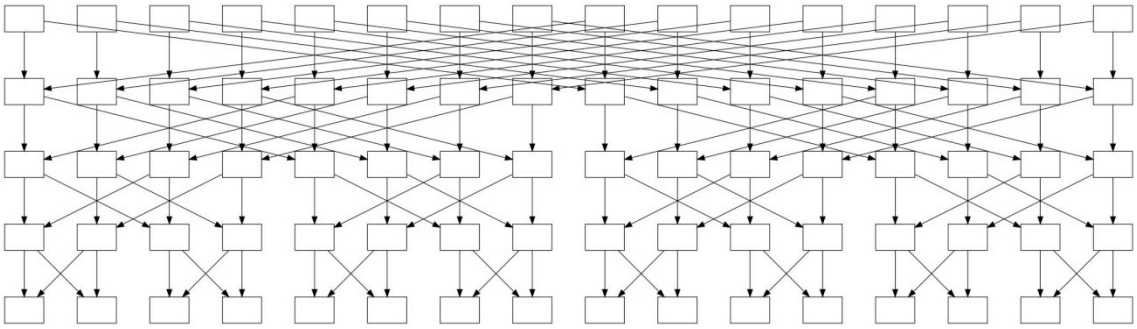


Figure 14 - Data movement graph for a 2^4 FFT

The most common use of Fourier transforms in HPC applications are as multi-dimensional FFTs.

If we consider a d dimensional FFT with sides N_i the total number of data-points in the FFT is given by $N = \prod_{i=1}^{d} N_i$ and the computational time on P processors is given by:

$$C = f\ \frac{N \log(N)}{P}$$

Multi-dimensional FFTs are usually implemented by performing each dimension of the FFT in turn. This gives rise to the same graph representation as a single large FFT consisting of the same number of points, the only difference between the two algorithms being the phase factors applied at each computational stage. In a multi-dimensional FFT there are no constraints on the order that each dimension is processed. In fact, stages of the FFT algorithm from different dimensions can be interleaved, provided the order within a dimension is preserved. However this only changes the assignment of initial data points to the initial graph nodes. The topological structure of the graph always remains equivalent to a graph for a single FFT of the same size.

A parallel implementation strategy for any FFT calculation therefore depends on choosing a set of data decompositions for each stage of the FFT so as to minimize the amount of data that needs to cross node boundaries. In general, an initial communication stage is required to place the data in the correct starting decomposition but in some cases it is possible to adapt the surrounding application to use this same decomposition.

Parallel multi-dimensional FFTs usually avoid this initial communication phase by starting from data decompositions where at least one of the dimensions are local to a node. Node-local FFT implementations are then applied to each dimension in turn interspersed by communication phases where necessary to change the data decomposition so that the FFTs in the next dimension become local(see Figure 15).

This also has the practical advantage that existing single node FFT libraries can be used to implement the computational phases.

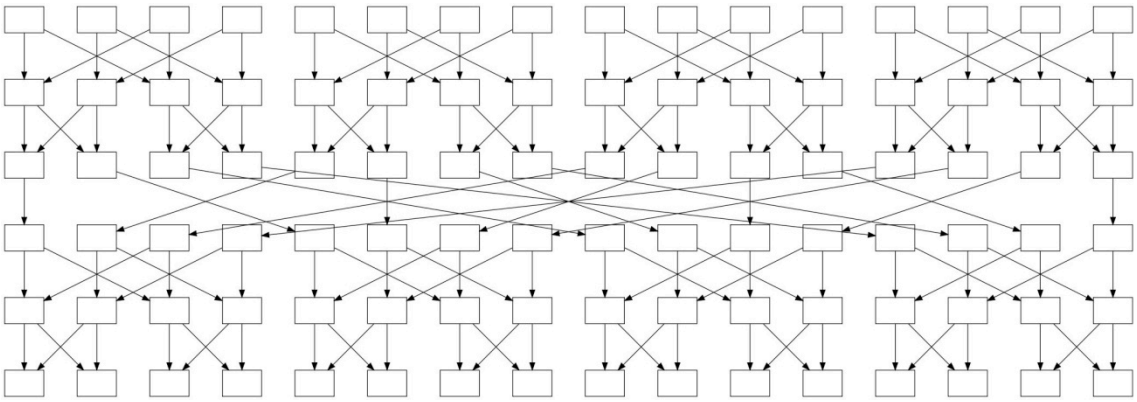


Figure 15 - Graph representation of a (2^2x 2^2) 2D FFT

Where the data can be equally distributed across processors this communication stage is equivalent to a MPI **MPI_Alltoall** collective operation. If the data cannot be equally distributed this corresponds to an **MPI_Alltoallv** operation.

This requires one fewer communication stage than the number of dimensions in the FFT. At low numbers of nodes this can be reduced by having decompositions where more one of the dimensions are local at the same time.

In principle it is also possible to decompose the global data graph in a way that the transformations over some of the dimensions are spread over more than one data decomposition. This gives greater freedom of choice in the number of processors that can be used. Unfortunately it is unlikely that the initial data redistribution phase can be avoided in this case, so this approach will only be appropriate for high dimensional transformations or applications where other constraints on the data decomposition already require an initial communication stage to be used.

5.1.1 Hardware model

We can model the communication network interface using a modified Latency/Bandwidth model where the message latency may depend on the number of active messages so the time to send x messages of size S is given by:

$$D_x = l(x) + b.x.S$$

We can expand the latency factor as a power series in x:

$$l(x) \equiv l_0 + l_1.x + l_2.x^2 \cdots$$

- l_0 can be interpreted as a pipelined latency including the network transfer time.
- l_1 can be interpreted as non-pipelined latency or per-message overhead; for example representing critical sections in the MPI library or the additional time needed to send message headers.
- l_2 can be interpreted as a cost associated with searching internal message queues (each message incurs a cost proportional to the number of outstanding messages).

The impact of the number of active messages is included in the model to capture the difference between sending a small number of large messages and sending a large number of small messages.

In practice it may be necessary to fit these parameters separately for different communication mechanisms (e.g. within and between nodes) and possibly for different message size regimes corresponding to different underlying communication protocols.

We also need to take account of contention with the network. This is a complex problem that can really only be addressed via simulation but a simple model can be

developed using an estimate of the bisection bandwidth B(x) available between a group of x processors.

For a single communication stage of the FFT calculation implemented as an all-to-all over P nodes this gives:

$$T = l(P-1) + b.\frac{(P-1)}{P}.\frac{N}{P}$$

The total communication time is then given by:

$$T_c = n_{step}T$$

If we include an estimate of the impact of network contention:

$$T_c = n_{step}\max(F,T)$$

where

$$F = f\frac{N}{B(P)}\frac{P-1}{P}$$

5.1.2 Latency minimisation

When using multiple communication steps, the number of simultaneous messages can be reduced by choosing decompositions that minimise the number of nodes that need to communicate at each step.

One way to achieve this is in multi-dimensional FFTs is to perform the decomposition dimension by dimension onto a (d-1) dimensional process grid then the data passes through the following decompositions:

Global volume	**Processor grid**	**Local volume**
$N_1, N_2 \ldots N_d$	$1, p_1, \ldots p_{d-1}$	$N_1, N_2/p_1, \ldots N_d/p_{d-1}$
$N_1, N_2 \ldots N_d$	$p_1, 1, \ldots p_{d-1}$	$N_1/p_1, N_2, \ldots N_d/p_{d-1}$
$N_1, N_2 \ldots N_d$	$p_1, p_2, 1, \ldots p_{d-1}$	$N_1/p_1, N_2/p2, \ldots N_d/p_{d-1}$
Etc.		

This breaks each communication step into disjoint sub-sets of nodes (of size p_i), reducing the number of messages each node has to handle at each step.

For a single communication stage of the FFT calculation this gives:

$$T_i = l(p_i-1) + b.\frac{(p_i-1)}{p_i}.\frac{N}{P}$$

With the total communication time given by:

$$T_c = \sum T_i$$

If we include an estimate of the impact of network contention:

$$T_c = \sum \max(T_i, F_i)$$

where

$$F_i = f\frac{N}{B(p_i)}\frac{p_i-1}{p_i}$$

5.1.3 Communication overlap

There is some room for communication/calculation overlap in the FFT calculation, but only when the potential parallelism of a decomposition strategy is not fully exploited. If multiple independent sub-graphs are allocated to the same processor, then the results from one sub-graph can be communicated in parallel with the evaluation of subsequent sub-graphs, but at the cost of increasing the number messages. Without any overlap the time to solution is given by:

$$T_{solution} = C + T_c$$

With overlap

$$T_{solution} > \max(C, T_c)$$

so this is only significant if the computation and communication costs are roughly equal.

5.1.4 Limiting factors for Exascale

There are a number of factors that will limit the performance of FFT calculations at the exascale. The primary challenge is to support the very high levels of concurrency in exascale systems.

Due to power constraints, exascale systems are expected to be built out of GHz processors with a total system concurrency of 10^9. This means that for a single FFT to fully utilise this concurrency it will need to be of a similar size e.g. at least a 1000^3. This is a relatively large system for many current uses of FFTs.

Current proposals for exascale systems are scaling up communication capabilities to a lesser degree than floating point performance. The inevitable consequence of these designs is that the fraction of application run-time spent in communication bound operations like FFTs will be expected to increase. Even if the full computational concurrency of the system is not utilised, an FFT may still provide better time to solution and energy usage than alternatives. On current petascale systems FFT performance is primarily determined by number of nodes rather than number of processors. It could easily be argued that on exascale systems, where power consumption is a major part of the overall cost, the correct design choice for FFTs is to achieve the minimum time to solution that the network allows, and it is a positive feature if this allows unused cores on to enter power saving sleep states. This would be particularly advantageous if the hardware was capable of re-assigning the saved power to increase the speed of the communication sub-system.

The main lack of scaling in the algorithm comes from the need to increase the number of communication stages as the number of nodes increases. Even in the worst case (equivalent to directly implementing the butterfly pattern) the number of communication stages only increases as log(P). This is not totally unacceptable scaling behaviour, as a 1000-fold increase in node count would still result in a factor of 100-performance increase. However as energy costs associated with communication will also increase proportional to the number of stages, this will result in an approximate 10-fold increase in the energy consumption for the same calculation. For this reason the node counts used to implement the FFT will probably have to be limited to values where only a small number of communication stages are used.

A cubic 3D FFT can be fully decomposed in 2 dimensions using the “pencil” decomposition while still only requiring 2 communication steps and exposing parallelism of: $P = N^{\frac{2}{3}}$. Parallelism of $P = \sqrt{N}$ can be achieved with 2 communication steps for any FFT by using an initial data redistribution step and decomposing the data graph as if it was a 2D transform.

Network contention is also a likely bottleneck at high processor counts. The network contention factor F in our model is inversely proportional to the bisection bandwidth of the network. For this factor to scale, the bisection bandwidth needs to increase linearly with number of nodes. This is not true for most commonly used network topologies, so

network contention is likely to become a limiting factor at some point. There will be no advantage to increasing the number of nodes beyond the point where network bisection bandwidth becomes the limiting factor on performance. If this crossover point occurs for $P < \sqrt{N}$, then the best available solution might be to use a two-stage strategy where the FFT calculation only uses a sub-set of the available nodes. The performance of the FFT will be ultimately determined by the bisection bandwidth of the network but the use of an initial data redistribution step means that the data decomposition used by the rest of the application can be chosen independently of the requirements of the FFT.

For a fixed problem size the network interface bandwidth costs for a single communication scale with the number of nodes.

The number of simultaneous messages in each communication step increases as the number of nodes increases so non-constant terms in the latency factor are also expected to become significant at exascale.

5.2 Iterative solver for sparse linear systems

One iteration of a typical iterative solver consists of sparse-matrix vector multiplication and some vector operations. The sparse-matrix vector multiplication is one of the most important computational kernels in the numerical simulation. There are many different formats for the storage of the matrix. However, the greatest improvement in the performance on a CPU can be achieved by a block oriented storage format. In the next section we describe the format of the matrix. So it is possible to determine how much memory will be used by different numbers of computational nodes, matrix sizes and matrix block sizes. This information is also useful to calculate the effort for transferring of the data to the external library or between different simulation modules (13). In the last section we describe the effort to solve numerical problem (large sparse linear system).

5.2.1 Blocked sparse matrix overview

Sparse matrices are characterised by a huge amount of vanishing elements and it is obviously desirable to store and access only the nonzero elements. One of the most popular formats to save and transfer sparse matrices is CSR format. For each sparsely populated row a part of the `col` array lists the column indices of the matrix entries. The matrix entries are stored in the `val` array in ascending column index order. Evidently, both arrays cover an equal number of matrix entries. Last but not least the row array is a list of indices pointing to each new row in the column index list.

Additionally, every matrix entry can consist of not only one element but of a dense block of values. In this case, the row and column indices remain unchanged as do the corresponding arrays. However, the value array enlarges by a factor of block size squared.

Main characteristic constants are listed in the following Table 3.

n_d	Matrix dimension
n_e	Number of nonzero matrix entries
n_b	Block size
$s_d = \frac{n_e}{n_d^2}$	Density / sparse factor

Table 3 - Sparse matrix characteristic constants

5.2.2 Sparse matrix file and data layout

The following table summarises a typical file data layout for blocked sparse matrices and lists the MPI file access method needed as well as the number of bytes to be read. It assumes a parallel file reading with *p* MPI threads.

Var	Count	Type (Bytes)	Description	Read access method
	20	char(1)	Name	File_read_all
n_d	1	int(4)	Dimension	File_read_all
n_b	1	int(4)	Block size	File_read_all
`row`	$n_d + 1$	int(4)	Row pointer array	File_read_at_all
`val`	$n_e n_b^2$	double(8)	Values	File_read_at_all
`col`	n_e	int(4)	Column pointer array	File_read_at_all
`rhs`	$n_d n_b$	double(8)	Rhs vector	File_read_at_all
`res`	$n_d n_b$	double(8)	Residual	File_read_at_all

Table 4 - File data layout for a typical sparse matrix.

5.2.3 Sparse matrix parallel read chain

In contrast to a dense matrix, the read chain of a sparse matrix serialises inherently due to the inner dependencies of the sparse matrix data parts. The header including the matrix dimension needs to be known to read the proper amount of the row pointer array. Respectively, this applies to the row and column pointer arrays.

Also, the later usage of the data influences the reading strategy. If the matrix data will be used in its native format each part, possibly partitioned further over threads, can be read in entirely. However, the matrix data will commonly be processed in a way or passed to other, even outer, structures that want to arrange the data their way. E.g., this happens when using external libraries like PETSc or Trilinos. In this case a complete matrix read is not necessary or may be impossible because twice the memory is required. Here, a chunk size s_c is specified denominating a number of rows. Each thread reads and passes his chunks individually to the external library.

In total, the reading chain covers the header first, followed by row and col arrays, the values and the vectors. Main header information, namely dimension and block size, must be present on every thread. The row array gets partitioned among all threads each one holding $M_{row} = \left(\frac{n_d}{p} + 1\right) \cdot 4$ bytes. It is either stored in local-type (every part starts with a fresh 0 index) or global-type (consecutively numbered across all parts) indexing. Respectively, the global number of nonzero entries must be collectively added (Allreduce) or broadcasted by the last thread.

The `col` and `val` arrays consist of $\frac{n_d}{p}/s_c$ number of distinct reads where s_c number of rows are processed. The estimated number of entries within each of these rows equals in average the product of the sparse factor s_d and the matrix dimension n_d.This yields

$$\frac{n_d}{p}/s_c \cdot M_{val} \approx (\frac{n_d}{p}/s_c) \cdot s_c \cdot (s_d n_d) \cdot n_b^2 \cdot 8$$

+

$$\frac{n_d}{p}/s_c \cdot M_{col} \approx (\frac{n_d}{p}/s_c) \cdot s_c \cdot (s_d n_d) \cdot 1 \cdot 4$$

bytes.

Both vectors are small enough to spare a further per-thread partitioning and cost 2 collective reads with

$$M_{rhs} = M_{res} = \frac{n_d}{p} n_b \cdot 8$$

bytes read each.

L_N	Network latency for collectives
L_F	File access latency
B_N	Network bandwidth
B_{FI}	File I/O bandwidth for identical data (MPI_File_read_all)
B_{FD}	File I/O bandwidth for disjoint data (MPI_File_read_at_all)

$$T = \sum_{i \in \{head,row,col,val,rhs,res\}} T_i$$

with

$$T_{head} = 2 \cdot L_F + \frac{4+4}{B_{FI}}$$

$$T_{row} = 1 \cdot L_F + \frac{M_{row}}{B_{FD}} + 1 \cdot L_N$$

$$\left.\begin{matrix} T_{col} \\ T_{val} \end{matrix}\right\} = \frac{\frac{n_d}{p}}{s_c} \cdot L_F + \frac{1}{B_{FD}} \cdot \left\{\begin{matrix} M_{col} \\ M_{val} \end{matrix}\right.$$

$$\left.\begin{matrix} T_{rhs} \\ T_{res} \end{matrix}\right\} = 1 \cdot L_F + \frac{M_{rhs}}{B_{FD}}$$

5.2.4 Sparse-matrix vector multiplication and dot product on exascale system

If the matrix is distributed between computational nodes by rows, the communication part can be in most cases overlapped. The local part of the vector has to be sent mostly to the neighbor's processes. The next important operation is the dot product. After the local part of the dot product has been calculated an MPI_Allreduce operation must be done. This operation is usually the bottleneck for iterative solvers. In the next section we compare the performance of these two operations for different system sizes and numbers of processes.

The bone matrix was assembled by static 3D finite element simulations of a bone-implant system (see (13) for more details). The block size of the original matrix was changed from 3x3 to 4x3 via additional zero elements. This is because the processors use the AVX registers. One register contains four values. The new block size allows the compiler or programmer to unroll the main loop of the matrix vector multiplication. This leads to a great improvement in performance. The additional effort is much smaller than the resulting benefits.

The computation was done on the Intel E5-2687W.The results are listed in Table 5 for two cases: when the data fits in RAM and in L3 cache.

Matrix dim. Nd (block 4x3)	Matrix size GiB	Size of streams (block 4x3)		Bandwidth GiB/sec	Time of Ax sec
		Load GiB	Store GiB		
54'537	268.262e-3	269.888e-3	1.625e-03	32.4	8.3e-03
1'704	6.8155e-03	6.866e-03	50.783e-6	88.9	77.32e-06

Table 5 – Performance of the MV multiplication

Let us assume that an exascale system could have about 10 million nodes each with two such processors. The size of the streams increases linearly with the matrix dimension and one node with 16 cores has 32 GiB RAM and 40 MiB cache. If we want to keep the data in cache on all10^7nodes the size of such matrix (n_d) is about $3.4 * 10^{10}$.If we want to keep the data of the same matrix in the memory we need a lot fewer nodes: 8,646.We assume that 50% of the RAM can be used for the matrix vector multiplication.

The efficiently calculation requires that the calculation time should be much greater as the communication time:

$$\frac{T_{comm}(n)}{T_{calc}(n)} \ll 1;\ n - \text{number of nodes}$$

The performance and time of the calculation depends mostly whether the data is in cache or in RAM. The time of matrix vector multiplication for n_d=$3.4 * 10^{10}$ is:

$$\boldsymbol{T_{calc}}(10^7) = 77.32 * 10^{-6}\ sec;\ data\ in\ cache$$

$$\boldsymbol{T_{calc}}(8{,}646) = \frac{16\ GiB\ per\ node}{2 * 32.4\ \text{GiB/sec}} = 0.247\ sec; data\ in\ RAM$$

Assume that the communication time depends logarithmic on the number of nodes(15) with Latency 1µs:

$$T_{comm}(10^7) = Latency * log_2 n \sim 23.25 * 10^{-6} sec; Latency \sim 1 * 10^{-6}$$

$$T_{comm}(8{,}646) = Latency * log_2 n \sim 13.08 * 10^{-6} sec; Latency \sim 1 * 10^{-6}$$

Now we have to check the efficiency of the parallelization:

$$\frac{T_{comm}(10^7)}{T_{calc}(10^7)} = 0.30; data\ in\ Cache$$

$$\frac{T_{comm}(8{,}646)}{T_{calc}(8{,}646)} = 5.30 * 10^{-5}; data\ in\ RAM$$

It is clear that the efficiency of the parallelization is much better if the data is in RAM. During the calculation the electric power of the processors and RAM would be about $180 * 10^6$ Watt if the data is in RAM and about $8{,}646 * 185$ Watt if the data is in cache. The other components of the node consume about 100 Watt. The electric power of the whole node is about 300 Watt during the communication. By these assumptions we can calculate the amount of energy, which we need for one iteration step:

$$Energy_{data\ in\ cache} \cong 77.32 * 10^{-6}\ [sec] * (2 * 185 + 100)\ [Watt] * 10^7 +$$

$$+\ 23.25 * 10^{-6} [sec] * (2 * 100 + 100) [Watt] * 10^7 =$$

$$= \underbrace{3.63 * 10^5}_{\text{Ax}} + \underbrace{6.975 * 10^4}_{\text{MPI Allreduce}} = \mathbf{4.33 * 10^5\ Joule}$$

$$Energy_{data\ in\ RAM} \cong 0.247 [sec] * (2 * 180 + 100) [Watt] * 8{,}646 +$$

$$+ 13.08 * 10^{-6} [sec] * (2 * 100 + 100) [Watt] * 8{,}646 =$$

$$= \underbrace{9.82 * 10^5}_{\text{Ax}} + \underbrace{3.39 * 10^1}_{\text{MPI Allreduce}} = \mathbf{9.82 * 10^5\ Joule}$$

Despite the fact that more than one thousand times more nodes have been involved in the calculation the energy consumption was two times lower. Even if the latency time would be higher by 10 times, we would not consume more energy with 10 million nodes as with almost 10 thousand. Solving the sparse linear system would still faster.

6 Conclusion

Firstly we summarize the major trends in hardware developments, which we will consider in more detail through the use of our developing "exascale algorithms and solvers" library:

- The performance of each node in an HPC system will continue to rise. This is realised by the increasing number of cores and their vector units (AVX).
- However memory bandwidth reduces the efficiency of computation greatly. An increase in the gap in performance between cache and memory can also be expected in the foreseeable future. On other hand the memory bandwidth cannot be fully exploited with only few cores.
- Communication latency is one of the main limiting factors for scalability. Part of the communication latency is not caused by the network hardware, but rather by various software layers, such as the MPI library or high-level network drivers.
- Hardware accelerators can be faster than one or even two processors. However, the accelerators also have clear disadvantages compared to a processor. This is especially true if the computation is launched on distributed systems where there is a lot communication, which cannot be overlapped.
- The hardware consumes a lot of power, even though its efficiency increases.
- We have also shown that the efficient use of exascale hardware should include active fault handling in message passing. If the cumulative runtime of a parallel application draws near processor lifetime, fault tolerant MPI implementations using spare cores will need to be considered.

Regarding limiting factors, the following points have been identified through the use of our developing "exascale algorithms and solvers" library:

- We should use the vector components of the processors (by using of suitable data structures and controlling of the compiler). This increases the performance and energy efficient of today's processors. It is also essential for the next generation of the hardware.
- We must clearly distinguish between the calculations in memory and in cache. This gives us the opportunity for better parameterisation and extension of the implementation for various cases. It makes also possible the integration of the future work in CRESTA Work package 2 "Power measurement across algorithms" into the library.
- It's strongly recommended to use only one maximal two (by dual socket system) communication processes (mpi process) per node. The threads and processes must be pinned to the cores.
- It is necessary not only to have a better network hardware, but also a better implementation of the collective operations (with possibility of direct use of network driver e.g. API for CRAY Aries interconnect). These are also being developed in WP4.
- Use every opportunity to overlap the communication and computation.
- Use the GPU only for such algorithms, which do not require frequent exchange of data with the host.
- It is also necessary to consider the dynamic allocation of resources (such as the active number of cores, usage of GPU) depending on the size of the problem, algorithms and allocated nodes. The usage of fault tolerant MPI implementations using spare cores should be considered as far as possible.

In order to achieve efficiency on exascale platforms, power consumption must be reduced not only at the hardware level but also at the software level. Some of the above items have expected efficiency improvements. Others, such as the switching of CPU frequency require further research.

A.1 Platforms

All systems, which are mentioned in the deliverable, are in-house systems in HLRS Stuttgart.

A.1.1 NEC Nehalem Cluster

Figure 16 - NEC Nehalem Cluster (Laki)

NEC Nehalem is consists with around 700 nodes. Each node is equipped with dual socket Intel Xeon X5560 @ 2.80GHz quad cores. Some of nodes are replaced with Intel Xeon E5-2670 @ 2.60GHz 8-cores each socket and some with 4 sockets AMD Interlagos, AMD Opteron Processor 6238, 12-cores each. RAM ranges from 12 to 256 GB. 20 nodes are powered with 2x NVIDIA tesla C1060 each and 2 nodes with GTX 680. Nodes are interconnected with InfiniBand and GigaEthernet.

Technical description	
Peak Performance	62 TFlops
Number of Nodes	700 Dual Sockel Quad Core
Processor I	Intel Xeon (X5560) Nehalem @ 2.8 GHz, 8MB Cache
Processor II	Intel Xeon (E5-2670) Sandy Bridge @ 2.60GHz 8, 20MB Cache
Memory/node	12 GB
Disk	80 TB shared scratch (lustre)
Node-node interconnect	infiniband, GigE
Accelerators	20 nodes provide 2x Nvidia Tesla C1060 2 provides GTX 680
graphical pre- and post processing nodes	6 nodes provide NvidiaQuadro 5800FX graphics card

A.1.2 CrayXE6

Figure 17- Cray XE6 (Hermit)

Cray XE6 (Hermit) is the modern system with 3552 compute nodes. Each node has an AMD 2 sockets Interlagos 16 cores each. Each node is equipped with 32 or 64 GB RAM. 28 nodes are powered with NVIDIA K20X. The nodes with GPGPU are with one CPU socket which means only 16-cores. All nodes are interconnected with Gemini interconnect.

Technical description (installation step 1)	
Peak performance	1.045 PFlops
Cabinets	38 with 96 nodes each
Number of compute nodes	3552
Number of compute cores	per node 2 sockets with 16 cores each: 113 664
Number of service nodes	96
Processor compute nodes	Dual Socket AMD Interlagos @ 2.3GHz 16 cores each
Accelerators	28 nodes with NVIDIA K20X
Memory/node	32 GB and 64 GB
Disk capacity	2.7 PB
Node-node interconnect	Cray Gemini
Special nodes	External Access Nodes, Pre- & Post processing Nodes, Remote Visualization Nodes
Power consumption	2 MW maximal

A.1.3 Workstation (Intel E5 2687W)

This station is equipped by the latest hardware. It is used not only for performance but also for power measurements.

Components	Model
Motherboard	Supermicro® X9SRA
CPU	Sandy Bridge Intel® E5 2687W (20 M Cache, up to 3.8 GHz, 8 cores, stepping C2)
RAM	Kingston® Server Premier ® 4x4 GB, DDR3-1600, ECC
Video	HD5450, 1GB DDR3 , PCI Express®
Power supply	Antec® EarthWatts® EA-450 Platinum, 450 Watt
Hard disc	WD Caviar® Black™ WD7502AAEX, 1 TB
Fans	One cpu fan Dark Rock Pro, and 3 chassis fan
Monitor	No
OS	Scientific Linux 6.x (Carbon); Kernel 2.6.32

7 Bibliography

1. **Wan, Ucilia.** http://www.lbl.gov/Science-Articles/Archive/NE-climate-predictions.html. *Berkeley Lab Researchers Propose a New Breed of Supercomputers for Improving Global Climate Predictions.* [Online] 2008.

2. **Ramirez, Alex.** The Mont-Blanc approach toward Exascale. [Online] 2012. http://www.montblanc-project.eu.

3. **Davis, Nick.** Cray Unveils the Cray XC30 Supercomputer -- the Next Generation of Its High-End Supercomputing Systems. *http://www.cray.com.* [Online] CRAY Inc., 2012. http://investors.cray.com/phoenix.zhtml?c=98390&p=irol-newsArticle&ID=1755982.

4. **Kanter, David.** Intel's Haswell CPU Microarchitecture. *real world technologies.* [Online] November 2012. http://www.realworldtech.com/haswell-cpu/.

5. **Booth, Stephen.** *D2.1.1 Architectural developments towards exascale.* s.l. : CRESTA Consortium Partners 2011, 2012.

6. *Power consumption of kernel operations.* **Uwe Küster, Dmitry Khabi.** [ed.] Michael Resch, et al. Stuttgart : Springer, forthcoming (Novermber, 2013). Sustained Simulation Performance.

7. **Wafai, Mhd. Amer.** Sparse matrix vector multiplications on graphic processors. *Dokumentenserver der Universität Stuttgart .* [Online] 2009. http://elib.uni-stuttgart.de/opus/volltexte/2010/5033/.

8. *LogGP: incorporating long messages into the LogP model—one step closer towards a realistic model for parallel computation.* **Alexandrov, Albert et al.** s.l. : Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures (1995), ACM, NewYork..http://doi.acm.org/10.1145/215399.215427., 1995.

9. *Low-Overhead LogGP Parameter Assessment for Modern Interconnection Networks.* **Hoefler, T., Lichei,A. and Rehm W.** s.l. : In Proceedings of the 21st IEEE International Parallel & Distributed Processing Symposium (2007).IEEE Computer Society.

10. *LogGP in Theory and Practice- An In-depth Analysis of Modern Interconnection Networks and Benchmarking Methods for Collective Operations.* **Hoefler, Torsten, Schneider, Timo and Lumsdaine, Andrew.** 9, October 2009, Simulation Modelling Practice and Theory, Vol. 17, pp. 1511-1521.

11. **José Gracia, Christoph Niethammer, Wahaj Sethi.** D4.5.2 Microbenchmark Suite. *CRESTA Consortium Partners 2011.* 2012.

12. The National Institute for Computational Sciences. *IO Lustre Tips.* [Online] National Institute for Computational Sciences, 2013. http://www.nics.tennessee.edu/computing-resources/file-systems/io-lustre-tips.

13. **Stephen P Booth, Dmitry Khabi, Gregor Matura, Christoph Niethammer, Harvey Richardson.** *Overview of major limiting factors of existing algorithms and libraries.* s.l. : https://cresta-project.eu, 2012.

14. *A large scale study of failures in high-performance-computing systems.* **Gibson, Bianca Schroeder and Garth A.** s.l. : International Symposium on Dependable Systems and Networks (DSN 2006). , 2006.

15. **Torsten Hoefler, William Gropp , Rajeev Thakur , and Jesper Larsson.** *Toward Performance Models of MPI Implementations for Understanding Application Scaling Issues.* 2010.