# Memory Bandwidth Contention: Communication vs Computation Tradeoffs in Supercomputers with Multicore Architectures

Johannes Langguth(✉), Xing Cai
Department of High Performance Computing
Simula Research Laboratory
Oslo, Norway
Email: {langguth,xingca}@simula.no

Mohammed Sourouri
Acando Norway
Oslo, Norway
Email: mohammed.sourouri@acando.no

***Abstract*—We study the problem of contention for memory bandwidth between computation and communication in supercomputers that feature multicore CPUs. The problem arises when communication and computation are overlapped, and both operations compete for the same memory bandwidth. This contention is most visible at the limits of scalability, when communication and computation take similar amounts of time, and thus must be taken into account in order to reach maximum scalability in memory bandwidth bound applications. Typical examples of codes affected by the memory bandwidth contention problem are sparse matrix-vector computations, graph algorithms, and many machine learning problems, as they typically exhibit a high demand for both memory bandwidth and inter-node communication, while performing a relatively low number of arithmetic operations.**

**The problem is even more relevant in truly heterogeneous computations where CPUs and accelerators are used in concert. In that case it can lead to mispredictions of expected performance and consequently to suboptimal load balancing between CPU and accelerator, which in turn can lead to idling of powerful accelerators and thus to a large decrease in performance.**

**We propose a simple benchmark in order to quantify the loss of performance due to memory bandwidth contention. Based on that, we derive a theoretical model to determine the impact of the phenomenon on parallel memory-bound applications. We test the model on scientific computations, discuss the practical relevance of the problem and suggest possible techniques to remedy it.**

## I. Introduction

Overlapping communication with computation is a standard strategy in parallel computing, and has been widely employed in various forms. Based on the underlying assumption that communication and computation employ disjoint resources, overlapping can provide a performance improvement of up to a factor of two for communication-heavy codes, at the cost of additional programming effort [1], [2].

For supercomputers based on multicore architectures, a commonly used programming approach is to employ MPI for explicit communication between the compute nodes, whereas OpenMP is adopted for shared memory intra-node parallelism and its implicit data movements. While there are multiple ways to accomplish this, the MPI progress model implies that dedicated communication threads are required to guarantee truly overlapped communication [3]. This usually means that only one core per node (or socket) is responsible for the MPI communication, while the remaining cores share the computation. In compute-bound problems, where the speed of computation is determined by the arithmetic capabilities of the cores, the strategy of overlapping communication with computation can be very effective. However, if the speed of the computation is limited by memory bandwidth, computation and MPI communication may compete for the same resource, i.e. the memory bandwidth.

While the exact amount of memory traffic due to MPI communication can vary with implementation, buffering, and protocol used, it is important to note that any MPI communication requires reading the message data at the source and writing it at the target at least once, with some additional overhead such as the envelope of an MPI message. Thus, memory bandwidth contention cannot be completely avoided when overlapping communication and computation in memory-bound applications.

Furthermore, recent trends in processor architecture have rendered the above problem more pressing than ever. During the last decade, multicore CPUs have completely replaced single-core processors in almost all types of computers, ranging from high performance computing (HPC) systems, to mobile devices. In particular, the HPC area has experienced a rapid growth in the number of cores per CPU, with 16 to 32 being common for new HPC systems today [4]. This increase in core count has led to an unprecedented capability of the CPUs to perform floating-point computations. However, memory bandwidth has not increased by the same pace, as shown in Figure 1.

We can see from Figure 1 that the memory bandwidth has increased at approximately the same rate as the flop/s per core, which is far slower than the rate at which the core count grew. Thus, if a computation draws heavily from memory, contention for the memory bandwidth might become a serious performance limit. As an example, let us compare Sandy Bridge (2012) with Broadwell (2016) in Figure 1. The memory bandwidth per CPU increased by $50\%$, from about $50$ GB/s

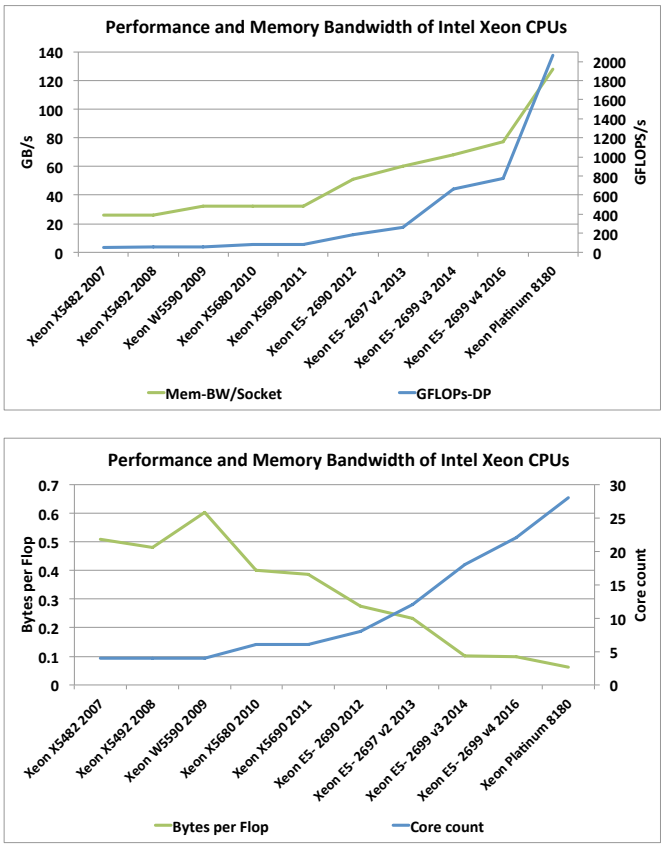



Fig. 1. Development of flop/s and memory bandwidth in Intel Xeon processors. The former is increasing significantly faster than the latter, as evidenced by the falling ratio between them. The main reason for that is the steadily increasing core count. Flop/s is given for double precision, single precision values are always twice that amount. Data is based on vendor specification collected by [5].

to 75 GB/s, while the number of cores grew by $175\%$ from 8 to 22. In case of a single Broadwell core performing the MPI communication, this communicating core in the worst case gets only $1/22$ of the total memory bandwidth, or only $1.5 \times 8/22 \approx 55\%$ of what a single communicating core on the Sandy Bridge processor got. In addition, the Broadwell core may have to handle $50\%$ more MPI traffic at the same time, since a memory-bound computation will run faster on Broadwell due to the higher memory bandwidth per CPU. It is easy to see that unless a code requires very little communication, the situation described above can be a serious impediment to scalability. On the Skylake generation of Xeon processors (e.g. Xeon Platinum 8180), memory bandwidth has grown significantly through the addition of two memory channels while the number of cores did not grow at the same rate, thus giving the processor a higher memory bandwidth per core. However, the arithmetic performance in flop/s has grown even faster.

Thus, the impact of the problem might be even higher if the original computation was not memory bound on Sandy Bridge. While the memory bandwidth has been increasing, the arithmetic performance has grown much faster since it increases linearly with the number of cores and also benefits from an increased flop/s per core. In the above example, the arithmetic performance[1] from Sandy Bridge to Skylake increased by about a factor of 7, making codes much more likely to become memory bound. In an extreme case, the same code might run seven times faster on Skylake, and thus require seven times the communication bandwidth, assuming everything else being equal. If the code indeed becomes memory bound on Skylake through the increased arithmetic performance and the above assumptions hold true, a single Skylake core tasked with communication will have about $90\%$ of the memory bandwidth available that a communicating core had on Sandy Bridge, causing communication to be more than seven times slower (relative to computation). While this clearly is an extreme case, it shows that memory bandwidth contention can be an increasingly relevant problem in modern supercomputers.

The remainder of this paper is organized as follows: After discussing related work in Section II, we begin by measuring memory bandwidth contention using a custom benchmark in Section III. Based on that, we derive a general model for assessing the potential loss of scalability in Section IV, followed by experiments with real-world codes in Section V. We report on further experiments on heterogeneous systems in Section VI and extend our model to that case, and present our conclusions in Section VII.

## II. Related Work

The phenomenon of memory bandwidth contention among multiple cores has been observed ever since large multicore processors became available. Several contributions, e.g. [6], [7], [8], have focused on memory bandwidth contention between different computing cores, i.e. a situation where sufficient memory bandwidth is available for a single core, but not $n$ times that bandwidth for all $n$ cores. This often results in reduced scaling on a single multicore processor. On the other hand, the focus of this paper lies on memory bandwidth contention between computing cores and communicating cores. While the cause of both problems lies within the processor architecture, the latter only affects the scalability of clusters that have multiple nodes.

A model that takes both memory bound computation and communication into account was presented in [9]. Similar to our work, this model aims to predict total running time. However, the crucial difference is that in [9], MPI performance is measured statically, without contention from the computation whereas we measure the effect of memory bandwidth contention on the communication performance. In [9], this leads to a constant predicted overlap between communication and computation. On the other hand, in our model the overlap can change based on the changing communication performance under contention. This difference has only a limited impact in [6], where a compute heavy application was

[1]The Skylake performance includes fused multiply-add, which contributes a factor of 2.

studied. However, in communication heavy cases such as the heterogeneous version of our test application, the difference can be significant. Thus, to sum up our contributions, in this paper we:

1) Provide a simple benchmark to accurately measure the memory bandwidth available to computation and MPI communication under contention.
2) Present a performance model that incorporates the loss of communication performance due to memory bandwidth contention.
3) Discuss the impact of the problem and show possible techniques to alleviate it.

## III. Contention Experiment

In order to quantify the effects of memory bandwidth contention, we use a custom benchmark tool to measure the MPI communication bandwidth that is obtainable by a single core while one or more non-MPI cores are busy with memory accesses for computation. This is achieved by concurrently measuring MPI bandwidth in a manner similar to the OSU MPI benchmark [10] and memory bandwidth with an OpenMP based STREAM Triad style benchmark [11].

Thus, the memory bandwidth contention benchmark is a hybrid MPI+OpenMP program. It uses nested parallel regions to test communication and memory bandwidth concurrently. This means that MPI must be called from within a parallel OpenMP region. In its basic version, only one thread per MPI process calls MPI, which requires the MPI_THREAD_FUNNELED thread safety level. Naturally, a version of benchmark that includes more than one such communication thread would require MPI_THREAD_MULTIPLE. On systems where multithreaded MPI is not available, it is also possible to design a benchmark that eschews the use of OpenMP and places one MPI process per core. Preliminary experiments showed that this generally yields similar results. However, all our experiments in this section are based on using hybrid MPI+OpenMP with one communication thread per MPI process. The actual number of threads assigned to the STREAM part varies from 0 (i.e. an MPI benchmark without contention) to the number of physical cores minus one. With a growing number of these memory-accessing threads, the contention for memory bandwidth increases, which generally leads to a reduced MPI bandwidth. For practical reasons, the contention is quantified separately, which means that when testing the MPI bandwidth, we make sure that the STREAM benchmark takes more time than the MPI test, and vice versa.

The MPI communication itself is performed in a logical ring, where each process sends to its successor and receives from its predecessor concurrently, which approximates a typical scenario in scientific computations. We ran the benchmarks on 4 nodes of our test systems. Their specifications are given in Table I. While both systems have the same type of network adapter, their network topology and the MPI implementations differ, and so do their benchmark results. The codes are compiled with different versions of Intel Compiler Suite in order to optimize the STREAM performance.

TABLE I
Specifications of the test systems. On Cartesius, we use the accelerator island. Sockets are given per node, cores are given per socket.

| System | Cartesius | Vilje |
|---|---|---|
| CPUs | Ivy Bridge | Sandy Bridge |
| #Sockets | 2 | 2 |
| #Cores | $2 \times 8$ | $2 \times 8$ |
| GHz | 2.5 | 2.6 |
| Interconnect | InfiniBand FDR | InfiniBand FDR |
| Topology | Fat tree | Hypercube |
| MPI system | Intel MPI | SGI MPT |
| MPI version | 5.0.3.048 | 2.14 |
| icc version | 15.0.0 | 15.0.1 |
| STREAM Triad | 75 GB/s | 75 GB/s |

We performed two experiments on each system, placing either one MPI process per node or one per socket. In both cases we observed a steady decline of the obtained MPI bandwidth with the number of contending threads. Since we aim to measure bandwidth, not latency, we communicate large messages of 4 MB each. To minimize interference from other network traffic on the supercomputers, we allocated an entire rack for each experiment. Results for the Cartesius [12] test system are shown in Figure 2.

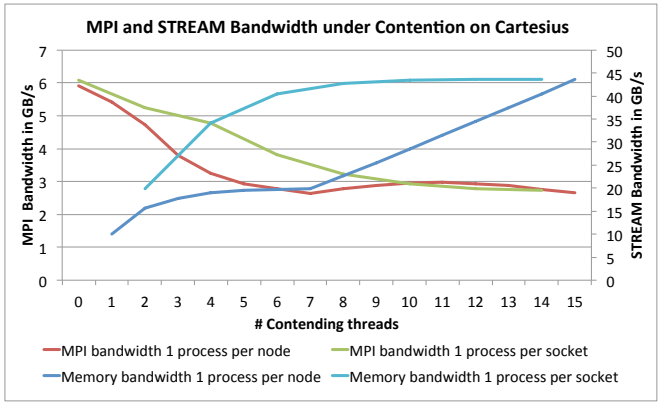



Fig. 2. MPI and STREAM bandwidths under memory bandwidth contention on the Cartesius Supercomputer.

Results for using two MPI processes per node (i.e. one per socket) are given as aggregates per node. For example, the bandwidth values related to using one contending non-MPI thread per process are reported as having 2 contending threads per node, and the bandwidths are twice the results measured for a single process. Naturally, values for odd numbers of competing threads do not exist. This allows us to directly compare performance for the same amount of hardware resources used.

When two MPI processes are used per node, the aggregate memory bandwidth that is obtainable by the non-MPI threads

increases faster with the number of active cores. This is because both memory controllers are in use immediately. For the case of using only one MPI process per node, the contending threads first occupy the first socket (due to the *compact* affinity of OpenMP threads), i.e. the second socket (and its memory controller) is not in operation until half of the total number of threads are used. When all the threads are used, there is no meaningful difference.

By the same token, comparing the one-MPI-process-per-socket and one-MPI-process-per-node scenarios, the measured MPI bandwidths are essentially the same when there are no contending threads or all cores on the node are active. In between, the higher memory bandwidth that stems from using both memory controllers results in a somewhat higher MPI bandwidth.

Finally, we quantify the loss due to memory bandwidth contention by calculating the *loss ratios* for both MPI and STREAM bandwidths, which we will label $L_N$ and $L_M$. The ratios are determined by dividing the uncontended bandwidths with the bandwidths under full memory bandwidth contention. We observe that the MPI bandwidth decreases from $5.8$ GB/s to about $2.7$ GB/s, incurring a loss ratio of $L_N = 2.2$, while the STREAM bandwidth drops from $75$ GB/s (this is not listed in Figure 2) to $43$ GB/s, a loss ratio of $L_M = 1.72$. Considering these ratios, it might be very difficult to gain any benefit from communication-computation overlap in memory-bandwidth-bound computations.

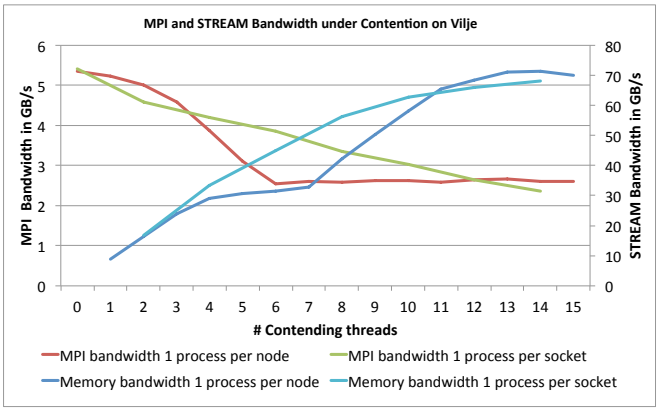



Fig. 3. MPI and STREAM bandwidths under memory bandwidth contention on the Vilje Supercomputer.

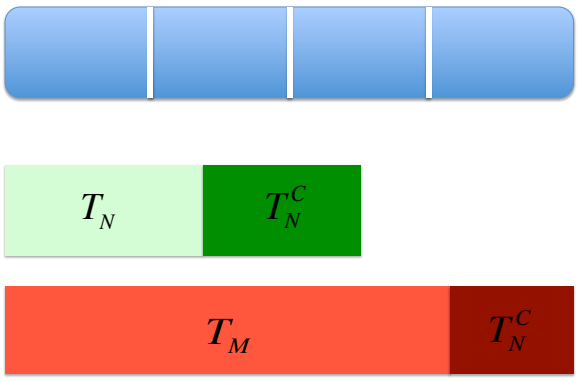
Our second test system is Vilje [13], which is operated by NTNU Trondheim. It has the similar computing hardware, but its network interconnect and topology differs from Abel.. While the individual numbers vary somewhat, the overall picture is very similar to Cartesius. Under heavy memory bandwidth contention, the communicating cores lose approximately half their MPI bandwidth. From these benchmark measurements, we can clearly see that contention can have a severe impact on performance when communication is overlapped with computation.

## IV. Performance Model and Computational Framework

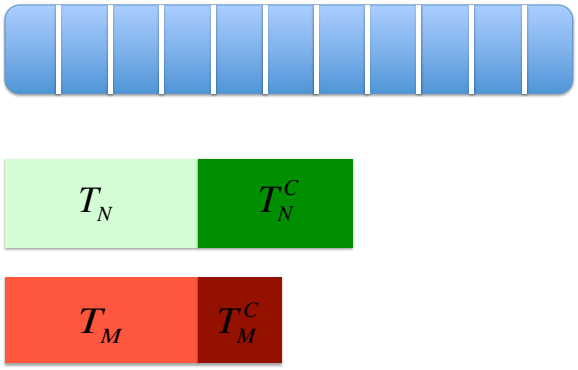
The results from the last section indicate that in any parallel program that is at least partially memory bound, if communication and computation are overlapped, memory bandwidth contention between them can be a significant problem. However, for didactic reasons, we discuss the problem in the context of time-stepped scientific computations which offer a clear structure that allows us to develop a precise model of the problem. Thus, we assume that the program executes a given number of iterations, each of which consists of a separator (or halo) computation in which all values that need to be communicated to other nodes are computed, an interior computation of the non-separator values, overlapped with communication, and a synchronization in the end. We further assume that the program runs on a potentially large number of nodes and uses MPI (or a similar system) to communicate between them. Cores within the node communicate via shared memory, either via MPI or OpenMP (or similar systems).

The interior computation for all non-separator values is executed concurrently with the MPI communication of the separator values. Assuming that the system can provide true overlap between computation and communication, both will be affected by memory bandwidth contention. Note in most cases the impact of not using overlap will be much higher than that of memory bandwidth contention. If this is not the case, an informed programmer can always forgo overlapping, thereby limiting the slowdown to a maximum of a factor of 2. In any case, because the separator values are processed before communication starts, their computation is not performed under memory bandwidth contention (with MPI communication). Therefore, we can omit it in our performance model for simplicity even though it needs to be added when predicting the actual overall performance.

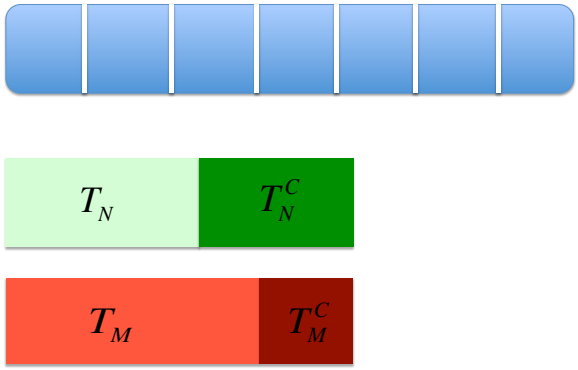
Figure 4 shows an example of the effect of memory bandwidth contention. Let $T_N$ denote the uncontended MPI communication time for a single time step determined via some model or benchmark, and $T_M$ the uncontended memory-bound computation time. In this example, the communication time is independent of the number of processors, such as in a 1D partitioning of a regular domain on a suitable network topology. A typical situation with a low number of nodes and subdomains is shown in Figure 4a, where computation takes much more time than communication. Due to contention, the actual times when overlapping are $T_N^C$ and $T_M^C$. If $T_N << T_M$, most likely we have $T_N^C << T_M^C$, and thus the total time per time step is $T_M^C$ which is only slightly longer than $T_M$. With strong scaling, the number of processors increases, thus reducing the computation time while communication time remains constant, as shown in Figure 4b. A model that ignores memory bandwidth contention will predict good scaling up to the point where $T_M = T_N$, while the actual point is reached at $T_M^C = T_N^C$, thereby overestimating scalability. Because $T_N^C$ is usually far longer than $T_N$, the difference is significant. Therefore, we aim to design a performance model that explains




(a) Few subdomains. Because communication is much faster than computation, the effect of memory bandwidth contention is small.




(b) Many subdomains. While communication and computation take the same time individually, memory bandwidth contention slows down communication and thus the whole program significantly.




(c) Scaling Maximum. To account for memory bandwidth contention, the number of subdomains must be reduced to attain the scaling peak.

Fig. 4. Simple example of the effect of memory bandwidth contention. For simplicity, the example assumes a 1D partitioning of a regular domain. A typical situation with a low number of subdomains is shown in 4a, while 4b shows problems arising in the situation with many subdomains/nodes. The optimum partitioning under memory bandwidth contention is shown in 4c.

this effect.

Our performance model is based on comparing the performance of memory access and network communication, denoted as $P_M$ and $P_N$ respectively. Since we focus on memory bound computations and high-volume communication, performance is measured in bytes per second in both cases. In every time step, the systems performs workloads $W_M$ and $W_N$ which are measured in bytes. Thus, individually processing these workloads takes time $T_M = W_M/P_M$ and $T_N = W_N/P_N$. Without memory bandwidth contention the total time that a single step takes would be $T_{tot} = \max\{T_N, T_M\}$, assuming communication and computation start immediately in a timestep.

Now, under memory bandwidth contention, the bandwidth of both memory and network is reduced. This performance under contention is denoted as $P_M^C$ and $P_N^C$. The time required to process the respective workloads at this reduced speed is $T_M^C = W_M/P_M^C$ and $T_N^C = W_N/P_N^C$. That is, while both operations are running they work with reduced performance. As soon as the first operation finishes, the other one switches back to full performance.

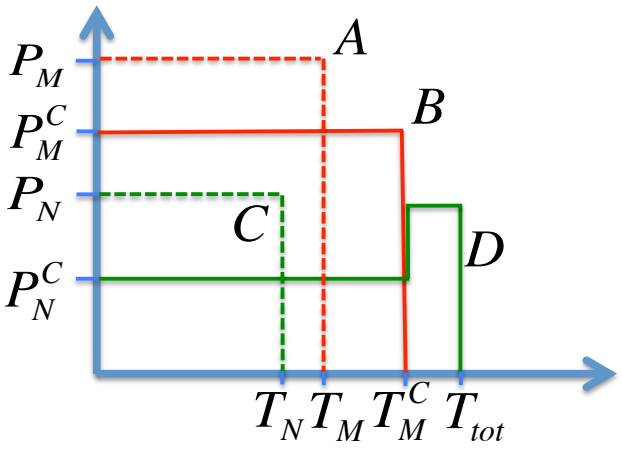



Fig. 5. Example graphical representation of the model showing performance over time. Communication time is shown in green, computation time in red. The dashed lines represent performance without memory bandwidth contention.

A graphical example is shown in Figure 5. The axes in the chart measure time and performance. Thus, the areas under the curves correspond to the workloads. The red curves represent computation, thus encompassing an area equal to $W_M$ and the green curves represent communication and encompass $W_N$. Without contention, the computation would run at performance $P_M$, following curve $A$ which is represented by the dashed red line and ending at $T_M$. Due to memory bandwidth contention however, the computation is performed at $P_M^C$ and finishes at $T_M^C$. Curve $B$ (solid red line) shows this computation. Note that the area under both curves is identical, as it amounts to $W_M$. By the same token, communication without contention would finish at $T_N$, following the dashed green curve $C$. However, it actually starts with reduced performance $P_N^C$ and continues until $T_M^C$. At that point, computation finishes and communication performance increases back to $P_N$, as shown by the solid green curve $D$. After performing an amount of communication that equals $W_N$, the time step ends at $T_{tot}$. Note that this example only shows the case where communication is slower than computation.

Assuming $T_M^C \leq T_N^C$, i.e. communication under contention is not faster than computation, the total time will be at least $T_M^C$. Clearly, $T_{tot}$ can never be more than $T_N^C$. To compute the additional time taken after $T_M^C$, one needs to multiply the difference between $T_M^C$ and $T_N^C$ with the apparent speedup experienced by the communication once memory bandwidth contention stops, which is simply $P_N^C/P_N$, i.e. the inverse of the performance loss due to memory bandwidth contention. In case of $T_M^C \geq T_N^C$, the same effect occurs but the roles of communication and computation are swapped. This effect can be captured in a concise formula, which allows us to predict the total running time due to memory bandwidth contention as:

$$T_{tot} = \min\{T_M^C, T_N^C\} + \max\left\{(T_M^C - T_N^C)\frac{P_M^C}{P_M}, (T_N^C - T_M^C)\frac{P_N^C}{P_N}\right\}$$

The *min* term represents the time under memory bandwidth contention, and the *max* term captures the speedup after contention stops. Like the separator computation, synchronization at the end of every time step, which is common in time-stepped codes, is omitted here, since it is independent of the other parts. Note that $P_M^C/P_M$ is just the inverse memory bandwidth contention loss ratio, and it is equal to $T_M/T_M^C$. The same holds for $P_N^C/P_N$. Thus, $T_{tot}$ can be predicted based on $T_M, T_N$, and the loss ratios $L_M, L_N$ alone.

The model can be illustrated in a graphical representation. For this purpose, we draw a chart where both axes represent time, as shown in Figure 6. The x-axis measures the time usage of communication for a given time used on computation, and vice versa for the y-axis. The green line thus shows the time usage for communication, starting at $(T_N, 0)$ and progressing to $(T_N^C, T_N^C)$. At this point, all communication is under contention, and additional computation will not slow it down anymore. Thus, for compute times larger than $T_N^C$, the communication time remains constant. The red curve describes communication time, and behaves in exactly the same way. Now, the total time usage depends on the distance of the intersection point from the origin in the $L_{\text{inf}}$ norm, since it is determined by the longer of the two running times. Thus, all points on the dotted inverted L-curves correspond to the same running time. Points to the left of the $45°$ line correspond to computation bound settings, while points below the line are communication bound. The example curves correspond to Figure 5, i.e. computation takes the full $T_M^C$. Therefore, the red curve intersects on the constant section. On the other hand, a small part of the communication is performed after the computation finishes, which means that the green curve intersects on the sloped section. The dashed red and green lines correspond to curves $A$ and $C$ in the example described by Figure 5. They show time usage in the absence of bandwidth contention for comparison.

We can also use the model to predict expected running times when changing the parallel system or code. For example, adding computations to a simulation would shift the red curve upwards. Adding accelerators to the system would most likely shift it downwards, as would improving the memory bandwidth. However, in the latter case, communication bandwidth under contention would most likely shift as well. When changing the number of processors, the effect depends on the change in the communication volume $W_N$. Figure 7 illustrates a situation in which $W_N$ is constant independent of the number of processors used, thus, there is only one green communication curve. On the other hand, we have four distinct computation curves $A, B, C$, and $D$, which represent performance under a different numbers of nodes[2]. Curve $A$ corresponds to the lowest performance. Here, communication under contention finishes before computation, leaving a small amount of computation without contention. In $B$ both parts take the same amount of time and thus the curves cross on the $45°$ line, at a higher overall performance. For $C$ and $D$, communication takes longer than computation in both cases. Without memory bandwidth contention, the running time would be $T_N$ in both cases, and given $C$, using additional processors would not speed up the execution. However, under memory bandwidth contention, $D$ is faster because the computation finishes earlier, at which point communication speeds up as it is no longer congested. The result is a faster overall execution, as evidenced by the fact that Curve $D$ intersects with the green communication curve on an inverted L-curve (represented by the dotted blue line) that crosses the $45°$ line closer to the origin.

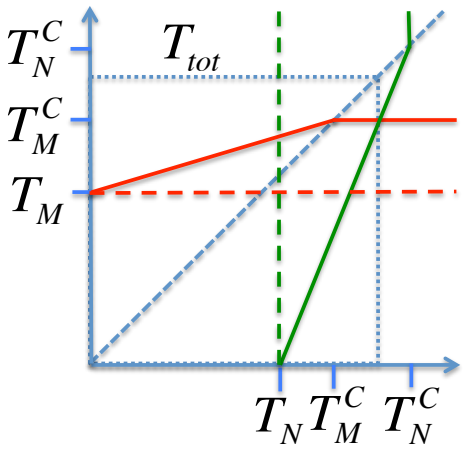



Fig. 6. Graphical interpretation of the performance model. Both axes plot time. Communication time is shown in green, computation time in red. Each dotted blue curve plots all points that take the same total time. The $45°$ line contains all points in which communication and computation take the same amount of time.

When communication is not constant under the number of processors, each of the red curves would be matched to a separate green curve. When the contention performance and the required communication volume for a given number of processors are known, e.g. in structured grid computations,

[2]Since we assume that our computations are bandwidth bound, the number of memory controllers, and thus sockets and nodes, determines performance. The number of cores does not.

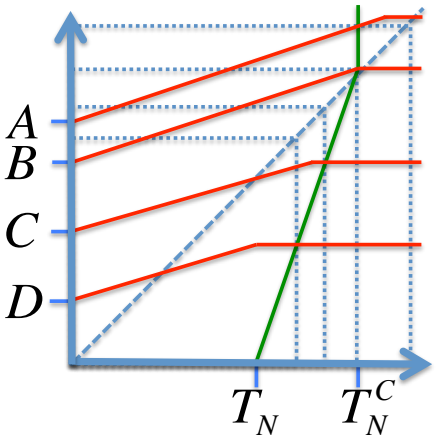



Fig. 7. Graphical interpretation of the performance model with multiple computation performance curves ($A$-$D$, shown in red) and a single communication performance curve shown in green. The remaining elements are the same as in Figure 6.

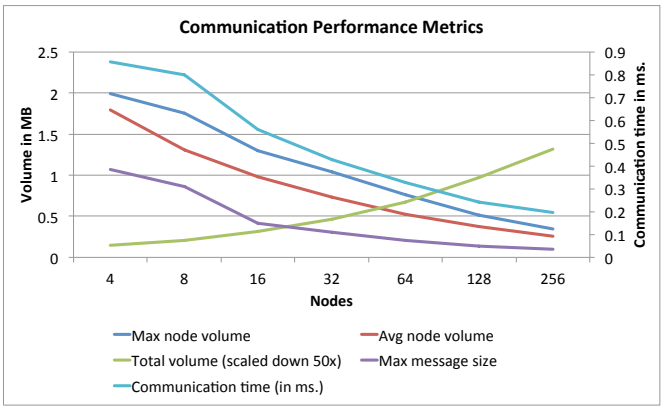



Fig. 8. Communication volumes and costs for the 3D heart mesh.

the communication curves can be derived easily, thus enabling the model to predict the overall performance accurately. Of course, this assumes that the network can handle the increased simultaneous traffic. Given a correct process placement, this usually is the case for modern topologies with high bisection bandwidths, but not for 3D Torus or similar networks. Our model does not take this effect into account, although, given suitable input data, this could be done by changing the communication curves.

Similar to the Roofline model [14], we only take throughput, not latency, into account. Thus, the results of the model can only be valid if large amounts of data are transferred together. However, this is the case for many physical simulations on large grids, where halo exchange messages can easily contain megabytes, which take on the order of milliseconds to transfer, while Infiniband latencies are on the order of microseconds [15]. In these cases the large effective reduction in communication bandwidth due to memory bandwidth contention can be expected to have a substantial impact on performance in communication-critical configurations, e.g. when using a large number of nodes.

## V. Homogeneous Computation Experiment

We investigate the impact of memory bandwidth contention on a real-world computation, using a 3D diffusion equation solver of a monodomain simulator from cardiac electrophysiology. The diffusion solver adopts an explicit cell-centered finite volume method on a tetrahedral mesh. The code was introduced in [16]. In effect, it performs a sparse matrix-vector multiplication on a stiffness matrix that is derived from an irregular mesh representing the human heart. The refined mesh contains 115 million tetrahedra. It is partitioned among the compute nodes using KaHiP [17], and reordered for maximum caching efficiency via PaToH [18]. We place one MPI process and thus one subdomain per node.

Figure 8 shows an analysis of the different metrics determining MPI communication performance. Owing to the quality of the partitioning, all relevant metrics, i.e. the maximum message size and the average and maximum communication volume decrease with the number of nodes. All volumes are given in megabytes. The total communication volume slowly increases with the number of compute nodes. The "total volume" values are scaled down by a factor of 50 in order to plot them on the same scale. We plot the pure communication time for a single timestep in ms on the secondary vertical axis. Clearly, unlike in Figure 7, the communication time decreases with the number of compute nodes in this application, but the rate of decrease is clearly sublinear. Thus, when increasing the number of compute nodes, we can expect to reach a point when communication becomes slower than computation, since the computation time is roughly linear in the number of nodes. However, based on the measurements we project that this will not happen until the number of compute nodes exceeds 8000.

Therefore, w.r.t. our model, we focus on the loss of computational performance due to memory bandwidth contention. To that end, we find $T_M$ experimentally, by running the code without any MPI communication, and also subtract the time it takes to compute the separators. Since the node-to-node communication carries an implicit synchronization we introduce an MPI_Barrier to make computations comparable. Note that due to the irregular nature of the computation, predicting $T_M$ via the Roofline model is not accurate enough for our purposes, although the performance is closely related to the STREAM bandwidth. We perform these experiments on the Vilje Supercomputer presented in Section III. Based on the measurements obtained via the memory bandwidth contention benchmark presented earlier, we can easily determine $T_M^C$ and $T_N^C$ by multiplying $T_M$ and $T_N$ (from Figure 8) with the loss ratios measured there. The values we obtained are listed in Table II.

In this manner, we can calculate $T_M^C - T_N^C$, i.e. the time that computation proceeds unimpeded by memory bandwidth contention, and divide it by the loss ratio. Doing so gives us predictions that are quite close to our actually measured

TABLE II
MODEL PREDICTIONS AND PREDICTION ERRORS FOR THE HOMOGENEOUS DIFFUSION CODE. $T_{tot}$ IS THE PREDICTED VALUE AND REAL THE MEASURED VALUE.

| Nodes | $T_M$ | $T_M^C$ | $T_N$ | $T_N^C$ | $T_{tot}$ | Real | Error |
|---|---|---|---|---|---|---|---|
| 4 | 124.58 | 137.54 | 0.86 | 1.96 | 124.76 | 131.38 | 5.3% |
| 8 | 63.72 | 70.35 | 0.80 | 1.83 | 63.89 | 65.34 | 2.3% |
| 16 | 32.37 | 35.74 | 0.56 | 1.28 | 32.49 | 31.16 | 4.1% |
| 32 | 16.21 | 17.90 | 0.43 | 0.98 | 16.30 | 16.10 | 1.2% |
| 64 | 7.57 | 8.36 | 0.33 | 0.75 | 7.64 | 8.13 | 6.4% |
| 128 | 3.48 | 3.85 | 0.24 | 0.55 | 3.54 | 3.89 | 10.2% |
| 256 | 1.71 | 1.88 | 0.20 | 0.45 | 1.75 | 1.99 | 13.9% |

running times. The model also indicates that improving MPI communication performance is unlikely to result in overall speedups, since MPI communication takes only a small fraction of the total time, assuming the contention is not reduced. We performed an additional experiment to test this by modifying the code to use multiple communication threads in each node. Results are shown in Figure 9, where values are given as performance per node, as differences in the total values would not be visible due to the strong scaling.

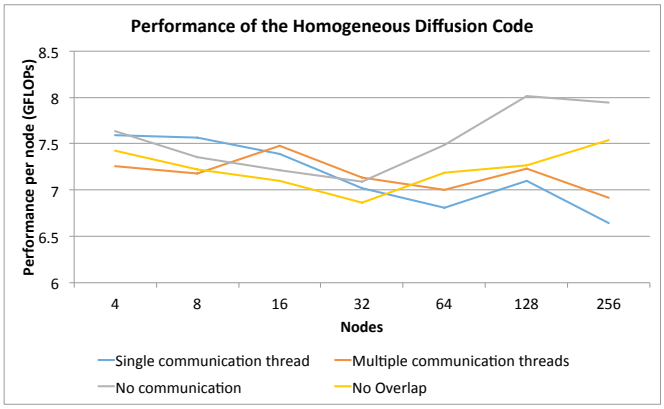



Fig. 9. Effects of using a single vs. multiple communication threads, and the total impact of communication.

The figure also shows the values of $T_M$ used in the above predictions. Note that there is no underlying reason for the fact that the no-communication version sometimes has weaker performance than others. This is simply due to the fact that for small numbers of nodes, the communication time is insignificant. Thus, we can consider the differences up to 32 nodes to be measurement noise. Clearly, the impact of communication only becomes visible for large numbers of nodes, when the ratio between communication and computation becomes larger. At that point, the no-communication version is clearly above the actual computations. For larger numbers of nodes, we also observe a small increase in performance per node. This is due to improved caching, since the strong scaling implies that progressively smaller workloads are assigned to each node. See [19] for an extensive discussion of this effect.

Furthermore, we clearly see that the impact of multiple communication threads is minor. The reason for this is that while it reduces communication time under memory bandwidth contention, it increases the loss ratio, and the total amount of memory bandwidth contention remains essentially the same. Another observation from Table II is that $T_M + T_N$ is usually almost the same as $T_{tot}$, which means that communication-computation overlap has no effect. We tested this by running a non-overlapping version of the code, the results of which are also plotted in Figure 9. We observe that the performance is very similar to both overlapping codes. In other words, the model can be used to determine whether the extra effort involved in developing an overlapping code is worthwhile, since in cases like this a much simpler non-overlapping code can provide the same performance. Note that the Gflops/s values presented here are actual performance values corresponding to those presented in [16], which include separator computations.

## VI. HETEROGENEOUS COMPUTATIONS

Modern heterogeneous systems such as Stampede, Piz Daint, and Titan possess one or more powerful accelerators per compute node, in addition to the CPUs. Consequently, they have significantly higher computational power per node, but their communication bandwidth is generally the same as on homogeneous systems. This implies that for the same task, such systems will perform the computation faster while communication happens at the same speed, which poses an additional challenge to scalability. In addition, due to the limited device memory available in most accelerators, it is usually necessary to employ a large number of nodes for the computation of large problems.

A common strategy (e.g. [20], [21]) for such systems is to perform an offloaded homogeneous computation, which means that all actual computation is performed by the accelerators, while the CPUs only handle administrative tasks and communication. When doing so, memory bandwidth contention on the CPU side is most likely of no concern. Furthermore, by using GPUDirect [22], it is also possible to communicate while bypassing the CPUs entirely. In that case, memory bandwidth contention might affect the device memory bandwidth on the accelerator and its connection to the network adapter.

On the other hand, it is also possible to perform the actual computation on both CPUs and accelerators concurrently, either by splitting the work on each node in multiple parts whose size is commensurate with the computational capabilities of the CPU and the accelerators (e.g. [16]), or by performing different parts of the computation on the different devices (e.g. [23]). We refer to both variants as truly heterogeneous computations. For these computations, the memory bandwidth contention problem occurring on the CPU may indirectly affect the accelerators by causing them to idle. For example, consider the 3D diffusion equation solver presented in the last section. We tested a GPU-only version of that code on a single node of Cartesius and found that the two K40 GPUs can provide 25 Gflops/s each, whereas the CPU-only version provides up to 10 Gflops/s when running on one compute node. Thus, following the guidelines given in [24], we could put up to

$10/(10 + 2 \times 25) = 16.6\%$ of the total workload on the CPU side, a strategy which works well for a single-node system. In a multi-node scenario under memory bandwidth contention, however, the actual performance provided by the CPUs will be lower. Furthermore, since the CPUs and GPUs need to synchronize at the end of each timestep, losing $10\%$ performance, i.e. 1 Gflop/s on the CPU means that the system incurs a combined additional loss of 5 Gflops/s on the two GPUs.

If we know $T_N$ and $T_M$, we can apply our model to predict the actual CPU performance, using the loss ratios of $L_N = 2.2$ and $L_M = 1.72$ measured in Section III for this system. Using the communication and computation times from the last section, we find that at 32 nodes, the uncontended communication takes about $0.5$ ms, while the uncontended computation takes 1 ms. We thus obtain $T_N^C = 1.1$ ms, and $T_{tot} = 1.46$ ms, i.e. a slowdown of $46\%$. Under that slowdown the GPUs will have a considerable amount of idle time, and we can expect a performance of $60/1.46 = 41.1$ Gflops/s per node, or 1315 Gflops/s in total for 32 nodes on Cartesius.

On the other hand, if we reduce the CPU load by half to $8.3\%$ and divide that work among the GPUs, the total time is expected to increase by $10\%$. However, the actual time under memory bandwidth contention on the CPU is reduced to $0.97$ms (from $1.46$ ms), which means that synchronization with the CPU no longer slows down the GPUs, resulting in a total performance of $1741$ Gflops/s. In [16], a similar loss in performance, due to an idealized but too high CPU workload ratio, was observed when using $64$ nodes with two GPUs each. It caused the performance of the heterogeneous version to drop below that of a pure GPU version.

More generally, in order to solve the heterogeneous load balancing problem under memory bandwidth contention, we need to transform the model equation from Section IV to incorporate the accelerator performance. To do so, we can assume that $T_M$ continues to denote the CPU time use. Let $T_A$ be the accelerator time use. Clearly, the time taken by the heterogeneous computation is the maximum of $T_A$ and $T_{tot}$ as computed by the model. However, $T_A$ can be linked to $T_M$. Similar to [24], let $w$ be the accelerator workload ratio, i.e. the fraction of $W_M$ that is assigned to the GPU. Consequently, a workload of $W_M(1-w)$ is assigned to the CPU. Note that neither CPU nor GPU need to be a single device. The ratios $w$ and $1-w$ can refer to a collection of devices as long as their performance is identical. Now clearly $T_M = W_M(1-w)/P_M$. Let $P_A$ be the performance of the accelerators, then $T_A = \frac{W_M w}{P_A}$ and consequently:

$$T_{tot} = \max\left\{\frac{W_M w}{P_A}, T_N^C + \frac{W_M(1-w)}{P_M} - \frac{T_N^C}{L_M}\right\}$$

if $\frac{W_M(1-w)}{P_M} L_M \geq T_N^C$ and otherwise

$$T_{tot} = \max\{\frac{W_M w}{P_A}, \frac{W_M(1-w)}{P_M} L_M + T_N - \frac{W_M(1-w)}{P_M}\frac{L_M}{L_N}\}$$

The difference between the two terms in the max expression is the time that one type of the devices is idle. Consequently, maximum performance is attained when they are equal. Note that if $T_N^C$ is very high, i.e. the system is communication bound, it is possible that no value $w \in [0,1]$ satisfies the equation. In that case setting $w = 1$, i.e. running the entire workload on the accelerators is the optimum solution as it minimizes the idle time. Furthermore, this model only works under the assumption that the workload can be split almost arbitrarily, and that performance is independent of the workload size. Due to cache effects, this is not always the case. Predicting this effect requires an even more complex model, such as the one presented in [19]. Finally, limited memory on the GPUs might further limit the feasible values for $w$.

Since the above considerations assume a CPU plus accelerator setup, the situation is somewhat different for the Intel Xeon Phi 7200 series (generally known as Knights Landing). In its main configuration, it acts as a self-hosted single-socket processor rather than an accelerator, which means that it operates in a manner comparable to a large multicore CPU. While the number of cores on a Knights Landing is very high (at least $64$), the memory bandwidth is also very high as long as data resides within the limited high-bandwidth memory. As a rough estimate, with a memory bandwidth of approximately 400 GB/s [25], each core would get 6.25 GB/s, which is approximately the realistically attainable bandwidth of 4x FDR InfiniBand. Thus, memory bandwidth contention would not be more problematic on the Xeon Phi than it is on regular multicore CPUs, as long as the high-bandwidth memory is used.

## VII. Conclusion

We have seen that memory bandwidth contention is a common problem in many distributed memory computations on multicore CPUs, affecting both computation and communication. Given the appropriate measurements, one can accurately determine whether communication or computation is the performance limiting factor under memory bandwidth contention. If the communication slowdown becomes the principal performance limiter, there are several potential ways of dealing with the problem:

- The easiest solution would be to assign priority of memory accesses to the communicating cores. However, for current hardware there is no portable way of doing so.
- In some cases, the problem can be alleviated by using more communication cores so that they together get a bigger share of the memory bandwidth. Memory bound problems can reduce the number of cores used for computation, and increase the number of cores for communication without losing overall performance.

- Existing MPI+OpenMP codes that use a single MPI process per node can be run using more MPI processes and fewer OpenMP threads per process. This might however be detrimental to data locality and should be used with caution.

For memory-bound problems that adopt the strategy of overlapping MPI communication with computation, there is no real way to avoid the computation slowdown due to contention, since data has to be read from memory for both communication and computation. However, the performance loss can be limited by correctly anticipating the slowdown and adjusting e.g. the load balance between the multiple sockets per node, if only one socket communicates, or by correcting the workload division between CPUs and accelerators in heterogeneous computations.

In these cases, given correct measurements of the input parameters, our model can help to determine optimal configurations and to understand problems with scalability arising from memory bandwidth contention. In future work, we will investigate the use of memory bandwidth in heterogeneous computations on both CPUs and accelerators in order to develop strategies for maximizing the scalability of such codes.

## Acknowledgements

The research work leading to these results has received funding from the Norwegian Research Council under the ExaComp project (RCN 251186) as well as the European Union's Horizon 2020 Programme under grant agreement number 671657. Research was performed using resources provided by the SURFsara Dutch national supercomputing center and Uninett Sigma2 in Norway.

## References

[1] T. H. Kaiser, S. B. Baden, Overlapping communication and computation with OpenMP and MPI, Scientific Programming 9 (2-3) (2001) 73–81.

[2] M. Sourouri, S. B. Baden, X. Cai, Panda: A compiler framework for concurrent CPU+GPU execution of 3D stencil computations on gpu-accelerated supercomputers, International Journal of Parallel Programming. 45 (3) (2017) 711–729.

[3] M. Si, A. J. Peña, J. Hammond, P. Balaji, M. Takagi, Y. Ishikawa, Casper: An asynchronous progress model for MPI RMA on many-core architectures, in: 2015 IEEE International Parallel and Distributed Processing Symposium, 2015, pp. 665–676.

[4] Top500 Supercomputing Sites, http://www.top500.org.

[5] K. Rupp, CPU, GPU and MIC Hardware Characteristics over Time, www.karlrupp.net/2013/06/cpu-gpu-and-mic-hardware-characteristics-over-time/.

[6] B. Putigny, B. Goglin, D. Barthou, A benchmark-based performance model for memory-bound hpc applications, in: 2014 International Conference on High Performance Computing Simulation (HPCS), 2014, pp. 943–950.

[7] D. Dauwe, E. Jonardi, R. Friese, S. Pasricha, A. A. Maciejewski, D. A. Bader, H. J. Siegel, A methodology for co-location aware application performance modeling in multicore computing, in: 2015 IEEE International Parallel and Distributed Processing Symposium Workshop, 2015, pp. 434–443.

[8] S. Bardhan, D. A. Menascé, Predicting the effect of memory contention in multi-core computers using analytic performance models, IEEE Transactions on Computers 64 (8) (2015) 2279–2292.

[9] X. Wu, V. Taylor, Performance modeling of hybrid MPI/OpenMP scientific applications on large-scale multicore supercomputers, Journal of Computer and System Sciences 79 (8) (2013) 1256–1268.

[10] J. Liu, B. Chandrasekaran, W. Yu, J. Wu, D. Buntinas, S. P. Kini, D. K. Panda, P. Wyckoff, Microbenchmark performance comparison of high-speed cluster interconnects, IEEE Micro 24 (1) (2004) 42–51.

[11] J. D. McCalpin, Memory bandwidth and machine balance in current high performance computers, IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter (1995) 19–25.

[12] Cartesius: the Dutch supercomputer, https://userinfo.surfsara.nl/systems/cartesius.

[13] About Vilje, https://www.hpc.ntnu.no/display/hpc/About+Vilje.

[14] S. Williams, A. Waterman, D. Patterson, Roofline: An insightful visual performance model for multicore architectures, Commun. ACM 52 (4) (2009) 65–76.

[15] G. Pfister, An Introduction to the InfiniBand Architecture, IEEE Press, 2001.

[16] J. Langguth, M. Sourouri, G. T. Lines, S. B. Baden, X. Cai, Scalable heterogeneous CPU-GPU computations for unstructured tetrahedral meshes, IEEE Micro 35 (4) (2015) 6–15.

[17] P. Sanders, C. Schulz, Think Locally, Act Globally: Highly Balanced Graph Partitioning, in: Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13), Vol. 7933 of LNCS, Springer, 2013, pp. 164–175.

[18] U. V. Çatalyürek, C. Aykanat, A hypergraph model for mapping repeated sparse matrix-vector product computations onto multicomputers, in: Proceedings of International Conference on High Performance Computing, 1995.

[19] J. Langguth, N. Wu, J. Chai, X. Cai, Parallel performance modeling of irregular applications in cell-centered finite volume methods over unstructured tetrahedral meshes, Journal of Parallel and Distributed Computing 76 (2015) 120–131.

[20] J. Zhou, Y. Cui, E. Poyraz, D. J. Choi, C. C. Guest, Multi-GPU implementation of a 3D finite difference time domain earthquake code on heterogeneous supercomputers, in: Proceedings of the International Conference on Computational Science, ICCS 2013, Barcelona, Spain, 5-7 June, 2013, Vol. 18 of Procedia Computer Science, Elsevier, 2013, pp. 1255–1264.

[21] T. Shimokawabe, T. Aoki, T. Takaki, T. Endo, A. Yamanaka, N. Maruyama, A. Nukada, S. Matsuoka, Peta-scale phase-field simulation for dendritic solidification on the TSUBAME 2.0 supercomputer, in: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC '11, ACM, New York, NY, USA, 2011, pp. 3:1–3:11.

[22] NVIDIA GPUDirect, https://developer.nvidia.com/gpudirect.

[23] J. Choi, A. Chandramowlishwaran, K. Madduri, R. Vuduc, A CPU–GPU hybrid implementation and model-driven scheduling of the fast multipole method, in: Proceedings of Workshop on General Purpose Processing Using GPUs, GPGPU-7, ACM, New York, NY, USA, 2014, pp. 64:64–64:71.

[24] J. Langguth, X. Cai, Heterogeneous CPU-GPU computing for the finite volume method on 3D unstructured meshes, in: The 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS 2014), IEEE Computer Society Press, Hsinchu, Taiwan, 2014, pp. 191–199.

[25] D. Doerfler, J. Deslippe, S. Williams, L. Oliker, B. Cook, T. Kurth, M. Lobet, T. M. Malas, J. Vay, H. Vincenti, Applying the roofline performance model to the Intel Xeon Phi Knights Landing processor, in: High Performance Computing - ISC High Performance 2016 International Workshops, ExaComm, E-MuCoCoS, HPC-IODC, IXPUG, IWOPH, P^3MA, VHPC, WOPSSS, Frankfurt, Germany, June 19-23, 2016, Revised Selected Papers, Vol. 9945 of Lecture Notes in Computer Science, 2016, pp. 339–353.