MB3 MS13– TinyMPI tasking prototype
Version 1.0

Document Information

Contract Number	671697
Project Website	www.montblanc-project.eu
Contractual Deadline	PM39
Dissemination Level	PU
Nature	Report
Authors	Alexandr Nigay (ETHZ), Timo Schneider (ETHZ), Torsten Hoefler (ETHZ)
Contributors	Alexandr Nigay (ETHZ), Timo Schneider (ETHZ), Torsten Hoefler (ETHZ)
Reviewers	Xavier Martorell (UPC), Daniel Ruiz Munoz (ARM)
Keywords	MPI, MPI virtualization, user-level threads, oversubscription, computation-communication overlap

Notices: *This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement N° 671697.*

Change Log

Version	Description of Change
v0.1	Initial version of the deliverable
v0.2	Changed figure sizes and wording
v0.3	Changed figures, wording, added future work
v1.0	Final version

Contents

Executive Summary

This document describes TinyMPI, a prototype MPI implementation which follows the non-traditional approach of *virtualizing* MPI, and presents the results of a research effort which employed TinyMPI as a research vehicle.

Traditional MPI implementations run at most one MPI rank per CPU core. TinyMPI runs more than one MPI rank per CPU core, i.e., it oversubscribes the CPU, with the goal of achieving automatic computation-communication overlap: when one MPI rank blocks, TinyMPI switches to another and continues using the CPU.

We have used TinyMPI as a tool in the research effort of answering the question, "How many ranks exactly to start on each CPU core?", the results of which — in the form of a model — are presented in this document along with a description of TinyMPI's internals.

TinyMPI supports the Arm architecture and is deployed on the Dibona cluster.

1 Implementation of TinyMPI

1.1 Brief description of TinyMPI

Computation–communication overlap and good load balance are two very important prerequisites for achieving high performance of parallel programs, even more so on large-scale machines with high core counts, such as the architecture envisaged by the Mont-Blanc project.

Unfortunately, those two traits are hard to achieve with MPI [Mes15], the *de facto* standard technology for network communication in HPC software, because this requires the use of the *nonblocking* interface of MPI, which makes the code more complex and hard to maintain.

An alternative solution is to use a *virtualized* MPI implementation. Traditionally, at most one MPI process is launched per CPU core, but a virtualized MPI would launch more than one process per core and keep switching between them. When one MPI process blocks in a communication call, the implementation can switch to a process which is ready to compute, thus achieving overlap of computation and communication. This approach permits the user code to use the conceptually-simpler *blocking* interface of MPI. Load balancing can also be implemented by migrating MPI processes between CPU cores and potentially even nodes—all done without any participation from the user code, which drives the complexity even further down.

TinyMPI is a virtualized MPI implementation developed under this project and used as a research vehicle for various investigations.

1.2 Implementation of TinyMPI

TinyMPI is written in C++11 and is almost header-only: essentially only the definitions of variables are placed in compiled files, while all the MPI logic resides in header files.

On the Arm architecture, TinyMPI uses the `makecontext`[1] family of functions to implement user-level threads and it uses Pthreads for kernel-level threads. Figure 1 summarizes TinyMPI's task topology. During startup, TinyMPI proceeds as follows:

1. The underlying non-virtualized MPI is used to start a single operating-system process on each distinct compute node (a distinct machine on the network).

2. Each of those processes spawns the requested number of kernel-level threads using Pthreads. Each of these threads is pinned to a different CPU core.

3. Each of the kernel-level threads spawns the requested number of user-level threads, each of which will become an MPI rank visible to the user application.

Two kinds of MPI ranks exist at this point: the MPI ranks of the underlying non-virtualized MPI, which are not visible to user code and are used internally by TinyMPI; and the MPI ranks exposed by TinyMPI to the user application.

TinyMPI implements network communication using RDMA via Infiniband Verbs API. Each MPI rank is associated with two queues: *waiting queue*, which holds local receive requests which have not been matched yet; and *unexpected queue*, which holds incoming send requests for which a local matching receive has not been posted yet. A posted receive first checks the unexpected queue, and if matching fails there, appends to the waiting queue. An incoming send request targeting this rank first checks the waiting queue and only then goes to the unexpected

[1] `man makecontext`

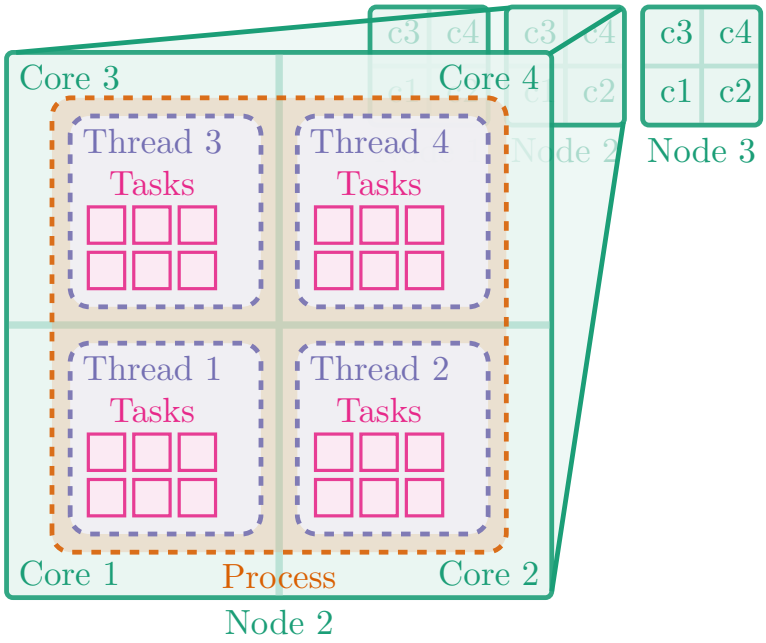

Figure 1: Task topology in TinyMPI: a single operating-system process launches on each compute node (e.g., node 2, a distinct machine on the network); a single kernel-level thread (e.g., Thread 1) launches on each CPU core (e.g., Core 1); multiple user-level threads (tasks) run on each kernel-level thread; each MPI rank is a distinct user-level thread (task).

queue. TinyMPI sends short messages eagerly, while large messages are sent with the rendezvous protocol [WSB+06].

Shared-memory communications are implemented via a single copy from the source buffer directly into the destination buffer—since all ranks on the same compute node are user-level threads spawned off the same process, they share a single address space and a direct copy is possible in user space.

TinyMPI still requires a regular, non-virtualized MPI implementation: the regular MPI is used for spawning processes, one per each node, and it is also used as a stage in collective operations (e.g., MPI's `MPI_Allreduce` is called as a part of TinyMPI's `TMPI_Allreduce`).

TinyMPI currently implements the following subset of MPI:

- `TMPI_Send`, `TMPI_Isend`
- `TMPI_Recv`, `TMPI_Irecv`
- `TMPI_Wait`, `TMPI_Waitall`
- `TMPI_Sendrecv`
- `TMPI_Barrier`, `TMPI_Allreduce`, `TMPI_Bcast`, `TMPI_Allgather`.
- `TMPI_Comm_size`, `TMPI_Comm_rank`
- `TMPI_Get_count`, `TMPI_Request_get_status`, `TMPI_Type_size`
- `TMPI_Wtime`, which can be switched to measure rank-POV time (which is described in the deliverable D7.7 [MB317]).

1.3 Using TinyMPI on Dibona

TinyMPI can be loaded on Dibona with `module load cnrs/tinympi/gcc7.2.1/0.1`.

For a code to work with TinyMPI, the following requirements must be observed:

- The code must be written in C++.
- TinyMPI defines the `main()` entry point on its own, and the user application's entry point must be put into a function `TMPI_main(int, char**)`.
- User code should *not* call `TMPI_Init()` and `TMPI_Finalize()`—TinyMPI is already initialized when `TMPI_main(int, char**)` is called, and it is finalized when that call returns.
- The code must use `TMPI_*` functions instead of `MPI_*`: TinyMPI's interface uses prefix `TMPI_*`. Symbols starting with `MPI_*` refer to the symbols of the underlying non-virtualized MPI. This applies to all symbols, including, e.g., `TMPI_COMM_WORLD`.
- Manually privatize global variables that are written by the user code: since TinyMPI maps MPI ranks to user-level threads which are spawned off a single process on each node, all ranks will share global variables within each node. If this is not what the user code expects, the globals have to be privatized, e.g., by passing them through function arguments.

TinyMPI provides two commands: `tinympicxx` and `tinympirun`.

Code should be compiled with `tinympicxx`, which is a wrapper around a C++11 compiler, providing TinyMPI's include and link flags. `tinympicxx` has been tested with GNU compilers. Any arguments given to `tinympicxx` are passed to the underlying compiler.

Compiled code is executed with `tinympirun`, which should be called as follows:
`tinympirun -nn <N> -nc <C> -nv <V> <user-binary> <user-args>`
Example: `tinympirun -nn 4 -nc 8 -nv 4 ./benchmark -q10`

The parameters have the following meaning:

- `-nn <N>` specifies the number of nodes (distinct machines connected via network) to use.
- `-nc <C>` specifies the number of CPU cores to use *on each node*, so the global number of cores used will be $N \times C$; fewer cores than available may be used, if desired.
- `-nv <V>` specifies the number of MPI ranks to launch *on each core* (this is the *virtualization ratio*), so the size of `TMPI_COMM_WORLD` is $N \times C \times V$.

2 Virtualization Ratio

This section presents the results of research work conducted using TinyMPI as a research vehicle.

2.1 Problem statement

The number of MPI ranks that a virtualized MPI implementation launches per CPU core is referred to as *virtualization ratio*. We will also refer to this parameter as V. Intuitively, V plays an important role in the overall performance: if $V = 1$, then no virtualization benefits can be seen at all; if V is too large, then the overhead of task switching and the increased volume of intra-node communication will outweigh all benefits.

Listing 1: Outline of the microbenchmark employed for developing the model for the virtualization ratio. The code mimics a two-dimensional stencil application. Each rank communicates with 4 neighbors.

```
int main(int argc, char** argv) {
    MPI_Init();
    for (int i = 0; i < ITERS; i++) {
        /* Computation phase */
        compute();

        /* Communication phase */
        MPI_Irecv(); MPI_Irecv();
        MPI_Irecv(); MPI_Irecv();

        MPI_Isend(); MPI_Isend();
        MPI_Isend(); MPI_Isend();

        MPI_Waitall();
    }
    MPI_Finalize();
}
```

Currently, the value of V has to be chosen empirically. However, for a parameter as important as this, it would be very beneficial to have a model which could allow choosing the value analytically. This section of the report presents our efforts in developing such a model.

2.2 Modelling the influence of the virtualization ratio

We have set out to design an analytical expression which, taking V and other relevant parameters as input, would produce the amount of time a given program will take to execute. We will call such a model a *virtualized model*. Such a model not only allows to choose the optimal value of V but also to compare the predicted running time with the current values, which can be used to determine whether a virtualized MPI will or will not benefit the application at hand even before commencing any porting efforts.

Since a virtualized model inherently depends on the behavior of the user application, a performance model for the user application, which we refer to as the *base application performance model*, must be found first.

2.2.1 Performance model of the user application

In our work we have used a synthetic microbenchmark which mimics a two-dimensional stencil computation. The outline of this code is presented in Listing 1.

The beneficial effects of MPI virtualization involve computation–communication overlap, which is the overlap of CPU activity and network activity. Hence, a virtualized model must be able to evaluate the influence of V on these two categories separately. In addition to that, for reasons that will be made clearer later, a virtualized model of our design requires the CPU activity to be broken down into two separate categories: computation and shared-memory communication.

The three necessary parts of the performance model of our microbenchmark are as follows:

- The computation time of each MPI rank:

$$T_{comp}(P) = T_{serial} \quad + \quad iters \cdot \frac{gxsz \cdot gysz \cdot kmax}{tsc \cdot 1024} \cdot \frac{1}{P} \tag{1}$$

- The shared memory communication time of each MPI rank:

$$T_{shmem}(P) = iters \cdot 4 \left(L_s + \frac{4 \cdot halo \cdot gxsz}{B_s} \right) \tag{2}$$

- The network communication time of each MPI rank:

$$T_{net}(N, C, V) = iters \cdot 4 \left(L_N + \frac{C \cdot \frac{8 \cdot halo \cdot gysz}{C \cdot V}}{min(B_{NIC-NIC}, C \cdot B_{CORE-NIC})} \right) \tag{3}$$

The network-communication model employs the maxrate model introduced by Gropp et al. [GOS16]. Parameter P denotes the number of MPI ranks; $gxsz$, $gysz$, $kmax$, $iters$, and $halo$ are the parameters of the application code itself; tsc is the frequency of the CPU, which is also a parameter of the microbenchmark; L_s, B_s denote the latency and bandwidth of shared-memory communication; L_N denotes the network latency; $B_{NIC-NIC}$ denotes the bandwidth between two network interface cards (NIC); $B_{CORE-NIC}$ denotes the peak bandwidth between a CPU core and the NIC installed in the same node; N is the number of compute nodes participating in the computation; C is the number of CPU cores within each compute node; V is the virtualization ratio.

The fit of this model is presented in Figure 2, in which the three components of the model are summed and plotted alongside measured data (blue dots). Note that the blue dots are very close together and in some cases completely overlapped by their median (red dots). In the bottom subplot we show the relative error of the model. All experiments have been performed on the Dibona cluster (Arm architecture). Since the experiment shown in Figure 2 uses strong-scaling and we want to keep the computation per process equal for all process numbers, we exclude some process numbers which would lead to a overly large domain size.

2.2.2 The virtualized model

The virtualized model that we have designed has the following shape:

$$\begin{cases} T_{virt}(N, C, V) = \max\left(cpu, net\right) & + \quad \frac{1}{2} \cdot T_{comp}(N \cdot C \cdot V) \\ cpu = V \cdot T_{comp}(N \cdot C \cdot V) & + \quad V \cdot T_{shmem}(N \cdot C \cdot V) \\ net = V \cdot T_{net}(N, \ C, \ V) \end{cases} \tag{4}$$

T_{virt} is the predicted running time of a virtualized MPI application running on N nodes, each of which has C CPU cores, and with V MPI ranks per core. T_{comp}, T_{net}, and T_{shmem} are the computation, network communication, and shared-memory communication components of the base application performance model.

Figures 3 to 7 present the fit of this model for the benchmark application presented in the previous section. We vary the number of cores used (parameter C) per Dibona node between 3 and 12. The base performance model of the application was designed to predict the running time in seconds, therefore the virtualized model also predicts running time in seconds. The running time is shown as a blue dot for each of the 25 runs (we also show the median in red). Green dots are used to indicate computation time per run. In addition we show the cummulative time spend in memcopy and waiting for messages, brown dots show the total overhead, i.e., subtracting computation time from the total runtime. This model and the prediction made by it can be used to make two decisions:

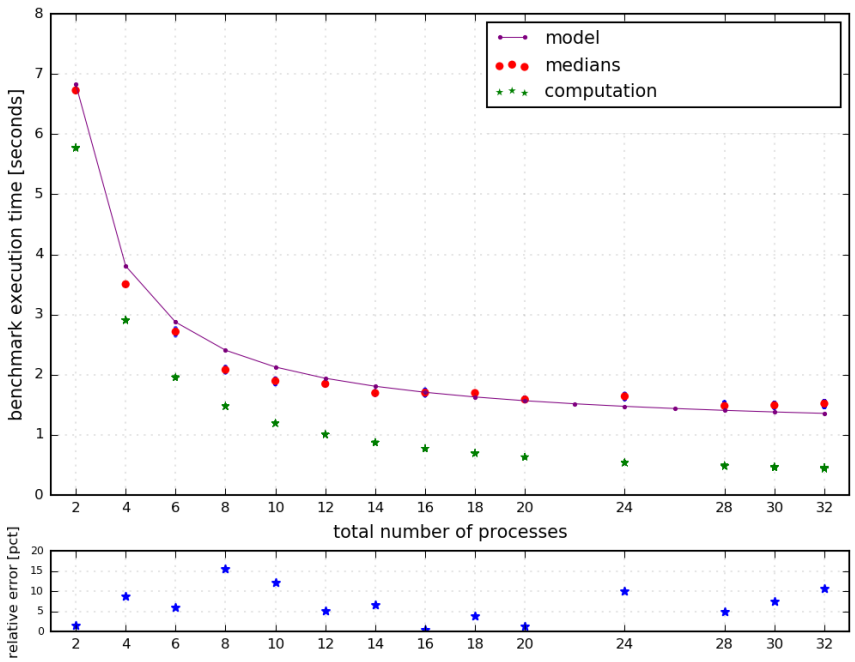

Figure 2: Performance model of the benchmark application employed for developing the virtualized model.

1. **Whether it will be beneficial to use a virtualized MPI implementation with a given user application.** Virtualized MPI implementations map MPI ranks to user-level threads rather than to operating-system-level processes in order to avoid paying too much overhead for each context switch. Therefore, virtualized MPI implementations require the calculation to be thread-safe, while MPI programs are usually not written that way because each MPI rank, usually being mapped to an OS-level process, is assumed to have its own private address space; hence, non-trivial development work will most likely be required to adapt an application to a virtualized MPI implementation, even given the existence of automated conversion tools (e.g., tools that automatically privatize global variables [ZNM+11, NZP+11]). For this reason, it is valuable to possess the ability to predict in advance whether or not these efforts would be justified by the benefits. A virtualized model can fulfill this need: if the virtualized model predicts, for a certain value of V, a running time lower than what can be achieved with a standard, non-virtualized MPI implementation, then it will be beneficial to use a virtualized MPI.

2. **Choice of V.** The other decision which is informed by the virtualized model is the choice of the optimal value of V. This is the value of V for which the model predicts the lowest running time (highlighted with circles in Figures 3 to 7). Such a decision is made after a code has been adapted to work with a virtualized MPI implementation. Without a model to inform this decision, the process of choosing V is entirely trial-and-error with repeated invocations of the program, which may be unsuitable if the calculation requires a large portion of a large machine at once.

The following two sections will describe the derivation of this model and discuss the potential for its improvement.

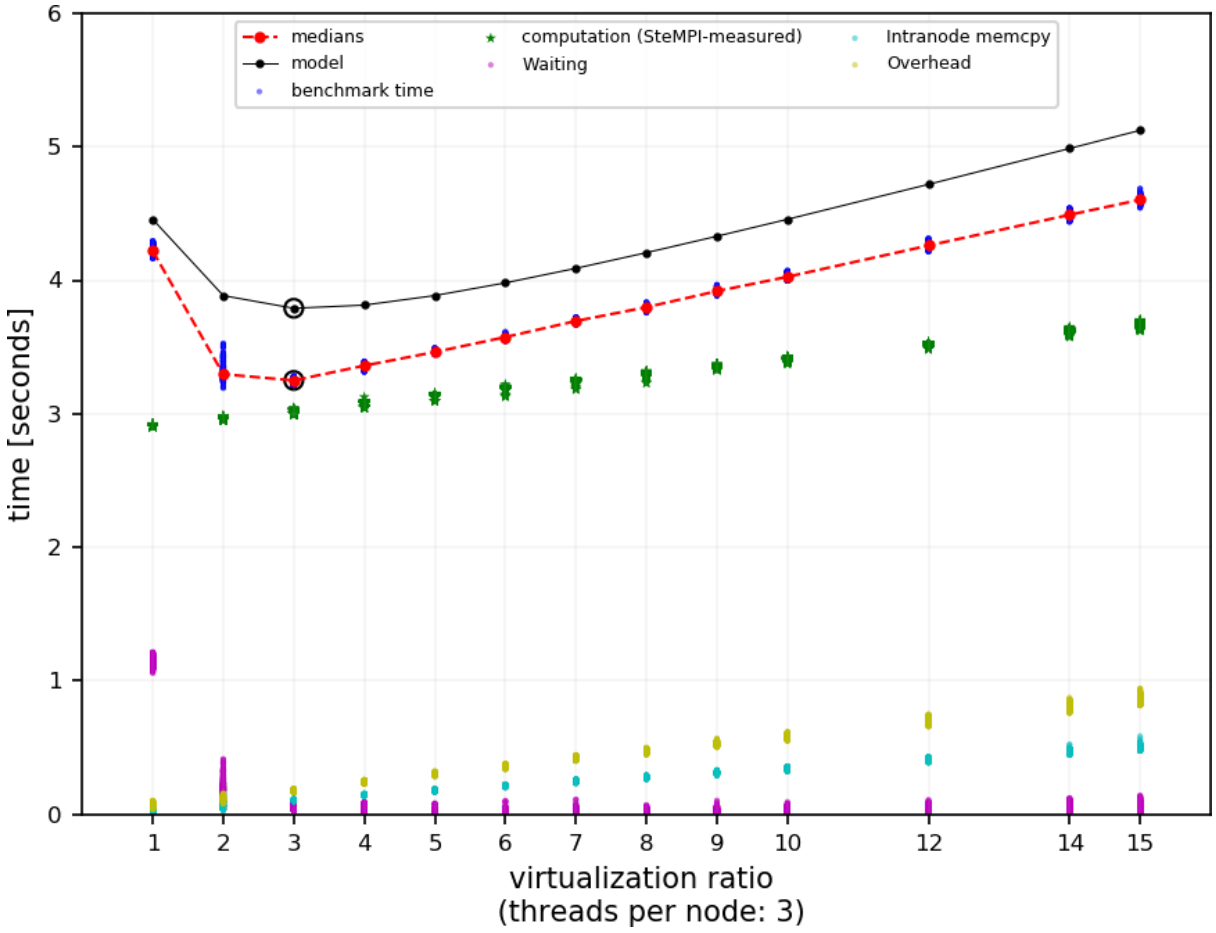

Figure 3: Fit of the proposed virtualized performance model. Using $C = 3$, the number of CPU cores per compute node used by the calculation. Within the plot, the virtualization ratio is varied (x axis), and the predicted and actual running times of the benchmark are plotted (y axis).

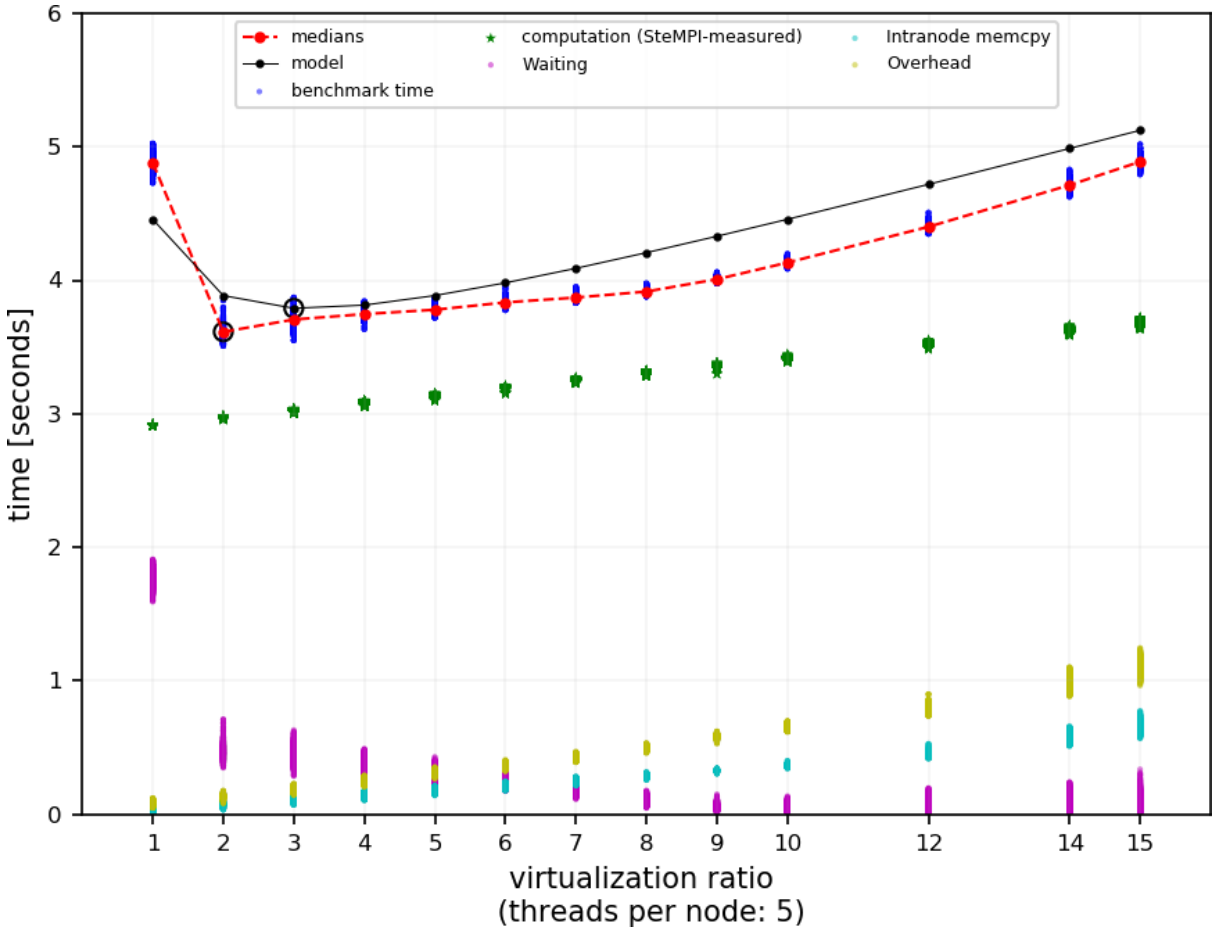

Figure 4: Fit of the proposed virtualized performance model. Using $C = 5$, the number of CPU cores per compute node used by the calculation. Within the plot, the virtualization ratio is varied (x axis), and the predicted and actual running times of the benchmark are plotted (y axis).

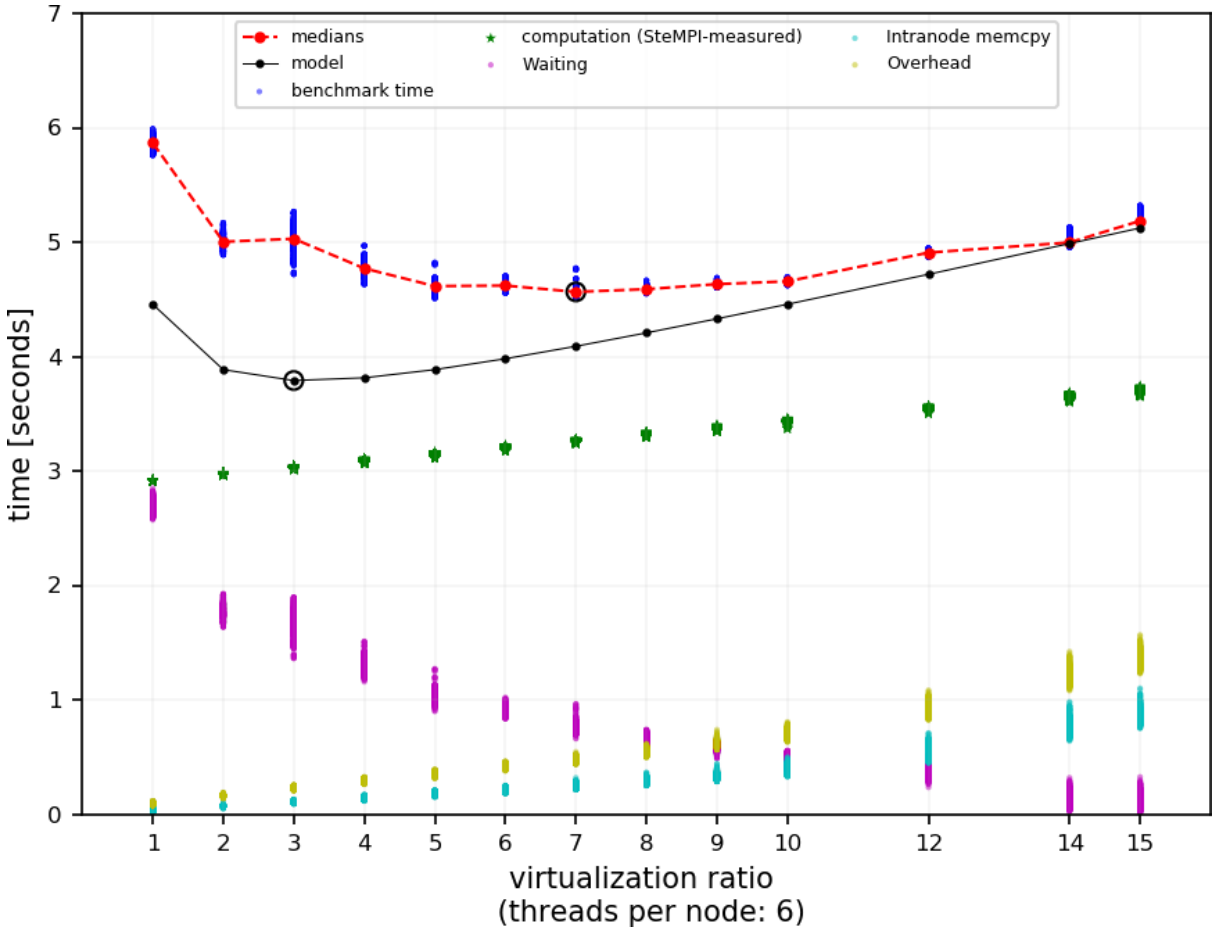

Figure 5: Fit of the proposed virtualized performance model. Using $C = 6$, the number of CPU cores per compute node used by the calculation. Within the plot, the virtualization ratio is varied (x axis), and the predicted and actual running times of the benchmark are plotted (y axis).

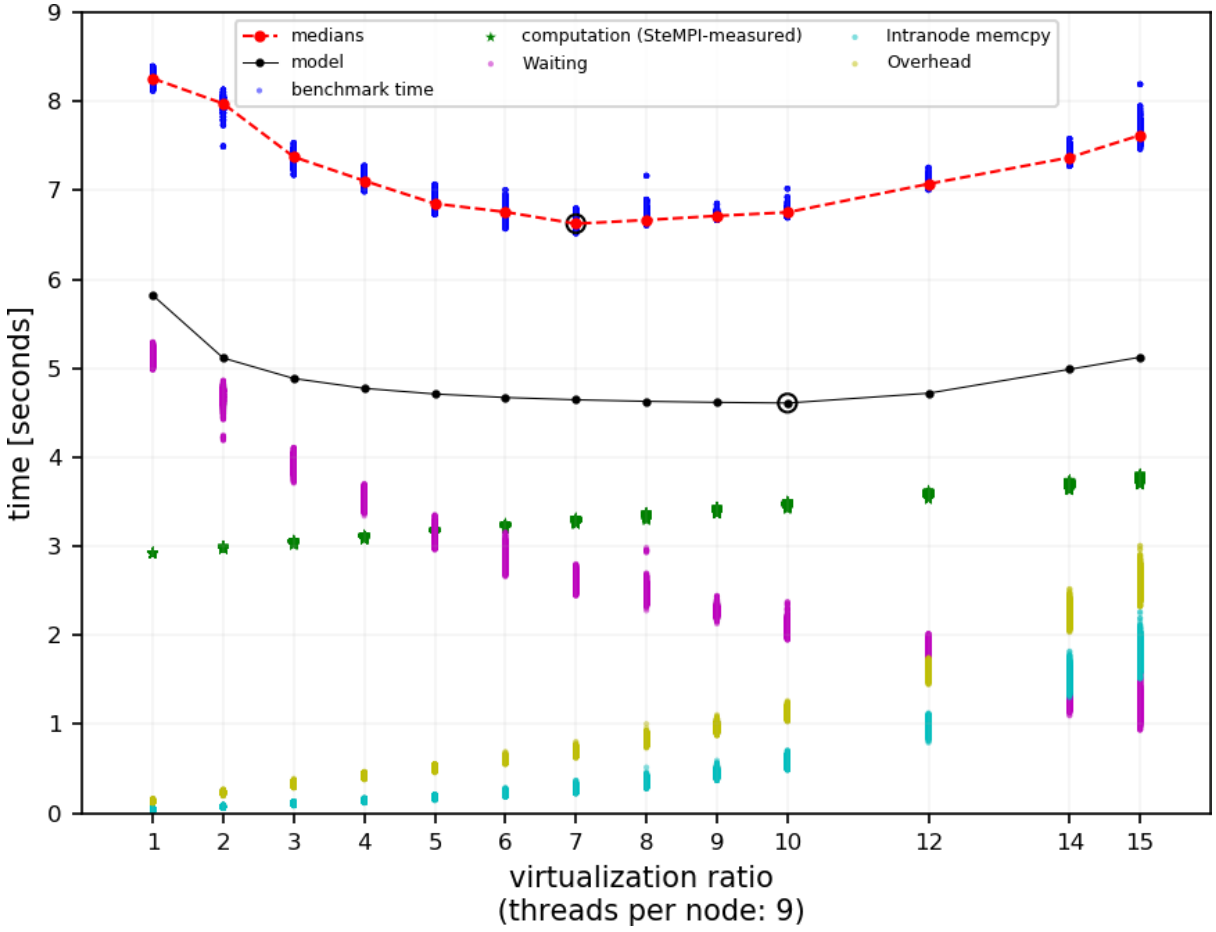

Figure 6: Fit of the proposed virtualized performance model. Using $C = 9$, the number of CPU cores per compute node used by the calculation. Within the plot, the virtualization ratio is varied (x axis), and the predicted and actual running times of the benchmark are plotted (y axis).

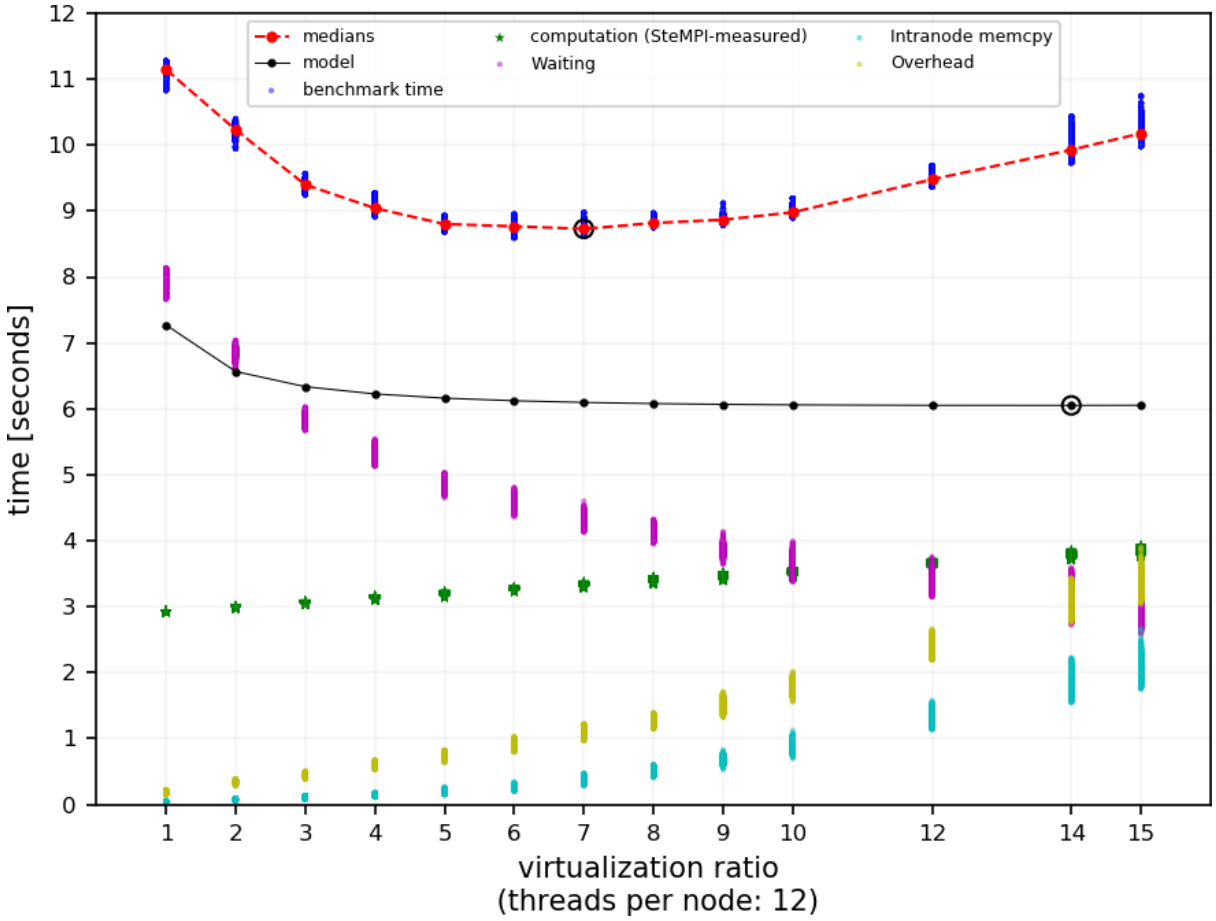

Figure 7: Fit of the proposed virtualized performance model. Using $C = 12$, the number of CPU cores per compute node used by the calculation. Within the plot, the virtualization ratio is varied (x axis), and the predicted and actual running times of the benchmark are plotted (y axis).

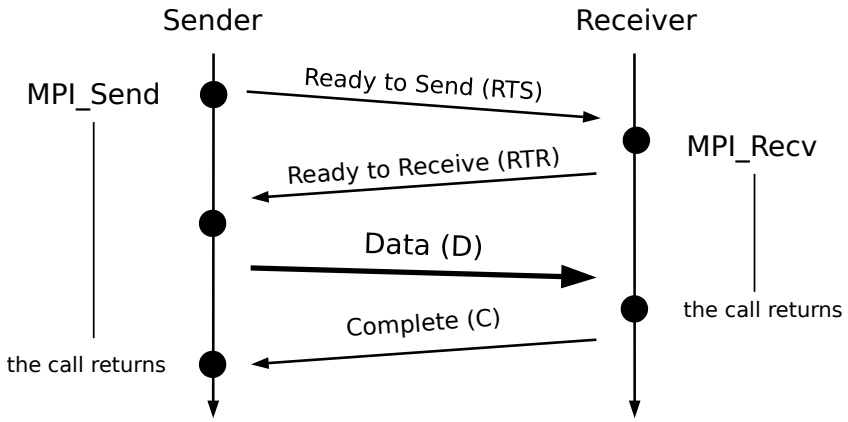

Figure 8: The flow of the MPI rendezvous protocol [WSB+06].

2.2.3 Derivation of the virtualized model

As V changes, the performance of the application changes too. The U-shape of both the prediction data and the actual data shows that V influences the performance both negatively and positively, depending on the value—large enough values of V actually cause a decline in performance. We can decompose this complex observed effect into factors which improve the performance and factors which decrease the performance. We will now list these factors, of which our model includes a subset, and discuss them.

Factors that improve performance (decrease overall running time):

- Improvement in computation–communication overlap.

Factors that worsen performance (increase overall running time):

- Increase in the volume of shared-memory communications.
- Increasing cumulative overhead of the virtualized MPI implementation: more communication calls and task-switching overhead.
- Replicated serial part of the application.

Improvement in computation–communication overlap (positive influence). Our virtualized model is based on capturing the effect of the improvement in achieved computation–communication overlap. We do this by modelling an ideal overlap and including an adjustment factor, which accounts for the non-optimality of overlap in practice.

The "$\max(cpu, net)$" term models the ideal overlap—if the CPU activity and the network activity are fully overlapped, then their combined running time equals the largest of the two, since the other is completely hidden. In practice, only a fraction of full overlap will be achieved, and this fraction should depend on V.

In the following we will be considering the rendezvous protocol of MPI [WSB+06], in which data transfer is preceded by a handshake between the communicating ranks. The protocol is summarized in Figure 8. The MPI implementation has to progress the protocol on several occasions: the sender has to handle an incoming ready-to-receive message, and the receiver has to handle an incoming ready-to-send message.

The model assumes that the imperfections of overlap come from the fact that CPU-activity bursts of a currently-computing rank prevent another rank from having its network communications being progressed by the CPU. In this case, the MPI implementation gets an opportunity to

progress the network protocol only in-between compute bursts. Therefore, if a ready-to-receive message arrives during a compute burst, it will only be handled when that burst finishes, which results in lost opportunity for computation–communication overlap. Assuming that a ready-to-receive message can arrive with equal probability throughout a compute burst, and taking into account that compute bursts cumulatively amount to $T_{comp}(N \cdot C \cdot V)$ units of time, the expected amount of lost overlap opportunity will amount to half of the total compute time, which is represented by the $\frac{1}{2} \cdot T_{comp}(N \cdot C \cdot V)$ term added on top of the max term that models the ideal full overlap.

Increase in the volume of shared-memory communications (negative influence). As is evident from the formula, shared-memory communications are considered to belong to CPU activity, which is because these are usually implemented via a memory copy performed by the CPU. That is why the T_{shmem} is added into the *cpu* term of the virtualized model.

As V increases, the share of the problem assigned to a particular compute node gets split into progressively smaller pieces, each of which is handled by a separate MPI rank, thus necessitating communication between them. This additional communication is not present in a non-virtualized configuration.

The shared-memory communication component of the base application performance model, T_{shmem} is multiplied by V in the virtualized model. Each CPU core executes V MPI ranks concurrently, one at a time, including the T_{shmem} component of each, which stack up to $V \cdot T_{shmem}$ on each CPU core.

Increasing cumulative overhead of MPI (negative influence). Related to the increasing volume of shared-memory communications, the number of communication calls also increases with increasing V. In addition to shared-memory communication calls, this also affects network communication calls. The larger the V is, the more communication calls to the MPI implementation are made. Naturally, each call incurs overhead, which may accumulate into a considerable amount if V reaches large values.

In our investigation, we have modelled this overhead, denoted T_{ovhd}, as a constant amount of time per communication call (`MPI_Isend` and `MPI_Irecv`), which was measured empirically. This cumulative overhead is added to the CPU portion of the virtualized model as the $V \cdot T_{ovhd} \cdot K$, where K is the number of communication calls, both shared-memory and network, made by each MPI rank during the whole duration of the program. This term was omitted from Equation 4 to avoid clutter. It modifies the *cpu* component of the model to be of the following form:

$$cpu = V \cdot T_{comp}(N \cdot C \cdot V) \; + \; V \cdot T_{shmem}(N \cdot C \cdot V) \; + \; V \cdot K \cdot T_{ovhd} \tag{5}$$

Replicated serial part of the application (negative influence). Adhering to the MPI programming model, each MPI rank effectively executes its own instance of the program, from the entry point and until termination. This means that the serial part of the algorithm, i.e., the share of the calculation which is not parallelized, is executed by each MPI rank. A virtualized MPI implementation runs V MPI ranks per CPU core, which results in V copies of the serial part of the calculation be executed by each CPU core. The serial phase may include reading data from a file or building a big data structure, which, if executed V times, may add up to a considerable amount of time. Therefore, it is important for the base performance model of the application to take this T_{serial} into account when formulating the computation-time model T_{comp}.

Now that we discussed the positive and negative influences, we use the discussed equations to model the computation time and the network time:

Modelling the computation time. The computation component of the base performance model, T_{comp}, is multiplied by V in the virtualized performance model. This stems from the fact that each CPU core executes V MPI ranks, each of which spends T_{comp} amount of time in computation. It is important to note that T_{comp} is expected to decrease with increasing V for a fixed global problem size—the more processes participate in a calculation over a fixed domain, the smaller share of it each process gets assigned—as is expected to be the case with all codes which decompose the domain.

Modelling the network time. The network component of the base model, T_{net}, is also multiplied by V in the virtualized model. All MPI ranks residing on a particular compute node will multiplex onto a single network-interface card (or other fixed quantity) in a traditional architecture. Our model assumes that network operations originating on a compute node do not get overlapped, which results in the network time being added up, hence the V factor. In our example, the contention for network bandwidth, both between NICs and from the CPU to the NIC, is handled by the base performance model (the maxrate model component of Equation 3).

2.2.4 Discussion of the virtualized model

Having described the derivation of the virtualized model, we return to Figures 3 to 7 to analyze how well the model fits the empirical data.

We can see that for low values of C the model is very close to measured data, and, importantly, the predicted data traces the shape of the empirical data accurately, which allows the minimum point of the U-shaped curve to be found exactly, except for the one-off error with $C = 5$. However, for the larger values of C the virtualized model underestimates the running time.

Ultimately, this is caused either by the virtualized model not taking all relevant effects into account and/or by the inaccuracy of the base performance model of the benchmark application. The virtualized model's inaccuracies come from the inaccurate approximation of the degree of achieved computation–communication overlap—the model overestimates it, resulting in smaller running time than is measured. In the parameter configurations where the virtualized model is inaccurate, network-communication time is greater than the CPU-activity time.

3 Conclusion and Future Work

In this report we have presented TinyMPI, a *virtualized* MPI implementation, and the results of the research effort of developing a performance model informing the choice of the *virtualization ratio* (the number of MPI ranks per core), the main parameter of a virtualized MPI implementation.

A virtualized MPI implementation aims to provide computation–communication overlap, as well as load balance and other benefits, seamlessly to the user's MPI application. The main distinguishing feature of such implementations is that they launch more than one MPI rank per CPU core, map them to user-level threads, and manage them completely without involving the operating system's kernel.

In the future we plan to make TinyMPI easier to use for end-users by providing better tooling to convert a traditional MPI codes, i.e., a compiler pass which automatically privatizes shared global variables and rewrites **MPI_*** calls to the appropriate **TMPI_*** ones. This will also make it easier to perform additional case studies, i.e., using different applications.

Acronyms and Abbreviations

- MPI – Message Passing Interface
- CPU – central processing unit
- ULT – user-level thread
- HPC – high-performance computing
- RDMA – remote direct memory access
- API – application programming interface
- NIC – network interface card
- OS – operating system

References

[GOS16] William Gropp, Luke N. Olson, and Philipp Samfass. Modeling MPI Communication Performance on SMP Nodes: Is It Time to Retire the Ping Pong Test. In *Proceedings of the 23rd European MPI Users' Group Meeting*, EuroMPI 2016, pages 41–50, New York, NY, USA, 2016. ACM.

[MB317] Intermediate report on enhancements to message passing. Deliverable D7.7 of the Montblanc-3 project, 2017.

[Mes15] Message Passing Interface Forum. Message-Passing Interface, 2015.

[NZP+11] Stas Negara, Gengbin Zheng, Kuo-Chuan Pan, Natasha Negara, Ralph E. Johnson, Laxmikant V. Kalé, and Paul M. Ricker. Automatic MPI to AMPI program transformation using photran. In Mario R. Guarracino, Frédéric Vivien, Jesper Larsson Träff, Mario Cannatoro, Marco Danelutto, Anders Hast, Francesca Perla, Andreas Knüpfer, Beniamino Di Martino, and Michael Alexander, editors, *Euro-Par 2010 Parallel Processing Workshops*, pages 531–539, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[WSB+06] Tim S. Woodall, Galen M. Shipman, George Bosilca, Richard L. Graham, and Arthur B. Maccabe. High performance RDMA protocols in HPC. In Bernd Mohr, Jesper Larsson Träff, Joachim Worringen, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 76–85, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[ZNM+11] G. Zheng, S. Negara, C. L. Mendes, L. V. Kale, and E. R. Rodrigues. Automatic handling of global variables for multi-threaded MPI programs. In *2011 IEEE 17th International Conference on Parallel and Distributed Systems*, pages 220–227, Dec 2011.