Towards Achieving Transparent Malleability Thanks to MPI Process Virtualization

Hugo Taboada[1,2(✉)], Romain Pereira[1,3], Julien Jaeger[1,2], and Jean-Baptiste Besnard[4]

[1] CEA, DAM, DIF, 91297 Arpajon, France
hugo.taboada@cea.fr
[2] Université Paris-Saclay, CEA, Laboratoire en Informatique Haute Performance pour le Calcul et la simulation, 91680 Bruyères-le-Châtel, France
[3] INRIA, EPI AVALON, ENS Lyon, LIP, Lyon, France
[4] ParaTools SAS, Bruyères-le-Châtel, France

Abstract. The field of High-Performance Computing is rapidly evolving, driven by the race for computing power and the emergence of new architectures. Despite these changes, the process of launching programs has remained largely unchanged, even with the rise of hybridization and accelerators. However, there is a need to express more complex deployments for parallel applications to enable more efficient use of these machines. In this paper, we propose a transparent way to express malleability within MPI applications. This process relies on MPI process virtualization, facilitated by a dedicated privatizing compiler and a user-level scheduler. With this framework, using the MPC thread-based MPI context, we demonstrate how code can mold its resources without any software changes, opening the door to transparent MPI malleability. After detailing the implementation and associated interface, we present performance results on representative applications.

Keywords: MPI · Malleability · Compiler

1 Introduction

The field of High-Performance Computing (HPC) is witnessing an increasing complexity in hardware, with hybridized architectures combining distributed and shared-memory parallelism along with accelerators. This requires programs to incorporate multiple programming models, potentially with diverse computing abstractions, to fully utilize the capabilities of the underlying computing substrate. This complexity has consequences both on the programs themselves and on their efficient utilization of the hardware. Programs in HPC are typically large simulation codes, often consisting of millions of lines of code. Therefore, adapting to new architectures is a planned and complex process. The chosen technology should be able to remain relevant for a sufficient duration to avoid

A. Bienz et al. (Eds.): ISC High Performance 2023 Workshops, LNCS 13999, pp. 28–41, 2023.
https://doi.org/10.1007/978-3-031-40843-4_3

tying the codebase to a particular technology or vendor, necessitating parallelism abstractions [4]. The process is often piece-wise, leading to incomplete parallelism and parallel limitations due to fork-join patterns. Parallel runtimes play a crucial role in abstracting hardware changes. However, due to increasing hardware constraints, low-level behaviours start to impact applications. OpenMP is successful in porting *existing* code to new architectures in a piece-wise manner by adding `#pragma parallel for`. However, the slower adoption of recent OpenMP advances suggests that this is less effective for tasks and OpenMP targets, despite the preservation of optional pragmas. On the MPI side, the programming model is deeply embedded in most of the HPC payloads due to its robustness and ability to constantly evolve to handle new hardware while providing bare-metal performance. However, this strength is also a weakness as it encloses parallelism expression in a relatively low-level and fine-grained form. The end-users are directly using what could have become the "assembly" of a higher-level abstraction.

Our paper highlights the imminent complexity barrier that HPC is facing due to architectural evolutions and technical debt accumulated by HPC applications. Although runtimes have evolved and new abstractions have been provided, they often do not translate to applications [5,19]. The empirical nature of the usage of supercomputers, driven by domain-specific scientists who are mainly interested in their results and given virtually unlimited computing power with little or no quota, has resulted in heuristics of doing good science that prioritize final results over efficient use of parallel machines. However, with the rise of GPUs and multi-level parallelism, the need for efficient use of parallel machines is more pressing than ever. Ideally, the solution would be to rewrite applications using new abstractions to address new hardware. However, this requires teaching domain-specific scientists how to code in parallel abstractions or providing them with tools that hide the complexity of parallelism. Several languages, including domain-specific ones, have tried to define more expressive ways of parallelizing programs [4,17,29]. Acknowledging the existence of inefficient (by design) and potentially transitively inefficient parts of parallel programs, the question of malleability then becomes preponderant.

2 Fast is Not Efficient

Time to result is not a correct heuristic to define how efficient a computation is. Indeed, from the well-known parallel efficiency formula, one may add computing resources to a computation while contributing only at the margin to the final result. As a consequence, sometimes a slower run is making much better use of the underlying hardware while leaving space for other runs on the freed resources [3, 10]. Malleability is the systematic and automated application of such reasoned running configuration for a given heuristic. And for more optimized applications, co-scheduling also provides opportunities for resource sharing for example during alternating phases of computing or to cope with load imbalance. In this work, we propose to implement transparent application co-location using the MPC thread-

based MPI and a specialized compiler enabling the conversion of MPI+OpenMP applications to threads.

3 The Multi-processor Computing Runtime

MPC is a thread-based implementation of MPI, where MPI processes run in threads instead of traditional UNIX processes. This approach has various benefits, including reduced memory usage, faster launch times, and improved intra-node messaging (see Sect. 6). Moreover, MPC has its own implementation of OpenMP and a user-level scheduler that is necessary for transparently running programs in co-routines, as discussed in Sect. 3.2. Additionally, MPC provides a specialized compiler that facilitates the porting of regular MPI+OpenMP codes to a thread-based context.

3.1 MPC Compiler Additions: Privatization

The process of converting global variables to Thread-Local Storage (TLS) variables using MPC's privatizing compiler [6] is crucial to enable the thread-based execution of regular MPI programs inside threads. This is necessary because global variables can cause conflicts when loaded in the same address space. The MPC privatizing compiler converts these global variables to MPC's TLS variables, which enables loading the same program multiple times in a single address space without any conflicts. The privatizing compiler is based on a modified version of GCC and includes a pass that converts global variables and other TLS levels (such as OpenMP thread-locals) to MPC's TLS. This pass has been implemented in a component called libextls (extended TLS) [6].

After a program has been privatized, it can be executed in the context of a thread, initially to meet the needs of a thread-based MPI. However, as we will discuss later in this paper, we aim to expand this concept to hybrid computing, where various executables are loaded in the same address space. In such a setup, the MPC runtime (MPI and OpenMP) can better address performance differences between the different applications.

3.2 User-Level Threads

Once MPI processes are threads, it becomes interesting to consider their execution inside user-level threads. This is a convenient way to hide waiting time, replacing it with context switches [31]. When a *task* is delayed, the resources can be used productively by moving to another task, instead of remaining in a busy loop. MPC is built around the concept of co-routines, and to make this approach viable, it is crucial to not only privatize threads but also to wrap the complete Pthread interface to capture potentially blocking calls, redirecting them to the scheduling loop inside MPC. Otherwise, a lock could be held by a pending task. Using MPC's *m:n* scheduler, it is already possible to run two programs on the

same resources in an oversubscribed fashion, meaning that the two programs will alternate on the underlying Pthread (co-routines).

By extending this idea to malleable programs, two programs can run on the same resources [25]. In addition, since MPI programs are now running inside threads, it is possible to move them on the node, changing their affinities. This is analogous to changing the target pending list in the scheduler without having to handle data dependencies, which can be complex in the case of regular MPI processes, but trivial in shared-memory. This extended support for threads and the user-level scheduler provides opportunities for fine-grained control over cross-scheduling of runtimes [13,18,32].

3.3 OpenMP Runtime

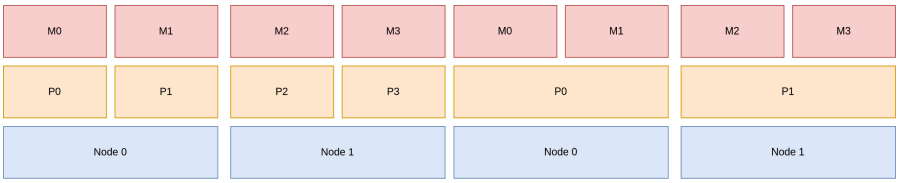
The MPC framework includes an OpenMP runtime that utilizes the same user-level thread and privatization infrastructures as the MPI runtime. This runtime is designed to be compatible with both GOMP and Intel/LLVM ABIs, enabling it to run OpenMP programs generated by different compilers. One significant advantage of co-locating both OpenMP and MPI threads in the same scheduler is the potential for runtime-stacking [11,28]. This refers to the idea of combining multiple runtimes for improved scheduling. Recent research has shown that it is possible to convert MPI waiting time into OpenMP task progress [32]. We aim to utilize this feature, in conjunction with privatization, in our co-scheduling approach, which relies on MPI process virtualization.

4 MPI Process Virtualization

MPI process virtualization is the process of converting MPI processes into threads within compute nodes [6,31]. This approach allows for new scheduling opportunities for parallel applications by implementing cross-job malleability at the node level, instead of relying on a single scheduler. In this section, we will provide an overview of this concept. We will discuss how it involves using a node-level scheduler, specifically MPC, to enable job malleability, resource sharing, and mitigation. Additionally, we will explore the compilation aspect of gathering multiple programs in the same binary and the executive part of leveraging OpenMP tasking, building on previous work [32], to optimize performance.

4.1 The Need for Node-Level Schedulers

As shown in Fig. 1a, a traditional HPC scheduler allocates resources based on UNIX processes. Typically, there is a one-to-one correspondence between UNIX processes and MPI processes. In contrast, our approach using MPC involves colocating and co-scheduling multiple programs within a single address space on each node, as illustrated in Fig. 1b. One advantage of this approach is that it requires fewer processes to run a given program. This is because there is no longer a direct mapping between UNIX and MPI processes; MPI processes



(a) Process-based MPI. (b) Virtualized MPI processes.

Fig. 1. Comparison between regular scheduling approach and virtualized MPI processes.

are now threads. Furthermore, the work we present in this paper has indirect implications for malleability pivoting with a centralized scheduler (e.g., using the Process Management Interface or PMIx), as it can now be partially a local scheduling decision that is fully dynamic and arbitrary. Our two-level scheduling approach can leand to multiple profitable scenarios that have been explored in the literature:

- **Resource sharing** [1,12,16,22,24,30]: two applications residing in the same address space can dynamically exchange their resources without the need for copying or remapping of data. This allows the underlying scheduler to balance the compute resources from all collocated applications without complex code adaptation. For instance, two applications that require linear algebra computations on the node could have their respective tasks scheduled on the GPUs with a global view of the system, potentially achieving higher utilization than when allocating GPUs to each program separately.
- **Overprovisioning** [8,21,25,36]: in this configuration we run multiple applications per core, forcing them to alternate on the resources. In this case, the waiting time is directly recovered by the yield mechanism of the user-level scheduler, hiding latency. This case is of course bound to the availability of enough memory on the node.
- **Specialization** [14,15,35]: this opens the way to define in-process services providing facilities such as I/O to all programs running on the node in a very efficient manner (true zero-copy). Doing so may open the way to machines with smaller service islands, I/O being part of the job themselves with dedicated resources. This provides strong advantages to in-situ scenarios.

Overall, the main advantage of this two-level scheduler model is that scheduling decisions at the node level is now local, and there is no need to rely on an external component, except when more resources are needed. In the context of this work, we consider that we have colocated multiple applications in the same address space, and they now share their CPUs. Moving an application is now equivalent to moving a simple thread.

Therefore, as shown in Fig. 2, malleability inside the node becomes practical for any MPI+OpenMP application without requiring specific porting or modification. This is due to MPC's privatizing compiler that ensures applications

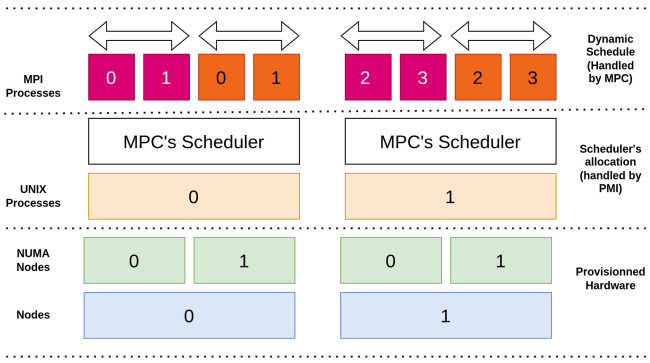


Fig. 2. Illustation of node-level malleability in the context of virtualized MPI processes.

can run inside threads, and we can simply adjust the resources allocated to each of these applications while sharing all the runtime components between the two programs. In the next section, we will describe how an application can be converted to a virtualized program inside MPC.

4.2 Virtualizing Programs

The process of virtualizing an existing program is very similar to the one done inside MPC for privatization purposes, which involves running the same program image multiple times in a single process [6]. It consists of compiling the program with a modified compiler that converts global variables and thread-local variables to the extended TLS hierarchy of MPC, which features an additional "process" level that matches each MPI process in the final runtime memory image. This process preserves TLS optimizations [6]. As shown in Fig. 1b, once this is done, and because the `main` function of the application has been renamed using a preprocessor directive, MPC simply launches N threads with their respective pinning, TLS context, and ranks to mimic the execution of the process in MPI, except that it is now in threads. This process has been reliably used by MPC to run C, Fortran, and C++ applications, with C++ being the most complex due to potential dynamic initializers, global objects, and templates. The idea of virtualizing MPI processes is not new and has a long research history [20,25,31]. AMPI [31], FG-MPI [26] and the Charm++ [24] runtimes have demonstrated many benefits of virtualized MPI processes, such as load-balancing [31] and dynamicity [23]. The closest related work to the idea we present in this paper is Adaptive Jobs [23]. This work moves the scheduler inside virtualized MPI processes and proposes to allocate programs in these address spaces. Conversely, in our approach, we stay much closer to the existing parallelism interfaces and runtimes, with the goal of transparently running *existing* programs in a virtualized configuration. This is only possible because MPC features its own OpenMP implementation and a Pthread interface to redirect locking calls to the user-

level scheduler. When multiple programs are living in the same address space, there are opportunities for *runtime stacking*, and it is this aspect that we aim to highlight in this paper.

Returning to our initial goal of generalizing the approach to multiple programs, we needed to load multiple privatized programs in the same address space. To achieve this, we compiled the codes with the `-fpic -rdynamic -shared` flags, making the code position independent (as shared libraries) and exposing symbols in the final binary (and therefore the main). With this done, we were able to dynamically open binaries and their shared library dependencies with the help of the loader. The programs now run inside the same process, and we eventually replace their `MPI_COMM_WORLD` with a subset of the actual world, matching the actual program splitting. As MPC is also an MPI runtime, doing so is straightforward, as it simply involves replacing the communicator in the MPI process context. Note that this last step can become even more transparent using the session model, in which the application builds its `COMM_WORLD`, enabling explicit redirection. There are of course security issues when colocating programs in a single address space, and therefore such a scenario can only be envisioned for binaries belonging to the same user, and not as a way to run general programs in such a configuration. Memory segmentation from the Operating System and the zero-page mechanism are essential security measures to ensure perfect impermeability in between address spaces. Besides, signals are also handled by the controlling process and therefore, events such as `SIGINT` cannot be handled individually in such a configuration.

4.3 Co-scheduling Multiple Programs in a Single Process

Thanks to the nature of the MPC thread-based MPI+OpenMP runtime, multiple programs can run within a single UNIX process. Now comes the question of *co-scheduling* [2] each program to guarantee efficient progression of each application. The first possibility is through cores oversubscription with kernel threads (e.g., *pthreads*), letting the operating system preemptively schedule threads of each program. With n cores and $m > n$ threads, this is the *m:n* co-scheduling approach. Oversubscription has been widely adopted in runtimes for overlapping synchronization idleness on cores, on AMPI [31] or LLVM OpenMP [34] for instance. Yet, it can degrade performances when a high number of asynchronous operations concurrently progresses [21,36], one source of overheads being the operating system [7,33]. The second possibility is an *n:n* approach: each core is assigned a single kernel thread, and the scheduling decision is taken by the MPC user space scheduler. In particular, recent work on the MPC OpenMP *task* scheduler has shown improvement over the LLVM *m:n* approach for target tasks offloading [18].

The final goal of our work, which we are yet to reach, is to deploy a co-scheduling approach as depicted on Listing 1.1. Line 1 to 6 corresponds to program dynamic loading, line 10 the *n:n* threads binding is created. From lines 11 to 19, each program starts execution within an OpenMP task with its own *fiber*, which is an MPC extension hinting at the OpenMP runtime to execute tasks

on their stack, enabling more scheduling flexibility. Descendant MPI processes, OpenMP parallel regions, and tasks created by each program will then run as part of a single MPC OpenMP task scheduler responsible for co-scheduling. Note that our execution model assumes tasks can migrate between *teams* which is not compliant with current OpenMP specifications and therefore still requires dedicated work. Recent work on Free Agent Threads [27] provided a solution to this limitation, improving the malleability of OpenMP tasking. Our execution model also relies on runtimes interoperability [32] so that a task blocking within an MPI call is automatically preempted instead of retaining its core.

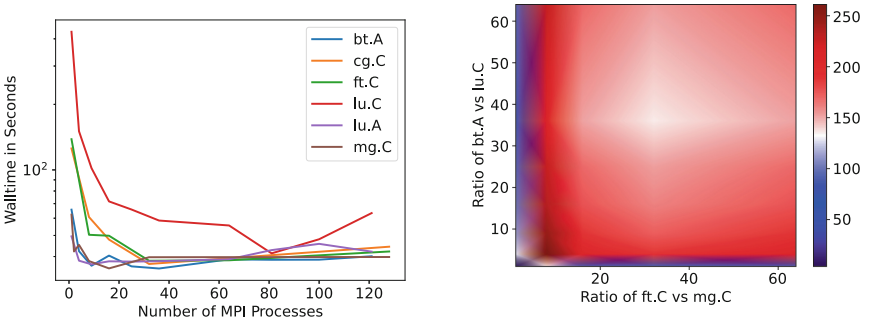
Listing 1.1. Pseudo-code of the launcher wrapper

```
1  extern struct program_t {
2      int argc;
3      char ** argv;
4      int (*main)(int, char **);
5  } * programs;
6  extern int n_programs;
7
8  int main(void)
9  {
10     # pragma omp parallel
11     # pragma omp single
12     {
13         for (int i = 0 ; i < n_programs ; ++i)
14         {
15             program_t * p = programs + i;
16             #pragma omp task fiber
17                 p->main(p->argc, p->argv);
18         }
19     }
20     return 0;
21 }
```

5 Co-scheduling Experiment

In this section, we present the results of our initial experiments, which involved running applications on the same node in order to identify optimal execution times. The purpose of this setup was to identify potential "sweet spots" for execution time, which could eventually become more dynamic as shared-memory models used within applications become more flexible, particularly in terms of their ability to be invasive. However, it is important to note that the results presented here are limited by the static nature of the allocation used in our experiments.

In Fig. 3a, we show the scalability of the various kernels, which have different problem sizes and exhibit different strong scaling behavior. The smaller kernels exhaust their parallelism more quickly than the larger ones. This behavior is



(a) Scalability of the individual kernels on the machine.

(b) Combination of four programs bt.A, lu.C, ft.C and mg.C over the machine (walltimes)

Fig. 3. Walltime for various resource breakdowns for co-located NAS MPI benchmarks.

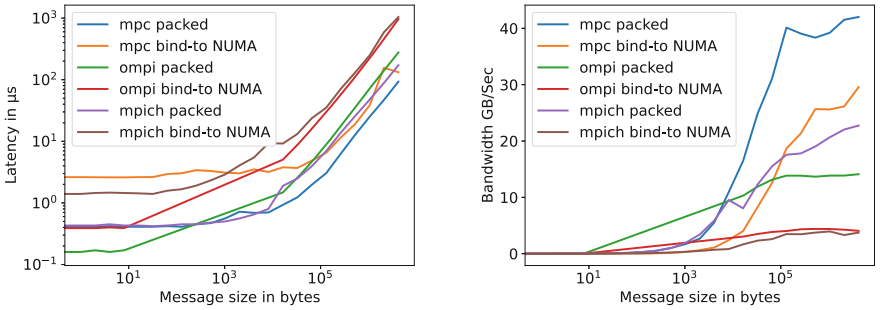
also present in the individual nodes of larger simulations, particularly when considering dynamic behavior over time.

To further investigate co-location, we selected four of these benchmarks (bt.A, lu.C, ft.C, and mg.C) and ran them on a single node. As expected from the scaling behavior, there are sweet spots of scheduling for these four programs, which are indicated by the lower values in Fig. 3b. This suggests that dynamically reshaping jobs over time to follow these sweet spots could lead to improved performance. However, this can be difficult to accomplish through models alone, which is why our future work will focus on developing a node-level shared runtime to support malleability.

Co-locating programs is not a new idea, and several MPI runtimes and experiments dedicated to malleability have demonstrated similar gains to the ones we present here [2,9,24,30]. However, one of the main motivations for our work is that MPC can convert existing programs to shared-address space transparently, while still supporting regular MPI and OpenMP. This simplifies the process of making programs malleable and expands the range of applications that can benefit from this approach.

6 Networking Performance

As a thread-based MPI, MPC is capable of exchanging MPI messages between MPI Processes (now threads) running in the same process. In addition, it also supports all intermediate configurations, including a regular process-based execution. Besides, MPC has support for standard HPC networks, including SHM, Infiniband, Portals 4 BXI [?] interconnect and libFabric. It means co-scheduled programs can run distributed as regular MPI applications, malleability would only occur at the level of nodes, in between colocated programs thanks to the shared runtimes (mostly OpenMP for compute).



(a) Latency for IMB-MPI1 Ping-Pong.

(b) Bandwidth for IMB-MPI1 Ping-Pong.

Fig. 4. Comparison of MPC, OpenMPI and MPICH over various biding configuration using the IMB-MPI1 Ping-Pong benchmark.

Figure 4 presents the performance results of running the Intel IMB-MPI1 Ping-Pong benchmark between two processes co-located on the same node. This simple benchmark allows us to assess the relative bandwidth and latency performance between co-located programs. The tests were conducted on a bi-socket AMD EPYC 7413 machine with a total of 48 cores. The "packed" configuration means processes are running on nearby cores, while the "scattered" configuration means processes are spread across the two NUMA nodes. We measured the performance using MPC 4.1.0, OpenMPI 4.1.5, and MPICH 4.1, all of which were installed through spack using the default configuration flags.

In Fig. 4a, we observe that MPC has lower latencies for larger messages. However, for smaller messages, other runtimes, particularly OpenMPI, can perform better. The main reason for this is that in MPC, the `THREAD_MULTIPLE` support is always on, due to its thread-based nature, which leads to locking requirements and impacts the micro-optimizations needed to reach performance levels comparable to OpenMPI. However, thanks mostly to the improved bandwidth as messages are moved through direct `memcpy` (as shown in Fig. 4b), MPC yields lower latencies for larger messages. Other runtimes may use SHM fragments or more optimal Cross-Memory Attach (CMA) to transfer such large buffers, which are less direct than a regular *memcpy*.

Overall, we have demonstrated that running programs in shared memory with MPC can result in improved messaging bandwidth in some cases. This improvement is more pronounced for larger messages, but smaller messages still suffer from locking overhead. To address this limitation, we are currently in the process of fully rewriting the networking stack in the MPC runtime. Our goal is to improve performance and reduce overhead for all message sizes, further enhancing the benefits of running programs in shared memory with MPC.

7 Conclusion

High-performance computing architectures are evolving towards increased flexibility in utilizing node-level resources. However, shared-memory resources are often underutilized due to a lack of collaboration between programming models. For instance, MPI may get stuck in a busy-wait loop while OpenMP may have pending tasks. To address this issue, we utilized the virtualization capabilities of the MPC thread-based MPI + OpenMP runtime to transparently port MPI applications to a space where they can be adapted to shared memory with less effort. This process is analogous to what MPC does for MPI programs, utilizing its Extended TLS model. In this work, we presented two types of co-scheduling: oversubscription to eliminate busy waiting by using context yields and a unified scheduler that balances computing on a whole node, including between different virtualized programs. We also showcased the OpenMP code that we plan to use to achieve this.

We then conducted co-scheduling experiments that showed the potential for dynamic scheduling at the node level to compensate for imbalance and inefficiencies by opening a common task-oriented scheduler to all runtimes. Eventually, we have shown that by running in shared memory there were opportunities to improve MPI messaging performance. Going forward, we plan to further develop this work on the OpenMP side, seeking a practical way to define node-wide task stealing, which is currently not feasible between different parallel regions. This would enable OpenMP programs co-located in the same node (and virtualized in the same process) to dynamically balance their computations.

References

1. Acun, B., et al.: Parallel programming with migratable objects: Charm++ in practice. In: SC 2014: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 647–658. IEEE (2014)
2. Aguilar Mena, J., Shaaban, O., Beltran, V., Carpenter, P., Ayguade, E., Labarta Mancho, J.: OmpSs-2@Cluster: distributed memory execution of nested OpenMP-style tasks. In: Cano, J., Trinder, P. (eds.) Euro-Par 2022. LNCS, vol. 13440, pp. 319–334. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-12597-3_20
3. Arima, E., Comprés, A.I., Schulz, M.: On the convergence of malleability and the HPC PowerStack: exploiting dynamism in over-provisioned and power-constrained HPC systems. In: Anzt, H., Bienz, A., Luszczek, P., Baboulin, M. (eds.) ISC High Performance 2022. LNCS, vol. 13387, pp. 206–217. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-23220-6_14
4. Beckingsale, D.A., et al.: RAJA: portable performance for large-scale scientific applications. In: 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), pp. 71–81. IEEE (2019)
5. Bernholdt, D.E., et al.: A survey of MPI usage in the us exascale computing project. Concurr. Comput. Pract. Exp. **32**(3), e4851 (2020)
6. Besnard, J.B., et al.: Introducing task-containers as an alternative to runtime-stacking. In: Proceedings of the 23rd European MPI Users' Group Meeting, pp. 51–63 (2016)

7. Bierbaum, J., Planeta, M., Hartig, H.: Towards efficient oversubscription: on the cost and benefit of event-based communication in MPI. In: 2022 IEEE/ACM International Workshop on Runtime and Operating Systems for Supercomputers (ROSS), Los Alamitos, CA, USA, pp. 1–10. IEEE Computer Society (2022). https://doi.org/10.1109/ROSS56639.2022.00007. https://doi.ieeecomputersociety.org/10.1109/ROSS56639.2022.00007
8. Bierbaum, J., Planeta, M., Härtig, H.: Towards efficient oversubscription: on the cost and benefit of event-based communication in MPI. In: 2022 IEEE/ACM International Workshop on Runtime and Operating Systems for Supercomputers (ROSS), pp. 1–10. IEEE (2022)
9. Bungartz, H.J., Riesinger, C., Schreiber, M., Snelting, G., Zwinkau, A.: Invasive computing in HPC with X10. In: Proceedings of the Third ACM SIGPLAN X10 Workshop, pp. 12–19 (2013)
10. Cantalupo, C., et al.: A strawman for an HPC PowerStack. Technical report, Intel Corporation, United States; Lawrence Livermore National Lab. (LLNL) (2018)
11. Carribault, P., Pérache, M., Jourdren, H.: Enabling low-overhead hybrid MPI/OpenMP parallelism with MPC. In: Sato, M., Hanawa, T., Müller, M.S., Chapman, B.M., de Supinski, B.R. (eds.) IWOMP 2010. LNCS, vol. 6132, pp. 1–14. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13217-9_1
12. Cores, I., González, P., Jeannot, E., Martín, M.J., Rodríguez, G.: An application-level solution for the dynamic reconfiguration of MPI applications. In: Dutra, I., Camacho, R., Barbosa, J., Marques, O. (eds.) VECPAR 2016. LNCS, vol. 10150, pp. 191–205. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-61982-8_18
13. Dionisi, T., Bouhrour, S., Jaeger, J., Carribault, P., Pérache, M.: Enhancing load-balancing of MPI applications with workshare. In: Sousa, L., Roma, N., Tomás, P. (eds.) Euro-Par 2021. LNCS, vol. 12820, pp. 466–481. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-85665-6_29
14. Dorier, M., Dreher, M., Peterka, T., Wozniak, J.M., Antoniu, G., Raffin, B.: Lessons learned from building in situ coupling frameworks. In: Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization, pp. 19–24 (2015)
15. Duro, F.R., Blas, J.G., Isaila, F., Carretero, J., Wozniak, J., Ross, R.: Exploiting data locality in Swift/T workflows using Hercules. In: Proceedings of NESUS Workshop (2014)
16. El Maghraoui, K., Desell, T.J., Szymanski, B.K., Varela, C.A.: Dynamic malleability in iterative MPI applications. In: Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2007), pp. 591–598. IEEE (2007)
17. Fanfarillo, A., Burnus, T., Cardellini, V., Filippone, S., Nagle, D., Rouson, D.: OpenCoarrays: open-source transport layers supporting coarray Fortran compilers. In: Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models, pp. 1–11 (2014)
18. Ferat, M., Pereira, R., Roussel, A., Carribault, P., Steffenel, L.A., Gautier, T.: Enhancing MPI+OpenMP task based applications for heterogeneous architectures with GPU support. In: Klemm, M., de Supinski, B.R., Klinkenberg, J., Neth, B. (eds.) IWOMP 2022. LNCS, vol. 13527, pp. 3–16. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-15922-0_1
19. Hori, A., et al.: An international survey on MPI users. Parallel Comput. **108**, 102853 (2021)

20. Hori, A., et al.: Process-in-process: techniques for practical address-space sharing. In: Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing, pp. 131–143 (2018)
21. Iancu, C., Hofmeyr, S., Blagojević, F., Zheng, Y.: Oversubscription on multicore processors. In: 2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS), pp. 1–11. IEEE (2010)
22. Iserte, S., Mayo, R., Quintana-Orti, E.S., Pena, A.J.: DMRlib: easy-coding and efficient resource management for job malleability. IEEE Trans. Comput. **70**(9), 1443–1457 (2020)
23. Kalé, L.V., Kumar, S., DeSouza, J.: A malleable-job system for timeshared parallel machines. In: 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID 2002), pp. 230–230. IEEE (2002)
24. Kale, L.V., Zheng, G.: Charm++ and AMPI: adaptive runtime strategies via migratable objects. In: Advanced Computational Infrastructures for Parallel and Distributed Applications, pp. 265–282 (2009)
25. Kamal, H., Wagner, A.: Added concurrency to improve MPI performance on multicore. In: 2012 41st International Conference on Parallel Processing, pp. 229–238. IEEE (2012)
26. Kamal, H., Wagner, A.: FG-MPI: Fine-grain MPI for multicore and clusters. In: 2010 IEEE International Symposium on Parallel & Distributed Processing Workshops and Phd Forum (IPDPSW), pp. 1–8 (2010). https://doi.org/10.1109/IPDPSW.2010.5470773
27. Lopez, V., Criado, J., Peñacoba, R., Ferrer, R., Teruel, X., Garcia-Gasulla, M.: An OpenMP free agent threads implementation. In: McIntosh-Smith, S., de Supinski, B.R., Klinkenberg, J. (eds.) IWOMP 2021. LNCS, vol. 12870, pp. 211–225. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-85262-7_15
28. Loussert, A., Welterlen, B., Carribault, P., Jaeger, J., Pérache, M., Namyst, R.: Resource-management study in HPC runtime-stacking context. In: 2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), pp. 177–184. IEEE (2017)
29. Marowka, A.: On the performance portability of OpenACC, OpenMP, Kokkos and RAJA. In: International Conference on High Performance Computing in Asia-Pacific Region, pp. 103–114 (2022)
30. Martín, G., Marinescu, M.-C., Singh, D.E., Carretero, J.: FLEX-MPI: an MPI extension for supporting dynamic load balancing on heterogeneous non-dedicated systems. In: Wolf, F., Mohr, B., an Mey, D. (eds.) Euro-Par 2013. LNCS, vol. 8097, pp. 138–149. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40047-6_16
31. Pei, Y., Bosilca, G., Yamazaki, I., Ida, A., Dongarra, J.: Evaluation of programming models to address load imbalance on distributed multi-core CPUs: a case study with block low-rank factorization. In: 2019 IEEE/ACM Parallel Applications Workshop, Alternatives To MPI (PAW-ATM), pp. 25–36 (2019). https://doi.org/10.1109/PAW-ATM49560.2019.00008
32. Pereira, R., Roussel, A., Carribault, P., Gautier, T.: Communication-aware task scheduling strategy in hybrid MPI+OpenMP applications. In: McIntosh-Smith, S., de Supinski, B.R., Klinkenberg, J. (eds.) IWOMP 2021. LNCS, vol. 12870, pp. 197–210. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-85262-7_14
33. Radojkovic, P., et al.: Measuring operating system overhead on CMT processors. In: 2008 20th International Symposium on Computer Architecture and High Performance Computing, pp. 133–140 (2008). https://doi.org/10.1109/SBAC-PAD.2008.19

34. Tian, S., Doerfert, J., Chapman, B.: Concurrent execution of deferred OpenMP target tasks with hidden helper threads. In: Chapman, B., Moreira, J. (eds.) LCPC 2020. LNCS, vol. 13149, pp. 41–56. Springer International Publishing, Cham (2022). https://doi.org/10.1007/978-3-030-95953-1_4
35. Vef, M.A., et al.: GekkoFS-a temporary distributed file system for HPC applications. In: 2018 IEEE International Conference on Cluster Computing (CLUSTER), pp. 319–324. IEEE (2018)
36. Wende, F., Steinke, T., Reinefeld, A.: The impact of process placement and oversubscription on application performance: a case study for exascale computing (2015)