Task-Based Performance Portability in HPC

Maximising long-term investments in a fast evolving, complex and heterogeneous HPC landscape

White Paper

Olivier Aumage
Paul Carpenter
Siegfried Benkner

etp4hpc.eu

05/10/2021

@etp4hpc

ETP 4 HPC

EUROPEAN TECHNOLOGY PLATFORM FOR HIGH PERFORMANCE COMPUTING

Introduction

As HPC hardware continues to evolve and diversify and workloads become more dynamic and complex, applications need to be expressed in a way that facilitates high performance across a range of hardware and situations. The main application code should be platform-independent, malleable and asynchronous with an open, clean, stable and dependable interface between the higher levels of the application, library or programming model and the kernels and software layers tuned for the machine. The platform-independent part should avoid direct references to specific resources and their availability, and instead provide the information needed to optimise behaviour.

This paper summarises how task abstraction, which first appeared in the 1990s and is already mainstream in HPC, should be the basis for a composable and dynamic performance-portable interface. It outlines the innovations that are required in the programming model and runtime layers, and highlights the need for a greater degree of trust among application developers in the ability of the underlying software layers to extract full performance. These steps will help realise the vision for performance portability across current and future architectures and problems.

Key insights

- **Porting applications is difficult and must be successful in a short time.** HPC hardware evolves quickly and supercomputers have a fast turnover. On the other hand, HPC applications have a long lifetime. Usage practices are themselves evolving towards more complex and dynamic scenarios, involving simulation, code coupling, (in-situ) analysis and visualisation, interactive computing, urgent computing, and malleability.

- **Applications should therefore be expressed in a way that facilitates performance portability**, as agnostic as possible (and no more) of the architectural particularities. The advantages are more rapid porting to new platforms and the flexibility to adapt to new application scenarios. One of the hardest aspects to address is the need to build trust among developers that the underlying software layers can extract full performance from a performance-portable codebase rather than relying on hand optimisation.

- **Task-based programming models allow HPC programmers to express applications and workflows in such a performance portable way**. As global optimisation and coordination is needed across all levels of the software stack, task-based models identify pieces of work to be done, thereby providing a clean abstraction for the programmer to reason about the program. The runtime layer should handle parallel scheduling, overlapping of computation and communication, data placement and transfers, resiliency, load balancing and dynamic resource sharing among applications.

- **Open standardisation is one of the best ways to drive adoption**. HPC application developers require a commitment to long-term maintenance, support and high performance across platforms. There is a vibrant ecosystem of programming model research in Europe adopting various approaches, the most successful of which should be standardised.

Key recommendations

- The community should promote best practices in application porting:
  - Promote greater trust in the ability of lower-level software layers to extract full performance, rather than displaying “impressive” optimisation tricks.
  - Confine adaptation efforts to short kernels, while preserving the global application structure, favouring composability.
  - Separate the concerns of experts in distinct fields to let them jointly contribute to the optimisation effort in a coordinated, interference-free manner, from application domain, to algorithm, to programming model, to runtime, to platform.

- Research in task-based programming models and runtimes should:
  - Improve efficiency and extend scope of runtime resource management decisions.
  - Support new platforms/hardware, e.g. FPGAs, PIM, and usage models, e.g. malleability.
  - Enable cooperation of task-based models with existing HPC components, such as compilers, job schedulers, application deployment managers, MPI, etc.
  - Develop a portfolio of tools around task-based models, such as performance analysis, verification and debugging frameworks.

- Europe should make the most of its vibrant ecosystem of programming model research:
  - Continue to foster various approaches motivated by and evaluated using large-scale applications.
  - Continue to incorporate the most successful European-led approaches into open standards. But standardisation needs to be done at the right level of maturity and not all technologies may be a good fit for a single model.
  - Evolve programming models and runtimes to support new technologies, such as new types of accelerators, FPGAs and, in future, Processing-in-Memory (PIM). The support should not be specific to hardware vendors, but be open and standardised, allowing kernel code portability across vendors.

Shortening application deployment and optimisation efforts on complex, heterogeneous, high turnover HPC supercomputers

Supercomputer platforms, such as those in PRACE Tier-0, are at the current state of the art in computer hardware technology, and as such, have short production lifetimes, at most 5 to 10 years. New machines must therefore deliver scientific and industrial results right upon entering production. At the same time, the technology race is leading to increasingly complex, heterogeneous, diverse computing hardware as vendors devise new solutions to deliver ever higher performance, making the software adaptation effort challenging. This evolution can be seen in the diversity of technical solutions, with and without accelerators, of various kinds, and with diverse and heterogeneous memory systems, in the most powerful systems at the Top500 and Green500 rankings. For instance, Top500 #1 Fugaku uses Fujitsu's ARM derived A64FX processors, #2 Summit and #3 Sierra systems combine IBM's Power9 processors with NVidia Volta GV100 GPUs, #4 TaihuLight is built on Sunway SW26010 many-core processors, etc.

Application development is usually expensive and time consuming, and this is especially true in HPC, where applications tend to have long lifetimes to capitalise on developer investment and embody domain expertise that has grown over time. It is essential that the application's architecture is stable and that all its prerequisites have long-term availability, maintenance and high performance across current and future target platforms. Applications are also becoming more diverse and complex as new users and science fields come to HPC. For example, scientific simulations now often need to be tightly coupled with AI-based data analytics in an efficient way.

Task-based runtime systems perfectly cope with such evolving hardware and software. Since there is little time for software developers to port and tune applications specifically for new platforms, applications need to be expressed in a way that facilitates high performance across a range of hardware and situations. As much as possible, the majority of the application code should not be dependent on specific resources or their availability, so that, in a simple example, instead of dividing up the work by the number of cores and/or devices the application should use an abstraction like a task-loop or parallel for. Resilience, load balancing and data placement should also be handled at lower software layers. This requires a greater degree of trust in the ability of the underlying software layers to extract high performance, and a larger focus on long-term maintainability, portability and performance rather than (impressive) short-term performance tricks.

Task-based programming models allow developers to write applications in a way that preserves such stability by confining the adaptation effort mostly on select kernel routines that can be encapsulated into tasks with known inputs and outputs. The same task abstraction can be used to build libraries and domain-specific languages that can in turn be used by the application. Once standardised, the application developers can rely on wide support.

Tasks as an abstraction for programming first appeared in 1995 [Blum95], and became mainstream in HPC programming with the publication of OpenMP 3.0 in 2008 [OpenMP3]. Since then, OpenMP's support for tasks has been incrementally refined, to support dependencies, task-loops and reductions, amongst other things, with European research in programming models at the forefront of this innovation.

As task-based programming models are reaching maturity, research effort should now focus on (a) improving and extending the scope of delegated management decisions, (b) developing an interoperable portfolio of tools around task-based models, (c) extending the cooperation and composability of task-based models with existing components and programming models, and (d) supporting new platforms and architectures, such as FPGAs and Processing in Memory (PIM).

Usual HPC application design practices

HPC applications are most commonly built around variants of the Bulk-Synchronous Parallelism (BSP) model, alternating phases of intra-node computations with phases of inter-node exchanges, separated by synchronisation operations, either using explicit message passing communications with MPI or abstracted through Partitioned Global Address Space (PGAS) approaches. The same BSP pattern is also commonly used at the node level to exploit multiprocessors and cores, either through MPI again, using multiple processes per node, or through OpenMP, using multiple threads in a fork/join manner. In the same way, synchronous kernel offloads to accelerators or GPUs using NVIDIA CUDA, OpenCL, or through OpenMP/OpenACC constructs, can also be seen as declinations of BSP.

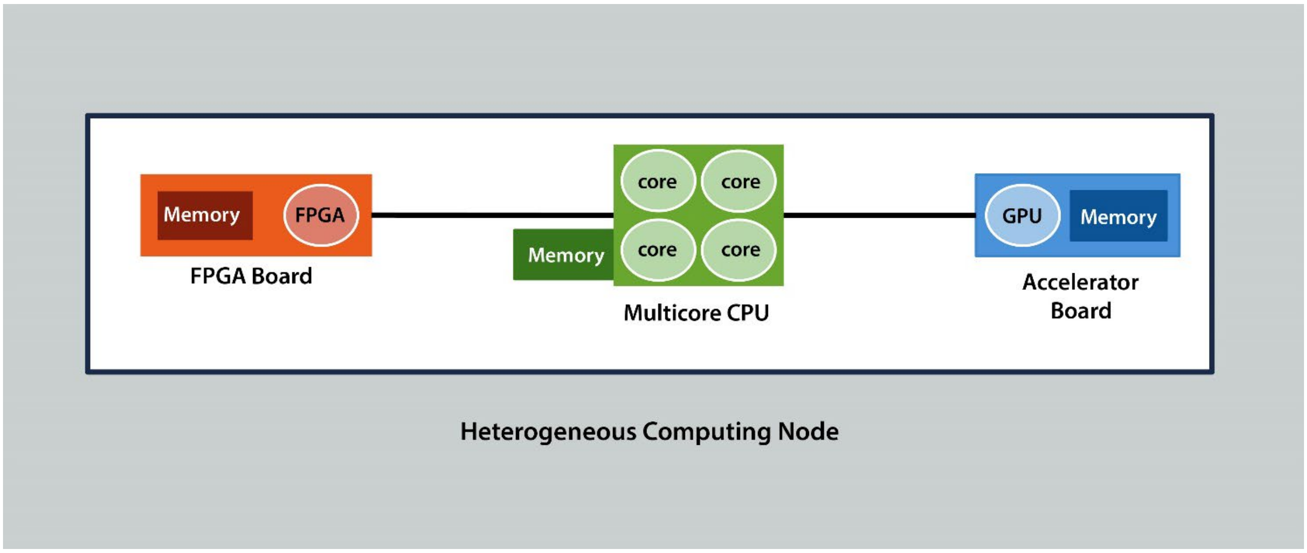


Fig. 1: Illustration of a heterogeneous computing node equipped with a multicore CPU, a GPU-based accelerator board equipped with its own embedded memory and reconfigurable FPGA device.

However, the prominent issue with this BSP approach is the difficulty in ensuring a proper load balancing and coordination across these multiple levels, and the resulting amount of processing unit cycles wasted in synchronisation operations. Using non-blocking programming techniques may help to reduce the amount of wasted cycles. Yet, this makes the burden even higher for the application programmer, for instance to decide which pairs of operations can safely be overlapped, or to determine which amount of CPU computation balances some GPU kernel offload. Factoring in additional sources of complexity and imbalances, such as instruction set variants, cache hierarchies, non-uniform memory access (NUMA) architectures, specialised memory areas with distinct performance patterns (HBM/MCDRAM, NVM), or reconfigurable FPGA computing devices (see Fig.1), for instance, the effort required for hand-tuning an application on a given architecture becomes largely impractical by itself, and much of the investment becomes irrelevant or counterproductive on a different architecture. Moreover, the performance variability across multiple runs due to the interplay of multiple factors such as frequency scaling, thermal throttling, and general system noise defeats the most careful static execution planning.

Moreover, the entanglement of the application algorithms with unrelated synchronisation, load balancing and more generally with parallelism management operations makes the resulting code difficult to read, to maintain and to evolve. Single-source programming models such as SYCL or Intel's DPC++, and to some extent OpenACC and OpenMP, attempt to address these issues by providing HPC programmers with a means to potentially target diverse hardware computing resources from a unique source code. In the short term, this may indeed be convenient. However, those programmers then become dependent on the availability of the right compiler to target a given combination of CPUs, accelerators, and other kinds of computing devices. Their immediate burden is lightened, but their control is reduced in the process. In addition, this hinders the ability to combine multiple programming models within the same application.

Overall, such common HPC application design techniques therefore suffer from significant issues to keep up with a fast evolving supercomputing landscape. Task-based programming models such as StarPU [Agullo17] (Inria, France), OmpSs [Duran11] (BSC, Spain), OCR-Vx [Dokulil16] (University of Vienna, Austria) and DuctTeip [Zafari19] (University of Uppsala, Sweden), developed in Europe, as well as non-European initiatives such as HPX [Kaiser14], Legion [Bauer12], OCR [OCRspec], PaRSEC [Hoque17], to name a few, all define a clear software design path to address these issues.

A change of approach is needed, which goes beyond simply expressing bulk parallelism in the form of tasks, such as, for example, by converting "parallel for" loops into task-loops. It is necessary to exploit nested tasks, with the possibility of expressing offloading at multiple levels if needed.

Programming for performance portability

Performance portability designates a property of applicative codes to maximise efficiency across a broad range of hardware platforms, while minimising the specific adaptation effort required on each platform. Programming for performance portability requires a software design approach that isolates core application algorithms, i.e. long-term pieces of code persistent across platforms, from kernels, i.e. short-term code tuned for specific hardware. Moreover, machine-dependent work management should not be reinvented for the application and tangled up with it, but delegated to dedicated runtime systems.

Principles of task-based parallel programming

Parallel programming is a matter of assigning pieces of work to workers. Parallel programming models can be classified in three broad classes, whether they let programmers primarily focus on managing *entire processes*, such as with MPI, focus on *managing workers*, named “threads”, within such processes in the case of thread-based parallel programming models, or focus on *expressing the pieces of work* to be done, named “tasks”, in the case of task-based models. The common idea of task-based parallel programming models is to let the application programmer define elementary blocks of source codes, e.g. “pieces of work”, as individual tasks, and then express the application core algorithms in terms of sequences of those tasks.

A task-based programming model typically works together with an execution model supplied by a runtime system. The application submits execution requests for tasks in sequences to the runtime system, oftentimes inheriting understood sequential semantics from the original program, while the runtime system executes the tasks in the background on the available processing units in parallel, using some scheduling algorithm. Additional directives and constraints let the application core express semantic information and hints such as relationships among tasks or relationships between tasks and pieces of data, to guide the parallel execution process.

Task-based execution models

Task work requests are collected and executed in the background by a runtime system. The runtime system is therefore in charge of driving the execution of the application tasks while efficiently exploiting the available hardware resources.

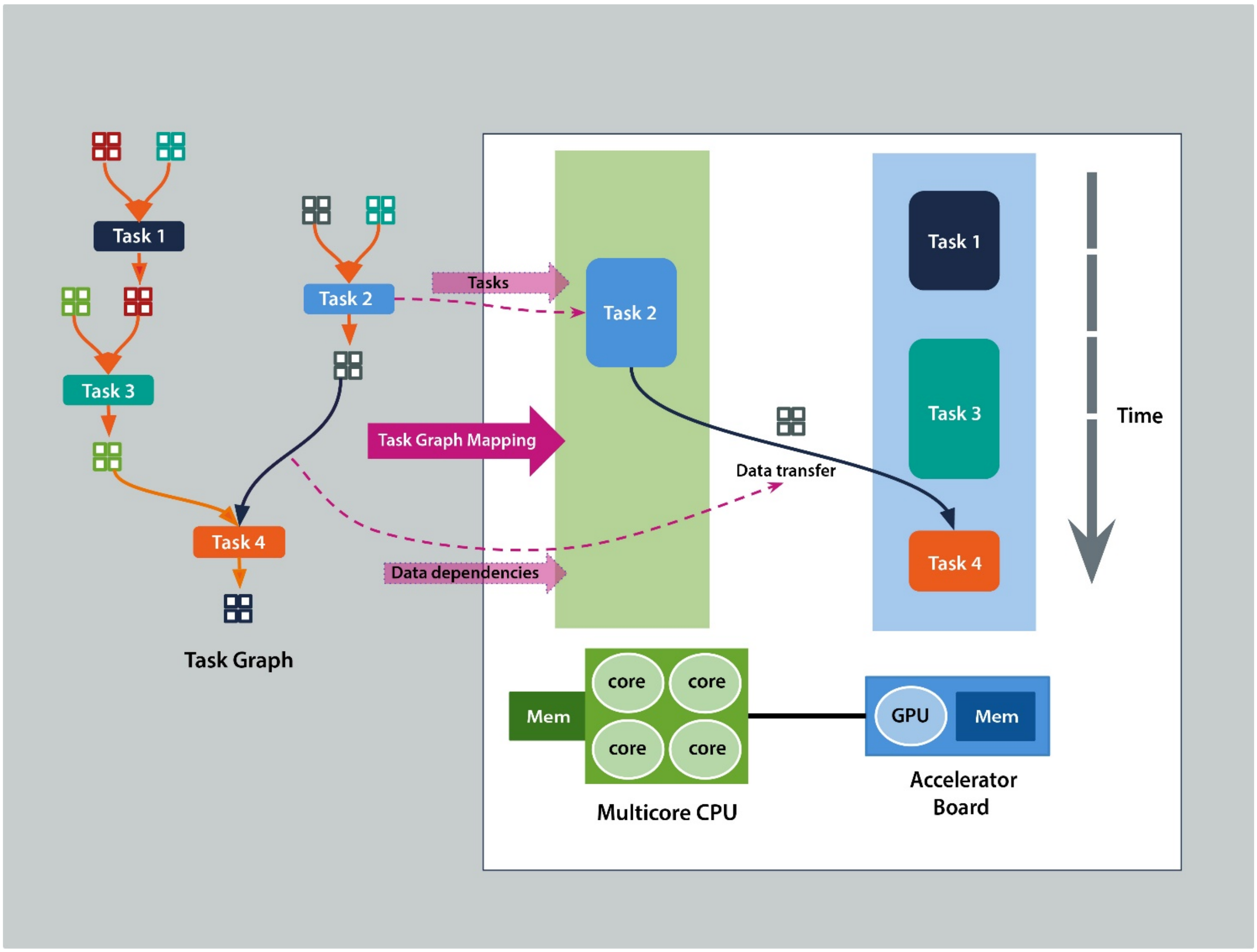


Fig. 2: Illustration of the way the abstract representation of an application given by the task graph can be used by a runtime system to map tasks on available, possibly heterogeneous platforms. Tasks can be dynamically mapped to appropriate processing units using scheduling algorithms. Data consistency between devices can be enforced using data dependence edges. Prefetch operations can be triggered automatically to overlap data transfers with computation.

The semantics information constituted from the flow of submitted tasks and their dependencies is commonly referred to as the task graph. This task graph synthesises the application structure in a conceptual object that can be exploited by the runtime system in many ways to perform operations on behalf of the application programmer, beyond the parallel execution of tasks, see Fig. 2. The task graph may be hierarchical, with nested tasks, in order to control the maximum degree of parallelism, and inherent synchronisation and contention, managed at a time. Data dependence edges in the task graph, for instance, let the runtime system infer when to trigger data transfers between computing nodes, as well as inside nodes between memory spaces (main memory, specialised memory, device-embedded memory). Computation/communication overlap and data prefetches can be handled transparently, in the process, to minimise idle times. The runtime system can even eliminate redundant data transfers using data replication and caching techniques. Other examples include using task graph information to enable dynamic resource management, malleable jobs, fault tolerance enforcement, to guide checkpointing and recovery operations, or deciding when to page data to disk and when to fetch it back in out-of-core execution scenarios on large datasets, for instance.

Thanks to their flexibility, task-based models can co-exist alongside other common HPC models, including other task-based models. This enables scenarios such as the incremental transformation of legacy codes into task-enabled codes for languages such as C, C++ and Fortran, the composition of multiple models such as tasks with MPI, the coupling of multiple codes, and even the active interoperability between models, as demonstrated in H2020 Project INTERTWinE [H2020INTW].

Properties of task-based programming models

Task-based programming models invite programmers to express applications in terms of elementary tasks and dependence relationships. This semantically rich abstract representation enables runtime systems to handle the resource management and load balancing duty on behalf of programmers. Furthermore, the clear separation of the stable machine-independent application structure from well identified machine dependent task routines greatly simplifies porting efforts. The resulting performance portability ensures a quick short term return on investment on new platforms, while preserving long term software development investment.

Using a task-based model, programmers focus on decomposing their programs into tasks, while delegating it to the runtime system to decide how tasks are mapped to the available execution units of a parallel system and how data is managed across the memory hierarchy. This change in mentality from managing processes or threads of execution to describing the work to be done is in some ways similar to the move from assembler to Fortran/C or the move to structured programming. In all cases, application development is easier, safer and more efficient for human programmers and application analysis and optimisation is easier, safer and more effective for tools.

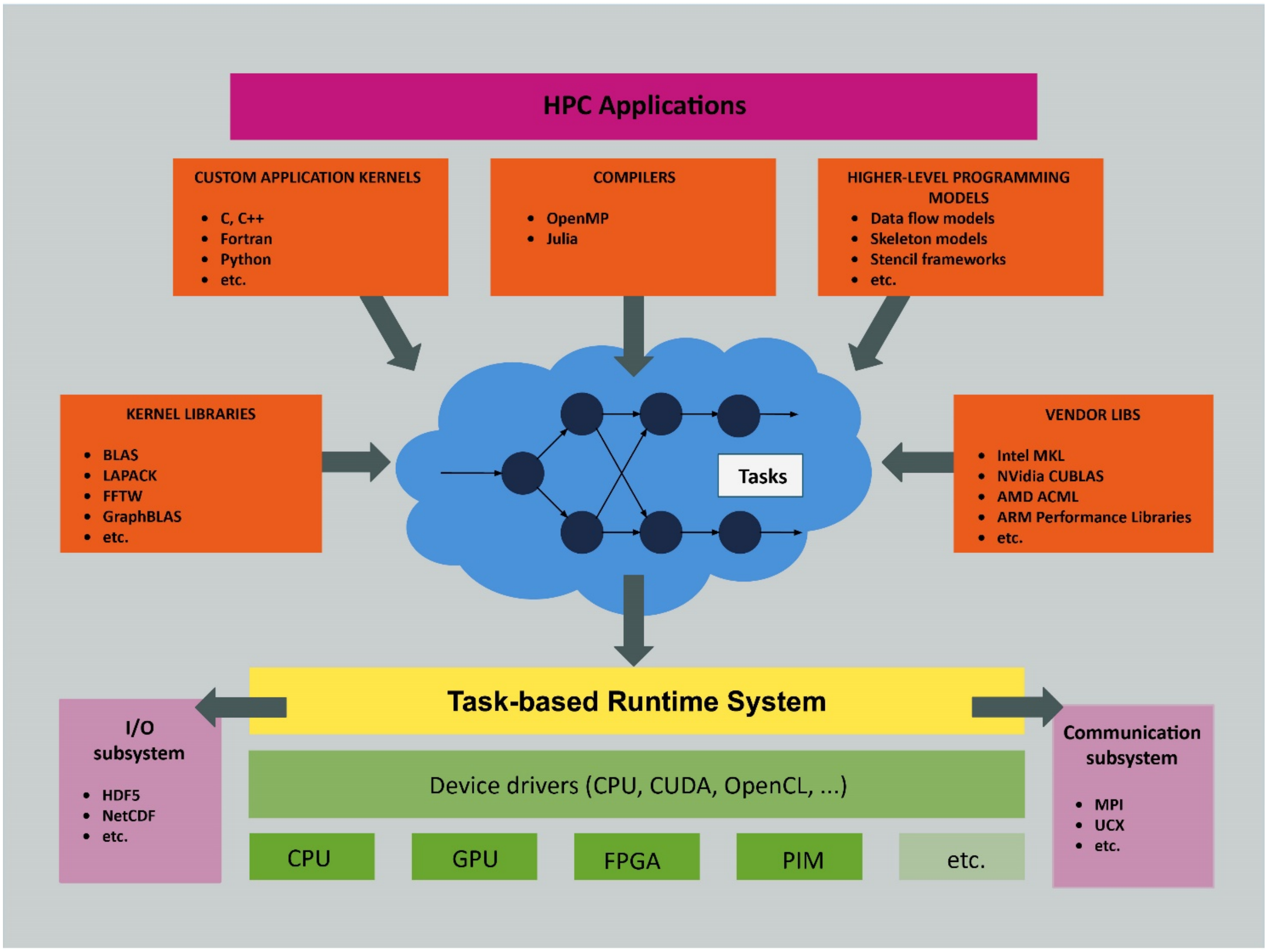


Fig. 3: Overview of the task-based programming approach. An application submits tasks to a runtime system in charge of executing them on the available resources, thanks to the property of management delegation. Task implementations can be supplied through multiple means ranging from custom routines to kernels from highly optimised vendor libraries, thanks to the property of separation of concerns. Hardware-specific device drivers bring the property of performance portability.

Task-based models facilitate performance portability [Dokulil16, Agullo17, Garcia20] by enforcing a clear separation between the task submission side of an application, to be kept largely unchanged across platforms, and the architecture-specific code concentrated in variants of the task kernel routines for CPUs, GPUs, FPGAs, PIM, and so on. The load balancing work over available computing units is handled by the runtime system scheduler. Task-based models also naturally bring the property of separation of concerns as a key added benefit, especially in HPC. Fig. 3 presents an overview of the task-based programming approach and illustrates these properties.

Indeed, the application core algorithm designers, task kernels designers, task scheduling algorithm designers, runtime systems designers can be distinct persons, experts in their own domain, contributing to the same application in a safe and efficient manner, thanks to the proper isolation of each specific concern. In particular, the popularity of highly optimised routines from vendor-specific linear algebra libraries such as Intel MKL or NVIDIA CUBLAS show the interest of a portability strategy concentrated on kernels.

Task-based models inherently increase application robustness as well. Using task relationship information with other tasks and with pieces of data, low-level concerns such as task synchronisation and data consistency management can be handled transparently. Further, task routines abiding by the functional programming concept of a pure function, that is, with no hidden internal side effects or dependencies, can be scheduled under a deadlock-free guarantee in any constraints-compliant order. Tasks can also have known side effects and dependencies, as required for deadlock-free interoperability with other programming models or APIs, such as MPI.

In some specific application cases, simpler abstractions may be more convenient and effective than tasks. Examples of such high level layers include data-flow programming models, stencil-oriented models, or skeleton programming models, for instance (see Fig. 3). These models may however be offered as high level programming layers on top of task-based programming environments, to get the benefit of an increased case-specific abstraction, while preserving the performance portability and ecosystem integration properties of task-based models.

Task-based parallelism initiatives in Europe

StarPU

The StarPU runtime system [StarPU, Agullo17] is developed in Bordeaux, France at Inria and the LaBRI laboratory. A key characteristic of StarPU is its performance model-based heterogeneous scheduling ability on computing nodes equipped with accelerators. It can be programmed either directly, as in the FLUSEPA code from Airbus [Flusepa], through libraries such as the Chameleon [Chameleon] (dense) and PaStiX [PaStiX] (sparse) linear algebra solvers, or from higher level programming models such as OpenMP and SkePU.

OmpSs

OmpSs is the parallel programming model from Barcelona Supercomputing Center, which is part of a 10+ years' investment in programming model research [Duran11, Perez17]. It provides annotations to a sequential program with a single address space/namespace which is able to execute in parallel through the runtime computation of task dependencies. OmpSs has been the forerunner for tasking in OpenMP 3.0 and task dependencies in OpenMP 4.0, and many improvements in subsequent versions of OpenMP, such as reductions, commutative, concurrent, task-loops were the result of research on OmpSs. The latest version, OmpSs2, is supported by the Mercurium source-to-source compiler and Nanos6 runtime system.

OCR-Vx

The OCR-Vx asynchronous, task-based runtime system [OCRVx] developed at the University of Vienna, Austria, is based on the Open Community Runtime specification [OCRspec] proposed in the context of the US XStack (Exascale Software Stack) initiative. The central idea of OCR is to decouple a program's computations and data from the actual execution units and memory locations of a parallel system, based on the concepts of event-driven tasks and data blocks. To support dynamic program adaptation and facilitate fault tolerance, both tasks and data blocks are relocatable. OCR-Vx has been implemented in C++ for shared and distributed-memory systems.

Recommendations for 2024–2025

Best practices

The community should promote best practices in application porting. Application developers should be encouraged to trust low-level software layers to extract performance from declarative information. Adaptation efforts should focus on short kernels within a stable application structure. As applications are getting ready for the exascale transition, now is also the time to encourage long term-oriented, performance-portable programming using task-based models, for instance, with the help of PRACE and national supercomputing entities.

Research in task-based programming models

Research in task-based programming models should continue to improve and extend low-level resource management and data locality decisions through more expressive abstractions and compiler/runtime intelligence. New platforms and hardware, such as FPGAs and PIM, provide important new challenges, often benefiting from the re-evaluation of previous ideas in the new context. Similarly, the programming model support layers should be extended to support new usage models such as malleability, for which the tasking model provides a natural way for the program to react to changing resource availability. In addition to supporting tasking within a node, as hybrid MPI+X, we should continue investigating tasking programming models across nodes. In complex applications, programming model interoperability and composition are important, as is interoperability with other components such as the job scheduler, for which standardisation will be important, perhaps via PMIx. The task graph information and separation of concerns bring the opportunity for more accurate performance analysis and debugging tools. In addition, the HPC ecosystem should be ready to incorporate ideas from other fields that face similar problems, e.g. in cloud computing.

Ecosystem and standardisation

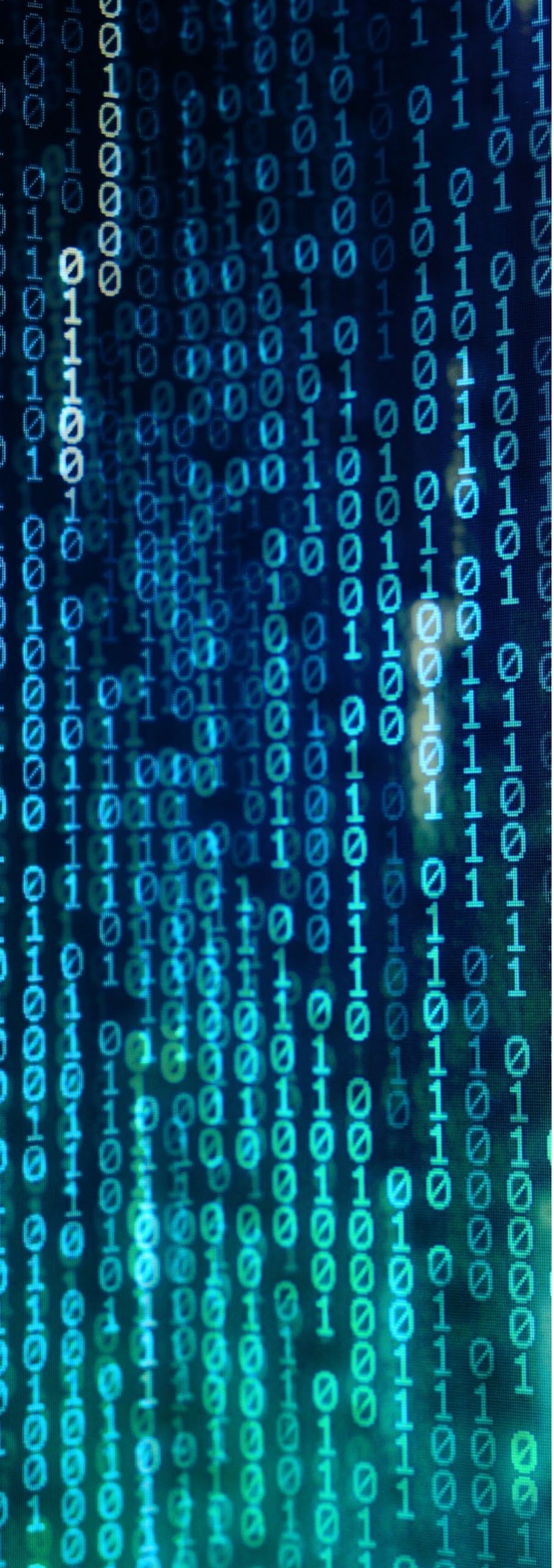
There is a vibrant ecosystem of programming model research in Europe, and now is the time to increase the TRL and make them ready for production use in exascale supercomputers using large-scale applications. Since long-term software development depends on the stability of all prerequisites, including stable and supported implementations of the programming models, the most successful approaches should be incorporated into open standards. Tremendous progress has been achieved and a broad consensus has been reached, especially through the OpenMP initiative. It may not be possible to incorporate everything into OpenMP, but open standardisation is one of the best ways to enable programming model uptake and portability across vendors.

Conclusions

This white paper has reviewed the case for task-based performance portability and identified what is needed to drive greater adoption. We discuss why the task abstraction provides a clean, stable and dependable interface between the machine-independent application programmer, library or domain-specific language implementation and the machine-dependent software layers. We also outline the academic and industrial research needed to realise the vision of performance portability across current and future architectures and problems.

References

- [Agullo17] Emmanuel Agullo, Olivier Aumage, Mathieu Faverge, Nathalie Furmento, Florent Pruvost, Marc Sergent, and Samuel Thibault. Achieving High Performance on Supercomputers with a Sequential Task-based Programming Model. IEEE TPDS, 2017. DOI: http://dx.doi.org/10.1109/TPDS.2017.2766064.
- [Bauer12] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: expressing locality and independence with logical regions. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12), 2012. PDF: https://legion.stanford.edu/pdfs/sc2012.pdf.
- [Blum95] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. 1995. Cilk: an efficient multithreaded runtime system. *SIGPLAN Not.* 30, 8 (Aug. 1995), 207–216. DOI: https://doi.org/10.1145/209937.209958.
- [Chameleon] Chameleon Dense Linear Algebra library: https://gitlab.inria.fr/solverstack/chameleon.
- [Dokulil16] J. Dokulil, M. Sandrieser and S. Benkner, "Implementing the Open Community Runtime for Shared-Memory and Distributed-Memory Systems," 2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP), 2016, pp. 364-368, doi: 10.1109/PDP.2016.81.
- [Duran11] A. Duran, E. Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, X. Martorell and Judit Planas. Ompss: a Proposal for Programming Heterogeneous Multi-Core Architectures, Parallel Process. Lett., 2011. Volume 21, pp.173–193.
- [Flusepa] Code FLUSEPA: https://hal.inria.fr/hal-01507613.
- [Garcia20]. Garcia-Gasulla, M., Mantovani, F., Josep-Fabrego, M., Eguzkitza, B. and Houzeaux, G., 2020. Runtime mechanisms to survive new HPC architectures: a use case in human respiratory simulations. *The International Journal of High Performance Computing Applications*, *34*(1), pp.42-56.
- [H2020INTW] H2020 INTERTWinE: http://www.intertwine-project.eu/.
- [Hoque17] Reazul Hoque, Thomas Herault, George Bosilca, and Jack Dongarra. Dynamic task discovery in PaRSEC: a data-flow task-based runtime. In Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA '17), 2017. DOI: https://doi.org/10.1145/3148226.3148233.
- [Kaiser14] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. HPX:A task based programming model in a global address space. In 8th international conference on partitioned global address space programming models PGAS’14, 2014. DOI: https://doi.org/10.1145/2676870.2676883.
- [OCRspec] OCR specification: https://www.univie.ac.at/ocr-vx/doc/ocr-v1.1.0.pdf
- [OCRVx] OCR-Vx website: https://www.univie.ac.at/ocr-vx/
- [Perez17] Perez, Josep M., et al. "Improving the integration of task nesting and dependencies in OpenMP." 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS). IEEE, 2017.
- [OpenMP3] OpenMP Application Program Interface. Version 3.0 May 2008. Available: https://openmp.org/wp-content/uploads/spec30.pdf.
- [PasTiX] PasTiX Sparse Linear Algebra library: https://solverstack.gitlabpages.inria.fr/pastix/
- [StarPU] StarPU website: https://starpu.gitlabpages.inria.fr/
- [Zafari19] Afshin Zafari, Elisabeth Larsson, Martin Tillenius. DuctTeip: An efficient programming model for distributed task-based parallel computing. Parallel Computing, Vol. 90, 2019. DOI: https://doi.org/10.1016/j.parco.2019.102582.



Authors:

Olivier Aumage is a full time researcher at Inria in Bordeaux since 2003; his main research topics include parallel runtime systems, programming models and languages for high performance computing applications.

Paul Carpenter is a Senior Researcher at Barcelona Supercomputing Center since 2011. His research interests include performance-portable parallel programming models, system software and energy-efficient architectures.

Siegfried Benkner is a professor of Computer Science at the University of Vienna where he focuses on research in programming models, languages compilers, runtime systems, and tools for high performance computing.

This work was supported by the Spanish Government (contract PID2019-107255GB), Generalitat de Catalunya (contract 2014-SGR-1051), and the European Union's Horizon 2020 research and innovation programme under grant agreements No 955606 (DEEP-SEA) and No 754337 (EuroEXA). Paul Carpenter holds the Ramon y Cajal fellowship under contracts RYC2018-025628-I of the Ministry of Economy and Competitiveness of Spain. This work was supported by the French Government (contract ANR-19-CE46-0009), Région Nouvelle Aquitaine (contract 018-1R50119) and the European Union's Horizon 2020 research and innovation programme under grant agreements No 671602 (INTERTWinE) and No 801015 (EXA2PRO). This work was supported by the Austrian Science Fund grant P29783.

Cite as: O. Aumage et al., Task-Based Performance Portability in HPC, ETP4HPC White Paper, 2021, doi 10.5281/zenodo.5549731

DOI: 10.5281/zenodo.5549731