Distributed task execution: Opportunities, challenges and lessons learnt from OmpSs-2@Cluster

Paul Carpenter
Barcelona Supercomputing Center, Spain

Omar Shaaban
Barcelona Supercomputing Center, Spain

Juliette Fournis d'Albiat
Barcelona Supercomputing Center, Spain

Isabel Piedrahita
Barcelona Supercomputing Center, Spain

Abstract

This talk will present recent advances in extending OmpSs-2 to distributed-memory systems, highlighting three contributions and the associated challenges. OmpSs-2@Cluster employs a common address space and weak accesses to support concurrent task creation and dataflow execution across nodes. Achieving good performance and scalability on 16 to 32 nodes requires detailed performance analysis together with a set of optimizations and runtime techniques, which I will outline in the talk. Second, I will describe how task offloading, in combination with BSC's Dynamic Load Balancing (DLB), enables OmpSs-2@Cluster to mitigate load imbalance in MPI + OmpSs-2 programs with minimal application changes. Third, I will explain how the runtime can exploit the iterative structure of certain task dependency graphs to precompute communications and execute iterative regions efficiently, yielding performance and scalability comparable to state-of-the-art asynchronous MPI+X. Together, these results indicate that distributed tasking can combine productivity, adaptability, and high performance in modern HPC applications.

2012 ACM Subject Classification Computing methodologies → Parallel computing methodologies; Software and its engineering → Parallel programming languages; Software and its engineering → Distributed programming languages; Software and its engineering → Runtime environments; Computer systems organization → Distributed architectures

Keywords and phrases Task-based programming, distributed-memory clusters, programming models, runtime systems, task scheduling, data dependency management, load balancing, asynchronous communication

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2026.8

Category Invited Talk

Funding This work is funded by the Barcelona Zettascale Laboratory (REGAGE22e00058408992), backed by the Spanish Ministry for Digital Transformation and of Public Services, within the framework of the Recovery, Transformation, and Resilience Plan - funded by the European Union - NextGenerationEU. It is also supported by the Spanish State Research Agency - Ministry of Science and Innovation under contract PID2019-107255GB-C21/MCIN/AEI/10.13039/501100011033 and Ramon y Cajal fellowship RYC2018-025628-I/MCIN/AEI/10.13039/501100011033 and by "ESF Investing in your future", as well as by the Generalitat de Catalunya (2017-SGR-1414).

© Paul Carpenter, Omar Shaaban, Juliette Fournis d'Albiat, and Isabel Piedrahita;
licensed under Creative Commons License CC-BY 4.0
17th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures and 15th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2026).
Editors: Davide Baroffio, Paola Busia, Lev Denisov, and Nitin Shukla; Article No. 8; pp. 8:1–8:6
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Task-based programming has become a powerful abstraction for expressing parallelism and managing complexity in modern HPC, and it is increasingly accepted for node-level parallelism. Tasks were introduced in OpenMP 3.0 in 2008 and substantially strengthened in OpenMP 4.0 (2013) with explicit task dependencies, enabling dependency-driven asynchronous execution. Later OpenMP revisions added more advanced tasking capabilities, e.g. taskgroups, task reductions and detached tasks, and improved the integration with accelerators. Task-based execution is also widely used in libraries and runtimes such as Intel Threading Building Blocks (TBB) [16] and in systems including Cilk-derived frameworks, HPX, StarPU [2], PaRSEC [7, 12], Legion [6] and OmpSs [10].

Task-based approaches have likewise proven successful in workflow systems such as COMPSs [13] and Pegasus [9], where tasks naturally correspond to coarse-grained units of work and communication costs can be amortized over longer execution times. Nevertheless, despite over fifteen years of research and clear benefits at both the finest scale (node-level parallelism) and the coarsest scale (workflows), task-based programming has not displaced message passing in the intermediate regime of distributed HPC applications. In this setting, the overheads of task graph management, dependency tracking and data versioning can become prohibitive for fine- to medium-grained tasks on distributed memory, limiting scalability. Moreover, dynamic scheduling and implicit communication can reduce performance predictability, leading to performance anomalies and unexpected bottlenecks that are difficult to diagnose.

OmpSs-2@Cluster [1, 5] is a research platform for exploring distributed task-based execution at a moderate granularity, building on the refined semantics of OmpSs-2 [4] and a runtime designed for scalable cluster execution. It evolves earlier OmpSs@Cluster work by Bueno et al. [8] and incorporates lessons from earlier efforts in distributed task execution. While retaining tasks and dependencies as the core abstraction, OmpSs-2@Cluster mitigates the scalability challenges that arise when task creation, dependency tracking and data management span multiple nodes.

A key design element is support for weak accesses (also known as weak dependencies), as introduced by OmpSs-2 [15]. A weak access indicates that the task does not directly access the data region but its nested subtasks may do so. This allows a parent task to begin execution before the completion of any data transfers required by its children, thereby avoiding unnecessary synchronization and overlapping communication with subtask creation and related dependency management. Weak accesses are a mechanism that supports grouping of tasks into a coarser-grained unit to be offloaded to another node. OmpSs-2@Cluster also employs fragmented region dependencies to interoperate between coarse-grained accesses passed among nodes and fine-grained accesses manipulated on each node. Together these mechanisms aim to make task-based execution more scalable on distributed-memory clusters.

The remainder of this extended abstract provides an overview of the substantial effort devoted over the years to performance analysis and optimizations in OmpSs-2@Cluster. It also discusses the opportunities, challenges and recent progress along two complementary directions: first, inter-node load balancing in MPI + OmpSs-2 programs; and second, exploiting iterative program structure to amortize the costs of task graph construction and management.

2 Runtime and optimizations

OmpSs-2@Cluster uses the same compiler as regular OmpSs-2 and relies on an open-source fork of the Nanos6 runtime known as Nanos6@Cluster. Early development of Nanos6@Cluster was carried out in a branch of the Nanos6 code base, with regular upstreaming of changes.

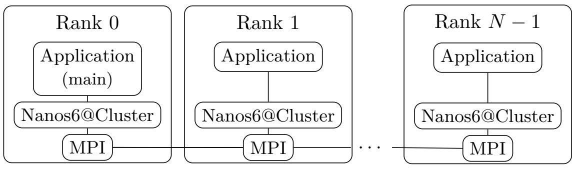
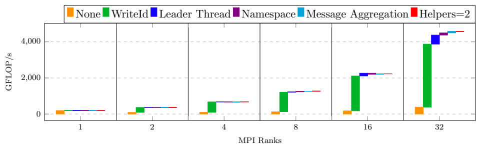
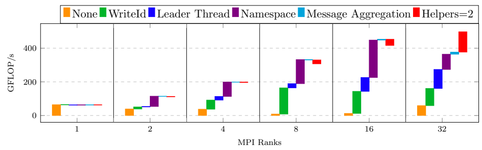


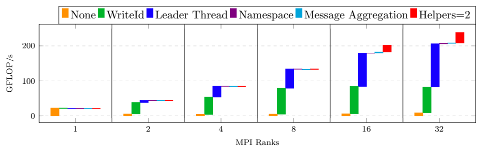
Figure 1 OmpSs-2@Cluster architecture: each rank is a peer and `main` runs as a task on Rank 0.



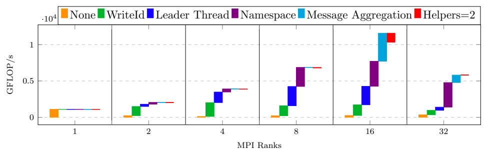
(a) Matrix–matrix multiplication 16384 × 16384



(b) Matrix–vector multiplication 32768 × 32768



(c) Jacobi benchmark 32768 × 32768



(d) Cholesky factorization 32768 × 32768

Figure 2 Performance impact of key optimizations on MareNostrum 4. Reproduced from [14].

This approach was later abandoned, as the significantly higher maturity of Nanos6 and its requirement for stable shared-memory–oriented internal interfaces made it difficult to accommodate the experimental and rapidly evolving features needed for distributed execution, some of which involved intrusive changes to these internal APIs. Moreover, maintaining a separate fork allowed the small research-focused OmpSs-2@Cluster team to delay certain technical transitions, most notably the migration from the legacy source-to-source Mercurium compiler to LLVM, in order to concentrate on core runtime development.

As shown in Figure 1, each MPI rank runs an independent instance of Nanos6@Cluster, with all instances communicating as peers via MPI. To simplify data management across ranks, each process establishes an identical virtual address space using `mmap`, allowing tasks to refer to the same memory addresses regardless of the rank on which they execute.

While the basic mechanism for task offloading was relatively straightforward to implement and completed within a few months, achieving satisfactory performance required substantial runtime optimizations developed over several years. Figure 2 illustrates the cumulative impact of these optimizations on performance. As the figure suggests, different benchmarks benefit from different subsets of optimizations. In practice, performance was often sensitive to low-level implementation details, and any such cumulative view depends on the order in which optimizations are introduced in the figure, which is to some extent arbitrary and chosen for explanatory purposes.

The main optimizations implemented in Nanos6@Cluster include WriteID, a form of data versioning used to avoid redundant data transfers; LeaderThread, which dedicates a thread to handle incoming MPI messages such as newly offloaded tasks and to process message completions; and Namespace, which eliminates unnecessary host-mediated messages between consecutive tasks offloaded to the same rank. Additional improvements include message aggregation, which coalesces control messages when multiple accesses become ready, and multiple low-priority Helper tasks that assist with message handling and runtime progress when compute resources would otherwise be idle. Together, these optimizations substantially reduce overheads and enable scaling to approximately 16–32 nodes for the evaluated small-scale benchmarks.

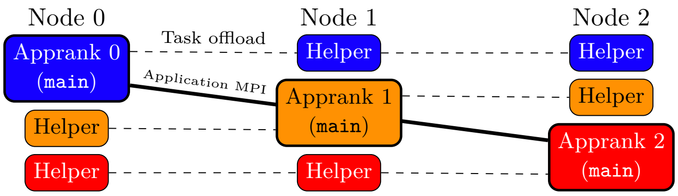


Figure 3 Architecture of MPI+OmpSs-2@Cluster. Application ranks (appranks) communicate via MPI and helper ranks on some other nodes can execute tasks from heavily loaded appranks.

3 Dynamic Load Balancing (DLB)

Load imbalance is a long-standing source of inefficiency in high-performance computing. It is commonly addressed at application level through techniques such as mesh partitioning, domain decomposition, or manual work redistribution, often guided by problem-specific heuristics. While effective, these approaches entangle load-balancing concerns with application logic and may require substantial code refactoring, complicating development and long-term maintenance. Although OmpSs-2@Cluster does not scale sufficiently to serve as the primary distributed-memory programming model for large-scale HPC applications, it is well suited to addressing residual load imbalance in hybrid MPI+OmpSs-2 programs. In this context, OmpSs-2@Cluster complements static partitioning by redistributing work at runtime.

The basic approach is illustrated in Figure 3. Each MPI rank visible to the application (hereafter referred to as an application rank or apprank) is shown in a different colour, with a single apprank per node in this example. To mitigate load imbalance in an apprank, additional helper ranks are deployed on a subset of other nodes. These helper ranks are full runtime instances that execute tasks offloaded from a given apprank within a dedicated process, providing isolation between appranks while enabling dynamic redistribution of work at runtime.

Load balancing is done at three levels. First, at coarse granularity, helper ranks are activated based on a prediction of upcoming load imbalance. The prediction is calculated by the runtime and passed to an external solver, which determines the minimum number of helpers required for each apprank and allocates these helpers to lightly-loaded nodes. The decisions are implemented by the runtime. Second, at medium granularity, the runtime employs BSC's Dynamic Load Balancing (DLB) [11] library to assign CPU cores to the appranks and active helpers on the same node. Finally, at fine granularity, the runtime instances dynamically offload tasks to helper ranks in order to fully utilize the allocated cores.

4 Distributed Taskiter

The main limits to the scalability of OmpSs-2@Cluster arise from the sequential creation of tasks and computation of their dependencies on Rank 0, as well as the centralized resolution of top-level task dependencies on the same rank. These bottlenecks are partially mitigated through strong support for task nesting, which increases effective task granularity, and through the *Namespace* optimization, which reduces the need for centralized dependency management. However, these mechanisms have largely been pushed to their practical limits within the current runtime design. A complementary approach is therefore to exploit structural regularities in the task graph itself, under programmer direction, enabling substantial reductions in the cost of task creation and dependency management.

Many scientific applications employ iterative methods or multi-step simulations in which the same directed acyclic task graph is executed repeatedly at each timestep or iteration. To address this common pattern, the *taskiter* construct was proposed in 2023 [3]. A loop can be annotated with `taskiter` provided that each iteration generates the same top-level dependency graph and the program remains valid if the code inside the loop body but outside any task is executed just once. The runtime instantiates the tasks once and represents the repeated execution of this acyclic structure as a cyclic task graph across iterations.

Distributed taskiter [18] extends this concept to OmpSs-2@Cluster. When the runtime encounters a loop annotated with `taskiter`, the loop is offloaded to all ranks, each of which locally instantiates the full task dependency graph. The runtime then partitions this cyclic graph across nodes, and each rank precomputes the MPI transfers in which it participates. Compared with MPI + OmpSs-2, the only overhead is the one-time initialization cost, after which the loop body is executed without any control messages. By integrating MPI communications directly into the application's task graph, distributed taskiter naturally overlaps computation and communication. Experimental results show that this approach achieves throughput matching or exceeding that of MPI + OpenMP. In some cases, for example 3D wave parallelism in the Gauss–Seidel heat equation, the asynchronous tasking approach exposes substantially more parallelism than fork–join MPI + OpenMP, and distributed taskiter achieves performance on par with state-of-the-art TAMPI [17] + OmpSs-2 (see [18]).

5 Conclusions

While task-based programming has proven effective at node level and workflow scale, our experience with OmpSs-2@Cluster confirms that extending fine-grained task graphs to distributed memory quickly encounters scalability limits related to centralized task creation and dependency management. Addressing these issues required substantial runtime engineering effort and a sequence of optimizations to enable practical scalability to tens of nodes.

The paper highlights two complementary directions in which distributed tasking provides tangible benefits. First, OmpSs-2@Cluster can be used selectively to mitigate residual load imbalance in hybrid MPI+OmpSs-2 applications. By combining task offloading with BSC's DLB library, our approach improves resource utilization with minimal disruption to existing application structure. Second, for applications with regular iterative structure, distributed taskiter demonstrates that exposing and exploiting task-graph regularity can fundamentally reduce runtime overheads. Similar ideas may apply to other kinds of task graph structure.

Overall, these results suggest that distributed tasking is most effective when applied judiciously, either as a targeted mechanism to address specific inefficiencies such as load imbalance, or in conjunction with programmer-provided structure that enables the runtime to avoid repeated control overheads. While OmpSs-2@Cluster cannot replace MPI as the dominant distributed-memory programming model for large-scale HPC, it demonstrates that task-based abstractions can deliver productivity, adaptability, and competitive performance when their limitations are explicitly acknowledged and addressed.

References

1 Jimmy Aguilar Mena, Omar Shaaban, Vicenç Beltran, Paul Carpenter, Eduard Ayguadé, and Jesus Labarta. OmpSs-2@Cluster: Distributed memory execution of nested OpenMP-style tasks. In *European Conference on Parallel Processing*, 2022. doi:10.1007/978-3-031-12597-3_20.

2 Emmanuel Agullo, Olivier Aumage, Mathieu Faverge, Nathalie Furmento, Florent Pruvost, Marc Sergent, and Samuel Paul Thibault. Achieving high performance on supercomputers with a sequential task-based programming model. *IEEE Transactions on Parallel and Distributed Systems*, 2017. doi:10.1109/TPDS.2017.2766064.

3 David Álvarez and Vicenç Beltran. Optimizing iterative data-flow scientific applications using directed cyclic graphs. *IEEE Access*, 2023. `doi:10.1109/ACCESS.2023.3269902`.

4 Barcelona Supercomputing Center. OmpSs-2 specification, 2021. URL: `https://pm.bsc.es/ftp/ompss-2/doc/spec/`.

5 Barcelona Supercomputing Center. OmpSs-2@Cluster releases, 2022. URL: `https://github.com/bsc-pm/ompss-2-cluster-releases`.

6 Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012. `doi:10.1109/SC.2012.71`.

7 George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Herault, Pitior Luszczek, and Jack Dongarra. Dense linear algebra on distributed heterogeneous hardware with a symbolic DAG approach. Technical report, Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), 2012. URL: `https://www.osti.gov/servlets/purl/1173290`.

8 J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguadé, and J. Labarta. Productive programming of GPU clusters with OmpSs. In *IEEE 26th International Parallel and Distributed Processing Symposium*, 2012. `doi:10.1109/IPDPS.2012.58`.

9 Ewa Deelman, Gurmeet Singh, Mei-Hui Su, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G. Berriman, John Good, Anastasia Laity, and Daniel S. Katz. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3), 2005. `doi:10.1155/2005/128026`.

10 Alejandro Duran, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. OmpSs: a proposal for programming heterogeneous multi-core architectures. *Parallel processing letters*, 21(02), 2011. `doi:10.1142/S0129626411000151`.

11 Marta Garcia, Julita Corbalan, R.M Badia, and Jesus Labarta. A dynamic load balancing approach with SMPSuperscalar and MPI. *Facing the Multicore - Challenge II. Lecture Notes in Computer Science, vol 7174*, 2012.

12 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*, 2017. `doi:10.1145/3148226.3148233`.

13 Francesc Lordan, Enric Tejedor, Jorge Ejarque, Roger Rafanell, Javier Alvarez, Fabrizio Marozzo, Daniele Lezzi, Raül Sirvent, Domenico Talia, and Rosa M Badia. ServiceSs: An interoperable programming framework for the cloud. *Journal of grid computing*, 12(1), 2014. `doi:10.1007/s10723-013-9272-5`.

14 Jimmy Aguilar Mena. *Methodology for malleable applications on distributed memory systems*. PhD thesis, Universitat Politècnica de Catalunya, 2022. URL: `http://dx.doi.org/10.5821/dissertation-2117-380814`.

15 J. M. Perez, V. Beltran, J. Labarta, and E. Ayguadé. Improving the integration of task nesting and dependencies in OpenMP. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2017. `doi:10.1109/IPDPS.2017.69`.

16 James Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., USA, 2007.

17 Kevin Sala, Xavier Teruel, Josep M Perez, Antonio J Peña, Vicenç Beltran, and Jesus Labarta. Integrating blocking and non-blocking MPI primitives with task-based programming models. *Parallel Computing*, 2019. `doi:10.1016/j.parco.2018.12.008`.

18 Omar Shaaban Ibrahim ali, Juliette Fournis d'Albiat, Isabel Piedrahita, Vicenç Beltran, Xavier Martorell, Paul Carpenter, Eduard Ayguadé, and Jesus Labarta. Leveraging iterative applications to improve the scalability of task-based programming models on distributed systems. *ACM Transactions on Architecture and Code Optimization*, 22(3):1–27, 2025. `doi:10.1145/3743134`.