Transparent load balancing of MPI programs using OmpSs-2@Cluster and DLB

Jimmy Aguilar Mena
Barcelona Supercomputing Center
Barcelona, Spain

Omar Shaaban
Barcelona Supercomputing Center
Barcelona, Spain

Victor Lopez
Barcelona Supercomputing Center
Barcelona, Spain

Marta Garcia
Barcelona Supercomputing Center
Barcelona, Spain

Paul Carpenter
Barcelona Supercomputing Center
Barcelona, Spain

Eduard Ayguade
Barcelona Supercomputing Center
Barcelona, Spain

Jesus Labarta
Barcelona Supercomputing Center
Barcelona, Spain

ABSTRACT

Abstract Load imbalance is a long-standing source of inefficiency in high performance computing. The situation has only got worse as applications and systems increase in complexity, e.g., adaptive mesh refinement, DVFS, memory hierarchies, power and thermal management, and manufacturing processes. Load balancing is often implemented in the application, but it obscures application logic and may need extensive code refactoring. This paper presents an automated and transparent dynamic load balancing approach for MPI applications with OmpSs-2 tasks, which relieves applications from this burden. Only local and trivial changes are required to the application. Our approach exploits the ability of OmpSs-2@Cluster to offload tasks for execution on other nodes, and it reallocates compute resources among ranks using the Dynamic Load Balancing (DLB) library. It employs LeWI to react to fine-grained load imbalances and DROM to address coarse-grained load imbalances by reserving cores on other nodes that can be reclaimed on demand. We use an expander graph to limit the amount of point-to-point communication and state. The results show 46% reduction in time-to-solution for micro-scale solid mechanics on 32 nodes and a 20% reduction beyond DLB for *n*-body on 16 nodes, when one node is running slow. A synthetic benchmark shows that performance is within 10% of optimal for an imbalance of up to 2.0 on 8 nodes. All software is released open source.

ACM Reference Format:
Jimmy Aguilar Mena, Omar Shaaban, Victor Lopez, Marta Garcia, Paul Carpenter, Eduard Ayguade, and Jesus Labarta. 2022. Transparent load balancing of MPI programs using OmpSs-2@Cluster and DLB.

The final publication is available at ACM via http://dx.doi.org/10.1145/3545008.3545045

1 INTRODUCTION

Load imbalance is one of the oldest and most important sources of inefficiency in high performance computing. The issue is only getting worse with increasing application complexity, e.g. adaptive mesh refinement (AMR) and modern numerics, and system complexity, e.g. DVFS, complex memory hierarchies, thermal and power management [2], OS noise, and manufacturing process [27]. Load balancing is usually applied in the application [6, 28, 36], but it is time consuming to implement, it obscures application logic and may need extensive code refactoring. It requires an expert in application analysis and may not be feasible in very dynamic codes. Static approaches like global partitioning are applied at fixed synchronization points that stop all other work [33, 43], and they rely on a cost model that may not be accurate. This task gets harder as the size of the clusters continues to increase.

This paper presents an automated approach for load balancing of MPI + OmpSs-2 [12] programs that relieves the application from the burden of balancing the load across nodes. We extend OmpSs-2@Cluster [3], which enables OmpSs-2 tasks to be offloaded to other nodes, to employ Dynamic Load Balancing (DLB) [22] mechanisms to handle fine-grained and coarse-grained load imbalance. Only minor and local changes are required to an MPI+OmpSs-2 program to make it compatible with our model, as no additional markup is required beyond the existing annotation of the task accesses. The single mechanism of task accesses is used to compute dependencies for parallel task execution, for data locality on the node, and for data transfers and locality optimizations on multiple nodes. We leverage the malleability of OmpSs-2 and OpenMP in terms of their ability to adjust to dynamically varying numbers of cores. Our approach uses DLB as the underlying mechanism through its Lend When Idle (LeWI) module [23] to react to fine-grained load imbalance and DLB's Dynamic Resource Ownership Management (DROM) module to address coarse-grained load imbalance by reserving cores on other nodes that can be reclaimed on demand.

A key aspect of our work is the use of an expander graph in which each MPI rank in the application directly offloads work to a small number of other nodes. This approach limits the amount of point-to-point communication and state that needs to be maintained, and it provides a path to scalability to large numbers of nodes.

We evaluate application-level imbalance using a micro-scale solid mechanics application with up to 32 nodes, and reduce the execution time by 46% compared with single-node DLB. This is only 7% above the theoretical time-to-solution with perfect load balancing. We evaluate system-level imbalance by executing an n-body Barnes–Hut simulation with two ranks per node on up to 16 nodes, one of which has a reduced clock frequency of 1.8 GHz, while the rest run at 3.0 GHz. DLB reduces the execution time by 16% and our approach reduces the execution time by a further 20%, with respect to the same baseline. This is an example where our dynamic approach operates in addition to the application's own load balancing technique, in this case Orthogonal Recursive Bisection (ORB), whose cost model does not adapt to varying node performance. We use a synthetic benchmark to show that our approach provides good load balancing across a wide range of application imbalance, within 10% of perfect load balancing for an imbalance of up to 2.0 on 8 nodes.

The main contributions of this paper are:

- An extension of the OmpSs-2@Cluster model to provide interoperability with MPI for the purposes of load balance.
- Mechanisms for fine-grained load imbalance (via LeWI) and coarse-grained load imbalance (via DROM). New core allocation algorithms, one using local convergence and the other using a global linear program formulation.
- A way to limit the amount of point-to-point communication and state by using an expander graph.

2 RELATED WORK

Load imbalance is a problem as old as parallel programming. It is crucial in distributed memory paradigms as the distributed data usually needs to be explicitly moved. There are many proposed solutions in the literature, most of them directly provided by the application itself, e.g., BT-MZ [28], discrete event simulation [36] or Monte Carlo simulations [6]. However, we consider a general solution that relieves applications from the load balancing burden. Moreover, it must be a solution that addresses all sources of load imbalance. In a general way, load balancing solutions can be divided into two categories, approaches applied before the execution and approaches that react during the execution. One of the best-known solutions to distribute the work before the execution is the mesh partitioner METIS [30]; some studies show that mesh partitioning strongly affects performance [44]. The main limitation of this approach is that it cannot handle a dynamic load imbalance that changes during the execution. Repartitioning methods are also based on mesh partitioning that is applied periodically during the execution [33, 43]. The two main challenges of these methods are that it is not trivial to determine a load heuristic to predict the load and that the repartitioning process is time-consuming. Thus, users must apply it with caution.

The other kind of methods are those that are applied at execution time. One of the most common approaches in this direction is to add a second level of parallelism based on shared memory. Several shared memory runtime systems implement dynamic work-sharing and work-stealing among threads to mitigate the effects of load imbalance in shared memory. The Intel Threading Building Blocks [41], currently known as oneTBB, is a C++ template library that provides functions, interfaces, and classes to parallelize an application. Cilk [16], now known as the OpenCilk project, is a programming language based on C and C++ designed for multithreaded parallel computing. OpenMP [38] is the most widely used shared memory programming model in HPC, the most recent related works to improve load balance consist of extending it to increase its malleability [34]. However, several studies [18, 25, 45] show that hybridizing can help improve load balance, but not in all situations, depending on the code, level of imbalance, the communication patterns, and the memory access patterns.

Several approaches choose to redistribute the data so that the load is better balanced. Charm++ [29] is an object-oriented parallel programming language that employs object migration to achieve load balance. The load balancing capabilities of Charm++ must be triggered manually or automatically after a load imbalance is detected. Moreover, Charm++ performs permanent migration, i.e., workpieces are migrated to a processing unit and will remain there until a potential new re-balancing step is executed. Adaptive MPI (AMPI) [15] is an implementation of MPI that uses the load balancing capabilities of Charm++. Balasubramaniam et al. [7] also propose a library that dynamically balances MPI processes by predicting the load and migrating the data accordingly. Martín et al. propose FLEX-MPI [46] an extension to MPI that, based on profiling information, will redistribute the data to improve the load balance. All these solutions need coarse-grained and rather persistent load imbalances to be efficient.

CHAMELEON [31] is a library for load balancing of task-parallel MPI+OpenMP programs. It uses the OpenMP target offloading construct to define task data accesses, and data is copied back to the parent after task execution. It is the closest approach to ours. In comparison with CHAMELEON, OmpSs-2@Cluster supports task dependencies, using the same data access specifications to describe task dependencies and data locality/copies. We execute the task in a context with the same virtual address space, simplifying the porting and debugging of programs that work on data structures with pointers. We use a small number of helpers per MPI rank in the program, organized as a sparse expander graph, so as to limit the amount of communication and coordination among MPI ranks. Whereas CHAMELEON is only reactive to fine-grained load imbalance, OmpSs-2@Cluster uses DLB's LeWI module [23] to react to fine-grained load imbalance as well as DLB's DROM module [19] to reserve cores to address coarse-grained load imbalance.

Load balancing is of concern across a wide range of distributed systems, beyond HPC. Examples include web search [21] and web servers [37]. Ingress [1] is a load balancer for Kubernetes [32], which balances requests from the network across multiple Kubernetes pods. These approaches are optimized for external mostly independent items of work, whereas our approach is tailored for HPC jobs, which are tightly-coupled batch jobs.

3 BACKGROUND

3.1 OmpSs-2, Nanos6 and Mercurium

OmpSs-2 [12] is the second generation of the OmpSs programming model. It is open source and mainly used as a research platform to explore and demonstrate ideas that may be proposed for future standardization in OpenMP. Like OpenMP, OmpSs-2 is based on directives and it enables parallelism in a dataflow way [40]. OmpSs-2

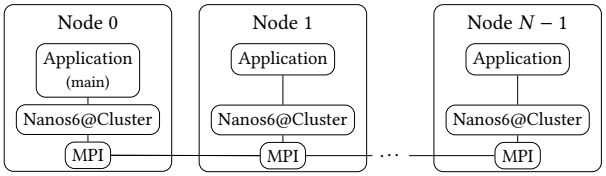


Figure 1: OmpSs-2@Cluster architecture in which each node is a peer. The main function runs as a task on Node 0.

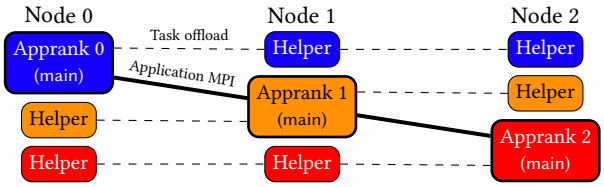


Figure 2: Architecture of MPI+OmpSs-2@Cluster. Application ranks (appranks) communicate via MPI and helper ranks on some other nodes can execute tasks of heavily loaded appranks. The main function runs as tasks on the appranks.

differs from OpenMP in the thread-pool execution model, targeting of heterogeneous architectures through native kernels, and asynchronous parallelism as the main mechanism to express concurrency. Task data accesses are used as a single mechanism to compute dependencies for task ordering and to determine data locality/data copies. OmpSs-2 extends OmpSs and OpenMP to improve task nesting and fine-grained dependences across nesting levels [4, 39].

The reference implementation comprises the source-to-source Mercurium [9] compiler and the Nanos6 [10] runtime. The runtime computes task dependencies and schedules and executes tasks, respecting the implied task dependency constraints and performing data transfers and synchronizations.

3.2 OmpSs-2@Cluster

OmpSs-2@Cluster [3, 13] is the task offloading extension of OmpSs-2, which extends OmpSs-2 tasking across multiple nodes. Any OmpSs-2 program with a full specification of task dependencies is compatible with OmpSs-2@Cluster. A functional multi-node version of an existing OmpSs-2 program can usually be obtained by enabling task offload via the configuration file provided to the runtime system. Improvements beyond the first version can be made incrementally, based on observations from performance analysis.

OmpSs-2@Cluster inherits task ordering from a sequential version of the code and it uses a common virtual memory layout across workers, which avoids address translation and allows direct use of existing data structures with pointers. All tasks can be marked as offloadable (to another node) or not (fixed on the same node as the task's parent). Task scheduling is flexible, so that an offloadable task created on any node may be executed locally or offloaded to any other node (it cannot be migrated once started). Offloadable tasks may have dependencies just like any other tasks, and the satisfiability of dependencies and data location are passed through a distributed dependency graph. Data copies are done eagerly where required, so there is no automatic write-back to the original node, unless the data value is needed by a task or a taskwait. Each node runs an instance of the Nanos6 runtime, which coordinate as peers as shown in Figure 1. The underlying communication for control messages and data transfers is done using MPI. The instances of the runtime system overlap the construction of a distributed dependency graph, enforcing of dependencies, task scheduling, transferring of data, and task execution.

3.3 Dynamic Load Balancing (DLB)

DLB [22] is a library that enables dynamic load balancing among multiple processes on the same node, from either the same or different applications. It exploits the malleability features of OmpSs-2 and OpenMP; i.e., the ability to dynamically adapt to varying resources at runtime, in this case the number of cores. DLB is organized into modules, each of which provides an independent service that is compatible with and complementary to the other modules. The currently available modules are Lend When Idle (LeWI) [23], Dynamic Resource Ownership Management (DROM) [19] and Tracking Application Live Performance (TALP) [35].

LeWI enables fine-grained load balancing by providing a straightforward API to lend cores when they would otherwise be idle and to borrow them, when needed, by a different process on the node. Crucially, the lender may reclaim the cores as soon as they are needed again. Under the control of the runtime (either Nanos6 or OpenMP), LeWI provides the mechanism to respond to a fine-grained imbalance. DROM enables coarser-grained load balancing by providing an API to change the semi-permanent ownership of cores among the processes on the node. Ownership of cores in proportion to the average load provides the ability for processes to reclaim the right number of cores when they are needed. TALP is another module that measures the parallel efficiency by intercepting MPI calls. The data obtained by TALP is available to the application at runtime and it can be output as a report at the end.

4 PROGRAMMER'S MODEL

The OmpSs-2@Cluster programmer's model is extended to support interoperability with MPI so it can be used for load balancing of MPI+OmpSs-2 programs. The overall approach is illustrated in Figure 2. As for any MPI program, the application's main function executes in SPMD (Single Program Multiple Data) fashion across the compute nodes. The MPI ranks visible to the application are known as appranks (or application ranks), and they communicate in the normal way using MPI (solid line in Figure 2). Unlike regular MPI, however, each application rank (or apprank) is supported by a small number of helper ranks on other nodes. These helper ranks (connected by dashed lines) can execute tasks offloaded by their apprank. Suitable tasks are defined using regular OmpSs-2 task annotations. Figure 3 shows an example n-body kernel that is a slightly simplified extract from the n-body application used in the evaluation. There is a single offloadable task that calculates the forces on a number of bodies. The pragma annotation defines the task together with its inputs and outputs.

The application is compiled in the same way as any MPI+OmpSs-2 program, using Mercurium, and it is executed in the normal way for an MPI program, using an extra configuration parameter to the runtime system to enable task offloading.

```
for(int j=0; j < num_bodies; j+= block_size) {
  int n = MIN(block_size, num_bodies-j);
  #pragma oss task in(bodies[j;n]) out(forces[j;n]) \
     in(cells[0;num_cells])
  for (int k=j; k<j+n; k++) {
     // Compute forces[k]
  }
}
```

Figure 3: Example OmpSs-2@Cluster code for *n*-body kernel

Two superficial changes are required in the application source code to support MPI+OmpSs-2@Cluster. Firstly, all occurrences of MPI_COMM_WORLD should be replaced with a call to the relevant runtime API call, nanos6_app_communicator(), to obtain the communicator to be used by the application. Secondly, since the runtime requires MPI communication before and after the execution of the main function, the application should not itself call MPI_Init() or MPI_Finalize(), as these calls are made by the runtime. Making these changes in the application, rather than through a custom `mpi.h` passed to the application ensures portability across all systems and maintains the OmpSs-2 property that code without OmpSs-2 pragmas may be compiled directly with the host compiler.

MPI calls on the application's communicator are valid, so long as the task and all its ancestors (i.e. its parent, parent's parent, etc.) are non-offloadable. This is consistent with the normal way of developing MPI+OpenMP or MPI+OmpSs-2 programs, which use tasks to perform compute operations, but not communication, due to the risk of deadlock. The intention is to avoid having to intercept MPI calls to manage MPI communication among offloaded tasks.

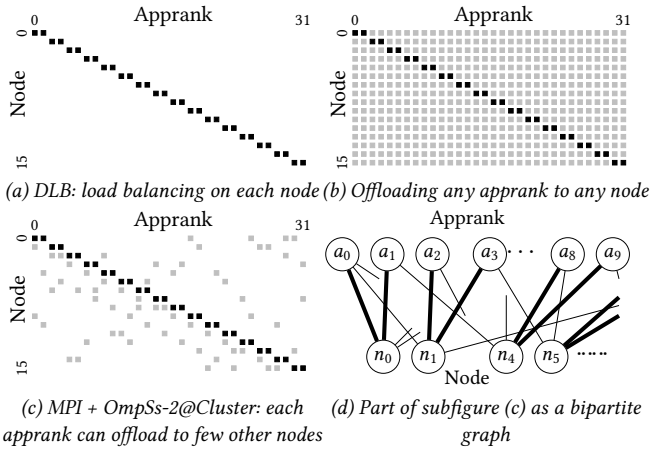
We maintain the property of regular OmpSs-2@Cluster that tasks are always executed in a process with the same virtual address structure as their apprank [3]. The different appranks have isolated virtual address spaces, so that different objects on different appranks may be allocated to the same virtual address, even if they are both accessed by tasks executed (within different appranks) on the same physical compute node. This simplifies the porting of applications that use global or static data or that use libraries that do so. It allows the programmer to not have to worry about placing global and static data at different addresses, which could otherwise cause bugs that are difficult to track down.

5 IMPLEMENTATION

5.1 Overall architecture

In Figure 2, each apprank and helper rank is a process with multiple threads that can execute tasks. If the application is well-balanced, then only the appranks are involved in the computation, in the same way as any MPI+OmpSs-2 program. In this case, the helper ranks will remain idle. As normal, multiple appranks on the same node can respond to small load imbalances, using DLB, by shifting cores to the appranks with more load. At some point, however, it will become necessary to break the confinement of a single node and offload work to helper ranks on other nodes.

The aim is not merely to spread the load of a single heavily loaded apprank or node, but to avoid any bottlenecks where a local load imbalance can get "stuck" within a single node or a small group of nodes. At the same time, we wish to minimise the number of helper ranks in the system, since each helper rank implies point-to-point communication and state and scheduling complexity.



(a) DLB: load balancing on each node *(b) Offloading any apprank to any node*

(c) MPI + OmpSs-2@Cluster: each apprank can offload to few other nodes *(d) Part of subfigure (c) as a bipartite graph*

Figure 4: Work spreading of 32 appranks on 16 nodes. Offloading tasks to a small number of other nodes allows the work to be spread across all nodes used by the application.

5.2 Spreading of work

Figure 4(a) represents the mapping of appranks (application ranks) to nodes when an application executes with 32 appranks on 16 nodes, two appranks per node. Assuming that DLB is enabled, the load can be balanced among the two appranks (black squares) on the same node, but any load imbalance is confined to a node. Figure 4(b) is the opposite extreme, where each apprank executes its main function on the same rank as before (black square). It can still balance the load with the other apprank on the same node, but it can also offload tasks to any other node (grey squares in the same column). This configuration provides the maximum ability to spread work, but it requires a lot of state for point-to-point communication and for each rank to keep track of where to send the work. Changing behaviour of the different appranks in response to each other causes a lot of variability in the execution. Finally, Figure 4(c) is an example following the philosophy of this paper: each apprank executes tasks across a small number of nodes, providing the ability to spread work while limiting the amount of state and variability.

Static work spreading: The simplest approach is to allow each apprank to directly offload tasks to a small number of other nodes that have been chosen before the application begins execution. A large body of work exists on *expander graphs*, which have the properties we need, so we first translate the problem into the language of graph theory. Rather than arranging the appranks and nodes into a grid, we define a bipartite graph of appranks and nodes and draw an edge between an apprank and a node if the apprank can execute tasks on that node. Figure 4(d) shows part of the graph representation of the scenario from Figure 4(c). Not all edges are drawn, as in the full graph, each apprank has degree three and each node has degree six. Each edge corresponds to a worker process. The apprank itself is identified using a thicker solid line and the helpers are identified with thinner lines.

There are several definitions of an expander graph in the literature [26], but for our purposes we define a bipartite expander graph as a bipartite graph for which $|N(A)| \geq (1+\epsilon)|A|$ for some large $\epsilon > 0$, for every subset A of at most half of one of the partitions of the graph [20]. In our context, A is a subset of the appranks and $N(A)$ is the set of all nodes that are adjacent to at least one apprank of A. $|A|$ is the number of appranks under consideration and $|N(A)|$ is the number of nodes over which the work of A can be divided. The definition means that the work belonging to each subset of the appranks can, in principle, be spread across a good number of nodes, which grows with the number of appranks. This achieves our aim of avoiding bottlenecks on the ability to spread local load imbalances. The definition is a strict one that applies to every subset, not merely a probabilistic one, so no matter where the load imbalances arise, the work can be spread out across a good number of nodes.

It is well-known that a large randomly-chosen graph is an expander graph with high probability [17, 20]. We add the constraints that each apprank has the same number of incident edges, as do the nodes (it is *bipartite biregular*), and generate a random graph according to these constraints. Small graphs are generated using a heuristic-based search or known-optimal solution. For small graphs up to about 32 nodes, we also run some checks to avoid bad graphs, i.e., with limited connectivity, by calculating the vertex isoperimetric number (the minimal value of $1+\epsilon$ in the above equation). Each graph is stored for future executions so that it is only created once.

The number of edges per apprank is a user-provided parameter, known as the *offloading degree*. An offloading degree of one corresponds to the baseline without task offloading. An offloading degree of two means that each apprank can execute tasks on its main node and one additional node. Since the offloading degree is known in advance, as is the assignment of appranks and helper ranks to nodes, the initialization of all Nanos6 instances is done at initialization time. The results are generally insensitive to the offloading degree, so long as it is large enough to accomodate the application's level of imbalance and to provide enough connectivity given the number of nodes. It can be set as recommended in Section 7.3.

Dynamic work spreading: The static approach has a parameter (the offloading degree), which must be provided by the user. Depending on the value of this parameter, a fixed number of helper ranks is created at the beginning of the execution. These helper ranks require at least one core each, leaving fewer resources for the actual computation, even if the helper ranks turn out to not be required. Moreover, the optimal bipartite graph depends on the application and input data. It could also take account of specific communication latencies and thereby depend on the physical topology of the nodes allocated to the application.

A better approach may therefore be to grow the expander graph dynamically. This would allow the execution to adapt to the program and system characteristics, and it would remove the offloading degree parameter. Doing so is compatible with all the contributions of this paper, and is a natural extension of our work. The main change to the runtime would be to extend it to support dynamic process spawning. Our experience, however, discussed in Section 7.3, shows that the benefit would likely not be sufficient to compensate for the extra implementation and evaluation complexity.

5.3 Fine-grained load balancing via LeWI

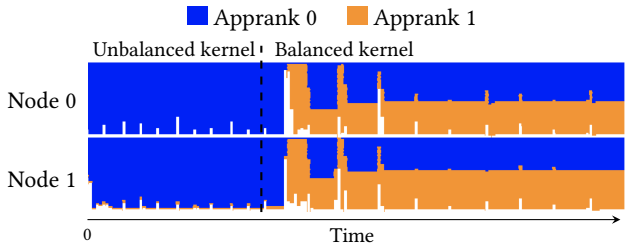
At any time, a fine-grained load imbalance may appear among the workers on a node. LeWI is the mechanism that allows a worker process to lend otherwise idle cores to another process on the same node that can make use of them. As the borrower process completes tasks more quickly, it can be scheduled more tasks from the main (or other) process, as explained in Section 5.5.

5.4 Coarse-grained load balancing via DROM

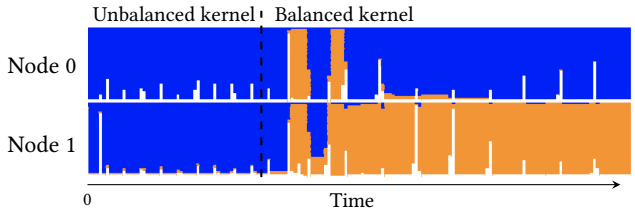
At any time, each CPU core is owned by one of the appranks or helper ranks executing on that node. At the beginning of the execution, each helper rank owns one core (the minimum possible with DLB), and ownership of the remaining cores is divided equally among the appranks on the node. On MareNostrum 4, for example, which has 48 cores per node, each helper rank of Figure 4(c) starts with one owned core and each apprank starts with 22 owned cores. Ownership of cores is updated dynamically as the execution progresses. We propose two approaches for doing so, a local convergence approach and a global solver approach, both described below.

5.4.1 Local convergence approach. As the program executes, each apprank's task scheduler (Section 5.5) tries to balance the load by exploiting all cores that are assigned to the apprank. At the same time, the local convergence approach tries to balance the load-per-core among the workers running on each node. Both processes are local to the apprank or node. Each worker measures its average number of busy cores, i.e., the average number of cores executing tasks or runtime code except the idle loop. The workers on a node coordinate to ensure that all cores have an owner and that the number of cores owned by each worker is proportional to its average number of busy cores. The combined effect of these two mechanisms is to try to keep all cores in the system equally busy.

This approach is simple to implement and understand, it has no global communication and low overhead, and it does a good job of balancing the loads among the appranks. It is not guaranteed, however, to minimise the amount of task offloading. Figure 5(a) shows an example with two appranks on two nodes. The x-axis is time and the y-axis shows the number of cores executing for Apprank 0, in blue, and Apprank 1, in orange. Node 0 is shown in the top half of the trace and Node 1 is shown in the bottom half of the trace. The first half of the execution has an unbalanced load with almost all computation on Apprank 0. We see that the local approach almost immediately starts executing the tasks of Apprank 0 on both nodes, making full use of the computational resources. The second half of the execution, however, has an equally balanced load across the two appranks. Since before this point, both nodes have almost all cores owned by Apprank 0, the first expensive tasks from Apprank 1 must wait for cores to become available. Once the work of Apprank 0 is complete, the tasks of Apprank 1 are scheduled across both nodes. Both nodes see the same pattern of load, and they react in the same way, by increasing the number of cores owned by Apprank 1, to converge towards equal ownership. The outcome is that Apprank 0 offloads half of its tasks from Node 0 to Node 1 and Apprank 1 offloads half of its tasks from Node 1 to Node 0. It is clearly unnecessary to offload tasks when the load is balanced, and Figure 5(b) shows the optimal approach. It starts in the same way as Figure 5(a), but once the load becomes balanced,



(a) Local convergence approach



(b) Global solver approach

Figure 5: Coarse-grained load balancing: The local approach balances the load but does not minimise task offloading, as both appranks of the balanced kernel execute tasks across both nodes. The global approach balances the load and minimises task offloading at the cost of global communication.

there is no unnecessary offloading of tasks. It was obtained using the global solver approach, described in the next section.

5.4.2 Global solver approach. The global approach employs an external solver to determine how many cores should be allocated to each worker process. Similarly to the local approach, it uses the number of busy cores as an estimate of the amount of work, but the work is summed across all workers in the apprank. It then uses a linear program formulation to minimise the value of:

$$\max_{\text{Apprank } a} \frac{\text{Total work on } a}{\text{Total cores on } a}, \tag{1}$$

subject to the constraints that each worker owns at least one core, the sum of owned cores on each node is at most the number of physical cores, and that each apprank may only own cores that it is adjacent to in the bipartite expander graph, e.g., in Figure 4(d). This is an Integer Linear Program, because the numbers of owned cores must be integers, but it is sufficient to solve the continuous problem and round to an integer number of owned cores per worker that sums to the total number of physical cores. A heuristic counts the cores on the apprank itself as being marginally faster, in order to prefer to not offload unless necessary. The precise value of this incentive is not critical, as the solver will tend to take it no matter how small, and we use a value of one part in 10^{-6}, i.e. the "Total work on a" in the numerator of Equation 1 is the sum of the non-offloaded work and $1 + 10^{-6}$ times the offloaded work. Since the number of busy cores includes runtime overheads, the global policy is able to converge to a solution that provides additional resources, if necessary, for runtime execution overheads.

The global policy has the advantage that it will always find an optimal solution that balances the load and minimises task offloading. Its disadvantages are that it requires periodic global communication and it centralizes the work of determining the core allocation onto a single node. The implementation has a separate Python process using CVXOPT [5], and it executes the global solver every two seconds. We always run the solver on the first node, which in many cases happens to be the highest loaded node. But it could of course be migrated to the least loaded node. The time to solve the global allocation problems for the 32-node experiments in Section 7 is approximately 57 ms. Running the solver every two seconds gives an overhead of about 6%. A single global solver process is therefore sufficient for up to about 64 nodes.

Since the time to solve the linear program grows approximately quadratically with the size of the graph, larger graphs than 32 nodes should be partitioned and solved in parts on multiple nodes. These 32-node groups are very likely to contain heavily and lightly loaded nodes and allow almost complete load balancing. It is a significant improvement above the existing DLB approach, which only supports load balancing among the processes on a single node.

5.5 Task scheduling

To maintain load balance, the scheduler makes a tentative scheduling decision whenever a task becomes ready. If the best node, according to data locality, currently has fewer than two tasks per-core, then the task is scheduled to that node immediately. Otherwise, it there is an alternative node with fewer than two tasks per core, the task will be immediately scheduled to that node instead. Two tasks per core allows one task to be executing and another to have the data transfer (if any) initiated in advance and be blocked ready to execute as soon as the executing task finishes. If all nodes already have at least two tasks per core, then the newly ready task is held in a queue, and will be stolen as tasks complete.

When calculating the number of tasks per core, the number of cores is the number owned via DROM. It does not take account of any short-term lending or borrowing of cores via LeWI. This is because the lent cores are not really gone, as they can easily be reclaimed if needed, while borrowed cores may have to be returned at any moment. Offloading a task is a "final" decision because once a task has been offloaded to a node, it cannot be recalled and it cannot be rescheduled or migrated to another node. By not taking temporary cores for granted, we ensure that there are always sufficient cores to execute the offloaded tasks in a timely manner.

6 METHODOLOGY

6.1 Quantifying imbalance

We quantify the application's load imbalance using the imbalance:

$$\mathit{Imbalance} = \frac{\text{Maximum}_{\mathit{apprank}}\ \mathit{load}}{\text{Average}_{\mathit{apprank}}\ \mathit{load}} \geq 1. \tag{2}$$

This metric is dimensionless, and it directly relates the maximum load, which gives a lower bound on the length of the critical path, to the average load, which estimates the length of the critical path with perfect load balance. It is preferable to other metrics, such as the standard deviation of loads, that have no direct connection to the problem to be solved. As formulated the load imbalance ignores any imbalance among the cores due to task scheduling. An imbalance of 1.0 is perfect load balance, whereas an imbalance of 2.0 indicates that the critical path has roughly twice the length that it would have with a perfect load balance. The maximum possible

value for the imbalance is the number of appranks, which would correspond to the case where all of the work is on one apprank.

6.2 Benchmarks

We use two application programs, Alya MicroPP and *n*-body. Alya MicroPP is a 3D finite element library for micro-scale solid mechanics in composite materials [24]. The code has unbalanced execution due to the mix of linear and non-linear finite elements. The *n*-body code [14] is a parallel implementation of Barnes–Hut [42] using Orthogonal Recursive Bisection to equalise the work across the ranks. Both applications are implemented in C++ with parallelization using MPI and OpenMP/OmpSs-2.

We also use a synthetic benchmark to increase confidence that our approach works for a range of scenarios beyond those found in the applications. The synthetic benchmark has a configurable imbalance (Equation 2). Each iteration of the program has 100 tasks per core, of average duration 50 ms. The task durations are different on the different appranks to meet the target imbalance. The execution time of the tasks on the worst-case rank is 50 ms multiplied by the target imbalance. The other execution times are uniformly distributed over the space of values respecting the constraints.

6.3 Hardware platform

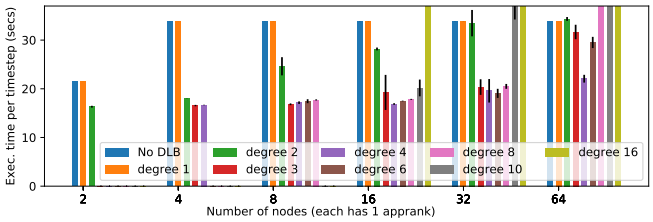
Most experiments in this paper were performed on up to 64 nodes of the general-purpose block of the MareNostrum 4 supercomputer [8]. MareNostrum 4 comprises 3456 compute nodes, each with two 24-core Intel Xeon Platinum sockets. We use normal memory capacity nodes, which have 96 GB physical memory (2 GB per core). The interconnect is 100 Gb Intel Omni-Path with a full-fat tree. The experiments with a slow node were performed on Nord3 [11], which has a newer version of Slurm that supports heterogeneous allocations, as needed to run different nodes of the same job at different clock frequencies. Nord3 has 756 compute nodes, each with two 8-core Intel E5-2670 SandyBridge sockets at 3.0 GHz (normal) or 1.8 GHz (slow).

7 RESULTS

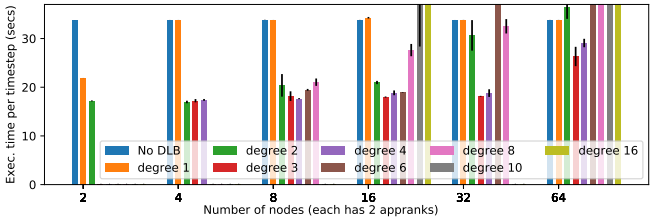
7.1 Application performance

Figure 6 shows the application performance results for MicroPP and *n*-body weak scaling. The *y*-axis is the execution time and the *x*-axis is the number of nodes. Figure 6(a) and (b) show the results for MicroPP on two to 64 nodes of MareNostrum 4. Figure 6(a) has one apprank per node and Figure 6(b) has two appranks per node, i.e. 4 to 64 appranks. The baseline result (blue) is without task offloading or DLB. When there is just one apprank per node, single-node DLB makes no difference, as expected. When there are two appranks per node, the benefit from DLB alone (degree one) is also generally small, because some of the heavily loaded ranks share a node. All baseline and degree one results have no helper ranks, so all cores are used for computation.

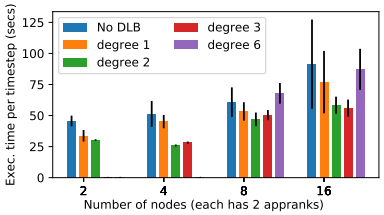
Enabling task offloading via OmpSs-2@Cluster, however, makes a large improvement to performance in all situations. Assuming a moderate offloading degree of four, the time-to-solution is reduced by 49% on 4 nodes and 47% on 32 nodes, compared with DLB. Both are close to the perfect load balancing. Offload to a single extra node (degree two) has good results for small numbers of nodes, but as the number of nodes increases the limited graph connectivity becomes



(a) MicroPP on MareNostrum 4 (1 apprank per node)



(b) MicroPP on MareNostrum 4 (2 appranks per node)



(c) n-body on Nord3 with one slow node (2 appranks per node)

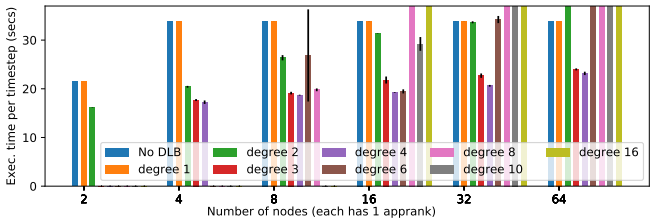
Figure 6: MicroPP and *n*-body application performance with global allocation policy. With an offloading degree of four, i.e., when each apprank can execute tasks across four nodes including its own, the global policy improves load balance beyond DLB, for all configurations.

a constraint on the ability to balance the load. An offloading degree of three or four provides good results in all situations. Increasing the offloading degree to an excessive value, of eight or more, starts to affect performance, justifying the design decision to use few helper ranks per apprank.

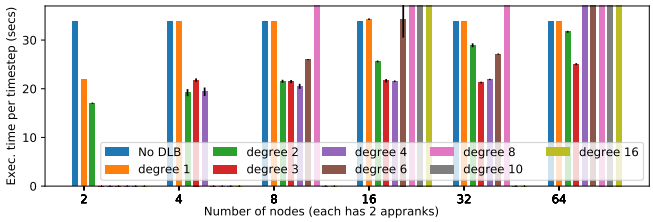
Figure 6(c) shows the results for *n*-body on Nord3 with one slow node. *n*-body is in itself a balanced application, as it uses Orthogonal Recursive Bisection (ORB) each timestep to rebalance the work. In this example with a slow node, however, ORB does not perform well. Here, on 16 nodes and two appranks per node, we see a 16% improvement when employing single-node DLB and a further 20% improvement (with respect to the same baseline) when enabling OmpSs-2@Cluster task offloading with degree 3. The single apprank per node results are not shown as the baseline performance was much worse, likely due to an issue with scheduling tasks across the two NUMA nodes. Nord3 has an older CPU architecture, with 16 cores per node, so with two appranks per node, the offloading degree should be at most four. From both applications, the conclusion is that the global core allocation achieves excellent improvements in load balance, with an offloading degree of four.

7.2 Local core allocation

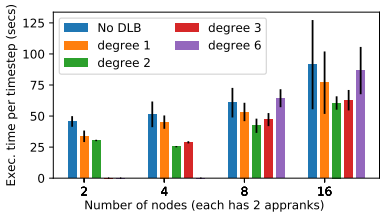
Figure 7 shows the results using the local allocation policy. Overall, local allocation performs slightly worse than global allocation due



(a) MicroPP on MareNostrum 4 (1 apprank per node)



(b) MicroPP on MareNostrum 4 (2 appranks per node)



(c) n-body on Nord3 with one slow node (2 appranks per node)

Figure 7: MicroPP and *n*-body performance with local allocation policy. The local policy has about 10% higher optimal execution time for MicroPP on 32 nodes than the global policy and it is generally more sensitive to the offloading degree (number of nodes among which each apprank can execute tasks).

to the issue with unnecessary task offloading (Section 5.4.1). We see that the local policy provides similar results for small numbers of nodes: reducing the time for two appranks per node on 4 nodes by 43%. But the local policy tends to offload too many tasks, and on 32 nodes the improvement is only 38%, compared with 47% found above for the global policy. It also tends to be more sensitive to a well-tuned offloading degree, with execution time rising for an offloading degree above 4 for MicroPP.

7.3 Sensitivity analysis on kernel imbalance

Figure 8 is a sweep of the execution time, as a function of the imbalance, for the synthetic test program (Section 6.2). The x-axis is the imbalance, defined in Equation 2, and the y-axis is the execution time per iteration, in seconds. In all three subplots, the baseline MPI+OmpSs-2 program with single-node DLB is indicated with the blue line. Since there is one apprank per node, there is no benefit from single-node DLB, so the case without DLB is not shown.

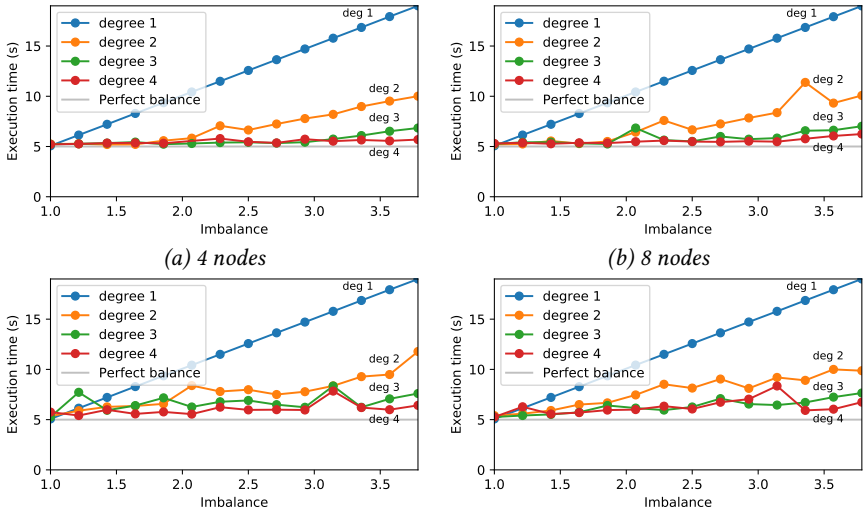
Figures 8(a), (b) and (c) show that, on four to 64 nodes, an offloading degree of four (red) provides the best results across the whole range of application imbalance from 1.0 to 4.0. This is similar to MicroPP, where an offloading degree of four also generally provided the best results. For small numbers of nodes, up to about eight nodes, it is sufficient for the offloading degree to be at least as large as the imbalance. This is clearly seen in Figure 8(a), on four nodes, where an offloading degree of 2 is sufficient for up to an imbalance of 2.0 and an offloading degree of 3 is sufficient for an imbalance up to 3.0. But there is no penalty for a moderately higher offloading degree. For larger numbers of nodes, the graph connectivity becomes an issue. On 64 nodes, an offloading degree of 4 provides the most dependable results even for small levels of imbalance, and is within 20% of optimal for imbalances in the range 1.0 to 2.0. The conclusion is that for up to 64 nodes, an offloading degree of four is sufficient, which is dramatically lower than full connectivity (an offloading degree of 64). The fact that there is no benefit for smaller offloading degrees when the imbalance is small supports our claim that a static expander graph is sufficient (Section 5.2).

7.4 Fine-grain (LeWI) and coarse-grain (DROM)

In order to understand the role of LeWI and DROM, Figure 9 shows execution traces for MicroPP with and without LeWI and DROM, with four appranks on four nodes. The outcome is similar for larger numbers of nodes, but this example gives more intelligible traces. The x-axis is time, and all timelines have the same scale, so the length of the trace is proportional to execution time. The y-axis indicates the number of cores for each apprank (colours), busy executing tasks or non-idle-loop runtime code (left-hand traces) or owned (right-hand traces). The four nodes are shown from top to bottom.

Figure 9(a) shows the original MPI+OmpSs-2 trace, in which there is a clear imbalance among the nodes, due to the greater amount of work done by Apprank 0. We see in Figure 9(b) that each apprank owns the cores on its node. Figure 9(c) employs LeWI, but not DROM. In this case, once Apprank 1 finishes an iteration, LeWI reacts to the fine-grained load imbalance by allowing tasks from Apprank 0 to be offloaded to Node 1, and Apprank 0 executes its tasks across both nodes. The use of remote cores when temporarily borrowing cores is well under 100% due to the issue described in Section 5.5. It is important not to be too aggressive when making use of borrowed cores, because the cores could be reclaimed at any moment. In fact, Apprank 1 reclaims its cores at the beginning of the next iteration. Figure 9(d) shows the same static assignment of core ownership, since LeWI does not change the ownership of cores. The execution time is reduced to 83% of that of the baseline.

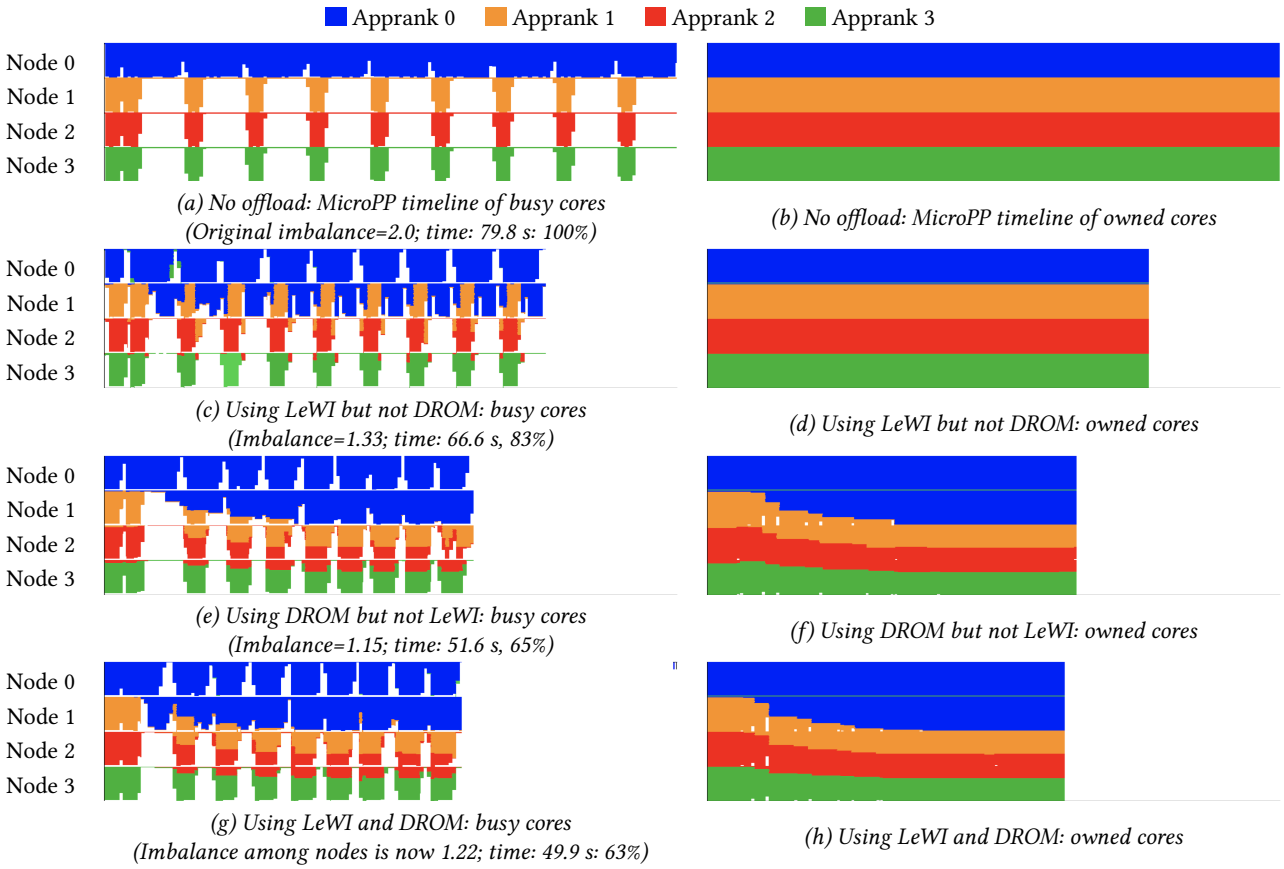
Figure 9(e) employs DROM rather than LeWI. DROM updates the ownership of cores to ensure load balance and high utilization in the later iterations. After a few iterations, almost all cores on Node 0 and Node 1 are consistently used by Apprank 0. In order to let this happen, Apprank 1 shifts its work to Node 2. The trace shown in Figure 9(e) uses the global core allocation policy, but the same effect occurs with the local policy. We see that the execution time is reduced further, to 65% of that of the baseline execution. Figure 9(f) shows how the core ownership converges to this optimal result. Finally, Figure 9(g) employs both LeWI and DROM. LeWI is active in the first iteration, by reacting to the load imbalance immediately. But DROM quickly adjusts to the steady-state imbalance, ensuring an optimal load balance in later iterations. This shows how LeWI and DROM complement each other and together give the best execution.



(a) 4 nodes *(b) 8 nodes*

(c) 32 nodes *(d) 64 nodes*

Figure 8: Execution time of synthetic application, with one apprank per node, using LeWI and DROM, as a function of the imbalance. An offloading degree of four provides consistently good results on up to 64 nodes.



(a) No offload: MicroPP timeline of busy cores (Original imbalance=2.0; time: 79.8 s: 100%)

(b) No offload: MicroPP timeline of owned cores

(c) Using LeWI but not DROM: busy cores (Imbalance=1.33; time: 66.6 s, 83%)

(d) Using LeWI but not DROM: owned cores

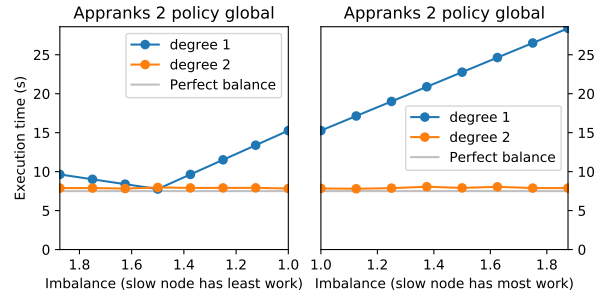
(e) Using DROM but not LeWI: busy cores (Imbalance=1.15; time: 51.6 s, 65%)

(f) Using DROM but not LeWI: owned cores

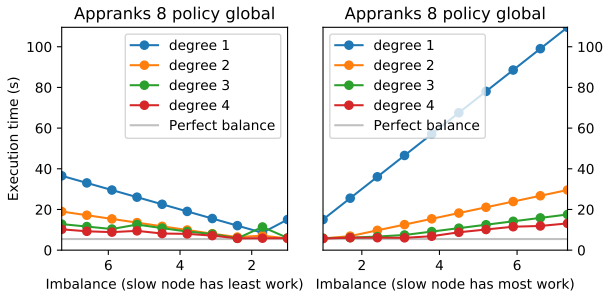
(g) Using LeWI and DROM: busy cores (Imbalance among nodes is now 1.22; time: 49.9 s: 63%)

(h) Using LeWI and DROM: owned cores

Figure 9: Traces of Alya MicroPP on four nodes with degree two. LeWI allows Apprank 0 to borrow cores on Node 1 when they would otherwise be idle. DROM adapts the long-term ownership of cores to address the steady load imbalance.

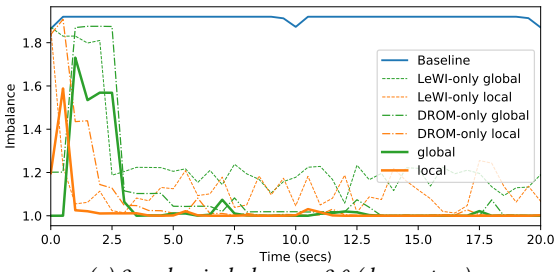


(a) 2 nodes

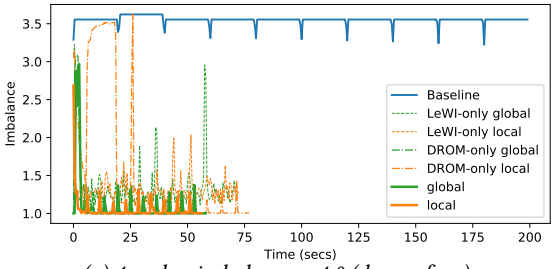


(b) 8 nodes

Figure 10: Synthetic test program with one emulated slow node (3 times slower than the other nodes), showing that tasks are offloaded to balance the loads.



(a) 2 nodes, imbalance = 2.0 (degree two)



(a) 4 nodes, imbalance = 4.0 (degree four)

Figure 11: Convergence of imbalance among nodes for synthetic benchmark. Local converges faster than global when LeWI is enabled. DROM is essential to reduce the node imbalance to close to 1.0.

7.5 Sensitivity analysis with slow node

Figure 10 shows a sweep of the execution time as a function of the imbalance, for a synthetic test case with one slow node that is three times slower than the other nodes. Being a synthetic example unlike *n*-body, it is not actually a slow node, just emulated by the task durations. The *y*-axis is the execution time per iteration, and the *x*-axis is the imbalance. Increasing imbalance to the left indicates the case where the slow node has the *least* work and increasing imbalance to the right indicates that the slow node has the *most* work. Figure 10(a) has two nodes, and a degree of 2 has almost flat execution time per iteration, close to the optimal (grey line) across the whole range of imbalance. With eight nodes, we see a similar story to before. When the slow node has the most work (to the right), the execution time is close to flat, as long as the offloading degree is a little higher than the imbalance. As before, we see that on offloading degree of four provides the best and most consistent results, and it is able to handle the imbalance up to an imbalance of 4.0.

7.6 Convergence with synthetic benchmark

Figure 11 shows time series plots for the synthetic benchmark: two nodes with an imbalance of 2.0 and four nodes with an imbalance of 4.0. The *x*-axis is time and the *y*-axis is the imbalance among the nodes, given by $(\text{Maximum}_{node}\ load)/\ (\text{Average}_{node}\ load)$. The current load is the total average number of busy cores (see Section 5.4), meaning that the imbalance among nodes is updated more frequently than application-level measurements.

We see a similar picture in both subplots of Figure 11. Both the local and global policies, when using DROM (with or without LeWI) are able to reduce the imbalance among the nodes to close to 1.0. Using LeWI but not DROM, the imbalance fluctuates around 1.2 in both scenarios, which is consistent with the behaviour observed in the MicroPP traces (Section 7.4). The local policy converges quicker than the global policy, as it operates continuously whereas the global policy is updated every two seconds. We also see that LeWI helps to accelerate the speed of convergence of the local policy, in both scenarios, since cores are allocated to a helper rank in proportion to the average number of busy cores. By offloading tasks in reaction to a fine-grained load imbalance, LeWI helps to accelerate core usage. LeWI does not accelerate the speed of convergence of the global policy, as the solver responds to the total work on the apprank, but LeWI does reduce the peak near the beginning of the application, in Figure 11(a). Overall we again see that the combination of LeWI and DROM provides the best results.

8 CONCLUSIONS

Load balancing has been an important concern of high-performance computing for a long time. This paper introduces an automatic and transparent load balancing mechanism for MPI + OmpSs-2 programs, which relieves the application programmer from the burden of balancing loads across nodes. We keep the programming model as simple as possible, with only minor local changes being needed to existing MPI+OmpSs-2 programs. Our method uses OmpSs-2@Cluster to offload tasks to other nodes and it employs DLB as the underlying mechanism to allocate the resources (cores) on each node. We leverage the DROM module of DLB to reserve cores on other nodes and thereby address coarse-grained load imbalances, and use the LeWI module to react to fine-grained imbalances. Our system uses a sparse expander graph to minimise the amount of point-to-point communication and state. We show 46% reduction in time-to-solution for MicroPP solid mechanics on 32 nodes and

20% reduction beyond DLB for n-body on 16 nodes, when one node is slow. We perform a sensitivity analysis on imbalance, and obtain results within 10% of perfect load balancing for an imbalance of up to 2.0 on 8 nodes. All software is released open source, with the hope that our work can be expanded upon and adopted in practice.

9 ACKNOWLEDGEMENT

This research has received funding from the European Union's Horizon 2020/EuroHPC research and innovation programme under grant agreement No 955606 (DEEP-SEA) and 754337 (EuroEXA). It is supported by the Spanish State Research Agency - Ministry of Science and Innovation (contract PID2019-107255GB and Ramon y Cajal fellowship RYC2018-025628-I) and by the Generalitat de Catalunya (2017-SGR-1414).

REFERENCES

[1] 2022. Ingress | Kubernetes. https://kubernetes.io/docs/concepts/services-networking/ingress/.
[2] Bilge Acun, Phil Miller, and Laxmikant V Kale. 2016. Variation among processors under turbo boost in HPC systems. In *Proceedings of the 2016 International Conference on Supercomputing*. 1–12.
[3] Jimmy Aguilar Mena, Omar Shaaban, Vicenç Beltran, Paul Carpenter, Eduard Ayguade, and Jesus Labarta. 2022. OmpSs-2@Cluster: Distributed memory execution of nested OpenMP-style tasks. In *European Conference on Parallel Processing: Euro-Par 2022*.
[4] David Álvarez, Kevin Sala, Marcos Maroñas, Aleix Roca, and Vincenç Beltran. 2021. *Advanced Synchronization Techniques for Task-Based Runtime Systems*. Association for Computing Machinery, New York, NY, USA, 334–347.
[5] Martin Andersen, Joachim Dahl, and Lieven Vandenberghe. 2022. *CVXOPT: Python software for convex optimization*. https://cvxopt.org/
[6] Humayun Arafat, Ponnuswamy Sadayappan, James Dinan, Sriram Krishnamoorthy, and Theresa L Windus. 2012. Load balancing of dynamical nucleation theory Monte Carlo simulations through resource sharing barriers. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. IEEE, 285–295.
[7] Mahadevan Balasubramaniam, Kevin Barker, Ioana Banicescu, Nikos Chrisochoides, Jaderick P Pabico, and Ricolindo L Carino. 2004. A novel dynamic load balancing library for cluster computing. In *Third International Symposium on Parallel and Distributed Computing/Third International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*. 346–353.
[8] Barcelona Supercomputing Center. 2017. MareNostrum 4 (2017) System Architecture. https://www.bsc.es/marenostrum/marenostrum/technical-information.
[9] Barcelona Supercomputing Center. 2021. *Mercurium*. https://pm.bsc.es/mcxx
[10] Barcelona Supercomputing Center. 2021. *Nanos6*. https://github.com/bsc-pm/nanos6
[11] Barcelona Supercomputing Center. 2021. Nord III User's Guide. https://www.bsc.es/support/Nord3-ug.pdf.
[12] Barcelona Supercomputing Center. 2021. *OmpSs-2 Specification*. https://pm.bsc.es/ftp/ompss-2/doc/spec/
[13] Barcelona Supercomputing Center. 2022. OmpSs-2@Cluster releases. https://github.com/bsc-pm/ompss-2-cluster-releases
[14] Patrik Barkman. 2019. Parallel Barnes–Hut algorithm. https://github.com/barkm/n-body.
[15] Milind Bhandarkar, Laxmikant V Kalé, Eric de Sturler, and Jay Hoeflinger. 2001. Adaptive load balancing for MPI programs. In *International Conference on Computational Science*. Springer, 108–117.
[16] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. 1996. Cilk: An efficient multithreaded runtime system. *Journal of parallel and distributed computing* 37, 1 (1996), 55–69.
[17] Gerandy Brito, Ioana Dumitriu, and Kameron Decker Harris. 2018. Spectral gap in random bipartite biregular graphs and applications. *arXiv preprint arXiv:1804.07808* (2018). https://arxiv.org/pdf/1804.07808.pdf.
[18] Franck Cappello and Daniel Etiemble. 2000. MPI versus MPI+ OpenMP on the IBM SP for the NAS Benchmarks. In *SC'00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*. IEEE, 12–12.
[19] Marco D'Amico, Marta Garcia-Gasulla, Víctor López, Ana Jokanovic, Raül Sirvent, and Julita Corbalan. 2018. DROM: Enabling Efficient and Effortless Malleability for Resource Managers. In *Proceedings of the 47th International Conference on Parallel Processing Companion*. 1–10.
[20] Vida Dujmović, Anastasios Sidiropoulos, and David R Wood. 2015. Layouts of expander graphs. *arXiv preprint arXiv:1501.05020* (2015).
[21] Daniel E Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. 2016. Maglev: A fast and reliable software network load balancer. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. 523–535.
[22] Marta Garcia, Julita Corbalan, R.M Badia, and Jesus Labarta. 2012. A Dynamic Load Balancing Approach with SMPSuperscalar and MPI. *Facing the Multicore - Challenge II. Lecture Notes in Computer Science, vol 7174* (2012).
[23] Marta Garcia, Julita Corbalan, and Jesus Labarta. 2009. LeWI: A runtime balancing algorithm for nested parallelism. In *2009 International Conference on Parallel Processing*. IEEE, 526–533.
[24] Guido Giuntoli, Jimmy Aguilar, Mariano Vazquez, Sergio Oller, and Guillaume Houzeaux. 2019. A FE 2 multi-scale implementation for modeling composite materials on distributed architectures. *Coupled Systems Mechanics* 8, 2 (2019), 99.
[25] David S Henty. 2000. Performance of hybrid message-passing and shared-memory parallelism for discrete element modeling. In *SC'00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*. IEEE, 1369–1386.
[26] Shlomo Hoory, Nathan Linial, and Avi Wigderson. 2006. Expander graphs and their applications. *Bull. Amer. Math. Soc.* 43, 4 (2006), 439–561.
[27] Yuichi Inadomi, Tapasya Patki, Koji Inoue, Mutsumi Aoyagi, Barry Rountree, Martin Schulz, David Lowenthal, Yasutaka Wada, Keiichiro Fukazawa, Masatsugu Ueda, Masaaki Kondo, and Ikuo Miyoshi. 2015. Analyzing and mitigating the impact of manufacturing variability in power-constrained supercomputing. In *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–12.
[28] Haoqiang Jin and Rob F Van der Wijngaart. 2006. Performance characteristics of the multi-zone NAS parallel benchmarks. *J. Parallel and Distrib. Comput.* 66, 5 (2006), 674–685.
[29] Laxmikant V Kale and Sanjeev Krishnan. 1996. Charm++: Parallel programming with message-driven objects. *Parallel programming using C++* (1996), 175–213.
[30] George Karypis and Vipin Kumar. 1995. *METIS – Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0.* Technical Report.
[31] Jannis Klinkenberg, Philipp Samfass, Michael Bader, Christian Terboven, and Matthias S Müller. 2020. CHAMELEON: reactive load balancing for hybrid MPI+OpenMP task-parallel applications. *J. Parallel and Distrib. Comput.* 138 (2020), 55–64.
[32] Kubernetes. 2022. Kubernetes. https://kubernetes.io/.
[33] Hui Liu. 2008. Dynamic Load Balancing on Adaptive Unstructured Meshes. In *2008 10th IEEE International Conference on High Performance Computing and Communications*. 870–875. https://doi.org/10.1109/HPCC.2008.12
[34] Victor Lopez, Joel Criado, Raúl Peñacoba, Roger Ferrer, Xavier Teruel, and Marta Garcia-Gasulla. 2021. An OpenMP Free Agent Threads Implementation. In *OpenMP: Enabling Massive Node-Level Parallelism*, Simon McIntosh-Smith, Bronis R. de Supinski, and Jannis Klinkenberg (Eds.). Springer International Publishing, Cham, 211–225.
[35] Victor Lopez, Guillem Ramirez Miranda, and Marta Garcia-Gasulla. 2021. TALP: A Lightweight Tool to Unveil Parallel Efficiency of Large-scale Executions. In *Proceedings of the 2021 on Performance EngineeRing, Modelling, Analysis, and VisualizatiOn STrategy*. 3–10.
[36] Sina Meraji and Carl Tropper. 2011. Optimizing techniques for parallel digital logic simulation. *IEEE Transactions on Parallel and Distributed Systems* 23, 6 (2011), 1135–1146.
[37] NGINX. 2022. Advanced Load Balancer, Web Server, & Reverse Proxy - NGINX. https://www.nginx.com/.
[38] OpenMP Architecture Review Board. 2021. OpenMP Application Programming Interface, Version 5.2. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf Accessed: 2022-04-19.
[39] J. M. Perez, V. Beltran, J. Labarta, and E. Ayguadé. 2017. Improving the Integration of Task Nesting and Dependencies in OpenMP. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 809–818.
[40] Josep Pérez, Rosa M. Badia, and Jesús Labarta. 2008. A Dependency-Aware Task-Based Programming Environment for Multi-Core Architectures. *IEEE International Conference on Cluster Computing, ICCC*, 142–151.
[41] James Reinders. 2007. *Intel Threading Building Blocks* (first ed.). O'Reilly & Associates, Inc., USA.
[42] John K Salmon. 1991. *Parallel hierarchical N-body methods*. Ph. D. Dissertation. California Institute of Technology.
[43] Kirk Schloegel, George Karypis, and Vipin Kumar. 1997. Multilevel Diffusion Schemes for Repartitioning of Adaptive Meshes. *J. Parallel and Distrib. Comput.* 47, 2 (1997), 109–124. https://doi.org/10.1006/jpdc.1997.1410
[44] Zhi Shang. 2014. Impact of mesh partitioning methods in CFD for large scale parallel computing. *Computers & Fluids* 103 (2014), 1–5.
[45] Lorna Smith and Mark Bull. 2001. Development of mixed mode MPI/OpenMP applications. *Scientific Programming* 9, 2, 3 (2001), 83–98.
[46] Felix Wolf, Bernd Mohr, and Dieter an Mey. 2013. *Euro-Par 2013: Parallel Processing: 19th International Conference, Aachen, Germany, August 26-30, 2013, Proceedings*. Vol. 8097. Springer.