Exploiting OpenMP Malleability with Free Agent Threads and DLB

Joel Criado(✉), Victor Lopez, Joan Vinyals-Ylla-Catala, Guillem Ramirez-Miranda, Xavier Teruel, and Marta Garcia-Gasulla

Barcelona Supercomputing Center, Barcelona, Spain
{jcriado,vlopez,jvinyals,gramire1,xteruel,martag}@bsc.es
http://www.bsc.es

Abstract. This paper presents the evolution of the free agent threads for OpenMP to the new role-shifting threads model and their integration with the Dynamic Load Balance (DLB) library. We demonstrate how DLB efficiently manages the malleability exposed by the role-shifting threads to address load imbalance issues. We use two real-world scientific applications, one of them with a coupling case, to illustrate the potential of this approach. In addition, we also demonstrate that the new implementation is more usable than the former one, letting the runtime system automatically make decisions that were to be made by the programmer previously. All software is released open source.

Keywords: Dynamic load balancing · Free agents · OpenMP · Tasking · Malleability

1 Introduction

During the last years in the HPC community, both hardware and software are getting ready for the exascale era. On the one hand, hardware boosts the computational power of nodes by increasing the number of cores per node and using different accelerators as much as augmenting the number of nodes. On the other hand, software needs to evolve to use this massive computational power efficiently.

One of the main characteristics of software that has proven necessary to deal with the immense computational power is malleability. Malleability allows dealing with heterogeneous hardware, noise at all levels, load imbalance, communication inefficiencies, and dynamic workloads, among other issues.

Moreover, with the growing variety in hardware architectures, portability is another *must-have* characteristic for all the software components because it is not a sustainable approach to port every software to each newly designed platform.

In this challenging scenario, all the software stack layers must be malleable, flexible, and portable. We have already seen this direction in using workloads

H. Anzt et al. (Eds.): ISC High Performance 2022 Workshops, LNCS 13387, pp. 162–175, 2022.
https://doi.org/10.1007/978-3-031-23220-6_11

instead of monolithic applications [6], job schedulers managing adaptable applications [7,17], the development of malleable codes [8], and parallel programming models offering dynamic malleability [10].

In this work, we focus on OpenMP; the OpenMP parallel programming model has embraced malleability since its appearance, instead of much more rigid parallel models such as MPI, which only recently has started to offer this feature. However, even the original malleability of the OpenMP model has proven not to be enough. For this reason, we extend our previous free agent threads proposal that expands the malleability of the programming model outside the parallel construct. This new feature allows the Dynamic Load Balancing (DLB) library to exploit the malleability of hybrid MPI+OpenMP applications further, achieving better efficiencies.

The main contributions of this paper are the following: a new implementation of the free agent threads; this new implementation aims to add a lower overhead, be more usable, and offer an extensible framework; the integration of the free agent threads with the DLB library, and the demonstration using two real HPC applications.

The remaining of this document is organized as follows. Section 2 reviews the related work that can be found in the literature. In Sect. 3, we explain the details of the proposed implementation and how it has been integrated with the DLB library. In Sect. 4, we present the performance evaluation, and finally, in Sect. 5, we summarize the paper's findings in the conclusions.

2 Related Work

This work relies on two main principles: a task-based programming model, where the parallel decomposition leverages the creation of unstructured work units (called tasks), and runtime malleability, in terms of resource allocation, of the associated task-based programming runtimes.

Among the set of task-based programming models currently used in HPC, we can find:

The *OmpSs* programming model [1,9] expresses parallelism using compiler directives. These directives are transformed at compile time into runtime services with well-defined semantics. Among others, programmers may create new tasks, wait for their execution, establish the proper order of task executions, atomic/critical memory updates, etc. When ignoring directives, sequential behavior is expected.

The *Intel Threading Building Blocks* (TBB) [13] is a C++ template library that allows program parallelization through tasks. Programmers may use high-level or low-level interfaces to spawn tasks; with such information, the runtime creates a Task Dependency Graph and executes tasks in parallel when possible.

The *Intel Cilk++ SDK* [12] is a C++ language extension that includes a runtime library within the family of Intel compilers. It allows expressing the parallelism through a few language keywords (similar to compiler directives), which ease iteration space decomposition, stand-alone tasks creations, and the synchronization among these work units. The current version of this approach is the *OpenCilk project* [15], under the *Massachusetts Institute of Technology* (MIT) supervision.

The *OpenMP* programming model [16] includes a task-based approach (in addition to its traditional work-sharing model). The tasking sub-model allows creating new tasks, waiting for their execution, and adequately ordering tasks using data dependences. This tasking approach is similar to the aforementioned OmpSs programming model, although the execution model is still bound to the creation of parallel regions.

Resource usage malleability is the other pillar on which our implementation relies. Changing the number of assigned processing elements at runtime requires a parallel decomposition that does not depend on them. This requirement removes from the equation the traditional OpenMP work-sharing constructs. Once we start a parallel region, the number of threads participating in it must remain constant until the end of the construct. Otherwise, some fundamental definitions will be broken (e.g., the barrier directive or a static distribution of iterations among threads).

Task-based approaches, instead, will ease resource malleability. As the created tasks are not bound to a particular thread before starting their execution, the number of threads (and the number of cores, consequently) may change at runtime. Several existing implementations leverage such malleability:

The *OmpSs* programming model implements a Thread Manager module, which provides support to the number of threads and their bindings to the underlying CPUs. The OmpSs Thread Manager may also interact with the *Dynamic Load Balance* library [7,11]. This library gathers information about the system occupancy beyond the process level, having an overall picture of the whole node status. With this information, it can decide and change which should be instant resource ownership for each of the processes along with the program execution.

Some OpenMP implementations also include the idea to use additional threads, not directly included in the parallel region, to help execute tasks. The *Hidden Helper Threads* feature [18] implemented in the LLVM compiler presents a common use case in which the target construct may leverage the presence of these other threads to relieve the critical path. The main differences with our current implementation are: 1) the *Hidden Helper Threads* approach does not allow to change the number of threads; and 2) it is restricted to the use of target tasks, while our proposal intends to be more generic[1].

Our initial implementation of *free agent threads* [14] combines two previous approaches. On one hand, we implement the mechanism on top of an OpenMP runtime fully integrated as a standard programming model. On the other hand, we implement it in such a generic way that all the instantiated tasks could leverage the presence of these additional threads to be executed. The DLB component is responsible for increasing or reducing the number of threads participating in

[1] This is also the reason we are not comparing against this proposal; the study's use cases do not take any benefit from the *Hidden Helper Threads* implementation.

the process. This previous work demonstrated how free agent threads could address load imbalances problems inherent in some HPC applications.

The current version, presented in this work, generalizes and simplifies the implementation by allowing an existing thread to change its role during the execution. Taking advantage of existing threads that no longer participate in the parallel region reduces the cost of creating and managing such threads. In addition, the implementation is more consistent with the definitions of the model itself regarding the *limit of threads*. We also prepare the runtime to host other types of roles in the future. We believe it is interesting for OpenMP users and developers to increase the model's extensibility further. For instance, the standard could consider dedicating specific threads to execute communication tasks (i.e., the thread role will be *communicator*). Finally, it also simplifies the way programmers interact with the execution of the resulting programs by pushing the rationale of specific decision-making configurations as part of the OpenMP runtime (i.e., automatize parameters). The evaluation section will show that the runtime's automated decisions always improve the best configuration used in the previous implementation.

3 Implementation

Our previous free agent thread implementation used a mechanism of two pools of threads, one containing the *initial* plus the *worker threads* and the other pool containing the set of free agent threads. The idea behind was to have a representative thread per processor and enable either the *worker thread* or the free agent thread depending on whether the thread on that processor was needed for a *parallel region*.

After evaluating our first approach, we observed two undesirable situations. Firstly, a *worker thread* and a free agent thread both bound to the same processor could be active simultaneously. When the first was needed for a new *parallel region* while the second was still executing an *explicit task*, i.e., a task generated by a `task` construct, thus provoking a short time-lapse of processor oversubscription. Secondly, should the OpenMP model implement a new type of thread, its implementation may also be done using a third pool of threads overcomplicating the implementation. Our new implementation solves both problems by using the same thread running with different roles and adding extensibility to the model.

The new free agent thread and role-shifting implementation presented in this paper are based on the LLVM OpenMP runtime version 14.0.0.

3.1 The LLVM OpenMP Runtime

The LLVM OpenMP runtime implements *parallel regions* as follows. When a thread encounters a `parallel` construct, the thread creates the structure for the team of threads and assigns as many threads as needed to the team. Threads will be created the first time that the runtime needs such threads. Upon completion of the *parallel region*, threads are suspended and moved to a thread pool structure until another *parallel region* is encountered. If subsequent *parallel regions* do not need more threads than any other previous region, existing threads will be reused.

The LLVM OpenMP runtime implements the thread fork-join model using two different kinds of barriers. The first kind of barrier is called *fork-barrier*, and this is where all idle threads are waiting until they are needed for some team. When an idle thread is assigned to a team, the thread is *released* and executes an *implicit task*, which is the task assigned to each team member that includes all the *parallel region* code. One particularity of this barrier is that a thread is released as soon as it is ready; whether the other threads participating in the same *parallel region* have arrived at the barrier is irrelevant.

The second kind of barrier is called *join-barrier*, and it is used to join all threads at the end of a *parallel region*. It is a more traditional barrier where all threads must reach the barrier before the rest may proceed. After that, all threads again enter the *fork-barrier*.

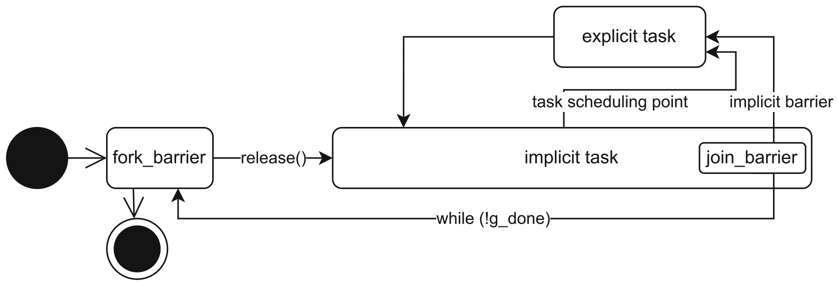


Fig. 1. OpenMP worker thread flowchart.

The described flowchart of a worker thread is shown in Fig. 1. A thread may reach task scheduling points while executing its *implicit task*, typically when encountering `taskgroup`, `taskwait`, `barrier` constructs, etc. The implementation may perform a task switch at this point, beginning or resuming the execution of an *explicit task* bound to the same team. Once it reaches the implicit barrier, a worker thread may also execute *explicit tasks*.

3.2 The Role-Shifting Threads

The role-shifting threads are an evolution of the current OpenMP threads. We can differentiate two types of OpenMP threads: the *initial* and *worker* threads. When a non-nested *parallel region* ends, all the *worker threads* become idle until the *initial thread* encounters another *parallel region*. The idea behind role-shifting threads is to use the already existing idle threads to perform different jobs based on their available roles.

Under the new model, all threads, including the *initial thread*, can have from 0 to n potential roles, but only one of them can be active at a time. The *initial* *thread* may not, and probably must not, use any role, but we do not enforce the restriction for simplicity in the specification. The *worker* role is implicit in all the threads since they may be able to participate in a *parallel region* at any time.

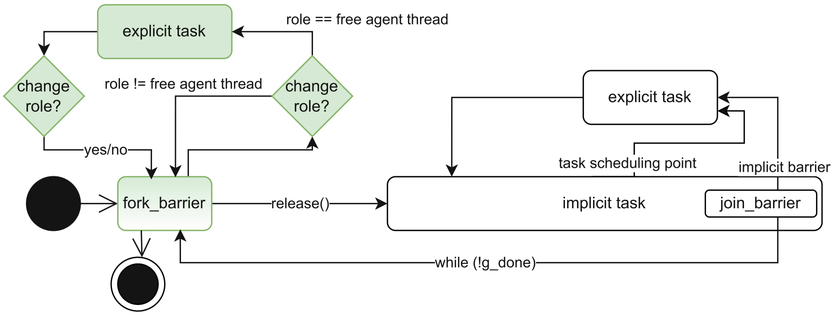


Fig. 2. OpenMP role-shifting thread flowchart.

A thread can shift its active role at different points. At the start of a *parallel region*, all the required threads must abandon their current role and execute their assigned *implicit task*; after finishing it, they may shift to one of their potential roles. Regarding the free agent role, these threads may change their role before and after executing an *explicit task*. These role-shifting points are depicted in Fig. 2. In the future, other roles may use the same shifting points and introduce new ones if required.

New API Routines. We have extended the OpenMP API to interact with the role-shifting threads model and introduced the concept of *global thread id* in the runtime to interact with the API. This thread id is a unique identifier assigned at thread creation and lasts for the entire execution. The global thread id must not be confused with the current OpenMP thread number, which identifies each thread participating within a parallel region.

- `int omp_get_thread_id(void)`: Obtains the global thread id of that thread.
- `int omp_get_thread_roles(int tid, omp_role_t *roles)`: Returns the number of potential roles for thread with *global id* `tid` and sets `roles` to the potential roles of the thread.
- `void omp_set_thread_roles(int tid, omp_role_t roles)`: Sets the potential roles of the thread with *global id* `tid` to `roles`. It will remove all previous potential roles from the thread. If `tid` is higher than the current number of threads, the runtime will create a new thread with the appropriate `roles`.

Environment Variables. We propose a unique environment variable to unify all the role-shifting threads model:

`OMP_ROLES`: Indicates the initial number of threads with the desired potential roles. Usage examples:

`OMP_ROLES="{role1},{role2},{role1,role3}"`. Three different threads, one with `role1`, one with `role2`, and another with `role1` and `role3`.

`OMP_ROLES="{role1},{role2,role4}*3"`. Four different threads, one with `role1` and three with `role2` and `role4`.

New OMPT Callback Signature. We propose a new OMPT callback to identify when a thread shifts its active role.

`void ompt_callback_thread_role_shift(ompt_data_t *thread_data, ompt_role_t prior_role, ompt_role_t next_role)`: Each thread emits the callback each time it changes the active role: `prior_role` indicates the previous active role, and `next_role` indicates the new active role of that thread.

3.3 Integration with DLB

We have integrated the free agent threads role from the role-shifting threads implementation with the *Lend When Idle* (LeWI) module of DLB. LeWI aims at optimizing the performance of hybrid applications (MPI+OpenMP) by improving their load balance. Figure 3 shows how LeWI operates for an unbalanced application. When an MPI process executes a blocking call, it lends all the CPUs it has at that moment, and other processes may acquire them for their use. After exiting the MPI call, the process reclaims all the CPUs it owns, and it can continue its execution transparently.

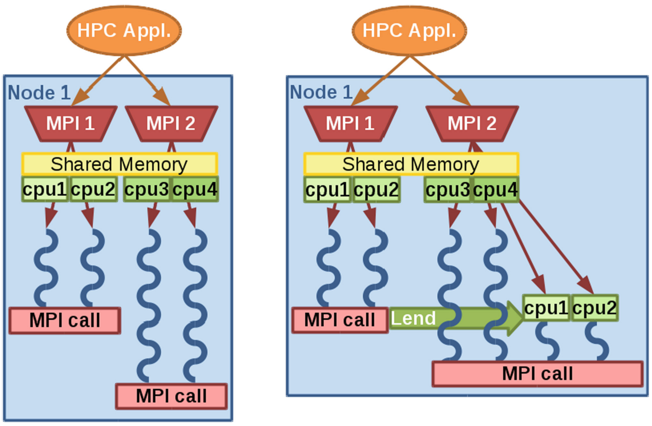


Fig. 3. Example of DLB and LeWI balancing algorithm. On the left is an unbalanced hybrid application. On the right, the application is balanced using LeWI.

Regarding the OpenMP integration, we capture specific OMPT callbacks and perform different actions with DLB:

- **Parallel_begin**: We must register when executing a parallel region and the number of threads associated with it. Those threads are required for the entire execution of the region and cannot shift their role at any moment.
- **Parallel_end**: The parallel region ends, and DLB may use the threads from the former parallel region for load balancing purposes.
- **Task_schedule**: When starting the execution of a task, DLB tries to acquire a CPU from any other process if there are more pending tasks. When a free agent thread ends the execution of a task, it returns the CPU if it has been reclaimed, or it lends the CPU if there are no more pending tasks, or it proceeds silently.
- **Thread_begin**: We extract the global thread id for each thread and set the affinity of the free agents to their correspondent CPU.
- **Thread_role_shift**: When a thread changes from worker to free agent, we deactivate it if the CPU has been reclaimed or there are no more pending tasks.

When an MPI process receives a CPU for the first time, it creates a new thread with the role of free agent, and it assigns that CPU for the rest of the execution to that thread. When it receives that same CPU in the future, instead of creating a new thread, it will change the role of that thread with the API. We also tried a different strategy where we rebind inactive threads to new CPUs when possible, but it had more overhead and was more sensitive to system noise (e.g., CPU preemptions by the OS), so it was discarded.

4 Evaluation

In this section, we present the performance evaluation of the proposed implementation. In this evaluation, we will compare three versions of each application:

- **Original**: The original application executed as in a production run.
- **Double-pool**: The original application using the DLB load balancing library with the LLVM free agent threads implementation based on the double pool of threads. For this implementation, the user must provide the maximum number of free agent threads used per MPI process. We will consider this variable in the evaluation as *Num. free agent threads*.
- **Role-shifting**: The original application using the DLB load balancing library with the LLVM free agent threads implementation based on the role-shifting. This version does not need additional parameters, and the runtime automatically decides the number of free agent threads.

4.1 HPC Environment

All the experiments presented in this work have been obtained using MareNostrum4. MareNostrum4 is a supercomputer based on Intel Xeon Platinum processors; each node comprises two sockets (Intel Xeon Platinum 8160 CPU) with

24 cores each at 2.10 GHz for a total of 48 cores per node and 96 GB of main memory. Its nodes are connected using a 100 Gbit/s Intel Omni-Path network. It houses 3456 nodes accounting for a total of 165888 cores.

The runtime, DLB, and all the applications have been compiled using the Intel 2017.4 suite, and the MPI library used to run is Intel MPI 2017.4 version. We use DLB version 3.0 [2] and the extended LLVM OpenMP runtime library [3] to support the free agent threads in all cases.

For the evaluation, we test 2 HPC applications used in production runs, a parallel remesher, ParMmg, and a simulation code for high-performance computational mechanics, Alya.

4.2 ParMmg

ParMmg [5] is a parallel remesher developed by INRIA, based on top of the sequential Mmg remesher. Mesh adaptation is widely used in computational solid mechanics (CSM) and computational fluid dynamics (CFD) domains to improve the quality of the solution. The application is written in C and parallelized with MPI. The input set used in the study is prepared to do a weak scaling using power of two MPI ranks from 2 to 256 processes.

We added an OpenMP taskification on the main loop iterations to implement a second level of parallelism in that region that allows us to exploit the load balancing capabilities of DLB.

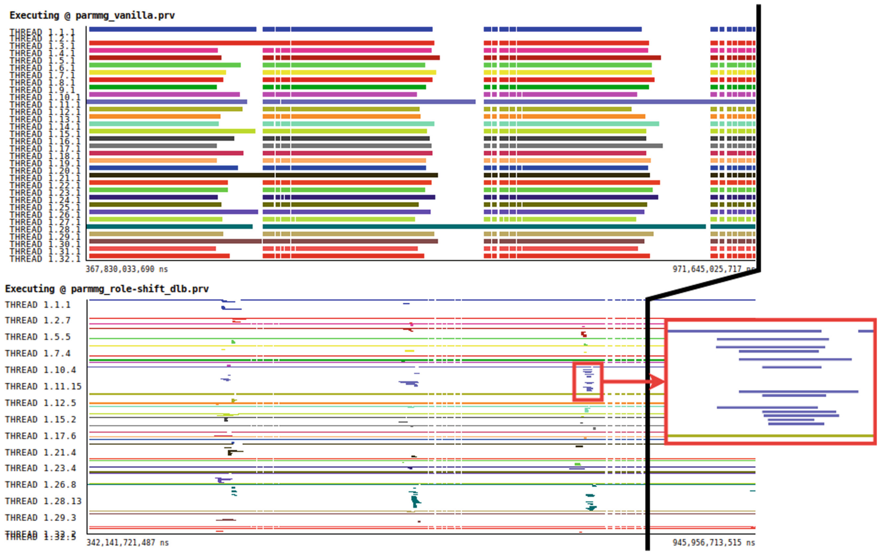


Fig. 4. Top: ParMmg Paraver trace execution of 3 iterations using 32 MPI ranks. Bottom: Same execution using role-shifting threads and DLB. Each color represents different MPI ranks, and both traces are at the same duration scale.

ParMmg presents an irregular load imbalance among the different iterations, as seen in the top trace of Fig. 4. In this figure, we show a Paraver trace of an execution of ParMmg using 32 MPI ranks; each horizontal line corresponds to one MPI process, and in the x axis is represented the time. White means that the MPI process is not doing useful computation, i.e., it is inside an MPI call, and any other color means computing. In this trace, we can identify three steps and observe that the load distribution changes from one iteration to another. For these two reasons, ParMmg can benefit from the load balancing capabilities of DLB because the load imbalance can not be predicted and changes dynamically during the execution.

We can see the same execution using DLB and role-shifting threads in the bottom trace of the same figure. We can observe that each MPI process can now have more than one OpenMP thread; these are the different lines below an MPI process. We can also observe how the additional threads are used to speed up the execution of the most loaded MPI ranks.

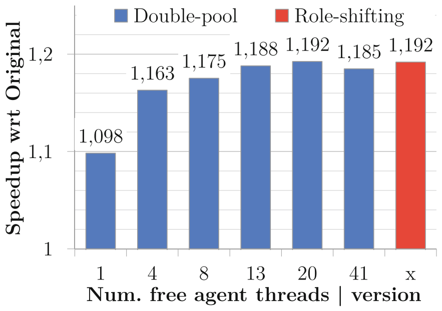


Fig. 5. ParMmg speedup with DLB and different free agents implementations.

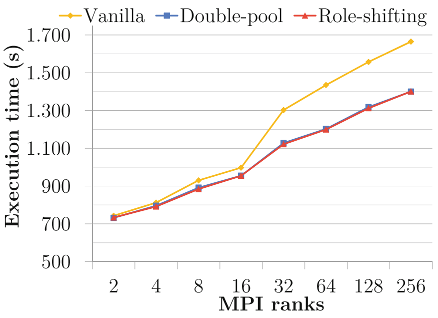


Fig. 6. ParMmg execution time.

In Fig. 5, we show the speedup obtained with the different versions with respect to the original execution of ParMmg without DLB add free agent threads. In the x-axis, we show the different versions of the free agent threads implementations and the different number of free agent threads used for the *Double-pool* implementation. There are two important outcomes from this plot. On the one hand, the role-shifting implementation obtains the same performance as the best configuration of the double-pool implementation. On the other hand, the performance of the double-pool implementation depends highly on the number of free agent threads that the user specifies.

Figure 6 displays the execution time of ParMmg using different number of MPI ranks (and cores) on the x-axis. The different versions are represented by different lines. The number of free agent threads allowed per MPI rank is set to the best configuration measured in the previous experiment for the Double-pool version. As ParMmg is a weak scaling application, the ideal execution would be a flat line. We can see that for all the cases, the execution using DLB and free agent threads improves the performance of the vanilla ParMmg code. For all the executions, the role-shifting implementation performs as well as the best configuration of the double-pool implementation.

4.3 Alya

Alya [19] is a high-performance computational mechanics code that can solve multiple physics, standalone or coupled. Most of the problems it can address come from the engineering realm. Among the different physics solved by Alya, we can mention incompressible and compressible flows, non-linear solid mechanics, chemistry, particle transport, heat transfer, turbulence modeling, electrical propagation, etc. Alya was specially designed for massively parallel supercomputers and is part of the *Unified European Application Benchmark Suite* (UEABS), a set of 13 highly scalable, relevant, and publicly available codes. Alya is written in Fortran and parallelized at different levels, including MPI, SIMD, OpenMP, and GPUs. This paper will use the MPI+OpenMP version, and the OpenMP parallelization will be used only for load balancing. The executions will be launched as an MPI-only execution (one core per MPI rank, 1 OpenMP thread per process). This is because the OpenMP parallelization of Alya is not exhaustive in all the code and is not used in production runs.

The use case executed in this paper is a production combustion problem, coupling the fluid solution on the one hand with the chemical reaction on the other [4,20]. In Fig. 7, we can see a trace of the execution of Alya with 768 MPI ranks. The first 96 MPI ranks are solving the fluid, and the remaining 672 the chemical reaction. In this trace, the grey color represents useful computation, the other colors represent the MPI calls executed by the program. We can identify two time steps and the two coupled problems in the trace. We can observe that the computing region before the *MPI_Barrier* (red) is the more time-consuming one, and at the same time, it presents a significant load imbalance.

We evaluate three different executions of Alya, the original code, using DLB and the double-pool implementation of the free agent threads, and using DLB with the role-shifting version. In Fig. 8a, we can see the speedup obtained when using DLB and free agents with the different versions with respect to the original execution of Alya, using 768 MPI ranks in 768 cores for all cases. In the x-axis, we show the different number of free agent threads enabled for the double-pool implementation. We can see that the role-shifting version achieves a better speedup than the best configuration of the double-pool implementation. We can also observe that the performance of the double-pool implementation depends on the number of free agent threads enabled by the user.

In Figs. 8b and 8c, we can see the same study running Alya with 1152 and 1536 MPI ranks. In both plots, we can see that the role-shifting implementation outperforms all the configurations of the double-pool one. Alya's tasks have finer grain than ParMmg (a few milliseconds per task), and the program benefits from the reduction in overhead in the runtime and DLB integration. It is also interesting to notice that the best configuration of the double-pool implementation is not consistent between the executions with the different number of MPI ranks.

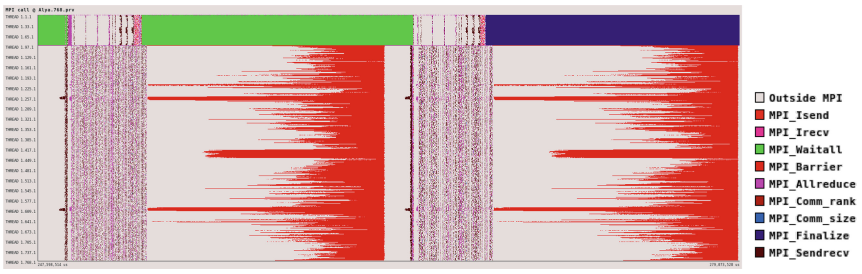
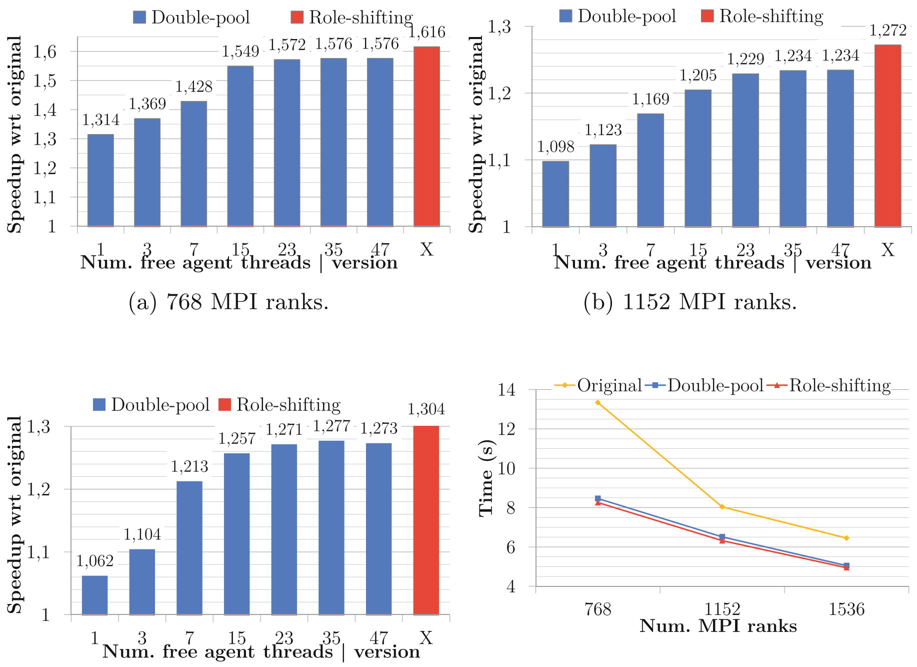


Fig. 7. Alya Paraver trace execution of 2 iterations coupling 96 MPI ranks for the fluid simulation and 672 MPI Ranks for the chemical simulation.



(a) 768 MPI ranks.

(b) 1152 MPI ranks.

Fig. 8. Speedups obtained running Alya with different implementations of free agents and DLB, and execution time with different number of MPI ranks

Figure 8d shows the execution time achieved by the different versions when running Alya varying the number of MPI ranks. We show that the use of DLB and free agent threads improves the performance of the original Alya code in all the cases. In this plot, we use the best configuration achieved in the previous experiments for the double-pool implementation. However, the best option is to use the role-shifting version of the free agent treads implementation.

5 Conclusions

This paper presents a new extension of the OpenMP programming model, allowing their threads to have different roles. The previous free agent threads have been merged into this new implementation as a role. With this approach, the model has a unique pool of threads, in contrast to the previous one, employing fewer resources. Moreover, the role-shifting approach is an opportunity to include more roles in the model, which may lead to more improvements in terms of malleability and flexibility.

Previously, the user had to select the desired number of free agent threads at the start of the execution, but the role-shifting allows for changes at runtime. This change makes the model more flexible for the users and tools using the OMPT interface from OpenMP. This fact is reflected in the evaluation, where the role-shifting model delivered the same or better performance than the double-pool model without any tunning required.

Furthermore, we demonstrate how the free agent threads proposal increases the malleability of the OpenMP standard, thus, allowing tools like DLB to exploit it to achieve better efficiencies. To this end, the role-shifting model has been integrated with DLB.

In Sect. 4, we have demonstrated how DLB improves the performance of hybrid applications, exploiting the malleability exposed by OpenMP tasks by enabling and disabling threads with the free agent role. The results showed speedups from $1.2x$ to $1.62x$ in two real-world scientific applications, mending their load imbalances.

Overall we show the relevance of malleability at the different levels of the software stack, such as applications and different programming models to achieve performance. Also, the need to isolate the user from these low-level decisions and that the different runtime systems must coordinate to use the computational resources efficiently.

Acknowledgements. This work has received funding from the DEEP Projects, at the European Commission's FP7, H2020, and EuroHPC Programmes, under Grant Agreements 287530, 610476, 754304, and 955606. In the latter (DEEP-SEA), national contributions from the involved state members match the EuroHPC funding. It also has the support of the Spanish Ministry of Science and Innovation (Computacion de Altas Prestaciones VIII: PID2019-107255GB).

References

1. Barcelona Supercomputing Center: OmpSs Specification. https://pm.bsc.es/ompss, Accessed Mar 2022
2. DLB repository. https://github.com/bsc-pm/dlb/commit/7e91a80a, Accessed Mar 2022
3. LLVM repository. https://github.com/bsc-pm/llvm/commit/3c5352db, Accessed Mar 2022

4. Cavaliere, D.E., Kariuki, J., Mastorakos, E.: A comparison of the blow-off behaviour of swirl-stabilized premixed, non-premixed and spray flames. Flow Turbulence Combust. **91**(2), 347–372 (2013). https://doi.org/10.1007/s10494-013-9470-z
5. Cirrottola, L., Froehly, A.: Parallel unstructured mesh adaptation using iterative remeshing and repartitioning. Research Report RR-9307, INRIA Bordeaux, équipe CARDAMOM (2019). https://hal.inria.fr/hal-02386837
6. Conejero, J., Corella, S., Badia, R.M., Labarta, J.: Task-based programming in COMPSs to converge from HPC to big data. Int. J. High Perf. Comput. Appl. **32**(1), 45–60 (2018)
7. D'Amico, M., Garcia-Gasulla, M., López, V., Jokanovic, A., Sirvent, R., Corbalan, J.: DROM: enabling efficient and effortless malleability for resource managers. In: Proceedings of the 47th International Conference on Parallel Processing Companion, p. 41. ACM (2018)
8. Desell, T., Maghraoui, K.E., Varela, C.A.: Malleable applications for scalable high performance computing. Clust. Comput. **10**(3), 323–337 (2007)
9. Duran, A., et al.: OmpSs: a proposal for programming heterogeneous multi-core architectures. Parallel Process. Lett. **21**, 173–193 (2011)
10. El Maghraoui, K., Desell, T.J., Szymanski, B.K., Varela, C.A.: Dynamic malleability in iterative mpi applications. In: Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2007), pp. 591–598. IEEE (2007)
11. Garcia, M., Labarta, J., Corbalan, J.: Hints to improve automatic load balancing with LeWI for hybrid applications. J. Parallel Distrib. Comput. **74**(9), 2781–2794 (2014)
12. Intel Corporation: Intel Cilk++ SDK Programmer's Guide (2009). https://www.clear.rice.edu/comp422/resources/Intel_Cilk++_Programmers_Guide.pdf
13. Intel Corporation: Intel Threading Building Blocks (2011). https://www.inf.ed.ac.uk/teaching/courses/ppls/TBBtutorial.pdf
14. Lopez, V., Criado, J., Peñacoba, R., Ferrer, R., Teruel, X., Garcia-Gasulla, M.: An OpenMP free agent threads implementation. In: McIntosh-Smith, S., de Supinski, B.R., Klinkenberg, J. (eds.) IWOMP 2021. LNCS, vol. 12870, pp. 211–225. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-85262-7_15
15. Massachusetts Institute of Technology: OpenCilk Language Extension Specification Version 1.0 (2021). https://cilk.mit.edu/docs/OpenCilkLanguageExtensionSpecification.htm
16. OpenMP Architecture Review Board: OpenMP Application Programming Interface, Version 5.2 (2021). https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf, Accessed 14 Mar 2022
17. Prabhakaran, S., Neumann, M., Rinke, S., Wolf, F., Gupta, A., Kale, L.V.: A batch system with efficient adaptive scheduling for malleable and evolving applications. In: 2015 IEEE International Parallel and Distributed Processing Symposium, pp. 429–438 (2015)
18. Tian, S., Doerfert, J., Chapman, B.: Concurrent execution of deferred openmp target tasks with hidden helper threads. In: Chapman, B., Moreira, J. (eds.) Languages and Compilers for Parallel Computing, pp. 41–56. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-95953-1_4
19. Vázquez, M., Houzeaux, G., Koric, S., et al.: Alya: multiphysics engineering simulation toward exascale. J. Comput. Sci. **14**, 15–27 (2016)
20. Zhang, H., Garmory, A., Cavaliere, D.E., Mastorakos, E.: Large eddy simulation/-conditional moment closure modeling of swirl-stabilized non-premixed flames with local extinction. Proc. Comb. Inst. **35**(2), 1167–1174 (2015)