An OpenMP Free Agent Threads Implementation

Victor Lopez(✉), Joel Criado, Raúl Peñacoba, Roger Ferrer, Xavier Teruel, and Marta Garcia-Gasulla

Barcelona Supercomputing Center (BSC), Barcelona, Spain
{vlopez,jcriado,rpenacob,rferrer,xteruel,martag}@bsc.es

Abstract. In this paper, we introduce a design and implementation of the free agent threads for OpenMP. These threads increase the malleability of the OpenMP programming model, offering resource managers and runtime systems flexibility to manage threads and resources efficiently. We demonstrate how free agent threads can address load imbalances problems at the OpenMP level and at an MPI level or higher. We use two mini-apps extracted from two real HPC applications and representative of real-world codes to demonstrate this. We conclude that more malleability in thread management is necessary, and free agents can be regarded as a practical starting point to increase malleability in thread management.

Keywords: OpenMP · Tasks · Free agent · Malleability · Dynamic load balancing

1 Introduction

In the current race for exascale, the new HPC architectures are going in two main directions to achieve their goal. On the one hand, adding accelerators to the compute nodes, and on the other one, increasing the number of cores per socket. These trends challenge parallel programming models to provide support, transparency, and performance in these new architectures. When increasing the number of cores per socket, the challenge is using a high number of cores efficiently. To address this challenge, undoubtedly, all heads are turning to look at OpenMP as the most widely used shared memory programming model.

Noise, load imbalance, complex code, or lack of parallelism, among others, are some of the pitfalls that can jeopardize efficiency when using architectures with a high number of cores per socket. To address these issues is no longer enough to fight them; we need to adapt. Parallel programming models need to offer flexibility (i.e., the execution model is not predetermined, several external factors need to be considered, such as the current state of the system) and malleability (i.e., the ability to increase or decrease the hardware resources used at any time) to adjust the execution at runtime and make it transparent and

S. McIntosh-Smith et al. (Eds.): IWOMP 2021, LNCS 12870, pp. 211–225, 2021.
https://doi.org/10.1007/978-3-030-85262-7_15

straightforward for the user or developer of the code. Moreover, the different layers of the software stack, job schedulers, resource managers, distributed memory programming models, or shared-memory programming models must cooperate and coordinate.

This paper presents a design and implementation of the free agent threads in the LLVM OpenMP runtime. The free agent threads increase the malleability and flexibility of OpenMP, allowing extra threads to execute tasks in idle computational resources, and at the same time, offering a tool that will help coordinate the workload between different resource managers or runtime systems. Since tasks were introduced in OpenMP, there has been an interest in having free agent or task-only threads in the model [18]. Now it is one of their objectives in their roadmap for OpenMP 6.0 [5].

The remainder of this paper is organized as follows. In Sect. 2, we review the current state of the art of different task-based programming models and their malleability and other approaches that try to exploit malleability to improve efficiency. Later, in Sect. 3, we discuss the design decisions regarding the definition and context of the free agent threads within the OpenMP standard. In the following section, we explain some relevant implementation details of our proposal. In Sect. 5, we present the evaluation of the proposal. For this evaluation, we consider two use cases, in the first one a load imbalance problem at the OpenMP level among different parallel regions. The second use case considers a load imbalance between MPI processes that can be solved using a Dynamic Load Balancing Library and the free agent threads implementation. Finally, in Sect. 6, we will summarize the conclusions gathered from this work in the last section.

2 Related Work

Several programming models are implementing a pure task-based approach versus a thread-based one. A pure task-based programming model relies on creating work units that could be executed by any processing element available on the system and does not usually tie the resulting parallel decomposition to any hardware resource.

The OmpSs [4,8] programming model expresses the application parallelism through task-generating constructs. A task construct is a compiler directive or a source code comment that the compiler can interpret with well-defined semantics. Tasks are also annotated with clauses to specify certain behaviors (e.g., the data associated with the task; and if this data is read, written, or updated). In addition to these task-generating constructs, the programmer has another mechanism to handle the synchronization among tasks and guarantee correctness accessing shared memory.

The Intel Threading Building Blocks [12] component, currently known as oneTBB, is a C++ template library that allows parallelizing an application breaking it down into tasks. The programmer may use any of the TBB prepackaged high-level interfaces (i.e., Generic Parallel Algorithms, Parallel STL,

or Flow Graph interfaces) or directly using its low-level interface to create tasks. A TBB task is an entity that defines a small computation unit and its associated data. With that information, the runtime can create a task dependency graph and execute tasks in parallel.

Intel Cilk++ SDK [11] is a language compiler add-on and a runtime library included in the Intel compiler family. It allows expressing parallelism using only three keywords: `cilk_spawn` (to create a task), `cilk_for` (to parallelize loops), and `cilk_sync` to wait for completion. In addition to these three fundamental keywords, the Cilk++ SDK offers other services to handle most parallel programming challenges (locks, reducers, etc.). The current incarnation of the Cilk language families is the OpenCilk [13] project, maintained by the Massachusetts Institute of Technology. The project also includes an open-source implementation of the Cilk concurrency platform, compatible with the Cilk Plus language extension to C and C++.

The OpenMP [16] programming model, in its version 3.0 [14], also included a task-based approach. With the `task` construct, programmers were able to annotate tasks. Since its version 4.0 [15], they could also annotate them with the `depend` clause, enabling the runtime to compute the task dependency graph and properly synchronize the task execution order. The main problem of this tasking extension is that the execution model is still bound to the creation of parallel regions, perpetuating the rigid fork-join pattern of this model.

Task-based parallel approaches ease the malleability of parallel executions. And malleability allows adapting the use of underlying resources, and, in some instances, it also allows to adapt it dynamically. This is the case of the aforementioned OmpSs programming model. Its implementation includes a module, the Thread Manager, which determines the number of threads and their binding to the underlying CPUs. Furthermore, this module may agree with an external component (e.g., a resource manager) which may decide to extend or reduce the number of CPUs used at any given time. The resource manager may collect information from different processes running in the node, which improves the quality of this decision.

The Dynamic Load Balance library (DLB [9]) is one of these resource managers. This software implements several policies to decide the usage and/or the ownership (DROM [7]) of CPUs by a set of parallel processes linked to it. Then, the library can shrink (in the phases it has not declared enough parallelism) or expand (when the application reaches stages with a significant number of concurrent tasks) the number of threads for a given process. The ideal situation occurs when a process may yield its CPUs to another one that requires them.

DLB can easily interoperate with OmpSs due to the remarkable malleability of this programming model [10]. OmpSs can increase or reduce the number of threads participating in the execution of a given program almost at every single point. This is not the case with the OpenMP programming model. Once the application starts executing a parallel region with a certain number of threads, it is impossible to change the number of participants; it will break the semantics of work-sharing. But the execution of tasks does not require a constant number

of worker threads, neither that just threads from the current parallel region are the only candidates to execute these tasks.

Some extensions of OpenMP attempt to include the idea of using additional threads, not participating in the current parallel region, to help in the execution of the instantiated tasks. Using the hidden helper threads implementation [19], the authors propose to leverage not currently active worker threads to participate in the offload of target regions to the device. This is a common use case: offloading kernels to a GPU while executing the sequential part of the OpenMP program and losing potential performance due to unused CPUs. The main difference of this extension compared to our proposal is that hidden helper threads do not allow dynamically changing the number of threads, where the OpenMP standard does not impose any restriction. In addition, our proposal aims to be more generic, and it allows executing any task rather than restricting these threads to execute target tasks. This is also the main reason we have not used this implementation as a comparison counterpart. We are not targeting devices other than host, and, in addition, we base our fundamental source of improvement on dynamically changing the number of threads (which is not possible with this implementation of hidden helper threads).

3 Proposal

We present our design of free agent threads as an addition to the OpenMP specification to increase the malleability of the programming model. Our proposal is driven towards making free agent threads as much flexible as possible. They should be treated as helper threads that can be enabled or disabled, and the OpenMP runtime will use them whenever possible.

The OpenMP specification distinguishes between *implicit task*, which is the task implicitly assigned to any thread participating in a parallel region, and *explicit task*, which is the task generated by a `task` construct. In our proposal, free agent threads are *OpenMP threads* that will not be considered when encountering a parallel region. Their only purpose is to execute *explicit tasks*.

Free agent threads will neither participate in any team synchronization point, such as `barrier` constructs or *implicit barriers*. They will, however, be part of the initial thread contention group and will participate in other synchronization constructs such as `critical` or `atomic`.

A task executed in a free agent thread may contain other parallelism-related constructs, although we have not explored all the possibilities, and further investigation would be needed. The `parallel` construct is one of them, and probably the one that presents more difficulties to compose with free agent threads concerning nesting level, CPU bindings, etc. We believe that this construct should be initially restricted for tasks executed in free agent threads. A free agent thread could also encounter a `taskgroup` or a `taskyield` construct, or any other construct that causes a task switching point. The only issue here is that the free agent thread is not guaranteed to exist when the task becomes ready again. Therefore, all tasks executed in a free agent thread should be considered *untied tasks*.

3.1 Considered Aspects in the Design

Free Agent Threads Are Not Organized in Teams. OpenMP threads are typically grouped in *teams*. An *OpenMP team* is a set of one or more threads created for a specific parallel region, whether the implicit parallel region or a region generated from a `parallel` construct. In the case of explicit parallel regions, the thread that encounters the `parallel` construct creates the *team*. All the threads in that team will participate in the execution of the parallel region.

During the initial design discussions, we explored the idea of free agent threads being part of the same team, as in the hidden helper thread implementation. It certainly has some benefits, like an already defined task scheduler model and implementation. But, it makes the model too strict for the use cases that have motivated us for this article. We want to propose a model where free agent threads are free to steal explicit tasks from any other team, not just the tasks bound to a specific team or a thread set. Furthermore, the term *team* is well defined in the OpenMP specification, and we believe that expanding its definition for including free agent threads would be confusing.

By not constituting an exclusive *team* or forming part of any other regular *team*, free agent threads will not participate in some team-wide synchronization constructs, such as *barriers*. But they acquire some advantages:

- The number of participating free agent threads may be dynamic. Unlike *teams*, the free agent threads group is an asynchronous structure. It will be created during the initialization, but the number of participating free agent threads might be modified at any time by using a runtime library routine.
- The execution of explicit tasks by free agent threads is not limited to tasks bound to their *team* since there is none. Explicit tasks are still bound to the thread set of the current team and optionally to the free agent threads set.

Free Agent Threads Might Be Dynamically Enabled or Disabled. There is a necessity for application developers, users, and third-party tools to have mechanisms to set the initial values or to dynamically change the number or the state of free agent threads. The runtime must provide tools in the same way that allows setting or modifying the number of threads.

These mechanisms are detailed in Sect. 3.3, but we distinguish some concepts. They may be explicitly set using an environment variable, a runtime library routine, an OMPT entry point, or decided by the implementation. First, the total number of *existing free agent threads* is self-explanatory but does not tell their situation, only that they are known. Then, the global *free agent threads policy* is a single value that affects all the existing threads and states whether they are enabled or disabled. And last, the *free agent thread state* is a per-thread value that manages whether a specific free agent thread may execute some explicit tasks, but only if the global policy allows it.

3.2 The free_agent Task Clause

Free agent threads are intended for executing explicit tasks in situations where the parallel region cannot exploit all the parallelism in the system. The `task` construct generates an explicit task from the code for the associated structured block, with an accordingly created data environment for the task that will be destroyed when the structured block is completed. Since the task becomes an independent entity of work, any free agent thread will execute it as long as the task has been deferred.

Although, until now, tasks were supposed to be executed by any team member, so developers may have written code relying on that. Listing 1.1 shows a task where some data is stored in a private buffer indexed by *thread number*. The operation is not protected with a mutual exclusion because the developer expected only one thread to modify this address. If `omp_get_thread_num()` returns 0 then the above assumption is invalidated; if it returns a unique number, the program will probably incur a memory access violation.

Listing 1.1. Task invoking a team related function.

```
#pragma omp parallel
{
    #pragma omp task
    buffer[omp_get_thread_num()] += f();
}
```

Another example is shown in Listing 1.2, where at the end of the parallel region, the participating threads perform a reduction of their respective *thread-private* variables. In this example, if a free agent thread would have executed any task, their accumulated value in `counter` will not be added to `result`.

Listing 1.2. Reduction assuming that tasks are executed by threads in the team.

```
int counter = 0;
#pragma omp threadprivate(counter)
...
#pragma omp parallel
{
    #pragma omp taskgroup
    #pragma omp task
    counter += f();

    #pragma omp for schedule(static)
    for(int i=0; i<omp_get_num_threads(); ++i)
        #pragma omp atomic
        result += counter;
}
```

There may be other programming patterns where developers did not foresee that threads might execute explicit tasks outside the team. For this reason, we propose the new clause `free_agent(bool-expr)` for the `task` and `taskloop` constructs.

Since adding a new clause to many constructs might be time-consuming for application developers, we also propose an environment variable to set the default behavior: `OMP_FREE_AGENT_TASKS={true,false}`.

– In a `task` or `taskloop` construct, if a `free_agent` clause is present and evaluates to *true*, or if the environment variable `OMP_FREE_AGENT_TASKS` is set to `true` and a clause `free_agent` does not evaluate to *false*, the generated task may be executed by any thread in the team or by any free agent thread.

3.3 Proposed Mechanisms to Manage Free Agent Threads

We propose the following OpenMP environment variables to configure the initial state of free agent threads in an OpenMP program. We also offer a set of runtime library routines for applications to modify the state at run time. And finally, we propose a set of entry points in the OMPT callback interface for OMPT tools to gather information of free agent threads and enable or disable specific ones.

Environment Variables

– `OMP_FREE_AGENT_NUM_THREADS:` sets the initial number of free agent threads to use.
– `OMP_FREE_AGENT_PROC_BIND:` sets the thread affinity policy to be used for free agent threads. The value of this environment variable might be `true`, `false`, `initial`, `close`, or `spread`. This variable is the equivalent of `OMP_PROC_BIND` for free agent threads, except that it is relative to the initial thread.
– `OMP_FREE_AGENT_PLACES:` sets the place partition for free agent threads. The allowed values are the same as in `OMP_PLACES`.
– `OMP_FREE_AGENT_WAIT_POLICY:` sets the desired behavior of free agent threads that are waiting. Possible values are `active` or `passive`.
– `OMP_FREE_AGENT_POLICY:` sets the initial policy for free agent threads. Possible values are `enabled` or `disabled`. If the value is `enabled`, free agent threads will be able to execute explicit tasks. If the value is `disabled`, free agents must be suspended or even yet not created, and they must not execute any explicit task.
– `OMP_FREE_AGENT_TASKS:` sets whether all tasks are considered to have the `free_agent` clause.

Runtime Library Routines

– `int omp_get_num_free_agent_threads(void):` returns the number of existing free agent threads.
– `void omp_set_num_free_agent_threads(int num_threads):` affects the number of free agent threads to be used by the runtime. If `num_threads` is greater than the current number of free agent threads, the runtime may create new ones. If `num_threads` is less than the current number of free agent threads, exceeding threads are destroyed or suspended, but they will not count as existing free agent threads.
– `void omp_set_free_agent_policy(omp_free_agent_policy_t policy):` sets the global policy, same as the variable `OMP_FREE_AGENT_POLICY`.

Entry Points in the OMPT Callback Interface

- `int ompt_get_num_free_agent_threads(void)`: returns the number of existing free agent threads.
- `int ompt_get_free_agent_thread_id(void)`: returns the internal thread identifier of the free agent thread. The number must be in the range of *0..n-1*, where *n* is the number of existing free agent threads.
- `void ompt_set_free_agent_thread_state(int free_agent_id, int state)`: sets the individual state of the specified free agent thread. The `state` argument can be either `enabled` or `disabled`.

4 Implementation

We have implemented a subset of the free agent threads proposal in the LLVM OpenMP runtime [1]. Of the ∼80 kSLOC of the runtime (not counting the library for `target` support), our implementation required changing ∼800 SLOC. This suggests that a complete implementation of our proposal would have reasonable implementation complexity.

The runtime creates one operating system thread (a `pthread` in Linux) for each free agent thread. The number of free agents threads is defined by the environment variable `OMP_FREE_AGENT_NUM_THREADS`. Free agents can be enabled or disabled, and `OMP_FREE_AGENT_POLICY` characterizes their initial state. Creation of the free agent threads happens simultaneously the runtime initializes, typically upon encountering the first OpenMP construct or OpenMP API call.

The LLVM OpenMP runtime keeps two data structures related to the team of a parallel region. One corresponds to the proper *team of threads*, and another one is named the *task team*, which exists only if threads of the team create explicit tasks. There is one queue of explicit tasks ready to be executed for each thread of the team. When a task team is first created, all the free agent threads are *allowed* to execute tasks of that task team. During the finalization of the parallel region (when all the explicit tasks of that team have been completed), free agents are not allowed to execute tasks of the finishing task team anymore.

The lifecycle of an enabled free agent thread is a loop for each of the allowed task teams. Once the free agent thread *enters* a task team, it executes as many explicit tasks as possible. It does this by stealing tasks from other (regular) threads of the task team. While executing an explicit task, a free agent thread is logically inside the task team, but it does not belong to the team of threads. The semantics of team-requiring operations such as a call to `omp_get_thread_num` or usage of `threadprivate` variables are for now intentionally left undefined. Once no more tasks remain in the task team, the free agent thread *leaves* it. Once all the allowed teams have been processed, the free agent thread is suspended to avoid a busy loop.

When a thread of the team creates an explicit task, if there is a suspended free agent thread, then the runtime will resume it. Free agent threads are also resumed when they are enabled by the user code and periodically when threads of the team are executing tasks.

A free agent thread (with free agent thread number n) can create an explicit task while executing another explicit task. When this happens, the new task is added to the queue of the corresponding thread number n of the task team (modulo the number of threads of the team).

In general, the LLVM OpenMP runtime does not defer the execution of explicit tasks created in inactive parallel regions (regions executed by teams with only one thread). However, to support `detached` tasks, the LLVM OpenMP runtime can defer tasks also in inactive parallel regions. Our implementation leverages this feature to allow deferring tasks created in such regions when free agents are available. This enables a scenario where `OMP_NUM_THREADS=1` and `OMP_FREE_AGENT_NUM_THREADS` ≥ 1.

5 Evaluation

The free agent threads implementation presented in this article has been tested with applications to evaluate its performance. We expose two different use cases to demonstrate the potential of free agent threads in different scenarios.

We analyze a pure OpenMP application in the first use case that presents a load imbalance between two nested parallel regions. Free agent threads execute explicit tasks encountered in the most loaded parallel regions, thus alleviating the load balance issue.

The second use case shows a task-based MPI+OpenMP application that presents a load imbalance among processes. A third-party tool, DLB, can exploit the free agent threads enable and disable mechanism to modify the number of productive threads assigned to a process to fix the load imbalance in hybrid applications.

All the results have been obtained on the MareNostrum 4 supercomputer. It is composed of compute nodes with two sockets Intel Xeon Platinum 8160 2.1GHz 24-core and 96 GB of main memory. Regarding the software, we used the Intel compiler (version 17.0.4), a modified LLVM OpenMP runtime (based on LLVM 11.0.0, OMP version 5.0.20140926), and DLB 3.0. Since we used the Intel compiler, we have not implemented the `free_agent` clause, and we assume that free agent threads can safely execute all tasks.

5.1 Use Case: Fixing Load Imbalance Between Parallel Regions

When an OpenMP thread reaches a task scheduling point, it may suspend the execution of the current task and switch to a different task bound to the same team. This task scheduling model allows, among other things, to use threads that may have already finished their work to execute other pending explicit tasks encountered in the same team. This model is crucial to avoid load balance issues when the task creation is not perfectly distributed. Also, it does not necessarily impact the performance in simpler algorithms such as a single thread creating all the explicit tasks since all the threads in the team will participate in their execution. However, threads may only switch to other tasks in the same team.

In cases where the application has nested parallel regions, idle threads in one parallel region cannot help and execute the tasks of a different parallel region.

The Density Matrix Renormalization Group (DMRG++) is a condensed matter physics application developed at ORNL used to study the superconductivity properties of materials. For our study, we used a mini-app [3,6] that captures the computation core of DMRG++. The code has been slightly modified from previous versions; the structure of the code used is shown in Listing 1.3.

Listing 1.3. DMRG++ code structure.

```
for (int it = 0; it < NIts; ++it) {
  #pragma omp parallel for num_threads(X)
  for (int ipatch = 0; ipatch < npatches; ipatch++) {
    // ...
    #pragma omp parallel for schedule(dynamic, 1) num_threads(Y)
    for (int jpatch = 0; jpatch < npatches; jpatch++) {
      // ...
      #pragma omp taskloop
      for (int k = 0; k < k_size; k++) {
        // Loop body
      }
    }
  }
}
```

Figure 1 shows two Paraver [2,17] traces of two different DMRG++ executions. The Paraver traces in the figure represent a timeline in which the X-axis is the elapsed time, the Y-axis the OpenMP threads, and explicit tasks are shown in blue for each thread that executes them. The first trace shows an iteration of a DMRG++ execution with two levels of nesting; 4, 4, distributed on 16 logical CPUs. The program has been executed with `OMP_PLACES="{0,1,2,3},{4,5,6,7},{24,25,26,27},{28,29,30,31}"` to bind each OpenMP thread to a specific core. It can be appreciated how the load imbalance of one of the innermost parallel regions causes the threads of the other team to wait.

Fig. 1. DMRG++ trace execution with a 4,4 nesting running on 16 CPUs, and same execution running with free agent threads. Both traces are at the same duration time scale.

The second trace is the same configuration but enabling free agent threads at the end of the iteration. Those free agent threads are different *pthreads*, and Paraver draws them in another row, but they use the same logical CPUs as the threads that just finished their region. As done with the execution without free agents, the same `OMP_PLACES` is used, and the clause `OMP_FREE_AGENT_PLACES="{0,1,2,3,4,5,6,7,24,25,26,27,28,29,30,31}"` is used to determine where the free agents can be executed. This use case shows how free agent threads can be exploited on otherwise unproductive CPUs to increase the parallelism when needed.

The performance results of DMRG++ with free agent threads are shown in Fig. 2. Different nesting configurations have been evaluated with a variable range of free agent threads and wait policies *active* and *passive*. The nomenclature $N \times M$ in the legend represents the OpenMP threads per nesting level: N for level 1, M for level 2, executed in as many CPUs as needed to bind only one OpenMP thread to a logical CPU, without considering free agent threads, e.g., the configuration "2×4 passive" has been executed with `OMP_NUM_THREADS=2,4`, `OMP_WAIT_POLICY=passive`, and using only eight logical CPUs, regardless of the number of free agent threads used. The speedup values represent the relative performance of each case with zero free agent threads, so only the effect of free agent threads is shown. The number of free agent threads is limited to the number of threads in the second level. Since free agent threads are running on the CPUs of the faster parallel region, it would not be efficient to increase the number of free agent threads to higher values.

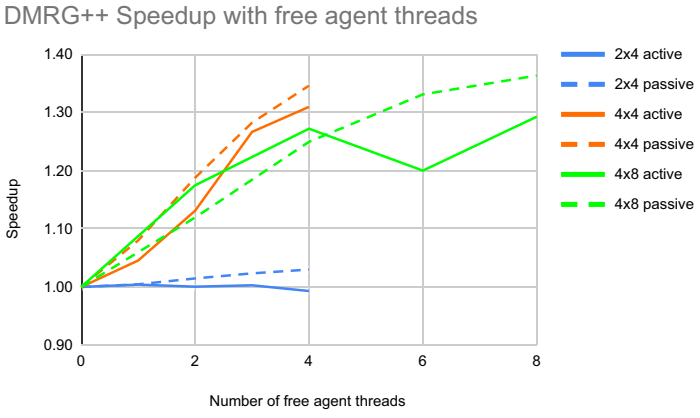

Fig. 2. Speedup of DMRG++ with free agent threads.

As it can be observed in the figure, the runtime can dynamically enable the free agent threads to increase the number of active threads executing tasks. Thus, load imbalance between parallel regions may be reduced and improve the overall performance execution. In this case, we obtain up to a 36% speedup when using free agent threads on a pure OpenMP application.

5.2 Use Case: Solving Load Imbalance in a Hybrid Application with DLB as an OMPT Tool

In the second use case, we analyze a more common situation: load imbalance among processes. However, to efficiently use the free agent threads to solve this load imbalance, we need a third-party tool responsible for enabling and disabling the free agent threads of each process.

For this use case, we analyze a kernel extracted from Alya [20], a computational fluid dynamics (CFD) code optimized for HPC environments developed at BSC. This kernel presents an iterative pattern of MPI communications followed by a region of task-based computation with a slight load imbalance, $LB = 0.74$. The task-based region also challenges resource balance techniques since the average task duration is only 200 μs. Figure 3 shows a Paraver trace of a hybrid MPI+OpenMP execution and a trace of the same configuration with DLB. In the second trace, DLB runs as an OMPT tool monitoring the OpenMP events of each MPI process and selectively enabling or disabling free agent threads to fix the load imbalance with temporary helper threads. The right-hand side of the figure shows a zoom of a few processes at the end of an iteration. It can be appreciated how some free agent threads are enabled, acting as helper threads only when a logical CPU becomes idle after another process reaches an MPI synchronization call.

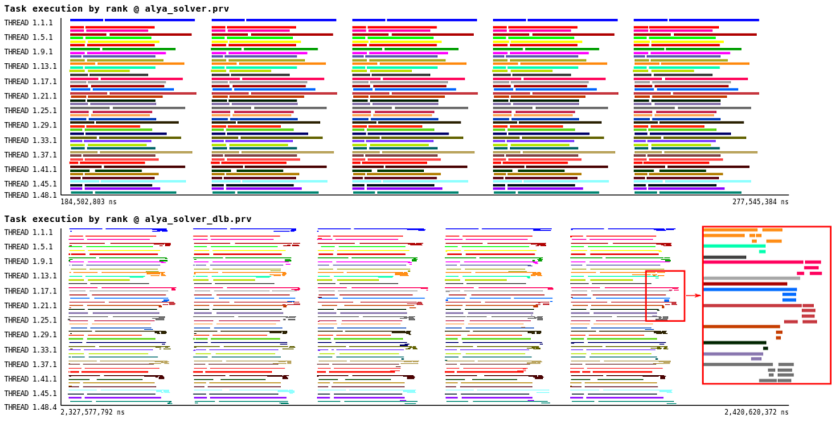

Fig. 3. On top, Alya Paraver trace execution running on 48 CPUs. On the bottom, execution with DLB and free agent threads. Each color represents different MPI ranks. Both traces are at the same duration time scale.

Figure 4 shows the speedup comparison of the Alya kernel running with 48 MPI ranks, 1 OpenMP thread each, and a variable number of free agent threads. Due to the fine granularity of the tasks, using more threads than needed causes a slight performance drop, which may have been caused either by our implementation or by some scheduling decision in DLB. There is still some future

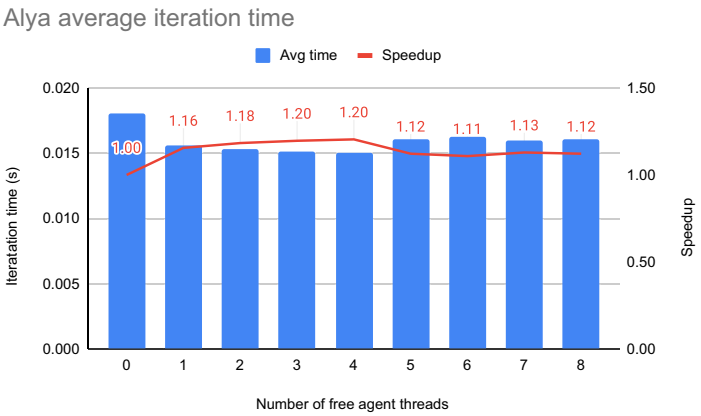

Fig. 4. Time and Speedup of Alya running 48 MPI processes, with DLB and a variable number of free agent threads.

work in how DLB manages free agent threads, but even with a proof of concept implementation, the execution was always 10–20% faster.

6 Conclusions and Future Work

In this paper, we have presented a proposal to extend the OpenMP programming model and execution model. Our proposal intends to relax the rigid fork-join approach by allowing the OpenMP threads to participate within the parallel region and outside of it. This approach enables leveraging the assigned processing elements when the OpenMP program has not fork threads yet. For nested parallelism, we can generalize this statement as free agent threads may participate in the execution of work units when the application has not reached the inner level of parallelism it was designed for.

The free agent threads are designed to execute tasks. We consider tasks are helpful to guarantee the required malleability of an application. A resource manager can further exploit this characteristic to balance assigned resources between processes. As the free agent threads are not directly bound to a parallel region, their number may increase or decrease during the program execution. Then, tasking models and dynamic free agent threads are a powerful combination to maximize the application performance.

In Sect. 5, we have presented the results of two different scenarios: intra- and inter-process levels. In both cases, we have proved performance benefits by using a small set of free agent threads. In the intra-process use case, we handle fixing the imbalance between OpenMP parallel regions, obtaining up to a 36% speedup. In the inter-process use case, we solve the load imbalance of a hybrid application using the DLB resource manager, and it obtains up to 20% speedup compared to the baseline.

As future work, we plan to further develop more use-cases that can potentially leverage the use of free agent threads. We will also investigate the potential interaction of such threads with other OpenMP mechanisms, as it could be the work-sharing construct (with dynamic schedulers) or TLS based data. Finally, we also plan to study different schedulers and implementation alternatives of our reference framework, especially when a task executed by a free agent thread creates more tasks.

Acknowledgements. This work has been done as part of the European Processor Initiative project. The European Processor Initiative (EPI) (FPA: 800928) has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement EPI-SGA1: 826647. It has also received funding from the European Union's Horizon 2020/EuroHPC research and innovation programme under grant agreement No 955606 (DEEP-SEA); and the support of the Spanish Ministry of Science and Innovation (Computacion de Altas Prestaciones VIII: PID2019-107255GB).

References

1. LLVM OpenMP Runtime. https://openmp.llvm.org. Accessed 18 May 2021
2. Paraver: a flexible performance analysis tool. https://tools.bsc.es/paraver. Accessed 21 May 2021
3. Alvarez, G.: The density matrix renormalization group for strongly correlated electron systems: a generic implementation. Comput. Phys. Commun. **180**(9), 1572–1578 (2009)
4. Barcelona Supercomputing Center: OmpSs Specification. https://pm.bsc.es/ompss. Accessed 04 Nov 2020
5. de Supinski, B.R.: Recent, Current and Future OpenMP Directions: OpenMP 5.1 and More!. https://www.openmp.org/wp-content/uploads/OpenMP_SC20-deSupinski.pdf. Accessed 01 July 2021
6. Criado, J., et al.: Optimization of condensed matter physics application with OpenMP tasking model. In: Fan, X., de Supinski, B.R., Sinnen, O., Giacaman, N. (eds.) IWOMP 2019. LNCS, vol. 11718, pp. 291–305. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-28596-8_20
7. D'Amico, M., Garcia-Gasulla, M., López, V., Jokanovic, A., Sirvent, R., Corbalan, J.: DROM: Enabling Efficient and Effortless Malleability for Resource Managers, p. 41 (2018)
8. Duran, A., et al.: OmpSs: a proposal for programming heterogeneous multi-core architectures. Parallel Process. Lett. **21**, 173–193 (2011)
9. 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)
10. Garcia-Gasulla, M., et al.: MPI+ X: task-based parallelisation and dynamic load balance of finite element assembly. Int. J. Comput. Fluid Dyn. **33**(3), 115–136 (2019)
11. Intel Corporation: Intel Cilk++ SDK Programmer's Guide (2009). https://www.clear.rice.edu/comp422/resources/Intel_Cilk++_Programmers_Guide.pdf
12. Intel Corporation: Intel Threading Building Blocks (2011). https://www.inf.ed.ac.uk/teaching/courses/ppls/TBBtutorial.pdf

13. Massachusetts Institute of Technology: OpenCilk Language Extension Specification Version 1.0 (2021). https://cilk.mit.edu/docs/OpenCilkLanguageExtensionSpecification.htm
14. OpenMP Architecture Review Board: OpenMP Application Programming Interface, Version 3.0 (2008). http://www.openmp.org/
15. OpenMP Architecture Review Board: OpenMP Application Programming Interface, Version 4.0 (2013). http://www.openmp.org/
16. OpenMP Architecture Review Board: OpenMP Application Programming Interface, Version 5.1 (2020). https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf. Accessed 22 March 2021
17. Pillet, V., Labarta, J., Cortes, T., Girona, S.: Paraver: A tool to visualize and analyze parallel code. In: Proceedings of WoTUG-18: Transputer and Occam Developments, vol. 44, pp. 17–31 (1995)
18. Sunderland, D., Olivier, S.L., Hollman, D.S., Evans, N., de Supinski, B.R.: Making OpenMP Ready for C++ Executors (2019). https://www.osti.gov/biblio/1559921
19. Tian, S., Doerfert, J., Chapman, B.: Concurrent Execution of Deferred OpenMP Target Tasks with Hidden Helper Threads. Springer (2020)
20. Vázquez, M., Houzeaux, G., Koric, S., et al.: Alya: multiphysics engineering simulation toward exascale. J. Comput. Sci. **14**, 15–27 (2016)