Master Thesis

Master in Innovation and Research in Informatics:
High Performance Computing

Design and implementation of role-shifting threads in the LLVM OpenMP runtime

June 2022

Author: Joel Criado Ledesma

Supervisor: Víctor López Herrero (UPC, BSC-CNS)

Co-supervisor: Marta Garcia Gasulla (BSC-CNS)

Facultat d'Informàtica de Barcelona (FIB)

Acknowledgments

This master thesis and the master's, in general, have been quite the journey, and I wouldn't be at this point without the support from my colleagues, family, and friends.

I want to thank all my colleagues from the BePPP group, especially my advisors, Victor and Marta. They have helped me in the most challenging moments of the thesis, and I couldn't be finishing it without them. Thanks for all your help and guidance.

Finally, I want to dedicate this work to my grandmother Maria. Wherever you are, I hope you are proud of what I have accomplished.

Abstract

Efficiency is a must in the HPC world. Supercomputers are extensively used in public research institutions, and the CPU time is limited and should be used responsibly. Users can improve their applications to that end, but tools and programming models must continue improving accordingly. Nowadays, supercomputers are heterogeneous machines and keep increasing the number of cores. Making all the software stack, from applications to runtimes, more malleable and flexible should be one of the community goals. It is not only necessary for the heterogeneity of the resources, but it will also help deal with system noise, load imbalances, communication inefficiencies, and dynamic workloads.

This thesis presents the role-shifting threads, an evolution of the current OpenMP threads designed to improve the malleability and flexibility of the model. Depending on their roles, when not required in a *parallel region*, the threads may be used for other purposes. The free agent role has been added to the model, aiming to execute *explicit tasks* outside *parallel regions*. In addition, the Clang compiler has been modified to comply with the additions to the runtime. Furthermore, the role-shifting model has been integrated with the Dynamic Load Balance (DLB) library, used to dynamically solve imbalances in MPI+OpenMP applications. All software is released open-source.

The utility of the role-shifting threads, and the free agent role, is demonstrated with three real-world scientific applications, one of them with a coupling case. Speedups from 1.192x to 2.44x are reached with the new runtime compared to a vanilla run, demonstrating the potential of this new model.

Keywords: OpenMP | LLVM | Clang | Runtime | Parallelism | Tasking | Free agents | MPI | Dynamic load balancing | High Performance Computing

Contents

List of Figures

Listings

Chapter 1

Introduction

1.1 Motivation

The HPC community is immersed in the exascale race nowadays. The use of accelerators is widely extended and will be a crucial factor in achieving exascale performance. Nevertheless, extracting the best performance from the existing components (i.e., CPUs) is also of great importance. For that reason, programming models must provide support and performance for the new architectures and deliver good efficiencies for a high number of cores per socket. As OpenMP [22] is the most extended shared-memory parallel programming model, the HPC community has big hopes in it to address these challenges.

In recent years, the number of cores per socket has increased constantly. In that scenario, problems like computational load imbalance, system noise, or lack of parallelism may significantly affect the performance of HPC applications. Therefore, the programming models must adapt to those problems during execution, offering flexibility and malleability to adjust the execution at runtime transparently to the user.

Role-shifting threads are an evolution of the current OpenMP threads designed to increase the malleability of OpenMP and, at the same time, offer more possibilities to the developers of applications. The main idea behind role-shifting threads is to have a unique pool of threads that the runtime can use to perform different jobs. As usual, they will participate in the OpenMP *parallel regions* as regular *workers* and do other things based on their roles when not required for those regions.

The main role integrated into the runtime is the free agent [20] role. Previously, free agents were a stand-alone type of thread, but they have been adapted to become a new role. Threads with this role as active do not cooperate in executing the *implicit tasks* from a `parallel` construct; instead, they are dedicated to executing *explicit tasks*. Free agent threads increase the malleability and flexibility of OpenMP, allowing the runtime to execute *tasks* in idle resources. Free agent threads are one of the main objectives for the OpenMP 6.0 standard [4].

1.2 Goals

This project aims to improve the OpenMP runtime by introducing the concept of roles. This will allow the runtime to have a unique pool of threads that perform different operations based on their available roles and runtime decisions. In particular, the goals of the project are the following:

- Design and implement the concept of roles in the LLVM OpenMP runtime, and prepare it to perform shifting operations of roles transparently to the user.

- Integrate the free agent threads as a supported role for the runtime.
- Prepare the runtime to include more roles in the future easily.
- Integrate the Dynamic Load Balance (DLB) library [14] with the role-shifting threads.
- Add a new clause to the `task` and `taskloop` directives of OpenMP to indicate if a free agent can execute a *task*. Modify the LLVM Clang compiler to generate appropriate code with this new clause.
- Evaluate the role-shifting threads and the DLB integration performance.
- Push the role-shifting threads proposal to the OpenMP standard.

1.3 Document organization

The rest of this document is organized as follows. Chapter 2 gives a high-level explanation of the topics related to the project: OpenMP, LLVM, free agent threads, and DLB. Chapter 3 covers the design of the role-shifting threads, stating the previous runtime issues and describing the changes needed to overcome them. It also describes the methodology used in the development. In Chapter 4, the changes made to the LLVM source code are detailed, including both the OpenMP runtime and the Clang compiler. After that, Chapter 5 explains in detail how the role-shifting threads have been integrated with DLB. The document continues with Chapter 6, where the performance is evaluated using three HPC applications. Finally, Chapter 7 explains the conclusions of the project and the future work.

Chapter 2

State of the art

2.1 OpenMP

OpenMP [22] is an application programming interface (API) for shared-memory multiprocessors programming. It is supported by a wide variety of compilers and architectures and works with C, C++, and Fortran. OpenMP offers different compiler directives, also called `pragmas`, to annotate the code and express the parallelism, a set of API calls, and environment variables.

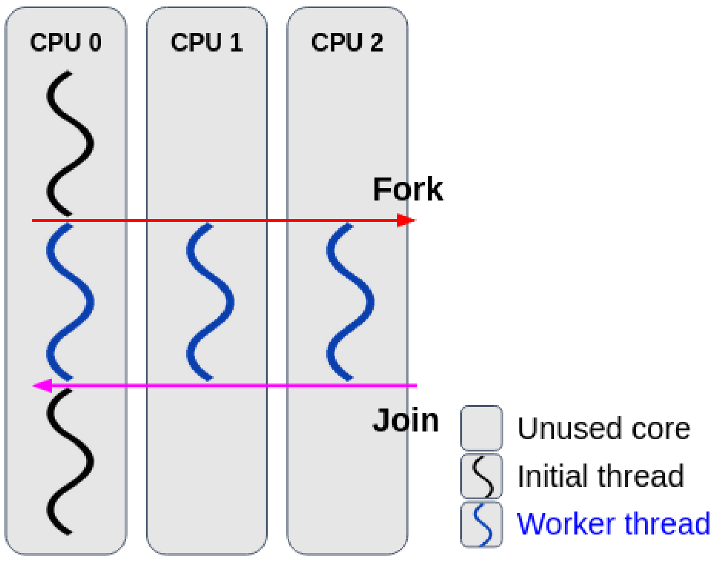


Figure 2.1: Fork-join model.

OpenMP uses a fork-join model. When a thread encounters a code region marked with the `parallel` construct, it creates some threads (fork) that will form an OpenMP *team*. Those threads will run concurrently and execute all the *implicit* and *explicit tasks* of the *parallel region* associated with them. After that, all the threads reach the same point, and the *parallel region* is closed (join). Figure 2.1 shows an example of a fork-join execution.

OpenMP provides a specification of the API but not an implementation. At the time of writing, the most recent specification is the 5.2 version released in November 2021, but no implementation fully supports it yet. The most widely extended implementations are the GNU and LLVM ones, which are Open Source, and some vendor-specific implementations like Intel and ARM.

2.1.1 Threading

OpenMP employs threads, execution entities with its own stack and *threadprivate* memory, to perform the useful work. All the threads under OpenMP are considered OpenMP threads, and they can also fall under other categories:

- **Idle thread**: An OpenMP thread currently not participating in any *parallel region*.
- **Initial thread**: The thread that starts the execution and initializes the runtime.
- **Primary thread**: An OpenMP thread with *thread number* 0. It is either an *initial thread* or a thread that encounters a `parallel` construct, creates a *team* and the *implicit tasks*, and executes one of the *tasks* with *thread number* 0.
- **Worker thread**: An OpenMP thread that participates in a *parallel region* executing one of the *implicit tasks*, but it is not the *primary thread*.
- **Thread number**: A number used to identify the OpenMP threads of a *team*. The *primary thread* has the *thread number* zero, and consecutive numbers identify the other threads of the *team*.

2.1.2 Worksharings

Worksharings are a type of OpenMP directive used to divide the execution of a region of code among the associated *team* members. These regions have a synchronization point at the end, where all the threads will wait until everyone has finished their work unless a `nowait` clause is specified. The most relevant worksharing directives for the project are the following two:

- **Single**: Specifies that the encountering region has to be executed only by one thread of the *team*, and it does not require it to be the *primary thread*. The rest of the threads will wait at the end of the region unless a `nowait` clause is specified. The `single` directive is usually used with the `task` directive (see Section 2.1.3) to generate all the *explicit tasks* in a serial fashion.
- **Worksharing-Loop**: This construct is specified with the `for` directive in C/C++ and `do` in Fortran. It is used to divide the work of one loop (or more if specified with the `collapse` clause) among the *team's* threads and execute it in parallel. The loops must have a particular structure (e.g., only evaluates one expression), called the canonical form, so the flexibility it offers is quite limited.

2.1.3 Tasks

A *task* is a specific instance of code and its data environment that the OpenMP runtime can schedule for execution by threads. There are two main types of *tasks*:

- **Implicit task**: A task generated by an *implicit parallel region* or when a `parallel` construct is explicitly used.
- **Explicit task**: All the *tasks* that are not *implicit*. They are generated with different OpenMP constructs. For the sake of simplicity, we will call *explicit tasks* as *tasks* from now on.

Generally, *tasks* offer more flexibility to the programmer compared to worksharings to define work. The user can define *task dependencies* to determine the execution order of the *tasks*, allowing the user to exploit parallelism in some situations where worksharings fail. Also, *tasks* work perfectly with linked lists and recursivity and offer mechanisms to tackle those properly. However, *tasks* are more complex to program than worksharings.

The most relevant tasking constructs for the project are the following ones:

- **Task**: Specifies that the code region marked with the construct is an *explicit task*. Usually, after creating a *task*, that thread will defer its execution, and any thread in the *team* may execute the *task*. However, if the *task* is generated inside a *serial team*, it will be executed immediately by the same thread that created it.

- **Taskloop**: Specifies that the iteration of one or more associated loops must be executed using *tasks*. Each task may consist of one or more iterations of the related loops.

- **Taskwait**: Indicate that the thread must wait on the completion of *tasks* created by the *current task* (child tasks).

2.1.4 OMPT

OMPT is an interface for first-party tools introduced in OpenMP 5.0. OMPT provides mechanisms to initialize the tool, determine the capabilities of an OpenMP implementation, examine the OpenMP state information associated with a thread, interpret the call stack of an OpenMP thread, receive notifications about certain OpenMP events, a tracing interface for OpenMP target devices, and a runtime library to control a tool from an OpenMP application.

Callback functions are the most relevant feature for the project. These are routines from the first-party tool invoked directly from the OpenMP runtime when certain events happen, enabling the tool to perform certain operations based on the notifications received. The functions are registered in the initialization provided by the OMPT interface, so the runtime knows which address should call in every case.

The most relevant callbacks for the project are the following ones:

- `void ompt_callback_thread_begin_t(ompt_thread_t thread_type, ompt_data_t *thread_data)`: The callback is used when an OpenMP thread is created. The `thread_type` argument indicates if the thread starts as a worker (`ompt_thread_worker`) or as a free agent (`ompt_thread_other`).

- `void ompt_callback_parallel_begin_t(ompt_data_t * encountering_task_data, const ompt_frame_t *encountering_task_frame , ompt_data_t *parallel_data, unsigned int requested_parallelism, int flags, const void *codeptr_ra)`: The callback is used when a *parallel* or *teams region* starts. The callback is useful to discern if the code is entering a *parallel region* and the level of nesting, as well as the number of OpenMP threads requested for the region.

- `void ompt_callback_parallel_end_t(ompt_data_t *parallel_data, ompt_data_t *encountering_task_data, int flags, const void *codeptr_ra )`: The callback is used when a *parallel* or *teams region* ends. It serves as the counterpart of the previous callback.

- `void ompt_callback_task_create_t(ompt_data_t *encountering_task_data , const ompt_frame_t *encountering_task_frame, ompt_data_t * new_task_data, int flags, int has_dependences, const void *codeptr_ra)`: The callback is used each time the runtime creates a *task*. It allows to control the number of *explicit tasks* created and make decisions based on that.
- `void ompt_callback_task_schedule_t(ompt_data_t *prior_task_data, ompt_task_status_t prior_task_status, ompt_data_t *next_task_data)`: The callback is used each time the runtime makes a *task scheduling* decision. The `prior_task_status` argument indicates the type of scheduling point reached. The value of `ompt_task_switch` indicates the execution start of a *task*, and `ompt_task_complete` indicates the execution ending of a *task*.

2.2 LLVM

LLVM [19] is an umbrella project for the development of toolchains and compilers consisting of various subprojects. It started as a research project at the University of Illinois and nowadays is widely used in the academic and industry worlds. Two subprojects are relevant for this work: the OpenMP runtime and the Clang C/C++ compiler.

2.2.1 OpenMP runtime

The OpenMP runtime is divided into two main libraries: *libomp* and *libomptarget*. The first is the *host* library, which manages the threading and task models. The other is the *device* library, designed to offload the computations to such devices. *Libomp* is the relevant one for this project; the document will refer to it as LLVM OpenMP runtime from now on.

The most relevant feature of the runtime for this project is the implementation of the parallel region and its barriers and the management of threads.

When a thread encounters a *parallel region*, that thread becomes the *primary thread* for that region. The thread will create all the necessary structures for the *team* of threads and assign as many threads as needed to the *team*. The first time this happens, the *worker threads* are created. An actual `pthread_create` is used here. Therefore the OpenMP threads are mapped to actual OS threads. Upon completion of the *parallel region*, all the *worker threads* are placed in a structure called thread-pool. The runtime will reuse the threads in the thread-pool for subsequent *parallel regions* and will create new threads only after emptying the thread-pool.

The runtime implements the fork-join model using two different barriers. All the threads not needed for a *parallel region* wait at the fork-barrier. When an idle thread is assigned to a *team*, it is released from the barrier and starts executing its assigned *implicit task*. Note that as soon as a thread is ready, it will start executing and not wait for other threads to arrive at the barrier. This type of barrier is called release-barrier.

After ending their structured block associated with the `parallel` construct, the threads reach the join-barrier. Here, all threads must wait until every thread involved in the *parallel region* reaches the barrier. Then, all the threads advance to the fork-barrier.

Figure 2.2 shows the described behaviour of a *worker thread*. During the execution of the *implicit task*, it may reach a *task scheduling* point, where the thread can execute *explicit tasks*. These points are typically `taskwait`, `taskgroup`, and `barrier` constructs, as well as the implicit barrier at the end of a *parallel region*.

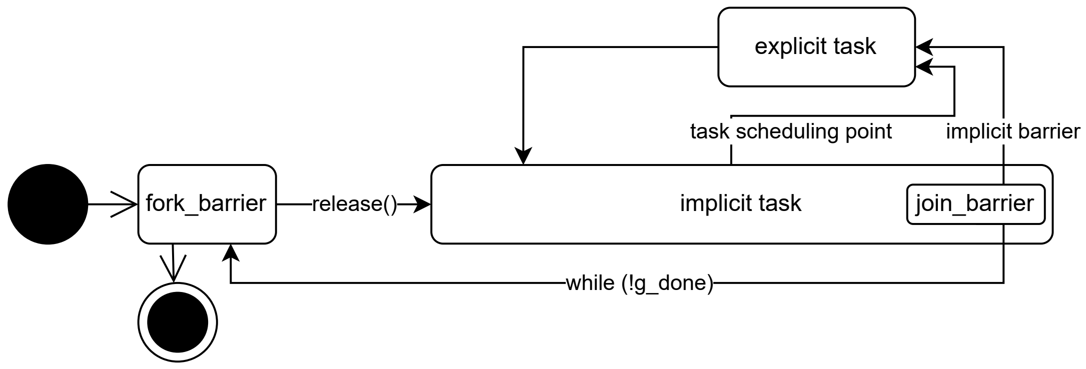


Figure 2.2: OpenMP worker thread flowchart.

2.2.2 Clang compiler

Clang [18] is the open-source compiler's front-end for the C family of programming languages of the LLVM project. One of Clang's key features is compatibility with most GCC flags. Other relevant features are faster compile times, better memory usage than other compilers, highly expressive error and warnings diagnostics, and a modular library-based architecture.

LLVM also has libraries dedicated to the middle-end and back-end of the compilation, which work in conjunction with Clang to generate the final executable. LLVM uses the LLVM Intermediate Representation (IR) in the middle-end to perform the optimizations. It is designed to be lightweight, low-level, and expressive simultaneously. Therefore, the programmer/user can understand it, and the compiler can still optimize the code easily. The front-end lowers the source code into the LLVM IR for the optimizer phase. Then the back-end generates the final executable from the optimized LLVM IR for the target architecture.

Internally, the LLVM compilation is divided into six different steps:

- Driver: The Clang executable is, in fact, a small driver which controls the execution of all the tools involved in the compilation. Typically, the user will not interact with the driver but use it transparently.
- Preprocessing: It handles the tokenization of the source files and expands the macros and included files.
- Parsing and Semantic analysis: It parses the tokens from the preprocessor and generates a parse tree. Then, it performs a semantic analysis of the code, detecting the types of expressions and the correctness of the code. When needed, it will emit errors and warnings. After that, the Abstract Syntax Tree (AST) is generated.
- Code generation and Optimization: This step translates the AST to LLVM IR. Then, several optimizations are applied to the code, including target-specific optimizations. Finally, it translates the resulting LLVM IR into machine code (.s extension files).
- Assembler: It translates the output of the last phase into object files.
- Linker: It merges multiple object files into a single executable or dynamic library.

Since this project aims to add a new clause for the OpenMP task and taskloop constructs, the relevant part of the compiler will be the front-end. In particular, the Parser, Sema (**Sema**ntic analyzer), and CodeGen (**Code Gen**eration) modules of Clang need to be modified.

2.3 Free Agent Threads

Free agent threads [20] are a new type of thread designed to increase the malleability and flexibility of OpenMP. These threads serve a unique purpose: execute *explicit tasks*. This approach aims to relax the rigid fork-join model of OpenMP, having an additional group of threads that can execute *tasks*.

Free agent threads must not be confused with the Hidden Helper Threads [24]. These threads are designed in a similar way; they form a separate set of threads and are used in the offloading of *target regions* to a device. In contrast, free agent threads are designed to execute any *explicit tasks*, with their scope not tied to *parallel regions*, and allow for dynamically changing the number of active threads.

Free agent threads are not part of any *team*. They are not considered when encountering a `parallel` construct; neither can they execute any *implicit task* or participate in a worksharing. Regarding the synchronization constructs, they do not participate in `barriers` (explicit or implicit) but will participate in `atomic` and `critical` sections.

Not forming part of a *team* and not constituting a *team* on their own has a couple of advantages:

- The number of free agent threads is dynamic. They are created at initialization, and after that, the number of active free agents can be modified using a runtime library routine. Therefore, the amount of threads is no longer limited by the `parallel` construct.

- Free agent threads are not limited to executing *tasks* from their *team* since they do not have any. Instead, they may execute *tasks* from any *team*.

Some OpenMP programs use `threadprivate` variables or distribute the work using the `omp_get_thread_num` routine. If a free agent thread executed one of those *tasks*, the program would probably have an incorrect result. For that reason, the initial free agent proposal included the `free_agent(bool-expr)` task clause. By default, a free agent thread cannot execute a *task*, and when the clause is present and the `boolean` evaluates to `true`, a free agent may execute the *task*.

2.3.1 Double-pool implementation

The double-pool is the first implementation of the free agent threads by Lopez et al. [20]. It is characterized by having two separate pools of threads: one with regular OpenMP threads and another with free agent threads. Each of these threads is backed by a different Linux *pthread*.

The execution model is shown in Figure 2.3. The execution has four OpenMP threads, three free agent threads, and only four CPUs. Free agent threads use a CPU freely until a *parallel region* starts. At that point, the *worker threads* are activated in the CPUs used by the free agents, which are deactivated. At the end of the *parallel region*, the free agents are reactivated, and the *worker threads* are put to sleep. If the *worker thread* that should occupy

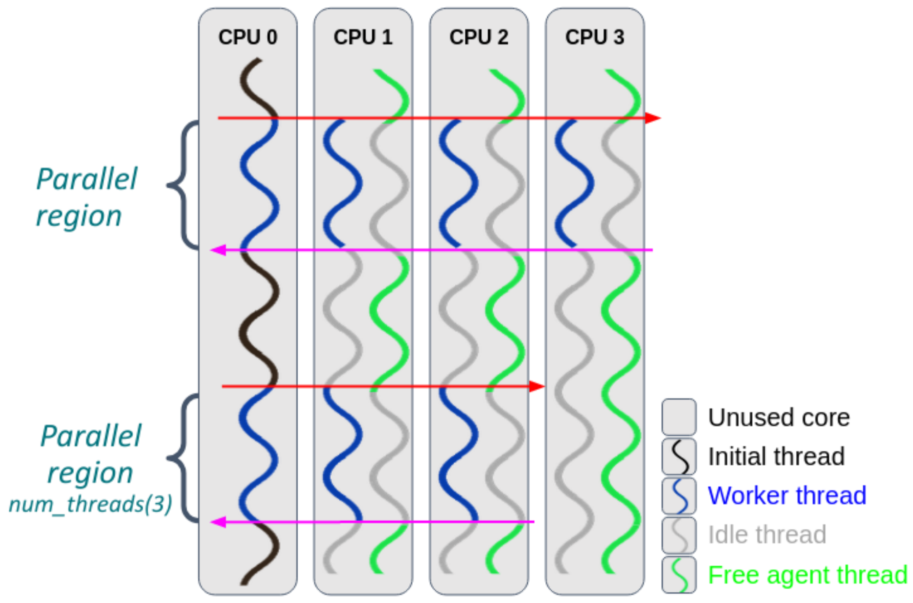


Figure 2.3: Double-pool execution example.

the CPU is not reclaimed in a *parallel region*, the free agent thread can continue using the CPU.

The number of free agent threads is decided with the `OMP_FREE_AGENT_NUM_THREADS` environment variable. When set, free agents will be created at runtime initialization but not at any other point. Therefore, the number of free agent threads must be decided before starting the execution. In addition, the `OMP_FREE_AGENT_POLICY` is used to determine if free agents are active from the beginning or not. If they start disabled, the user must enable them with an API call. Otherwise, they will not participate in the execution.

The LLVM runtime does not defer the execution of explicit tasks when executing a *serial parallel region* (i.e., a *parallel region* with a single OpenMP thread). When free agent threads exist, they could participate in the execution of these *tasks*. Therefore, the implementation allows deferring *tasks* in that regions when free agents are available (`OMP_FREE_AGENT_NUM_THREADS` ≥ 1).

2.4 DLB

DLB [14] (Dynamic Load Balancing) is a library developed at Barcelona Supercomputing Center (BSC-CNS) aimed to speed up hybrid applications (i.e., applications that use both distributed and shared memory parallelism) and optimize the utilization of computational resources.

One of DLB's main goals is to be transparent to the user. Therefore, DLB can be used via the `LD_PRELOAD` mechanism without needing to recompile and link the application again. Nevertheless, DLB also offers a public API, allowing the users to fine-tune their applications and obtain different metrics at runtime.

Since version 3.0 DLB includes three independent modules: LeWI, DROM, and TALP. The following subsections explain each of them.

2.4.1 LeWI

LeWI [13] (**Le**nd **W**hen **I**dle) is used to optimize the performance of hybrid applications without previous analysis of the application. It improves the load balance by redistributing the CPUs inside a node from one process to another. This approach can tackle imbalances from all kinds of sources: data, architectural, algorithm, etc.

Figure 2.4 shows an example of LeWI improving the load balance of a hybrid application. Initially, the application has an imbalance at the outer level (MPI). The second process takes more time to reach the MPI call, and the CPUs from the first process are entirely idle during that time. With LeWI activated, it will detect that the first process has reached a blocking call and lend the idle CPUs to the other process. That way, the program uses all the available resources and finishes the execution faster.

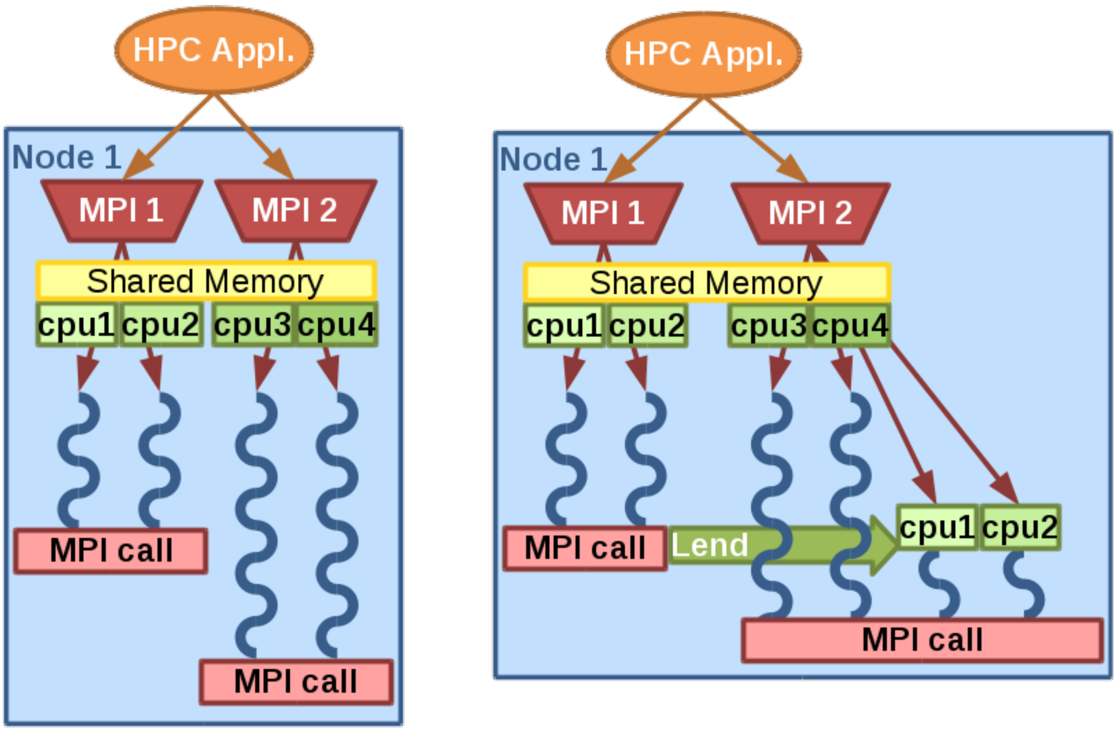


Figure 2.4: Example of LeWI balancing algorithm. On the left, an unbalanced hybrid application. On the right, the same application with LeWI activated and the unbalance solved.

2.4.2 DROM

DROM [10] (**D**ynamic **R**esource **O**wnership **M**anager) is the second module of DLB, introduced in version 2.0. DROM is used to modify the resources assigned to an existing process, modifying the process affinity and the thread affinity. DROM offers an API that an external entity can use, i.e., job scheduler (e.g., SLURM), resource manager, etc., and is already integrated with MPI, OpenMP, and OmpSs [11].

Figure 2.5 shows an example of a DROM use case. In this scenario, an application is running using six CPUs, three per process. At some point, a second application with a higher priority has to be run. Then, the job scheduler uses DROM to modify the resources of app 1 and gives two CPUs to app 2.

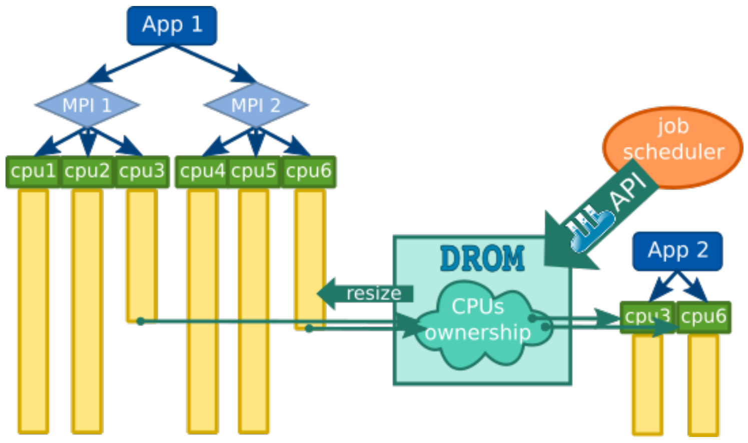


Figure 2.5: Example of DROM. The job scheduler reassigns computational resources from one application to another with higher priority.

2.4.3 TALP

TALP [21] (**T**racking **A**pplication **L**ive **P**erformance) is the newest module of DLB, introduced in version 3.0. TALP is a lightweight, portable, extensible, and scalable tool for measuring the parallel performance of applications. The metrics obtained with TALP allow the users to evaluate their application efficiencies both at runtime and post-mortem. With the API, the user or resource manager can gather metrics at runtime and then adapt the execution based on the dynamic state of the application. A typical scenario of TALP can be seen in Figure 2.6.

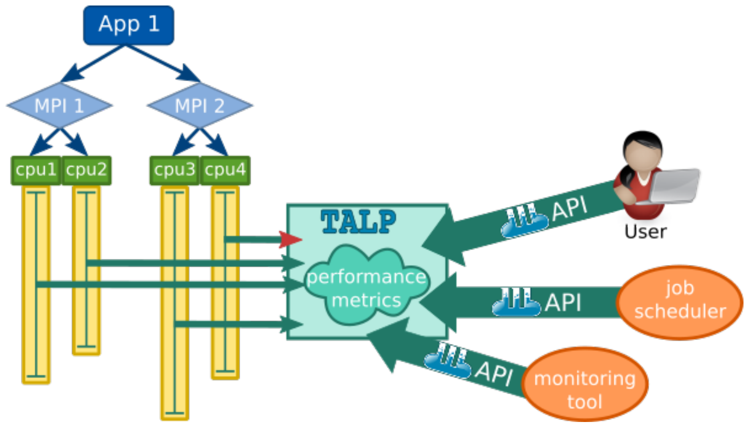


Figure 2.6: Example of TALP usage. An application with two MPI ranks and two CPUs per rank is running, and three different actors gather metrics from the application using TALP.

Chapter 3

Design

The idea of the role-shifting threads came to us after receiving feedback from the OpenMP community about the free agent threads [20] proposal (see Section 2.3.1) and analyzing in detail the performance of the implementation. The drawbacks of the previous implementation were the following ones:

- The total number of free agent threads was fixed at runtime initialization. The number of active free agents could be changed during the execution, but it was limited to the number decided at the start. This limited the malleability of the model drastically.
- An OpenMP thread and a free agent thread were bound to the same physical CPU. Switching the active thread of a CPU is usually really time-consuming, so a strategy where this is not needed is preferred.
- OpenMP has a `thread-limit-var` that indicates the maximum amount of threads that can be created. The more threads the runtime has, the sooner it may reach the limit.

To solve these drawbacks, the main idea is to have a unique pool of threads with several potential roles and shift their roles as the execution demands. The following sections describe all the things considered for the design of the role-shifting threads.

3.1 Roles

The roles are the foundation of these new threads. Roles define the actions that a thread can perform at that moment, and a thread can *shift* from one role to another to perform a different set of actions. The design distinguishes between two kinds of roles:

- **Potential roles**: The available roles of a thread. Each thread can have from 0 to n roles.
- **Active role**: The current role of the thread. It must be listed in the *potential roles* of the thread. A single role may be active simultaneously.

Both *workers* and the *initial thread* can have roles since this simplifies the specification. The *initial thread* must probably not use any role since it typically performs some critical functions, and the runtime cannot afford to delay them. Nevertheless, this is implementation-defined, meaning that it is not defined in the standard but decided by each implementation.

Regarding the *worker* role, this is an implicit role in the model. All the role-shifting threads may be required to participate in a *parallel region* at some point, so the role must be available to them always.

At the start of the project, the free agents were implemented as stand-alone threads [20]. One of the primary efforts of this master thesis will be to adapt them as a role in the model. Regarding other roles, I will adapt the runtime to implement them in the future easily but leave the implementation of them as future work.

A thread can shift its role at different points. When a *parallel region* starts, all the required threads must abandon their current role (if they have any) and execute their assigned *implicit task*. After finishing the *implicit task*, the threads may shift again to one of their potential roles. Figure 3.1 illustrates the role-shifting points for free agents. When a thread is acting as a free agent, it may shift its role before and after executing an *explicit task* but not while executing it.

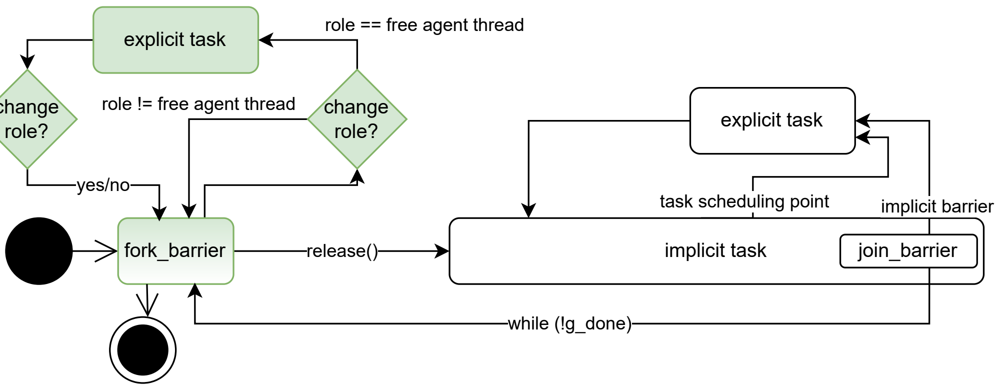


Figure 3.1: OpenMP free agent role flowchart.

Figure 3.2 shows an execution analogous to the one from Figure 2.3 but using the role-shifting model. Here, gray arrows represent a role-shift operation. Only one thread per CPU is active during the entire example with two different roles.

3.2 API

We proposed two different APIs to interact with the model during the design phase. In general terms, the first one operates with a bulk of threads, while the second one operates on individual threads.

To operate on specific individual threads, the user must be able to identify them uniquely, something that OpenMP does not allow now. The OpenMP *thread number* is used to identify a thread inside a *parallel region*, but it is not a global identifier. Therefore, we introduced a global identifier per thread for the whole execution and an API call to obtain it. The threads are numbered from 0 to $n-1$, and the IDs must be consecutive, so if *thread 1* and *thread 3* exist, *thread 2* must also exist.

```
int __kmp_get_thread_id();
```

Listing 3.1: Global thread ID get function

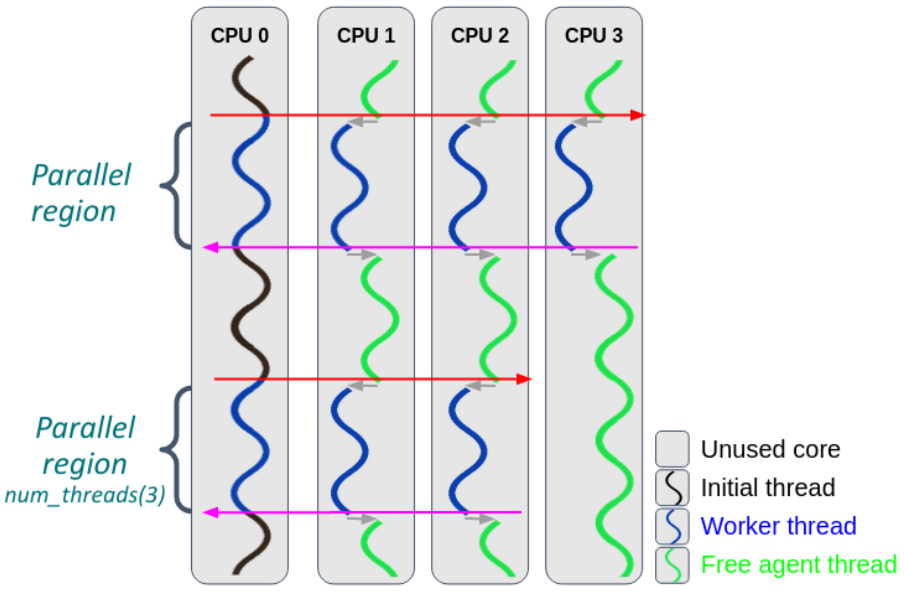


Figure 3.2: Role-shifting execution example.

To allow for multiple potential roles simultaneously, the implementation should use an enumeration where each bit represents a role. The structure must ensure that a value for no role exists.

```
typedef enum omp_role{
    OMP_ROLE_NONE = 0,
    OMP_ROLE_FREE_AGENT = 1 << 0,
    OMP_ROLE_OTHER_ROLE = 1 << 1,
    OMP_ROLE_ANOTHER_ROLE = 1 << 2
} omp_role_t;
```

Listing 3.2: OMP Roles enumeration example

The first API operates on batches of threads. The setter call gives the roles `roles` to the number of threads specified with `how_many` and ensures that the other threads do not have that role. If the user demands more threads than the existing number of threads, the runtime will create them with the proper `roles`. The getter function allows to consult the current number of threads with the potential role `role`.

```
int  __kmp_get_num_threads_role(omp_role_t role);
void __kmp_set_thread_roles(int how_many, omp_role_t roles);
```

Listing 3.3: Batch of threads API calls.

The second API operates on individual threads using the global thread id. The setter function gives the potential roles `roles` to thread `tid` and removes any other role from the thread. If the user wants to create a new thread, it has to use a `tid` greater than the current number of existing threads. The getter function allows to consult the number of roles for thread `tid` and sets `roles` with the potential roles of that thread.

```
int  __kmp_get_thread_roles(int tid, omp_role_t *roles);
void __kmp_set_thread_roles(int tid, omp_role_t roles);
```

Listing 3.4: Individual threads API calls

3.3 Environment variables

When we presented the free agent threads [20], the reviewers and the conference audience criticized the fact that the model had too many environment variables. Therefore, I have tried to simplify this for the role-shifting threads and reduced the amount from six to two.

- `OMP_FREE_AGENT_CLAUSE_DEFAULT`. Indicates if all the tasks may be executed by threads with the free agent role as active. The accepted values are `true` and `false`. Works in conjuction with the task clause introduced in Section 3.4.
- `OMP_ROLES`. A list with the potential roles per thread desired at initialization. Usage example:
  - `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*.

3.4 Task clause

As explained in Section 2.3, free agents are not suited for the execution of tasks with some particularities. When a program has both suitable and unsuitable tasks for free agent execution, using an environment variable that applies to all tasks is not enough. Therefore, tasks must have a specific clause indicating if they are eligible for free agent execution. In Listing 3.5 an example of the clause usage can be found.

```
#pragma omp taskloop free_agent(true)
for(i = 0; i < n; i++){
  //This task may be executed by any
    thread
}
```

```
#pragma omp task free_agent(false)
{
  /*This task must be executed by
    a thread from this team */
  v[omp_get_thread_num()] = ...;
}
```

Listing 3.5: Usage of free agent clause in task and taskloop constructs.

Figure 3.3 summarizes the steps and preferences to decide if a free agent can execute a task. When the `free_agent` clause is present, it takes preference. If there is no clause, then the behaviour defined by the environment variable is used. Finally, if the environment variable is not set, the task cannot be executed by a free agent.

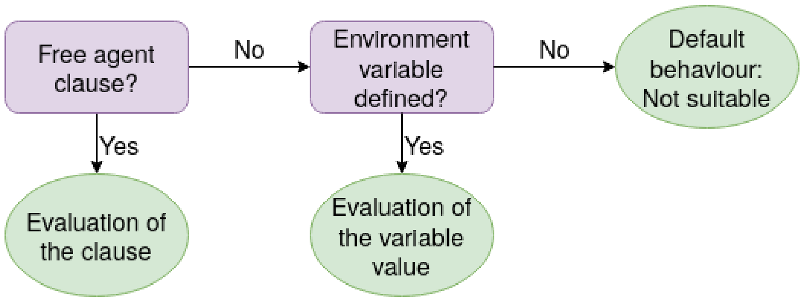

Free agent clause?
No
Environment variable defined?
No
Default behaviour: Not suitable
Yes
Evaluation of the clause
Yes
Evaluation of the variable value

Figure 3.3: Task execution decision for free agents.

3.5 Methodology

The OpenMP runtime and the Clang compiler (and LLVM in general) are used by many users for several different purposes. When performing changes to such elements, one must ensure that all the other features of the program still work, performing tests for both new and old features.

During the development, both the runtime and DLB have been tested regularly to detect any error that may have been introduced. Both code distributions come with a set of tests, which are perfect for that purpose. CI/CD (Continuous Integration, Continuous Deployment) has been used to automatize and ease that part.

GitLab [16] repositories have been used for LLVM [2] and DLB [1]. When pushing to DLB, the server starts a pipeline where the code is compiled, and the tests are run. If any error is detected, the commit is not accepted. However, the LLVM repository is more extensive than DLB, and performing those tests on the server is not viable. For that reason, the proper CI/CD has been done in another server using Jenkins [7].

Figure 3.4 shows a Jenkins pipeline of the LLVM CI/CD. The DLB Jenkins pipeline is the same, so it is not shown for simplicity. The three steps work as follows:

- Make distributable: The code from the repository is packed in a tarball.
- Staging: This step is done in *Mestral*, a server used exclusively by Jenkins to perform builds and tests. The code is extracted from the tarball and then built and installed in the machine. Finally, the OpenMP tests are passed.
- Production: At last, the code is deployed in production clusters. It also extracts, compiles, and installs the code and then performs the tests.

Each node of the process is sequential (but the production nodes are performed in parallel), and when a step fails, the process is aborted, ensuring that no errors are introduced into production machines. After that, the new features introduced are tested in MareNostrum 4 manually because the test suite does not include tests for them. Adding tests into the suite for the new features would be interesting. However, since the amount of work required is considerable, we decided to prioritize performing small tests and focusing on the implementation side, so that remains as future work.

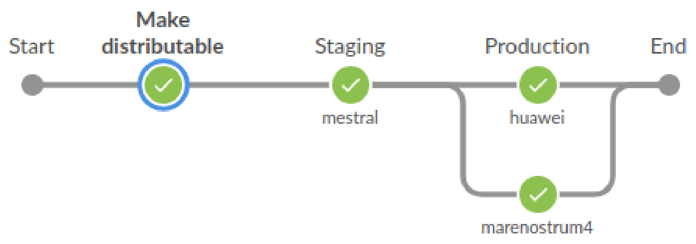


Figure 3.4: CI/CD pipeline.

Chapter 4

Implementation

The role-shifting threads implementation consists of all the necessary structures and algorithms to allow threads to change roles and the integration of free agent threads as a new role. It has been done with the LLVM OpenMP runtime (version 14.0.0), and the code is publicly available online [2].

4.1 Role-shifting

The runtime has a custom struct to maintain all the relevant information of the threads called `kmp_base_info_t`. For role-shifting purposes, I have added the variables shown in Listing 4.1.

```
typedef struct KMP_ALIGN_CACHE kmp_base_info{
    ...//Some variables
    omp_role_t th_potential_roles; //Enum with all the roles this thread can
    have
    std::atomic<omp_role_t> th_active_role; //Current role of this thread
    omp_role_t th_pending_role;
    std::atomic<bool> th_change_role;  //Indicates the thread to change the
    role to th_pending_role ASAP
    ...//Other variables
} kmp_base_info_t;
```

Listing 4.1: Additional variables for role-shifting purposes

The initial thread and the other threads operate in a *producer-consumer* fashion. In each role-shifting point, the primary thread will prepare all the structures and variables needed for the new role and add or remove threads to the free agent list when that role is involved. After that, it sets the `th_pending_role` with the appropriate role the thread must take, and then the atomic variable `th_change_role` to indicate that everything is ready for a role-shift. Listing 4.2 shows a snippet of code from `__kmp_join_call` and a helper routine, where the primary thread transforms team members into free agents.

```
static void __kmp_transform_team_threads_to_FA(int gtid){
    kmp_info_t *new_thr;
    kmp_team_t *team = __kmp_threads[gtid]->th.th_team;
    int i;
    for(i=team_size-1; i>=0 && active_free_agent <= allowed_free_agent; i--){
        new_thr = team->t.t_threads[i];
        //Master thread doesn't change its role
        if(new_thr == this_thr) continue;
        //Only consider threads with the free agent role available
        if(!(new_thr->th.th_potential_roles & OMP_ROLE_FREE_AGENT)) continue;

        ...//Preparation of thread structures for the free agent role

        ...//Placement of the thread into the free agent list

        new_thr->th.th_pending_role = OMP_ROLE_FREE_AGENT;
        KMP_ATOMIC_INC(&__kmp_free_agent_active_nth);
        KMP_ATOMIC_ST_RLX(&new_thr->th.th_change_role, true);
    }
}

void __kmp_join_call(...){
    ...//Some code
    __kmp_transform_team_threads_to_FA(gtid);
    __kmp_free_team(root, team USE_NESTED_HOT_ARG(master_th));
    ...//More code
}
```

Listing 4.2: Transform a member of the team into a free agent.

On the *consumer* side, the threads will be waiting in the fork barrier, on their way to it, or executing tasks if they are free agents. Listing 4.3 shows a snippet of code from the main wait spin loop in the `__kmp_wait_template` function. The threads shift their active role to the one indicated in `th_pending_role` and emit the role-shifting callback.

```
__kmp_lock_suspend_mx(this_thr);
if(this_thr->th.th_change_role){
    omp_role_t prv_role = KMP_ATOMIC_LD_RLX(&this_thr->th.th_active_role);
    omp_role_t nxt_role = this_thr->th.th_pending_role;
    if(prv_role == OMP_ROLE_FREE_AGENT && nxt_role == OMP_ROLE_NONE){
        this_thr->th.th_current_task = this_thr->th.th_next_task;
        this_thr->th.th_next_task = NULL;
    }
    KMP_ATOMIC_ST_RLX(&this_thr->th.th_change_role, false);
    KMP_ATOMIC_ST_RLX(&this_thr->th.th_active_role, nxt_role);
    __kmp_unlock_suspend_mx(this_thr);
#if OMPT_SUPPORT
    ...//OMPT role_shift callback here
#endif
    continue;
}
__kmp_unlock_suspend_mx(this_thr);
```

Listing 4.3: Example of role-shifting operation.

Listing 4.3 also shows how a thread that shifts from free agent to worker sets its own current task. When a free agent is executing an explicit task, the value of `th_current_task` cannot be modified. Therefore, the primary thread of a new parallel region must prepare all the `implicit tasks` of the team and store the address on another variable (shown in Listing 4.4). After that, when the free agent reaches the role-shifting point, it changes the *current task* and can proceed to execute it.

```
void __kmp_push_current_task_to_thread(kmp_info_t *this_thr, kmp_team_t *team,
                                       int tid){
    if(tid == 0){
        //Master thread things.
    }
    else{
        team->t.t_implicit_task_taskdata[tid].td_parent =
            team->t.t_implicit_task_taskdata[0].td_parent;
        if(KMP_ATOMIC_LD_ACQ(&this_thr->th.th_active_role) ==
              OMP_ROLE_FREE_AGENT){
            this_thr->th.th_next_task =
                &team->t.t_implicit_task_taskdata[tid];
        }
        else{
            this_thr->th.th_current_task = &
                team->t.t_implicit_task_taskdata[tid];
        }
    }
}
```

Listing 4.4: Implicit task push to the threads of the team.

4.2 Free agent management

The number of free agents is managed with two global variables (see Listing 4.5). The first one determines the maximum number of threads that can simultaneously have the free agent role. It can be set with an environment variable and the API, with the default value of 0. The second variable indicates the number of active threads with the role at the time. The runtime ensures that it is always between 0 and `__kmp_free_agent_num_threads`.

```
extern int __kmp_free_agent_num_threads; //Max number of FA allowed
extern std::atomic<int> __kmp_free_agent_active_nth; //Actual number of FA
    active
```

Listing 4.5: Number of free agents' global variables.

In the double-pool implementation, free agent threads were created at the runtime initialization and placed in the `root` structure. However, that strategy did not align with the role-shifting idea. Therefore, this had changed in the new implementation. Now, as free agent is just a role, the threads are placed in the `kmp_info_t **__kmp_threads` structure with all the threads in the system.

Free agents are managed with a list. The pointer to the list's beginning is a global variable, and each thread has a variable in its structure pointing to the next element in the list (see Listing 4.6). The runtime keeps the list ordered by global thread id, typically inserting elements at the end of the list, which is pointed by `__kmp_free_agent_list_insert_pt`. On some occasions (e.g., when API calls are involved or there is nested parallelism), a thread may need to be inserted in the middle. In that case, the runtime scans the list from the beginning until finding the proper position.

The threads are removed from the beginning of the list most of the time. When API calls are involved, a thread may be removed from the middle of the list. Therefore, the impact on the performance of insertions and deletions to the list should be low, making the role-shift operation as fast as possible.

```
extern volatile kmp_info_t *__kmp_free_agent_list;
extern kmp_info_t *__kmp_free_agent_list_insert_pt;

typedef struct KMP_ALIGN_CACHE kmp_base_info{
    ...//Some variables
    kmp_info_p *th_next_free_agent;
    ...//Other variables
} kmp_base_info_t;
```

Listing 4.6: Free agent list variables.

Most of the time, the list and the other variables are operated exclusively by the *primary thread*. However, when there is nested parallelism or the user uses the API setter functions, two or more threads may try to read/write the structures concurrently. Therefore, they are protected with a lock (`__kmp_forkjoin_lock`) for reading and writing. This lock existed previously in the runtime and was used to protect certain structures (e.g., `__kmp_threads` and `__kmp_thread_pool`) in the fork/join regions, where several threads may access them simultaneously under nested parallelism. The lock is acquired in the API calls when needed. Since it was already acquired in the other scenarios, only a few changes were required for that.

I also contemplated adding an entirely new lock for those variables. However, I detected a situation when this lock could be crossed with the `__kmp_forkjoin_lock` and cause a deadlock. Also, since the fork-join lock was already acquired for most situations, reusing it for free agent management should help keep the overhead minimum.

4.2.1 Allowed task teams management

The `allowed_teams` (see Listing 4.7) is a per-thread structure indicating the `task teams` from where a free agent can grab tasks. When a `task team` is created, it is added to the list and removed when it ends. In the double-pool implementation, since all the free agents were created from the start and accessible during the entire execution, these lists were always up to date with the alive `task teams`. However, now a thread may not hold the role of free agent when a `task team` is created/removed, or a new free agent may be created in the middle of the run. Therefore, the `allowed_teams` needed some changes to accommodate them to the role-shifting threads model.

```c
typedef struct KMP_ALIGN_CACHE kmp_base_info{
    ...//Some variables
    //List of teams we can enter as a free agent thread
    kmp_bootstrap_lock_t allowed_teams_lock;
    int allowed_teams_capacity;
    int allowed_teams_length;
    kmp_task_team_t** allowed_teams;
    ...//Other variables
} kmp_base_info_t;
```

Listing 4.7: Per thread allowed teams variables.

For that purpose, the runtime now has a global list of allowed teams (see Listing 4.8). The `length` variable indicates the number of elements and the first empty position in the array, and the `capacity` is the number of positions currently allocated in the array. It has a lock to perform any operation on the array, both read and write.

```c
extern kmp_bootstrap_lock_t __kmp_free_agent_allowed_teams_lock;
extern int __kmp_free_agent_allowed_teams_capacity;
extern int __kmp_free_agent_allowed_teams_length;
extern kmp_task_team_t** __kmp_free_agent_allowed_teams;
```

Listing 4.8: Global allowed teams variables.

Listings 4.9 and 4.10 show an example of add and remove operations to the structure. It can be noted that the add/remove operations are also performed on the active free agent threads. When a free agent steals a task, it needs to read the `allowed_teams`, so maintaining the local copy updated prevents the threads from accessing the `__kmp_free_agent_allowed_teams_lock` in a more critical region. The entire functions, including `__kmp_add_global_allowed_task_team` and `__kmp_remove_global_allowed_task_team` can be found in `kmp_tasking.cpp` source file.

```
void __kmp_task_team_setup(kmp_info_t *thr, kmp_team_t *team, int always){
    ...//Some code
    __kmp_acquire_bootstrap_lock(&__kmp_forkjoin_lock);
    __kmp_acquire_bootstrap_lock(&__kmp_free_agent_allowed_teams_lock);
    kmp_task_team_t* task_team = team->t.t_task_team[thr->th.th_task_state];
    __kmp_add_global_allowed_task_team(task_team);
    if(KMP_ATOMIC_LD_RLX(&__kmp_free_agent_active_nth) > 0){
        //Add the task team to the active free agents.
    }
    __kmp_release_bootstrap_lock(&__kmp_free_agent_allowed_teams_lock);
    __kmp_release_bootstrap_lock(&__kmp_forkjoin_lock);
}
```

Listing 4.9: Adding a `task team` to the global structure.

```
void __kmp_task_team_wait(kmp_info_t *thr, kmp_team_t *team, int wait){
    kmp_task_team_t *task_team = team->t.t_task_team[thr->th.th_task_state];
    ...//Some code
    __kmp_acquire_bootstrap_lock(&__kmp_free_agent_allowed_teams_lock);
    __kmp_remove_global_allowed_task_team(task_team);
    if(KMP_ATOMIC_LD_RLX(&__kmp_free_agent_active_nth) > 0){
        //Remove the task team from the active free agents.
    }
    __kmp_release_bootstrap_lock(&__kmp_free_agent_allowed_teams_lock);
}
```

Listing 4.10: Removing a `task team` from the global structure.

The last relevant operation with the global allowed teams is the copy one. When a thread shifts from worker to free agent, the global structure is copied to the thread structure. Listing 4.11 shows an example from the function `transform_thread_to_FA`, which is invoked by the API calls. The same thing occurs when a thread is created with the free agent role. When a shift from free agent to worker happens, the runtime only sets `thr->th.allowed_teams_length` to 0, indicating that the array is empty (not shown in the listing for simplicity).

```
static void transform_thread_to_FA(kmp_info_t *th){
    ...//Some code
    __kmp_acquire_bootstrap_lock(&__kmp_free_agent_allowed_teams_lock);
    if(th->th.allowed_teams == NULL){
        __kmp_init_bootstrap_lock(&th->th.allowed_teams_lock);
        th->th.allowed_teams_capacity =
            (__kmp_free_agent_allowed_teams_length > 0)
            ? __kmp_free_agent_allowed_teams_length*2
            : 4;
        th->th.allowed_teams = (kmp_task_team_t **)__kmp_allocate(sizeof(
    kmp_task-team_t *) * th->th.allowed_teams_capacity);
    }
    else if(th.th_allowed_teams_capacity <=
            __kmp-free_agent_allowed_teams_length){
        __kmp_realloc_thread_allowed_task_team(th,
    __kmp_free_agent_allowed_teams_length*2, FALSE);
    }
    th->th.allowed_teams_length = __kmp_free_agent_allowed_teams_length;
    __kmp_copy_global_allowed_teams_to_thread(th);
    __kmp_release_bootstrap_lock(&__kmp_free_agent_allowed_teams_lock);
    KMP_MB();
}
```

Listing 4.11: Example of a copy of the global allowed teams to a thread when the API is used.

4.2.2 Task creation by free agents

During the execution of a task, a free agent may encounter a `task` or `taskloop` construct. In that situation, the task must be managed as if the free agent was part of the team. When the task is `undeferred` (i.e., it has to be executed by the `encountering thread`), the free agent must execute it immediately after creating it. Otherwise, the task must be pushed to a thread dequeue. The `team` has a dequeue per thread so that each one can push tasks to its dequeue, but free agents must choose one to push the task.

In the double-pool implementation, all free agents pushed the tasks to the `thread 0` of the team, which could lead to a high contention in that dequeue and provoke a significant overhead. To avoid that, I opted to push the tasks to all the different threads in the team. When a free agent thread *steals* a task from a thread (see Listing 4.12), it stores the thread identifier in its own structure. Later, when pushing the task, it will read that identifier and push the task to that dequeue (see Listing 4.13).

```
typedef struct KMP_ALIGN_CACHE kmp_base_info{
    ...//Some variables
    int victim_tid; //Id of the owner thread of the last stolen task
} kmp_base_info_t;

static inline int __kmp_execute_tasks_template(...){
    kmp_task_t *task;
    ...//Some code
    while(1){
        while(1){
            handleServices();
            task = NULL;
            if(use_own_tasks){
                if(thread->th.th_active_role == OMP_ROLE_FREE_AGENT){
                    if(KMP_ATOMIC_LD_ACQ(&thread->th.th_change_role))
                        return false;
                    task = __kmp_steal_task(threads_data[tid].td.td_thr, ...);
                    if(task) thread->th.victim_tid = tid;
                }
            }
            ...//Loop body
        }
        ...//Loop body
    }
}
```

Listing 4.12: Stealing of tasks by a free agent in `__kmp_execute_tasks_template`.

```
static kmp_int32 __kmp_push_task(kmp_int32 gtid, kmp_task_t *task){
    ...//Some code
    kmp_int32 tid;
    if(thread->th.th_active_role == OMP_ROLE_FREE_AGENT)
        tid = thread->th.victim_tid;
    else tid = __kmp_tid_from_gtid(gtid);
    ...//Some code
    //Find tasking deque specific to encountering thread
    thread_data = &task_team->tt.tt_threads_data[tid];
    ...//Rest of the function
}
```

Listing 4.13: Pushing of tasks by a free agent in `__kmp_push_task`.

4.3 Thread creation at initialization

All previous versions of the LLVM runtime created all the threads on demand when encountering a new parallel region. The first time, it created all the necessary threads for the region. After that, the threads were placed in the `__kmp_thread_pool` structure. In subsequent parallel regions, the runtime always reused the threads from the pool and only created new ones when the region required more threads than the existing ones.

One of free agents' many advantages is the OpenMP syntax's simplification. Listings 4.14 and 4.15 show an example of this. On the left is a code with the classical OpenMP structure, indicating the parallelism with a `parallel` construct, the taskification of a loop with `taskloop`, and the `single` construct to ensure that the tasks are created only once. The role-shifting threads and the role of free agent allow dropping the usage of the `parallel` and `single` constructs as shown on the right side, simplifying the syntax.

```
int main(int argc, char *argv[]){
    ...//Some code
    #pragma omp parallel
    #pragma omp single
    {
        #pragma omp taskloop
        for(i = 0; i < n; i++){
            ...//Loop body
        }
    }
    ...//Some code
}
```

Listing 4.14: Example of a functional code with a parallel construct.

```
int main(int argc, char *argv[]){
    ...//Some code
    #pragma omp taskloop
    for(i = 0; i < n; i++){
        ...//Loop body
    }
    ...//Some code
}
```

Listing 4.15: Example of a functional code without a parallel construct.

To support that, the creation of the threads must be moved to the runtime initialization instead of doing it only when encountering a `parallel` construct. Now, the first thread creation point is moved to the initialization of the runtime, called *middle_init* in the LLVM runtime. At that point, the runtime reads the environment variables `OMP_NUM_THREADS` and `KMP_NUM_FREE_AGENTS` and creates the indicated amount of threads. When the first one is strictly greater, the threads without a role are placed in the thread pool. Otherwise, all the threads start with the free agent role as their active role. Listing 4.16 shows the part of the code where threads are created.

```
int num_workers = __kmp_dflt_team_nth - __kmp_free_agent_num_threads;
int num_fa = (num_workers < 0) ?
    __kmp_free_agent_num_threads : __kmp_dflt_team_nth;
kmp_root_t *root = __kmp_threads[0]->th.th_root;
for(i = 1; i < num_workers; i++){
    __kmp_allocate_thread_middle_init(root, OMP_ROLE_NONE, i);
}
for(; i < num_fa; i++){
    __kmp_alllocate_thread_middle_init(root, OMP_ROLE_FREE_AGENT, i);
}
```

Listing 4.16: Initial thread creation in `__kmp_do_middle_initialize()`.

4.4 Global thread ID

As explained in Section 3.2, the OpenMP runtime now has the concept of global thread id to uniquely identify each thread, which is used by the API calls. The identifier is placed in the `kmp_desc_base_t struct` (see Listing 4.17), which can be accessed from the thread's `th_info` variable.

```
typedef struct kmp_desc_base{
    ...//Some variables
    int ds_gtid;
    int ds_thread_id; //Global thread id from 0 to n-1.
    ...//More variables
} kmp_desc_base_t;
```

Listing 4.17: Global thread identifier declaration.

In the same structure, there is also an integer called `gtid`, uniquely identifying a thread in the runtime and corresponding to the thread's position in the `__kmp_threads` array. When the Hidden Helper Threads were added to the runtime, the developers decided to give them `gtids` from 1 to n. Therefore, since all the *standard* threads must have consecutive identifiers, this new variable is added. The initialization is done in the `__kmp_create_worker()` routine, as shown in Listing 4.18.

```
void __kmp_create_worker(int gtid, kmp_info_t *th, size_t stack_size){
   ...//Some code
   th->th.th_info.ds.ds_gtid = gtid;
   if(gtid != 0){
      th->th.th_info.ds.ds_thread_id = TCR_4(__kmp_init_hidden_helper_threads)
                              ? gtid
                              : gtid - __kmp_hhiden_helper_threads_num;
   }
}
```

Listing 4.18: Global thread identifier initialization.

4.5 OMPT Callback

I have added a new OMPT callback to the runtime to notify when a thread performs a role-shift operation. Listing 4.19 displays the callback signature. Any first-party tool (e.g., DLB) can register a routine with this signature, and the runtime will invoke it every time the event happens. Listing 4.20 shows how the runtime invokes the callback from the `kmp_wait_release.h` source file.

```
typedef void (*ompt_callback_thread_role_shift_t) (
    ompt_data_t *thread_data,
    ompt_role_t prior_role,
    ompt_role_t next_role
);
```

Listing 4.19: OMPT role-shift callback signature.

```
#if OMPT_SUPPORT
    ompt_data_t *thread_data = nullptr;
    if(ompt_enabled.enabled){
        thread_data = &(this_thr->th.ompt_thread_info.thread_data);
        if(ompt_enabled.ompt_callback_thread_role_shift){
            ompt_callbacks.ompt_callback(ompt_callback_thread_role_shift)(
                thread_data, (ompt_role_t)prv_role, (ompt_role_t)nxt_role);
        }
    }
#endif
```

Listing 4.20: OMPT role-shift callback invocation from the runtime.

4.6 API

4.6.1 Individual threads

This API operates on a specific OpenMP thread using its global thread identifier. The function headers are displayed in Listing 4.21; their code can be found in the `kmp_runtime.cpp` source file. The get function obtains the number of potential roles of thread `tid` and the value of the potential roles in `r`.

```
/* Returns the number of roles of the thread with thread_id==tid,
   r holds the actual potential roles of the thread. */
int __kmp_get_thread_roles(int tid, omp_role_t *r)
/* Gives the roles r to the thread with thread_id==tid.
   It overrides the previous roles of the thread. */
void __kmp_set_thread_roles(int tid, omp_role_t r)
/* Returns the (global) thread id of the calling thread.
   Doesn't correspond to the gtid of the runtime. */
int __kmp_get_thread_id();
```

Listing 4.21: Individual threads API calls.

Regarding the setter, the user can query for several roles at the time. The thread `tid` will get all of them as potential roles and lose any other potential role that had previously. When the active role is removed from the potential roles, the thread is signaled to change to `OMP_ROLE_NONE` and placed in the thread pool. With this mechanism, a user can stop a free agent when desired. When the `tid` used is greater than the number of threads created (i.e., any thread has that ID), the function will create a new thread with the potential roles specified. The `thread id` assigned to that thread will be the next consecutive identifier (as explained in Section 4.4), independently of the value given to `tid`.

Finally, the `__kmp_get_thread_id()` routine can be used to obtain the identifier of the calling thread.

4.6.2 Bulk of threads

This API operates on a group of threads. The function headers are displayed in Listing 4.22; their code can be found in the `kmp_runtime.cpp` source file. The get function obtains the number of threads with the potential role `r`. The API expects a single role in `r`, not a combination of different roles. If the user desires to query for more than one role, it must make separate calls to the API.

```
/* Returns the number of threads with the role r. */
int __kmp_get_num_threads_role(omp_role_t r)
/* Gives the roles r to how_many threads.
   It overrides the previous roles of the thread. */
void __kmp_set_thread_roles(int how_many, omp_role_t r)
```

Listing 4.22: Bulk of threads API calls.

In contrast, the user can query for several roles in the setter function and give the potential roles to exactly `how_many` threads. When the `how_many` value is less than the number of existing threads, the threads selected are those with `thread_id` from $n - 1 - how\_many$ to $n - 1$, where n is the current number of threads. If the number is greater, the function creates as many threads as necessary (if the `thread-limit-var` allows it) and gives the potential roles `r` to all of them. If the threads affected had any previous role, they will be removed. When the active role is removed from the potential roles, the threads are signaled to change to `OMP_ROLE_NONE` and placed in the thread pool.

4.7 Environment variables

As explained in Section 3.3, two new environment variables have been introduced in the model. The first one is `KMP_FREE_AGENT_CLAUSE_DEFAULT`, which is analogous to `OMP_FREE_AGENT_CLAUSE_DEFAULT` and accepts a boolean value. The second one is `KMP_FREE_AGENT_NUM_THREADS`, which serves as a replacement for `OMP_ROLES` for the moment. Since there is only one valid role in the model, I decided to simplify this variable for the time being. This variable accepts any positive integer or zero.

```
static kmp_setting_t __kmp_stg_table[] = {
   ...//Parsing of env variables
   //Free agent threads
   {"KMP_FREE_AGENT_NUM_THREADS", __kmp_stg_parse_num_free_agent_threads,
      __kmp_stg_print_num_free_agent_threads, NULL, 0, 0},
   {"KMP_FREE_AGENT_CLAUSE_DEFAULT", __kmp_stg_parse_free_agent_clause_default,
      __kmp_stg_print_free_agent_clause_default, NULL, 0, 0},

{"", NULL, NULL, NULL, 0, 0}}; // settings
```

Listing 4.23: Parsing of the environment variables related to free agents.

Listing 4.23 shows a snippet of code from `kmp_settings.cpp` where the two environment variables are parsed and stored into the global runtime variables of Listing 4.24. This code is called at runtime initialization.

```
extern int __kmp_free_agent_num_threads; /*Max number of free agents allowed.
Initialized with env variable KMP_FREE_AGENT_NUM_THREADS */
extern int __kmp_free_agent_clause_dflt; /* Value obtained fom env variable
KMP_FREE_AGENT_CLAUSE_DEFAULT */
```

Listing 4.24: Global variables obtained from environment variables.

4.8 Task clause

The implementation of the `free_agent` clause is divided into two parts. First, the compiler must interpret it correctly and create appropriate structures that the runtime can understand. Then, the runtime must check each time a free agent executes a task if it is allowed or not.

4.8.1 Runtime

Each task in the runtime has the `kmp_tasking_flags_t td_flags` variable, where it stores the value of its flags; the struct is displayed in Listing 4.25. This structure must have 32 bits exactly, with 16 bits for the compiler and 16 for libraries. At the start of the project, only eight compiler bits were used, so there were enough bits for the `free_agent` clause.

```
typedef enum fa_clause{
    FREE_AGENT_CLAUSE_UNSET = 0,
    FREE_AGENT_CLAUSE_FALSE = 1,
    FREE_AGENT_CLAUSE_TRUE = 2,
} fa_clause_t;

typedef struct kmp_tasking_flags{ //Total struct mus be exactly 32 bits
    /* Compiler flags */ /* Total compiler flags must be 16 bits */
    ...//Some flags
    unsigned free_agent : 2; /* 0 == unspecified in the task construct
                              * 1 == cannot be executed by free agent
                              * 2 == can be executed by free agent
                              * 3 == not used currently */
    unsigned reserved : 6; //Reserved for compiler use */

    /*Library flags */ /* Total library flags must be exactly 16 bits */
    ...//Library flags
} kmp_tasking_flags_t;
```

Listing 4.25: Tasking flags structure.

The compiler is in charge of setting the flag with the appropriate value, indicating if the clause was present or not and its value (if it was), and the runtime only has to consult it. When the threads grab tasks to execute, they call the function `__kmp_task_is_allowed` to know if they can execute that tasks. Listing 4.26 shows the part of the function where this is checked. Note that when the clause is not present in the `task` construct, the value of the environment variable is checked (`__kmp_free_agent_clause_dflt`).

```
static bool __kmp_task_is_allowed(int gtid, ...){
    //Tied tasks check
    if(){
        ...//
    }
    //When the thread is free agent, the task must have the fa clause.
    if(__kmp_threads[gtid]->th.th_active_role == OMP_ROLE_FREE_AGENT){
        if((task->td_flags.free_agent == FREE_AGENT_CLAUSE_FALSE) ||
           (task->td_flags.free_agent == FREE_AGENT_CLAUSE UNSET &&
            !__kmp_free_agent_clause_dflt))
           return false;
    }
    //Check for mutexinoutset dependencies
    if(){
        ...//
    }
    return true;
}
```

Listing 4.26: Free agent clause value check at runtime.

4.8.2 Clang

The clang compiler must evaluate the boolean in the `free_agent` clause when present or detect that the clause was not used, and set the values according to the enum in Listing 4.25. This must be done for both `task` and `taskloop` constructs.

The first step is to create a class in clang for the clause. The `OMPFreeAgentClause` can be found in the *clang/include/clang/AST/OpenMPClause.h*. The functions and variables of the class will be used in all the steps involved in the compilation of the clause. Then, the

OpenMP tablegen (*llvm/include/llvm/Frontend/OpenMP/OMP.td*) has been modified to let LLVM know which class is in charge of the clause and which constructs use the clause. Listing 4.27 shows the relevant part of the tablegen (the `task` construct is not shown since it is similar to `taskloop`).

```
def OMPC_FreeAgent : Clause<"free_agent"> {
    let clangClass = "OMPFreeAgentClause";
    let flangClass = "ScalarLogicalExpr";
}
def OMP_TaskLoop : Directive<"taskloop">{
    let allowedClauses = [
        ...//Some allowed clauses
    ];
    let allowedOnceClauses = [
        ...//Some allowed only once clauses
        VersionedClause<OMPC_Priority>,
        VersionedClause<OMPC_FreeAgent>
    ];
    let allowedExclusiveClauses = [
        ...//GrainSize and NumTasks clauses
    ];
}
```

Listing 4.27: Free agent clause in the LLVM OpenMP tablegen.

The next step is to make the Parser understand the clause. The `free_agent` clause can only appear once and must have a unique expression inside, the same as the `final` clause. Since Clang was already parsing this type of clauses, only one new `case` was necessary, as shown in Listing 4.28.

```
OMPClause *Parser::ParseOpenMPClause(...){
    ...//Some code
    switch (CKind){
    case OMPC_final:
    case OMPC_free_agent:
    case ...//More clauses
        ...//Parsing of single value clauses that only appear once
    ...//Rest of the clauses
    }
}
```

Listing 4.28: Free agent clause parsing.

Once the Parser is handled, it is time to modify the Semantic Analyzer (*clang/lib/Sema/SemaOpenMP.cpp*). First, Sema needs a function to call on well-formed clauses. The function declaration is shown in Listing 4.29. The routine code can be found in the source file and is copied from the `final` clause since their behavior must be the same. In the end, it generates an `OMPFreeAgentClause` object.

```
/// Called on well-formed 'free-agent' clause
OMPClause *ActOnOpenMPFreeAgentClause(Expr *FreeAgent, SourceLocation StartLoc,
                                      SourceLocation LParenLoc,
                                      SourceLocation EndLoc);
```

Listing 4.29: Free agent clause Semantic Analyzer `ActOnOpenMPFreeAgentClause` routine.

The Sema has different functions for each kind of directive. The next step is to specify what should happen when encountering the `free_agent` clause in that directives. For the directives that accept the clause, the `ActOnOpenMPFreeAgentClause` routine is called (Listing 4.31), and an error message is produced for the others (Listing 4.31).

```
OMPClause *Sema::ActOnOpenMPSingleExprClause(OpenMPClauseKind Kind, ...){
    OMPClause *Res = nullptr;
    switch (Kind) {
    case OMPC_final:
        Res = ActOnOpenMPFinalClause(Expr, StartLoc, LParenLoc, EndLoc);
        break;
    ...//Some valid cases
    case OMPC_free_agent:
        Res = ActOnOpenMPFreeAgentClause(Expr, StartLoc, LParenLoc, EndLoc);
        break;
    ...//Some not valid cases
    default:
        llvm_unreachable("Clause is not allowed");
    }
    return Res;
}
```

Listing 4.30: Free agent clause allowed example in Sema.

```
OMPClause *Sema::ActOnOpenMPClause(OpenMPClauseKind Kind, ...){
    switch (Kind) {
        ...//Some used cases and their actions
        case OMPC_if:
        case OMPC_final:
        ...//More similar cases
        case OMPC_free_agent:
        default:
        llvm_unreachable("Clause is not allowed");
    }
    return Res;
}
```

Listing 4.31: Free agent clause not allowed example in Sema.

The last step is to modify the CodeGen section of Clang to generate the tasks with the flag properly. First, CodeGen uses the `OMPTaskDataTy` struct to store the values of task construct clauses. Listing 4.32 shows the addition to that struct to accommodate the `free_agent` clause.

```
struct OMPTaskDataTy final {
    ...//Some task variables
    llvm::Value *FreeAgent = nullptr;
    ...//More variables
};
```

Listing 4.32: Free agent clause OMPTaskDataTy struct from CodeGen.

The next relevant part of CodeGen is the one in charge of evaluating the clauses: `EmitOMPTaskBasedDirective`. Here, it gets the `OMPFreeAgentClause` from the Sema step and evaluates the condition. The evaluation has two modes: when the expression is constant, the value is evaluated in the front-end; otherwise (e.g., the value of the expression depends on a runtime variable), it emits code to compute the value of the boolean expression at runtime. The values 0x200 and 0x100 are used to set the expected bits from the runtime structure (see Listing 4.25). No bit is set when the clause is not present. Since Clang does not set any bit and the runtime initializes the `free_agent` bits to 0, the effect of *unspecified in the task construct* is achieved.

```
void CodeGenFunction::EmitOMPTaskBasedDirective(const OMPExecutableDirective
        &S, const OpenMPDirectiveKind CapturedRegion,
        const RegionCodegenTy &BodyGen,
        const TaskFGenTy &TaskGen, OMPTaskDataTy &Data){
    ...//Check for other clauses
    //Check if the task has 'free_agent' clause.
    if(const auto *Clause = S.getSingleClause<OMPFreeAgentClause>()){
        enum {FREE_AGENT_CLAUSE_TRUE = 0x200, FREE_AGENT_CLAUSE = 0x100};
        const Expr *FreeAgent = Clause->getFreeAgent();
        bool CondConstant;
        if(ConstantFoldsToSimpleInteger(FreeAgent, CondConstant)){
            Data.FreeAgent = CondConstant
                            ? Builder.getInt32(FREE_AGENT_CLAUSE_TRUE)
                            : Builder.getInt32(FREE_AGENT_CLAUSE_FALSE);
        } else {
            llvm::Value *B EvaluateExprAsBool(FreeAgent)l
            Data.FreeAgent = Builder.CreateSelect(
                B, Builder.getInt32(FREE_AGENT_CLAUSE_TRUE),
                Builder.getInt32(FREE_AGENT_CLAUSE_FALSE));
        }
    }
    ...//Other clauses checks
}
```

Listing 4.33: Evaluation of the `free_agent` clause in CodeGen.

Finally, the function `emitTaskInit` generates the task. Here, the tasking flags are added one by one into the `TaskFlags` variable, which serves to set the `kmp_tasking_flags_t` on the runtime side. Since the `Data.FreeAgent` variable has the correct value (either 0x200, 0x100, or no value represented by `nullptr`) from `EmitOMPTaskBasedDirective`, CodeGen only needs to perform an or operation of that variable with `TaskFlags`. The relevant code is displayed in Listing 4.34.

```
CGOpenMPRuntime::emitTaskInit(CodeGenFunction &CGF, SourceLocation Loc,
                              const OMPExecutableDirective &D,
                              llvm::Function *TaskFunction, QualType SharedsTy,
                              Address Shareds, const OMPTaslDataTy &Data){
    ...// Generation of some task variables
    llvm::Value *TaskFlags =
        Data.Final.getPointer()
            ? CGF.Builder.CreateSelect(Data.Final.getPointer(),
                                       CGF.Builder.getInt32(FinalFlag),
                                       CGF.Builder.getInt32(/*C=*/0))
            : CGF.Builder.getInt32(Data.Final.getInt() ? FinalFlag : 0);
    TaskFlags = CGF.Builder.CreateOr(TaskFlags, CGF.Builder.getInt32(Flags));
    if(Data.FreeAgent) {
        TaskFlags = CGF.Builder.CreateOr(TaskFlags, Data.FreeAgent);
    }
    ...//Generation of the other tasking flags and the rest of the variables.
}
```

Listing 4.34: Construction of a `task` in CodeGen.

Chapter 5

DLB Integration

This chapter covers the integration of the OpenMP role-shifting threads with the LeWI (see Section 2.4.1) module of DLB. The source code is publicly available online [1].

LeWI is designed to perform load balance operations of applications with both distributed and shared memory programming models. MPI is the programming model of choice for distributed memory in the HPC community, and DLB is totally integrated with it.

Regarding the shared memory programming models, DLB is capable of working with OmpSs and OpenMP. Before the addition of OMPT, the user had to modify its application with some calls to the DLB API; otherwise, LeWI could not act. The addition of the OMPT interface offered the possibility to capture the use of OpenMP constructs in the user code and perform the balancing of resources transparently. It has been integrated with DLB since its release in OpenMP 5.0.

The role-shifting threads have been included in LeWI as a new OMPT thread manager. With that, DLB has three different OpenMP thread managers: vanilla OMPT, the double-pool implementation of free agents, and the role-shifting threads. The user can select which to use with the `DLB_ARGS` environment variables and the flag `--ompt-thread -manager`.

5.1 LeWI general concepts

LeWI (see Section 2.4.1) is a resource manager aiming to optimize applications' load balance. LeWI is in charge of managing the CPUs inside a node with shared memory. It relies on a shared memory programming model (e.g., OpenMP) to manage the threads and has several mechanisms (e.g., APIs, callbacks) to interact with the runtime and take profit from the threads.

The available CPUs are owned uniquely by a single process with all the rights over them. It can keep the CPUs when needed and claim them back whenever necessary. The owner does not change during the execution. When the owner is not using the CPU, other processes may use it.

Figure 5.1 explains the states a CPU has from the DLB point of view. It also displays the actions that the processes can perform to interact with DLB and the CPUs.

When the execution starts, all the CPUs transition from *Disable* (i.e., not in the system) to *Busy*. While the owner uses them, they will remain in that state. At some point, the process may lend a CPU and become *Idle*. When a CPU is not in use, other processes can acquire it, and the CPU becomes *Borrowed*. A borrowed CPU can be released by the process that acquired it and transition to *Idle*; in addition, the owner can claim it back, and the CPU goes to *Claimed*. When a CPU is *Claimed*, the process that borrowed it must return it as soon as possible, and the CPU becomes *Busy* again.

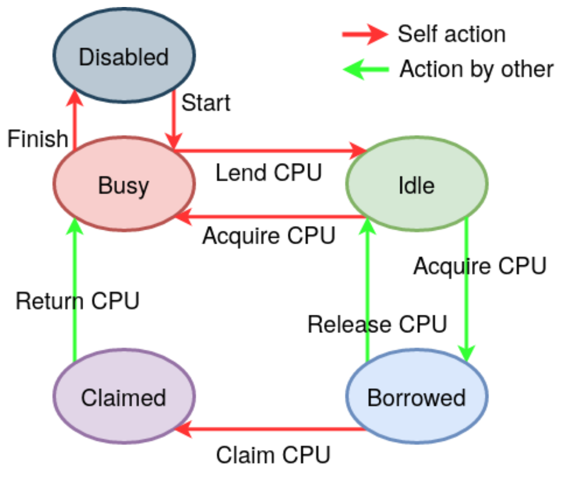


Figure 5.1: DLB CPU states and actions.

5.2 Thread manager implementation

When the role-shift manager is selected, LeWI calls the init function `omptm_role_shift__init`. Here, all the global variables are properly initialized and allocated if needed. The first part of the function is shown in Listing 5.1. Here, the node size where the process runs is consulted, along with the CPU affinity mask of the process. It also reads the values from the environment variables `OMP_NUM_THREADS` and `KMP_NUM_FREE_AGENT_THREADS`. Finally, it detects if the LeWI module is active.

```
static int system_size;
static bool lewi = false;
static atomic_int num_free_agents = 0;
static cpu_set_t process_mask;
static int default_num_threads.

void omptm_role_shift__init(pid_t process_id, const options_t *options){
    /*Initialize static variables*/
    system_size = mu_get_system_size();
    lewi = options->lewi;
    ...//Some variables
    num_free_agents = __kmp_get_num_threads_role(OMP_ROLE_FREE_AGENT);
    shmem_procinfo_getprocessmask(pid, &process_mask, 0);
    //omp_get_max_threads cannot be called here, try using the env. var.
    const char *env_omp_num_threads = getenv("OMP_NUM_THREADS");
    default_num_threads = env_omp_num_threads
                        ? atoi(env_omp_num_threads);
                        : CPU_COUNT(&process_mask);
}
```

Listing 5.1: Initialization of global variables.

The initialization continues in Listing 5.2. The `cpu_data` and `cpu_by_id` are allocated and the second one is initialized to -1. The former holds the status of all the system cores and if a thread with the free agent role uses that CPU, while the latter maps each CPU to the specific thread running in it. In addition, the mask of the primary thread is initialized to zero, and the initial number of threads is calculated.

```c
typedef enum CPUStatus {
    OWN      = 0
    UNKNOWN  = 1 << 0,
    LENT     = 1 << 1,
    BORROWED = 1 << 2
} cpu_status_t
typedef struct CPU_Data{
    cpu_status_t ownership;
    bool         fa;
} cpu_data_t;
static cpu_data_t *cpu_data = NULL;
static int *cpu_by_id = NULL;
static cpu_set_t primary_thread_mask;
static atomic_int registered_threads = 0;

void omptm_role_shift__init(pid_t process_id, const options_t *options){
    ...//
    cpu_data = malloc(sizeof(cpu_data_t)*system_size);
    cpu_by_id = malloc(sizeof(int)*system_size);
    CPU_ZERO(&primary_thread_mask);
    registered_threads = (default_num_threads > num_free_agents)
                       ? default_num_threads
                       : num_free_agents;
    int i;
    for(i = 0; i < system_size; i++) cpu_by_id[i] = -1;
}
```

Listing 5.2: Allocation of global structures for CPU management.

Following that, the `cpu_data` array is initialized. The CPUs that appear in the `process_mask` are marked as `OWN`, while the others are `UNKNOWN`. It also assigns CPUs to the threads that are created at initialization. Finally, the `active_mask` is initialized with the CPU of the primary thread since it is the only one active at the moment.

```c
static int primary_thread_cpu;
static cpu_set_t active_mask;
void omptm_role_shift__init(pid_t process_id, const options_t *options){
    ...//
    int encountered_cpus = 0;
    for(i = 0; i < system_size; i++){
        if(CPU_ISSET(i, &process_mask)){
            if(++encountered_cpus == 1){
                primary_thread_cpu = i;
                CPU_SET(i, &primary_thread_mask);
                cpu_by_id[encountered_cpus - 1] = i;
            } else if(encountered_cpus <= default_num_threads){
                cpu_by_id[encountered_cpus - 1] = i;
            }
            cpu_data[i].ownership = OWN;
        } else {
            cpu_data[i].ownership = UNKNOWN;
        }
        cpu_data[i].fa = false;
    }
    memcpy(&active_mask, &primary_thread_mask, sizeof(cpu_set_t));
}
```

Listing 5.3: Initialization of `cpu_data` and assignation of CPUs to threads.

The last step done at initialization is setting the DLB callbacks. This are functions called by LeWI when a CPU is given (`cb_enable_cpu`) or removed (`cb_disable_cpu`) from the process. An example for the first callback is given in Listing 5.4.

```
void omptm_role_shift__init(pid_t process_id, const options_t *options){
    ...//
    if(lewi){
        int err;
        err = DLB_CallbackSet(dlb_callback_enable_cpu,
                              (dlb_callback_t)cb_enable_cpu, NULL);
        if(err != DLB_SUCCESS){
            warning("DLB_CallbackSet enable_cpu: %s", DLB_Strerror(err));
        }
        ...//Two more callbacks setting
    }
}
```

Listing 5.4: LeWI `cb_enable_cpu` callback setting.

The thread manager has three different parts that work together: the DLB callbacks to activate/deactivate cores, the DLB callbacks for blocking functions (e.g., `MPI_Barrier`, `MPI_Allgather`), and the OMPT callbacks.

5.2.1 Enable and disable CPUs

The thread manager has two callbacks to enable and disable the CPUs of the process. The callbacks may be called from the thread manager itself or triggered by DLB when a certain action occurs (e.g., the process receives a CPU from another rank). Listing 5.5 has the `cb_disable_cpu` code. The `cpuid` argument identifies the logical CPU to deactivate. First, the status of the CPU is updated. After that, if a free agent is using the CPU, the thread manager removes the role from it. That way, the thread will probably go to sleep and leave the CPU entirely for the process that acquires it.

```
static void cb_disable_cpu(int cpuid, void *arg){
    if(cpu_data[cpuid].ownership == BORROWED)
        cpu_data[cpuid].ownership = UNKNOWN;
    else if(cpu_data[cpuid].ownership = OWN)
        cpu_data[cpuid].ownership = LENT;
    if(cpu_data[cpuid].fa){
        int tid = get_id_from_cpuid(cpuid);
        if(tid >= 0){
            ATOMIC_DEC(&num_free_agents);
            __kmp_set_thread_roles(tid, OMP_ROLE_NONE);
        }
    }
}
```

Listing 5.5: Disable cpu.

The `cb_enable_cpu` (see Listing 5.6) starts also updating the CPU's state. Then, it gets the global id of the thread that was running previously on that CPU. If the value is -1, it means that it is the first time that this process has received this CPU and that any thread was running there before. In that case, it uses the API to create a new thread with the free agent role and assigns the CPU to the thread. A thread used the CPU previously when the value is 0 or greater. The thread manager wakes it up using the runtime API if it was a free agent.

```
static void cb_enable_cpu(int cpuid, void *arg){
    if(cpu_data[cpuid].ownership == LENT){
        cpu_data[cpuid].ownership = OWN;
        return;
    }
    else if(cpu_data[cpuid].ownership == UNKNOWN)
        cpu_data[cpuid].ownership = BORROWED;
    int pos = get_id_from_cpu(cpuid);
    if(pos >= 0){//A thread was running here previously
        if(cpu_data[cpuid].fa){
            ATOMIC_INC(&num_free_agents);
            __kmp_set_thread_roles(pos, OMP_ROLE_FRE_AGENT);
        }
    }
    else if(pos == -1){
        cpu_data[cpuid].fa = false;
        ATOMIC_INC(&num_free_agents);
        __kmp_set_thread_roles(system_size, OMP_ROLE_FREE_AGENT);
        cpu_by_id[ATOMIC_INC(&registered_threads)] = cpuid;
    }
}
```

Listing 5.6: Enable cpu.

5.2.2 MPI interception

The thread manager performs actions when going in and out of an MPI blocking call. When entering, the CPUs from the process are lent to DLB so that other processes may acquire them. Listing 5.7 shows the relevant part of the `IntoBlockingCall` function. The status of the CPUs is consulted and updated to `LENT` or `UNKNOWN`, depending on who is the original owner. Finally, all the CPUs are lent with the `DLB_LendCpuMask` call.

```
void omptm_role_shift__IntoBlockingCall(void){
    if(lewi){
        cpu_set_t cpus_to_lend;
        CPU_ZERO(&cpus_to_lend);
        int i;
        for(i = 0; i < system_size; i++){
            if(cpu_data[i].ownership == OWN){
                cpu_data[i].ownership == LENT;
                CPU_SET(i, &cpus_to_lend);
            }
            else if(cpu_data[i].ownership == BORROWED &&
                    CPU_ISSET(i, &process_mask)){
                cpu_data[i].ownership == UNKNOWN;
                CPU_SET(i, &cpus_to_lend);
            }
        }
        DLB_LendCpuMask(&cpus_to_lend);
    }
}
```

Listing 5.7: LeWI `IntoBlockingCall` function.

After finalizing the MPI call, each process has to get back its own CPUs. The `OutOfBlockingCall` code is shown in Listing 5.8. First, it updates the status of the primary thread and then reclaims to DLB all its own CPUs. The `DLB_Reclaim` function will end up calling `cb_enable_cpu` for each CPU, so the state is properly updated.

```c
void omptm_role_shift__OutOfBlockingCall(void){
    if(lewi){
        cb_enable_cpu(cpu_by_id[global_tid], NULL);
        DLB_Reclaim();
    }
}
```

Listing 5.8: LeWI `OutOfBlockingCall` function.

5.2.3 OMPT Callbacks

The OMPT callbacks are DLB functions called by the OpenMP runtime each time a specific action occurs, depending on the callback's type. Listing 5.9 shows the code for the `thread_begin` callback, called each time a thread is created. The `thread_type` argument indicates the type of thread created, and a value of `ompt_thread_other` indicates that the thread has been created with the active role of free agent.

The first action that the thread performs is to register its global thread id, independently of the type of thread. Then, those that started as free agents from the `cb_enable_cpu` are bound to their assigned CPU. They will only run in this CPU for the rest of the execution. After that, if the CPU has been reclaimed, it is returned to DLB. Finally, if the pending tasks have been exhausted, the CPU is disabled and lent to DLB so that other processes may use it.

```c
void omptm_role_shift__thread_begin(ompt_thread_t thread_type,
                                    ompt_data_t *thread_data){
    global_tid = __kmp_get_thread_id();
    int cpuid = cpu_by_id[global_tid];
    if(thread_type == ompt_thread_other && cpu_data[cpuid].ownership != OWN){
        cpu_set_t thread_mask;
        cpu_data[cpuid].fa = true;
        CPU_ZERO(&thread_mask);
        CPU_SET(cpuid, &thread_mask);
        pthread_setaffinity_np(pthread_self(),sizeof(cpu_set_t),&thread_mask);
        if(DLB_CheckCpuAvailability(cpuid) == DLB_ERR_PERM){
            if(DLB_ReturnCpu(cpuid) == DLB_ERR_PERM){
                cb_disable_cpu(cpuid, NULL);
            }
        }
        else if(ATOMIC_LD(&pending_tasks) == 0){
            cb_disable_cpu(cpuid, NULL);
            if(!CPU_ISSET(cpuid, &process_mask))
                DLB_LendCpu(cpuid);
        }
    }
}
```

Listing 5.9: Thread begin callback.

The `thread_begin` callback is the only function where the *pthreads* are given a new CPU affinity mask with a unique CPU. Therefore, the thread will only be reused if that CPU is received again. Otherwise, a new thread is created. Initially, the thread manager dynamically changed the CPU affinity mask of idle threads to the new CPUs, but that strategy delivered less performance. It was common that the process received the original CPU of a thread that had just migrated to another CPU. Therefore, the strategy of a unique call to `pthread_setaffinity_np` is used.

Listing 5.10 has the pseudocode of the `parallel_begin` and `parallel_end` callbacks, which work in conjunction. The `parallel_begin` marks that the runtime is inside an explicit parallel region and the number of threads it uses. At that point, the thread manager knows that the threads requested will shift to workers if they were free agents. When the region ends, the `parallel_end` is called and the `in_paralell` variable is updated to reflect that the region has ended.

```
void omptm_role_shift__parallel_begin(...){
    if(outermost_parallel){
        in_parallel = true;
        current_parallel_size = nthreads;
    }
}
void omptm_role_shift__parallel_end(...){
    if(outermost_parallel){
        in_parallel = false;
    }
}
```

Listing 5.10: Parallel begin and end callbacks.

The `task_create` (see Listing 5.11) is executed each time the runtime creates a task, independently of its type. The thread manager is only interested in *explicit tasks* since free agents can execute them. For the *explicit tasks*, the `pending_tasks` counter is increased and a call to `DLB_AcquireCpus` is done. This function tries to get a CPU that another process has lent to DLB. If the call is successful (i.e., DLB had one idle CPU, and this process was the first to claim it), the `DLB_AcquireCpus` will end up calling the `cb_enable_cpu` function and waking up or creating a thread with the free agent role.

```
void omptm_role_shift__task_create(..., int flags, ...){
    if(flags & ompt_task_explicit){
        ATOMIC_INC(&pending_tasks);
        DLB_AcquireCpus(1);
    }
}
```

Listing 5.11: Task create callback.

The `task_schedule` callback (see Listing 5.12) is emitted each time a task performs a scheduling event, identifying the type of event with the `prior_task_status` argument. There are two relevant events: the `ompt_task_switch` (i.e., when a task starts its execution) and `ompt_task_complete`. On the first one, the `pending_tasks` counter is decremented, and when more tasks are pending to be executed, the thread manager tries to acquire a CPU. When the task finishes, the thread manager checks if the CPU has been reclaimed and the number of pending tasks in the same way as in the `thread_begin` callback and performs the same actions when that happens.

```
void omptm_role_shift__task_schedule(ompt_task_status_t prior_task_status,...){
    if(prior_task_status == ompt_task_switch){
        if(ATOMIC_DEC(&pending_tasks) > 1){
            DLB_AcquireCpus(1);
        }
    }
    else if(prior_task_status == ompt_task_complete){
        /*Return CPU if reclaimed*/
        if(cpu_reclaimed){
            ...//Return cpu
        }
        /*Lend CPU if no more tasks*/
        else if(pending_tasks == 0){
            ...//Lend cpu
        }
    }
}
```

Listing 5.12: Task schedule callback.

Both `task_begin` and `task_schedule` follow a greedy strategy, trying to acquire a new CPU for the process continuously. This strategy has delivered the best performance for LeWI with previous thread managers, so it has been kept like that for the role-shifting threads. Nevertheless, exploring other heuristics as future work could be interesting and see if another increases the performance.

The last callback is the `thread_role_shift`, the new addition of the runtime. The relevant part of the code is shown in Listing 5.13. When the thread shifts from worker to free agent, it checks if the CPU has been reclaimed or if there are no more pending tasks, as in `thread_begin` and `task_schedule` callbacks.

```
void omptm_role_shift__thread_role_shift(..., ompt_role_t prior_role,
                                         ompt_role_t next_role){
    if(prior_role == OMP_ROLE_NONE){
        if(next_role == OMP_ROLE_FREE_AGENT){
            /*Return CPU if reclaimed*/
            if(cpu_reclaimed){
                ...//Return cpu
            }
            /*Lend CPU if no more tasks*/
            else if(pending_tasks == 0){
                ...//Lend cpu
            }
        }
    }
}
```

Listing 5.13: Role shifting callback.

The time that it takes a thread to reach the `thread_role_shift` callback since the API call is used to change its role is undetermined. It cannot be assumed that the CPU is still available. Therefore, the thread must be deactivated as soon as possible. Otherwise, it could interfere with the execution of another process and degrade the global performance.

5.3 Thread manager selection

As the chapter's introduction explains, three different thread managers coexist together. To allow that, DLB has a selection of the active thread manager when initializing. The function `setup_omp_fn_ptrs` registers the functions of the selected thread manager in the `omptm_funcs` global variable. Then, in the function `omptool_initialize` all the OMPT callbacks are set if they have been defined. Listing 5.14 shows the relevant part of the code. In that example, the `thread_begin` callback is set for the role-shift thread manager, but no the `thread_end` one. Finally, the DLB callbacks (e.g., when entering an MPI call) are called as shown in the `omptool__into_blocking_call` routine. This way, DLB can use the same functions independently of the thread manager selected.

```
typedef struct{
    ...//Init & Finalize functions
    /*MPI calls */
    void (*into_mpi)(void);
    void (*outof_mpi)(void);
    /*OMPT_Callbacks */
    omptm_callback_thread_begin_t thread_begin;
    omptm_callback_thread_end_t thread_end;
    ...//Other OMPT callbacks
} openmp_thread_manager_funcs_t;
static openmp_thread_manager_funcs_t omptm_funcs = {0};

static void setup_omp_fn_ptrs(omptm_version_omptm_version){
    if(omptm_version == OMPTM_OMP5){
        ...//Setting for regular OpenMP 5.X
    }
    else if(omptm_version == OMPTM_FREE_AGENTS){
        ...//Setting for the double pool implementation
    }
    else if(omptm_version == OMPTM_ROLE_SHIFT){
        omptm_funcs.into_mpi = omptm_role_shif__IntoBlockingCall;
        omptm_funcs.thread_begin = omptm_role_shift__thread_begin;
        omptm_funcs.thread_end = NULL;
        ...//Rest of the functions for role-shifting threads.
    }
}

static int omptool_initialize(...){
    ...//Some initialization code
    if(omptm_funcs.thread_begin){
        err += set_ompt_callback(ompt_callback_thread_begin,
                                    (ompt_callback_t)omptm_funcs.thread_begin);
    }
    ...//Rest of the initialization
}

void omptool__into_blocking_call(void){
    if(omptm_funcs.into_mpi)
        omptm_funcs.outof_mpi();
}
```

Listing 5.14: Thread manager selection.

Chapter 6

Evaluation

The runtime and the DLB integration have been evaluated with three HPC applications: DMRG++ [3], ParMmg [6], and Alya [15]. While benchmarks are a good tool to evaluate the performance of runtimes and systems, demonstrating that state-of-the-art HPC production codes can benefit from them is better. In the end, the final users of the advances made in the HPC computer science field are the developers and users of such applications.

Three versions of the applications have been evaluated:

- Vanilla: The original application executed with the compiler's runtime.
- Double pool: The application with the previous free agent runtime [20].
- Role-shifting: The application with the runtime developed in this project.

6.1 Environment

All the experiments have been done in the Marenostrum 4 supercomputer, hosted at Barcelona Supercomputing Center. It has 3456 nodes based on Intel processors and does not have GPUs. Each node has two sockets with Xeon Platinum 8160 @ 2.10 GHz processors, with 24 cores per CPU. The RAM per node is 96 GB, divided into 12 DIMMs at 2667 MHz. In addition, there are a few nodes with a total of 384 GB. The nodes are interconnected with Intel Omni-Path at 100 Gbit/s.

The runtime, DLB, ParMmg, and Alya have been compiled using the Intel 2017.4 suite. DMRG++ has been compiled with the Intel 2020.1 suite since it needed some OpenMP features unavailable in the 2017.4 suite. The MPI library used is Intel MPI 2017.4 in all cases. Extrae [17] 3.8.3 and Paraver [23] are used to generate traces and visualize them.

Alya and ParMmg have been evaluated with DLB. In that case, the codes have been slightly modified to include calls to `DLB_Barrier`. This call performs a barrier similar to `MPI_Barrier`, but only with the processes of the same node, and it is placed just before the MPI calls where the load imbalance happens. After that, the interception of MPI is turned off, and the only entry points of `IntoBlockingCall` and `OutOfBlockingCall` are the `DLB_Barrier` calls. This way, DLB is used only in the regions of interest and does not interfere in regions where lots of MPI calls are done in a short time since that could lead to performance losses.

Alya and ParMmg experiments were done in the middle of the project for publication in the HPCMALL workshop [9]; DMRG++ experiments were performed at the end of the project. Therefore, the versions of the runtime used are not the same. Nevertheless, some partial performance tests have been done with ParMmg and Alya to check if the performance has changed, finding variations below 1%.

6.2 DMRG++

DMRG++ is an HPC application developed by ORNL. It implements the Density Matrix Renormalization Group (DMRG) algorithm, which is used to obtain the low-energy physics of quantum many-body systems. The application is written in C/C++ and was launched back in 2009.

The version used for the tests is the DMRG++ mini-app [8,12], which captures the most computational intense kernel of the application. In that part, the app performs Kronecker products on matrices. The operation is a generalization of the outer product of matrices, and although it may resemble the matrix multiplication, it must not be confused with it.

Listing 6.1 shows the structure of the code. The program has a main matrix that has several sub-matrices per cell. Figure 6.1 shows the density of the matrix. The central cells have bigger matrices, resulting in more computations and load imbalances among the iterations. The i and j loops traverse the main matrix and are parallelized with `parallel for`. Then, the k loop accesses the cell's sub-matrices and is taskified via the `taskloop` construct. Originally, the k loop also had a `parallel for`, but it was changed so the free agents could execute the tasks.

```
for(int its = 0; its < NITS; its++){
    #pragma omp parallel for schedule(static)
    for(int i = 0; i < n; i++){
        #pragma omp parallel for schedule(dynamic, 1)
        for(int j = 0; j < n; j++){
            int size_k = Matrix[i][j]->A.size();
            #pragma omp taskloop default(shared)
            for(int k = 0; k < size_k; k++){
                ...//Task body
            }
        }
    }
}
```

Listing 6.1: DMRG++ miniapp structure.

Figure 6.2 shows two paraver traces of an execution with 2 threads in the outer parallel and 24 in the inner one. In all paraver traces, the x-axis displays the execution time, and the y-axis has a row per thread used in the execution. The information in the middle are the events, and the user has a considerable variety of them available. In this case, the Useful duration is selected, showing the periods when a thread is doing valuable computations. The colors are a gradient to determine the duration of the burst; green means low values and blue high values of the event.

The top trace corresponds to a vanilla execution using the Intel OpenMP runtime, while the bottom trace uses the role-shifting LLVM runtime. Both traces are in the same time scale, meaning they have the same duration from start to end. Having the same time scale in two traces serves to visualize the differences between them easily. In this case, both runtimes take the same time to solve the problem.

The top trace shows how all the threads work from start to end, with no apparent load imbalance. Dividing the rows matrix from Figure 6.1 in 2 (the outer level of parallelism) results in the same amount of work, so no imbalances are expected in this scenario.

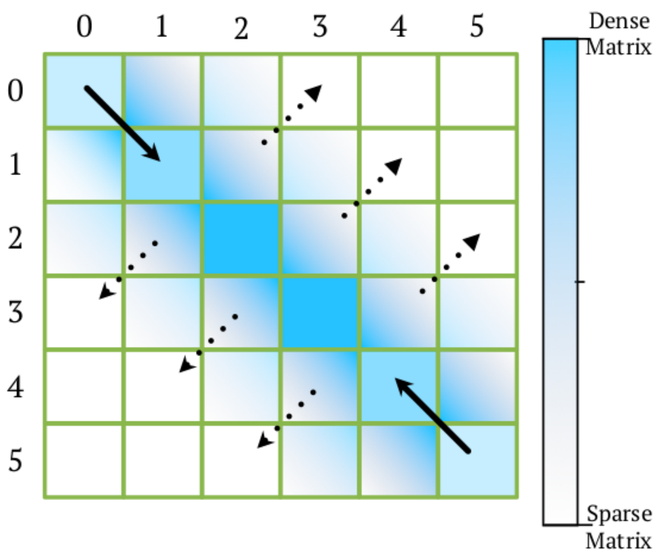


Figure 6.1: DMRG++ main matrix density distribution.

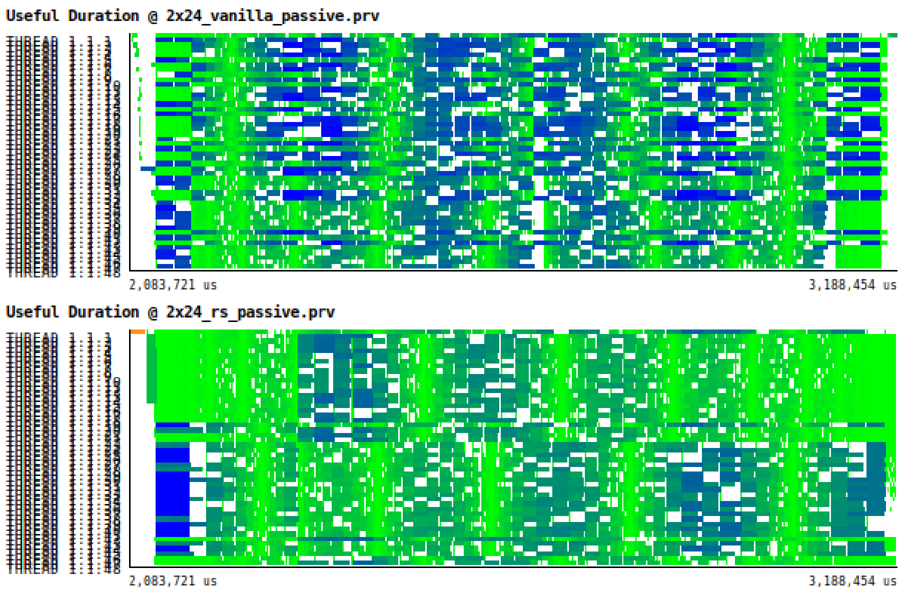


Figure 6.2: DMRG++ traces using 48 OpenMP threads. Top: vanilla version. Bottom: execution with role-shifting threads. Both traces use 2 threads in the outer parallel and 24 in the inner one.

Figure 6.3 shows two paraver traces of an execution with 4 threads in the outer parallel and 12 in the inner one. The top trace uses the Intel OpenMP runtime, and the bottom one uses the role-shifting LLVM runtime. Both traces are in the same time scale. The problem solved is the same as in Figure 6.2, but it is partitioned differently since 4 threads are used in the outer level, causing load imbalances. This is reflected in the top trace, with half of the threads working only for a reduced time at the start. On the bottom trace, the threads shift from worker to free agent and start executing tasks, balancing the execution and speeding it up.

The experiments with DMRG++ have been done in a full node of MareNostrum 4,

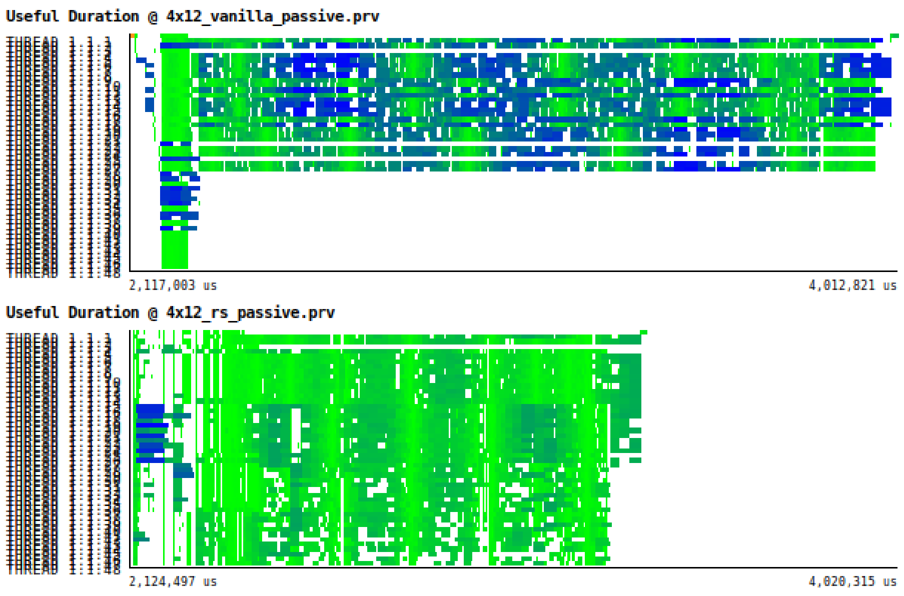


Figure 6.3: DMRG++ traces using 48 OpenMP threads. Top: vanilla version. Bottom: execution with role-shifting threads. Both traces use 4 threads in the outer parallel and 12 in the inner one.

testing different thread configurations for the two parallels. The results are shown in Figure 6.4. The y-axis displays the execution time, and the x-axis the parallel configurations. The first value is the number of threads used for the outer parallel, and the second is the threads of the inner parallel. The double-pool implementation is not evaluated here since it produced segmentation fault for all the configurations but 2x24.

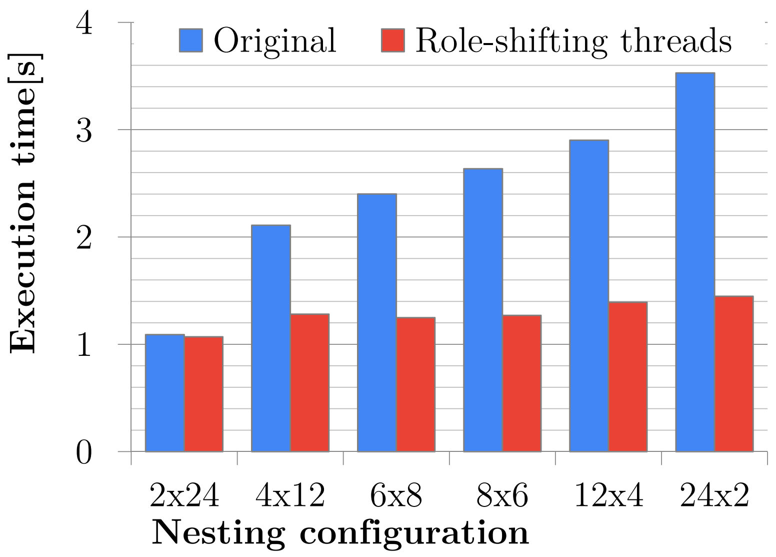


Figure 6.4: DMRG++ execution time with different parallel configurations.

Both implementations achieve a similar execution time with the 2x24 configuration, which presents almost no load imbalance. That proves the low overhead of the role-shifting implementation, even under nested parallelism. The other configurations present several load balance problems, reflected in the execution time of the vanilla application. Nevertheless, the role-shifting threads leverage that problem and present execution times near the 2x24 execution. When using the 24x2 configuration, the role-shifting runtime achieves a speedup of 2.44x compared to the same execution with the Intel runtime.

6.3 ParMmg

ParMmg is an HPC application for parallel mesh adaptation of 3D volume meshes, based on top of the sequential Mmg remesher, both developed by INRIA. Remeshing is a technique widely used in the computational solid mechanics (CSM) and computational fluid dynamics (CFD) fields, aiming to improve the mesh quality and, therefore, the final solution. The application is written in C and was originally parallelized only with MPI.

Listing 6.2 shows the structure of the main loop where the computations occur. OpenMP tasks have been added to generate work for the free agent threads. After that, the program goes into a communication-intensive phase without any OpenMP constructs. Therefore, a call to DLB_Barrier is used here to use DLB only in that part of the code.

```
for(int its; its < size; its++){
    for(int i = 0; i < size; i++){
        ...//Some code
        #pragma omp task
        {
            ...//Task body
        }
    }
    DLB_Barrier();
    MPI_Barrier();
}
```

Listing 6.2: ParMmg main loop structure.

Figure 6.5 has two paraver traces using the same input and 32 MPI processes. The first one is the original code with the compiler runtime. The second one uses the role-shifting threads runtime and LeWI. Both traces are in the same time scale. Here, the event represented is the MPI rank, meaning that each color represents a different process. White spaces mean that the MPI process is not doing useful computation (e.g., it is waiting in an MPI blocking call).

Iterations three to five are displayed. The load imbalance at the end of the big computation can be seen on the top trace (and also in the bottom one, but it is easier to look at the top one since it has few lines). It is interesting to see how the load of the ranks evolves as the execution progress. The black line marks the same point in the execution. Thanks to the usage of DLB, additional threads with the free agent role are activated in idle CPUs, and the execution progress faster. The bottom trace also has a zoom at the end of the last iteration to view more clearly how the additional threads are used.

Figure 6.6 shows the speedup of the two runtimes in conjunction with DLB compared to the vanilla version. The number of MPI ranks is 32 for all the runs. The double-pool implementation has been tested with several number of free agent threads to find the optimal since the value must be set at the start of the execution. For the double-pool, the best speedup is 1.192x, achieved with 20 free agent threads. The same speedup is reached with the role-shifting threads runtime but without the need to test several configurations.

In addition, a weak-scaling experiment has been done. The results are displayed in Figure 6.7. For the double-pool implementation, 20 free agent threads have been used for

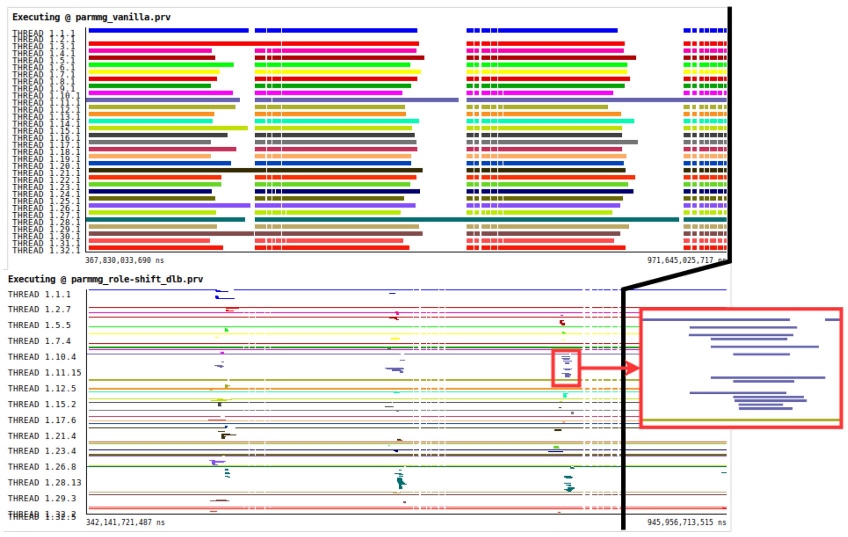


Figure 6.5: Paraver traces of three iterations using 32 MPI ranks. Top: vanilla version. Bottom: execution with role-shifting threads and DLB.

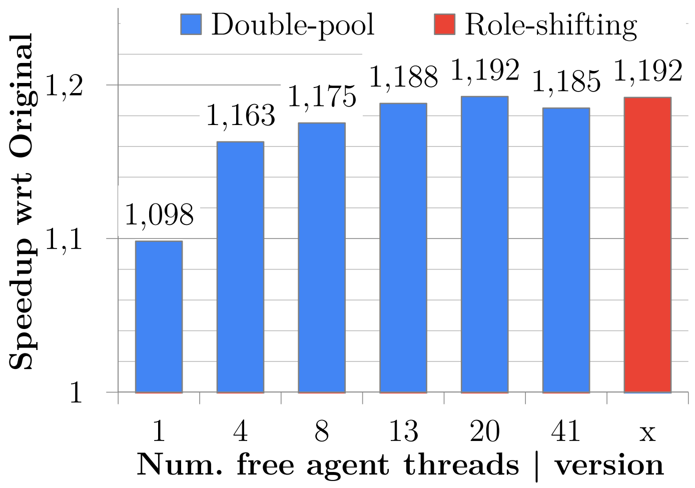


Figure 6.6: ParMmg speedup with DLB, compared to the vanilla code.

all the runes with more than 20 MPI ranks. Otherwise, the number of free agents is set to the number of MPI processes. In a weak-scaling test, the execution time should ideally be constant, but in this case, it keeps increasing as more processes are added. Nevertheless, using DLB with both versions, the execution time is always reduced, without any significant difference between double-pool and role-shifting. At 256 MPI ranks, both runtimes reduce the execution time by 260 seconds, achieving a speedup of 1.19x.

6.4 Alya

Alya is a high-performance CFD application designed to solve engineering coupled multi-physics problems. It incorporates several physics solvers, such as chemistry, non-linear solid mechanics, and incompressible flows. The program is written in Fortran and is parallelized with MPI and OpenMP, and has support for GPUs and SIMD. The multi-physics coupling is achieved by having different instances of Alya running concurrently. Each one

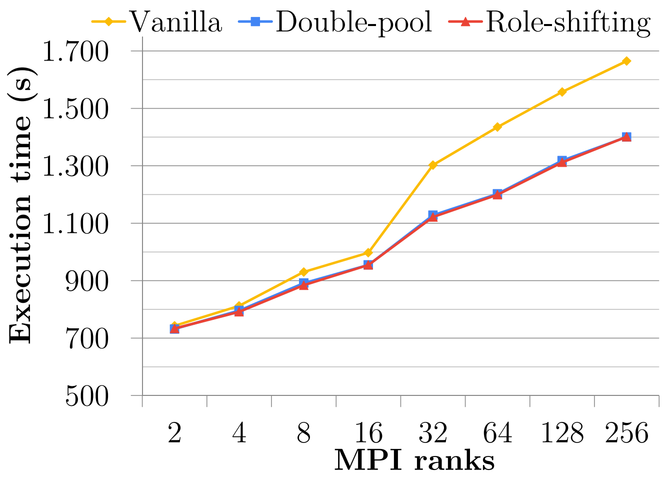


Figure 6.7: ParMmg weak-scaling experiment execution time.

solves one of the problem's physics and communicates with other instances when needed via MPI.

For this study, the application is launched with one core per MPI rank, and the OpenMP parallelization is used only for load balance purposes. Alya's OpenMP parallelization is not exhaustive; therefore, it is preferred to add more MPI ranks rather than more cores per rank. Nevertheless, the OpenMP parallelization will be helpful when running with DLB.

The input used is a production combustion problem. It is a coupled use case with two different physics: fluid simulation and chemical reactions [5, 25]. An execution trace is shown in Figure 6.8. The trace has 768 MPI ranks; the first 96 are dedicated to the fluid, and the others are used for the chemical reaction. The MPI calls are displayed in the trace, and the legend is depicted on the right. The trace has two iterations, and the two physics can be clearly identified since they follow completely different patterns. The most time-consuming part is located in the chemical solver, just before an `MPI_Barrier`, and has a poor load balance.

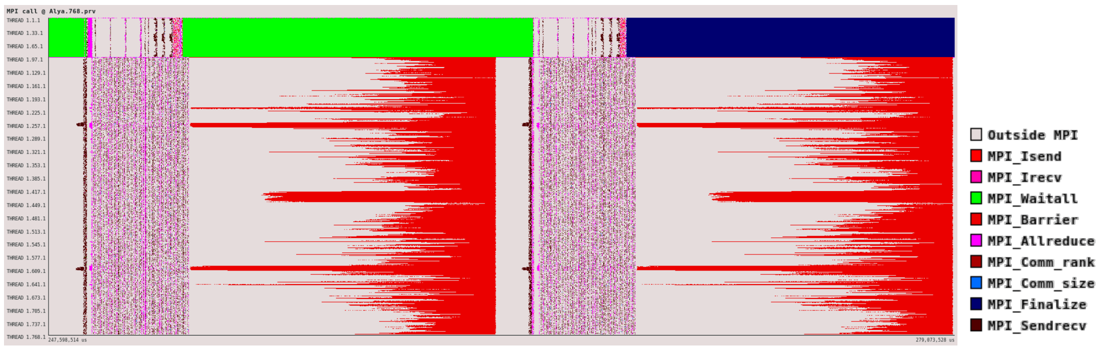
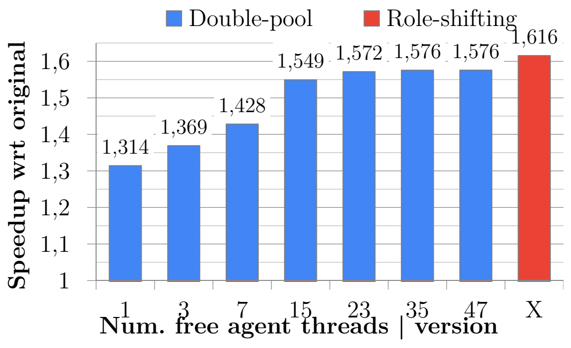
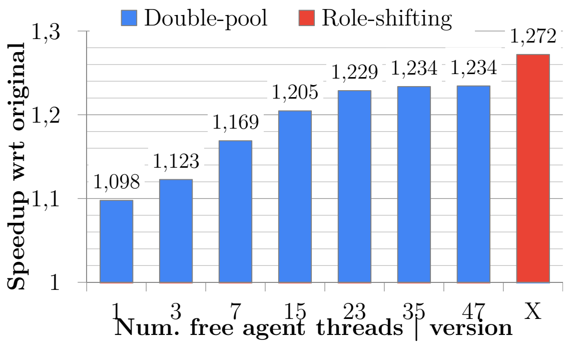


Figure 6.8: Paraver trace of 2 Alya's iterations using 768 MPI processes. The first 96 ranks solve the fluid and the other 672 solve the chemical simulation.

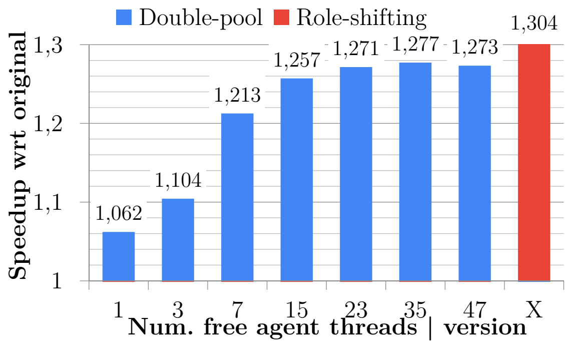
Figure 6.9 shows the speedup of the two runtimes compared with a vanilla run. Figure 6.9a uses 768 MPI ranks, Figure 6.9b 1152, and Figure 6.9c 1536. The application is tested with a different number of free agent threads for the double-pool implementation, following the same strategy as with ParMmg. The best speedup is achieved with 35 free agent threads in the three versions, ranging from 1.576x to 1.277, with 768 and 1536 MPI ranks, respectively. The role-shifting implementation delivers a speedup of 1.616x with 768 processes and 1.304 with 1536, surpassing the previous implementation without the need to tune the number of free agent threads.



(a) 768 MPI ranks.



(b) 1152 MPI ranks.



(c) 1536 MPI ranks.

Figure 6.9: Alya speedup with DLB, compared to the vanilla code, and using different number of MPI ranks.

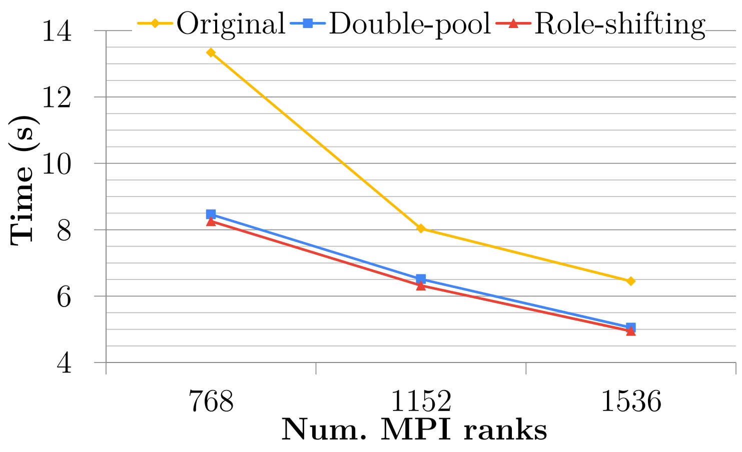


Figure 6.10: Alya strong-scaling experiment execution time.

A strong-scaling experiment has been performed with Alya, using 35 free agent threads per MPI rank. The results are shown in Figure 6.10. The usage of DLB in conjunction with free agent threads significantly improves the application's performance in all cases. The role-shifting runtime delivers better performance for all the MPI configurations than the double-pool. Therefore, it should be the runtime of choice when using Alya.

The results found in Alya do not align with the ones from ParMmg. Here, the role-shifting implementation is better than the double-pool, while with ParMmg, they were on equal terms. These differences come from overhead and task granularity. ParMmg's tasks have an average granularity of several seconds, while Alya's tasks take only a few microseconds. The double-pool implementation is more sensitive in that scenario since it requires switching the active thread of the CPU. Avoiding that, the role-shifting runtime can deliver better performance in fine-grain scenarios.

Chapter 7

Conclusions

This final master's thesis presents and develops the idea of the role-shifting threads, an extension of the LLVM OpenMP runtime. Free agent threads are now the first role of the model. In addition, the Clang compiler has been extended with support for the `free_agent` task clause. Finally, the runtime has been integrated with the Dynamic Load Balance library.

The role-shifting runtime has been evaluated with a mini-app and two complete HPC applications, one of them with a coupled execution. Comparisons with the vendor runtime (i.e., Intel) and the double-pool LLVM runtime showed the same or better results. In addition, the role-shifting runtime removes the need to decide the number of free agents for the execution. It delivers the best performance directly, easing the tasks of the user.

The goals stated at the start of the project (see Section 1.2) have been fulfilled successfully. Currently, my research group has an ongoing proposal for the free agent threads addition to the OpenMP standard, which the OpenMP committee is refining. Since the feedback so far is positive, the addition is in the OpenMP roadmap [4], and the results shown here are promising, we are confident about their addition to OpenMP 6.0 next year.

A publication in the recent ISC2022 conference [9] originated from this project. There, we present the role-shifting threads and focus on the DLB integration. In addition, the project also contributed to the original free agent threads publication [20].

7.1 Future work

Like many other software developments, this project is an ongoing effort. From my research group, we are planning to support the runtime and the integration with DLB for a long time. This includes refining the algorithms, performing more tests, solving any bug in the code, and answering any doubts from our users.

In this document, several points have been stated as future work. The following list summarizes them:

- Solve some minor bugs in the runtime. So far, we are aware of a bug in the LLVM master's branch and would like to mend it.
- Perform more exhaustive tests and integrate them into the respective testing suites.
- Merge the main git branch into the role-shifting thread branch. Some new addition to the runtime could be missing at the moment, so that would make it more appealing to new users.

- Explore ideas for new roles.
- Try different heuristics in DLB regarding when to ask for more CPUs, and evaluate their performance.
- Continue pushing the free agent threads into the OpenMP 6.0 standard.
- Integrate DLB with the free agent threads of OpenMP 6.0 if that happens.

Bibliography

[1] DLB repository. https://github.com/bsc-pm/dlb/tree/free_agents - Last accessed June 2022

[2] LLVM repository. https://github.com/bsc-pm/llvm/tree/omp-role-shift - Last accessed June 2022

[3] Alvarez, G.: The density matrix renormalization group for strongly correlated electron systems: A generic implementation. Computer Physics Communications **180**(9), 1572–1578 (2009)

[4] Bronis R. de Supinski: Recent, Current and Future OpenMP Directions: OpenMP 5.1 and More!, https://www.openmp.org/wp-content/uploads/OpenMP_SC20-deSupinski.pdf, accessed: 2022-01-31

[5] 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 and Combustion **91**(2), 347–372 (Sep 2013). https://doi.org/10.1007/s10494-013-9470-z, https://doi.org/10.1007/s10494-013-9470-z

[6] Cirrottola, L., Froehly, A.: Parallel unstructured mesh adaptation using iterative remeshing and repartitioning. Research Report RR-9307, INRIA Bordeaux, équipe CARDAMOM (Nov 2019), https://hal.inria.fr/hal-02386837

[7] Continuous Delivery Foundation: Jenkins. https://www.jenkins.io// - Last accessed June 2022

[8] Criado, J., Garcia-Gasulla, M., Labarta, J., Chatterjee, A., Hernandez, O., Sirvent, R., Alvarez, G.: Optimization of condensed matter physics application with OpenMP tasking model. In: International Workshop on OpenMP. pp. 291–305. Springer (2019)

[9] Criado, J., Lopez, V., Vinyals-Ylla-Catala, J., Ramriez-Miranda, G., Teruel, X., Garcia-Gasulla, M.: Exploiting OpenMP malleability with free agent threads and DLB. In: International Conference on High Performance Computing. pp. In–press. Springer (2022)

[10] 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)

[11] Duran, A., Ayguadé, E., Badia, R.M., Labarta, J., Martinell, L., Martorell, X., Planas, J.: OmpSs: a proposal for programming heterogeneous multi-core architectures. Parallel Processing Letters **21**(02), 173–193 (2011)

[12] Elwasif, W., D'azevedo, E., Chatterjee, A., Alvarez, G., Hernandez, O., Sarkar, V.: MiniApp for Density Matrix Renormalization Group Hamiltonian Application Kernel. In: 2018 IEEE International Conference on Cluster Computing (CLUSTER). pp. 590–597. IEEE (2018)

[13] Garcia, M., Corbalan, J., Labarta, J.: Lewi: A runtime balancing algorithm for nested parallelism. In: Parallel Processing, 2009. ICPP '09. International Conference on. pp. 526–533 (Sept 2009). https://doi.org/10.1109/ICPP.2009.56

[14] Garcia, M., Labarta, J., Corbalan, J.: Hints to improve automatic load balancing with lewi for hybrid applications. Journal of Parallel and Distributed Computing **74**(9), 2781–2794 (2014)

[15] Garcia-Gasulla, M., Houzeaux, G., Ferrer, R., Artigues, A., López, V., Labarta, J., Vázquez, M.: MPI+ X: task-based parallelisation and dynamic load balance of finite element assembly. International Journal of Computational Fluid Dynamics **33**(3), 115–136 (2019)

[16] GitLab Inc.: GitLab. https://about.gitlab.com/// - Last accessed June 2022

[17] Llort, G., Servat, H., González, J., Giménez, J., Labarta, J.: On the Usefulness of Object Tracking Techniques in Performance Analysis. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. pp. 29:1–29:11. SC '13, ACM, New York, NY, USA (2013). https://doi.org/10.1145/2503210.2503267, http://doi.acm.org/10.1145/2503210.2503267

[18] LLVM Foundation: Clang project webpage. https://clang.llvm.org/ - Last accessed June 2022

[19] LLVM Foundation: LLVM project webpage. https://llvm.org/ - Last accessed June 2022

[20] Lopez, V., Criado, J., Peñacoba, R., Ferrer, R., Teruel, X., Garcia-Gasulla, M.: An openmp free agent threads implementation. In: International Workshop on OpenMP. pp. 211–225. Springer (2021)

[21] Lopez, V., Ramirez Miranda, G., Garcia-Gasulla, M.: 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. p. 3–10. PERMAVOST '21, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3452412.3462753, https://doi.org/10.1145/3452412.3462753

[22] OpenMP Architecture Review Board: OpenMP 5.2 Specification. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf - Last accessed June 2022

[23] 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)

[24] Tian, S., Doerfert, J., Chapman, B.: Concurrent execution of deferred openmp target tasks with hidden helper threads. In: International Workshop on Languages and Compilers for Parallel Computing. pp. 41–56. Springer (2020)

[25] 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. Proceedings of the Combustion Institute **35**(2), 1167–1174 (2015)