

Evaluation of Heuristic Task-to-Thread Mapping Using Static and Dynamic Approaches

Mohammad Samadi¹ *, Tiago Carvalho¹, Luís Miguel Pinho¹, and Sara Royuela²

¹ Polytechnic Institute of Porto & INESC TEC, Porto, Portugal
{mmas, tdc, lmp}@isep.ipp.pt

² Barcelona Supercomputing Center, Barcelona, Spain
sara.royuela@bsc.es

Abstract. The increasing needs for high-performance computing capabilities in real-time systems is pushing the need for the use of complex parallel and heterogeneous platforms. This complicates and even invalidates the use of traditional mechanisms to derive and provide real-time guarantees. Task-to-thread mapping is a key process, not only to provide an optimal performance by enhancing load balance and reducing the overheads of the parallel runtime system, but also to enable predictable execution.

To address these somehow contradictory requirements, this work leverages the OpenMP programming model and its implementation in the LLVM compilation framework by virtue of its programmability, performance, portability, and the recently demonstrated time-predictability. In this context, the paper provides an evaluation of different approaches to perform task-to-thread mapping, particularly comparing dynamic mapping, with mapping decisions made during execution, with a static mapping approach, where a fixed mapping is recorded in a preliminary execution, and then repeated in all executions. Static mapping is usually the preferred approach in real-time systems, due to being perceived as more predictable, whilst dynamic mapping is generally perceived as providing higher average performance.

The obtained results allow to understand that although dynamic mapping indeed provides in general higher performance, it also may provide less variability, thus partially addressing the predictability concern, and opening opportunities for dynamic mapping in real-time high-performance systems.

Keywords: OpenMP · Task-to-thread mapping · Heuristics · Timing analysis · Dynamic mapping · Static mapping.

1 Introduction

The increasing complexity and scale of societal challenges are driving the need for high-performance computing (HPC) capabilities beyond its long-established

* PhD Candidate, Universitat Politècnica de Catalunya, Barcelona, Spain

applicability in scientific applications. This is the case of complex cyber-physical systems like those currently used to implement advanced functionalities in domains traditionally implemented in embedded devices. Clear examples of these computing capabilities are those exposed by smart mobility functionalities like the advanced driving assistance system tested in the Florence tramway network [15], adaptive optics applications based on artificial intelligence methods used in the astronomy domain [12], and the distributed architecture proposed by Philips in the TRANSACT project for image-guided therapy and diagnostic imaging systems [8].

Experts in these domains are however not familiar with the intricacies of modern parallel architectures. Therefore, simple yet powerful interfaces are of paramount importance to simplify the efficient exploitation of such architectures. In this scope, OpenMP has been lately considered for its use in real-time HPC applications, not only by virtue of its programmability and efficiency [7], but also because the language committee is introducing features, like detached tasks [11], free agent threads [3], and cancellations [17], that are fostering the spread of the language beyond its classic HPC domain. Furthermore, several studies indicate the compatibility between the OpenMP tasking model and commonly used mechanisms for real-time analysis [16] and functional correctness [13], key for the use of this programming model in real-time HPC systems.

The OpenMP specification leaves, however, the door open to implementors to decide how the system runs. For example, many features in the specification are hints (e.g., the `priority` clause, which defines a possible order in the execution of ready tasks, and the `affinity` clause, recommending the location where tasks can be executed) or they have an implementation defined behaviour (e.g., the user defining of a number of threads higher than those available in the system). Furthermore, the tasking model is, by nature, a descriptive mechanism that only exposes what can be parallelized rather than expressing how to do it. In this scope, the task-to-thread mapping process is crucial to provide real-time HPC applications with the timing guarantees they require.

Common OpenMP runtime implementations implement task-to-thread mapping as a completely dynamic process based on work-stealing for an enhanced load balance. In the case of GCC [2], the system is implemented with a single queue storing all ready tasks, and letting all threads steal from that queue. In the case of LLVM [1], instead, each thread has its own queue, and idle threads can steal work from other threads. The task-to-thread mapping can only be adjusted through mechanisms that allow, for example, to define the number of threads in a team (e.g., with the `num_threads` clause or the `OMP_NUM_THREADS` environment variable), or how OpenMP threads are mapped to processors in a device (e.g., with the `proc_bind` clause or the `OMP_PROC_BIND` environment variable).

This highly dynamic behaviour, intended for better balancing work-load and achieving higher performance, is however contradictory with the required capability in real-time systems for providing reasoning on time predictability. Noteworthy, it is highly common to introduce offline knowledge in real-time systems to enable a more static nature for scheduling decisions, e.g., worst-case execution

time, fixed-priority scheduling, and, most relevant for this work, static mapping approaches. The general impression in the real-time systems community is that dynamic mapping performs better in terms of performance, but static mapping provides less variance (i.e., less jitter), allows better prediction of data placement and enables tighter offline schedulability analysis.

In this regard, this work has been performed to devise and study static task-to-thread mapping algorithms to use OpenMP in high performance real-time systems [9,6,18]. These algorithms take decisions based on timing or structural properties of the parallel code, to devise, previous to execution, the mapping of the parallel tasks to the threads. During execution, the dispatching process is a simple enforcement of the planned mapping.

Based on these facts, a previous work [6] has proposed a series of heuristics to specify task-to-thread mapping using two different phases. First, the *allocation phase* allocates ready tasks to allocation queues, and then the *dispatching phase* selects tasks from queues and dispatches them to OpenMP threads. Later, another work [14] extends the previous work to consider parallel multi-queue systems (like LLVM) and it has proven its benefits for some of the proposed heuristics compared to previous methods (through simulation). An off-line simulation of the execution creates a static mapping of the individual tasks to the threads, which is then executed by an existing custom LLVM implementation. Nevertheless, this work also opens the discussion on the use of these approaches during execution, not only beforehand to provide static mappings, and provides preliminary evaluation of the implementation of the heuristics in the OpenMP scheduler of LLVM.

This paper further extends this previous task-to-thread mapping approach [14] by studying the behavior of some of these heuristics under different executing conditions, comparing their dynamic behaviour with that of the static counterpart. The main contribution is therefore the comparison of the execution of the heuristics considering decisions made during execution, against static enforcement of the mapping. This evaluation is not directly comparable with the study in [14] because this paper also provides the following minor contributions: (i) static mapping is determined by recording the execution in the real platform instead by simulation, (ii) changes in the implementation of one of the heuristics (the one selecting the queue based on the minimum total execution time) to reduce its overhead, and (iii) implementation of a new heuristic in the dispatching phase.

2 Background

OpenMP is the de-facto standard API for parallel programming shared memory machines. It is composed of a set of compiler directives, runtime routines, and environment variables that influence the dynamic behavior of the system. Early releases included a *thread-centric model* useful for exploiting loop-intensive and data-parallel kernels in a prescriptive fashion. Later releases, however, moved to a more descriptive fashion introducing a powerful *task-centric model* that

enables highly dynamic and irregular parallelism. This model allows expressing different characteristics for the tasks, including priorities, to enable an ordering in the execution of ready tasks, affinity, to provide hint about how tasks should be assigned to resources, or tiedness properties, enabling the binding of task to thread or, otherwise, allowing tasks to be resumed by a different thread than the one that started their execution.

The tasking model is particularly interesting to represent high-performance real-time systems because the Task Dependency Graph representation of the data-flow between tasks perfectly maps with the DAG-based theory of timing analysis for real-time systems. Furthermore, preliminary works have provided guarantees about the time predictability of the OpenMP tasking model. For these reasons, this work focuses on applications that expose parallelism using OpenMP tasks. Although our kernels only use the `task` directive, any task generating construct could be considered. Figure 1 shows an example of the type of kernel this work is targeting, where a `parallel` directive (line 4) creates a team of threads, a `single` directive (line 5) allows only one thread to execute the sequential region inside. In the meantime, the rest of threads in the team wait in the implicit `barrier` at the end of that region (line 19) until there are ready tasks created through the `task` directive (line 14). A task that creates another task is *parent* of the created task, while the created task is its *child*. All tasks created by the same task are *siblings*. In OpenMP, tasks become ready when all their input dependencies (including the `in` and `inout` modifiers) from sibling tasks are satisfied.

```

1 void heat_propagation (void) {
2   ...
3   for (int i=0; i<Iters; i++) {
4     #pragma omp parallel
5     #pragma omp single
6     for (int ii=0; ii<NB; ii++)
7       for (int jj=0; jj<NB; jj++) {
8         int inf_i = 1 + ii * bx;
9         int sup_i = ((inf_i + bx) < sizex - 1) ? inf_i + bx : sizex - 1;
10        int inf_j = 1 + jj * by;
11        int sup_j = ((inf_j + by) < sizey - 1) ? inf_j + by : sizey - 1;
12        #pragma omp task depend(in: u[inf_i-bx][inf_j], u[sup_i][inf_j], \
13                               u[inf_i][inf_j-by], u[inf_i][sup_j]) \
14                               depend(inout: u[inf_i][inf_j])
15        for (int i = inf_i; i < sup_i; ++i)
16          for (int j = inf_j; j < sup_j; ++j)
17            u[i][j] = 0.25 * (u[i][j-1] + u[i][j+1] + u[i-1][j] + u[i+1][j])
18          ;
19        // Implicit barrier
20      }
21    ...
22  }

```

Fig. 1: Parallel simulation of heat propagation using OpenMP.

Extensions to the tasking model to reduce the contention of the runtime in resolving dependencies and creating tasks [10,19] are being proposed to the language committee. These extensions are based on storing the Task Dependency Graph that represents the parallel execution of an OpenMP task-based program. The graph can be generated statically (if the information required can be derived at compile-time) or dynamically (recording the execution at run-time). Prototype implementations are publicly available in LLVM upstream³ for the dynamic version, and a custom LLVM⁴ for both the static and the dynamic versions. This work assumes a TDG for a given application has been previously generated, and hence can be used to store metrics about the execution time of tasks. To that end the custom LLVM is used and extended for the purpose of task-to-thread mapping.

3 Related Work

Mainstream OpenMP schedulers, i.e., LLVM and GCC, support Breadth First Scheduling (BFS), in which the thread executing sequential code creates all child tasks and, when stopped at some memory fence, it starts executing them; other threads can execute tasks as soon as these are ready. In GCC, all threads place tasks and steal tasks from a centralized queue; in LLVM, instead, each thread encountering a ready task places it to its own queue, reducing the potential contention, and threads can steal tasks from other threads' queues. To further avoid overhead, in LLVM, each queue owner inserts and removes tasks from the head of its queue, while the rest do it from the tail.

Work-stealing is a mechanism widely used and well-suited for enhancing load-balance [5]. Consequently, previous works [16] have evaluated the time predictability of the work-conserving approach of common OpenMP implementations based on dynamic mapping. This work concluded that tied tasks are not an option in critical systems due to the extreme pessimism they introduce in the resulting scheduling, while untied tasks might be a solution to reduce the pessimism.

Nonetheless, predictability and high-performance might still be opposed characteristics because the dynamism introduced with dynamic mapping may cause very pessimistic results in timing analysis [4]. Consequently, real-time HPC applications may potentially suffer from bad scheduling decisions. For that reason, timing analysis is classically approached using static solutions. This is the case of Melani et al. [9], who enabled OpenMP static scheduling for safety-critical systems, based on single-queue, using two methods: An optimal one that performs static allocation using an Integer Linear Programming (ILP) formulation, and a sub-optimal one that calculates a worst-case makespan bound (close to the optimal method) using various heuristics, including Longest Processing Time (LPT), which selects the task with the longest WCET, Shortest Processing Time (SPT), which selects the task with the shortest WCET, Largest Number of Successors

³ <https://github.com/llvm/llvm-project/commit/36d4e4c9b5f6cd0577b6029055b825caaec2dd11>

⁴ <https://gitlab.bsc.es/ppc-bsc/software/llvm-taskgraph>

in the Next Level (LNSNL), which selects the task with the largest number of immediate successors, Largest Number of Successors (LNS), which selects the task with the largest number of total successors, and Largest Remaining Workload (LRW), which selects the task with the largest workload to be executed by its successors. The most important problems in existing mapping algorithms are that they do not use temporal conditions of the system in each stage of the mapping process, as well as do not apply a multi-queue system to work-balance threads and reduce contention.

Based on the good results provided by the heuristic algorithms, which enhance the predictability of the system without losing performance, further work has been proposed following heuristic-based mechanisms. Samadi et al. [14] presented a heuristic task-to-thread mapping for parallel real-time systems. This work considers the multi-queue tasking system, where each thread includes an individual queue, to reduce the contention. The mapping process is carried out through two phases: allocation and dispatching. In the allocation phase, the most appropriate queue for each OpenMP task is selected using six heuristics, with the objective of improve the temporal predictability of the system through improving the load balancing of the queues based on the number of tasks, the execution time, and the response time. These heuristics (called allocation heuristics) allow selecting the queue with the minimum number of task-parts (MNTP), the next thread (NT), the thread with the most recent idle time (MRIT), the queue with the minimum total execution time (MTET), the queue with the maximum total response time (MTRT), or the most suitable queue through a multi-criteria decision (TMCD) based on some of the previous heuristics. In the dispatching phase, an idle thread selects a ready task from its queue, if any, using three heuristic algorithms with the objective of improving the work-conserving nature of the mapping process by selecting the shortest parts or increase the possibility of obtaining new parts by selecting the task with the longest response time. These heuristics (called dispatching heuristics) select the task with the minimum execution time (MET), the maximum response time (MRT), or a multi-criteria decision (MCD) based on the features of the MET and MRT. The performance of these heuristics has been evaluated based on simulations under different configurations (i.e., various system models and number of threads), as well as experiments using two kernels from real-world applications under different configurations (i.e., various number of threads and binding options). In most of the cases, the MTET allocation heuristic and the MET dispatching heuristic perform better than the other algorithms.

The main problems with the last mentioned heuristic-based mapping is that (i) the implementation of the MTET heuristic in LLVM is not efficient, as the scheduler should update the information of allocation queues before each task allocation process, which can increase the overhead of the mapping process, (ii) the MRT heuristic needs more information of the system (i.e., the total execution time of tasks and the system deadline), which can also increase the mapping overhead, (iii) the evaluation did not consider embarrassingly parallel applications that might exploit the full capabilities of the hardware, and (iv) the

experiments are carried on based on the WCET, and hence they do not consider the variability of the results. Consequently, this paper targets these problems by updating the MTET, replacing the MRT with a new heuristic algorithm, and evaluating the heuristics in more scenarios, respectively. Moreover, as also mentioned in the introduction, it is essential to evaluate the performance and time predictability of task-to-thread mapping algorithms under both dynamic and static approaches. This process is needed to clarify how much performance is lost in static mapping compared to dynamic mapping. This feedback should be obtained from this paper to identify which approach (i.e., dynamic or static) is efficient for using the heuristic mapping in parallel real-time systems.

4 Methodology for Static Mapping

This section describes the details of the proposed methodology for static task-to-thread mapping. For this work the mapping of tasks to threads (based on dynamic mapping) is first recorded and then the applications are executed based on the recorded mapping. Figure 2 shows a detailed workflow explaining the recording process needed for static mapping.

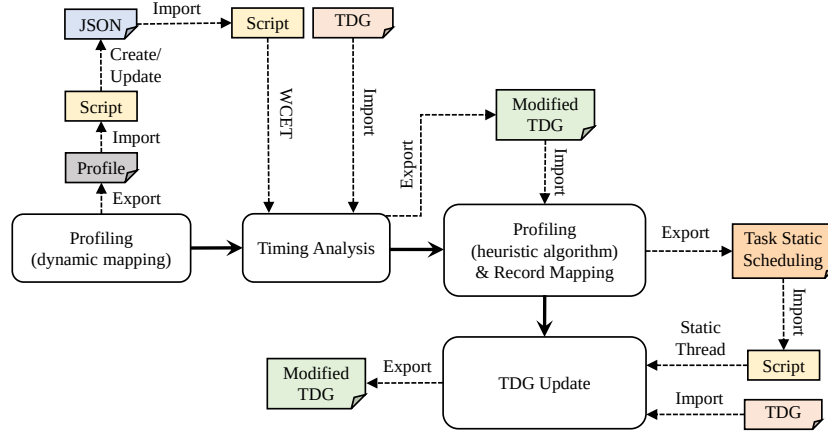


Fig. 2: Record the mapping of tasks with threads used in static mapping.

During the profiling phase based on dynamic mapping, the vanilla LLVM scheduler is used together with the Extrae⁵ instrumentation tool to gather the execution time of tasks. This profiling information is used by a script⁶ to create (in the first iteration) or update (in the remaining iterations) a JSON file that includes the structure of the Task Dependency Graph (TDG) and the execution times of all tasks.

⁵ <https://tools.bsc.es/extrae/>

⁶ <https://gitlab.bsc.es/ampere-sw/wp2/tdg-instrumentation-script/>

During the timing analysis phase, a script determines the task execution time based on the minimum, average or maximum of the execution times stored in the JSON file (for this work, applications are run 10 times). This information is used to calculate the WCET of each task, which is then inserted as an attribute (called *exe.time*) to each node in a TDG that will be used by the runtime to orchestrate the parallel execution. This TDG, stored in a C++ file, contains information of all nodes such as a static ID, number of successors, and the thread that executed the task.

To carry out the record mapping part, the mapping of tasks to threads is recorded while the program is executing. This process is performed separately for all the pairs of the heuristic algorithms belonging to the allocation and the dispatching phases, including MNTP-MET, MNTP-HET, MTET-MET and MTET-HET. Furthermore, it is performed in the LLVM inside the `__kmp_remove_my_task()` and `__kmp_steal_task()` functions. Finally, the recorded information is stored in a structure that contains pairs of *task.id*, *thread.id*.

During the TDG update phase, the TDG for the static mapping is created based on the recorded information. An additional script is used to merge the original TDG with the *static.thread* recorded for each task (where -1 is the default and causes the task to be executed by any idle thread). The modified TDG will be used in the execution phase to execute the program based on static mapping.

5 LLVM extensions

This section describes the adaptations made to the LLVM OpenMP runtime, which are publicly available⁷, for (i) improving the MTET heuristic and (ii) implementing a new heuristic algorithm for the dispatching phase, called highest execution time (HET).

Improvement of the MTET Heuristic MTET is the allocation heuristic that selects queue with a minimum total execution time. In the original implementation of MTET [14], before selecting the most appropriate queue for a ready task in the allocation phase, the total execution times of existing tasks in all queues are first calculated and then the queue with the minimum time is selected. As this process is used for each task allocation, it increases the mapping overhead considerably. Consequently, the implementation has been improved by materializing the time expend in execution for each thread. More specifically, the *kmp_base_thread_data* structure, which stores the data of each thread's queue, is incremented with new member of type *kmp_int64* and called *tot_exe_time*, which contains the updated total execution time (in ns) of the tasks existing in the queue (0 when the structure is initialized). The value of this new member is updated each time a task is pushed into or removed from a thread's queue, considering the execution time that has been pre-computed for that task.

⁷ <https://github.com/m-samadi/heuristic-mapping-LLVM>

HET Heuristic Algorithm The HET heuristic selects the task from an allocation queue that has the highest workload, i.e., associated execution time. The main goal of introducing this heuristic is to reduce the workload of each queue and increase its chance of getting new tasks with some allocation heuristics, such as MTET. In the LLVM implementation, each task is searched in the corresponding Task Dependency Graph (more than one can coexist, as more than one application can be running in parallel), and the task with higher execution task is selected. The complexity of the algorithm is $\mathcal{O}(n)$, where n is the number of tasks in the queue.

6 Performance Evaluation

This section describes the experiments done to evaluate the performance of the heuristics when used for static mapping (off-line) and dynamic mapping (on-line). The process is carried out using several applications under different configurations.

6.1 Experimental Setup

Applications. The evaluation uses 3 well-known kernels from the HPC domain:

- *Axpy*, a Level 1 operation in the Basic Linear Algebra Subprograms (BLAS) package that combines scalar multiplication and vector addition. The code, presented in Figure 3a generates a TDG with 127 independent tasks.
- *Heat*, a simulator of heat diffusion based on the Gauss-Seidel method implementing a stencil computation. The code, presented in Figure 1, generates a TDG with 640 tasks and 2128 dependencies.
- *SparseLU*, a matrix decomposition implementing a diamond-like form of irregular parallelism. The code, presented in Figure 3b, generates a TDG with 1496 tasks and 3960 dependencies.

Hardware and Operating System. The platform used for the experiments is an NVIDIA Jetson AGX Xavier with an 8-core NVIDIA Carmel Arm v8.2 64-bit CPU and 32GB 256-bit RAM. Unlike HPC clusters, this platform allows tuning the real-time behavior of the system, so applications are executed with the highest real-time priority, i.e., `SCHED_FIFO`, based on the off-the-shelf Linux OS and executing only one application at a time. It is worth noting that this platform is more suitable for high-processing applications, such as those used in automotive industry, which enable the use of artificial intelligence and data analysis processes. That is why we choose this platform for the evaluation results.

Compiler and runtime system. This work is built upon an extended version of LLVM/Clang 17.0.0 that supports OpenMP 5.0 and the extensions for the materialization of a Task Dependency Graph (TDG), so-called *Taskgraph*⁴. The extensions implemented for this work are also publicly available⁷. The task stealing feature is disabled.

```

1 void saxpy(double *x, double *y) {
2   for (int i=0; i<Iters; i++) {
3     #pragma omp parallel
4     #pragma omp single
5     for (int i = 0; i < N; i+=BS1) {
6       #pragma omp task
7       for(int j = 0; j < BS1; j++)
8         y[i+j] += a * x[i+j];
9     }
10  }
11 }

```

(a) Apxy.

```

1 void sparselu(float **M) {
2   for (int i=0; i<Iters; i++) {
3     #pragma omp parallel
4     #pragma omp single
5     for (int kk=0; kk<S; kk++) {
6       #pragma omp task firstprivate(kk) shared(M) depend(inout: M[kk*S+kk])
7       lu0(M[kk*S+kk]);
8     }
9     for (int jj=kk+1; jj<S; jj++) {
10      #pragma omp task firstprivate(kk, jj) shared(M) \
11      depend(in: M[kk*S+kk]) depend(inout: M[kk*S+jj])
12      if (M[kk*S+jj] != NULL) fwd(M[kk*S+kk], M[kk*S+jj]);
13    }
14    for (int ii=kk+1; ii<S; ii++) {
15      #pragma omp task firstprivate(kk, ii) shared(M) \
16      depend(in: M[kk*S+kk]) depend(inout: M[ii*S+kk])
17      if (M[ii*S+kk] != NULL) bdiv (M[kk*S+kk], M[ii*S+kk]);
18    }
19    for (int ii=kk+1; ii<S; ii++)
20      for (int jj=kk+1; jj<S; jj++) {
21        #pragma omp task firstprivate(kk, jj, ii) shared(M) \
22        depend(in: M[ii*S+kk], M[kk*S+jj]) depend(inout: M[ii*S+jj])
23        if ((M[ii*S+kk] != NULL) && (M[kk*S+jj] != NULL))
24          if (M[ii*S+jj]==NULL) M[ii*S+jj] = allocate_clean_block();
25          bmod(M[ii*S+kk], M[kk*S+jj], M[ii*S+jj]);
26      }
27    }
28  }
29 }
30 }

```

(b) SparseLU.

Fig. 3: Parallel kernels implemented with OpenMP.

Configurations. The applications are executed using different number of threads, i.e., 4 and 8, and binding options, i.e., spread and close. Each configuration and pair of heuristics is executed 50 times; the first 10% of the results are omitted (considering that a warm-up phase) and only the remaining results are included in the experiments. Note that in our long-term evaluations running on the hardware platform mentioned above, we found that 10% is sufficient to exclude anomaly results from the rest.

6.2 Experiments

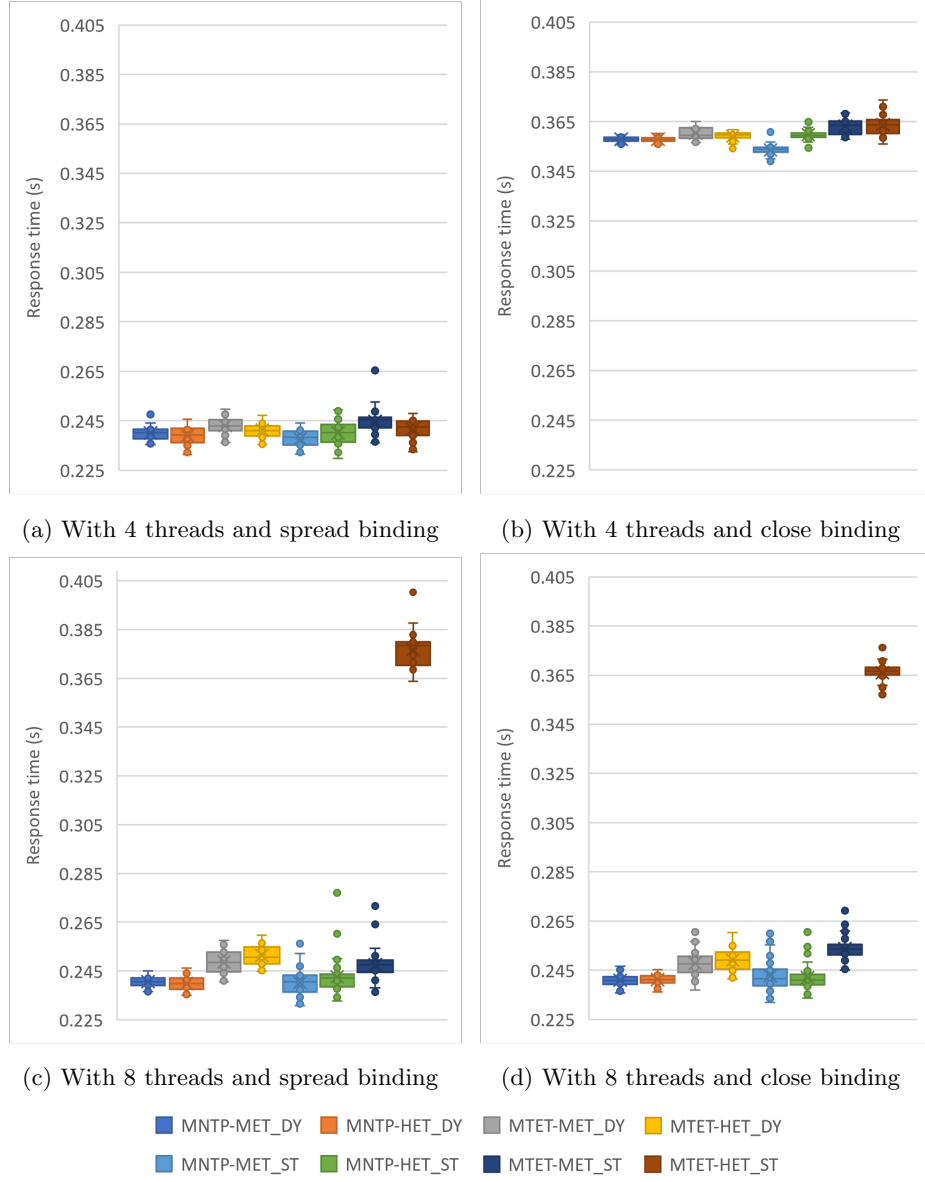
This section shows the experimental results of running the different applications and configurations on the target platform using the proposed pairs of heuristics in dynamic mapping (the name of the heuristics is followed by “DY”) and static mapping (the name of the heuristics is followed by “ST”).

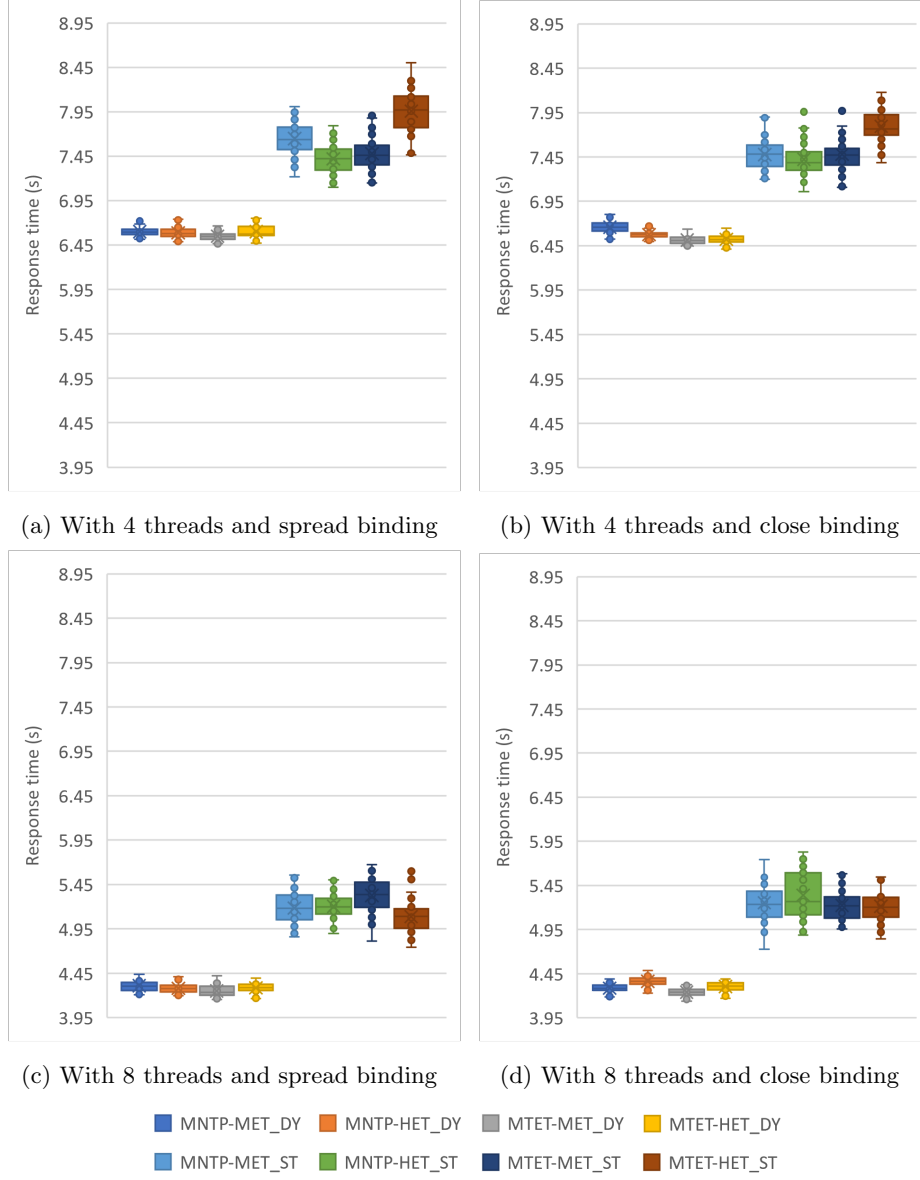
Figure 4 shows the box-plots with the results obtained for the *Axy* under 4 configurations, i.e., close and spread binding, and 4 and 8 threads. The x-axis shows the different pairs of heuristics and the y-axis shows the execution time in seconds (s). Based on the results, we can extract the following conclusions: (i) for dynamic mapping, the variability of MNTP-X is smaller than that of the other heuristic pairs, (ii) the response time obtained with MNTP-X is slightly lower than the others, (iii) the response time with MTET-HET in static mapping when using 8 threads is noticeably higher than the others, as HET selects coarser tasks from the queue, and (iv) the response time obtained in the case with 4 threads and spread binding has not been decreased by increasing the number of threads to 8, as the number of (L2 Level) cache misses is high with spreading tasks across cores. Overall, this results show that there is not a considerable difference between the results obtained in dynamic mapping with those obtained in static mapping, meaning that the heuristics serve equally in both cases. These results were expected for this application because (a) all tasks have the same workload, and (b) tasks have no data dependencies among them, and hence all tasks are allocated to the queues at the beginning of the execution.

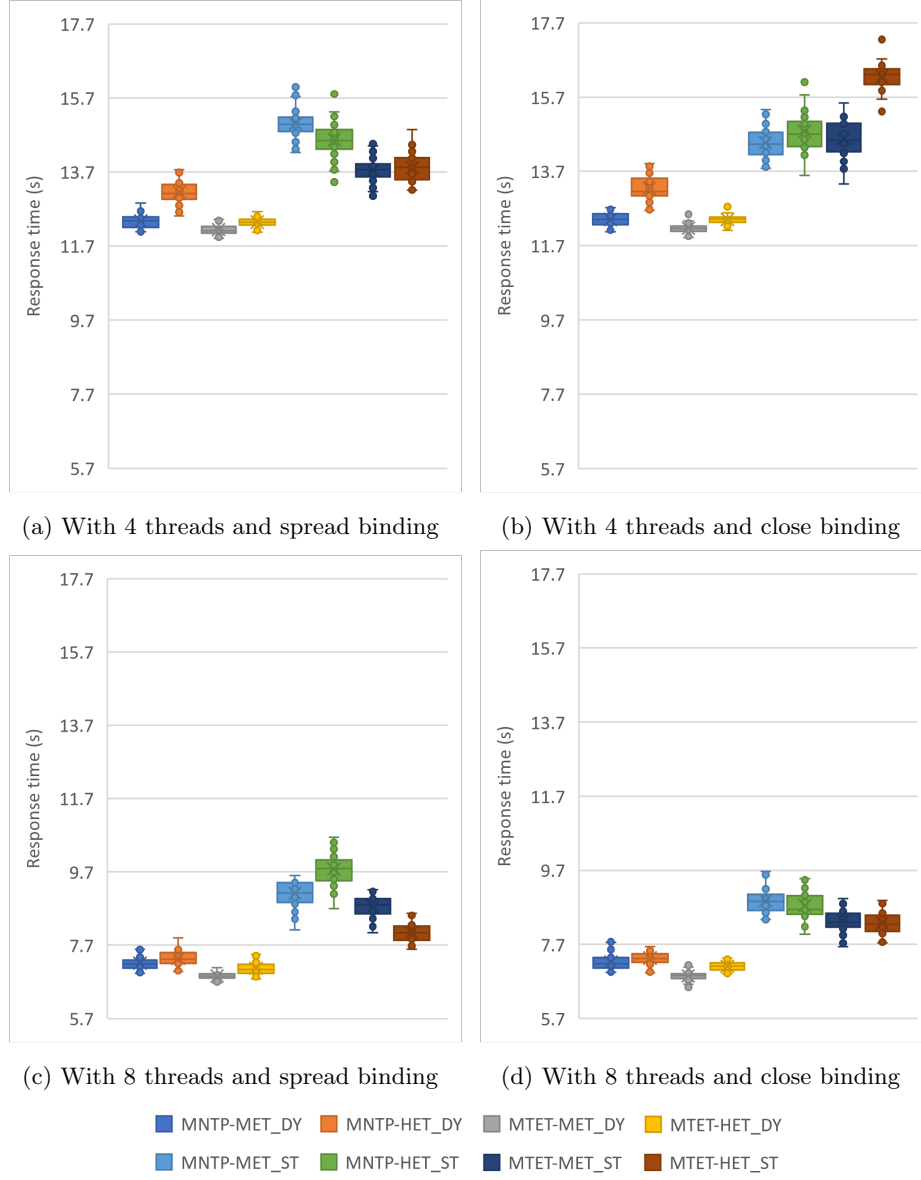
Figure 5 shows the box-plots with the results obtained for the *Heat* under the same configurations as before. From these results, we can conclude that: (i) the response time and variability of results achieved in dynamic mapping are lower than that in static mapping, as there are data dependencies between the tasks in this application and, therefore, the heuristics are not work-conserving in static mapping, (ii) both the static and the dynamic versions show the same scalability when changing the number of threads, (iii) MTET-MET consistently shows slightly better performance and variability than the other pairs.

Figure 6 shows the box-plots with the results obtained for the *SparseLU* under the same configurations as before. From these results, we can conclude that: (i) the response time and variability given in dynamic mapping are, again, better than those given in static mapping (similar to the results obtained with the *Heat* application), and (ii) the response time and variability of the results achieved with MTET-X are lower than those achieved with MNTP-X in most of the results, as MTET selects the queue based on having the least execution time and therefore improves the work-conserving of the mapping. Overall, the TDG of this application is more irregular than that of the previous applications (no dependencies in *Axy* and a diamond-shaped TDG in *Heat*) and this seems to result in more noticeable difference among the results obtained with the heuristics.

From the experimental results under different applications and configurations, it can be summarized that the response time and variability of the results in the application without data dependency in the cases with dynamic and static

Fig. 4: Experimental results of the *Axy* application.

Fig. 5: Experimental results of the *Heat* application.

Fig. 6: Experimental results of the *SparseLU* application.

approaches are close to each other, while the results in the applications with data dependencies in dynamic mapping are smaller than those in static mapping.

7 Conclusions

High-performance computing can be applied in real-time systems to improve their ability to process real-time data. The main challenge in this regard is the competing aspects between HPC (performance and scalability, among others) and real-time computing (predictability and dependability, among others). In this scope, task-to-thread mapping in parallel real-time applications executing on multi-core platforms can be elaborated using heuristics to efficiently achieve their timing requirements while addressing the performance needs of the system.

This paper proposed a methodology for static scheduling in parallel real-time applications using the OpenMP framework based on recording the mapping of tasks to threads in real executions. This process is carried out based on different allocation and dispatching heuristics applied for task-to-thread mapping. Two minor contributions have also been made, including (i) an enhancement on the LLVM implementation of one of the heuristics in order to reduce the mapping overhead, and (ii) a new heuristic algorithm for the task dispatching process. Furthermore, the performance of the dynamic and static mappings in parallel real-time applications was evaluated using the kernels of three real-world applications under various configurations (e.g., different number of threads).

The evaluation results showed that in the application without data dependencies, the performance of the heuristics in dynamic and static mapping is similar to each other, while in the applications with data dependencies, their performance in dynamic mapping is better than that in static mapping. More important, variability in the dynamic mapping approach is lower than static mapping. This is a breakthrough in the research for real-time systems, which always considered static approaches to be more predictable; therefore, this work opens the door to further studies about dynamic mapping for real-time systems.

Based on the experiments, it can be concluded that the performance of nowadays real-time systems (e.g., complex cyber-physical systems) can be noticeably improved, as the application response time is usually reduced by providing more resources. In the application without data dependency (i.e., *Apsy*), (i) the heuristic algorithms, in the close binding, perform better than the spread binding and (ii) MTET-HET almost performs worse than the others. Furthermore, in the applications with data dependencies, especially in *SparseLU*, MTET-MET mostly gives lower response time and variability in dynamic mapping, compared to the others, and also in static mapping but in some cases.

As there is still a discussion in the real-time community about the use of dynamic or static mapping regarding the performance, timing predictability, and schedulability, the future work should evaluate the functionality of these mappings under a wider set of conditions, especially using artificial intelligence and data analytics applications, which need to process huge real-time data at very high speed. Additionally, this evaluation will be carried out under differ-

ent hardware platforms with higher parallel capabilities (e.g., RISC-V, Kalray MPPA) that have in turn been considered in real-time systems, as well as in clusters as those needed, e.g., in the adaptive optics applications that correct image distortions on-the-fly in big telescopes [12].

8 Acknowledgment

This work has been funded by the RESPECT project (Departament de Recerca i Universitats de la Generalitat de Catalunya record No 2021 PROD 00179).

References

1. LLVM OpenMP documentation (2024), <https://openmp.llvm.org/>
2. OpenMP - GCC Wiki (2024), <https://gcc.gnu.org/wiki/openmp>
3. Criado, J., Lopez, V., Vinyals-Ylla-Catala, J., Ramirez-Miranda, G., Teruel, X., Garcia-Gasulla, M.: Exploiting openmp malleability with free agent threads and dlb. In: International Conference on High Performance Computing. pp. 162–175. Springer (2022)
4. Fritz, F., Schmid, M., Mottok, J.: Accelerating real-time applications with predictable work-stealing. In: International Conference on Architecture of Computing Systems. pp. 241–255. Springer (2020)
5. Garcia, M., Corbalan, J., Labarta, J.: Lewi: A runtime balancing algorithm for nested parallelism. In: 2009 International Conference on Parallel Processing. pp. 526–533. IEEE (2009)
6. Gharajeh, M.S., Royuela, S., Pinho, L.M., Carvalho, T., Quiñones, E.: Heuristic-based Task-to-Thread Mapping in Multi-Core Processors. In: IEEE 27th International Conference on Emerging Technologies and Factory Automation (ETFA). pp. 1–4. IEEE (2022)
7. Gmys, J., Carneiro, T., Melab, N., Talbi, E.G., Tuytens, D.: A comparative study of high-productivity high-performance programming languages for parallel meta-heuristics. *Swarm and Evolutionary Computation* **57**, 100720 (2020)
8. Gómez, A., Alfonso, I., Coronel, J., Esclapez, M.D., Ferrer, J.: TRANSACT: Towards safe and secure distributed cyber-physical systems. In: Actas de las XXVI Jornadas de Ingeniería del Software y Bases de Datos (2022)
9. Melani, A., Serrano, M.A., Bertogna, M., Cerutti, I., Quinones, E., Buttazzo, G.: A static scheduling approach to enable safety-critical OpenMP applications. In: 2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC). pp. 659–665. IEEE (2017)
10. Munera, A., Royuela, S., Quiñones, E.: Towards a qualifiable openmp framework for embedded systems. In: 2020 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 903–908. IEEE (2020)
11. Pereira, R., Martin, M., Roussel, A., Carribault, P., Gautier, T.: Suspending OpenMP Tasks on Asynchronous Events: Extending the Taskwait Construct. In: International Workshop on OpenMP. pp. 66–80. Springer (2023)
12. Pou, B., Ferreira, F., Quinones, E., Gratadour, D., Martin, M.: Adaptive optics control with multi-agent model-free reinforcement learning. *Optics express* **30**(2), 2991–3015 (2022)

13. Royuela, S., Ferrer, R., Caballero, D., Martorell, X.: Compiler analysis for OpenMP tasks correctness. In: Proceedings of the 12th ACM International Conference on Computing Frontiers. pp. 1–8 (2015)
14. Samadi, M., Royuela, S., Pinho, L.M., Carvalho, T., Quiñones, E.: Time-Predictable Task-to-Thread Mapping in Multi-Core Processors. *Journal of Systems Architecture* **148**, 103068 (2024)
15. Serrano, M.A., Marín, C.A., Queralt, A., Cordeiro, C., Gonzalez, M., Pinho, L.M., Quiñones, E.: An Elastic Software Architecture for Extreme-Scale Big Data Analytics. In: Technologies and Applications for Big Data Value, pp. 89–110. Springer (2021)
16. Serrano, M.A., Melani, A., Vargas, R., Marongiu, A., Bertogna, M., Quinones, E.: Timing characterization of OpenMP4 tasking model. In: International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES). pp. 157–166. IEEE (2015)
17. de Supinski, B.R., Scogland, T.R., Duran, A., Klemm, M., Bellido, S.M., Olivier, S.L., Terboven, C., Mattson, T.G.: The ongoing evolution of OpenMP. *Proceedings of the IEEE* **106**(11), 2004–2019 (2018)
18. Verucchi, M., Olmedo, I.S., Bertogna, M.: A survey on real-time dag scheduling, revisiting the global-partitioned infinity war. *Real-Time Systems* **59**(3), 479–530 (2023)
19. Yu, C., Royuela, S., Quiñones, E.: Taskgraph: A Low Contention OpenMP Tasking Framework. *IEEE Transactions on Parallel and Distributed Systems* (2023)