Automatic, Abstracted and Portable Topology-Aware Thread Placement

Jens Gustedt, Emmanuel Jeannot, Farouk Mansouri

Automatic, Abstracted and Portable Topology-Aware Thread Placement

Jens Gustedt[†‡] Emmanuel Jeannot[*] Farouk Mansouri[*]

[†] *INRIA Nancy Grand Est, France*
[‡] *ICube – Université de Strasbourg, France*
[*] *INRIA Bordeaux, France*

***Abstract*—Efficiently programming shared-memory machines is a difficult challenge because mapping application threads onto the memory hierarchy has a strong impact on the performance. However, optimizing such thread placement is difficult: architectures become increasingly complex and application behavior changes with implementations and input parameters, *e.g* problem size and number of threads. In this work, we propose a fully automatic, abstracted and portable affinity module. It produces and implements an optimized affinity strategy that combines knowledge about application characteristics and the platform topology. Implemented in the back-end of our runtime system (ORWL), our approach was used to enhance the performance and the scalability of several unmodified ORWL-coded applications: matrix multiplication, a 2D stencil (Livermore Kernel 23), and a video tracking real world application. On two SMP machines with quite different hardware characteristics, our tests show spectacular performance improvements for these unmodified application codes due to a dramatic decrease of cache misses and pipeline stalls. A comparison to reference implementations using OpenMP confirms this performance gain of almost one order of magnitude.**

***Keywords*-Thread placement, Task based runtimes, Hardware affinity, Parallel programming.**

I. Introduction

The trend for an increasing number of cores in computing architectures leads to a significant increase in the internal complexity of machines. In particular, the cache architecture is now usually structured hierarchically between cores (*e.g* into sockets and processors) and a centralized memory topology for symmetric multiprocessor (SMP) system is replaced with distributed memory architectures such as *AMD Hyper Transport* and *INTEL QPI* architectures. Thus, in a HPC context we are nowadays more and more confronted with *N*on *U*niform *M*emory *A*ccess (NUMA) hardware.

Achieving high-performance in thread-based frameworks requires to place threads and data very carefully according to their affinities: sharing data and synchronizing threads benefits from shared caches, while intensive memory access benefits from localized memory allocations and accesses that are exclusive. Since this optimization is key, thread-based frameworks try to propose different levels of affinity abstraction. However, obtaining good hardware affinity usually requires an in-depth knowledge of the underlying architecture as well as the application behavior. Obtaining the affinity between threads is a difficult task that often requires the programmer to manually map threads onto resources. The goal of this paper is to show that, with the right abstraction, it is easy, for the runtime system to compute an efficient thread mapping. Hence the contribution of this paper is to propose an automatic (no need for the programmer to specify anything: the code is unchanged), abstracted (no need for the programmer to understand what is affinity and topology) and portable (no need for the programmer to specify any configuration based on the target machine: performance is portable across different architectures) affinity module for thread-based runtimes. Transparent to the user, our module computes and enables an optimized binding strategy that takes the hardware topology and the application characteristics into account. Absolutely no modification of the application and no tuning on the hardware is required. As a proof of concept, the module is implemented as an affinity add-on of static thread-based runtime system named the Ordered Read Write Location (ORWL)

This paper is organized as follows. In Section II, we present related work about affinity for thread mapping. After that, Section III describes the context and the background of our work including tools and frameworks we use. In Section IV, we introduce the affinity module that connects knowledge about application and platform structure and explain its implementation. Implementations of two benchmarks and a real-world application based on our ORWL framework are presented in Section V. Section VI presents measurements of these benchmarks that prove how our affinity module helps to improve the benchmark performance by a substantial factor up to 9x without changing a line of code in the benchmark code itself or reconfiguring the execution. These validations are complemented with a comparison to reference implementations based on OpenMP for parallelization and affinity optimization. Finally, Section VII summarizes our achievements and discusses them.

II. Related Work

Thread based programming models like OpenMP propose high level interfaces to optimize thread affinity. OpenMP uses environment variables to enable binding strategies. For example, by setting `KMP_AFFINITY` with Intel's implementation of OpenMP it is possible to adapt a program

execution to the targeted topology and to choose the cores on which threads are executed. The recent versions of the GCC runtime system (GOMP) implements similar functionality through the variable `GOMP_CPU_AFFINITY`. These vendor-specific approaches for affinity modules are now ported into the last OpenMP standard 4.5. It specifies the interfaces `OMP_PLACES` and `OMP_PROCBIND`. All these interfaces are at a high level and relatively easy to use by inexperienced programmers. However, they are not aware of the actual application behavior when they decide where to bind each thread. Moreover, the proposed strategies are highly generic. Hence, as we will show in our experiment section, a strategy may be efficient for one machine and inefficient for another. Also, it is difficult to anticipate, without testing, which strategy will behave best.

In [1], the authors propose an extension based on a *location* directive to provide affinity of OpenMP threads and data. A more recent work [2] proposes a novel OpenMP directive to bind OpenMP tasks to threads, NUMA nodes or close to some data. However, in these solutions the programmer still needs to manually insert specific directives to match the graph of tasks and the architecture topology. In addition, they still have to investigate the application behavior to find the correct binding strategy. Neither the OpenMP standard nor existing thread-based runtimes currently allow mechanisms to automatically produce and set an adapted affinity strategy of threads. An other issue is that, each time the application code changes, the directive has to be adapted or modified to ensure performance gain and to preserve the sequential semantic. OmpSs [3] is a variant of OpenMP that targets heterogeneous architectures. However, as it is based on the OpenMP model it inherits the problems of general OpenMP solutions as we have highlighted above.

To provide thread affinity and data locality in thread based runtimes other work focuses on scheduling. [4] introduces locality for the OpenMP task-scheduling by extracting dependency information at compile time. [5], [6] use task and buffer placement to improve the data locality of the scheduler of a data flow graph tool named OpenStream. These approaches are well oriented for dynamic runtimes that distribute fine grained tasks over worker threads. However, they are not adapted for applications with a limited number of tasks and a coarse granularity. Here, dynamic scheduling could be not efficient because of granularity and generates unnecessary overhead. Also, many applications such our video tracking implementation (see below) require static scheduling. Here, each task must be bound on a specific core, such that the execution is ordered and the provision of video frames is regularly sequenced.

Task based runtime systems such as StarPU [7], or PARSEC [8] are also related to this research. In StarPU, several scheduling policies are available[1]. As StarPU targets heterogeneous system, the proposed strategies are dynamic and based on the current state of the runtime system. The default StarPU scheduler, called *eager*, does not take task affinity into consideration. The locality-aware work stealing algorithm (*lws*) dynamically schedules tasks on the worker which releases it, trying to enforce some kind of locality management. However, it does not use data dependencies and data sharing. PARSEC features affinity management by enforcing strict owner-compute rules. Results are very competitive but this requires the user to map of the data (e.g. distribution of a matrix) explicitly onto the resources and also to describe tasks dependencies. An approach to compute the data mapping automatically has been proposed by a one of the co-authors of this paper [9]. However such approaches are bound to the parameterized task graph model and are only suited for codes with static control if there are affine access and loop bounds. Therefore many applications do not fit into this model.

Hence, to the best of our knowledge, there are no other solutions that automatically provide binding mechanisms in a static thread-based runtime, that are independent of the target machine, that are portable and that do not require modifications of the application code.

III. Background and Context

The ORWL model and library: The ORWL "Ordered Read-Write Lock" programming model [10] is a programming concept for the management of shared resources in a parallel environment: data, storage spaces, levels of cache or the I/O devices. These resources are abstracted in the ORWL model by the notion of *locations* which are used for sharing control of such resources between tasks. The model presents the concurrent access to a resource/location by using a FIFO that holds requests (requested, allocated, released) issued by the tasks. These tasks are implemented by threads. The FIFO controls the access order and locks and maps the resource for some threads either exclusively (for a writer) or shared (for a set of readers). The reference implementation of ORWL offers several abstractions in the form of a C-based library such that parallel applications can easily be expressed. The following primitives are available:

- **orwl_task** is a primitive the programmer should use to decompose their application. Such tasks only interact via their access to locations.
- **orwl_location** is the primitive to represent a shared resource between the tasks. It could be data (identical contents at varying memory addresses), memory (a specific address), a computational unit (CPU or accelerator) or an I/O device.
- **orwl_handle** implements a primitive to link the locations to the appropriate tasks with read or write access.
- **ORWL_SECTION** defines a critical section that manages the access of threads to the location (resource). Once entered in such a section, the task that requested

[1] See http://starpu.gforge.inria.fr/doc/html/Scheduling.html

read or write access obtains the data concurrently or exclusively.

Listing 1. ORWL: pipeline of tasks

```
/* define the name of the location's buffer*/
ORWL_LOCATIONS_PER_TASK(main_loc);
orwl_init();
/* Scale our own location(s) to the appropriate size. */
orwl_scale(sizeof(double));
/* Create handles for the locations that we are
interested in. We  will create a chain of dependencies
from task 0 to task 1 etc. */
orwl_handle here = ORWL_HANDLE_INITIALIZER;
orwl_handle there = ORWL_HANDLE_INITIALIZER;
/* Have our own location writable. */
orwl_write_insert(&here,
                  ORWL_LOCATION(orwl_mytid, main_loc),
                  orwl_mytid);
/* link the "there" handle where appropriate */
if (orwl_mytid)
  orwl_read_insert(&there,
                   ORWL_LOCATION(orwl_mytid - 1, main_loc),
                   orwl_mytid);
/* Now synchronize and coordinate requests of all tasks. */
orwl_schedule();
/* All tasks create a critical section that guarantees
exclusive  access to their location. */
ORWL_SECTION(&here) {
  /* Map the buffer in our virtual address space. */
  double * wval = orwl_write_map(&here);
  *wval = init_val(orwl_mytid);
  /* All ids > 0 read from their predecessor. */
  if (orwl_mytid > 0) {
    /* Block until the data is available. */
    ORWL_SECTION(&there) {
      double const* rval = orwl_read_map(&there);
      /* Do some dummy computation. */
      *wval = (*rval + *wval) * 0.5;
    }
  }
}
```

Listing 1 shows an example of an ORWL program with a pipeline of tasks. There are two locations per tasks: "`there`" where it reads from the previous tasks and "`here`" where it writes locally.

In addition, the library proposes some specific primitives to easily implement iterative tasks that alternate access to a shared resource that is represented by a (**orwl_location**). In this case, they can use an adapted iterative handle **orwl_handle2**. To synchronize the iterations of the different tasks the programmer disposes of **ORWL_SECTION2**. Before its termination, such a section introduces a new query in the FIFO that requests the resource for the next iteration. Thereby, each task may run a series of iterations that are autonomously synchronized by their access to the resource. This iterative access guarantees the consistency of data, deadlock-freeness and fairness for the decentralized event-based execution. Thanks to these primitives users may express a high level of parallelism within applications while avoiding the manual use of a low-level C interface to manage lock synchronization and communication between threads.

The HWLOC library for locality management: The hardware locality tool "HWLOC" [11] exposes a portable (across OS and architecture) an abstracted view of the hardware topology to the developer and the runtime system. HWLOC provides a library that extracts the topology of a NUMA machine and allows for exploring it. For instance, it is possible to know the cache hierarchy, the different cache sizes, the number of cores with their numbering. It also provides a way to bind threads to (a set of) cores as well as to query and change the binding.

TreeMatch: TreeMatch [12] is a library for performing process placement based on the topology of the machine and the communication pattern of the application. TreeMatch provides a mapping of the processes to the processors/cores in order to minimize the communication cost of the application. According to the problem size, it chooses between different placement algorithms such that a low running time is maintained.

IV. Abstracted affinity add-on for ORWL thread-based runtimes

A. Concept

As described above, ORWL is a resource-centric manager. It enables to construct a set of application tasks and concurrently executes them according to their FIFO access to the resources. To ensure consistency throughout an event-based execution, the ORWL runtime additionally deploys control threads and a lock mechanism that manage lock synchronization and data transfer. These control threads freeze and thaw processing threads of concurrent tasks according to the availability of resources. Thereby, the model of execution is highly decentralized.

The relationship between application tasks and OS threads can be modeled in different ways:

- *One thread per task:* Each task is executed with by one OS thread and can access several locations.
- *Several threads/operations per task:* Here several subtasks named *operations* cooperate to execute a task. Each operation is executed by an OS thread and will typically be responsible for one location of the task. Thus, here a task is executed by as many threads as there are locations.

Our aim is to propose a placement strategy that optimizes data locality. To do so, we exploit application information as it is gathered from ORWL primitives, namely the topology of the task-location graph and the sizes of the locations. We automatically compute the task/thread affinity using information about shared locations and their FIFO when the runtime system instantiates and composes them. The ORWL programming model exposes all the required pieces of information: the tasks, the amount of data they share or exchange (i.e the location) and their connectivity (i.e. the location they share). Therefore, there is no need to modify the code or to add any directive to gather that information. Indeed, in Listing 1, when the **orwl_schedule** function is called, all these characteristics are known to the ORWL runtime system. Thereby, when this function is called, we

are able to construct a matrix (see Fig. 1) that expresses the communication volume between tasks and then to compute the mapping.

At the other end we use HWLOC to obtain the topology of the underlying platform in an automated and portable way. With these two types of information we apply an allocation strategy that aims to reduce the communication between the NUMA nodes. Simultaneously, it optimizes the cache sharing within each of these NUMA nodes.

To compute the allocation we use Algorithm 1 that is based on the TreeMatch Algorithm [12]. We have adapted it in two ways for our needs. First, we have enhanced it to account for over-subscription when there are fewer computing resources than tasks. For compute-bound application, it is generally better not to exceed the available resources by dimensioning the application to the number of physical cores (this what we have systematically done in all our experiments here). Alternatively, some applications may have a minimum requirement for the number of tasks (and thus threads) that exceeds the number of resources.

A second adaptation takes control threads of ORWL into account. This algorithm depends on the available computing resources, in particular on the presence of hyperthreaded cores.

Algorithm 1: The Mapping Algorithm

```
   Input: T// The topology tree
   Input: m // The communication matrix
   Input: D // The depth of the tree
1  m ← extend_to_manage_control_threads(m)
2  T ← manage_oversubscription(T,m)
3  groups[1..D − 1]=∅ // How nodes are grouped on each level
4  foreach depth← D − 1..1 do // We start from the leaves
5      p ← order of m
6      groups[depth]←GroupProcesses(T,m,depth) // Group
         processes by communication affinity
7      m ←AggregateComMatrix(m,groups[depth])        // Aggregate
         communication of the group of processes
8  MapGroups(T,groups) // Process the groups to build the
     mapping
```

Algorithm 1 is run at launch time, once the topology tree is given by HWLOC and the communication matrix is computed by the ORWL runtime system. It provides a mapping of the computing entities (the threads) to the cores. These threads are then bound to the cores using HWLOC. It proceeds as follows. First, depending on the topology tree and the presence of hyperthreading we optionally extend the communication matrix to account for control threads. If hyperthreading is available, on each physical core we reserve one hyperthread sibling for control and one for computation. Otherwise, if there are more cores than tasks, we extend the communication matrix such that control threads will be mapped onto spare cores. If none of this suffices, control threads will not be mapped explicitly and we let the system schedule them. Second, we check if oversubscribing is needed. We compare the number of leaves of the tree with the order of the communication matrix. Optionally, we add a new level to this tree such that we have enough virtual resources to compute the allocation.

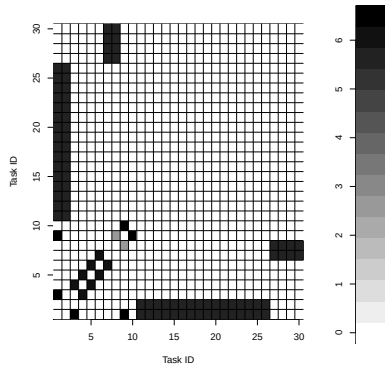


Figure 1. Communication matrix of the video tracking application (see Sec. V-C) – logarithmic gray scale

Then, computing entities of the communication matrix (being computation threads and optionally control threads) are grouped according to their affinity and the topology of the machine starting from the leaves of the topology tree. At the upper levels these groups are merged recursively. The function `GroupProcesses` makes k groups of size a, where a is the arty of the considered level and such that $a * k = p$. Here, p is the order the communication matrix and hence the number of processes or groups.

The internal algorithm engine of `GroupProcesses` is optimized such that, depending on the problem size, we go from an optimal but exponential algorithm to a greedy one that is linear[2]. Before going from depth l to $l - 1$ we need to aggregate the communication matrix in order to compute the affinity between the groups. This is done by the function `AggregateComMatrix`. Once we have built this hierarchy of groups, we match it to the topology tree such that each thread is assigned to a leaf (function `MapGroups`). If oversubscribing is required, ORWL tasks are mapped to the physical cores by going up one level in the tree. If hyperthreading is available, we map only one compute intensive task per physical core, and leave hyperthreaded sibling cores to control threads.

Fig. 1 illustrates the communication matrix of the video tracking application used in Sec. V-C. Thread ID, correspond to the different tasks of the application as coded using ORWL and given by the ORWL runtime. Once Algorithm 1 is applied, we obtain the mapping of the tasks given in Fig. 2. The machine is similar to the one used in Table I. Each task has a green box with the ID corresponding to the one used in Fig. 1 and its name. We see that most of the pipeline Tasks 0 to 9 are placed on the same socket

[2]Hence, the runtime overhead is kept negligible for current SMP machines.

whereas Tasks 1 and 7 are mapped to another node as they communicate with other tasks (resp. `gmm split` and `ccl split`). Finally, cores 22 and 23 are automatically reserved for control threads.

B. Implementation

Our affinity module is implemented as an add-on to the ORWL framework. To enable the affinity optimization with the *fully automatic and transparent mode*, the ORWL user only has to set the environment variable `ORWL_AFFINITY` to `1`. This variable is checked at runtime (initialization time of ORWL) and the appropriate affinity for threads is computed and set behind the scenes as described above. Moreover, for experimented developers preferring the *advanced mode* we added some affinity functions to ORWL's API. This API is useful in two different settings: 1) debugging and testing, 2) to handle dynamic situations where the number of tasks and the affinity between tasks change at run time, that is if the communication matrix is changing dynamically.

- **orwl_dependency_get**: compute task dependencies of the application and the resulting communication matrix for the underlying threads.
- **orwl_affinity_compute**: compute the optimized thread mapping from the current communication matrix and the hardware topology.
- **orwl_affinity_set** : set the biding of each thread according to the computed mapping.

None of the functions of that API take parameters or return values, they only change the internal state of the ORWL runtime.

When using this API in addition to the automatic computation at startup, the thread mapping is still automatic: once the connection between tasks and location has changed in the program, the new affinity is computed by explicitly calling **orwl_dependency_get**, then, Algorithm 1 is called with **orwl_affinity_compute** and the new computed thread mapping is committed with **orwl_affinity_set** .

V. Benchmarks and Application

In this section we present applications implemented to validate our affinity module within the ORWL runtime. In order to test the performance of our approach in different contexts, we use, on the one hand two benchmarks with different characteristics: matrix multiplication as an example of compute bound problem and the Livermore Kernel 23 which is rather memory bound. On the other hand, we use the HD video tracking application as a real-world validation.

A. Livermore Kernel 23

The Livermore Kernel 23 is a classic benchmark taken from LinPack [13] to simulate a 2-D implicit hydrodynamics fragment. The core computation of the benchmark is given in Listing 2 where each element of the matrix called `za` is computed using four neighbors elements (`N`, `S`, `E` and `W`) and five coefficient matrices (`zb`, `zr`, `zu`, `zv`, `zz`). In addition, a global loop repeats this computation for a certain number of iterations or until convergence.

This algorithm is memory bound and is difficult to vectorize because of the loop structure. Usually it is parallelized by pipelining the computation over blocks of the initial two-dimensional data matrix (starting from the upper left block down to the lower right one).

Listing 2. The Livermore Kernel 23 loops

```
for (l=1; l<=loop; l++)
  for (j=1; j<m; j++)
    for (k=1; k<n; k++){
      qa =   za[j+1][k]*zr[j][k] + za[j-1][k]*zb[j][k]
           + za[j][k+1]*zu[j][k] + za[j][k-1]*zv[j][k]
           + zz[j][k];
      za[j][k] += 0.175*(qa - za[j][k]);
    }
```

The blocks on the same diagonal can be computed in parallel. The wave of computation therefore traverses the matrix `NW` to `SE`.

In ORWL, the idea for this 2D stencil is to have a task for each matrix block of matrix `za`, and a location between each pair of communicating tasks (i.e. each communicating block of `za` in the code). The detailed ORWL implementation is described in [14].

B. Matrix multiplication

Dense matrix operations are important elements in scientific and engineering computing. Matrix multiplication has been largely studied for high performance computation. In our case study, we focus on computing $C = A * B$ with a row aligned matrix. For our implementation we use the well-known block cyclic algorithm. It consists of dividing the matrices `A` and `B` into blocks which are processed in parallel during a number of phases. During these phases, the matrix blocks circulate between tasks.

Internally, to compute the multiplication of the blocks we use the `DGEMM` from BLAS library interface. Using different BLAS implementations is easily possible and has been tested. For all the reported tests, we use Intel's MKL implementation of BLAS, as it provided the best results.

The MKL library also features a multithreaded version of `DGEMM` using OpenMP. We compare our approach with this MKL version by varying the number of threads and the binding strategies.

The ORWL implementation: In our ORWL implementation each block of rows of the result matrix `C` corresponds to a task/thread using the **orwl_task** primitive. A task processes the elements of a block of rows of the matrix `C` and circulates the input columns of the matrix `B` to the neighboring tasks by using ORWL's "locations". Each task is connected to its own location (current block of columns of `B`) and to the location of the precedents tasks (next block of columns of `B`) by using the **orwl_handle** primitive. The result blocks of the matrix `C` are then computed with a complete circulation of the input columns of matrix `B`.

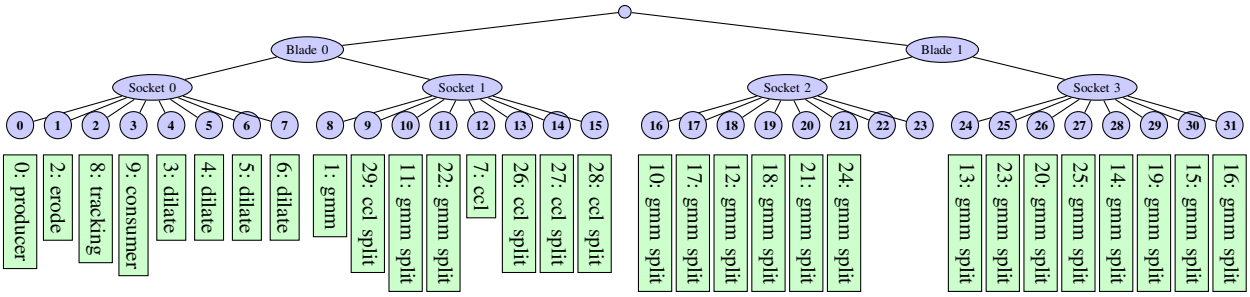


Figure 2. Task allocation on 4 socket NUMA machine of the video tracking application (see Sec. V-C). Note that, for space reason, we do not describe the cache hierarchy, while this hierarchy is also given by HWLOC and is actually used by our algorithm for the mapping.

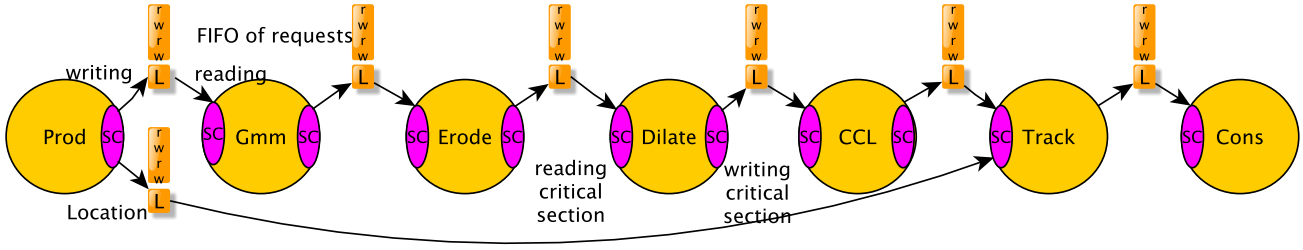


Figure 3. Illustration of the ORWL implementation of the DFG video tracking application. Yellow nodes represent task, purple regions represent read/write **ORWL_SECTION**. Each task is connected to locations (orange squares) using handles. In the experiments, Dilate is repeated 3 times. GMM and CCL are split into 16 and 4 sub-tasks (see Fig. 2 for the mapping on a 32 cores systems).

C. HD Video Tracking

The video tracking application follows moving objects as they are seen by several cameras. Recently this type of application has become important for video surveillance of public spaces or for traffic control. To track moving objects in a video, several algorithmic approaches have been explored [15]. In our study, we are interested to process high definition video with a tracking algorithm that detects the motion with a foreground-background extraction technique [16].

The data-flow graph ORWL implementation: The application is iterative with repetitive processing applied on each frame. This is usually modeled as a synchronous data-flow graph (DFG) [17] as shown in Fig. 3. The nodes of the graph represent the functions of the algorithm. The edges represent the exchange of data between functions through FIFO channels. This model expresses pipeline parallelism where each function can be executed as soon as its input data are available.

As shown in Section III, the ORWL model allows programmers to easily decompose iterative applications as dependent tasks. We implement the DFG model in ORWL by representing each node of the graph by an iterative **orwl_task** processing the input data at each iteration. To manage dependencies between tasks in the ORWL model we use the **orwl_location** and **orwl_handle2** primitives. Each task has its own "location" connected by a write handle for the output data. With read handles, it connects to the location/data of preceding tasks that it needs to perform its computation. Then, during processing a critical section is used to reserve the own location exclusively and to recover the input data from the "locations" of the preceding tasks.

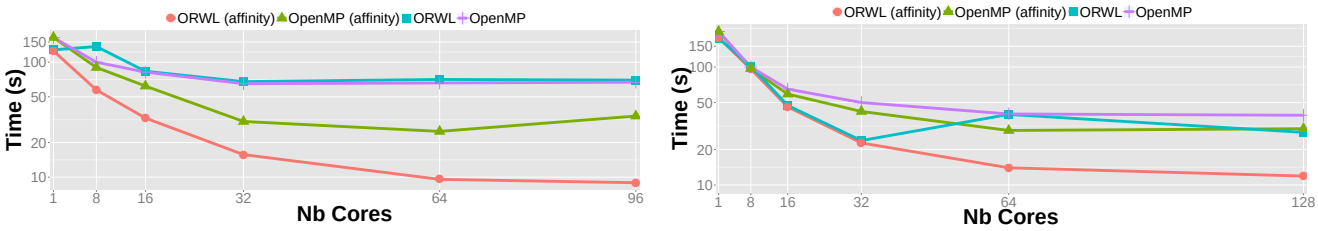
In addition, we add some DFG-specific features. An **orwl_fifo** primitive is used to store a new version of output data intermediately such that the lock for other readers/writers can quickly be released. An **orwl_split** primitive helps to split the data of a location into several pieces that can be processed in parallel by other tasks or operations. Here, the latter is used to split the tasks `GMM` and `CCL` because these two are the most expensive and form bottlenecks for the pipeline, see Fig. 3.

The ORWL tasks are executed in parallel by different threads. Our implementation allows to process multiple input images concurrently: we exploit task parallelism by a pipeline and data parallelism with split-merge.

VI. Experiments and Results

A. Testbeds and architectures

We use two SMP machines from the PlaFRIM platform. Their characteristics are described in Table I. The SMP12E5 platform is a newer generation than SMP20E7. It features hyperthreading and enables performance counters.



(a) SMP12E5 (b) SMP20E7

Figure 4. The processing times of Livermore Kernel 23 (log scale)

Table I
THE MULTI-CORE ARCHITECTURES USED FOR THE EXPERIMENTS

Name	SMP12E5	SMP20E7
OS	Red Hat 4.8.3-9	SUSE Server 11
Kernel	3.10.0	2.6.32.46
Cores per socket	8	8
NUMA nodes	12	20
Socket per NUMA	1	1
NUMA groups	12	20
Socket	E5-4620	E7-8837
Clock rate	2600MHz	2660MHz
Hyper-Threading	Yes	No
L1 cache	32K	32K
L2 cache	256K	32K
L3 cache	20480K	24576K
Memory Interconnect	NUMAlink6 (6.5GB/s)	NUMAlink5 (15GB/s)
GCC	5.1	5.1
ICC/MKL	14.0/11.1	14.0/11.1

B. Experiments

In this section we will assess the performance difference from enabling our affinity module in the benchmarks we present in Section V. Our goal is not to compare different programming models but to compare different approaches of managing affinity and binding thread to cores. Therefore, on the one hand, we compare the performances of the ORWL implementations based on affinity optimization to the native ORWL implementations. On the other hand, we also compare the results achieved with our module against the best one that is achieved for the same application coded in OpenMP and using different optimizations. We experimented several OpenMP placement strategies but, due to lack of space, we only report those with the best performance. We use OpenMP as a reference because it is the most widely used thread-based programming model that proposes an abstracted thread placement module. We intend to prove that it is possible to simultaneously get better abstraction and performance compared to OpenMP. In addition to the performance results, we present measurements of hardware and software counters collected with the benchmarks to explain the differences.

1) Livermore Kernel 23: Fig. 4 shows the execution times of 100 iterations of Livermore Kernel 23 implementations that process a 16384x16384 matrix of double precision elements on scalable hardware configurations. Each block of the matrix is processed by several operations: 1 for computing central block and 3 for updating borders with neighbourhood blocks. Each operation is executed with one thread so we use 4 threads for each block except for the run with 1 block. The OpenMP implementation is equivalent to the ORWL implementation described in Section V. It is based on introducing `#pragma parallel for` directives with static scheduling of chunks over the threads. Both implementations use the same number of thread for each run and we map each thread on one core.

The non-optimized ORWL and OpenMP implementations scale until 16 cores. After that they perform badly and stabilize at about 65 seconds for SMP12E5 and about 40 seconds for SMP20E7. We experimented many different optimization settings for OpenMP. Here, the *"OpenMP (affinity)"* line corresponds to the best performance that was achieved with `OMP_PLACES=cores` and `OMP_PROC_BIND=close/spread` (both implementations giving the same results): they slightly enhance the performances up to 1.3x on SMP20E7 and about 2.5x on SMP12E5. Setting `close` or `spread` for `OMP_PROC_BIND` means that the threads are bound to cores either as close as possible to the master thread or scattered across the available cores. In none of these cases, the topology or the thread affinity are used to compute the mapping. In contrast to that, our solution takes the topology of the machine, the availability (or not) of hyperthreads, and the affinity between tasks[3] into account. In this case, our affinity modules scales even more than the OpenMP affinity setting and reaches about 3x on SMP20E7 (without hyperthreading) and about 8x on SMP12E5 (with hyperthreading).

We also studied the hardware and software counters of the machine for a 64 cores run, see Table II. There is a strong correlation between cache misses and cycle stalls: each cache miss leads to a loss of about 10 to 14 cycles. Our affinity management reduces these counters by a substantial factor. Compared to OpenMP Affinity we have -78% cache misses and -72% stalled cycles. On the other hand the ORWL approach generates much more context switches, but their total number is still 5 orders of magnitude lower than

[3]tasks that share a location are mapped close to each other

Table II
ACCUMULATED HARDWARE/SOFTWARE COUNTERS FOR LIVERMORE KERNEL 23 ON SMP12E5 (64 CORES)

	ORWL	ORWL (Affinity)	OpenMP	OpenMP (Affinity)
Billions of L3 misses	81	14.2	81	64
Billions of stalled cycles front-end	840	200	840	720
context switches	99 778	89 151	745	210
CPU migrations	15 960	0	203	0

for the other counters.

We explain the bad performances of the native implementation, when they use more than one socket, by the scheduling policy performed by the respective systems. In fact, threads are dynamically placed onto cores of the machine with different policies: the system of the SMP12E5 (with Linux 3.10) tries to reduce the number of used NUMA nodes by even using the hyperthreads, while the scheduler of the SMP20E7 (Linux 2.6.32) spreads threads evenly over the 20 NUMA nodes of the machine. This explains the performance gain of our module as we are better in managing locality and memory accesses by taking into account task affinity.

On the other hand, because ORWL generates a lot of control threads to manage access to the locations, the number of thread migrations and of context switches is much higher for ORWL than for the others. However, this seems not to impact the performance. On modern Linux systems a context switch has a cost of about 100 ns. Hence, around 100 000 context switches that are spread over 64 cores correspond to an overhead of fewer than 2 ms. This is negligible compared to the overall runtime.

We also see that CPU migration is reduced to 0 when enabling the affinity strategies (both for OpenMP or ORWL) as we have a strict binding of the threads to the cores. The lack of performance difference between the non-affinity versions, while ORWL exhibits much more migrations, can be explained by the fact that these migrations mainly concern control and management threads and not the compute threads implementing the tasks.

2) Matrix multiplication: Fig. 5 presents performance comparisons of several implementations multiplying two 16384x16384 matrices of double precision elements.

For a reduced number of cores inside a socket, all native implementations (without affinity) scale. The 3 MKL implementations (one without affinity and 2 with scatter or compact affinity) are slightly better than the two ORWL implementations. With 8 cores, they reach about 95 Gflop/s on SMP12E5 and about 65 Gflop/s on SMP20E7. However, with more than 8 cores, so more than one socket, the performance deteriorates for all implementations and stops scaling before using all cores of both architectures. Indeed, the MKL implementation does not take the topology of the machine into consideration when mapping threads. For instance, on the hyperthreaded machine (SMP12E5), the compact strategy tends to use the cores that are closest to the master thread. Hence, it maps two compute threads

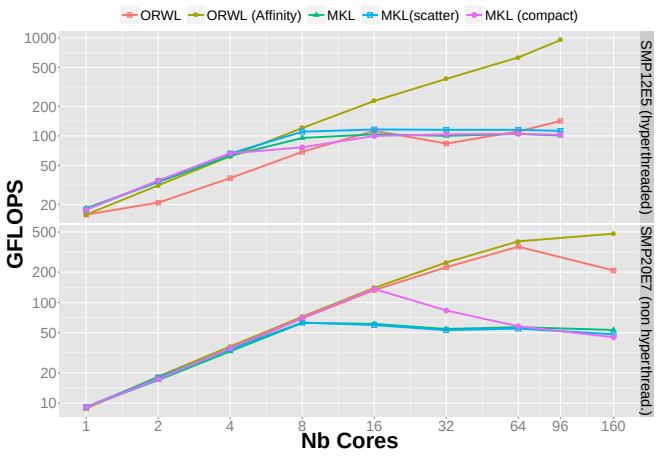


Figure 5. FLOP/s performances of the Matrix multiplication implementations, x and y axis are in a logarithmic scale.

Table III
ACCUMULATED HARDWARE/SOFTWARE COUNTERS OF MATRIX MULTIPLICATION ON SMP12E5 (64 CORES)

	Billions of L3 misses	Billions of stalled cycles	CPU mig.	Context sw.
ORWL	102	8110	28963	153 265
ORWL (Affinity)	13.8	980	0	125 368
MKL	140	8850	486	2 863
MKL (Affinity scatter)	99	8140	0	2 750
MKL (Affinity compact)	89	8520	0	3 001

on two hyperthread siblings. Since we are executing a computation-bound kernel these solutions are worse than the scatter strategy. By enabling our affinity module (that takes characteristics of the target machine and the thread affinity into account to compute an optimal placement based on grouping threads executing neighbour tasks), the performance of all ORWL implementations is enhanced without changing any line of code. It reaches a maximum of 1 Tflop/s on SMP12E5 and 0.5 Tflop/s on SMP20E7 using all cores of the machines. In contrast to that, the MKL implementations based on affinity optimization[4] stagnate and do not improve the performances.

Again, we have gathered hardware and software counters for the 64 cores run on the SMP12E5 machine, see Table III. We see that the ORWL implementation considerably reduces the number of L3 cache misses and pipeline stalls compared to the affinity-based MKL implementation. In contrast to that, the number of CPU migrations and context switches are considerably higher for ORWL than for the others.

These results can be consistently explained as above: our management of locality leads to much improved execution times, due to reduced cache misses and stalls. Again much higher numbers for thread migration and context switches do not influence execution times much. On the other hand,

[4] `KMP_AFFINITY=granularity=core,compact` and `KMP_AFFINITY=granularity=core,scatter`

Figure 6. FPS (logarithmic scale) of HD video tracking

the MKL implementation once it is using more than one socket, is not able to manage data efficiently.

Another interesting observation is that the compact strategy outperforms the scatter strategy for 16 cores, that is when using two sockets. Hence, we conclude that the compact strategy enforces a better locality by keeping threads closely together. For more than 16 cores (and more than 2 sockets) both strategies have similar performance. This shows that the compact strategy is not sufficient to overcome the lack of awareness for affinity. However, on 8 cores, we see that the scatter strategy is slightly better on the SMP12E5 while the compact strategy is slightly better on the SMP20E7 machine meaning that tuning performance with these parameters does not lead to portable improvements.

3) Video tracking: Fig. 6 shows the produced frames per second (FPS) of several implementations of the video tracking application on SMP12E5 and SMP20E7 architectures. The implementations we use are based on 30 tasks/threads to process 3 video resolutions: HD (720x1280 pixels), Full HD (1920x1080 pixels), and 4K (3840x2160 pixels). As video tracking is a streaming application, the aim is to accelerate the FPS in a hardware restricted environment. Thus, we use only 4 sockets (30 cores) of the architectures. The OpenMP implementation uses fork-join in each stage of the image processing pipeline by introducing `#pragma parallel for` with static scheduling of chunks.

We see that the ORWL affinity implementation based on our module enhances the performance of the native implementations. In fact, it accelerates by about 4.5x on SMP12E5 and about 2.5x on SMP20E5 without any code re-factoring or modification. However, we tried many OpenMP implementation and the affinity version based on `OMP_PLACES`. Only the best solution is shown here and it does not produce a comparable performance enhancement. It accelerates the FPS by about 2x on SMP12E5 and 1.5x on SMP20E7.

Table IV
ACCUMULATED HARDWARE/SOFTWARE COUNTERS OF VIDEO TRACKING ON SMP12E5 (30 CORES, HD VIDEO)

	ORWL	ORWL (Affinity)	OpenMP	OpenMP (Affinity)
Billions L3 misses	158	49	151	120
Billions of stalled cycles front-end	160	83	840	660
context switches	413821	329263	99778	22241
CPU migrations	61390	0	15960	0

Again, the ORWL affinity mapping (see Fig. 2) by taking the whole ecosystem into consideration is able to produce much better and portable performance than approach that do not take these characteristics into consideration. To assess the difference of the performance for our affinity optimization, we present some hardware and software counters in Table IV. Here again, we see that the affinity optimization produced by our strategy allows for significantly decrease the cache misses of the ORWL implementation and the CPU stall time. In contrast to that, the OpenMP affinity interfaces do not significantly decrease these counters.

These performance results are again consistent with or interpretation that our affinity module improves locality of data access substantially. We also see that for ORWL, the improvement is even greater on the SMP12E5 (with hyper-threading) than on the SMP20E7 (without) while the opposite holds for the OpenMP version. This validates our strategy to map all the threads of a task to the same physical core, such that they can share caches. The potential of the architecture is then best exploited by assigning one of the two hyperthreaded cores of the same physical core to the computation thread and the other to the control threads.

VII. CONCLUSION AND DISCUSSION

In this paper, we presented an affinity module for thread-base computations in a resource-centric runtime system. Our module improves the software to hardware mapping based on automatically extracting and matching communication behavior and hardware topology. Thanks to this module, users get full abstraction of the affinity of codes and the performance portability to the architecture. Application writers do not need to worry about the architecture complexity by investigating its topology and characteristics or to profile their application to extract the computation and communication behaviors.

In Sections VI, we experimented our approach on 3 applications: a Livermore Kernel 23 benchmark, block cyclic matrix multiplication and a real world HD video tracking application. We used 2 multi-core architectures with different characteristics. In all these cases, we show that our placement approach enhances the performances of the native ORWL implementation and allows for getting the maximum potential out of machines with a good scalability. In addition, it outperforms other non-topology-aware approaches whereas we tried many different locality optimizations. Indeed, as soon as we scale beyond one or two sockets,

standard approaches fail to improve performance because they are unable to take affinity and topology into account. Interestingly enough, we have observed that, in contrast to our approach, for OpenMP, the optimizations with the best performance are application specific. Moreover, when we move from a target architecture not featuring hyperthreading to a target featuring hyperthreading, the proposed gains are even more substantial showing that our approach takes the most benefit of the available resources. Hence, our approach is oblivious of the target architecture. To explain the gain we have monitored hardware and software counters. For each of the studied application they exhibit the same behavior, namely a pronounced decrease of L3 cache misses as well as stalled cycles with our strategy. This shows that our affinity strategy enables the same low-level optimizations on all these applications.

The keys of success of our strategy are the following. First, we take structural information how threads share data into consideration. Then, we also pay attention to the execution platform (topology, presence or absence of hyperthreading, etc.). Last, we abstract these characteristics in the programming model to achieve portable performance. Our goal was not to compare programming interfaces. Our principal contribution is to show that, implementing an efficient, portable, automatic and abstracted thread placement module is achievable as soon as the programming model exhibits the right abstraction and the runtime system implements a good mapping strategy. Therefore, we believe that the proposed approach is generic and can be integrated in other runtime systems as soon as the programming model provides the necessary abstraction: expressing the data shared by threads.

ACKNOWLEDGMENT

This work was funded by the Inria IPL *multicore*. The experiments presented in this paper were carried out using the PlaFRIM experimental testbed, being developed under the Inria PlaFRIM development action with support from Bordeaux INP, LaBRI and IMB and other entities: *Conseil Régional d'Aquitaine*, *Université de Bordeaux* and CNRS (and ANR in accordance to the *programme d'investissements d'Avenirs*, see https://www.plafrim.fr/).

REFERENCES

[1] L. Yi, *Enabling Locality-aware Computations in OpenMP*. University of Houston, 2011. [Online]. Available: https://books.google.fr/books?id=oJJwMwEACAAJ

[2] P. Virouleau *et al.*, "Description, Implementation and Evaluation of an Affinity Clause for Task Directives," in *IWOMP 2016*, Nara, Japan, Oct. 2016.

[3] J. Bueno *et al.*, "Productive programming of gpu clusters with ompss," in *26th Parallel & Distributed Processing Symposium (IPDPS)*. IEEE, 2012, pp. 557–568.

[4] A. Muddukrishna, P. A. Jonsson, and M. Brorsson, "Locality-aware task scheduling and data distribution for OpenMP programs on NUMA systems and manycore processors," *Scientific Programming*, 2015.

[5] A. Drebes *et al.*, "Topology-aware and dependence-aware scheduling and memory allocation for task-parallel languages," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 3, pp. 30:1–30:25, Aug. 2014.

[6] ——, "Scalable task parallelism for NUMA: A uniform abstraction for coordinated scheduling and memory management," in *Proc. of the 2016 Intern. Conf. on Parallel Architectures and Compilation*, ser. PACT '16. New York, NY, USA: ACM, 2016, pp. 125–137. [Online]. Available: http://doi.acm.org/10.1145/2967938.2967946

[7] C. Augonnet *et al.*, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[8] G. Bosilca *et al.*, "Parsec: Exploiting heterogeneity to enhance scalability," *Computing in Science & Engineering*, vol. 15, no. 6, pp. 36–45, 2013.

[9] E. Jeannot, "Symbolic Mapping and Allocation for the Cholesky Factorization on NUMA machines: Results and Optimizations," *Intern. J. of High Performance Computing Applications*, vol. 27, no. 3, pp. 283–290, 2013.

[10] P.-N. Clauss and J. Gustedt, "Iterative Computations with Ordered Read-Write Locks," *Journal of Parallel and Distributed Computing*, vol. 70, no. 5, pp. 496–504, 2010. [Online]. Available: http://hal.inria.fr/inria-00330024/en

[11] F. Broquedis *et al.*, "hwloc: A generic framework for managing hardware affinities in HPC applications," in *2010 18th Euromicro Conference*, Feb 2010, pp. 180–186.

[12] E. Jeannot, G. Mercier, and F. Tessier, "Process Placement in Multicore Clusters: Algorithmic Issues and Practical Techniques," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 4, pp. 993–1002, Apr. 2014.

[13] J. Dongarra *et al.*, *LINPACK Users' Guide*. Society for Industrial and Applied Mathematics, 1979. [Online]. Available: http://epubs.siam.org/doi/abs/10.1137/1.9781611971811

[14] P. N. Clauss and J. Gustedt, "Experimenting iterative computations with ordered read-write locks," in *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, Feb 2010, pp. 155–162.

[15] S. Ojha and S. Sakhare, "Image processing techniques for object tracking in video surveillance – a survey," in *Pervasive Computing (ICPC), 2015 Intern. Conf. on*, Jan 2015, pp. 1–6.

[16] T. Yang *et al.*, "Real-time multiple objects tracking with occlusion handling in dynamic scenes," in *2005 IEEE Computer Society Conf. on Computer Vision and Pattern Recognition (CVPR'05)*, vol. 1, June 2005, pp. 970–975 vol. 1.

[17] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, Sept 1987.