Fully-abstracted affinity optimization for task-based models

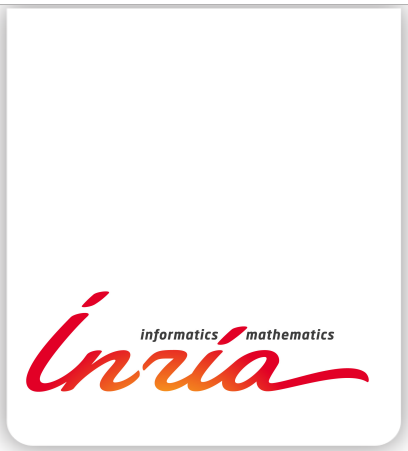
Jens Gustedt, Emmanuel Jeannot, Farouk Mansouri

Fully-abstracted affinity optimization for task-based models

Jens Gustedt **Emmanuel Jeannot** **Farouk Mansouri**

RESEARCH REPORT
N° 8993
Dec 2016
Project-Team Camus

ISSN 0249-6399 ISRN INRIA/RR--8993--FR+ENG



Fully-abstracted affinity optimization for task-based models

Jens Gustedt*† Emmanuel Jeannot‡ Farouk Mansouri‡

Project-Team Camus

Research Report n° 8993 — Dec 2016 — 21 pages

Abstract: Task-based models and runtimes are quite popular in the HPC community. They help to implement applications with a high level of abstraction while still applying different types of optimizations. An important optimization target is hardware affinity, which concerns to match application behavior (thread, communication, data) to the architecture topology (cores, caches, memory). In fact, realizing a well adapted placement of threads is a key to achieve performance and scalability, especially on NUMA-SMP machines. However, this type of optimization is difficult: architectures become increasingly complex and application behavior changes with implementations and input parameters, *e.g* problem size and number of thread. Thus, by themselves task based runtimes often deal badly with this optimization and leave a lot of fine-tuning to the user. 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 architecture's topology. Implemented in the backend of our task-based runtime 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 SGI SMP machines with quite different hardware characteristics, our tests show spectacular performance improvements for this unmodified application code due to a dramatic decrease of cache misses. A comparison to reference implementations using OpenMP confirms this performance gain of almost one order of magnitude.

Key-words: Task based runtimes, Hardware affinity, Parallel programming

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

RESEARCH CENTRE NANCY – GRAND EST

615 rue du Jardin Botanique
CS20101
54603 Villers-lès-Nancy Cedex

Abstraction complète de l'optimization de l'affinité pour des modèles à base de tâches

Résumé : Modèles et environnements à base de tâches sont très populaires dans la communauté du HPC. Ils aident à implanter des applications à un niveau d'abstraction élevé en permettant néanmoins différents types d'optimisation. Une cible d'optimisation importante est l'affinité, qui consiste à lier le comportement de l'application (processus légers, communication, données) à la topologie de l'architecture (cœurs, caches, mémoire). Réaliser un placement bien adapté des processus est un levier effectif pour atteindre performance et passage à l'échelle, en particulier sur des machines NUMA-SMP. Néanmoins, ce type d'optimisation est difficile : les architectures deviennent de plus en plus complexes et le comportement des applications change selon les implantations et les paramètres d'entrées, p. ex. la taille du problème ou le nombre de processus. Souvent les environnements d'exécution gèrent mal ce type d'optimisation par eux-mêmes et laissent beaucoup de réglages minutieux à l'utilisateur. Avec ce travail nous proposons un module pour controler l'affinité qui est complètement automatique, abstrait et portable. Il produit et implante une stratégie d'affinité optimisée qui combine les connaissances sur les caractéristiques de l'application et sur la topologie de l'architecture. Implanté comme module interne de notre environnement d'exécution ORWL, notre approche a été utilisé pour améliorer la performance et le passage à l'échelle de plusieurs applications ORWL non-modifiés : multiplication de matrices, un stencil 2D (Livermore Kernel 23), et une application réelle de poursuite vidéo. Sur deux machines SMP de SGI avec des caractéristiques matérielles relativement différents nos tests montrent des améliorations spectaculaires pour ces applications non-modifiés, dû a une baisse très notable des défauts de caches. Une comparaison avec des implantations de référence utilisant OpenMP confirme ce gain de performance de presque un ordre de grandeur.

Mots-clés : environment d'excution basé sur des processus légers, affinité matérielle, programmation parallèle

1 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 with *e.g.* OpenMP 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 level of affinity abstractions. However, obtaining good hardware affinity usually requires an in-depth knowledge of the underlying architecture as well as the application behavior. In this paper, we propose a fully abstracted and portable 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 task-based framework named the Ordered Read Write Location (ORWL)

This paper is organized as follows. In Section 2, we present related work about affinity for thread-based frameworks. After that, Section 3 describes the context and the background of our work including tools and frameworks we use. In Section 4, we introduce the affinity module that connects knowledge about application and platform structure and explain its implementation. Task based implementations of two benchmarks and a real world application based on our ORWL framework are presented in Section 5. Section 6 presents runtime 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 7 summarizes our achievements and gives some perspectives for future work.

2 Related work

Binding each computing task to its own dedicated processor is fairly common nowadays. Indeed, if a task is not bound, the operating system may migrate it to another CPU (e.g. whenever a daemon wakes up or a new thread is

scheduled). This causes a load imbalance and increase system noise. Binding each thread prevents migration from happening and thus keeps processor caches hot. However, implementing a naive binding of threads is not enough to get optimized execution and scalability. In fact, applying a smart strategy of binding which reduce global communication cost is necessary to reduce CPUs stall time.

Software tools like HWLOC [1], LibUMA [2], `sched_setaffinity()` from Linux interface *sched.h* or `pthread_setaffinity_np()` in POSIX interface *pthread.h* allows to bind threads of the application to the architecture in order to avoid migration problems due to system scheduling. However, these tools are low level programming interface: the user has to manually codes the binding strategy for each thread of the application taking into account the optimization of data locality. In addition, to make a portable strategy of thread affinity, he has to deal with targeted architectures and adapt its binding policy for each one.

Thread based models of programming like OpenMP propose a higher level interfaces to optimize thread affinity by using some environment variable to enable binding strategies. With the Intel runtime for example, by using `KMP_AFFINITY` it is possible to adapt the targeted topology and to chose on which cores we want to execute which threads. It is also possible to set some global strategy by compacting or scattering threads over cores. The recent versions of the GCC runtime (GOMP) also allows to do more or less the same by setting the variable `GOMP_CPU_AFFINITY`. These approaches of affinity modules that had been proposed by different vendors are now ported into the last OpenMP standard 4.5. It specifies the `OMP_PLACES` and `OMP_PROCBIND` interfaces for the affinity policy. All these interfaces are high level and easy to use for inexperienced programmers. However, they do not look at the actual application behavior before deciding where to bind each thread. The strategies they propose is too generic and may be inappropriate. Thus, the programmer need to use the low level tools cited above in order to specify the adapted binding strategy.

In [3], the authors propose an extension based on a *location* directive to make affinity of OpenMP threads and data. More recent work [4] propose a novel OpenMP directive to control the affinity of OpenMP tasks to be bound on threads, NUMA node or close to some data. However, in these solutions the user need to manually insert specific directive to match graph of tasks and architecture topology which need low level handling. In addition, he still has to investigate the application behavior and find the good strategy of binding. Indeed, neither the OpenMP standard nor existing thread-based runtimes currently allow to automatically produce and set an adapted affinity strategy of threads. Last, each time the application code changes, the directive have to be adapted and modify to ensure performance gain and the sequential semantic. OmpSS [5] is a variant of OpenMP that targets heterogeneous architectures. However, as it is based on the OpenMP model it suffers from the same problem of general OpenMP solutions as highlighted above.

To provide thread affinity and data locality in thread based runtimes other work focuses on scheduling. [6] introduces locality for the OpenMP task-scheduling by extracting dependency information at compilation time. [7] uses data locality and task placement to improve the scheduler of a data flow graph tool named

OpenStream. These approaches are well oriented for dynamic runtimes that distribute fine grain tasks over worker threads. However, they are not adapted for applications with limited number of tasks and a coarse granularity. Here dynamic scheduling is not efficient and has an unnecessary overhead. In addition, several applications require static scheduling. *E.g.* our video tracking implementation needs each task to be bound on a specific core, such that the execution is ordered and the FPS count is stable.

Task based runtime systems such as StarPU [8], or PARSEC [9] are also related to this research. In StarPU, several scheduling policy 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 release 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 extremely good but this requires the user to define the mapping of the data (e.g. distribution of the matrix) onto the resources explicitly and also to describe tasks dependencies. An approach to compute the data mapping automatically as been proposed by a one of the co-author of this paper [10]. However such approaches are bound to the parameterized task graph model and are only suited for code with static control if there are affine access and loop bounds and 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 and do not require to update or adapt the application code.

3 Background and context

The ORWL model and library The ORWL "Ordered Read-Write Lock" programming model [11] 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 *location* which are used for sharing data 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 and the manager of the FIFO controls the access order and locks the resource for some threads or allocates it to the appropriate threads. 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

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

application into a directed acyclic graph (DAG) of tasks.

- `orwl_location`: is the primitive to represent a shared resource between the tasks. It could be data (contents), memory (a specific address), a computational unit (CPU or accelerator) or a 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 read or write access obtains the data concurrently or exclusively.

In addition, the library proposes some specific primitives to easily implement iterative tasks when they perform some iterations where they alternates 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` which repeatedly introduces queries for the resource. By this, each task runs a series of synchronized iterations. This iterative access to resources guarantees the consistency of data, deadlock-freeness and fairness for the decentralized event-based execution. Thanks to these primitives the user may express a high level of parallelism within its application while avoiding the manual use of a low level C interface to manage lock synchronization and or to communicated between threads.

The HWLOC library for locality management The increasing numbers of cores, shared caches and memory nodes within machines introduce complex hardware topology. High-performance computing applications now have to carefully adapt their placement and behavior according to the underlying hierarchy of hardware resources and their software affinities. In fact, shared-memory or synchronization between tasks can benefit from shared caches, while intensive memory access requires local memory allocations. Thus, exploiting modern architectures requires simultaneously an in-depth knowledge of the underlying architecture *and* of the application behavior. The hardware locality tool "hwloc" [1] exposes a portable abstracted view of the hardware topology to the developer or the runtime system. It also provides a way to bind threads to (a set of) cores. It may significantly improve performance by having runtime systems place their tasks and adapt their communication strategies with respect to hardware affinities.

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. It implements different placement algorithms that are switched according to the input size so as to maintain a low running time.

4 Abstracted affinity add-on for ORWL thread-based runtimes

4.1 Conception

As described above, ORWL is a resource centric manager. It allows to construct a set of tasks and concurrently executes them according to their FIFO access to the locations (resources). To ensure coherence throughout an event-based execution, the ORWL runtime uses threads together with a lock mechanism. A pool of threads is used to execute tasks and to manage the locks synchronization and the data transfer that are associated to locations. These control threads may freeze and thaw processing threads of concurrent tasks according to the availability of resources. Thereby, the model of execution is highly decentralized.

The mapping of (application) tasks to threads can be chosen out of two possible ways:

- *One thread per task:* Each task is executed with by one thread from the pool and can access several locations.
- *Several threads/operations per task:* Here several sub-tasks named *operations* cooperate to execute a task. Each operation is executed by a thread from the pool and will typically be responsible of 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 task-graph topology and the location sizes. 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 models exposes all the required pieces of information: the tasks, their connectivity and the amount of data they share or exchange. Therefore there is no need to modify the code or to add any directive to gather that information. This allows to construct a matrix (see Fig. 1) that expresses the communication volume shared between tasks. At the other end we use the HWLOC to obtain the hardware topology of the underlying environment in a automated and portable way. From these two inputs, we develop an allocation strategy that aims to reduce the communication between the NUMA nodes. Simultaneously, it optimizes the shared caches inside each of them. We regroup threads that share data, and at the same time, we distribute threads over 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 less 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), but, some applications have a requirement for a minimum number

of threads that might exceed the number of resources. The second modification consists in taking the control and communication threads of ORWL into account. The algorithm here depends on the available computing resources and specially on the presence of hyperthreading in the architecture of the processors.

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 built the mapping
```

Algorithm 1 is run at launch time, once the topology tree is given by HWLOC and the communication matrix is computed by ORWL, and provides a mapping of the computing entities (the threads) to the cores. These threads can then be bound to the core 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 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 and we let the system schedule them. Second, we check if oversubscribing is required by comparing the number of leaves of the tree with the order of the communication matrix. We optionally add a new level to this tree such that we have enough virtual resources to compute the allocation.

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 of the problem size, we go from an optimal but exponential algorithm to a greedy and linear one such that the running time is kept below 1 second for mapping up to 5000 threads to 5000 cores. Before going from depth l to $l-1$ we need to aggregate the communication matrix in order to reflect the affinity between the groups. This is done by the function `AggregateComMatrix`. Once we have build this hierarchy of groups we match it to the topology tree

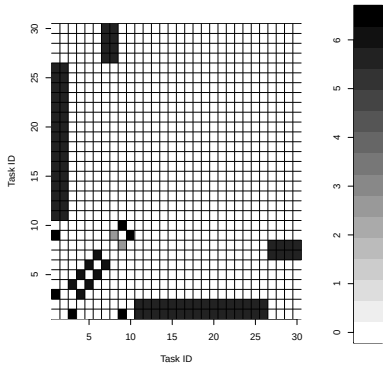


Figure 1: Communication matrix of the video tracking application (see Sec. 5.3) – logarithmic gray scale

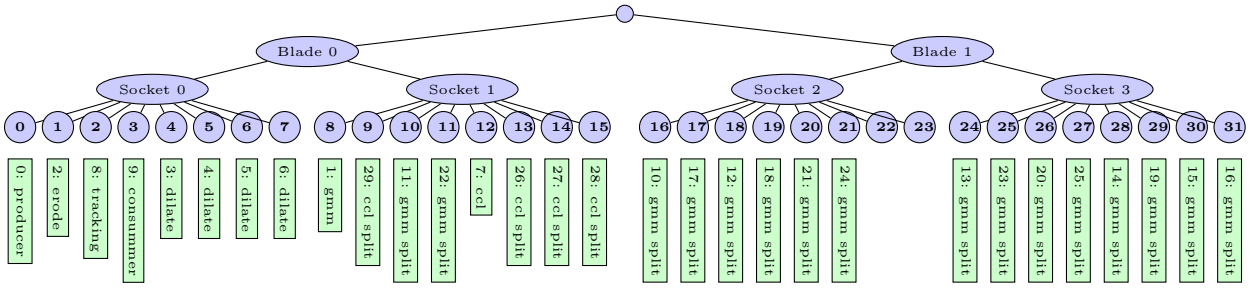


Figure 2: Task allocation on 4 socket NUMA machine of the video tracking application (see Sec. 5.3). Note that, for space reason, we do not describe the cache hierarchy, while this hierarchy is also given by hwloc and is actually used by our algorithm for the mapping

such that each thread is assigned to a leaf (function `MapGroups`). If oversubscribing has been 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. 5.3. 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 1. Each task has a green box with the ID corresponding to the one used in Fig. 1and its name. We see that the pipeline Tasks 0 to 9 are put on the same socket except Task 1 and 7 which are mapped to an other node as they communicate with other tasks (resp. gmm split and ccl split). Last, note that here core 22 and 23 are automatically reserved for control threads.

4.2 Implementation

Our affinity module is implemented as an Add-on for the ORWL framework. The C API of the ORWL model is enriched by adding some affinity functions and structures. To enable the affinity optimization with the fully automatic and abstracted mode, the ORWL user has only set the environment variable `ORWL_AFFINITY` to 1. This variable is checked at runtime and the appropriate affinity for the thread is computed and set behind the scenes as described above. It is also possible to use our affinity module in a semi-automatic mode (e.g. for debugging purpose): we developed some interface functions which may be called by the user:

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

5 Benchmarks and application

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

5.1 Livermore Kernel 23 benchmark

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 1 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 1: The Livermore Kernel 23 loops

```
for (l=1; l<=loop; l++){
  for (j=1; j<n; 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`.

The ORWL implementation To implement this algorithm with the ORWL model an intuitive method is to decompose the `za` matrix into several blocks. For each block, the inner computation is independent from the other blocks whereas the computation of edges or corners depends on some neighboring blocks. Thus, for each block we define a main operation (`MP`) that performs the computation and eight sub-operation (`SP`) that are used to export the frontier data (edges and corners) to the neighboring `MP`. Fig. 3 shows an example of a simple decomposition into four blocks. The four `MT` are numbered from 0 to 3 and the associated `ST` are prefixed with their direction.

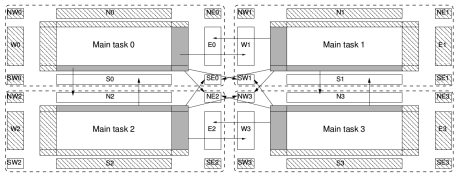


Figure 3: Illustration of blocks decomposition of Livermore Kernel 23 benchmark

Thus for this implementation several `orwl_task` primitives are each divided to 9 operations (functions). Each operation is executed by an independent thread and has its own `orwl_location` to exchange the shared data with neighbors. The read/write dependencies between operations of the matrix blocks are defined using the `orwl_handle` primitive which allows to ensure the computation coherency.

5.2 Matrix multiplication benchmark

Dense matrix operations are important elements in scientific and engineering computing. It is a benchmark that 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 consist 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. Fig. 4 illustrates this approach where the blocks of matrix `A` and `B` circulate once.

As kernels for block computations for this algorithm we use the `DGEMM` interface of the BLAS library standard. We compare two different implementations of `DGEMM`, Intel's library MKL and the opensource implementation ATLAS.

The MKL library also features an OpenMP multithreaded version of `DGEMM`. Depending on the number of threads it uses more or less cores. We compare our approach with this MKL version by varying the number of threads and the binding strategies.

The ORWL implementation As represented in Fig. 4, in the ORWL implementation we model each row of the result matrix `C` with a task/thread using the `orwl_task` primitive. A task processes the elements of a row of the matrix `C` and circulates the input columns of the matrix `B` to the neighbouring tasks by using ORWL's "locations". Each task is connected to its own location and to the locations of the precedents tasks by using the `orwl_handle` primitive. The result blocks of the matrix `C` are computed with a complete circulation of the input columns of matrix `B`. Such a block-cyclic algorithm is easily implemented in ORWL which allows to reach the following objectives: (1) Achieve a high abstraction level to decompose the matrix multiplication problem into parallel tasks. (2) Synchronise tasks and maintain data coherency without manipulating any thread primitive. (3) Benefit of the communication management and the overlapping with computations. (4) Allow to re-use optimized public domain or vendor specific compute kernels for the individual blocks.

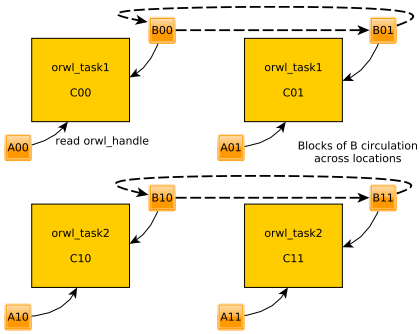


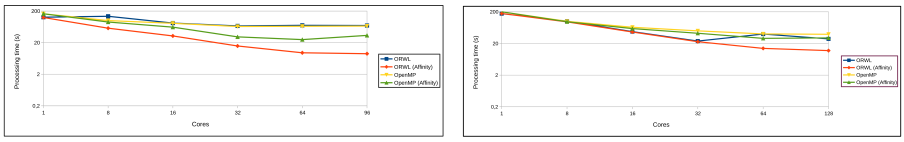
Figure 4: Illustration of the ORWL implementation for the block cyclic matrix multiplication

At any time, we only use a single thread that performs ATLAS or MKL `DGEMM` calls per ORWL task, while several other threads can perform control and communication tasks in the background.

In addition, due to our placement module, the user does not need to worry about the placement of tasks. Together with HWLOC, our binding strategy transparently improves thread affinity, without requiring a priori knowledge of the architecture.

5.3 HD Video Tracking application

The video tracking application follows moving objects seen by several cameras. Recently this application has become important for video surveillance of public spaces or the traffic control. To track moving objects in a video, several algorithmic approaches have been explored [14]. 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 [15]. This algorithm shows interesting results. However, it remains sensitive to the image size which limits its use in a "streaming" environment (without I/O buffering) for current HD video capture technology. There we observe high accuracy with many pixels and



(a) SMP12E5 (b) SMP20E7

Figure 5: The processing times (log scale) of Livermore Kernel 23

an strong increase in the data volume. To tackle this, we exploit the parallelism of multi-core architectures. Our goal is to accelerate the application such as to improve the frame rate per second (FPS).

The data-flow graph ORWL implementation To easily implement parallel video tracking application on a multi-core architecture we use the ORWL model. The application is iterative with repetitive processing applied on each frame. It is usually modeled as a synchronous data-flow graph (DFG) [16] as shown in Fig. 6. 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 task parallelism where each function is processed as soon as its input data are available.

As shown in Section 3, the ORWL model allows to easily decomposes 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 writing handle for the outgoing dependence (output data). At runtime, each ORWL task ahead in the reading critical section to recover the input data from the "location" of the previous tasks. Then, these pieces of data are independently dealt by each task and written to the location. In addition, we add some DFG-specific features: a `orwl_fifo` primitive to store modified data and then quickly free the lock for other readers/writers. A `orwl_split` primitive to split data of a location into several pieces processed by other tasks or operations "sub-task". The last primitive is used to split the tasks GMM and CCL. These two are the most expensive and form bottlenecks for the pipeline. This is illustrated in Fig. 6. The ORWL tasks are executed in parallel by different threads and process multiple input images concurrently: we exploit task parallelism in a pipeline mode and data parallelism in a split-merge mode.

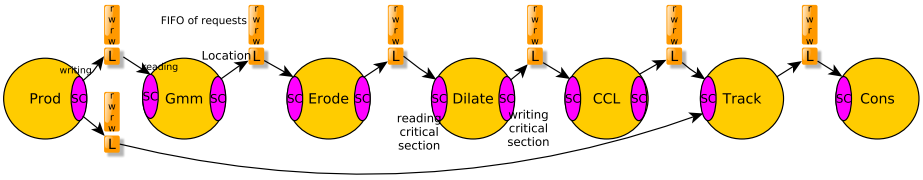


Figure 6: Illustration of the ORWL implementation of the DFG video tracking application. Yellow nodes represent task, purple regions represent read/write `ORWL_SECTION`. Each task is connected to locations (orange squares) using handles. In the experiments, Dilate is repeated 3 times. GMM and CCL are split into 16 and 4 sub-tasks.

6 Experiments and results

6.1 Test-beds and architectures

For our study we use two SMP architectures from the Plafrim platform [17]. The characteristics of these architectures are described in Table 1. The main difference between these two architectures is that the SMP12E5 platform is a newer generation and features hyperthreading and enables performance counters

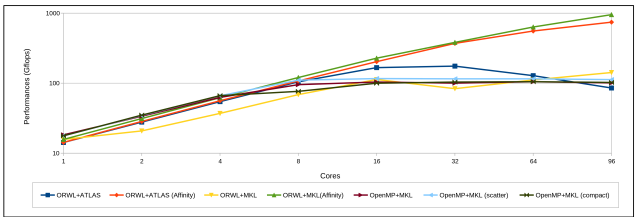
Table 1: The multi-core architectures used for the experiments

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/ATLAS	5.1/3.10.2	5.1/3.10.2
ICC/MKL	14.0/11.1	14.0/11.1

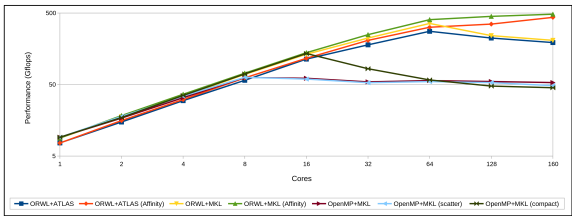
6.2 Experiments

This section contains results of experiments using the applications we present in Section 5. The main idea is to assess the performance difference from enabling our affinity module. Hence, we compare the performances of the ORWL implementations based on affinity optimization to the native ORWL implementations. Here only the system scheduler deals with computation and binding. Independent of the target machine, we use the same application code with the same configuration.

In addition, we compare the performances of our module to those achieved by optimizing the affinity of OpenMP implementations of the same applica-



(a) SMP12E5



(b) SMP20E7

Figure 7: FLOP/s performances of the Matrix multiplication implementations

tions. For all the cases we implemented several versions with different affinity optimizations as proposed in the literature. Due to lack of space only the best implementations are reported here. In addition to the runtimes we present measurements of hardware and software counters collected with the the benchmarks to explain the differences.

6.2.1 Livermore Kernel 23

Fig. 5 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. The OpenMP implementation is equivalent to the ORWL implementation described in Section 5. It is based on introducing `#pragma parallel for` directives with static scheduling of chunks over the threads.

ORWL and OpenMP implementations scale until 8 cores. After that they perform badly and stabilize on about 65 seconds on SMP12E5 and about 40 seconds on SMP20E7. With the affinity optimization based on `OMP_PLACES=cores` and `OMP_PROC_BIND=spred/close`, the both OpenMP implementations give the same results, they slightly enhance the performances up to 1.3x on SMP20E7 and about 2.5x on SMP12E5. However, our ORWL implementation with our affinity module scales even more and reaches about 3x on SMP20E7 and about 8x on SMP12E5.

We also studied the hardware and software counters of the machine for a 64 cores run, see Table 2. We see that the affinity management reduces the L3 cache misses by a substantial factor. We also see a strong correlation between

cache misses and cycle stalls: each cache miss leads to a loss of about 10 to 14 cycles. Our approach (ORWL and affinity management) has 5 time less cache misses and more than 3 times less stalled cycles. On the other hand the ORWL approach generates much more context switches.

Table 2: 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 frontend	840	200	840	720
context switches	99 778	89 151	745	210
CPU migrations	15 960	0	203	0

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 system a context switch has a cost of about 100 ns. Hence, ca. 100 000 context switches that are spread over 64 cores correspond to an overhead of less 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 concerns control and management threads and not the compute threads implementing the tasks.

6.2.2 Matrix multiplication

Fig. 7 presents performance comparisons of several implementations multiplying two 16384x16384 matrices of double precision elements on scalable configuration of the used architectures.

For a reduced number of cores inside a socket, all native implementations (without affinity) scale. MKL implementations based on OpenMP 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. By enabling our affinity module, without changing any line of code, the performance

Table 3: Accumulated Hardware/software counters of Matrix multiplication on SMP12E5 (64 cores)

	Billions of L3 misses	Billions of stalled cycles	CPU mig.	Context sw.
ORWL+ATLAS	99	7250	29 537	151 844
ORWL+ATLAS (Affinity)	14.5	1020	0	135 345
ORWL+MKL	102	8110	28963	153 265
ORWL+MKL (Affinity)	13.8	980	0	125 368
OpenMP+MKL	140	8850	486	2 863
OpenMP+MKL (Affinity scatter)	99	8140	0	2 750
OpenMP+MKL (Affinity compact)	89	8520	0	3 001

of all ORWL implementations is enhanced for a use outside the socket and scales to reach a maximum of 1 Tflop/s on SMP12E5 and 0.5 Tflop/s on SMP20E7 using all cores of machines. In contrary, the OpenMP implementations based on affinity optimisation[2] stabilize and can not enhance the performances. Moreover, we observe that using the MKL kernel within the ORWL implementation leads to slightly better results than ATLAS. Again, we have gathered hardware and software counters for the 64 cores run on the SMP12E5 machine, see Table 3. We see that the ORWL implementations can considerably reduce the number of L3 cache misses and pipeline stalls compared to the affinity-based OpenMP implementation. In contrast to that, the number of CPU migrations and context switches are considerably higher for ORWL than for the others.

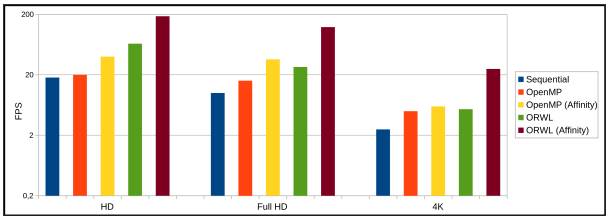
These results can be consistently explained as above: our management of locality leads to much improved execution times, due to reduced cache misses and stalls. Again much higher numbers for thread migration and context switches doesn't seem to influence execution times much.

Another interesting observation is that the OpenMP 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 tends to the same performance. This shows, that the compact strategy is not sufficient to overcome the lack of awareness for affinity.

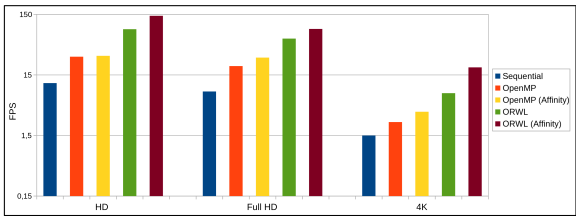
6.2.3 Video tracking

Fig. 8 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.

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



(a) SMP12E5



(b) SMP20E7

Figure 8: FPS of HD video tracking

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` does not produce a comparable performance enhancement. It accelerates the FPS by about 2x on SMP12E5 and 1.5x on SMP20E7. To assess the difference of the performance for our affinity optimization, we present some hardware and software counters in Table 4. Here again, we see that the affinity optimization produced by our strategy allows to 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.

Table 4: 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 frontend	160	83	840	660
context switches	413821	329263	99778	22241
CPU migrations	61390	0	15960	0

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 hyperthreading) 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.

7 Conclusion

In this paper, we presented a fully-abstracted affinity module for the static task-based model. Our module improves the software to hardware mapping based on automatically extracting and matching communication behavior and hardware topology. It computes and implements an optimized thread binding strategy that takes the characteristics of both architecture and application into account. Thanks to this module, users get full abstraction of the affinity of codes and their 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. Last and importantly, it should be noted that, even if we have implemented this solution within ORWL, the approach is generic and can be integrated in any runtime system as soon as affinity settings for its different computing parts (threads, processes, ...) is available and provided by abstractions in the programming model.

In Sections 6, 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 to get a 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.

In future work we want to extend our strategy to heterogeneous architectures (such as the KNL or GPUs). ORWL already works on such architectures and we need to extend the strategy by taking into account their specific characteristics (memory hierarchy, computation offloading, etc.)

Acknowledgment

Some 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/`). This work is also funded by the Inria IPL *multicore*.

References

[1] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, "hwloc: A generic framework for managing hardware affinities in HPC applications," in *2010 18th Euromicro Conference*, Feb 2010, pp. 180–186.

[2] A. Kleen, "A NUMA API for Linux," *Novel Inc*, 2005. [Online]. Available: http://halobates.de/numaapi3.pdf

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

[4] P. Virouleau, A. Roussel, F. Broquedis, T. Gautier, F. Rastello, and J.-M. Gratien, "Description, Implementation and Evaluation of an Affinity Clause for Task Directives," in *IWOMP 2016*, Nara, Japan, Oct. 2016.

[5] J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguade, and J. Labarta, "Productive programming of gpu clusters with ompss," in *26th Parallel & Distributed Processing Symposium (IPDPS)*. IEEE, 2012, pp. 557–568.

[6] 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.

[7] A. Drebes, K. Heydemann, N. Drach, A. Pop, and A. Cohen, "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.

[8] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "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.

[9] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Hérault, and J. J. Dongarra, "Parsec: Exploiting heterogeneity to enhance scalability," *Computing in Science & Engineering*, vol. 15, no. 6, pp. 36–45, 2013.

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

[11] 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

[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.

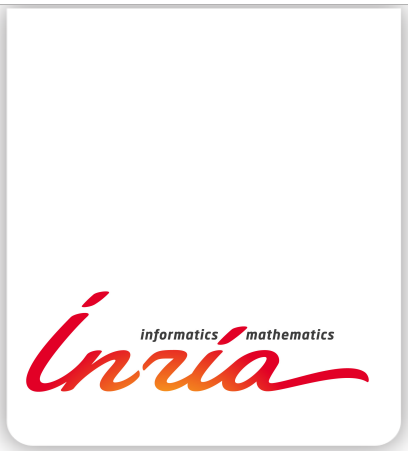
[13] J. Dongarra, C. Moler, J. Bunch, and G. Stewart, *LINPACK Users' Guide*. Society for Industrial and Applied Mathematics, 1979. [Online]. Available: http://epubs.siam.org/doi/abs/10.1137/1.9781611971811

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

[15] T. Yang, Q. Pan, J. Li, and S. Z. Li, "Real-time multiple objects tracking with occlusion handling in dynamic scenes," in *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, vol. 1, June 2005, pp. 970–975 vol. 1.

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

[17] Plafrim, "Plate-forme fédérative pour la recherche en informatique et mathématiques." [Online]. Available: https://plafrim.bordeaux.inria.fr/doku.php



RESEARCH CENTRE
NANCY – GRAND EST

615 rue du Jardin Botanique
CS20101
54603 Villers-lès-Nancy Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399