View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by INRIA a CCSD electronic archive server

Optimizing Locality by Topology-aware Placement for a Task Based Programming Model

Jens Gustedt, Emmanuel Jeannot, Farouk Mansouri

▶ **To cite this version:**

Jens Gustedt, Emmanuel Jeannot, Farouk Mansouri. Optimizing Locality by Topology-aware Placement for a Task Based Programming Model. IEEE Cluster 2016 Conference, Sep 2016, Taipei, Taiwan. pp.164 - 165, 10.1109/CLUSTER.2016.87 . hal-01416284

HAL Id: hal-01416284

https://hal.archives-ouvertes.fr/hal-01416284

Submitted on 14 Dec 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Optimizing Locality by Topology-aware Placement for a Task Based Programming Model

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

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

***Abstract*—The ordered read-write lock model (ORWL) is a modern framework that proposes high level abstractions for the decomposition of an application and for the management of synchronizations and communications. The implementation of the model reaches high performances thanks to a decentralized event-based runtime. In this paper, we propose to enrich ORWL by proposing a topology-aware placement module that is based on the Hardware Locality framework, HWLOC. The aim is double. On one hand we increase the abstraction and the portability of the framework, and on the other hand we enhance the performance of the model's runtime. We propose a placement policy, that takes the characteristics of the application, of the runtime and of the architecture into account. We validate and compare our approach with the Livermore kernel23 benchmarks.**

I. Introduction and background

Multicore architectures are a very good option for high performance computing thanks to NUMA or SMP technologies, but they are still hard to program. Indeed, a programmer faces challenging problems related to the expression and exploitation of parallelism. To ease their use while obtaining good performance, several programming models have been developed, such as OpenMP[1], TBB[2] or StarPU[3].

Among these programming models ORWL [4] *"Ordered Read-Write Locks"* provides a framework based on the management of shared resources in a parallel environment. These resources are abstracted in the ORWL model by the notion of *location*. Tasks executed by one or several threads concurrently access to a resource/location by using a FIFO that holds requests (requested, allocated, released) issued by threads. 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. However, as in many other programming models (e.g. StarPU, OpenMP) up to now the model offered no support to take the topology of the target machine into account. As we will see in our experiments, this hampers performance badly when an application is executed on many cores of a deeply hierarchical NUMA machine. Hence, being able to map task/threads according to the target topology is of extreme importance as the number of available cores in parallel NUMA architectures is increasing. In this work we aim to enrich the ORWL runtime with a tasks placement strategy. Thus we propose a placement add-on based on the HWLOC software (the de-facto standard for modeling NUMA system topologies) to get a portable abstraction of the architecture which may significantly improves performance by having runtime systems place their tasks and adapt their communication strategies with respect to hardware affinities. This strategy is tested with a stencil code and compared to state-of-the-art implementations and we show that we are able to outperform these substantially.

II. Topology-aware add-on of ORWL tasks

Our aim is to propose a placement strategy that optimises data locality. To do so, we exploit application information as it is gathered from ORWL runtime to construct a weighted matrix that expresses the communication volume between threads. On the other hand, we use the HWLOC library interface to obtain the hardware topology in an automated and portable way. From these two inputs, we develop an allocation strategy that aims at reducing the communication between the NUMA nodes while optimising the shared caches inside each of them. Therefore, we cluster threads that share data, and at the same time, distribute threads over NUMA nodes. To compute the allocation we use Algorithm 1 that is based on the TreeMatch algorithm [5]. We have adapted it in two ways to our needs. First, we have enhanced it to account for oversubscription when there are less computing resources than tasks. 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 $m \leftarrow$ extend_to_manage_control_threads($m$)
2 $T \leftarrow$ manage_oversubscription($T$,$m$)
3 groups[1..$D-1$]=$\emptyset$ // How nodes are grouped on each level
4 foreach depth$\leftarrow D-1..1$ do // We start from the leaves
5     $p \leftarrow$ order of $m$
6     groups[depth]$\leftarrow$GroupProcesses($T$,$m$,depth)// Group processes
      by communication affinity
7     $m \leftarrow$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 and provides a mapping of the computing entities (the threads) to the cores. 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 oversubscription is required by comparing the number of leaves of the tree with the order of the communication matrix and 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 arity of the considered level. 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 such that each thread is assigned to a leaf (function `MapGroups`).

III. The Livermore Kernel 23 Benchmark

The Livermore Kernel 23 is a classic benchmark taken from LinPack [6] to simulate a 2-D implicit hydrodynamics fragment. To implement this algorithm with the ORWL model an intuitive method is to decompose the 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 that performs the computation and eight sub-operation that are used to export the frontier data (edges and corners) to the neighbouring. 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 neighbours. The read/write dependencies between operations of the matrix blocks are defined using the `orwl_handle` primitive which allows to ensure the computation coherency. To validate our placement strategy we compare the performance of the ORWL implementations with our binding strategy against OpenMP of equivalent abstraction. We use an SMP machine composed of 24 sockets of 8 cores to process a 16384x16384 matrix of double precision elements during 100 iterations. The ORWL Bind implementation produces the best results. It reaches a minimum processing time of about 11 seconds which represent speedup of 5 and 2.8 compared to OpenMP and ORWL NoBind implementations, respectively. This clearly shows that our placement strategy pays off and allows to unfold the full potential of the architecture.

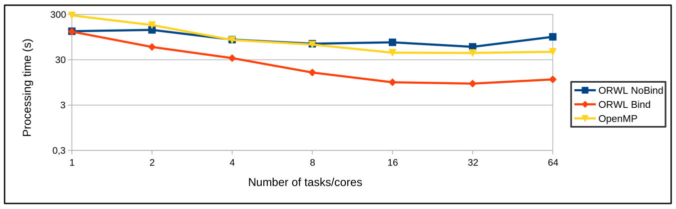

Fig. 1. Comparison of the processing time resulted from executing the 3 implementations of Livermore Kernel 23 on SMP architecture of 192 cores

IV. Conclusion

In this paper, we presented a locality study for the task-based ORWL model on multi-core architectures. We enriched this model by adding portable module of affinity optimisation based on the HWLOC framework and that automatically extracts task/threads affinity based on the way they are composed in the application. We proposed, a threads/tasks binding strategy taking into account the architecture topology and the application characteristics. Thanks to this module, the user gets a high level abstraction of the architecture while optimizing the locality of its implementation. We validated our approach on the Livermore kernel 23 benchmark and we shown that our placement approach enhances the performances of the ORWL implementation and outperforms OpenMP non topology-aware approaches. Indeed, as soon as we scale beyond one or two sockets, standard approaches that do not take into account the affinity and the topology fail improve performance. Here show that its is important to carefully bind the threads on the resources taking these two characteristics (topology and affinity) into account. Thus, we can claim that our approach is beneficial to the user for efficiently taking advantage of the computing power of multi-core architectures while getting a high level abstraction from it.

References

[1] B. Chapman, G. Jost, and R. v. d. Pas, *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[2] J. Reinders, *Intel Threading Building Blocks*, 1st ed. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2007.

[3] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," in *Euro-Par 2009*, Delft, Netherlands, Aug. 2009. [Online]. Available: https://hal.inria.fr/inria-00384363

[4] 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

[5] 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.

[6] 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