Using Graph Partitioning to Accelerate Task-Based Parallel Applications

Isaac Sánchez Barrera, Marc Casas, Miquel Moretó
Eduard Ayguadé, Jesús Labarta, Mateo Valero
Barcelona Supercomputing Center – Centro Nacional de Supercomputación (BSC-CNS), Barcelona, ES
{isaac.sanchez, marc.casas, miquel.moreto, eduard.ayguade, jesus.labarta, mateo.valero}@bsc.es

***Abstract**-Current high performance computing architectures are composed of large shared memory NUMA nodes, among other components. Such nodes are becoming increasingly complex as they have several NUMA domains with different access latencies depending on the core where the access is issued.*

In this work, we propose techniques based on graph partitioning to efficiently mitigate the negative impact of NUMA effects on parallel applications performance, which are able to improve the execution time of OpenMP parallel codes 2.02× times on average when run on architectures with strong NUMA effects.

I. INTRODUCTION

Since the end of Dennard scaling and the subsequent stagnation of the CPU clock frequency, computing infrastructures can only increase their peak performance via augmenting their number of computing units. In the High Performance Computing (HPC) context, this trend has brought an increase in the hardware components count as well as in the heterogeneity among them. As such, shared memory nodes, which are fundamental building blocks of HPC infrastructures, are experimenting an increase in the number of sockets they integrate. Besides the benefits in terms of a unified flat memory address space and large core counts, integrating many sockets into the same node exacerbates its Non-Uniform Memory Access (NUMA) effects, which can become a serious performance bottleneck if they are not properly handled.

To mitigate NUMA effects, techniques consisting in migrating threads, memory pages or

access time. Although these techniques are effective, they do not exploit any kind of application-specific information to predict accesses to remotely allocated data before a particular software component starts displaying this behavior. Oppositely, other approaches transfer the NUMA management responsibility to the programmer exploiting information at the application source code level to carry out NUMA-aware scheduling decisions [4], [5]. However, these approaches require significant code refactoring and programmer effort to be effective.

In this work, we show a novel approach to overcome the limitations of already existing methods for task-based programming models. Our techniques automatically mitigate NUMA effects on multiple NUMA-domain nodes without any kind of specific programmer intervention or application source code change. Our approach leverages runtime system metadata to exploit control and data dependences between the serial parts of parallel workloads and optimally schedule them in the context of a multi-socket NUMA node.

II. PARTITIONING THE TASK DEPENDENCY GRAPH (TDG) TO MITIGATE NUMA EFFECTS

A. Dependence Easy Placement (DEP)

Under the Dependence Easy Placement (DEP) policy, tasks are scheduled to the socket where most of their data dependences are allocated. To figure out which specific socket contains a particular block of data, the runtime system keeps a table to map blocks to sockets. The first address of a block is used as its identifier. Tasks that have no inputs, i.e., initialization tasks, are assigned to sockets via a round-robin fashion if most of its output is not allocated yet. In our approach there is a parameter to set the stride of the round-robin approach. When the task to be scheduled is not an initialization task and there is a tie between two or more sockets in terms of the tasks' dependences they contain, the socket is randomly chosen.

oning (RIP)

Partitioning (RIP) policy, task scheduling decisions are based on graph partitioning techniques. The TDG is built at runtime by leveraging information in terms of task dependences. The graph is updated every time new tasks are instantiated and partitioned once the execution goes through a barrier point or a limit in terms of the total number of tasks contained in the graph is reached, which we call the *window size* limit. The graph partitioning algorithm uses the TDG as input, weights its edges depending on the amount of bytes they represent and assigns tasks to

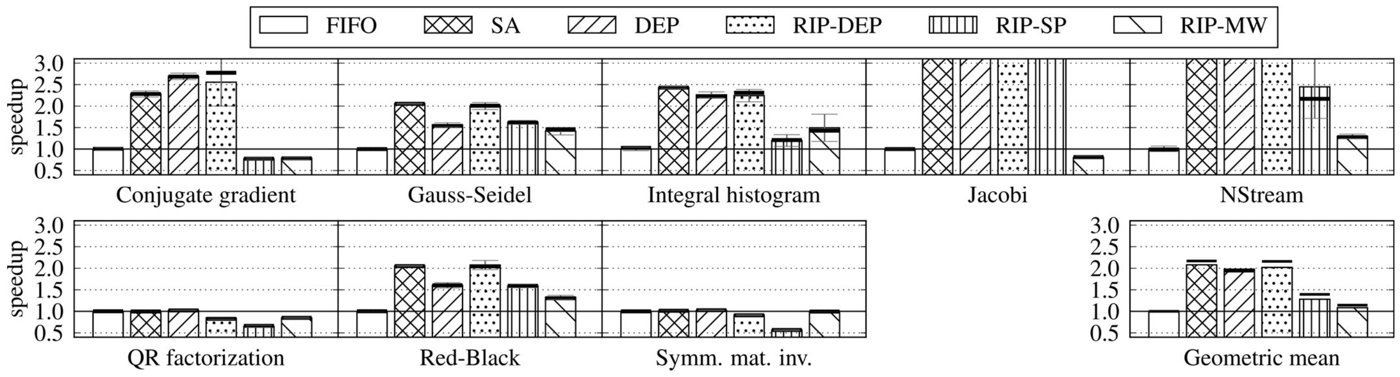

Fig. 1. Speedup results in an SGI Altix UV100 using 2 sockets. For Jacobi using SA, DEP, RIP-DEP and RIP-SP the values are 3.3; for NStream using SA, DEP and RIP-DEP the values are 4.1.

a particular socket taking into account the machine NUMA distances contained in the firmware. Once they are assigned to a socket, they are moved to the corresponding queue. For those tasks that are assigned to a given socket before they are ready to run, they are pushed to the correct queue once their dependences are met, without getting to the temporary queue at all. Once the initial subgraph has been partitioned, we consider three possible options to proceed:

1)RIP with Dependence Easy Placement (RIP-DEP): The RIP-DEP technique consists in propagating the partition obtained from the initial subgraph by taking into account where the tasks' input data resides. As such, if most of the input data of a given task resides in a particular socket, this task is assigned to be run on that socket. This technique is close to the DEP approach, but while DEP applies simple round-robin mechanisms, RIP-DEP partitions the graph.

2)RIP with Socket Propagation (RIP-SP): RIP-SP propagates the partition obtained from the initial subgraph by considering the placement of the predecessors of a particular task and weighting them according to the total amount of data they transfer to the targeted task. As such, the socket where most of the predecessors were executed tends to be chosen by the RIP-SP policy.

3)RIP with Moving Window (RIP-MW): In this case, the graph partitioner is run many times throughout the execution of the program. Once the subgraph contains a particular amount of tasks, the window size, or a barrier point is reached, the partitioning algorithm is run. Once the partitioner finishes its job, the oldest tasks are flushed from the graph and a new subgraph starts getting built, with an intersection between consecutive windows. This intersection is considered to allow the graph partitioner to exploit the already made partitions to generate the new ones, which is an optimization that aims at reducing the overhead.

I. EVALUATION

We evaluate the performance of the proposed mechanisms considering 8 different applications against two schedulers from the Nanos++ runtime:

First-In First-Out (FIFO) task scheduler that is unaware of data location. This is the baseline.

Socket Aware (SA) scheduler, which is driven by annotations at the source code level.

The results for an SGI Altix UV100 machine, with Intel Westmere-EX processors, are shown in Fig. 1. On average, DEP achieves speedups of 1.98× over the FIFO approach, while RIP-DEP, RIP-SP and RIP-MW achieve improvements of 2.02×, 1.28× and 1.09× respectively.

The strong NUMA effects of the Altix system allow the RIP-DEP technique to clearly beat the DEP approach due to the excellent speedups it achieves when dealing with the Gauss-Seidel and the Red-Black applications. The DEP technique is not able to emulate the optimal partition. In contrast, the partition obtained by RIP-DEP is close to the best possible one, which allows the RIP-DEP technique to achieve speedups of 2.01× in Gauss-Seidel and 2.08× in Red-Black, very close to the ones achieved by SA, which is 2.05× faster than FIFO in both cases.

ACKNOWLEDGMENT

This work has been supported by the RoMoL ERC Advanced Grant (GA 321253), by the European HiPEAC Network of Excellence and by the Spanish Ministry of Economy and Competitiveness under contract Computación de Altas Prestaciones VII (TIN2015-65316-P). M. Casas has been partially supported by the Secretary for Universities and Research of the Ministry of Economy and Knowledge of the Government of Catalonia and the Co-fund programme of the Marie Curie Actions of the European Union's 7th FP (contract 2013 BP B 00243). M. Moretó has been partially supported by the Ministry of Economy and Competitiveness under Juan de la Cierva postdoctoral fellowship number JCI-2012-15047.

REFERENCES

[1] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth, "Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems," in Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, 2013, pp. 381–394. doi:10.1145/2451116.2451157

[2] M. Diener, E. H. M. Cruz, P. O. A. Navaux, A. Busse, and H.-U. Heiß, "kMAF: Automatic Kernel-level Management of Thread and Data Affinity," in Proceedings of the 23rd International Conference on Parallel Architectures and Compilation, 2014, pp. 277–288. doi:10.1145/2628071.2628085

[3] M. M. Tikir and J. K. Hollingsworth, "Hardware monitors for dynamic page migration," J. Parallel Distrib. Comput., vol. 68, no. 9, pp. 1186–1200, 2008. doi:10.1016/j.jpdc.2008.05.006

[4] R. Al-Omairy, G. Miranda, H. Ltaief, R. M. Badia, X. Martorell, J. Labarta, and D. Keyes, "Dense Matrix Computations on NUMA Architectures with Distance-Aware Work Stealing," Supercomput. Front. Innov., vol. 2, no. 1, pp. 49–72, Jan. 2015. doi:10.14529/jsfi150103

[5] R. Vidal, M. Casas, M. Moretó, D. Chasapis, R. Ferrer, X. Martorell, E. Ayguadé, J. Labarta, and M. Valero, "Evaluating the impact of OpenMP 4.0 extensions on relevant parallel workloads," in OpenMP: Heterogenous Execution and Data Movements: 11th International Workshop on OpenMP, IWOMP 2015, Aachen, Germany, October 1-2, 2015, Proceedings, vol. 9342, C. Terboven, B. R. de Supinski, P. Reble, B. M. Chapman, and M. S. Müller, Eds. Cham: Springer International Publishing, 2015, pp. 60–72. doi:10.1007/978-3-319-24595-9_5