POSTER: Graph partitioning applied to DAG scheduling to reduce NUMA effects

Isaac Sánchez Barrera
Barcelona Supercomputing Center
Universitat Politècnica de Catalunya
isaac.sanchez@bsc.es

Marc Casas
Barcelona Supercomputing Center
Universitat Politècnica de Catalunya
marc.casas@bsc.es

Miquel Moretó
Barcelona Supercomputing Center
Universitat Politècnica de Catalunya
miquel.moreto@bsc.es

Eduard Ayguadé
Barcelona Supercomputing Center
Universitat Politècnica de Catalunya
eduard.ayguade@bsc.es

Jesús Labarta
Barcelona Supercomputing Center
Universitat Politècnica de Catalunya
jesus.labarta@bsc.es

Mateo Valero
Barcelona Supercomputing Center
Universitat Politècnica de Catalunya
mateo.valero@bsc.es

Abstract

The complexity of shared memory systems is becoming more relevant as the number of memory domains increases, with different access latencies and bandwidth rates depending on the proximity between the cores and the devices containing the data. In this context, techniques to manage and mitigate non-uniform memory access (NUMA) effects consist in migrating threads, memory pages or both and are typically applied by the system software.

We propose techniques at the runtime system level to reduce NUMA effects on parallel applications. We leverage runtime system metadata in terms of a task dependency graph. Our approach, based on graph partitioning methods, is able to provide parallel performance improvements of 1.12× on average with respect to the state-of-the-art.

CCS Concepts • **Computing methodologies** → *Parallel computing methodologies*; • **Computer systems organization** → Multicore architectures;

Keywords shared memory, scheduling, task-based programming model, NUMA, graph partitioning

ACM Reference Format:
Isaac Sánchez Barrera, Marc Casas, Miquel Moretó, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. 2018. POSTER: Graph partitioning applied to DAG scheduling to reduce NUMA effects. In *PPoPP '18: Principles and Practice of Parallel Programming, February 24–28, 2018, Vienna, Austria.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3178487.3178535

PPoPP '18, February 24–28, 2018, Vienna, Austria

1 Introduction

After the stagnation of CPU clock frequency, the performance of computing infrastructures is increasing thanks to the addition of more computing units, augmenting the number of hardware components, heterogeneity and complexity in parallel systems. This means an increase in the number of sockets they integrate, with the benefits of large flat memory address space and core counts but, at the same time, a stress in the *non-uniform memory access* (NUMA) effects.

Techniques at the OS level for mitigating these effects include migrating threads, memory pages or both [2, 3, 8]. However, they do not exploit application-specific information to predict accesses to remotely allocated data before the software starts showing this behavior, they take action when the application is already suffering from remote memory accesses. On the other hand, other approaches transfer responsibility to the programmer [5, 9], which decreases the code understability.

In this work, we consider techniques to be used in task-based applications which exploit the information contained in their *task dependency graph* (TDG). This structure is a *directed acyclic graph* (DAG) where nodes correspond to fragments of sequential code and edges are the task precedences defined by the data and control dependencies. To exploit this information, we consider either techniques that analyze the TDG by means of a simple heuristic or techniques based on advanced graph partitioning algorithms.

2 Exploiting the task dependency graph to reduce NUMA effects

2.1 Locality-aware scheduling (LAS)

By *locality-aware scheduling* (LAS), we refer to the approach proposed by Drebes et al. [4] in terms of a dynamic task and data placement policy based on two concepts: *deferred allocation*, in which the memory to store task output data is not allocated until the task placement is known, and *enhanced workpushing*, where tasks are scheduled in the NUMA region containing most of their data dependencies.

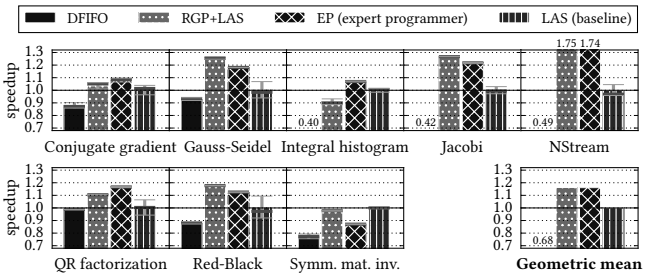

Figure 1. Results in an Atos Bull bullion S16 using 8 sockets, 4 cores/socket.

At the time of scheduling a task, the runtime explores its dependencies and weights the sockets using the size of the allocated input and output data. Then, the task is scheduled to the socket with the highest weight. If most of the data is unallocated, the final socket is randomly chosen among all sockets available to the runtime system. In case of a tie, the socket is chosen randomly among the tied ones. This is also a *propagation* technique: when the data is physically allocated (using some heuristic), tasks can be scheduled to run in cores that are near the memory modules containing the data, thus consuming it faster.

2.2 Runtime graph partitioning (RGP)

Under the *runtime graph partitioning* (RGP) family of policies, task scheduling decisions are based on graph partitioning techniques. The TDG is built at run time by leveraging information in terms of task dependencies. 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 —the *window size* limit— is reached. The partitioning algorithm uses the TDG as input, weights its edges depending on the amount of bytes they represent and assigns tasks to a particular part (corresponding with a specific socket) taking into account the memory latencies.

The initial subgraph, given by the first *window size* tasks, is partitioned with SCOTCH [6] while the runtime system creates new tasks. If tasks can be executed (once their input dependencies are solved) but the partition is still pending, they are stored in a temporary queue. Afterwards, there are different ways to propagate the partition: one option is to use a locality-aware policy, which corresponds to RGP+LAS.

2.2.1 RGP w/ locality-aware scheduling (RGP+LAS)

This technique based in graph partitioning propagates the partition obtained from the initial subgraph by using the physical location of the tasks' data dependencies. More specifically, this approach uses LAS to propagate the partition to the rest of the TDG. The main difference between using just LAS and RGP+LAS is the way of scheduling the first tasks: RGP+LAS uses the graph structure to do the initial schedule.

3 Evaluation

We evaluate the performance of RIP+LAS considering eight applications against other techniques: *Distributed first-in first-out* (DFIFO), unaware of data allocation, each task goes to a different CPU in a cyclic order; *Expert programmer* (EP), in which the schedule is hardcoded in the source code of the benchmark, and *Locality-aware scheduler* (LAS), explained above, used as the baseline.

The applications are written using OpenMP and built with the Nanos++ v0.10 runtime system and the Mercurium 2.0.0 (rev. c5a91d5) compiler [1, 7]. Figure 1 shows the results of the techniques in an Atos Bull bullion S16 machine using 8 sockets and 4 cores/socket.

Acknowledgments

This work has been partially supported by the RoMoL ERC Advanced Grant (GA 321253), the European HiPEAC Network of Excellence and the Spanish Government (contract TIN2015-65316-P). I. Sánchez Barrera has been supported by the Spanish Government under Formación del Profesorado Universitario fellowship number FPU15/03612.

References

[1] Jairo Balart, Alejandro Duran, Marc Gonzàlez, Xavier Martorell, Eduard Ayguadé, and Jesús Labarta. 2004. Nanos Mercurium: A Research Compiler for OpenMP. In *EWOMP*. http://people.ac.upc.edu/aduran/papers/2004/mercurium_ewomp04.pdf

[2] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. 2013. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. *SIGARCH Comput. Archit. News* 41 (2013), 381–394. https://doi.org/10.1145/2490301.2451157

[3] Matthias Diener, Eduardo H. M. Cruz, Philippe O. A. Navaux, Anselm Busse, and Hans-Ulrich Heiß. 2014. kMAF: Automatic Kernel-Level Management of Thread and Data Affinity. In *PACT*. 277–288. https://doi.org/10.1145/2628071.2628085

[4] Andi Drebes, Antoniu Pop, Karine Heydemann, Albert Cohen, and Nathalie Drach. 2016. Scalable Task Parallelism for NUMA: A Uniform Abstraction for Coordinated Scheduling and Memory Management. In *PACT*. 125–137. https://doi.org/10.1145/2967938.2967946

[5] Rabab al-Omairy, Guillermo Miranda, Hatem Ltaief, Rosa M. Badia, Xavier Martorell, Jesús Labarta, and David Keyes. 2015. Dense Matrix Computations on NUMA Architectures with Distance-Aware Work Stealing. *Supercomput. Front. Innov.* 2 (2015), 49–72. https://doi.org/10.14529/jsfi150103

[6] François Pellegrini. 1994. Static Mapping by Dual Recursive Bipartitioning of Process Architecture Graphs. In *SHPCC*. https://doi.org/10.1109/SHPCC.1994.296682

[7] Xavier Teruel, Xavier Martorell, Alejandro Duran, Roger Ferrer, and Eduard Ayguadé. 2007. Support for OpenMP Tasks in Nanos V4. In *CASCON*. https://doi.org/10.1145/1321211.1321241

[8] Mustafa M. Tikir and Jeffrey K. Hollingsworth. 2008. Hardware Monitors for Dynamic Page Migration. *J. Parallel Distrib. Comput.* 68 (2008), 1186–1200. https://doi.org/10.1016/j.jpdc.2008.05.006

[9] Raul Vidal, Marc Casas, Miquel Moretó, Dimitrios Chasapis, Roger Ferrer, Xavier Martorell, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. 2015. Evaluating the Impact of OpenMP 4.0 Extensions on Relevant Parallel Workloads. In *IWOMP*. 60–72. https://doi.org/10.1007/978-3-319-24595-9_5