A methodology approach to compare performance of parallel programming models for shared-memory architectures

Gladys Utrera, Marisa Gil, Xavier Martorell

Computer Architecture Department. Universitat Politècnica de Catalunya
c/ Jordi Girona, 1-3, 08034 Barcelona, Catalunya, Spain
gutrera@ac.upc.edu, marisa@ac.upc.edu, xavim@ac.upc.edu

Abstract. The majority of current HPC applications are composed of complex and irregular data structures that involve techniques such as linear algebra, graph algorithms, and resource management, for which new platforms with varying computation-unit capacity and features are required. Platforms using several cores with different performance characteristics make a challenge the selection of the best programming model, based on the corresponding executing algorithm. To make this study, there are approaches in the literature, that go from comparing in isolation the corresponding programming models primitives to the evaluation of a complete set of benchmarks. Our study shows that none of them may provide enough information for a HPC application to make a programming model selection. In addition, modern platforms are modifying the memory hierarchy, evolving to larger shared and private caches or NUMA regions making the memory wall an issue to consider depending on the memory access patterns of applications. In this work, we propose a methodology based on Parallel Programming Patterns to consider intra and inter socket communication. In this sense, we analyze MPI, OpenMP and the hybrid solution MPI/OpenMP in shared-memory environments. We demonstrate that the proposed comparison methodology may give more accurate predictions in performance for given HPC applications and consequently a useful tool to select the appropriate parallel programming model.

Keywords: MPI · OpenMP · NUMA · HPC · Parallel programming patterns.

1 Introduction

Current HPC platforms are composed of varying computation units capacities and features connected by diverse, increasingly powerful and complex networks to provide better performance not only for large size messages but also for massive receive/send from multiple nodes. These are characteristics foreseeable for the Exascale era.

From the point of view of the software, applications also tend to be composed of different data structures and the corresponding algorithms to read and modify this data. In addition, based on the data size and the order in which the data is processed, the performance in terms of scalability or reliability, can be affected depending on the programming model in use.

This scheme has led to several approaches considering the use of pure Message Passing library Interface (MPI) [**?**] versus OpenMP [**?**] primitives inside a node or exploring several levels of hybrid message-passing and shared-memory proposals to take advantage of the different cores' characteristics. Furthermore, modern platforms are also modifying the memory hierarchy differences, evolving to larger shared and private caches or NUMA (Non-Uniform Access Memory) regions.

UMA (Uniform Memory Access) architectures, commonly referred to as SMP (Symmetric Multiprocessing), have equal memory access latencies from any processor. On the contrary, NUMA architectures are organized as interconnected SMPs and the memory access latencies may differ between different SMPs. In this situation, the memory wall is an issue to consider depending on the memory access patterns the executing application exhibits: data message size, varied size of synchronization or mutex areas; and an inter-socket evaluation is necessary.

This increasing complexity at both low- and high-level makes a challenge the selection of the best programming model, to achieve the best performance on a specific platform. In this work, we take a pattern-based approach to analyze application performance, based on the scalability achieved, including data locality.

Contributions of this work:

- Review a methodology currently used to enhance parallel programming languages with the objective of comparing them.
- Performance comparison between MPI and OpenMP under the stencil and reduce parallel patterns.

The rest of the paper is organized in the following way: Section II introduces background, related work and our motivation; Section III presents the experimental results. Finally, in Section IV are the conclusions and future work.

2 Background and Motivation

The choice of the parallel programming model can be determinant in performance for a given application.

Standard programming models may ensure portability. However, when combined with the platform architecture there is not a general best approach.

In this work we focus on two standards: Message Passing Model (MPI) and OpenMP. OpenMP [**?**] is the de facto standard model for shared memory systems and MPI [**?**] is the de facto standard for distributed memory systems.

In the following subsections we introduce briefly MPI and OpenMP programming models. After that we make an overview of existing performance comparison studies, which conduct us to the motivation of our proposal.

2.1 OpenMP

OpenMP is a shared-memory multiprocessing Application Program Inference (API) for easy development of shared memory parallel programs. It provides a set of compiler directives to create and synchronize threads, and easily parallelize commonly used parallel-patterns. To that end, it uses a block-structured approach to switch between sequential and parallel regions, which follows the fork/join model. When entering a parallel region, a single thread splits into some number of threads, then when finishing that region, only a sequential thread continuous execution.

2.2 MPI

MPI is a message passing library specification for parallel programming on a distributed environment. In a message passing model, the application is composed of a set of tasks which exchange the data, local or distributed among a certain number of machines, by message passing. There exist several implementations like Intel MPI, and also open source like OpenMPI, MPICH.

Each task in the MPI model has its own address space and the access to others' tasks address space has to be done explicitly with message passing. Data partitioning and distribution to the tasks of the application is required to be programmed explicitly.

MPI provides point-to-point operations, which enable communication between two tasks, and collective operations like broadcast or reduction, which implement communication among several tasks. In addition, communication can be synchronous where tasks are blocked until the message passing operation is completed, or asynchronous where tasks can defer the waiting for the completion of the operation until some predefined point in the program. The size of the data exchanged can be from bytes to gigabytes.

2.3 Related work and Motivation

There is an interesting study by Krawezik et al. [?], which involved the comparison of some communication primitives from OpenMP and MPI, as well work-sharing constructs from OpenMP and evaluations of the NAS Benchmarks in shared-memory platforms. They recommend OpenMP for computing loops and MPI for the communication part. On the other hand, Kang et al. [?] suggest that for problems with small data sizes OpenMP can be a good choice, while if the data is moderate and the problem computation-intensive then MPI can be considered a framework. Another work by Piotrowski [?] which evaluates square Jacobi relaxation under pure MPI the hybrid version with OpenMP, showed that in a cluster of shared-memory nodes, none of the hybrid versions outperformed pure MPI due to longer MPI point-to-point communication times. In the work by Qi et al. [?] they also use the NAS benchmarks for the evaluation and show that with simple OpenMP directives the performance obtained is as good a with shared-memory MPI on IA64 SMP. They also claim that even OpenMP has easy programming, improper programming is likely to happen leading to inferior performance. In this sense, in the work by [?] they conclude that parallelization with OpenMP can be obtained with little effort but the programmer has to be aware of data management issues in order to obtain a reasonable performance. The performance comparison by Yan et al. in their work [?] on the 3D Discrete Element Method of ellipsoid-like complex-shaped particles, examine memory aspects, task allocation as well as variations in the combined approach MPI/OpenMP. They conclude that Pure MPI achieves both lower computational granularity (thus higher spatial locality) and lower communication granularity (thus faster MPI transmission) than hybrid MPI-OpenMP in 3D DEM, where computation dominates communication, and it works more efficiently than hybrid MPI-OpenMP in 3D DEM simulations of ellipsoidal and poly-ellipsoidal particles.

The reviewed works above perform interesting analysis on selected applications and/or algorithms but none of them provide enough information for a given HPC application other than the ones specifically analyzed, to make a proper programming model selection. Even more, there are contradictory recommendations derived from the fact that they were evaluated with different platform characteristics which is a relevant factor.

2.4 Selected patterns

Pattern-based parallel programming consists on a set of customizable parallel patterns used to instantiate blocks of applications. This approach is being empowered in the last years with the objective to provide parallel programming languages with additional features that bring more flexibility and reduce the effort of parallelization. In the work by De Sensi et al. [?], the authors demonstrate the feasibility of the approach.

In this work, we exploit the idea of pattern-based parallel programming languages to compare them in terms of performance.

We selected the *loop-of-stencil-reduce* pattern for being representative of many HPC applications. This parallel pattern is general enough to subsume other patterns like *map, map-reduce, stencil, stencil-reduction* and their usage in a loop [?].

Below we can see a pseudocode of our MPI and OpenMP implementation versions:

```
{ MPI version }
  data-distribution
  loop
     start-border-recv
     stencil-reduce (in, out)
     start-border-send
     complete-border-exchange
     swap-matrix-in-out
  end loop
```

```
{ OpenMP version }

  loop
     #pragma omp parallel for
      stencil-reduce (in, out)
     swap-matrix-in-out
  end loop
```

In order to make a fair comparison, we do not take into account any converge condition; the main loop has a fixed number of iterations. There are no dependencies within the blocks inside the stencil algorithm. At every loop there is an input matrix and an output matrix. Before next iteration, we perform a swap between both matrix. The parallelization is block-based. This means that a task perform the stencil algorithm on a block matrix like the one shown in Fig. **??**.

The memory access pattern for our implementation is a 5-point Stencil shown in Fig. **??**. In dark grey is shown the data shared in the border of each block. Each task share data with top, bottom, left and right tasks.

3 Experimental results

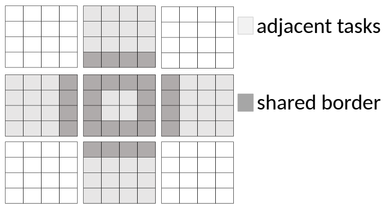

Fig. 1. Memory access pattern for a 5-point stencil

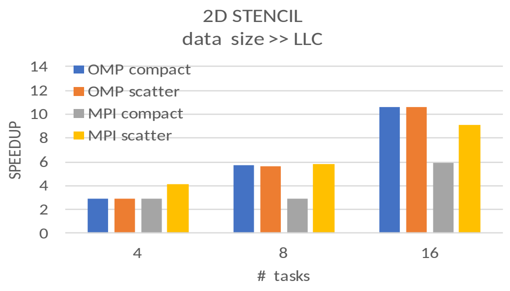

Fig. 2. OpenMP and MPI loop-stencil-reduce parallel pattern performance comparison when datasize does not fit in LLC. Bigger is better.

In this section we show the performance results from the evaluation of the implementation in MPI, OpenMP and MPI/OpenMP of the selected parallel pattern.

The executions were performed on NordIII [**?**], a supercomputer based on Intel SandyBridge processors, with Linux Operating System. It has 3.056 homogeneous compute nodes (2x Intel SandyBridge-EP E5-2670/1600 20M 8-core at 2.6 GHz) with 2GB per core. We use the Intel MPI library 4.1.3.049 and C compiler Intel/2017.4.

The evaluations are performed varying the number of tasks (1-16), task allocation (within a `NUMA-node` or inter `NUMA`-nodes) and data size. For the data size we consider two cases: 1) fits in the last-level cache (LLC); 2) does not fit in the LLC but fits in main memory.

The work distribution among tasks in our test is well-balanced, so load balancing issues are not tackle in this analysis. There are no dependencies between tasks during a given step, so tasks are embarrasingly parallel. The communication is performed between adjacent tasks (exchange of borders), but source data is from the previous step. Notice that, the input matrix is only read and the output matrix is only written (see the pseudocode at Section **??**).

The initialization of data structures is done in parallel taking care of the first touch Operating system data allocation policy to minimize remote accesses during calculation. Tasks are binded to cores in order to ensure allocation policies: 1) *compact*, that is in the same NUMA node; 2) *scatter*, that is equally distributed across NUMA nodes. The results are showed in Fig. **??** and Fig. **??**.

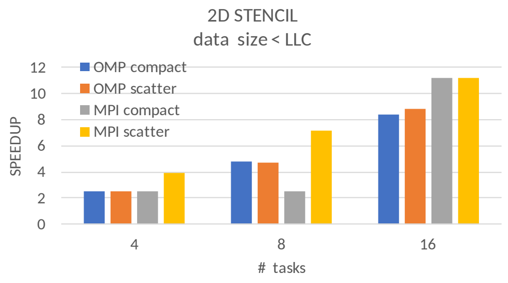

Fig. 3. OpenMP and MPI loop-stencil-reduce parallel pattern performance comparison when datasize does not fit in LLC. Bigger is better.

We can observe in Fig. **??** that the data size does not fit in the shared NUMA-node LLC, when the NUMA node is full (8 tasks) or both NUMA nodes are full (16 tasks) MPI performance degrades dramatically with respect to OpenMP. However, if data fits in LLC, as shown in Fig. **??**, then MPI has better performance when having 8 tasks allocated in different NUMA nodes or when the NUMA node is full (16 tasks).

The *stencil* parallel-pattern is characterized for having memory accesses which are updated by other tasks in previous steps. This means that such data has to be exchanged before doing the current calculation. There are memory accesses not only within the block of data processed by a given task, but also for data managed by neighbouring tasks (adjacent blocks). For shared-memory programming models, collaborating tasks allocated to different NUMA-nodes have a well-documented effect on memory access performance (e.g. OpenMP) [**?**]). This is not the case for distributed memory programming models (e.g. MPI). In parallel programming languages where memory is not shared among tasks (i.e. MPI), this is exchanged explicitely between steps (i.e. MPI_Isend and MPI_Irecv for our current implementation shown in section **??**. However, once the data is brought becomes local (task allocation is transparent).

Taking all this into account we can appreciate the NUMA effect when data fits in LLC. As memory is not an issue for this data size, MPI obtains better performance than OpenMP. The data managed by each MPI task is local. On the other hand, when data does not fit in LLC, then as MPI duplicates data faces memory problems with the consequent increment in cache misses, degrading performance (Fig. **??**). Notice that if allocating tasks in different NUMA nodes, this effect can be alleviated. Despite LLC misses in MPI for small data sizes are larger than LLC misses in OpenMP, the remote accesses generated by adjacent tasks penalize bringing better performance for MPI.

The hybrid approach MPI/OpenMPI performed worse than MPI and OpenMP in isolation when allocated one MPI task per NUMA-node, which means in our experimental platform, 2 task per node. The added memory overhead plus the thread creation and other overheads (e.g. remote memory access) do not compensate in performance. We believe that for larger NUMA-nodes this results may be different. We are currently undergoing this study.

In conclusion, for small data sizes and small number of tasks, both parallel programming languages can achieve the same performance no matter where tasks are allocated (lesser tasks, lesser interaction between them). When incrementing the number of tasks, there is more interaction between them penalizing remote accesses for OpenMP, but duplicating data at the same time for MPI.

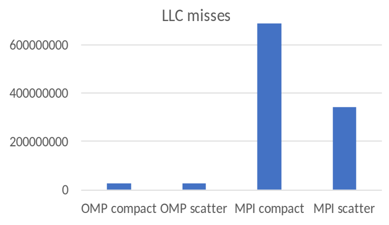

Fig. 4. Last Level Cache misses when datasize does not fit

4 Conclusions and Future Work

Based on the idea of pattern-based parallel programming we compared two standard like OpenMP, MPI and the hybrid approach MPI/OpenMP on a multicore platform with different memory accesses characteristics. We selected a scheme pattern representative of many HPC applications like a loop of stencil and reduce. We showed that both parallel programming languages can show different performance characteristics in applications depending on the data size being processed and the data locality which requires extra and conscious programming effort. Considering the hybrid approach to solve gaps in the context of a shared-platform showed to not work as good as separately.

In this work we only focused on data-parallel algorithms. We are planning to extend our study to other parallel patterns like pipeline and unstructured. In addition, the platform as already shown in terms of memory accesses has a significant role so we plan to enrich the work by studying heterogeneous platforms.

Acknowledgements.
This research was supported by the following grants Spanish Ministry of Science and Innovation (contract TIN2015-65316), the Generalitat de Catalunya (2014-SGR-1051) and the the European Commission through the HiPEAC-3 Network of Excellence (FP7/ICT-217068).