Advances in Engineering Software 84 (2015) 77–84

Contents lists available at ScienceDirect

Advances in Engineering Software

journal homepage: www.elsevier.com/locate/advengsoft

Accelerating engineering software on modern multi-core processors

Edson Borin [a,*], Philippe R.B. Devloo [b], Gilvan S. Vieira [a], Nathan Shauer [b]

[a] *Institute of Computing, University of Campinas, Brazil*
[b] *Faculty of Civil Engineering, University of Campinas, Brazil*

ARTICLE INFO

Article history:
Available online 16 January 2015

Keywords:
Parallel programming
Parallel processing
Cache-coherent Non Uniform Memory Access
Finite Element Methods
Multi-core processors
Shared memory

ABSTRACT

Recent multi-core designs migrated from Symmetric Multi Processing to cache coherent Non Uniform Memory Access architectures. In this paper we discuss performance issues that arise when designing parallel Finite Element programs for a 64-core ccNUMA computer and explore solutions for these issues. We first present the overview of the computer architecture and show that highly parallel code that does not take into account the aspects of the system memory organization scales poorly, achieving only 2.8× speedup when running with 64 threads. Then, we discuss how we identified the sources of overhead and evaluate three possible solutions for the problem. We show that the first solution does not require the application's code to be modified, however, the speedup achieved is only 10.6×. The second solution enables the performance to scale up to 30.9×, however, it requires the programmer to manually schedule threads and allocate related data on local CPUs and memory banks and rely on ccNUMA aware libraries that are not portable across operating systems. Also, we propose and evaluate "copy-on-thread", an alternative solution that enables the performance to scale up to 25.5× without relying on specialized libraries nor requiring specific data allocation and thread scheduling. Finally, we argue that the issues reported only happen for large data sets and conclude the paper with recommendations to help programmers to design algorithms and programs that perform well on such kind of machine.

1. Introduction

The continuous shrinking of transistors size has allowed microprocessor designers to put an ever-increasing number of computing cores into a single microprocessor chip. To make it easier for operating systems designers and programmers to take advantage of these extra cores, current microprocessor designs provide a single view of the memory system to every single core; in other words, the memory is shared among all cores. This model does not require programmers to explicitly move data between systems or memory modules when performing computation on different cores, which reduces the program complexity and allow programmers to code parallel programs with libraries such as pthreads [1] or Intel Thread Building Blocks (TBB) [2].

In order to enable performance to scale together with core count, recent designs migrated from SMP (Symmetric Multi Processing) to ccNUMA (cache coherent Non-Uniform Memory Access) based architectures. Similar to the SMP approach, the ccNUMA allows code running in any core to access any memory word. However, the memory access latency varies accordingly to the memory bank and the core position on the system, hence, parallel programs performance may vary accordingly to the threads and data location on the system.

Devloo et al. [3] implemented the substructuring technique proposed by Dohrmann [4] to parallelize a Finite Element software and accelerate its execution on multicore processors. The algorithm was tested on an 8-core SMP system and achieved almost linear speedup. However, when executing the code on a 64-core ccNUMA system the observed speedup of the algorithm was much less than linear [5]. In this paper we discuss performance issues that arise when designing parallel Finite Element Method programs for a parallel multi-core ccNUMA computer and explore solutions for these issues. This paper is based upon Borin and Devloo [5] and includes the following additional contributions:

- We propose "copy-on-thread", a heuristic that relies on the "first-touch" policy to transparently move data to local memory banks before computation.
- We present experimental results and show that, even at the cost of copying data, the "copy-on-thread" heuristic can improve performance significantly and enables the performance to scale up to 25.5× when running on a 64-core system.

* Corresponding author.

http://dx.doi.org/10.1016/j.advengsoft.2014.12.003
0965-9978/© 2014 Civil-Comp Ltd. and Elsevier Ltd. All rights reserved.

The paper is organized as follows: Section 2 provides background information on SMP and ccNUMA systems and present the related work. Section 3 presents the Finite Element Method software and how it was parallelized. Section 4 discuss how we identified the sources of overhead and provide performance results for the three different solutions. Finally, Section 5 presents our conclusions.

2. Background and related work

Early multi-core designs had two or more cores connected to a single shared main memory. Even though the computing core resources are duplicated, multiple cores read or write data through the same bus or memory controller, which can cause contention when multiple cores try to access the memory at the same time. Fig. 1 illustrates the topology of an early multi-core system with four cores. This model is known as Symmetric Multi Processing, or SMP.

Usually, memory access contention is not a problem on SMP systems with few cores. Most of the time the local instruction and data caches are capable of providing the core with instructions and data, reducing the number of accesses to the main memory. Also, the out-of-order execution model allows the cores to hide some of the latency introduced when the memory is accessed. Hence, with few cores attached to the same memory controller, it is unlikely that the competition for the memory controller will hurt the cores performance.

The increasing number of cores on recent designs changed this scenario. Instead of 2 or 4 cores, up to 64 cores are attached to the same shared memory system. In order to enable performance to scale together with core count, recent designs distributed the memory banks through the system, migrating from SMP (Symmetric Multi Processing) to ccNUMA (cache coherent Non-Uniform Memory Access) architectures. Similar to the SMP approach, the ccNUMA allows code running in any core to access any memory word. However, the memory banks are distributed and controlled by multiple memory controllers, enabling the cores to concurrently access different memory banks. As a result, memory access latency varies accordingly to the memory bank and the core position on the system. Fig. 2 illustrates the topology of a cache coherent Non-Uniform Memory Access architecture with four processing cores and four memory controllers. Each memory controller is connected to a memory bank, to a computing core and to the other memory controllers through the interconnecting bus.

The system illustrated in Fig. 2 allows core 1 to access the memory bank 1 at the same time core 2 accesses memory bank 2. Also, the system allows core 1 to transparently access data on other memory banks through the interconnecting bus, however, since the data has to be routed through the interconnecting bus, it typically takes longer to access data on memory banks that are not next to the processing core.

Despite the fact that different cores can access different memory banks in parallel, to make effective use of it, the data must be allocated on memory banks that are next to the cores that are executing the associated tasks (programs or threads). Notice that

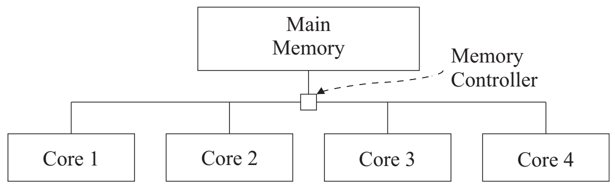

Fig. 1. Symmetric Multi Processing system.

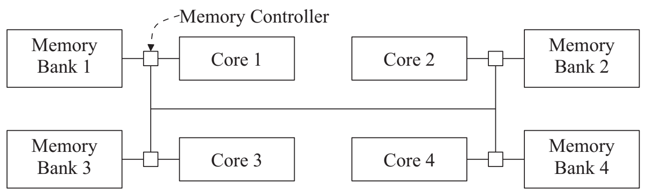

Fig. 2. Cache coherent Non-Uniform Memory Access system.

if all the data gets allocated in a single memory bank, all the cores will compete for the same memory controller, as in the SMP approach. Moreover, the cores that are far away from the memory controller will be penalized with additional latency when accessing the memory bank.

Several research works addressed the performance scalability issue on ccNUMA systems. Tikir and Hollingsworth [6] proposed a profile-driven mechanism to monitor the memory access behavior of an application and decide whether memory pages should be migrated or not. The authors used the hardware performance counters to monitor the memory accesses and trigger page migration, which is performed transparently by the operating system. The mechanism relies on specialized hardware support and modifications on the operating system.

Lof and Holmgren [7] modified the Solaris operating system to include a new mechanism named "affinity-on-next-touch". This mechanism allows the software to inform the operating system that the memory pages should be migrated to the memory bank close to the next cpu that touches (read or write) them. The mechanism requires the user to insert calls to the "affinity-on-next-touch" procedure on the software and the operating system must be modified to migrate the pages on demand.

Yang et al. [8] studied the effects of data locality on the performance of Gaussian 03 code executing on a multi-core ccNUMA system. The authors explored how memory interleaving and dynamic page migration affected the application performance on a SunFire X4600 M2 system and showed that proper data placement can accelerate the software up to 40% when running with 16 threads.

Awasthi et al. [9] proposed an adaptive first-touch page placement policy and a dynamic page-migration mechanism to improve memory access performance on ccNUMA systems. The adaptive first-touch page placement policy takes into account several statistics associated with the memory controller, such as queuing delays, row-buffer hit-rates and bank contention when allocating pages during the first-touch. Their dynamic page-migration mechanism tries to migrate pages between memory banks in order to reduce access delays. The authors report performance gains up to 35%, however, both techniques require the operating system to be modified.

Broquedis et al. [10] proposed ForestGOMP, a runtime system that combines a thread scheduler together with a NUMA-aware memory manager to improve the performance of OpenMP programs on NUMA architectures. Similar to previous work, the runtime system relies on operating system support to migrate data between memory banks connected to different memory controllers.

Ribeiro et al. [11] presented MINAS, an API and a runtime system to allocate and place data on NUMA architectures. The API abstracts to the developer the topology of the architecture and offers mechanisms to determine the initial allocation and placement of application data. The runtime system requires specialized NUMA support from the operating system.

Wittmann and Hager [12] proposed a software layer that reduces adverse effects of dynamic OpenMP and TBB task distribution on ccNUMA systems by sorting tasks into locality queues, each of which is preferably processed by threads that belong to the