Exploring the performance of fine-grained synchronization and data exchange across process boundaries on modern multi-core architectures

Jiri Dokulil[0000−0001−5709−8553] and Siegfried Benkner[0000−0002−6520−2047]

Faculty of Computer Science, University of Vienna, Vienna, Austria
{jiri.dokulil,siegfried.benkner}@univie.ac.at

Abstract. Whether to use multiple threads in one process (MPI+X) or multiple processes (pure MPI) has long been an important question in HPC. Techniques like in situ analysis and visualization further complicate matters, as it may be very difficult to couple the different components in a way that would allow them to run in the same process. Combined with the growing interest in task-based programming models, which often rely on fine-grained tasks and synchronization, a question arises: Is it possible to run two tightly coupled task-based applications in two separate processes efficiently or do they have to be combined into one application? Through a range of experiments on the latest Intel Xeon Scalable (Skylake) and AMD EPYC (Zen) many-core architectures, we have compared performance of fine-grained synchronization and data exchange between threads in the same process and threads in two different processes. Our experiments show that although there may be a small price to pay for having two processes, it is still possible to achieve very good performance. The key factors are utilizing shared memory, selecting the right thread affinity, and carefully selecting the way the processes are synchronized.

Keywords: synchronization · data movement · collocated applications.

1 Introduction

The increasing heterogeneity of HPC architectures has inspired research into different programming approaches. Task-based runtime systems are one example. By abstracting the work into many small tasks with dependencies, the runtime system is given the ability to better control where and when work is being executed, compared to, for example, MPI where the work to be done by a process running on a specific node is strictly prescribed by the application code.

With data movement being one of the major contributors to runtime and power consumption of modern high performance systems, in situ techniques are becoming an important way to improve efficiency of HPC systems. By processing data where and when it is generated, we reduce the resources needed to transfer and store the data.

In this paper, we explore how close we can get to this "best case" scenario with two separate processes. We explore the effects of using shared memory (provided by the operating system) and different synchronization mechanisms. It turns out that even with two separate processes, it is possible to get comparable performance to the single-process scenario. By correctly setting thread affinity and synchronizing the two processes, it is possible to even re-use cached data.

Our main contribution is designing, implementing, and running a benchmark aimed specifically at studying data transfers in the case of two collocated applications. Our findings can be used to aid the design of in situ analysis/visualization applications and other systems that require tight coupling of different processes. The experiments were performed on two different machines, using latest many-core architectures from Intel (Skylake) and AMD (Zen). The four Intel Xeon Scalable CPUs in one machine have a total of 80 cores, while the two AMD EPYC processors in the other server have 64 cores together.

2 Local data exchange

We will use the terms *producer* and *consumer* to denote the two pieces of code responsible for generating (producer) and reading (consumer) the data. Our main concern is the time it takes the consumer to read the data.

The consumer code might immediately follow the producer code, ensuring it runs right after the consumer on the same thread. Or, the consumer could be in a different thread or even in a different process. Then, some synchronization is required. The consumer needs access to the produced data, which is trivial in a single process, where all threads have access to the same data. If the consumer is in a different process, we can use services provided by the operating system to give the consumer direct access to the producer's memory with the data.

Ideally, we would want a CPU core to finish generating (and writing) the data and then switch immediately to the consumer and start reading the data. One way to achieve this is to set affinity of both the producer thread and the consumer thread to the same CPU core. This can easily be done, even if they belong to different processes. We can block the consumer thread on a synchronization primitive (e.g., a semaphore). When the producer is finished, it unblocks the consumer's thread and suspends itself. This forces the scheduler of the operating system to pick a next thread for execution. As the consumer thread is now active and affinitized to the core, it is likely to be picked.

3 Benchmark application design

We have implemented different experiments to try and compare different options for data exchange. They have several things in common. Each experiment uses a single block of memory for the data. The size of the block can be configured. Writing the data is simulated by treating the block as an array of integers and writing 1 to each element. When the data is read, a sum of all elements is produced, to prevent the compiler from optimizing the read away. The sum is performed in a way that makes it easy for the compiler to vectorize it. We measure the time it takes the consumer to read the whole data. Where possible, we also measure the time elapsed between the moment the producer finished writing the data and the moment the consumer starts reading the data. This gives us some estimate of the latency generated by the synchronization. Finally, the experiments can be configured to have the producer and the consumer invalidate their CPU cache before reading the data, to help us check whether the caches are actually being used.

Based on the setup, the producer and consumer might reside in the same thread (then the notification is a NOP and the second for-loop body is moved to the first for-loop), two threads in the same process, or two different processes.

Single thread To obtain a baseline, we consider the scenario where a single thread generates the data and then reads it. This is the closest coupling possible and should provide the best performance, but it may be technically challenging or impossible to use in practice.

File For comparison, we also tested the setup where the producer writes the data to a file which is immediately read by the consumer. To minimize noise, the producer and the consumer are in fact the same thread. The file is written, closed, and immediately opened for reading.

Two threads, synchronization via atomics The next setup is closer to the intended two-process layout, but it uses two threads in the same process. The two threads are synchronized using atomic operations. When the producer finishes writing the data, it atomically writes a flag (with release semantics) which the consumer keeps checking atomically (with acquire semantics).

Two processes, synchronization via atomics The setup in this case is the same as in the previous one, except that the threads belong to different processes and use a shared memory region to exchange data and for the atomics-based synchronization.

Two threads, synchronization via promises To test blocking synchronization, we have used C++11 promises. One promise is used to notify the consumer that the producer is finished and another promise is used to notify the producer that the consumer has finished.

Two processes, synchronization via semaphore With two processes, the promises cannot be used, so a pair of standard named semaphores is used instead, to achieve the same synchronization pattern.

4 Experimental evaluation

The experiments were performed on two different servers running Linux. One server contains four Intel Xeon Scalable Gold 6138 processors (Skylake architecture, 20 cores, AVX-512 support, 32 KB L1 data cache per core, 1 MB L2 cache per core and 27.5 MB last level cache in total). The total number of physical cores is 80, but all experiments were executed with Hyper-Threading enabled, the machine supports 160 hardware threads (logical cores).

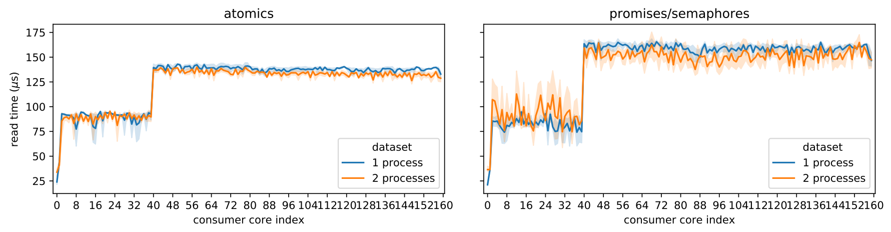

Fig. 1. Read time on the four socket Intel server.

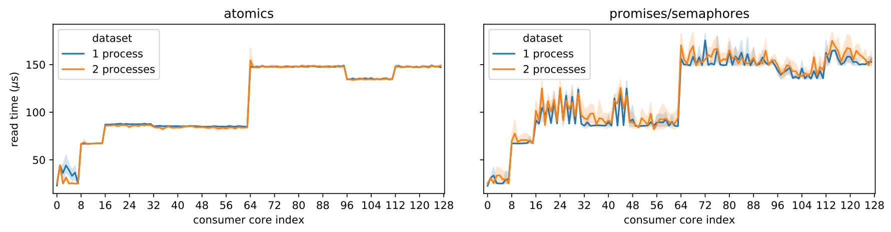

Fig. 2. Read time on the four socket AMD server.

The other server contains two AMD EPYC 7501 processors (Zen architecture, 32 cores, 32 KB L1 data cache per core, 512 KB L2 cache per core and 64 MB last level cache in total). The total number of physical cores is 64 and there are 128 logical cores.

Unless stated otherwise, the performance figures are read times in microseconds. The number of repetitions (the number of data blocks transferred per one application execution) is 100. Each application configuration is executed 5 times. The graphs show the average value of these 5 executions and 95% confidence intervals. Unless explicitly specified, the size of the data exchanged by the producer and the consumer is 1 MB. The producer is always on core 0.

As a baseline, the read time of the single threaded variant is 20.3 µs on the Intel server and 30.8 µs on the AMD server. If a file is used to exchange data, the read time is 518.0 µs and 365.1 µs on the Intel and AMD servers respectively.

4.1 Core selection

As we have already explained, both threads used in the experiment are affinitized to a specific core. The producer always uses core 0. In most experiments, the measured performance depends on the core selected for the consumer. In general, there are four different cases: using the same core (0), using the sibling logical core (1), using another core on the same CPU (2–40 or 2–32 for the Intel and AMD machines respectively), and using a core on other CPUs. Figures 1 and 2 show performance for different combinations on the Intel and AMD servers.

On the AMD CPUs, the performance of 1 process and 2 processes is comparable. When blocking synchronization is used, performance variability increases, but the read performance is very similar. On the Intel CPUs, the results are a

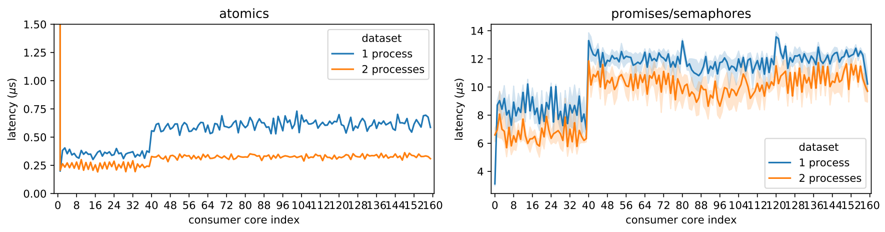

Fig. 3. The latency of synchronization between producer and consumer on the Intel server.

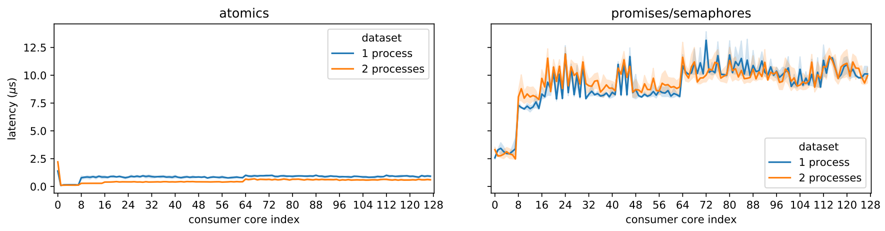

Fig. 4. The latency of synchronization between producer and consumer on the AMD server.

bit more interesting, as 2 processes slightly outperform 1 process in many cases, especially when communicating between different CPUs.

Another important factor to consider is latency. In our case, the time elapsed between the producer finishes and the consumer starts. In this time, the producer needs to send a signal to the consumer and the consumer thread needs to start running. We want to keep it as small as possible in order to not waste resources by having the consumer wait unnecessarily long. The measured performance is shown in Figures 3 and 4.

4.2 Data size

Figures 5 and 6 show the effect of the different data sizes. We use core 0 for producer and core 2 for consumer. Probably the most interesting observation is that while the performance of using 1 or 2 processes is comparable, the latency

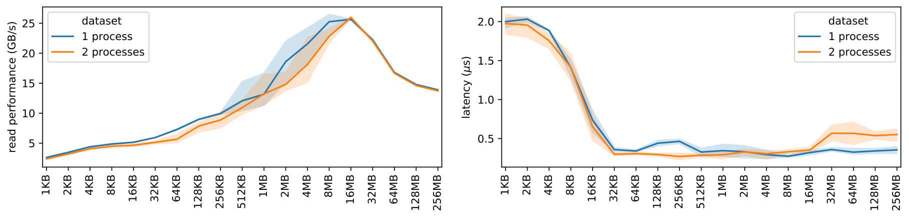

Fig. 5. The effects of data size, on the Intel server.

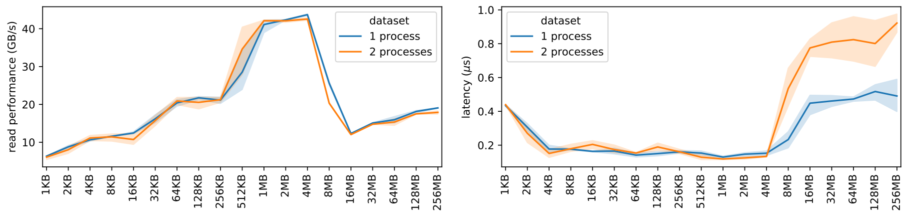

Fig. 6. The effects of data size, on the AMD EPYC server.

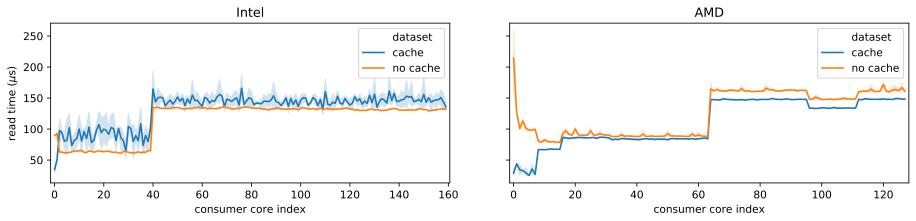

Fig. 7. The effects of cache, on the two architectures.

increases more when larger data sets are exchanged between two processes. The latency increase is probably due to the fact that the data of the consumer thread is forced out of cache. However, as we use different physical cores, the cache in question is the L3 cache. The latency increases the most as the data size approaches the L3 cache size, but it can be seen earlier.

4.3 Cache reuse

Figure 7 compares performance with and without data caching. To examine the no-cache behavior, extra work is performed by the producer and consumer to ensure that the caches contain no useful data. A surprising results is that on the Intel architecture, if the producer and consumer do not reside on the same core, the read performance increases and it also becomes much more stable. We believe this is caused by the fact that forcing the data into main memory is beneficial for data pre-fetching. The data is read in a sequential manner from the main memory, which is a best-case scenario for the hardware pre-fetcher. With caching, as the size of the data is the same as the size of the L2 cache, the data is most likely distributed across L2 and L3 caches, which also explains the much larger performance variability between executions, which is clearly visible from the confidence intervals in Figure 7.

5 Related work

Benchmark results and even various benchmarking software is widely available [2, 4, 7]. Many micro-benchmarks are created in an ad-hoc manner as part of development of a larger system. Memory performance is widely studied in HPC [6], including research focused on shared memory performance [3]. Our work focuses on the specific problem of exchanging data between two processes, which are closely coupled on a single node.

FastFlow [1] implements a system similar to our proposal for a single process, although the low-level implementation details are significantly different. We limited ourselves to the simplest posible C++ and POSIX primitives to eliminate as many variables as possible and explore the performance limits of the hardware and the operating system.

6 Conclusion and future work

The experiments confirm that it is possible to get very good data exchange performance even when two separate processes are used. In case of in situ visualization/analysis, this means that it is possible to achieve very close coupling of the simulation and visualization/analysis steps, even without actually combining both into a single application. Still, care needs to be taken to achieve good performance. In our future work, we will apply the results to our work on dynamic runtime systems and collocated applications [5].

Acknowledgments The work was supported in part by the Austrian Science Fund (FWF) project P 29783 Dynamic Runtime System for Future Parallel Architectures. The final authenticated version is available online at `https://doi.org/10.1007/978-3-030-22750-0_45`.

References

1. Aldinucci, M., Danelutto, M., Kilpatrick, P., Meneghin, M., Torquati, M.: Accelerating code on multi-cores with FastFlow. In: Euro-Par 2011 Parallel Processing. pp. 170–181. Springer Berlin Heidelberg (2011)
2. Bienia, C., Kumar, S., Singh, J.P., Li, K.: The PARSEC benchmark suite: Characterization and architectural implications. In: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques. pp. 72–81. PACT '08, ACM, New York, NY, USA (2008)
3. Bolosky, W.J., Scott, M.L.: False sharing and its effect on shared memory performance. In: Proceedings of the Fourth symposium on Experiences with distributed and multiprocessor systems (1993)
4. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Lee, S., Skadron, K.: Rodinia: A benchmark suite for heterogeneous computing. In: 2009 IEEE International Symposium on Workload Characterization (IISWC). pp. 44–54 (Oct 2009)
5. Dokulil, J., Benkner, S.: Adaptive scheduling of collocated applications using a task-based runtime system. In: 2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD). pp. 41–48 (2018)
6. McCalpin, J.D., et al.: Memory bandwidth and machine balance in current high performance computers. IEEE computer society technical committee on computer architecture (TCCA) newsletter **2**(19–25) (1995)

7. Sakalis, C., Leonardsson, C., Kaxiras, S., Ros, A.: Splash-3: A properly synchronized benchmark suite for contemporary research. In: 2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS) (2016)