Exploring Interprocess Work Stealing for Balanced MPI Communication

Kaiming Ouyang
University of California, Riverside
Riverside, California
kouya001@ucr.edu

Min Si
Argonne National Laboratory
Lemont, Illinois
msi@anl.gov

Zizhong Chen
University of California, Riverside
Riverside, California
chen@cs.ucr.edu

ABSTRACT

Workload balance among MPI processes is a critical consideration during the development of HPC applications. However, because of many factors such as complex network interconnections and irregularity of HPC applications, fully achieving workload balance in practice is nearly impossible. Although interprocess job stealing is a promising solution, existing shared-memory techniques that lack necessary flexibility or cause inefficiency during data access cannot provide an applicable job-stealing implementation. To solve this problem, we propose a new process-in-process (PiP) interprocess job-stealing method to balance communication workload among processes on MPI layers. Our initial experimental results show PiP-based job stealing can efficiently help amortize workload, reduce imbalance, and greatly improve intra- and intersocket ping-pong performance compared with original MPI.

1 INTRODUCTION

High-performance computing (HPC) application developers strive to balance workload among MPI processes in large-scale parallel environments in order to fully utilize hardware resources and obtain as high performance as possible. Unfortunately, achieving complete workload balance in practice is nearly impossible. Complex network interconnects can cause inconsistent communication time in spite of balanced communication workload, and the irregularity of HPC applications inherently hinders equally distributing workload among processes.

To support various applications, we believe that the workload imbalance problem can be resolved at runtime layer (e.g., in MPI) and interprocess job stealing is a promising solution. Although thread-based job stealing is an easily achieved and commonly applied technique [1–3, 5, 6] in multiple-thread environments because of the inherent shared-memory property of threads, these methods are impossible to be used for interprocess job stealing since **no address space is shared and data access is the most challenging part we need to cope with**.

Many existing shared-memory techniques can help us achieve interprocess data access. POSIX shared memory (POSIX-shm) [9] and XPMEM [7] are two commonly adopted. POSIX-shm is not flexible enough, however, because of the restriction of data access to the specific preallocated shared-memory buffer; XPMEM is more flexible and able to provide arbitrary access ability, but system calls and page fault overhead during the first data attachment and access are extremely high, factors that are intolerant during job stealing.

Process-in-process (PiP) [8] is a portable user space library that provides a shared virtual memory space for multiple parallel tasks on a node. PiP enables arbitrary data access between tasks (similar to threads) while maintaining privatized variable sets (similar to processes). PiP can be used to implement MPI and each MPI process is a PiP task.

In this poster, we present a flexible and efficient interprocess job-stealing method in the PiP-based MPI runtime in order to balance MPI communication workload. It allows idle processes (i.e., processes waiting in a blocking MPI call such as *MPI_Wait*) to steal communication work from other processes that are busy in user computation. This method can benefit workload imbalance that may happen in various MPI internal processes such as intranode communication, data packing/unpacking, and data reduction. We showcase the intranode communication aspect in this poster. Our method considers the NUMA-node properties and combines two stealing strategies—localized stealing and mixed stealing—to maximize performance gain. Our initial results show PIP-based job stealing can improve intranode communication performance up to 73.7% in a ping-pong benchmark.

2 BACKGROUND

In this section, we introduce three shared-memory techniques—POSIX-shm, XPMEM, and PiP [8]—and their corresponding data access pattern.

POSIX-shm is supported by POSIX-based operation systems and provides maximal portability. All processes allocate a shared-memory buffer in the beginning, and every process can exchange data only through the shared buffer (as shown in Fig. 1a).

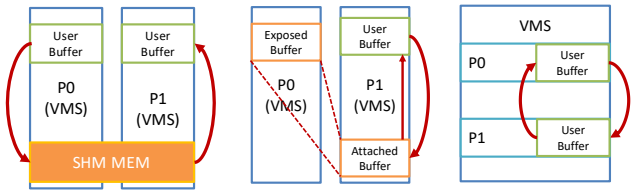
XPMEM is a more flexible shared-memory technique that supports processes accessing any part of the memory of other processes. Processes first call *xpmem_make* to expose a segment of the private buffer; other processes can call *xpmem_attach* to attach necessary segments into its own virtual memory space for data exchange (as shown in Fig. 1b).

PiP loads processes into the same virtual memory space. Every process can access any address of other processes such as threads, and data exchange can be performed by using *memcpy* as long as the buffer address is known in advance (as shown in Fig. 1c). PiP works on user space without any system call and kernel overhead, making PiP the best shared-memory technique for our method.

3 PIP-BASED JOB STEALING

We implement the job stealing approach based on a PiP-aware MPI implementation which is extended from MPICH-3.3b3 [8]. A job in MPI can be defined as any data movement and computation that happens in communication, data packing/unpacking, and reduction. We focus on the intranode communication aspect in this poster.

Consider two-copy based point-to-point (P2P) intranode communication [4]. The sender cuts the source data into multiple fixed



(a) POSIX-shm (b) XPMEM (c) PiP

Figure 1: POSIX-shm, XPMEM, and PiP shared-memory data access patterns: VMS means virtual memory space; red arrow lines mean possible data movements.

length chunks (64 KB in MPICH), copies each chunk into an intermediate "cell," and enqueues the cell into the receiver's receive queue; then, the receiver dequeues and copies the data out of the cell into the receiving buffer. In this scenario, we can define datacopy as a job that includes source to cell and cell to destination. Considering the locality property on a NUMA node, we propose two job-stealing algorithms: localized and mixed job stealing.

A **localized job-stealing algorithm** lets processes steal only jobs of local-NUMA processes. Each process has two local job queues Q_0 and Q_1. The sender that locates at NUMA node 0 enqueues jobs only into Q_0; if the receiver gets the "cell" from NUMA node 0, it also enqueues the receive job into its own Q_0. The same rule applies for NUMA node 1 processes. Idle processes on NUMA node 0/1 steal only jobs from Q_0/Q_1.

Instead of considering only local stealing, a **mixed job-stealing algorithm** allows processes to steal the jobs of remote-NUMA processes. In this algorithm, each process has three local job queues: Q_{any}, Q_0, and Q_1. The sender enqueues all jobs to Q_{any}, and any idle process can steal jobs from it. The receiver gets the "cell" and enqueues the receive job into Q_0 if the "cell" is enqueued by the NUMA 0 process. The same rule applies for Q_1.

Localized stealing is able to achieve maximal memory copy bandwidth in intra-socket communication, but it cannot utilize idle processes on remote sockets. In contrast, mixed stealing can make use of all idle processes, but remote processes can slow down the copy when local processes already saturate the bandwidth. Although in our preliminary study we design the algorithms separately, the goal is to combine them and design a compound stealing algorithm in order to achieve maximal performance boost in all situations.

To further improve the job stealing performance, we apply two optimizations in our implementation as descried below.

Out-of-Order Enqueue: Since the original MPICH does not consider interprocess interaction, the cell enqueue within a message is fully sequential, which hurts the concurrency of job stealing. Therefore, to increase the enqueue performance of stealing, we apply out-of-order enqueue for the chunks of each MPI message where each enqueue can be done without any order. We keep track of the buffer offset of the data in each cell to assure that the data is copied into the correct position in the destination buffer.

Dynamic Cell Size: Smaller cell size can benefit middle-range messages in stealing when idle processes are sufficient. This is because the number of available jobs becomes the bottleneck of the parallelism of stealing and a smaller cell size can produce more jobs and thus increases the job-stealing parallelism.

4 EXPERIMENTAL EVALUATION

The node used in our experiment is equipped with two Intel Xeon CPU E5-2695 v4 (each processor contains 18 cores) and 128 GB of memory. For software, we used MPICH-3.3b3 as our MPI reference implementation compiled by the gcc compiler 4.8.5.

We measure the intrasocket and intersocket ping-pong benchmark performance of original, localized job-stealing and mixed job-stealing MPI. The total number of processes is 36, with 0–17 processes on NUMA node 0 and 18–35 processes on NUMA node 1. For intrasocket ping-pong, ranks 0 and 1 perform ping-pong with each other; for intersocket ping-pong, rank 0 and rank 18 perform ping-pong with each other. All other processes are idle and perform stealing. We vary the message size of ping-pong from 64 KB to 8 MB. Figure 2 present the corresponding results.

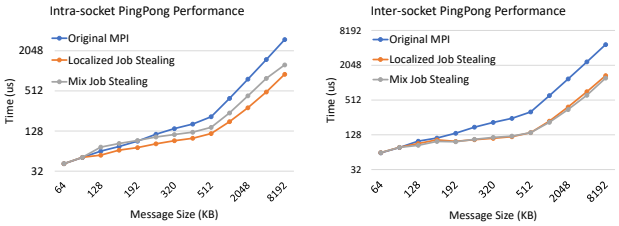


Figure 2: Ping-pong benchmark performance with varying message size.

Compared with original MPI, job stealing brings 69.8% and 73.7% performance rise in intra- and intersocket ping-pong, respectively. The larger the message size, the better the performance we can get using job stealing, because of the smaller percentage of job preparation overhead, more jobs to steal, and well-amortized workload.

As expected, localized stealing achieves better performance than mixed stealing does in intrasocket ping-pong because local processes can saturate local NUMA-node bandwidth by themselves and adding remote processes for stealing will only slow down the average copy speed. Mixed stealing, however, performs slightly better in intersocket ping-pong. This is because it can utilize all idle processes on NUMA nodes 0 and 1 and bandwidth is not bottleneck in this case in that copy happens on both NUMA nodes.

5 FUTURE WORK

For future work, we plan to explore the effects of stealing on other MPI internal processes such as data packing/unpacking and reduction. We will also evaluate the job stealing MPI implementation in real applications such as Nek5000 and NWChem.

ACKNOWLEDGMENTS

This material was based upon work supported by the U.S. Dept. of Energy, Office of Science, Advanced Scientific Computing Research (SC-21), under contract DE-AC02-06CH11357. The experimental resource for this paper was provided by the Laboratory Computing Resource Center on the Bebop cluster at Argonne National Laboratory.

REFERENCES

[1] Umut A Acar, Guy E Blelloch, and Robert D Blumofe. 2000. The data locality of work stealing. In *Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*. ACM, 1–12.

[2] Nimar S Arora, Robert D Blumofe, and C Greg Plaxton. 2001. Thread scheduling for multiprogrammed multiprocessors. *Theory of computing systems* 34, 2 (2001), 115–144.
[3] Robert D Blumofe and Charles E Leiserson. 1999. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)* 46, 5 (1999), 720–748.
[4] Darius Buntinas, Guillaume Mercier, and William Gropp. 2007. Implementation and evaluation of shared-memory communication and synchronization operations in MPICH2 using the Nemesis communication subsystem. *Parallel Comput.* 33, 9 (2007), 634–644.
[5] David Chase and Yossi Lev. 2005. Dynamic circular work-stealing deque. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 21–28.
[6] Danny Hendler and Nir Shavit. 2002. Non-blocking steal-half work queues. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*. ACM, 280–289.
[7] Nathan Hjelm, Pavel Shamis, and Jeff Squyres. 2018. XPMEM Linux Kernel Module. https://github.com/hjelmn/xpmem
[8] Atsushi Hori, Min Si, Balazs Gerofi, Masamichi Takagi, Jai Dayal, Pavan Balaji, and Yutaka Ishikawa. 2018. Process-in-process: techniques for practical address-space sharing. In *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 131–143.
[9] Michael Kerrisk. 2008. Overview of POSIX shared memory. http://man7.org/linux/man-pages/man7/shm_overview.7.html