DOI: 10.14529/jsfi200104

Online MPI Process Mapping for Coordinating Locality and Memory Congestion on NUMA Systems

Mulya Agung[1], *Muhammad Alfian Amrizal*[2], *Ryusuke Egawa*[3,1], *Hiroyuki Takizawa*[3,1]

Mapping MPI processes to processor cores, called process mapping, is crucial to achieving the scalable performance on multi-core processors. By analyzing the communication behavior among MPI processes, process mapping can improve the communication locality, and thus reduce the overall communication cost. However, on modern non-uniform memory access (NUMA) systems, the memory congestion problem could degrade performance more severely than the locality problem because heavy congestion on shared caches and memory controllers could cause long latencies. Most of the existing work focus only on improving the locality or rely on offline profiling to analyze the communication behavior.

We propose a process mapping method that dynamically performs the process mapping for adapting to communication behaviors while coordinating the locality and memory congestion. Our method works online during the execution of an MPI application. It does not require modifications to the application, previous knowledge of the communication behavior, or changes to the hardware and operating system. Experimental results show that our method can achieve performance and energy efficiency close to the best static mapping method with low overhead to the application execution. In experiments with the NAS parallel benchmarks on a NUMA system, the performance and total energy improvements are up to 34% (18.5% on average) and 28.9% (13.6% on average), respectively. In experiments with two GROMACS applications on a larger NUMA system, the average improvements in performance and total energy consumption are 21.6% and 12.6%, respectively.

Keywords: communication, congestion, locality, MPI, multi-core, NUMA, process mapping.

Introduction

In the fields of high-performance computing (HPC) and enterprise computing, a large-scale Symmetric Multi-Processing (SMP) system is typically built by using multiple processors to have more processor cores within a system. These processors have shared memories and on-chip memory controllers that form the base for NUMA (Non-Uniform Memory Access) multi-processors. Each processor consists of one or several groups of processor cores, each of which is physically associated with one or more memory controllers and memory devices. Such a group of processor cores is referred to as a NUMA node [13, 16, 25]. Although the NUMA nodes are generally connected by high-speed interconnect links such as QuickPath Interconnect (QPI) [32] and HyperTransport [25], accessing a remote NUMA node still needs a longer latency than accessing the data of the local memory device. Thus, the cost of remote memory access is higher than that of local memory access.

Currently, modern multi-core processors are widely used not only for shared-memory parallel processing but also for distributed-memory parallel processing, such as Message Passing Interface (MPI) [24]. In MPI, communication among MPI processes is explicit and is performed by sending and receiving messages. Each MPI process has a unique identifier called process ID or process rank. The process that sends the message is called a sender, while the process that receives the message is called a receiver. Thus, a communication event can be defined as one message with

[1]Graduate School of Information Sciences, Tohoku University, Japan
[2]Research Institute of Electrical Communication, Tohoku University, Japan
[3]Cyberscience Center, Tohoku University, Japan

its corresponding processes of a sender and a receiver. This pair is also referred to as a process pair [12, 21].

In NUMA systems, a communication event will access the local memory device if it is performed by a process pair whose sender and receiver are executed by different cores of the same NUMA node. On the other hand, it will access the memory device of the remote NUMA node if it is performed by a process pair whose sender and receiver are executed by different cores of different NUMA nodes. MPI provides extensions that enable faster intra-node communication through the use of shared memory, such as Nemesis for MPICH2 [8] and KNEM [17] for Open MPI [15]. However, as the cost of communication significantly affects the performance on NUMA systems, exploiting the communication behavior to optimize the mapping between MPI processes and processor cores is necessary to improve performance. Such process mapping methods are called *communication-aware process mapping* [12, 26].

In modern NUMA systems, process mapping becomes more challenging because a large number of processor cores in a system induce a large number of accesses to memory devices. As the number of processor cores increases, the number of communication that can simultaneously happen will also increase, causing congestion on shared caches and memory controllers. We refer to this congestion as *memory congestion*. Conventional work on MPI process mapping mostly focuses on improving the locality of communication by mapping processes frequently communicate with each other, to processor cores that are closer to each other in the memory hierarchy. Improving the locality of communications is important because it will reduce the cost of communication and congestion on interconnect links. However, only considering the locality is not sufficient to improve performance on modern NUMA systems. Furthermore, maximizing the locality can degrade performance because it potentially increases the memory congestion [2, 16].

To optimize the process mapping, it is necessary to analyze the communication behavior of the MPI applications. The communication behavior is determined by the communication among MPI processes of the application. A process does not necessarily need to communicate with all the other processes, and the time and amount of data exchanged among processes may vary. Conventional process mapping methods rely on offline profiling to trace the communication events and analyze the communication behavior. However, the offline profiling and analysis impose a high overhead and is not applicable if the application changes its communication behavior between executions. Furthermore, the data generated during profiling might be very large, requiring a time-consuming analysis [30].

In this paper, we present a process mapping method, called *Online Decongested Locality for MPI* (OnDeLoc-MPI), to tackle the locality and the memory congestion problems on modern NUMA systems. It consists of a mechanism that dynamically performs the process mapping for adapting to changes in the communication behavior, and a mapping algorithm to calculate the process mapping that can simultaneously reduce the amount of remote memory accesses and the memory congestion. OnDeLoc-MPI works at runtime during the execution of an MPI application. It does not require prior knowledge of the communication behavior, modifications to the application, or changes to the hardware and operating system.

The rest of the paper is organized as follows. In Section 1, we discuss related work and compare it to OnDeLoc-MPI. Then, we describe the procedure and implementation of OnDeLoc-MPI in Section 2. Experimental setup and results are presented in Section 3. This section also discusses the overhead of our method. Finally, conclusions and future work are summarized in Section 3.5.

1. Related Work

Various MPI process mapping methods have been proposed in the related studies. Most of the methods rely on offline profiling to trace communication between processes and to analyze the communication behaviors of the applications [2, 9, 21, 31]. The main drawback of these methods is the requirement of offline profiling, which has a high overhead and is potentially time-consuming. On the other hand, the proposed OnDeLoc-MPI does not have these disadvantages because it performs the process mapping dynamically at runtime during the execution of the application.

An online communication detection method, called CDSM, has been proposed in a related study [13]. It works on the operating system level during the execution of the parallel application. It detects the communication behavior from page faults, and uses this information to dynamically perform the process and thread mapping. CDSM employs a locality-based mapping algorithm to minimize the communication cost. The evaluation results of the related work have shown that CDSM can improve the performance of the MPI benchmarks. There are two key differences between CDSM and this work. First, OnDeLoc-MPI does not need to employ a communication detection mechanism because communication events can be traced directly from MPI events. Thus, it does not suffer from detection inaccuracy nor overhead caused by the communication detection mechanism. Second, OnDeLoc-MPI employs a mapping algorithm that aims to reduce both the communication cost and the memory congestion. In Section 3, we compare the performance results of OnDeLoc-MPI and CDSM, and discuss the benefits of our method.

A memory placement method, called Carrefour, has been proposed in [10]. It improves performance on modern NUMA systems by reconciling the data locality and the memory congestion problems. Carrefour works as a data mapping policy of the Linux kernel to dynamically place memory pages on NUMA nodes to avoid the congestion. Since the method works at system runtime, it suffers from the overheads caused by the memory access sampling and memory page replication. Lepers et al. [23] proposed a thread and memory placement method, called AsymSched, that considers the bandwidth asymmetry of asymmetric NUMA systems to minimize congestion on interconnect links and memory controllers on modern NUMA systems. It relies on continuous sampling of the memory accesses to analyze the communication among threads.

We cannot compare OnDeLoc-MPI with Carrefour and AsymSched because both methods require a sampling mechanism that is available only in AMD processors. However, in contrast to these methods, OnDeLoc-MPI analyzes the communication behavior directly from MPI communication events, and it does not require the communication detection and memory sampling mechanisms. Moreover, OnDeLoc-MPI works on the runtime system level, and it does not rely on a specific operating system or hardware. Compared with AsymSched, OnDeLoc-MPI focuses on reducing not only memory congestion but also the amount of remote accesses. Our evaluation results in Sections 3.2 and 3.3 show that the reduction of the amount of remote accesses can substantially improve performance and energy efficiency.

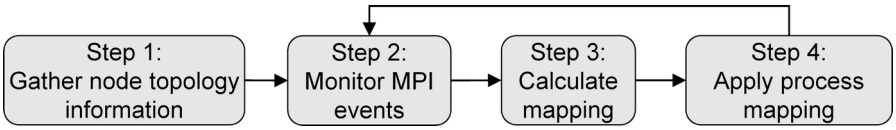
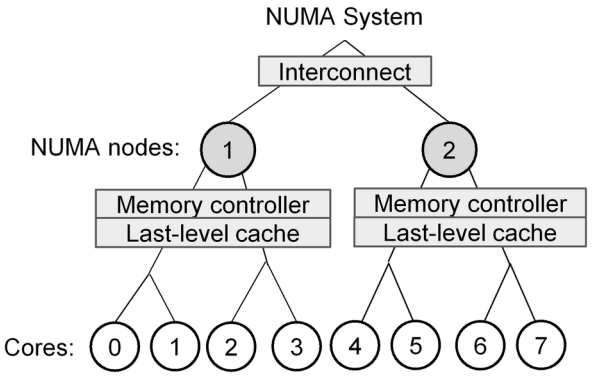


Figure 1. The procedure of OnDeLoc-MPI



(a) A node topology with eight cores

7	0	0	0	0	0	1	0	0
6	0	0	0	0	1	0	0	0
5	0	0	0	0	0	0	0	1
4	0	0	0	1	0	0	1	0
3	10	500	500	0	1	0	0	0
2	10	100	0	500	0	0	0	0
1	100	0	100	500	0	0	0	0
0	0	100	10	10	0	0	0	0
Process ID	0	1	2	3	4	5	6	7

(b) A communication matrix with eight processes

Figure 2. The examples of NUMA node topology and communication matrix

2. OnDeLoc-MPI: An Online Process Mapping Method for Coordinating Locality and Memory Congestion

In this section, we describe how OnDeLoc-MPI works during the execution of an MPI application. We first explain the procedure of the method, and then describe the implementation of the method in the MPI runtime system.

2.1. Procedure of OnDeLoc-MPI

Figure 1 shows the procedure of OnDeLoc-MPI, which consists of four steps:

1. Gather the node topology information of the NUMA system.
2. Monitor MPI communication events during the execution.
3. Calculate the MPI process mapping.
4. Apply the process mapping.

First, when the target application is launched, OnDeLoc-MPI obtains the information about the NUMA node topology of the target system. The topology is modeled as a tree to express the information about the locations of cache memories, memory controllers, and interconnect links. In NUMA systems considered in this work, each NUMA node is physically associated with a shared last-level cache (LLC) and an integrated memory controller, such as Intel-based and AMD-based NUMA systems [25]. Thus, the location of the NUMA node represents both the location of memory controllers and LLCs. This information is required because OnDeLoc-MPI focuses on reducing the amount of remote accesses through interconnects and reducing the congestion on the shared caches and memory controllers. Figure 2(a) shows an example of the model of a two-node NUMA system that consists of eight processor cores. The model also contains information about the physical identities of the NUMA nodes and the processor cores. This information is used later by the mapping algorithm to calculate the mapping.

Second, during the execution of the application, the communication behavior of the application is analyzed by monitoring the communication events among MPI processes. A commu-

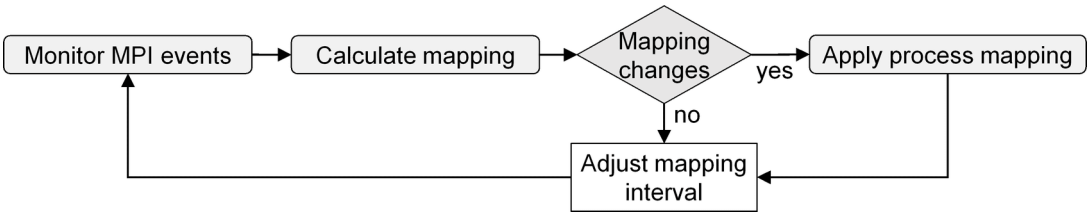


Figure 3. The mechanism of mapping interval adjustment

nication matrix is used to model the communication behavior, and it consists of identifiers of MPI processes and amount of communication among the processes. The communication matrix is a square matrix of order N_p, where N_p is the number of MPI processes executed by the application. It has the same number of rows and columns because each process can communicate with all the other processes. Figure 2(b) shows an example of the communication matrix for an application that consists of eight processes. Each cell (x, y) of the matrix contains the amount of communication between a pair of processes x and y, which is obtained by aggregating the volume and number of communication events between the pair. In the communication behavior shown by the matrix, processes 0 to 3 have a larger amount of communication than the other processes. Since the communication behavior of the application may change during the execution, we update the communication matrix periodically with a certain time interval, called *mapping interval*. At the beginning of the execution, the values of all cells are set equal to zero.

In the third and fourth steps, OnDeLoc-MPI uses the information about communication behavior to optimize the process mapping. In Step 3, the mapping is obtained by using an algorithm, called OnDeLocMap+ algorithm, which is executed every time the communication matrix is updated. The algorithm calculates the mapping that can improve the communication locality and the memory congestion, which is detailed in Section 2.2. Then, in Step 4, the calculated mapping is applied to the execution by assigning processor cores to processes according to the mapping. Since the mapping result of Step 3 can be different for each calculation, OnDeloc-MPI will migrate a process if the mapping of the process changes from the previous mapping.

As shown in Fig. 1, Steps 2 to 4 are performed iteratively during the execution of the application. We note that the mapping interval can significantly affect the performance results and overhead of our method. If the interval is too long, OnDeLoc-MPI may not adapt quickly to the changes in the communication behavior. On the other hand, a shorter interval will increase the overhead because it increases the frequency of updating the communication matrix and calculating the process mapping. We discuss the overhead of OnDeLoc-MPI in more detail in Section 3.4.

To reduce the overhead, OnDeLoc-MPI dynamically adjusts the mapping interval, which is shown in Fig. 3. We use a slope parameter, v, to automatically increase and decrease the mapping interval, where $v > 1$. If the calculated mapping does not differ from the previous mapping, the mapping interval is multiplied by v as in Equation 1. The mapping interval is increased because the current mapping is assumed to be stable. On the other hand, if the mapping differs from the previous mapping, the mapping interval is divided by v as in Equation 2. We shorten the mapping interval so that the mapping can adapt more quickly to changes in the communication behavior.

$$Interval_{curr} = Interval_{prev} \times v. \tag{1}$$

$$Interval_{curr} = \frac{Interval_{prev}}{v}. \tag{2}$$

2.2. Implementation of OnDeLoc-MPI

OnDeLoc-MPI has been implemented as a module for Open MPI runtime system [15]. The implementation consists of four parts:

1. A modification to the monitoring layer of the runtime system.
2. Data consolidation among MPI processes.
3. OnDeLocMap+ algorithm.
4. Data structures to store the communication matrices and process mapping.

2.2.1. Modification of the runtime system

For the implementation, we added a module in the monitoring layer of the Open MPI runtime system, and the module is started when an MPI application is launched by the runtime system. To perform Steps 1 and 2 of the OnDeLoc-MPI procedure, we have implemented two functions in the module. The first function obtains the node topology information of the system by using Hwloc library [7]. We use this library because it can provide both logical and physical indexes of the processor cores and the NUMA nodes. The second function monitors MPI communication events during the execution. During the monitoring, the amount of communication of each process pair is accumulated using a counter. This function uses a monitoring framework [6] that is built on top of the point-to-point management layer (PML) of the Open MPI stack [15]. We use PML because it can monitor point-to-point operations organizing a collective communication, and thus the communication events can be traced in both cases of point-to-point and collective communications.

For Steps 3 and 4, we create a thread, called *mapper thread*, that periodically calculates and applies the process mapping. This thread is a child thread of the process that has the local ID 0. In MPI, the local ID is the local rank of an MPI process within a system [15, 18]. It means that if an MPI application is executed with more than one NUMA system, OnDeLoc-MPI will create one mapper thread for each system, and each thread calculates and applies the process mapping separately for each system. To apply the process mapping, we assign the target processor cores to processes according to the mapping by using the `sched_setaffinity()` function call of the Linux system. However, the use of this function is not mandatory. Alternatively, some libraries such as Hwloc-bind [7] and Likwid-pin [29] can be used to assign processor cores to processes.

We are aware that, in an MPI application, two MPI processes running on different NUMA systems may communicate with each other. Thus, calculating the process mapping separately for each system may not result in the best mapping for the application. However, by performing the process mapping separately for each system, OnDeLoc-MPI does not suffer from the overhead of migrating processes from one system to another system. This overhead may surpass the benefit of our method because migrating MPI processes across systems potentially incurs a significant network overhead [5]. In the future, we will discuss the impacts of migrating MPI processes across NUMA systems to the performance results of our method.

2.2.2. Data consolidation

In an MPI application, each MPI process is executed on a processor core as a single operating system process. However, the PML monitors MPI communication events separately for each process, and in Linux and other UNIX operating systems, a process cannot directly access

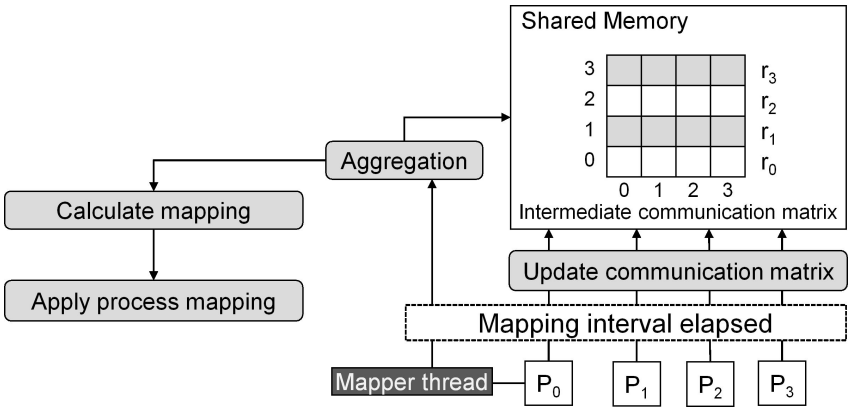


Figure 4. The consolidation mechanism with four MPI processes

the address space of another process. Thus, to determine the communication behavior of the application, it is necessary to consolidate the communication events among the processes.

The consolidation mechanism is shown in Fig. 4. For the consolidation purpose, we store an intermediate communication matrix in a shared memory region. Each row of this matrix has its own shared memory object, and thus the accesses to the matrix do not need to be locked since each row is updated only by one process. We use the POSIX shared memory API [22] to read and write the shared memory objects. However, the consolidation process may increase congestion on the shared region if a large number of processes frequently update the matrix. Thus, we also use the mapping interval to limit the frequency of updating the matrix.

During Step 2, each process updates its row in the intermediate matrix using the monitoring counters. The i-th row (r_i) is updated by the process with ID i (P_i). However, the value of each counter is accumulated during the monitoring step. If the communication matrix is updated with the accumulated values, OnDeLoc-MPI may not be able to detect changes in the communication behavior because the previous communication behavior may significantly influence the current result of communication behavior. Thus, to reset the communication behavior, we update cell (x, y) by subtracting the last value of the cell from the counter value for processes x and y. The mapper thread generates a consolidated matrix by aggregating the data from all rows of the intermediate matrix. The generated communication matrix is then used as the input to the mapping algorithm.

2.2.3. OnDeLocMap+ algorithm

The OnDeLocMap+ algorithm is depicted in Algorithm 1. We adopt the algorithm proposed in our previous work, called On-DeLoc [3], to implement OnDeLocMap+. A key difference between these two algorithms is that OnDeLocMap+ considers the previous mapping to calculate the current mapping. In the previous work, we have shown that the migration overhead has a significant impact on the performance results of On-DeLoc. To reduce this overhead, the OnDeLocMap+ algorithm prevents unnecessary process migrations by giving a higher priority to processes that have a higher amount of communication to be mapped to the same NUMA node of the previous mapping. We detail the difference between the two algorithms in the following description.

First, OnDeLocMap+ uses the topology model to construct the map between processor core IDs and process IDs (Line 1). The keys of the map represent the IDs of processor cores available

Algorithm 1 The OnDeLocMap+ Algorithm.

Input: T {The node topology tree}
Input: A {The communication matrix}
Input: $PrevM$ {The previous mapping}
Output: M {The map of processor core IDs and process IDs}

```
1:  M ← createMap(T)
2:  Pairs ← generatePairs(A)
3:  sortedPairs ← sortByAcomm(Pairs)
4:  i ← 0
5:  while i < num(Pairs) and numUnmappedCores(M) > 0 do
6:    current_pair ← sortedPairs[i]
7:    prev_nodes ← getPreviousNodes(current_pair, PrevM)
8:    if isNodesAvailable(prev_nodes) then
9:      mapPair(current_pair, prev_nodes)
10:   else
11:     mapPair(current_pair, nextNodeAvailable(T))
12:   end if
13:   i ← i + 1
14: end while
```

in the system, and each value represents the ID of the process mapped to the processor core of the key. At the beginning of the algorithm, all values are set to empty. Then, it generates pairs of processes from the communication matrix (Line 2). A pair of processes x and y is generated for each matrix cell (x, y) of the matrix. The algorithm then selects a process pair that has not been mapped to processor cores sequentially from the pairs with the highest to the lowest amount of communication. This selection is achieved by the sorting step in the algorithm (Line 3). A NUMA node is available for the mapping if it has one or more unmapped cores. In On-DeLoc, the *mapPair*() function always maps the processes to the processor cores of the NUMA node that is currently available in a round-robin fashion (Line 11). We aim to improve the locality by mapping two processes of a pair to the same NUMA node, while also reducing the memory congestion by mapping different pairs to the different NUMA nodes. However, in contrast to On-DeLoc, when selecting the target NUMA nodes for a process pair, OnDeLocMap+ first evaluates the NUMA nodes that have been previously mapped for the same pair (Lines 7–8). The function *getPreviousNodes*() will return two NUMA nodes, where each node is associated with each process of the pair. If the previous NUMA nodes are available, it will map each process of the pair to the processor cores of the previous NUMA node associated with the process (Line 9) so that each process of the pair will not be migrated to a different NUMA node. Otherwise, OnDeLocMap+ will map the pair to the processor cores in a round-robin fashion, the same way as the previous algorithm.

2.2.4. Data structures

For each MPI application, we allocate two arrays to store the two communication matrices. The first array is to store the consolidated communication matrix, and the other array is to store the intermediate matrix. Since the matrix is a square matrix of order N_p, the size of each

matrix scales quadratically with the number of MPI processes. The size of each matrix cell is 4 bytes, and thus the total size of the memory used for all the communication matrices is $(N_p^2 \times 2 \times 4)$ bytes. In addition to the communication matrices, we allocate a key-value map to store the previous mapping, where each element of the map consists of a processor core ID and its associated process ID. The size of the map scales linearly with N_p. The size of each element is 8 bytes, and thus the total size of the memory allocated for the key-value map is $(N_p \times 8)$ bytes.

3. Experimental Evaluation

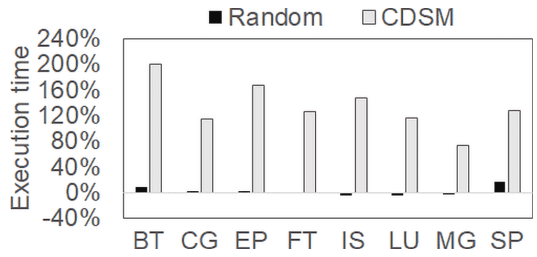
In this section, we present the experimental evaluation of OnDeLoc-MPI. Our main experimental results on a NUMA system consist of three parts: performance, energy consumption, and overhead. In addition, we provide and discuss our evaluation results with a larger NUMA system.

3.1. Experimental Setup

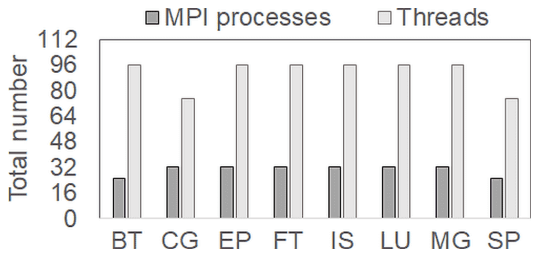
The main experiments have been conducted on a NUMA system, named Xeon2, that consists of two NUMA nodes and one Intel Xeon E5-2690 processor per NUMA node. The system has 32 logical cores in total, and it is running Linux OS kernel v3.2. The NUMA nodes are connected with QuickPath Interconnect (QPI) [32], and each NUMA node has 16 logical cores and an Integrated Memory Controller (IMC). Open MPI v3.1 is used as the MPI runtime system for the experiments. As workloads, we used eight MPI applications of the NAS Parallel Benchmarks (NPB) [4] v3.4 with the class C problem size. All the applications except BT and SP are executed with 32 processes using all cores in the system. BT and SP require a square number of processes, and thus we execute these two applications with 25 processes.

In all the experiments, we keep the mapping interval higher than or equal to 500 ms to limit the overhead. The parameter v is set equal to 2, and this value is chosen empirically from experiments with the NPB applications. We are aware that in parallel applications that change their communication behavior during their execution, this parameter will affect the performance results of our method. However, to determine the optimal value of v for a particular application, it is necessary to analyze its temporal communication behavior prior to the execution of the application. We avoid this analysis step because it will incur a high overhead [30]. In our future work, we will investigate the impacts of this parameter on the performance and overhead of our method.

We compare OnDeLoc-MPI with an online-based thread mapping method and two static mapping methods. The static mapping methods are Default and Static-best. Default is the original mapping of the MPI runtime system that maps the neighboring processes to processor cores of different NUMA nodes in a round-robin fashion. It represents the baseline of our experiments. Static-best mapping is obtained by using an offline-based method proposed in our previous work [2]. It first collects communication traces by preliminary running the target application. Then, it analyzes the communication behavior of the application and calculates the mapping using the CLB algorithm. We also evaluated the Treematch [21] algorithm to calculate the mapping. Since CLB shows lower execution times than Treematch, we show only the results for CLB. Note that Static-best mapping works offline prior to the execution of an application, and thus has a high overhead caused by the preliminary run and offline analysis.



(a) Performance results of CDSM, normalized to the baseline



(b) The total number of threads compared with the number of MPI processes

Figure 5. Evaluation results with CDSM

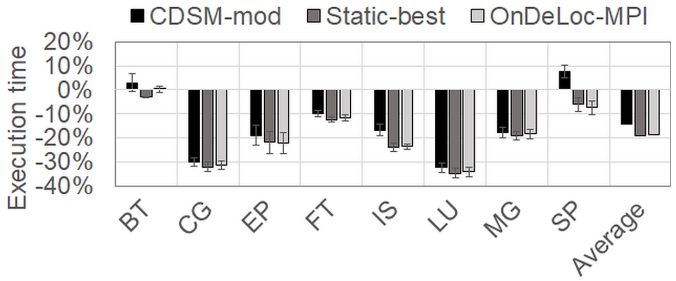


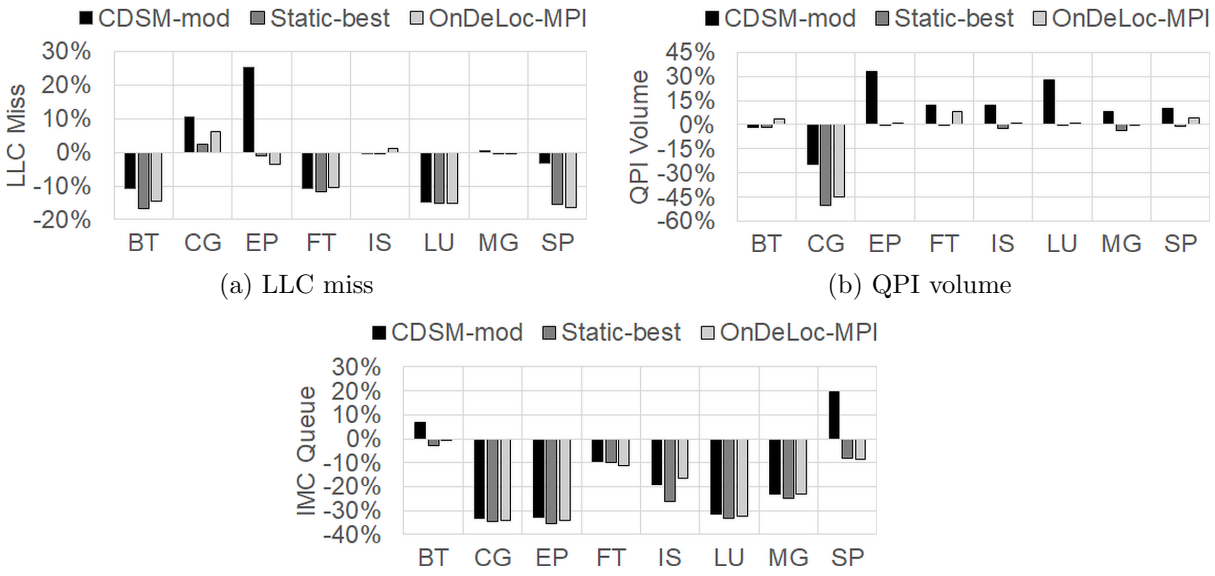
Figure 6. Performance results on Xeon2, normalized to the baseline

The online-based mapping method used for the evaluation is CDSM-mod, which is a modified version of CDSM. We modify CDSM because our experimental results show that it significantly degrades the performance of all the tested applications, which are contrast with the results shown in the related work [13]. Figure 5(a) shows the performance results of CDSM, compared with the baseline and a random mapping method. For the random mapping, we randomly generate a process mapping before each execution. As shown in the figure, CDSM shows higher execution times for all the applications. In BT, CDSM can increase the execution time by a factor of two compared with Default. We have observed that the performance degradation is caused by the inaccuracy of detecting the communication events among MPI processes. CDSM detects the communication events by analyzing the page faults of all threads of the application. However, in multithreaded MPI implementations, an MPI process can spawn multiple threads [14]. Thus, the total number of threads spawned by the runtime system can be higher than the number of MPI processes.

Figure 5(b) shows the total number of threads executed for the NPB applications with 32 number of MPI processes on Xeon2. Although the number of MPI processes is less than or equal to the number of processor cores of the system, the total number of threads executed during the execution is substantially higher than the number of cores and MPI processes. By detecting the communication among all threads of the application, CDSM cannot accurately detect the communication among MPI processes. To increase the accuracy, we modify the method to only include the parent threads of the MPI processes in the steps of detecting communication and calculating the thread mapping. We detect these parent threads from the first n threads created at the beginning of the execution, where n is equal to N_p.

3.2. Performance Evaluation

Figure 6 shows the performance results on the Xeon2 system. We measure the execution time of the applications with each mapping method. All the experimental results are the averages



(a) LLC miss

(b) QPI volume

(c) IMC queue

Figure 7. Performance monitoring results, normalized with the baseline

obtained from 10 sample executions, which are normalized to the results of the baseline method. We also provide the 95% confidence interval calculated with Student's t-distribution. The error line of the bar represents the confidence intervals of the samples.

On average, OnDeLoc-MPI shows higher performance improvements than those of Default and CDSM-mod. Compared with Default, the average performance improvement of OnDeLoc-MPI is 18.5%. The highest performance improvements are exhibited in CG and LU by 31.2% and 34%, respectively. CDSM-mod shows performance improvements from Default for most of the applications, indicating that our modification to CDSM effectively increases the communication detection accuracy, and thus it can increase the performance of the applications.

For all the applications except EP and SP, Static-best shows the highest improvements. However, the average improvement of OnDeLoc-MPI is only 0.5% lower than that of Static-best, which means that our method can achieve performance close to that of Static-best even without any kind of extensive profiling and analysis. Moreover, in SP, OnDeLoc-MPI achieves the highest performance improvement among the methods. These results indicate that in the case of SP, the static mapping is not sufficient to take into account the temporal changes of the communication behavior.

To investigate the sources of performance improvements, we evaluate the performance characteristics of the NPB applications. We use LLC misses, QPI volume and IMC queue metrics for the evaluation. These metrics are obtained by monitoring the Intel performance counters [20]. LLC misses represent the number of last-level cache misses across all NUMA nodes. IMC queue is the total queuing time of memory accesses in the memory controllers. A higher value of this metric indicates a longer queuing delay caused by the memory congestion. QPI volume is the total volume of data sent through interconnect links. A higher value of this metric indicates longer latencies from the remote memory accesses.

Figures 7(a), 7(b) and 7(c) show the results of last-level cache misses, QPI volume and IMC queue, respectively. These figures show that most of the applications gain a substantial performance improvement from reductions in the caches misses and IMC queuing delay. It means that the memory congestion has a significant impact on the performance of most of

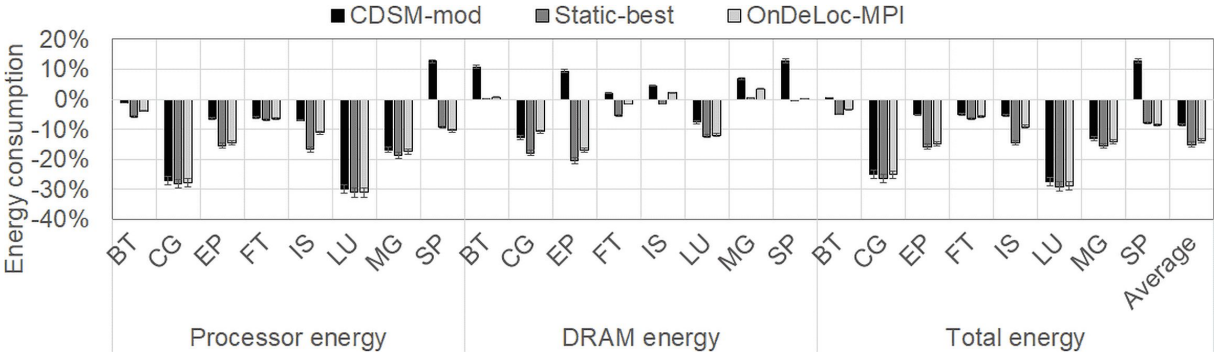


Figure 8. Energy consumption results on Xeon2, normalized with the baseline

the NPB applications. Moreover, in BT and SP, CDSM-mod increases the IMC queuing delay, and in most of the applications, it shows a higher IMC queue than those of OnDeLoc-MPI and Static-best. This fact suggests that only considering the locality is not sufficient to achieve the best performance for these applications. In the cases of BT and SP, the performance differences among the methods are smaller than that in the other applications. It is because, as shown in our previous work [3], these two applications have the communication behavior that can benefit from the Default mapping. In these two applications, most communication events are performed by the neighboring processes, and thus the Default mapping is sufficient to improve the performance of these applications.

In most of the applications, CDSM-mod shows a higher QPI volume (Fig. 7(b)) and a longer execution time (Fig. 6) compared with Static-best and OnDeLoc-MPI. By migrating the processes during the execution, CDSM-mod and OnDeloc-MPI potentially increase data traffic on interconnects because the migrated processes may need to access data that reside in a remote NUMA node. However, the performance improvements achieved by OnDeLoc-MPI are close to those of Static-best. Furthermore, even for the applications that cannot gain a significant performance improvement from Static-best mapping, such as BT, OnDeLoc-MPI does not reduce the performance of the applications. These results show that the migration overhead in online-based mapping methods can have a significant impact on the execution time. However, our method can effectively reduce this overhead.

3.3. Energy Consumption Evaluation

In this section, we discuss the energy consumption of the NPB applications on the Xeon2 system. We measure the processor energy and DRAM energy by using the Running Average Power Limit (RAPL) hardware counters [11]. As shown in the performance monitoring results, the process mapping methods have a significant impact on the cache misses, the interconnect traffic, and queuing delay in memory controllers. Therefore, the mapping methods will affect the energy consumption of not only the processor cores but also the interconnects and memory controllers. In the Intel processor used for the evaluation, each package of processor consists of core and Uncore components. Uncore refers to components that are apart from processor core, which include QPI and memory controllers [19]. To evaluate the energy consumption of processor core and Uncore components, we decompose processor energy consumption into core and Uncore energy consumptions. We measure core and Uncore energy also using the RAPL counters.

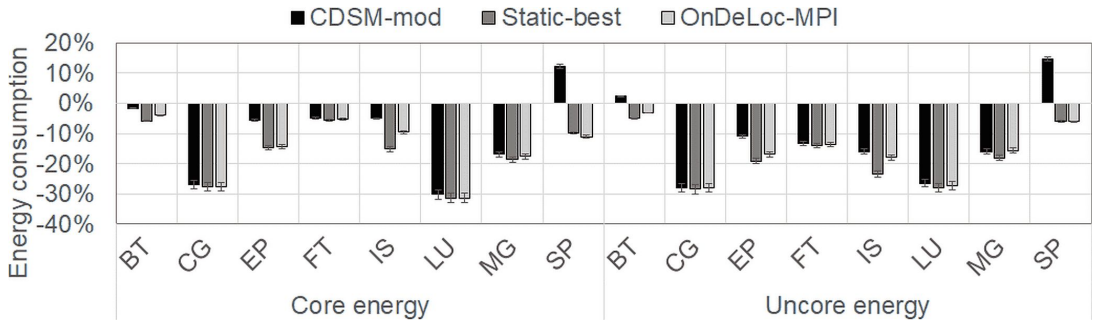


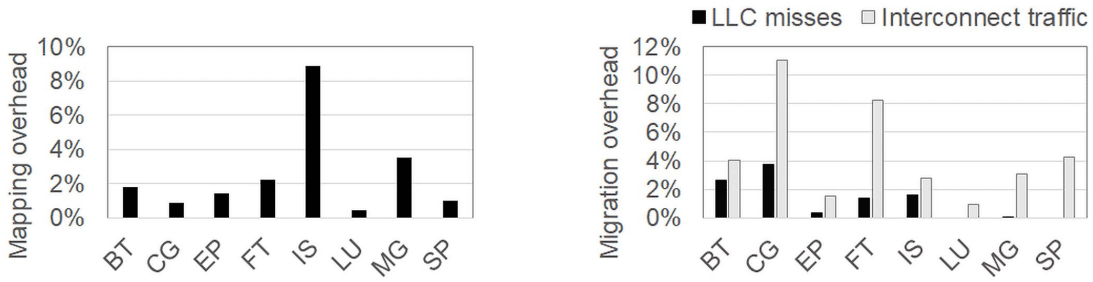
Figure 9. Core and Uncore energy consumptions on Xeon2, normalized with the baseline

Figure 8 shows the results of processor energy and DRAM energy for each mapping method on Xeon2. In all the applications, OnDeLoc-MPI shows a lower total energy consumption than those of Default and CDSM-mod. On average, the total energy is reduced by 13.6% compared with Default, and the highest reduction is 28.9% in the case of LU. In some applications, such as IS and MG, OnDeLoc-MPI and CDSM-mod increase DRAM energy. It is because these two online mapping methods use more DRAM to analyze the communication behavior and calculate the mapping. However, the increase in DRAM energy is relatively small compared with the decrease in processor energy. Since the total energy is mostly contributed by the processor energy, OnDeLoc-MPI achieves lower total energy consumption than the baseline in all the applications.

Figure 9 shows the results of core and Uncore energy consumptions. In all the NPB applications, OnDeLoc-MPI show lower core and Uncore energy consumptions than those of CDSM-mod and Default. The core energy consumption is reduced most in LU, with a reduction of 31.5%, and the Uncore energy consumption is reduced most in CG, with a reduction of 27.9%. On average, the core energy is reduced by 15% in OnDeLoc-MPI and 16.1% in Static-best, while Uncore energy is reduced by 16% and 17.7%, respectively.

In most of the applications, CDSM-mod shows higher core and Uncore energy consumptions than Static-best and OnDeLoc-MPI. The increases in core and Uncore energy is caused by the increases in execution time and interconnect traffic, respectively. Moreover, in BT and SP, CDSM-mod shows the highest Uncore energy because, as shown in Fig. 7(b) and 7(c), it also increases the queuing time in the memory controllers. These results show that compared to Default and CDSM-mod, OnDeLoc-MPI is more effective in reducing the energy consumption of interconnects and memory controllers.

In most of the applications, the lower execution time leads to the lower core and Uncore energy consumptions, indicating that the energy reductions are mainly contributed by the reduction in execution time. By reducing the execution time, OnDeLoc-MPI, CDSM-mod, and Static-best reduce the static energy consumption in most of the applications. However, as shown in BT and SP, OnDeLoc-MPI can achieve lower reductions in core and Uncore energy consumptions than those of Default and CDSM-mod with no significant differences in the execution time. This fact shows that OnDeLoc-MPI also reduces the dynamic energy consumption by reducing the number of cache misses and queuing time in the memory controllers.



(a) Mapping overhead

(b) Migration overhead

Figure 10. Overhead of OnDeLoc-MPI

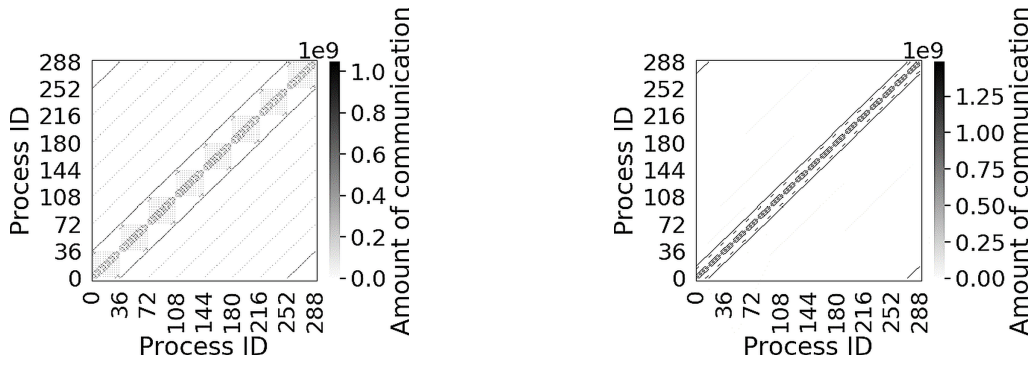
3.4. Overhead of OnDeLoc-MPI

OnDeLoc-MPI incurs overhead on the execution of an MPI application because it works during the application runtime. The overhead is caused by the computation of the mapping and the migration of processes. The mapping overhead consists of the repeated accesses to the intermediate and consolidated communication matrices and the execution of the mapping algorithm, while the migration overhead consists of increases in cache misses and interconnect traffic for the process after migration. In this section, we evaluate the overhead of OnDeLoc-MPI on the Xeon2 system.

The mapping overhead is shown in Fig. 10(a). It is evaluated by measuring the time in the function that accesses the communication matrices and to calculate the mapping. The values are the percentage of the execution time of each application. For all the applications, the mapping overhead is less than 9%, and the average overhead of the mapping is low, which is 2.5%. IS shows the highest mapping overhead because its execution time is much shorter than those of the other applications, and thus, the ratio of the mapping overhead to the execution time is higher than those of the other applications. However, the time used in IS for updating the communication matrices and calculating the mapping is less significant compared with the other applications. The results of mapping overhead show that the dynamic adjustment of the mapping interval can effectively reduce the overhead.

The migration overhead is evaluated by comparing the performance monitoring results of Static-best mapping and OnDeLoc-MPI for each application. However, in this evaluation, we disable the functions of OnDeLoc-MPI that repeatedly update the communication matrix and compute the mapping. The process mapping for each interval is provided offline prior to the execution. We obtain the mapping of each interval by preliminary running each application with OnDeLoc-MPI. Thus, in this evaluation, only the migration overhead affects the execution of each application.

Figure 10(b) shows the migration overhead on the cache misses and interconnect traffic. We obtain cache misses by aggregating the cache misses of all cache levels across the NUMA nodes. For all the applications, the migration overhead is less than 11%, and the highest overheads on the interconnect traffic are imposed in CG and FT. As shown in our previous work [3], CG has a wide variation of the amount of communication among processes. Moreover, these two applications have a high number of memory accesses. Thus, migrating a process potentially increases the amount of remote accesses because the process may need to access data that reside in the previous NUMA node. BT and CG show the highest overhead to last-level cache misses,



(a) Test Case A (b) Test Case B

Figure 11. The communication behaviors of the GROMACS applications

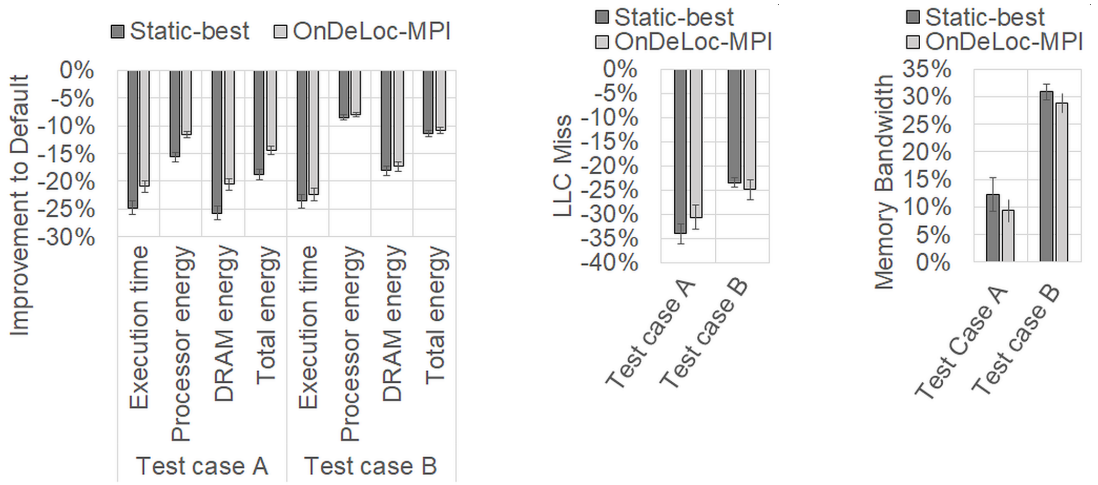
indicating that these two applications access cache memories more than the other applications. For the other applications, the migration overheads are small because, in these applications, the process mapping is more stable than those of CG and FT. OnDeLoc-MPI performs less migration during the execution of these applications.

The performance and overhead results show that the migration overhead mainly causes the performance differences between OnDeLoc-MPI and Static-best mapping. This overhead is affected by the numbers of memory accesses and changes in the communication behavior of the application. During the execution, some applications, such as BT, CG and FT access memory devices and change their communication behavior more frequently than the other applications. The migration overheads of OnDeLoc-MPI in these applications are the highest among the applications, and thus OnDeLoc-MPI shows a lower performance improvement compared with Static-best mapping. On the other hand, it shows lower migration overheads in the other applications, and thus OnDeLoc-MPI can achieve a comparable performance with Static-best in the other applications. Moreover, in SP, the benefit of online mapping surpasses the overhead of OnDeLoc-MPI, and thus it can achieve a higher performance than that of Static-best.

3.5. Evaluation on a Larger System

To evaluate our method with larger number of NUMA nodes and processor cores, we have conducted experiments on a NUMA system, named KNL4, which is based on Intel Xeon Phi Knights Landing (KNL) processor [28]. The system has 288 logical cores in total, and it is running Linux kernel v4.4. We configure the system as a four-node NUMA system by setting Sub-NUMA clustering (SNC) mode as the clustering mode of the KNL. In this cluster mode, the system is partitioned into four NUMA nodes, with 72 logical cores per NUMA node. We also use Open MPI v3.1 as the MPI runtime system for this evaluation.

In this evaluation, we use two biomolecular applications of the UAEBS workloads [27, 33]. The applications are Test Case A and Test Case B of the workloads that are based on GROMACS simulation [1]. The communication behaviors of these applications are shown by the communication matrices in Fig. 11. The darker cells indicate a higher amount of communication. We obtain these matrices from the last mapping result of OnDeLoc-MPI during the execution of these applications. These two applications are executed with 288 number of MPI processes to use all the processor cores of the system. We do not use the NPB applications because all the applications except EP cannot be executed with this number of processes. OnDeLoc-MPI is compared with



(a) Performance and energy (b) LLC misses (c) Memory bandwidth

Figure 12. Performance and energy consumption results on KNL4, normalized with the baseline

Default and Static-best mapping. We cannot compare our method with CDSM-mod because of its limitation to the previous version of Linux kernel. This fact highlights the advantage of implementing our method in the runtime system. which is that OnDeLoc-MPI does not depend on specific features of the hardware or operating system.

Figure 12(a) shows the results of performance and energy consumption on the KNL4 system. We cannot provide the Uncore energy in this evaluation because of the limitation of the RAPL counters on the KNL. However, we can still evaluate the impacts of the mapping methods on the total energy consumption by measuring processor and DRAM energy. As shown in the figure, OnDeLoc-MPI reduces the execution times and energy consumptions in both applications. Compared to Default, the execution time is reduced most in Test Case B, with a reduction of 22.3%, while the highest total energy reduction is shown in Test Case A, with a reduction of 14.3%. On average, the execution time and total energy consumption are reduced by 21.6% and 12.6%, respectively.

As shown in the figure, OnDeLoc-MPI shows lower improvements compared with Static-best. However, the performance and energy consumption results of these two methods are close to each other. The differences of execution time are 3.91% in Test Case A, and 1.24% in Test Case B, while the processor energy differences are 3.98% and 0.47%, respectively. The differences in Test Case A are higher than those in Test Case B because the communication behavior of Test Case A is more irregular than that of Test Case B. As shown in their communication matrices, the number of tasks that perform substantial amount of communication in Test Case A is higher than that in Test Case B. OnDeLoc-MPI updates the process mapping more frequently in Test Case A. The performance and energy consumption results of these two applications suggest that the overhead of OnDeLoc-MPI is still low even if the number of processes and NUMA nodes becomes larger.

For further evaluation, we measure the last-level cache misses and memory bandwidth metrics on the KNL4 system. Memory bandwidth is the bandwidth of memory accesses to the memory controllers. We cannot measure the QPI volume and IMC queue metrics because of the limitation of the monitoring counters on the KNL. However, the memory bandwidth metric can show the impacts of the mapping methods on the memory congestion because higher memory congestion leads to lower memory access bandwidth.

Figures 12(b) and 12(c) show the results of cache misses and memory bandwidth, respectively. These results suggest that both OnDeLoc-MPI and Static-best gain performance and energy improvements by reducing last-level cache misses and memory congestion. Compared with Default, OnDeLoc-MPI shows lower cache misses and higher memory bandwidth for both applications. The highest reduction of cache misses is 30% with Test Case A, while the highest improvement of memory bandwidth is 28.8% with Test Case B. Compared with Static-best, OnDeLoc-MPI shows higher cache misses in Test Case A, and shows lower memory bandwidth for both applications. However, the differences of the results between the two methods are small. The differences of cache misses are 3.37% in Test Case A, while the memory bandwidth differences are 2.9% and 2.1% in Test Case A and Test Case B, respectively.

Conclusions and Future Work

In this paper, we have proposed a process mapping method, called OnDeLoc-MPI, to address the locality and the memory congestion problems on NUMA systems. Our method works online during the execution of an MPI application, and dynamically performs the process mapping to adapt to changes in the communication behavior of the application. In contrast to the related work, OnDeLoc-MPI does not need offline profiling to analyze the communication behavior of the application, and does not rely on communication detection mechanisms and specific features of the hardware or operating system. Alternatively, it analyzes the communication behavior by monitoring the MPI communication events during the execution of the application.

OnDeLoc-MPI has been evaluated on a real NUMA system with a set of NAS parallel benchmarks. On average, it can achieve performance and energy improvements close to the best static method with low overhead. Compared with the default mapping of the MPI runtime system, the performance and total energy improvements are up to 34% (18.5% on average), and 28.9% (13.6% on average), respectively. In addition, OnDeLoc-MPI has been evaluated on a larger NUMA system with two GROMACS applications. On the larger system, it achieves performance and energy improvements up to 22.3% and 14.3%, respectively. Our evaluation results have shown that the performance and energy improvements are obtained from reductions in cache misses, interconnect traffic, and queuing delay in memory controllers.

During the execution of an MPI application, OnDeLoc-MPI imposes overhead from the mapping calculation and process migration. To reduce the overhead, OnDeLoc-MPI employs mechanisms to prevent unnecessary process migrations and automatically adjust the mapping interval. The evaluation results show that the mechanisms can effectively reduce the overhead. The mapping overhead to the execution time is less than 9%, and the migration overhead to interconnect traffic and cache misses is less than 11%.

As future work, we intend to evaluate the impacts of our method on the performance and energy consumption of a large cluster of NUMA systems. For this future work, we will investigate the overhead of migrating processes between systems, and the impacts of parameter v for different applications. We also intend to extend to our method for parallel applications with hybrid programming of MPI and multithreading, such as OpenMP and Pthreads.

Acknowledgements

This work is partially supported by Japan's Ministry of Education, Culture, Sports, Science and Technology (MEXT) Next Generation High-Performance Computing Infrastructures and

Applications R&D Program "R&D of A Quantum-Annealing-Assisted Next Generation HPC Infrastructure and its Applications" and Grant-in-Aid for Scientific Research(B) #16H02822 and #17H01706.

References

1. Abraham, M.J., Murtola, T., Schulz, R., et al.: GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers. SoftwareX 1-2, 19–25 (2015), DOI: 10.1016/j.softx.2015.06.001

2. Agung, M., Amrizal, M.A., Komatsu, K., et al.: A memory congestion-aware MPI process placement for modern NUMA systems. In: 2017 IEEE 24th International Conference on High Performance Computing, HiPC, 18-21 Dec. 2017, Jaipur, India. pp. 152–161. IEEE (2017), DOI: 10.1109/HiPC.2017.00026

3. Agung, M., Amrizal, M.A., Egawa, R., et al.: An automatic MPI process mapping method considering locality and memory congestion on NUMA systems. In: 2019 IEEE 13th International Symposium on Embedded Multicore/Many-core Systems-on-Chip, MCSoC, 1-4 Oct. 2019, Singapore. pp. 17–24. IEEE (2019), DOI: 10.1109/MCSoC.2019.00010

4. Bailey, D., Barszcz, E., Barton, J., et al.: The NAS Parallel Benchmarks. Int. J. High Perform. Comput. Appl. 5(3), 63–73 (1991), DOI: 10.1177/109434209100500306

5. Barak, A., Margolin, A., Shiloh, A.: Automatic resource-centric process migration for MPI. In: Träff, J.L., Benkner, S., Dongarra, J.J. (eds.) Recent Advances in the Message Passing Interface, 23-26 Sep. 2012, Vienna, Austria. pp. 163–172. Springer, Berlin, Heidelberg (2012), DOI: 10.1007/978-3-642-33518-1_21

6. Bosilca, G., Foyer, C., Jeannot, E., et al.: Online Dynamic Monitoring of MPI Communications. In: European Conference on Parallel Processing, 28 Aug-1 Sep. 2017, Santiago de Compostela, Spain. pp. 49–62. Springer, Cham (2017), DOI: 10.1007/978-3-319-64203-1_4

7. Broquedis, F., Clet-Ortega, J., Moreaud, S., et al.: Hwloc: A generic framework for managing hardware affinities in HPC applications. In: 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing, 17-19 Feb. 2010, Pisa, Italy. pp. 180–186. IEEE (2010), DOI: 10.1109/PDP.2010.67

8. Buntinas, D., Mercier, G., Gropp, W.: Implementation and shared-memory evaluation of MPICH2 over the Nemesis communication subsystem. In: Mohr, B., Träff, J.L., Worringen, J., Dongarra, J. (eds.) Recent Advances in Parallel Virtual Machine and Message Passing Interface, 17-20 Sept. 2006, Germany. pp. 86–95. Springer, Berlin, Heidelberg (2006), DOI: 10.1007/11846802_19

9. Chen, H., Chen, W., Huang, J., et al.: MPIPP: an automatic profile-guided parallel process placement toolset for SMP clusters and multiclusters. In: Proceedings of the 20th Annual

International Conference on Supercomputing, 28 June-1 July, 2006, Cairns, Queensland, Australia. pp. 353–360. ACM (2006), DOI: 10.1145/1183401.1183451

10. Dashti, M., Fedorova, A., Funston, J., et al.: Traffic management: A holistic approach to memory placement on NUMA systems. In: Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, Houston, Texas, USA. pp. 381–394. ACM, New York, NY, USA (2013), DOI: 10.1145/2451116.2451157

11. David, H., Gorbatov, E., Hanebutte, U.R., et al.: RAPL: Memory power estimation and capping. In: 2010 ACM/IEEE International Symposium on Low-Power Electronics and Design, 18-20 Aug. 2010, Austin, TX, USA. pp. 189–194. IEEE (2010), DOI: 10.1145/1840845.1840883

12. Diener, M., Cruz, E.H., Alves, M.A., et al.: Affinity-based thread and data mapping in shared memory systems. ACM Computing Surveys 49(4), 64 (2017), DOI: 10.1145/3006385

13. Diener, M., Cruz, E.H., Navaux, P.O., et al.: Communication-aware process and thread mapping using online communication detection. Parallel Comput. 43(C), 43–63 (2015), DOI: 10.1016/j.parco.2015.01.005

14. Dózsa, G., Kumar, S., Balaji, P., et al.: Enabling concurrent multithreaded MPI communication on multicore petascale systems. In: Proceedings of the 17th European MPI Users' Group Meeting Conference on Recent Advances in the Message Passing Interface, 12-15 Sept. 2010, Stuttgart, Germany. pp. 11–20. Springer-Verlag, Berlin, Heidelberg (2010), DOI: 10.1007/978-3-642-15646-5_2

15. Gabriel, E., Fagg, G.E., Bosilca, G., et al.: Open MPI: Goals, concept, and design of a next generation MPI implementation. In: European Parallel Virtual Machine/Message Passing Interface Users Group Meeting, 19-22 Sept. 2004, Budapest, Hungary. pp. 97–104. Springer, Berlin, Heidelberg (2004), DOI: 10.1007/978-3-540-30218-6_19

16. Gaud, F., Lepers, B., Funston, J., et al.: Challenges of memory management on modern NUMA systems. Commun. ACM 58(12), 59–66 (2015), DOI: 10.1145/2814328

17. Goglin, B., Moreaud, S.: KNEM: A generic and scalable kernel-assisted intra-node MPI communication framework. Journal of Parallel and Distributed Computing 73(2), 176–188 (2013), DOI: 10.1016/j.jpdc.2012.09.016

18. Gropp, W.: MPICH2: A new start for MPI implementations. In: Recent Advances in Parallel Virtual Machine and Message Passing Interface, 19-22 Sept. 2004, Budapest, Hungary. pp. 7–7. Springer, Berlin, Heidelberg (2002), DOI: 10.1007/3-540-45825-5_5

19. Hofmann, J., Fey, D., Eitzinger, J., et al.: Analysis of Intel's Haswell Microarchitecture Using the ECM Model and Microbenchmarks. In: Architecture of Computing Systems, ARCS 2016, 4-7 April 2016, Nuremberg, Germany. pp. 210–222. Springer, Cham (2016), DOI: 10.1007/978-3-319-30695-7_16

20. Intel: Intel Xeon Processor E5 and E7 v4 Product Families Uncore Performance Monitoring Reference Manual. `https://www.intel.com/content/www/us/en/products/docs/processors/xeon/xeon-e5-e7-v4-uncore-performance-monitoring.html` (2016)

21. Jeannot, E., Mercier, G., Tessier, F.: Process placement in multicore clusters:algorithmic issues and practical techniques. IEEE Transactions on Parallel and Distributed Systems 25(4), 993–1002 (2014), DOI: 10.1109/TPDS.2013.104

22. Kerrisk, M.: Linux/UNIX System Programming: POSIX Shared Memory. http://man7.org/training/download/posix_shm_slides.pdf (2015), accessed: 2019-05-14

23. Lepers, B., Quéma, V., Fedorova, A.: Thread and memory placement on NUMA systems: Asymmetry matters. In: Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference, 8-10 July 2015, Santa Clara, CA. pp. 277–289. Berkeley, CA, USA (2015), DOI: 10.5555/2813767.2813788

24. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard. http://www.mpi-forum.org (2012)

25. Molka, D., Hackenberg, D., Schöne, R.: Main Memory and Cache Performance of Intel Sandy Bridge and AMD Bulldozer. In: Proceedings of the Workshop on Memory Systems Performance and Correctness, Edinburgh, United Kingdom. pp. 4:1–4:10. ACM, New York, NY, USA (2014), DOI: 10.1145/2618128.2618129

26. Orduña, J.M., Silla, F., Duato, J.: On the development of a communication-aware task mapping technique. J. Syst. Archit. 50(4), 207–220 (2004), DOI: 10.1016/j.sysarc.2003.09.002

27. PRACE: Unified European Applications Benchmark Suite. www.prace-ri.eu/ueabs (2013), accessed: 2019-10-01

28. Sodani, A.: Knights landing (KNL): 2nd Generation Intel Xeon Phi processor. In: 2015 IEEE Hot Chips 27 Symposium, 22-25 Aug. 2015, Cupertino, CA, USA. pp. 1–24. IEEE (2015), DOI: 10.1109/HOTCHIPS.2015.7477467

29. Treibig, J., Hager, G., Wellein, G.: Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In: Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures, 13-16 Sept. 2010, San Diego, CA, USA. pp. 207–216. IEEE (2010), DOI: 10.1109/ICPPW.2010.38

30. Zhai, J., Sheng, T., He, J., et al.: Efficiently acquiring communication traces for large-scale parallel applications. IEEE Transactions on Parallel and Distributed Systems 22(11), 1862–1870 (2011), DOI: 10.1109/TPDS.2011.49

31. Zhang, J., Zhai, J., Chen, W., et al.: Process mapping for MPI collective communications. In: Euro-Par 2009 Parallel Processing, 25-28 Aug. 2009, Delft, The Netherlands. pp. 81–92. Springer, Berlin, Heidelberg (2009), DOI: 10.1007/978-3-642-03869-3_11

32. Ziakas, D., Baum, A., Maddox, R.A., et al.: Intel QuickPath Interconnect architectural features supporting scalable system architectures. In: 2010 18th IEEE Symposium on High Performance Interconnects, 18-20 Aug. 2010, Mountain View, CA, USA. pp. 1–6. IEEE (2010), DOI: 10.1109/HOTI.2010.24

33. Zivanovic, D., Pavlovic, M., Radulovic, M., et al.: Main memory in HPC: Do we need more or could we live with less? ACM Trans. Archit. Code Optim. 14(1), 3:1–3:26 (2017), DOI: 10.1145/3023362