A Memory Congestion-Aware MPI Process Placement for Modern NUMA Systems

Conference Paper · December 2017

DOI: 10.1109/HiPC.2017.00026

Mulya Agung
Tohoku University
14 PUBLICATIONS **13** CITATIONS
SEE PROFILE

Muhammad Alfian Amrizal
Tohoku University
19 PUBLICATIONS **28** CITATIONS
SEE PROFILE

Kazuhiko Komatsu
Tohoku University
51 PUBLICATIONS **328** CITATIONS
SEE PROFILE

Ryusuke Egawa
Tokyo Denki University
114 PUBLICATIONS **383** CITATIONS
SEE PROFILE

A Memory Congestion-aware MPI Process Placement for Modern NUMA Systems

Mulya Agung*, Muhammad Alfian Amrizal†, Kazuhiko Komatsu†, Ryusuke Egawa†, and Hiroyuki Takizawa†
*Graduate School of Information Sciences, Tohoku University
Sendai, Miyagi 980-8578, Japan
Email: {agung@sc.cc., alfian@sc.cc., komatsu@, egawa@, takizawa@}tohoku.ac.jp
†Cyberscience Center, Tohoku University

***Abstract*—MPI process placement is an important step to achieve scalable performance on modern non-uniform memory access (NUMA) systems. A recent study on NUMA architectures has shown that, on modern NUMA systems, the memory congestion problem could cause more severe performance degradation than the data locality problem because heavy congestion on memory controllers could cause long latencies. However, conventional work on MPI process placement has focused on locality to minimize the remote-access communication. Moreover, maximizing the locality may actually degrade performance because the load imbalance among nodes in a modern NUMA system may increase. Thus, a process placement algorithm must be designed to consider memory congestion. In this paper, a method to reconcile both the locality and the memory congestion on modern NUMA systems is proposed. This method statically analyzes the application communication pattern to optimize the process placement. A data clustering method is applied to the time-series data of the MPI communications in order to identify data traffics that potentially cause memory congestion. The proposed method has been evaluated with the NPB kernels on a real NUMA system and a simulation environment. Experimental results show that the proposed method can achieve 1.6x performance improvement compared with the current state-of-the-art strategy.**

***Keywords*-process placement; high performance computing; NUMA; many-core; congestion**

I. Introduction

Since the early 2000s, the number of cores in a processor has been increasing. All processor cores are not necessarily connected in a flat fashion, and they might be grouped for several reasons. For example, only the processor cores in the same group might share one cache memory. Moreover, in the fields of high-performance computing (HPC) and enterprise computing, a large-scale Symmetric Multi-Processing (SMP) system is often built by using multiple processors to have more processor cores within a system. In such a system, each processor can be considered as one or more groups of processor cores, and thus the system consists of multiple groups of processor cores. Such a group of processor cores is called a *node* if it is associated with a memory node of one or more dedicated memory controllers and memory devices [1][2][3][4]. Note that, although all the processor cores logically share a single memory space, each memory device is physically associated with a particular node in a modern SMP system. Figure 1 shows an example of the modern SMP system with four nodes and eight cores per node. Each node has a multicore CPU and a local DRAM connected by the memory controllers. Accessing the data on the memory device of another node is called a *remote-access communication*. In Figure 1, the remote-access communications are represented by the double arrows. Although the nodes are generally connected by high-speed interconnect links such as QuickPath Interconnect (QPI) [3], accessing a remote node still needs a longer latency than accessing the data of the local memory device. Since the memory access latency depends on the physical location of the data to be accessed, such a system is often called a non-uniform memory access (NUMA) system.

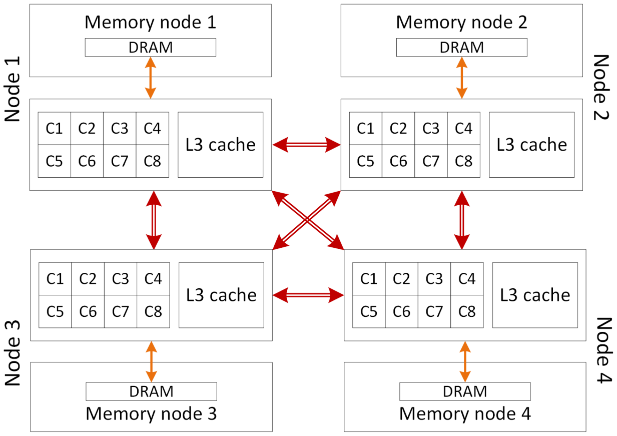


Figure 1. A modern NUMA system, with four nodes and eight cores per node.

To explicitly express data movement, this work assumes to use the Message Passing Interface (MPI) [5] to write parallel programs for such a system. Under the assumption, this paper discusses process placement in a NUMA system. Process placement is a crucial step to improve the application performance due to the non-uniform communication costs. In a so-called flat MPI application [6] discussed in this paper, the placement is achieved by mapping MPI processes onto processor cores, and assigning a unique identification number, a so-called MPI rank, to every MPI process. In practice, the process manager of the MPI implementation binds each MPI process to a processor core of a node according to its mapping policy.

Conventional work on process placement has mainly focused on the data access locality to reduce the performance penalty due to the remote-access latencies. The locality is improved by matching the communication pattern of the application to the topology of the underlying hardware [7][8][9]. Even in the case of a NUMA system considered in this paper, the locality-optimized approaches increase performance by decreasing the number of remote-access communications if the number of processor cores in one node is not large. However, a recent study on NUMA architectures has shown that the congestion on memory controllers and interconnect links, caused by memory traffics from data-intensive applications, significantly decreases performance of modern NUMA systems [1]. On a modern NUMA system, delays resulting from traversing a longer physical distance to reach a remote node are no longer the most important source of performance overhead [10][11].

If the number of processor cores in one node further increases, optimizing locality without considering memory congestion may decrease performance. While locality plays an important role in reducing the traffic on the interconnect links, it may cause congestion on memory controllers due to the increasing traffics on the controllers. Therefore, reconciling locality and congestion becomes an important problem to optimize the process placement on modern NUMA systems.

In this paper, a method to address the reconciling problem in MPI process placement is proposed. The proposed method considers both locality and congestion by monitoring the communication traffics in advance, analyzes the traffics, and then decides the process mapping by using the analysis results. It minimizes the load imbalance among the nodes, while also reducing the communication costs of the application. An algorithm called Congestion-aware Load Balance (CLB) is proposed to compute an optimized process placement that determines how to map these processes on the processor cores.

This paper presents CLB and several techniques to build a model of data traffics and also a model of the NUMA system topology. These techniques include a low-level monitoring tool to capture the time-series data of the MPI communications, and a data clustering method to identify the traffics that can cause congestion from the time-series data. Moreover, for validation purposes, performance evaluation using a modern NUMA system and a simulation environment has been conducted. The evaluation results show that the process placement computed by the proposed method achieves a shorter execution time than those of two greedy policies, and one of the state-the-art locality-optimized method, called Treematch [7].

The rest of the paper is organized as follows. Section II presents the problem and the method discussed in this work. Experiments that validate the proposed method are analyzed in Section III. The related studies are described in Section IV. Finally, Section V gives the conclusions and the future work of this paper.

II. Problem Statement and Method Description

This work focuses on MPI process placement that can reconcile the locality and congestion problems on modern NUMA systems. These problems emerge from the way of exchanging data among processes of an MPI application either through memory controllers or through interconnect links. An MPI application distributes its workloads among entities called MPI processes that run in parallel on the processors of a system. In an MPI application, all processes send and receive messages during the application execution. The data exchanges among MPI processes can be irregular, and an MPI process does not always communicate with all the other MPI processes. In addition, the time when the exchange occurs and the amount of data exchanged among processes may vary. The pattern of the data exchanges among MPI processes of an application is referred to as the communication pattern of the MPI application [12][13][14].

On a modern NUMA system, the data exchanges among processes running on the same node induce the traffic on the memory controllers of the node. On the other hand, the exchanges between processes running on two distinct nodes induce the traffic on interconnect and memory controllers on the remote nodes. As a result, if the traffic on a memory controller exceeds the bandwidth of the memory controller, the memory congestion will happen.

As described above, the communication pattern of an MPI application can be irregular. This means that some processes can induce more traffic to memory controllers and interconnect than other processes. As the amount of data exchanged becomes larger, load imbalance between the memory controllers and interconnect links will be getting worse. Moreover, each node of modern NUMA systems can have many processor cores. At the time when this paper is written, one node has up to 28 physical processor cores [15]. The traffic on memory controllers of one node will be higher because more processes can be placed onto one node. Therefore, the possibility of causing memory congestion in such systems would be higher than that in the previous systems.

The proposed process placement method considering both locality and the congestion is performed in the following steps:

1) Gather the node topology information of the target system.
2) Gather the communication pattern of the target application.
3) Identify the data traffics that can cause memory congestion.
4) Compute a mapping between the MPI process ranks and the processor cores.

The first step is to retrieve the node topology information of the target system. To achieve this, the topology of hardware components is investigated by using some tools such as Hwloc [4]. The topology is modeled as a tree to express the information about the locations of memory controllers and interconnect links. This model also includes the physical identity information of the nodes and processor cores. This information is required later by the mapping algorithm to match the MPI processes with the physical cores.

The second step is to gather the communication pattern of the target application. Currently, the proposed method relies on the application profiling by preliminarily running the target application. The method considers both spatial and temporal patterns of the communication. In addition, the number of MPI processes is considered to be constant during the application execution. To gather information of the communication pattern, time series data of the data exchanges are recorded by tracing the timestamp, the number of messages, and the data size of each MPI communication event. For the information gathering, in this work, we have implemented a low-level monitoring tool in an MPI implementation, Open MPI [16]. The monitoring tool is implemented using a monitoring framework that has been proposed in [17]. A new component has been added to the point-to-point management layer (PML) of the Open MPI stack to record the timestamp information. Since the PML can monitor point-to-point operations organizing a collective communication, the time-series data of the data exchanges can be traced in both cases of point-to-point and collective communications. Moreover, by tracing only a limited number of profiling elements, the monitoring does not severely disturb the application execution.

After the time-series data are obtained, the data are analyzed to identify when and where each communication event happened. Two processes that communicate with each other during the application execution are called a pair of processes. The concentrated traffic is caused by the concurrent communications, in the sense that multiple pairs of processes can communicate at the same time. These pairs can be identified by using the process rank and timestamp information of the communication events. In the proposed method, pairs of processes are grouped if those pairs communicate at almost the same time. A pair of processes can belong to more than one group if the pair communicates with each other several times at different timestamps.

In order to obtain the groups of pairs of processes, K-means clustering [18] is used. This clustering method needs to predefine a parameter, K, to specify the number of groups. However, the actual number of groups is generally unknown in advance, even though the parameter certainly affects the clustering results. Thus, it is important to estimate an optimal number of groups in advance of clustering. In this work, the Goodness of Variance Fit test (GVF) [19] is used for the estimation. The clustering method starts with $K = 2$, and then K is increased so as to maximize the GVF value, which becomes higher as the variance of timestamps within a group becomes smaller. As a result, pairs of processes are classified into the same group if they have similar timestamps.

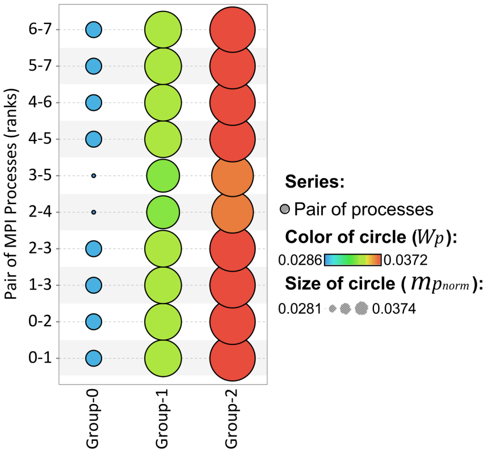


Figure 2. An example of groups for the CG kernel of NPB.

It is possible that the concentrated traffic happens several times and the communication pattern may vary each time. Since the placement of MPI processes is static, there is only one ideal mapping for the whole execution of an application. Therefore, the groups with higher traffics must have a higher priority to be optimized than those with lower traffics. To determine the order of priority for optimization, two metrics are calculated. The first metric is the load of a pair of processes, W_p, which is defined as the weighted sum of the normalized number of messages and the normalized size of messages exchanged by the pair of processes in every group:

$$Wp = \alpha \frac{m_p}{\sum_{i=1}^{n} m_i} + \beta \frac{s_p}{\sum_{i=1}^{n} s_i},$$

where m_p is the number of messages exchanged by a pair p, s_p is the size of messages exchanged by a pair p, and n is the number of pairs in all groups. The normalized values of the number of messages and the size of messages are weighted by α and β, respectively.

The second metric is L_c, which is the load of a group c:

$$Lc = \sum_{i=1}^{N_{p_c}} Wp_i,$$

where N_{p_c} is the number of pairs in a group, c, and Wp_i is the value of W_p for the i-th pair in a group, c.

Figure 2 shows an example of groups for the CG kernel in the NAS parallel benchmark suite (NPB) [20]. The vertically-aligned circles represent the process pairs in the same group. Each pair is identified by a pair of two MPI ranks whose processes communicate. The color and size of a circle represent the values of W_p and normalized m_p of the pair, respectively. This figure shows that the communication pattern of the CG kernel is irregular because the W_p values of the pairs of processes vary, and some

Algorithm 1 The CLB Algorithm.

```
Input: $n$ {number of nodes}
Input: $np$ {number of processes}
Input: $G$ {groups of process pairs}
Output: $B$ {buckets of processes}
1: $B \leftarrow create\_buckets(n, np)$
2: $sorted\_G \leftarrow sort\_by\_lc(G)$
3: $i \leftarrow 0$
4: while $size(B) < np$ and $i < size(G)$ do
5:     $P \leftarrow sorted\_G[i]$
6:     $sorted\_P \leftarrow sort\_by\_wp(P)$
7:     $distribute\_pairs(B, sorted\_P)$
8:     $i \leftarrow i + 1$
9: end while
```

processes do not communicate with each other. It is shown that the communications among eight MPI processes are characterized by the three groups of process pairs, and the heaviest traffic is caused by Group 2. If the processes in Group 2 are all placed onto the same node, the congestion on memory controllers will happen. Therefore, in order to reduce the congestion, the data traffics of this group will be distributed over different nodes.

After obtaining the number of groups, W_p of each process pair, and L_c of each group, the CLB algorithm is used to match the processes with the processor cores This algorithm, depicted in Algorithm 1, is able to compute a matching between the process ranks to the processor cores. The basic idea of this algorithm is to reduce the load imbalance by distributing the heavily-communicating pairs in terms of W_p over different nodes, while also reducing the number of remote-access communications by placing the two communicating processes to the same node.

The algorithm works as follows: first, CLB uses the topology model to construct the buckets in an array B with size n (Line 1). Each bucket represents a node and the capacity of each bucket represents the number of processor cores of the node. Then, all of the processes are put into the buckets in an order of their loads. This is greedily calculated with the constraints that a process cannot be in two or more buckets, and a bucket cannot have more than one process with the same rank. The algorithm considers the pairs sequentially from the largest W_p to the smallest W_p and from the groups with largest L_c to the smallest L_c. This is achieved by the sorting steps in the algorithm (Lines 2 and 6). In order to reduce the number of remote-access communications, the two processes of each pair are mapped to the same bucket as long as the constraints are not violated. The *distribute_pairs* function then distributes the load of each group to the nodes by mapping the pairs to the buckets in a round-robin fashion (Line 7). This function aims to balance the load of the heavily-communicating groups among the nodes as evenly as possible.

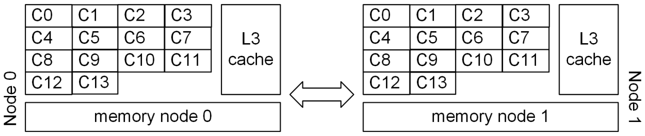


Figure 3. A topology of the Purple system, with two nodes and 14 cores per node.

III. Experimental Evaluation

In order to evaluate the performance gain by the proposed method, two experiments have been conducted. This section shows the experimental evaluation results and discusses the performance gain.

A. Evaluation on a Real System

The first experiment has been conducted on an Intel-based NUMA system, called *Purple*. In Section III-A, the experimental setup and methodology are described.

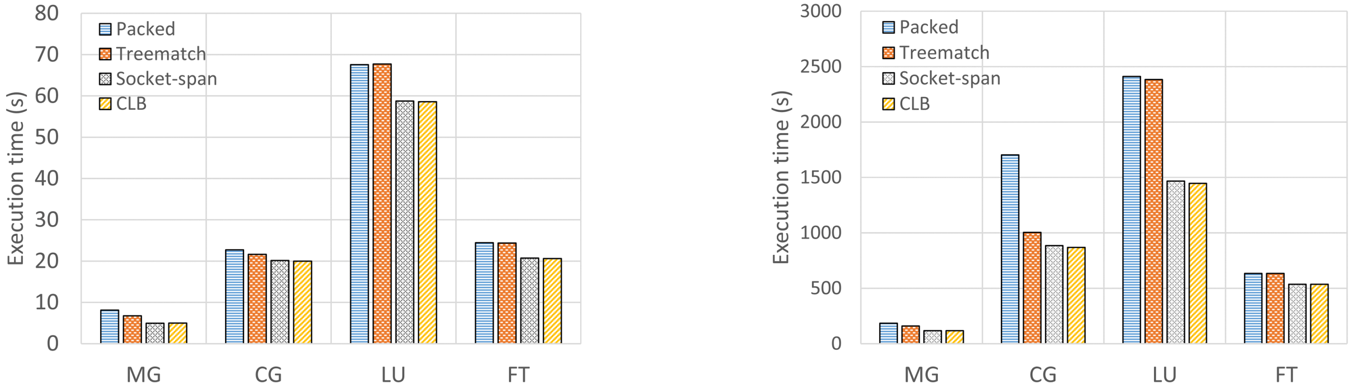
1) Experimental Setup: The Purple system is equipped with the main memory of 128 GB and two Xeon E5-2680-v4 processors of 14 cores each running at 2.6 GHz. The system consists of two nodes, each of which has 14 cores and 64 GB memory. Those nodes are connected via QuickPath Interconnect [3]. The node topology of the system is shown in Figure 3. Open MPI v3.0 [16] and Hwloc v1.11 [4] used in the following experiment.

Four particular kernels of NPB are used in the experiment:

- Multi-Grid (MG) kernel of irregular communication.
- Conjugate Gradient (CG) kernel of irregular memory access and communication.
- Fourier Transform (FT) kernel of the all-to-all communication pattern.
- Lower-Upper Gauss-Seidel kernel (LU) featuring a solver with irregular memory access.

CLB is compared with three process placement policies: Packed, Socket-span, and Treematch. Packed and Socket-span are greedy process placement policies. Packed corresponds to the logical identity (process i is mapped onto processor core i). In the Socket-span policy, processes are mapped onto processor cores in different nodes. This policy assigns MPI processes to different nodes in a round-robin fashion. Both Packed and Socket-span do not consider the application communication pattern for the process placement. In contrast, Treematch is a locality-optimized placement strategy. It maps processes onto cores depending on the amount of data exchanged among them. Treematch v0.4.1 is used in this work. For CLB, W_p is calculated with α = β = 1. Thus, the number and the size of messages have a same degree of contribution to W_p.

In this experiment, two classes C and D of the NPB kernels are used as the average and large problem sizes, respectively. Since the Purple system has 28 processor cores in total, the maximum number of MPI processes without



(a)

(b)

Figure 4. Average execution times of the NPB kernels for class C (a) and class D (b).

Table I
THE MAPPING RESULTS OF THE MG KERNEL WITH 16 PROCESSES.

	MPI Rank	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Mapping (node:core)	Packed	0:0	0:1	0:2	0:3	0:4	0:5	0:6	0:7	0:8	0:9	0:10	0:11	0:12	0:13	1:0	1:1
	Treematch	1:0	1:1	1:2	1:3	1:7	1:8	1:9	1:10	1:11	1:12	1:13	1:4	0:0	0:1	0:2	0:3
	Socket-span	0:0	1:0	0:1	1:1	0:2	1:2	0:3	1:3	0:4	1:4	0:5	1:5	0:6	1:6	0:7	1:7
	CLB	0:4	1:4	0:5	1:5	1:2	0:2	1:3	0:3	0:6	1:6	0:7	1:7	1:0	0:0	1:1	0:1

oversubscription is 28. However, the number of processes executing the MG, CG, and FT kernels must be power-of-two. Thus, 16 processes are launched for executing these kernels, and 28 processes using all the cores are launched only for the LU kernel.

2) Performance Comparison: Figures 4(a) and 4(b) depict the average execution times of the NPB kernels for classes C and D, respectively. Each average is calculated from 10 sample executions, and the sample means have 95% confidence intervals between 0.07 and 1.73 percentage of the mean. Thus, it is statistically reliable that the average execution times in Figures 4(a) and 4(b) are close enough to the true means of the samples. On average, CLB outperforms Packed and Treematch. Treematch shows a better performance than Packed because Treematch reduces the number of remote-access communications by considering the locality. However, to avoid remote-access communications, Treematch assigns heavily-communicating pairs to the same node as much as possible. As a result, it increases the number of memory access requests to memory controllers of a particular memory node. Since the load imbalance among memory controllers becomes worse, memory congestion potentially increases the total execution time. On the other hand, CLB prevents memory congestion by alleviating the load imbalance among memory controllers. As a result, CLB shows a better performance than Packed and Treematch. It is interesting to notice that Socket-span can achieve a shorter execution time than Treematch for all tested kernels. Although Socket-span does not consider the communication patterns of the kernels, its mapping distributes the data traffics over the two nodes. Hence, Socket-span alleviate the load imbalance among the two nodes.

Socket-span can achieve a comparable execution time with CLB because, on this system, the mapping results of both methods are similar. Table I shows the mapping result of each method for the MG kernel. Each MPI process is represented by the rank number, and each processor core is represented by a pair of identification numbers, (node:core). For example, core 2 (C2) of Node 1 in Figure 3 is represented by (1:2). It is shown that, although the core mappings of Socket-span and CLB are different, the node mappings of both methods are almost the same. In Socket-span, neighboring MPI processes are mapped to different nodes, thus node is allocated in the way of 0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1. In CLB, the sequence of the node identification number is 0,1,0,1,1,0,1,0,0,1,0,1,1,0,1,0. Neighboring MPI processes in CLB are likely to be allocated to different nodes, unlike in Packed and Treematch. As a result, the performance of CLB is similar with that of Socket-span.

As shown in Figure 4(b), when the problem size becomes larger, CLB and Socket-span show even larger performance gains. The maximum performance gain is obtained for the LU kernel because the amount of data exchange of the LU kernel is the largest among the tested benchmarks. For the LU kernel, both CLB and Socket-span show 1.6x performance improvement compared with Packed and Treematch. For the FT kernel, the performance gain is unchanged even if the problem size is changed. This is due to its all-to-all communication pattern. The remote-access penalty in such a small system is not large, and hence process placement

Table II
THE NORMALIZED EXECUTION TIMES OF THE PLACEMENT METHODS.

	Class C				Class D			
	MG	CG	LU	FT	MG	CG	LU	FT
Packed	1.62	1.14	1.15	1.18	1.57	1.96	1.67	1.18
Treematch	1.35	1.08	1.16	1.18	1.36	1.16	1.65	1.18
Socket-span	1.00	1.01	1.00	1.00	1.00	1.02	1.01	1.00

Table III
MEMORY CONGESTION EFFECTS OF THE MG KERNEL.

	Packed	Treematch	Socket-span	CLB
LLC miss ratio	31.74	23.05	22.9	22.73
Mem-ctrl imbalance	112.99	**77.52**	0.91	**0.33**
Memory latency	186.55	**185.81**	164.3	**159.62**
IPC	1.55	**1.86**	2.19	**2.28**

does not affect the performance.

Table II shows the normalized execution time of each method. In this table, the execution time of CLB is used as the baseline, and the normalized execution time of each method is computed. Hence, the results show how much each method needs a longer time than CLB. Lower is better. For all of the tested kernels, the normalized execution times of Packed and Treematch increase with the problem size. This is because the load imbalance among the nodes increases with the problem size, and both of them cannot prevent the load imbalance. In contrast, CLB can reduce the load imbalance for the two problem sizes. Hence, CLB achieves a better performance than the two methods. The experimental results show that, on this system, memory congestion has a larger impact on the execution time than the remote-access communication penalty.

For further discussions, the behaviors of the MG kernel are investigated using the Linux Perf tool [21]. In addition, Uncore driver of the Intel CPU [22] is used to profile the memory controller-related events. To illustrate that the performance differences are due to memory congestion, some supporting data for the placement methods are shown in Table III. Four key metrics are used for the analysis. LLC miss ratio is the percentage of L3 cache miss across all nodes. Memory controller imbalance is the standard deviation of the bandwidth utilization of the memory controllers across all nodes. A higher value of this metric implies a higher risk of memory congestion. Memory latency is the average number of cycles to complete a memory request from any node. Higher latencies will increase the execution time, because the CPU must stall for a longer time upon an LLC miss. IPC is the number of instructions per cycle. Thus, a larger IPC means a higher performance.

The data in Table III lead to three observations. First, Packed shows the lowest IPC because it has the highest LLC miss ratio and the longest memory latency. Second, CLB and Socket-span show a higher IPC than Treematch even though the difference in LLC miss ratio is small. This is because the memory latencies of CLB and Socket-span are smaller than that of Treematch. Note that the memory latency increases with the load imbalance of the memory controllers. Finally, both CLB and Socket-span significantly reduce the memory congestion by preventing the load imbalance. CLB results in the lowest load imbalance because it considers the communication pattern of the kernel to balance the load of the heavily-communicating processes among the nodes.

B. Simulation on SimGrid

The experiment results on the Purple system show that, in some cases, the Socket-span strategy can achieve the comparable performance gain with CLB. However, when the numbers of nodes and processes become larger, Socket-span might be unable to maintain the locality, and also unable to reduce the load imbalance accurately because it does not consider the communication pattern of an application. In order to examine the congestion effects when the numbers of nodes and processes become large, simulations of a larger-scale system are conducted. Furthermore, based on the simulation results, the performance gain of the proposed method under a heavier congestion environment is discussed. The SimGrid/SMPI simulator v3.14 [23] is used for the experiments because of its capability to capture the effect of communication distance and network congestion on the simulation. The NPB kernels are run using the online simulation approach that is supported by the SMPI framework [24]. In this approach, the kernels are executed on the virtual platform that attempts to mimic the behavior of the target platform.

1) Simulation Setup: Currently, SimGrid only supports the platform configuration for the network clusters. Hence, in order to simulate the memory congestion effects on the modern NUMA systems, a virtual platform of SimGrid is configured to imitate the topology of a modern NUMA system. The host resources of the SimGrid configuration represent the processor cores, and the link resources represent the memory controllers and interconnect links. In addition, a routing configuration represents the route between two processor cores. The memory controller links have a lower latency and a higher bandwidth than the interconnect links, and the usage of these links depends on the communication route between processor cores. The communication between processor cores in the same node uses the memory controller link associated with the node. On the other hand, the communication between two processor cores in distinct nodes uses the interconnect link between the nodes and the memory controller link of the remote node.

2) Simulation Verification: The accuracy of the simulation is verified by comparing its performance results to those of the real runs on the Purple system. The performance is defined as the execution time, and the results are represented by the execution time ratio of other methods to that of CLB. The verification uses the same NPB kernels and the same execution configuration such as the number of processes

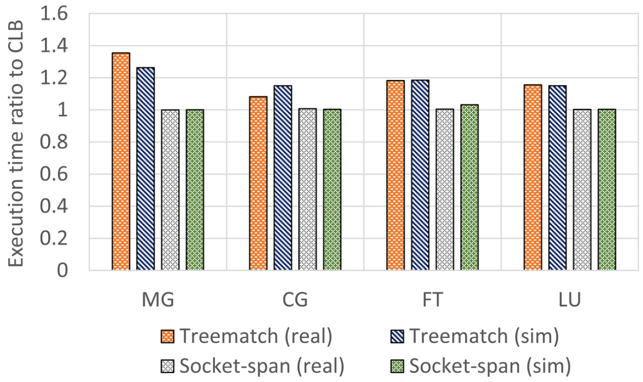


Figure 5. The verification results of the simulation.

used in Section III-A. The node topology of the Purple system shown in Figure 3 is used for the verification of the SimGrid simulation.

The platform was configured with 28 host resources and four link resources. These host resources represent the 28 processor cores of the Purple system. Two links are for the memory controllers, and the other two links are for the interconnect links. Each memory controller connects the host resources of the same node, and each interconnect link connects two host resources of the distinct nodes. The latency and bandwidth of each link are configured to their actual values observed on the Purple system using Intel Memory Latency Checker tool [25]. Since the congestion is sensitive to the bandwidths of memory controllers and interconnect links, to discuss the performance under a heavier congestion environment, these bandwidths are reduced while keeping their ratio constant. Even if the same application is executed, a heavier congestion will happen if the bandwidths are small. Finally, in order to simulate the behaviors of collective communications on the Purple system with Open MPI, the MPI environment in the simulation is configured to mimic Open MPI by using `--cfg=smpi/coll_selector:ompi`.

The verification results are shown in Figure 5. These results show that the simulation results are similar to those on the real system; the difference in execution time between simulation and experimental results is less than 8%. Therefore, the results demonstrate that the simulation is accurate enough to discuss the effects of the proposed method on the performance of a larger-scale system.

3) Performance Comparison: After the accuracy of the simulation results is verified, the performance evaluation using a different topology and several bandwidth configurations is conducted on the simulation environment. Packed was excluded in the following evaluation because, as shown in the previous results, it is already obvious to be the slowest placement method. The topology of a four-node NUMA system, shown in Figure 1, is used for the simulation. This topology represents the four-socket Intel QPI-based systems [3]. In addition to eight physical processor cores with one memory controller in each node, six interconnect links are employed on the system. Hence, in order to utilize all of the processor cores, 32 MPI processes are executed on the system. In the simulation, the performance is evaluated while changing the bandwidth in order to discuss the performance under a heavier congestion environment. The baseline bandwidth is the same as in the simulation verification. Then, the bandwidth is decreased to be 50%, 25%, 12%, and 5% of the baseline bandwidth.

Figure 6 shows the performance evaluation results of the tested kernels for the different bandwidth configurations. The execution time with CLB is always comparable to or shorter than those with the other methods. The results are evaluated by analyzing the latencies and bandwidth utilization of the links. The utilization information is provided by using the tracing feature of SimGrid.

Figure 6(a) shows the performance results of the CG kernel. In this figure, CLB shows a better performance than other methods in all the tested bandwidth configurations. This is because the communication pattern of the CG kernel is irregular. Socket-span distributes all the processes to the four nodes without considering the communication pattern. As a result, Socket-span suffers from high latencies of remote-access communication of the heavily-communicating processes. On the other hand, Treematch assigns the heavily-communicating processes to the same node. Hence, memory congestion increases as the allocated bandwidth of the memory controller becomes smaller. In contrast, CLB can effectively reduce the congestion by preventing the load imbalance. Furthermore, it reduces the remote-accesses of all processes by considering communication locality. Therefore, CLB achieves the best performance compared with the other methods.

Figure 6(b) depicts the performance evaluation results of the FT kernel. It is shown that Socket-span needs two times longer execution time than CLB, and thus CLB can achieve two times better performance than Socket-span. This is because the FT kernel features all-to-all communication pattern. Thus, Socket-span suffers from higher latencies of remote-access communications among all processes. On the other hand, both CLB and Trematch reduce the number of remote-accesses by considering communication locality. As a result, the difference in execution time between these two methods is only 7% as shown in the figure. However, the difference increases as the bandwidth decreases. This is because CLB can reduce memory congestion by preventing the load imbalance.

The LU kernel is in between the CG and FT kernels in term of regularity of the communication pattern. Hence, the results look similar to those in Figure 6(b). Similar trends as in Figure 6(c) can be seen in the case of executing the MG kernel in Figure 6(d). This is because the communication pattern of the MG kernel is also irregular. Since the data traffic from collective communications in the MG kernel is

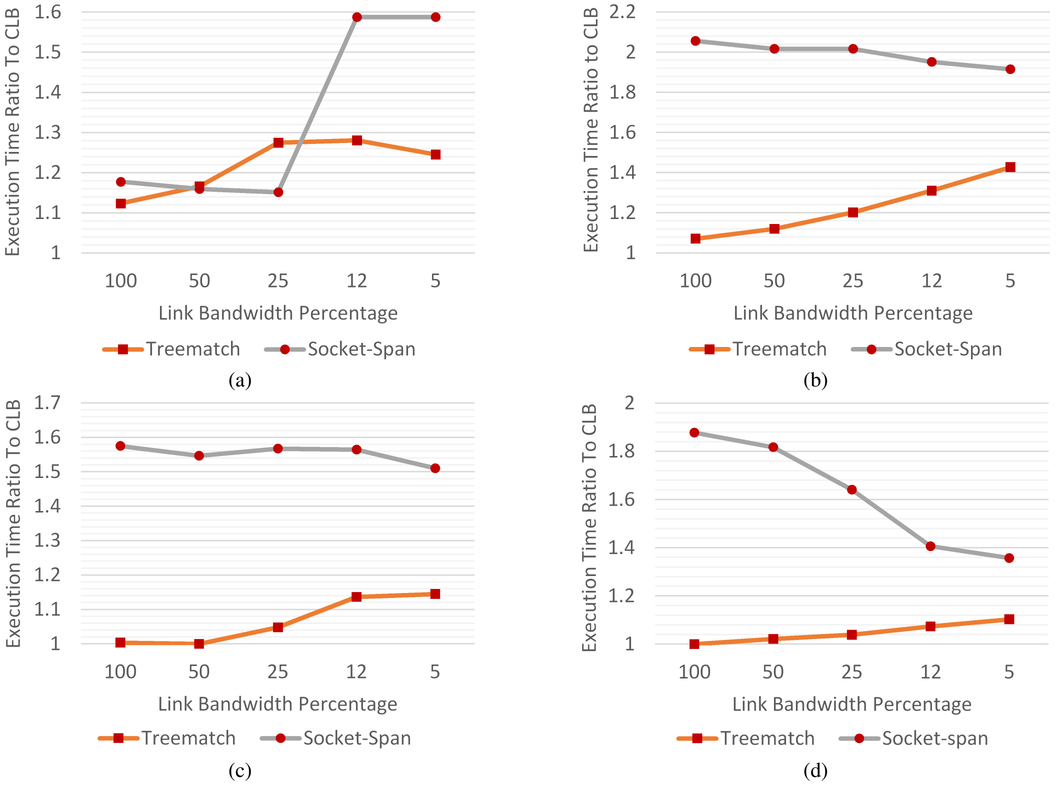


Figure 6. The simulation results of NPB kernels. (a) CG, (b) FT, (c) LU, (d) MG.

larger than that in the LU kernel, Socket-span suffers from higher latencies of remote-access communications among all processes. Thus, the difference in execution time between CLB and Socket-span in the MG kernel is larger than that in the LU kernel. The difference in execution time between CLB and Treematch in the MG kernel is smaller than those in the other kernels because the amount of data exchanged in the MG kernel is smaller than those in the other kernels.

The results of the LU and MG kernels also show that, when the bandwidth configurations are 100% (Figures 6(c) and 6(d)) and 50% (Figure 6(c)) of the baseline, the execution times of CLB and Treematch are equal. This is because the congestion level is low, and thus the load imbalance among the nodes has less impact on the execution time than the remote-access cost. Both CLB and Treematch can achieve a shorter execution time than that of Socket-span because both methods reduce the number of remote-accesses. However, when the congestion level increases, the execution time ratio of Treematch to CLB becomes higher. CLB can achieve a shorter execution time than that of Treematch by reducing the memory congestion. Thus, the results in Figure 6 show that CLB can achieve the best gain of performance in all the tested levels of congestion.

IV. Related Work

This section reviews the related work. First, the topology-aware MPI process placement methods focused on the data access locality are reviewed. Then, the topology-aware load balancing techniques used in the process placement are discussed. Finally, the proposed method is compared with a memory placement method considering memory congestion on modern NUMA systems.

A. Topology-aware MPI Process Placement

Various MPI process placement methods have been proposed in the related studies. The MPIPP framework uses the execution profile to place MPI processes to different nodes of a cluster [8]. Hendrickson and Leland proposed graph-based partitioning algorithms to optimize the process mapping [26]. Zhang et al. [9] and Ma et al. [27] proposed process placement strategies for collective communication. A more recent method was proposed by Jeannot et. al. with a tree-based algorithm called Treematch [7] where they took into account the application communication pattern and the hierarchical structure of the NUMA nodes. A typical method to compute the process placement has also been proposed in their paper. Treematch achieves better performance than the previous graph-based partitioning algorithms by taking into account the application communication pattern and the qualitative information of the NUMA node topology.

A key difference between the above related work and this work is that the related work mainly focused on the data access locality. In contrast, this work focuses on MPI process placement that can reconcile the locality and memory congestion problems on modern NUMA systems. Furthermore, by using a low-level monitoring, this work considers all kinds of MPI communications in the application and is not limited to the collective communications.

B. Topology-aware Load Balancing

Several topology-aware load balancing techniques have been proposed in the related studies. Bhatele and Kale [28] present the advantages of topology-aware mapping on a torus topology for molecular dynamics applications. NucoLB [29], HwTopoLB [30], and Treematch LB [31] proposed dynamic load balancing methods that consider the CPU load characterization of computing tasks and the latencies of network links. Their methods rely on the Charm++ runtime system and incur overhead from the task migration.

In contrast, this work can balance the traffics on memory controllers by using the information about data traffics given by a static analysis of the communication pattern. In this work, the processes are assigned to the processor cores when the application is launched. Thus, the process migration is not required. Moreover, the proposed method does not rely on a specific runtime system and can be used for legacy MPI applications.

C. Congestion-aware Memory Placement on Modern NUMA Systems

Dashti et al. [10] proposed a memory placement method, called Carrefour, to reconcile the data access locality and memory congestion on modern NUMA systems. The key difference between their method and this work is that Carrefour works as a Linux kernel policy to dynamically place memory pages on nodes to avoid the congestion. It incurs overheads from the memory access sampling and the memory page replication. In contrast, this work assigns processes to processor cores when the application is launched. Therefore, the overhead is incurred by the initial profiling step, and a rerun of the profiling is only required when the communication pattern of the MPI application has changed.

V. Conclusion and Future Work

In this paper, a new algorithm called CLB, which computes an MPI process placement to reconcile locality and memory congestion on modern NUMA systems, has been proposed. The algorithm reduces memory congestion by preventing the load imbalance among nodes. The data traffics that can cause congestion are identified by applying a data clustering method to the time-series data of the MPI communications.

To evaluate the proposed method, CLB has been compared to two greedy strategies and one of the state-of-the-art locality-optimized methods. Four kernels of NPB have been tested on a modern NUMA system and a simulation environment. The experimental results clearly show that CLB consistently outperforms the other methods, and 1.6x performance improvement compared with the state-of-the-art method has been exhibited. Moreover, CLB achieves a better performance than the other methods when the congestion increases.

Our future work will focus on improving CLB for the use of a large cluster of modern NUMA systems. To optimize CLB for these clusters, the higher latencies of the network controllers will be considered. In the experiments discussed in this paper, both values of α and β are set equally. Note that the mapping result of CLB may change if the values of α and β are not equal. The impact of changing these weights on the execution time will be discussed in our future work. Another interesting issue is the effect of the proposed method on energy consumption of the NUMA systems. Finally, although this paper discusses the MPI process placement, memory congestion awareness will be important also in thread scheduling with considering thread affinities. Therefore, we will also discuss the effect of the proposed method in a large-scale system, which needs hybrid programming of MPI and multi-threading such as OpenMP. The method proposed in this paper will also be effective for the thread affinity problems. This will be further discussed in our future work.

Acknowledgments

The authors would like to thank Mr. Shoichi Hirasawa of National Institute of Informatics for his meaningful discussion.

This research is partially supported by JST CREST "An Evolutionary Approach to Construction of a Software Development Environment for Massively-Parallel Heterogeneous Systems," DFG SPPEXA ExaFSA project, and Grant-in-Aid for Scientific Research(B) #16H02822. The first author is financially supported by the Data Science Program from the Ministry of Education, Culture, Sports, Science and Technology (MEXT) of Japan.

References

[1] F. Gaud, B. Lepers, J. Funston, M. Dashti, A. Fedorova, V. Quéma, R. Lachaize, and M. Roth, "Challenges of memory management on modern numa systems," *Queue*, vol. 13, no. 8, p. 70, 2015.

[2] D. Molka, D. Hackenberg, and R. Schöne, "Main memory and cache performance of intel sandy bridge and amd bulldozer," in *Proceedings of the Workshop on Memory Systems Performance and Correctness*, ser. MSPC '14. New York, NY, USA: ACM, 2014, pp. 4:1–4:10. [Online]. Available: http://doi.acm.org/10.1145/2618128.2618129

[3] D. Ziakas, A. Baum, R. A. Maddox, and R. J. Safranek, "Intel quickpath interconnect architectural features supporting scalable system architectures," in *2010 18th IEEE Symposium on High Performance Interconnects*, Aug 2010, pp. 1–6.

[4] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, “hwloc: A generic framework for managing hardware affinities in hpc applications,” in *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, Feb 2010, pp. 180–186.

[5] M. P. I. Forum, “MPI: A Message-Passing Interface Standard,” http://www.mpi-forum.org, Sept. 2012.

[6] F. Cappello and D. Etiemble, “Mpi versus mpi+openmp on the ibm sp for the nas benchmarks,” in *Supercomputing, ACM/IEEE 2000 Conference*, Nov 2000, pp. 12–12.

[7] E. Jeannot, G. Mercier, and F. Tessier, “Process placement in multicore clusters:algorithmic issues and practical techniques,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 4, pp. 993–1002, April 2014.

[8] H. Chen, W. Chen, J. Huang, B. Robert, and H. Kuhn, “Mpipp: an automatic profile-guided parallel process placement toolset for smp clusters and multiclusters,” in *Proceedings of the 20th annual international conference on Supercomputing*. ACM, 2006, pp. 353–360.

[9] J. Zhang, J. Zhai, W. Chen, and W. Zheng, “Process mapping for mpi collective communications,” in *European Conference on Parallel Processing*. Springer, 2009, pp. 81–92.

[10] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth, “Traffic management: a holistic approach to memory placement on numa systems,” in *ACM SIGPLAN Notices*, vol. 48, no. 4. ACM, 2013, pp. 381–394.

[11] X. Liu and J. Mellor-Crummey, “A tool to analyze the performance of multithreaded programs on numa architectures,” *ACM Sigplan Notices*, vol. 49, no. 8, pp. 259–272, 2014.

[12] C. Ma, Y. M. Teo, V. March, N. Xiong, I. R. Pop, Y. X. He, and S. See, “An approach for matching communication patterns in parallel applications,” in *2009 IEEE International Symposium on Parallel Distributed Processing*, May 2009, pp. 1–12.

[13] A. Faraj and X. Yuan, “Communication characteristics in the nas parallel benchmarks.” in *IASTED PDCS*, 2002, pp. 724–729.

[14] I. Lee, “Characterizing communication patterns of nas-mpi benchmark programs,” in *Southeastcon, 2009. SOUTHEASTCON’09. IEEE*. IEEE, 2009, pp. 158–163.

[15] Intel xeon platinum 8180 processor. https://ark.intel.com/, visited on October 12, 2017.

[16] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine *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*. Springer, 2004, pp. 97–104.

[17] G. Bosilca, C. Foyer, E. Jeannot, G. Mercier, and G. Papauré, “Online dynamic monitoring of mpi communications,” in *European Conference on Parallel Processing*. Springer, 2017, pp. 49–62.

[18] C. Elkan, “Using the triangle inequality to accelerate k-means,” in *ICML*, vol. 3, 2003, pp. 147–153.

[19] F. Declerq, “Choropleth map accuracy and the number of class intervals,” in *Proceedings of the 17th Conference and the 10th General Assembly of the International Cartographic Association*, vol. 1. Institute Cartogràfic de Catalunya Barcelona, 1995, pp. 918–22.

[20] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber *et al.*, “The nas parallel benchmarks,” *The International Journal of Supercomputing Applications*, vol. 5, no. 3, pp. 63–73, 1991.

[21] (2015) Perf. linux profiling with performance counters. [Online]. Available: https://perf.wiki.kernel.org/

[22] D. L. Hill, D. Bachand, S. Bilgin, R. Greiner, P. Hammarlund, T. Huff, S. Kulick, and R. Safranek, “The uncore: A modular approach to feeding the high-performance cores.” *Intel Technology Journal*, vol. 14, no. 3, 2010.

[23] K. Fujiwara and H. Casanova, “Speed and accuracy of network simulation in the simgrid framework,” in *Proceedings of the 2nd international conference on Performance evaluation methodologies and tools*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2007, p. 12.

[24] A. Degomme, A. Legrand, G. Markomanolis, M. Quinson, M. Stillwell, and F. Suter, “Simulating mpi applications: the smpi approach,” *IEEE Transactions on Parallel and Distributed Systems*, 2017.

[25] V. Viswanathan *et al.*, “Intel memory latency checker v3.0,” *URL: https://software.intel.com/en-us/articles/intelr-memory-latency-checker*, Mar. 2016.

[26] B. Hendrickson and R. Leland, “The chaco users guide: Version 2.0,” Technical Report SAND95-2344, Sandia National Laboratories, Tech. Rep., 1995.

[27] T. Ma, T. Herault, G. Bosilca, and J. J. Dongarra, “Process distance-aware adaptive mpi collective communications,” in *2011 IEEE International Conference on Cluster Computing*, Sept 2011, pp. 196–204.

[28] A. Bhatelé, L. V. Kalé, and S. Kumar, “Dynamic topology aware load balancing algorithms for molecular dynamics applications,” in *Proceedings of the 23rd international conference on Supercomputing*. ACM, 2009, pp. 110–116.

[29] L. L. Pilla, C. P. Ribeiro, D. Cordeiro, C. Mei, A. Bhatele, P. O. A. Navaux, F. Broquedis, J. F. Mhaut, and L. V. Kale, “A hierarchical approach for load balancing on parallel multi-core systems,” in *2012 41st International Conference on Parallel Processing*, Sept 2012, pp. 118–127.

[30] L. L. Pilla, P. O. A. Navaux, C. P. Ribeiro, P. Coucheney, F. Broquedis, B. Gaujal, and J. F. Mhaut, “Asymptotically optimal load balancing for hierarchical multi-core systems,” in *2012 IEEE 18th International Conference on Parallel and Distributed Systems*, Dec 2012, pp. 236–243.

[31] E. Jeannot, E. Meneses, G. Mercier, F. Tessier, and G. Zheng, “Communication and topology-aware load balancing in charm++ with treematch,” in *2013 IEEE International Conference on Cluster Computing (CLUSTER)*, Sept 2013, pp. 1–8.