Task Mapping for Coordinating Locality and Memory Congestion on NUMA Systems

TOHOKU UNIVERSITY
Graduate School of Information Sciences

Task Mapping for Coordinating Locality and Memory Congestion on NUMA Systems

（NUMA システムにおける局所性とメモリ負荷集中を考慮した
タスクマッピングに関する研究）

A dissertation submitted for the degree of Doctor of Philosophy
(Information Sciences)

Department of Computer and Mathematical Sciences

by

Mulya Agung

January 14, 2020

Task Mapping for Coordinating Locality and Memory Congestion on NUMA Systems

Mulya Agung

Abstract

Recent multicore systems have complex memory hierarchy with multiple cache levels and memory controllers, which leads to Non-Uniform Memory Access (NUMA) characteristics. In such systems, the mapping of parallel tasks to processor cores, called task mapping, has a significant impact on performance and energy efficiency because it affects the cost of communication among tasks. Most of existing approaches in efficient task mapping has focused only on improving the locality of memory accesses. Improving the locality is important because it will reduce the remote access penalty and congestion on interconnects. However, in recent NUMA systems, the high number of cores can cause the memory congestion problem, which could degrade performance more severely than the locality problem. As the number of cores increases, the number of concurrent communications will also increase. Maximizing the locality will increase the memory congestion because it will concentrate the concurrent communications on particular NUMA nodes. Therefore, considering only the locality is not sufficient to achieve a scalable performance on NUMA systems, and it is necessary to consider both the locality and the memory congestion to increase performance and energy efficiency of NUMA systems.

To optimize the task mapping, it is mandatory to analyze the communication behaviors of the parallel applications. Different applications may have different communication behaviors, which determine the impacts of task mapping on the performance and energy consumption of the applications. Most of the existing methods rely on offline profiling and analysis to trace the communications and analyze the communication behaviors prior to the execution of the application. On the other hand, some existing online methods analyze the communication behaviors and perform the task mapping during the execution of the application. Unlike offline methods, these methods can dynamically change the task mapping to adapt to changes in the communication behaviors of the application. However, the main drawback of the online methods is that they introduce overhead to the execution time of the application. In contrast, offline methods obtain the task mapping prior to the execution, and thus they do not impose overhead to the execution time of the application.

This dissertation aims to establish task mapping methods that consider both locality and memory congestion at the same time to achieve high performance and energy efficiency of NUMA systems. This dissertation contributes to task mapping from the following three aspects. First, this dissertation introduces a method to analyze and characterize the spatial and temporal communication behaviors of parallel applications. The method consists of two techniques to obtain the communication events among tasks, a data clustering method to identify the concurrent communications that cause the memory congestion, and a set of metrics to characterize the communication behaviors. The proposed metrics are important because of three reasons. First, they are necessary to evaluate the impacts of a task mapping method on the performance of a particular application. Second, the metrics can be used to evaluate the suitability of task mapping methods to a particular application. Third, by using the metrics, a parallel application is classified into one of three categories based on which of offline and online methods can increase the performance of an application.

The second contribution of this dissertation is to propose an offline task mapping method to address the locality and memory congestion problems. The proposed method works offline prior to the execution of a parallel application. The proposed method consists of three steps. First, it gathers the NUMA node topology of the target system. Then, it analyzes the spatial and temporal communication behaviors of the target application. Finally, it computes a task mapping that can coordinate the locality and the memory congestion. A mapping algorithm is proposed to calculate the mapping by using information about the NUMA node topology and the communication behaviors. The mapping result is then used for the execution of the target application. An extensive evaluation with parallel applications from three benchmark suites has been conducted to show that the proposed method can achieve substantial performance improvements in comparison with a dynamic mapping method and five static mapping methods, including the current state-of-the-art locality-based method.

The third contribution of this dissertation is to present an online task mapping method for coordinating locality and memory congestion. Unlike the proposed offline method, this method works online during the execution of the application. It does not require any information about the communication behaviors prior to the execution. During the execution, it dynamically analyzes the communication behavior and performs task mapping to adapt to changes in the communication behavior. Since the proposed method works at runtime, it introduces overhead to the execution of the application. The overhead is caused by repeatedly updating the mapping and thereby migrating tasks among cores. In this dissertation, the online method employs two mechanisms to reduce the overhead. The first mechanism is to dynamically adjust the mapping interval. The mapping interval is adjusted to limit the frequency of updating the mapping and to adapt the changes in the communication behavior. The second mechanism is a mapping algorithm that considers the current mapping to calculate the next mapping. The mapping algorithm prevents unnecessary task migrations by giving a higher priority to tasks that have a higher amount of communication to be mapped to the same NUMA node of the current mapping. An extensive evaluation has been conducted with a set of parallel benchmarks on two NUMA systems, and two biomolecular applications on a larger NUMA system. The evaluation results show that the proposed method can achieve performance and energy consumption close to the best static method with a low overhead.

The conclusions of this dissertation are as follows: task mapping coordinating the locality and memory congestion is crucial to achieving the scalable performance and energy efficiency in NUMA systems, analyzing the communication behaviors of parallel applications is a mandatory step for task mapping, and both offline and online task mapping are necessary to improve the performance and energy efficiency on NUMA systems.

Table of Contents

List of Figures

List of Tables

List of Acronyms

AMD	Advanced Micro Devices
DDR	Double Data Rate
DRAM	Dynamic Random-Access Memory
IMC	Integrated Memory Controller
KNL	Knights Landing
MCDRAM	Multi-Channel Dynamic Random-Access Memory
MPI	Message Passing Interface
NUMA	Non-Uniform Memory Access
NPB	NAS Parallel Benchmarks
PARSEC	The Princeton Application Repository for Shared-Memory Computers
QPI	QuickPath Interconnect

Chapter 1

Introduction

1.1 Background

Since the early 2000s, the number of cores in a processor has been increasing. Increases in the number of cores are accompanied by more complex memory hierarchies, consisting of several private and shared cache levels, and multiple memory controllers that form the base for *Non-Uniform Memory Access (NUMA) architectures.* Each processor consists of one or several sets of processor cores that are physically associated with one or more memory controllers and memory devices. Such a set of processor cores is referred to as a NUMA node [1, 2]. The main advantage of this architecture is that it provides a higher memory bandwidth and less latency because multiple nodes can be assembled into a single NUMA system [3, 4]. However, the main drawback is that the performance of memory accesses depends on the location of the data [5–7]. The NUMA nodes are generally connected by high-speed interconnect links such as Intel QuickPath Interconnect (QPI) [8] and AMD HyperTransport [9]. However, accessing the data that is located on the memory devices of remote NUMA nodes causes a performance penalty because it still needs a longer latency than accessing the data of the local memory devices [10]. Thus, the cost of remote memory access is higher than that of the local memory access.

The most important performance optimization for NUMA systems considered until now is to increase the locality of memory accesses. Various studies have focused on

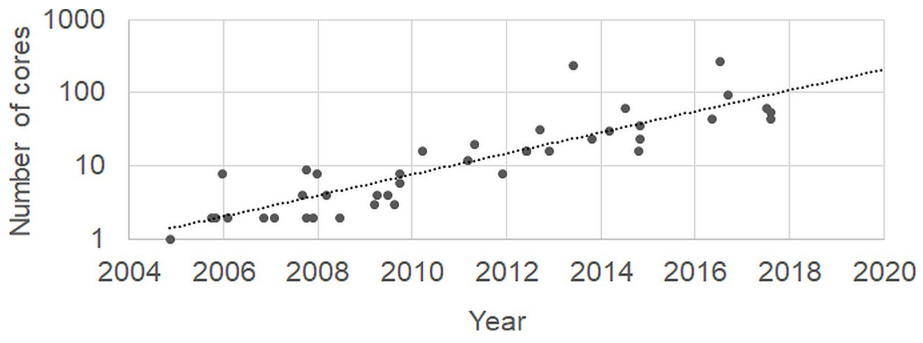


Figure 1.1: Number of logical cores per processor.

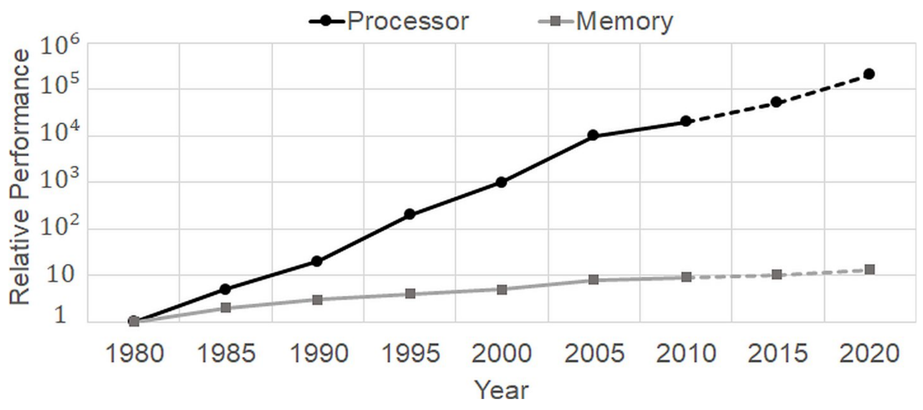


Figure 1.2: Performance gap between processor and memory.

increasing the locality in NUMA systems to improve the performance of parallel applications on the systems [3, 11, 12]. Improving the locality is important because it will reduce the remote access penalty. However, when the number of processor cores in a system increases, the impact of the locality on the performance becomes lower. The higher number of processor cores in the recent systems can induce a larger number of accesses to memory devices, causing congestion on the shared cache memories and memory controllers. This congestion is referred to as the *memory congestion* [1,13,14]. Recent studies on NUMA systems have shown that the memory congestion problem could degrade performance more severely than the locality problem because heavy congestion on memory controllers could cause long latencies [4, 13]. Furthermore, maximizing the locality can degrade performance because it potentially increases the memory congestion [1, 15].

As shown in Figure 1.1, every two years, the number of cores in a processor has been doubling [16]. The number of parallel tasks that can be executed by a processor at the

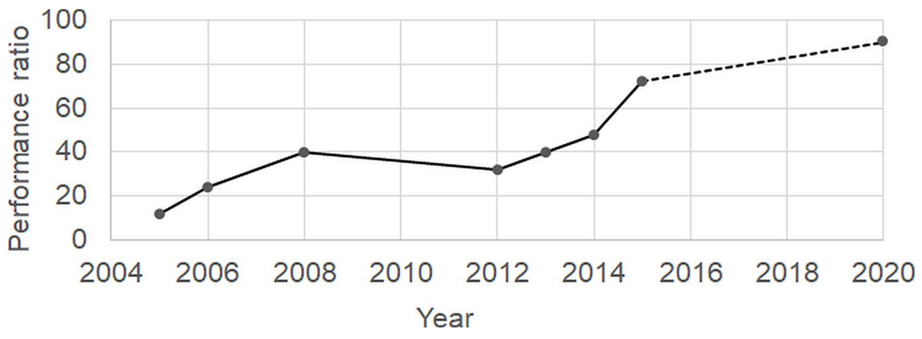


Figure 1.3: Ratio of CPU performance to memory bandwidth.

same time increases with the number of cores. However, as shown in Figure 1.2, the performance of a processor increased more rapidly than the performance of memory [17]. Over the last 20 years, the performance gap between processor and memory has been steadily increasing. The dashed curve shows the extrapolated values of the performance. The differences between the number of cores and memory bandwidth of the system is one main factor that contributes to the performance gap. As the number of cores increases, the risk of memory congestion also becomes higher. One way to alleviate the memory congestion is by increasing the memory bandwidth of the system. However, due to the physical limits, the memory bandwidth of a system cannot be increased beyond a certain point [18]. As an example of the memory bandwidth limitation, Figure 1.3 shows the ratio of the performance of processor to memory bandwidth in AMD and Intel-based NUMA systems [16]. The dashed curve shows the extrapolated values of the ratio. Every three to four years, the ratio has been doubling. As the memory bandwidth becomes the limiting factor for the performance, it is important to manage the memory congestion in NUMA systems efficiently.

A parallel application consists of multiple tasks, each of which is executed on a processor core as a thread or a process. On NUMA systems, the performance of applications can be improved by placing tasks to processor cores according to a policy that can consider various objectives. The task placement is achieved by mapping the tasks to processor cores, which is called *task mapping* [19]. On NUMA systems, task mapping can have a significant impact on the performance and energy consumption of the systems because the mapping affects the cost of the communications among tasks. As the cost of communication significantly affects the performance on NUMA systems, exploiting the communication behavior to map tasks to processor cores is important to increase performance and save energy. Such task mapping methods are called *communication-aware task mapping* [2, 20, 21].

To optimize the task mapping of a parallel application, analyzing the communication behavior of the application is necessary because different applications can have different communication behaviors. Moreover, on a NUMA system, the impacts of task mapping on performance and energy consumption of an application may vary depending on the communication behavior the application. In an application, a task does not necessarily need to communicate with all the other tasks, and the time and amount of data exchanged among tasks may vary. The diversity of the number of communications and amount of data exchanged among tasks is referred to as *spatial communication behavior*, while the changes of the spatial behavior over time is referred to as *temporal communication behavior* [22, 23].

Considering the spatial communication behavior of an application is necessary to reduce the total cost of communication or load imbalance among the NUMA nodes. However, to effectively reduce the memory congestion, it is also necessary to consider the temporal communication behavior of the application because memory congestion only occurs when multiple tasks are running on different processor cores access a particular node at the same time. Therefore, analyzing both the spatial and temporal communication behaviors is a mandatory step to optimize the task mapping on NUMA systems.

Conventional approaches to efficient task mapping have focused only on improving the locality by mapping tasks, which frequently communicate with each other, to processor cores that are closer to each other in the memory hierarchy [2, 11, 12, 24]. However, maximizing the locality does not always improve performance due to the memory congestion. Moreover, some approaches rely on offline profiling to analyze the communication behavior [11, 12, 25, 26], which cause a high overhead and may result in incorrect analysis if the communication behavior of the application changes during the execution. Therefore, all of these approaches are not sufficient to improve the performance and energy efficiency of

NUMA systems.

Some approaches use online mechanisms to overcome the limitations of offline profiling and analysis [2, 13, 14]. However, the offline and online approaches have advantages and disadvantages. The offline approach obtains task mapping prior to the execution, and the mapping is applied when the application is launched. Thus, it does not incur any overhead to the execution of the application. However, the overhead of this method is incurred by the offline profiling and analysis. In contrast, the online approach analyzes the communication behaviors and calculates the mapping during the execution of the application, and thus it introduces overhead to the execution time of the application. Moreover, the online approach will migrate tasks if the mapping changes during the execution, and the task migration can significantly affect performance and energy consumption [2, 27].

The goal of this dissertation is to improve the performance and energy efficiency of parallel applications on NUMA systems by using task mapping to address the locality and the memory congestion problems. Because of the advantages and disadvantages of offline and online approaches, this dissertation discusses offline and online task mapping to coordinate locality and memory congestion. To obtain the information of communication behaviors, this dissertation first proposes a method to analyze spatial and temporal communication behaviors of parallel applications, which includes a set of metrics to describe the communication behaviors that affect the locality and memory congestion. Then, the task mapping methods use the proposed analysis method to determine task mapping that can coordinate locality and memory congestion. The first mapping method analyzes the communication behaviors and computes the task mapping offline prior to the execution of the application. On the other hand, the second mapping method does not need offline profiling and analysis, and it analyzes the communication behaviors and performs the task mapping online during the execution of the application. Comparisons between the two methods are also discussed in this dissertation.

1.2 Research Problem and Objective

This dissertation focuses on task mapping to address the locality and memory congestion problems in NUMA systems that stem from the spatial and temporal communication behaviors of a parallel application. To coordinate locality and memory congestion, analyzing both spatial and temporal communication behaviors of the application is necessary because the timing of the communication affects the memory congestion. Thus, this dissertation proposes a method to analyze and characterize the spatial and temporal communication behaviors of parallel applications. This method consists of a toolchain to analyze the communication behaviors of the application, and a set of metrics to characterize the communication behaviors that affect the locality and the memory congestion.

After the communication behaviors of an application are obtained, a task mapping method can exploit the communication behaviors to optimize the task mapping for the application. Most of the related studies in task mapping focus only on improving the locality to reduce the overall communication cost. However, considering only the locality is not sufficient due to the memory congestion problem. Thus, this dissertation discusses task mapping to tackle the locality and memory congestion problems on NUMA systems. For the discussion, it first proposes an offline method that uses the information about the NUMA node topology and the communication behaviors to optimize the task mapping.

To analyze the communication behaviors of an application, most of the existing task mapping methods rely on offline profiling and analysis, which traces the communication events and analyzes the communication behaviors prior to the execution of the application. However, the offline profiling and analysis are potentially time-consuming and are not applicable if the application changes its behavior between executions. Thus, this dissertation also discusses online task mapping to address the locality and memory congestion problems. An online mapping method is proposed to coordinate both problems without requiring offline profiling and analysis.

1.3 Organization of the Dissertation

The rest of the dissertation is organized as follows. Chapter 2 discusses a method for analyzing and characterizing the communication behaviors of parallel applications. Chapter 3 discusses offline task mapping for coordinating locality and memory congestion. Chapter 4 discusses online task mapping for coordinating locality and memory congestion. Chapter 5 summarizes the conclusions of this dissertation.

Chapter 2

Analyzing the Communication Behaviors of Parallel Applications

2.1 Introduction

In a parallel application, the application workload is partitioned into tasks. During the execution of the application, tasks may communicate with each other to share data among them. As discussed in Chapter 1, analyzing the spatial and temporal communication behaviors of parallel applications is a necessary step in task mapping for coordinating locality and memory congestion. Analyzing the communication behaviors is necessary because different applications may have different communication behaviors, and the impacts of task mapping on a particular application depend on the communication behaviors of the application [21, 28].

During the execution of a parallel application, a task does not necessarily need to communicate with all the other tasks, and the time and amount of communication events among tasks may vary. The event that each time two tasks communicate with each other is called *a communication event*. The spatial communication behavior is determined by the diversity of the amount of communication, which can be represented by the number or volume of communication events. By exploiting this communication behavior, a task mapping method that aims to improve locality can reduce the overall communication cost by mapping tasks that have larger amounts of communication to processor cores within the same NUMA node. In contrast, a method that focuses on improving the balance of communication will map such tasks to cores of different NUMA nodes [29].

The temporal communication behavior is determined by the times that communication events occur during the execution [23, 30]. When multiple communication events among different tasks are in progress, they are referred to as *concurrent communications*. In conjunction with spatial communication behavior, temporal communication behavior can affect the memory congestion because the risk of memory congestion increases with a larger amount of concurrent communications. As spatial and temporal communication behaviors affect the locality and memory congestion, the impacts of task mapping on performance and energy consumption of an application are determined by the communication behaviors of the application.

In order to analyze the spatial and temporal communication behaviors of an application, there are two challenges that need to be addressed. The first is how to obtain the communication events, including information about the amount and the timing of communication. The other challenge is how to analyze and characterize the communication behaviors from the obtained communication events. The characterization is required for describing the communication behaviors that affect the locality and memory congestion. To address these two challenges, this chapter proposes a method, called *decongested locality profiler (DeLocProf)*, to analyze and characterize the spatial and temporal communication behaviors of parallel applications. The method consists of two techniques to obtain communication events from parallel applications, a data clustering method to analyze the spatial and temporal communication behaviors, and a set of metrics to characterize the communication behaviors.

This chapter is organized as follows. First, Section 2.2 briefly reviews related work that has focused on analyzing the communication behaviors of parallel applications. Then, DeLocProf is described in Section 2.3. Section 2.4 presents the experimental evaluation, and discusses the evaluation results with the proposed metrics. Finally, the conclusions of this chapter are summarized in Section 2.5.

2.2 Related Work

This section reviews the related studies that have focused on analyzing the communication behaviors of parallel applications. It first reviews existing methods for analyzing the communication behaviors of MPI applications, and then reviews existing methods for analyzing the communication behaviors of multithreaded applications. Finally, it reviews existing tools for analyzing parallel applications on NUMA systems.

2.2.1 Communication Behaviour Analysis in MPI

Most of the related studies in analyzing communication behaviors focused on applications that use message passing frameworks, such as Message Passing Interface (MPI) [31]. Communication behaviors of different MPI applications and benchmarks have been analyzed by [32–34] A methodology to analyze communication behavior is presented in [22, 35], where communication is described with spatial and temporal components. The proposed method of this chapter uses similar components to analyze the communication behaviors. However, in this chapter, the spatial and temporal components are applied to optimize the task mapping that can improve locality and reduce memory congestion. Moreover, the proposed method focuses on analyzing communication not only in MPI applications but also in multithreaded applications. The difference between communication in MPI and multithreaded applications is discussed in Section 2.3.2.

2.2.2 Communication Behavior Analysis in Multithreaded Applications

Some related studies have focused on characterizing the communication behaviors of multithreaded applications. In the related work [21,26], communication is analyzed by tracing memory accesses among threads, and the related work [21] had shown that the page usage behavior could be analyzed from the memory accesses. However, all the related studies consider only the spatial communication behavior to improve the locality, and do not account the memory congestion issue caused by spatial and temporal communication behaviors of the applications. A related work [23] has focused on analyzing spatial and temporal communication behaviors of PARSEC benchmarks. However, this related work is limited to a set of benchmark applications and does not consider the impacts of the communication behaviors on task mapping.

2.2.3 Memory Access Analysis in NUMA Systems

So far, several tools had been proposed to analyze parallel applications on NUMA systems, such as Memphis [36], Memprof [37] and MemAxes [38]. These methods analyze the memory access behavior by tracing the memory accesses among threads in parallel applications. However, these tools focus on memory access analysis, and do not take into account the communication among tasks of the applications.

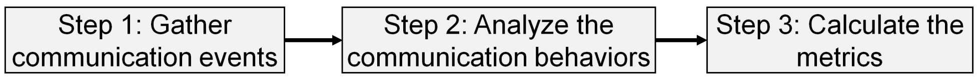

Step 1: Gather communication events
Step 2: Analyze the communication behaviors
Step 3: Calculate the metrics

Figure 2.1: The procedure of DeLocProf.

2.3 A Method to Analyze and Characterize the Spatial and Temporal Communication Behaviors

This section presents DeLocProf, which can analyze and characterize the spatial and temporal communication behaviors of parallel applications. Figure 2.1 shows the procedure of DeLocProf, which consists of three steps:

Step 1 Gather the communication events of the target application.

Step 2 Analyze the spatial and temporal communication behaviors.

Step 3 Calculate the metrics for characterizing the communication behaviors.

For Step 1, two techniques are proposed to obtain the communication events of a parallel application based on distributed-memory parallel processing and shared-memory parallel processing methods. Two different techniques are proposed because communication in a parallel processing method can be explicit or implicit, and a detection mechanism is required to detect implicit communication from memory accesses among tasks. For Step 2, this section proposes a data clustering method to detect concurrent communications from communication events. For Step 3, a set of metrics is introduced to characterize the communication behaviors that affect locality and memory congestion, and a metric to describe static and dynamic communication behaviors.

2.3.1 Gathering Communication Events of Applications Based on Distributed-memory Parallel Processing

In parallel applications based on distributed-memory parallel processing, such as MPI, a task is instantiated by a process, and each process has a unique identifier called a process ID. Thus, in MPI, process mapping is also referred to as task mapping [39]. In

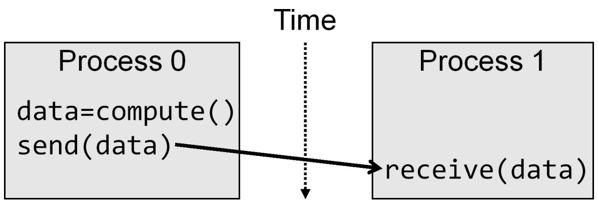


Figure 2.2: Explicit communication in MPI.

MPI, communication is explicit, and is performed by sending and receiving messages. Figure 2.2 shows an explicit communication between two MPI processes. The process that sends the message is called a sender, and the process that receives the message is called a receiver. For each communication, there is a pair of sender and receiver processes. Thus, a communication event can be defined as one message with its corresponding pair of sender and receiver. This pair is also referred to as a *task pair*.

In MPI, a process can communicate with other processes by using point-to-point (P2P) operations and collective operations. In P2P operations, a process sends messages to another process by explicitly specifying the ID of the receiver. On the other hand, rather than explicitly sending and receiving such messages, a collective operation involves communications among all processes in a communicator. In internal MPI implementation, a collective operation is typically implemented using multiple P2P operations [40,41]. Thus, in that case, each collective operation can be decomposed into P2P operations.

To trace the communication events, the target application is preliminarily run while monitoring the MPI communication events. For the monitoring purpose, a monitoring tool is implemented as a component of the monitoring framework that has been proposed in [42]. This framework is built on the top of the P2P management layer (PML) of the Open MPI stack [43]. Since the PML can monitor P2P operations organizing a collective communication, the communication events can be traced in both cases of P2P and collective communications. Furthermore, PML monitors not only MPI_Send-MPI_-Recv but also MPI_Isend-MPI_Irecv operations. Thus, the monitoring tool also traces both blocking and non-blocking communications. The tool generates a time-series dataset of communication events by recording the IDs of the sender and receiver, the data size, and the timestamp of each event. The timestamp is recorded to provide information

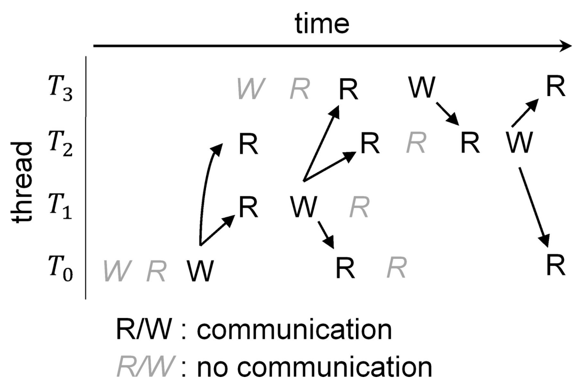


Figure 2.3: Memory accesses of different threads can be seen as their communications.

about the timing of the communication.

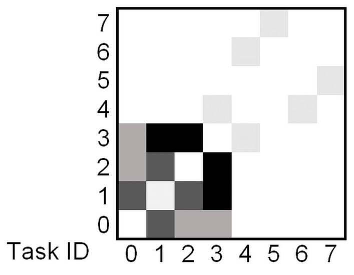
2.3.2 Gathering Communication Events of Applications Based on Shared-memory Parallel Processing

In parallel applications based on shared-memory parallel processing, such as OpenMP [44] and Pthreads [45], a task is instantiated by a thread of execution, and each thread has a unique identifier called thread ID. Thus, in shared-memory parallel processing, thread mapping is also referred to as task mapping [29]. The communication among threads is performed implicitly by accessing the shared memory space. By tracing accesses to memory addresses at the cache line granularity, a communication event can be defined as two consecutive write and read memory accesses from different threads to the same cache line. These two memory accesses are referred to as *communicating memory accesses* [23]. Figure 2.3 shows communicating and non-communicating memory accesses for one cache line. The communicating memory accesses are shown in black, while non-communicating memory accesses are shown in gray. Two different threads that perform the communicating memory accesses are called a pair of threads, and this pair is also referred to as a task pair.

To obtain the communication events, the target application is preliminarily run while monitoring the memory accesses performed by the threads of the application and detecting the communication events from the memory accesses. For the monitoring and

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
Task ID	0	1	2	3	4	5	6	7

(a) An example of communication matrix.



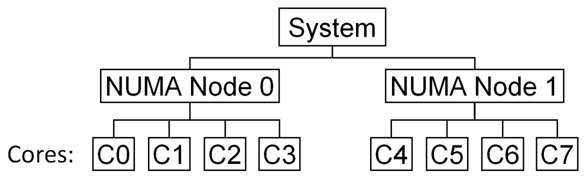
(b) Visualization of the communication matrix.

Figure 2.4: The representations of undirected communication graph for a parallel application that consists of eight tasks.

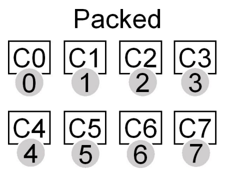
detection purpose, a tracing tool is implemented based on a dynamic binary instrumentation framework, called pin [46]. This tool detects communication from memory accesses of the threads at a granularity of cache line, which is a 64 byte-wide memory block. It generates a time-series dataset of communication events by recording the IDs of the pair of threads, the timestamp, and the data size of memory access of each event.

2.3.3 Analyzing Spatial and Temporal Communication Behaviors from Communication Events

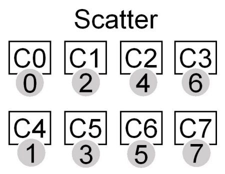
As discussed previously in Section 2.1, the spatial communication behavior is determined from the diversity of the amount of communication among tasks of the parallel application. To analyze this behavior, an undirected communication graph is built from the dataset of communication events, where nodes represent tasks and edges represent the amount of communication between each pair of tasks. The communication graph is represented as a matrix, which is called *a communication matrix* [2,23,26]. An example of the communication matrix is shown in Figure 2.4(a). Each cell (x, y) of the matrix contains the data size of communication events between a pair of tasks x and y, which represents the amount of communication between the pair. Since the communication graph is undirected, the communication matrix is symmetric. In the communication behavior shown by the matrix, most communication events are performed by the neighboring tasks, which are tasks that have IDs close to each other, and tasks 0 to 3 have a larger amount of communication



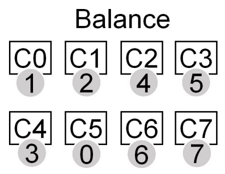
(a) An example of the system with eight cores.



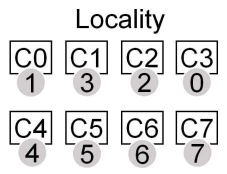
(b) The Packed mapping.



(c) The Scatter mapping.



(d) The Balance mapping.



(e) The Locality mapping.

Figure 2.5: Examples of a NUMA system and the outputs of Packed, Scatter, Balance, and Locality mappings.

than the other processes.

To analyze the spatial communication behavior, the communication matrix is normalized to its maximum value to limit the range of values, such as between 0 and 1. By normalizing the matrix, two communication matrices from different applications can be compared to each other. Then, to visualize the communication behavior, the normalized matrix is represented as a heat map. Figure 2.4(b) shows the visualization of the previous communication matrix shown in Figure 2.4(a). In the visualization, the darker cells indicate larger amounts of communication. With the spatial communication behavior represented by the communication matrix, a *locality-based* task mapping method can improve the locality by mapping tasks that have larger amounts of communication to processor cores of the same NUMA node, while a *balance-based* mapping method can improve the balance of communication by distributing tasks that have substantial amounts of communication to different NUMA nodes.

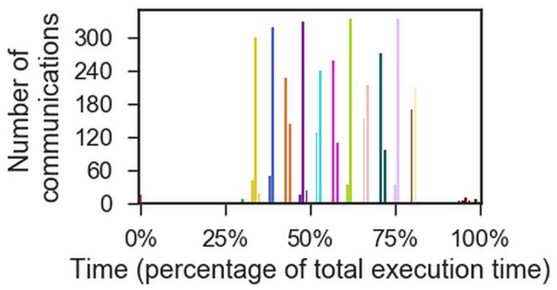
To illustrate how a task mapping method uses the communication behavior to determine the mapping, Figure 2.5 depicts examples of a system with eight cores and the outputs of four mapping methods: Packed, Scatter, Balance and Locality. In the figure, the white boxes represent the processor cores, while the gray circles represent the tasks. Packed and Scatter do not consider the communication behavior, while Balance and Locality take into account the communication behavior represented by the communication matrix shown in Figure 2.4. Packed maps the neighboring tasks to the same NUMA node, while Scatter maps the neighboring tasks to different NUMA nodes. These two methods are available in some MPI runtime environments [43, 47] and OpenMP libraries [48]. In the related work [49], Scatter corresponded to the Socket-span mapping. In the case where neighboring tasks have a larger amount of communication than the other tasks, Packed will increase the locality of communication, while Scatter will reduce the communication load imbalance among the NUMA nodes. In contrast to these methods, Balance and Locality use the communication matrix to determine the mapping. Balance maps the tasks that have the largest amount of communication to different NUMA nodes, while Locality maps the tasks that have the largest amount of communication to the same NUMA node.

Among the communication events of a parallel application, only the concurrent communications will affect memory congestion. Thus, in this dissertation, the analysis of temporal communication behavior focuses on identifying the concurrent communications from the communication events. For the analysis purpose, a data clustering method is proposed to identify concurrent communications from a time series dataset of communication events. The method is based on a weighted k-means clustering method [50] and uses the numbers of communication events as the weights for the clustering. The numbers of communication events are used as the weights because the concurrent communications are determined by the number of communication events that occur at times close to each other.

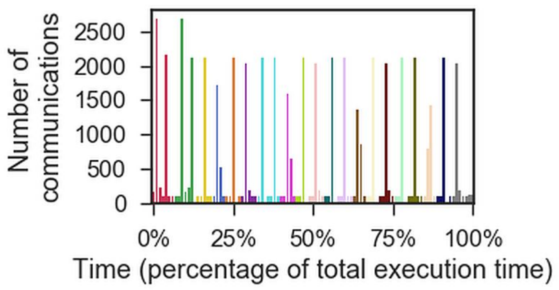
Given a set of timestamps of communication events $\{t_1, t_2, ..., t_n\}$ and a set of clusters $\{C_1, C_2, ..., C_k\}$, the clustering method aims to minimize the objective function j defined by Equation (2.1), where k is the number of clusters, μ_i is the mean of timestamps in a cluster, and $Ncomm_t$ is the number of communication events at timestamp t.

$$j = \sum_{i=1}^{k} \sum_{t \in C_i} Ncomm_t \|t - \mu_i\|^2. \tag{2.1}$$

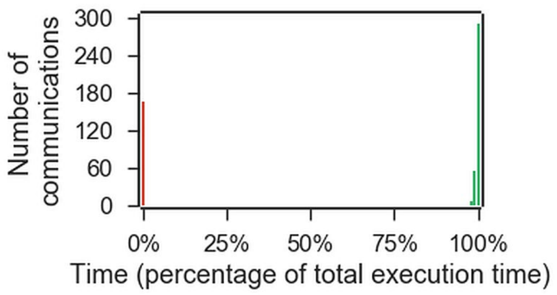
The clustering method needs to predefine a parameter, k, to specify the number of



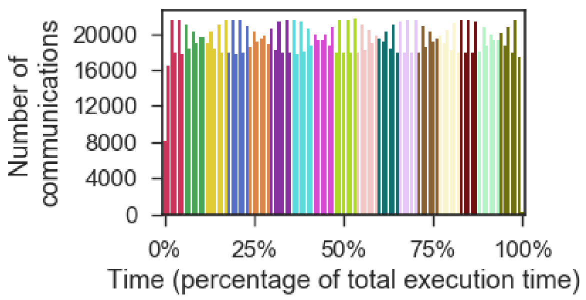
(a) IS-MPI.



(b) MG-MPI.



(c) EP-MPI.



(d) LU-MPI.

Figure 2.6: Temporal communication behaviors of four NPB-MPI applications.

clusters. However, the actual number of clusters is generally unknown in advance, even though the parameter certainly affects the clustering results. This parameter affects the accuracy of detecting the concurrent communications. If the value of k is too small, two communication events may be falsely identified as concurrent communications. Thus, it is necessary to estimate an optimal number of clusters for the clustering. Moreover, the execution time of k-means increases with the number of communication events. Since the number of communication events in long-running parallel applications can be very large, reducing the execution time of the clustering process also becomes necessary. For these two reasons, the proposed method selects k by using the Bayesian information criterion (BIC) [51]. This criterion is used because it has been empirically used not only to find an optimal value of k but also to accelerate the clustering process for large datasets [52]. The method iteratively tests the k parameters, and the acceleration is achieved by reducing the search space of k and reusing statistics of previous iterations.

Figure 2.6 shows examples of clustering results, which are obtained by analyzing IS-

MPI, MG-MPI, EP-MPI, and LU-MPI applications of the NAS Parallel Benchmarks (NPB-MPI) [53], with the class C input size. The x-axis and y-axis show the time elapsed during the execution, and the number of communication events, respectively. The figure shows that the number of communication events changes during the execution of each application. In IS-MPI, the communication events mostly occur in between 30% and 81% of the total execution time, while the communication events of MG-MPI and LU-MPI are distributed over the the execution time. In EP-MPI, the communications happen only at the beginning and the end of the execution. It is because EP-MPI is an embarrassingly parallel application, and the communications are only required for bcast and allreduce operations, which are executed at the beginning and the end of the execution.

The colors in Figure 2.6 show the clustering results of the four NPB applications, in which different colors represent different clusters. The figure shows that time periods are likely to be grouped in the same cluster when they are close to each other and have a large number of communication events. In IS-MPI, the clusters with the highest number of communication events are shown in the middle of the execution, while in MG-MPI, the clusters with the highest number of communication events are shown at the beginning of the execution. In EP-MPI, there are only two clusters, and the cluster shown at the end of the execution has the highest number of communication events. On the other hand, in LU-MPI, the number of communication events is distributed over all of the clusters. As also shown in the figure, a cluster can have a high number of communication events. If all the communication events of this cluster occur in the same NUMA node, the memory congestion will likely happen because all concurrent communications will access data from the same node. By using the information about concurrent communications, task mapping can reduce memory congestion by distributing different tasks that perform concurrent communications to processor cores of different NUMA nodes.

2.3.4 Communication Behaviors that Affect the Locality and the Memory Congestion

As discussed in Section 2.1, performance and energy consumption improvements according to a specific task mapping method depend on the communication behaviors of the target application. In this section, five metrics are proposed to describe the communication behaviors that affect the locality of communication and the memory congestion. The first two metrics are *communication load* and *communication-to-memory ratio.* These two metrics are used to describe the communication behaviors that can benefit from communication-aware task mapping. Two other metrics are called *communication concurrency* and *DRAM-to-memory ratio.* In conjunction with the previous metrics, these two metrics are proposed to describe the communication behaviors that can benefit from task mapping to reduce memory congestion. The last metric, called *communication locality*, is proposed to describe the communication behavior that can benefit from locality-based task mapping.

An improvement according to a specific communication-aware task mapping method depends on how much tasks are communicating. The improvement is expected to be higher for parallel applications, in which the total amount of transferred data is larger. The load of communication is described by using the Lcomm metric, which is defined as the total amount of communication by all tasks. Lcomm is calculated by

$$L\text{comm} = \sum_{i=1}^{T}\sum_{j=1}^{T} S\text{comm}[i][j], \tag{2.2}$$

where T is the total number of tasks and Scomm$[i][j]$ is the amount of communication between a pair of tasks i and j. The load of communication itself is not sufficient to evaluate if an application will gain performance benefit from communication-aware task mapping. If the number of non-communicating accesses is much higher than the number of communicating accesses, a communication-aware task mapping method might not significantly affect the overall memory access behavior. For this reason, the communication-to-memory ratio metric $CommR$ is defined as the ratio of load of communication to the total size of memory accesses of the tasks. $CommR$ is calculated by

$$CommR = \frac{L\text{comm}}{\sum_{i=1}^{T} MemV[i]}, \tag{2.3}$$

where $MemV[i]$ is the size of data accessed by task i during the whole execution. The expected performance gains are higher for parallel applications that have higher values of Lcomm and $CommR$.

For communication-aware task mapping methods that aim to reduce the memory congestion, it is necessary to evaluate how the communication among tasks affects the memory congestion. Even if the load of communication is high, the task mapping method might not give a performance benefit if most of the communication events do not happen simultaneously. In addition, the communication events may not access the memory controllers. If tasks access cache memory much more frequently than DRAM, a communication-aware task mapping method might not affect the congestion on memory controllers. In that case, the task mapping can affect the congestion on the shared caches. The communication concurrency and DRAM-to-memory ratio metrics are defined to evaluate the impacts of communication behaviors on the shared caches and memory controllers.

Communication concurrency ($CommC$) is defined as the average number of tasks per cluster. It is calculated by

$$CommC = \frac{\sum_{i=1}^{N_c} TaskN[C_i]}{T \cdot N_c}, \tag{2.4}$$

where N_c is the total number of clusters, and $TaskN[C_i]$ is the number of tasks of the communication events that belong to cluster C_i. These clusters are obtained from the method described in Section 2.3.3.

DRAM-to-memory ratio ($DramR$) is defined as the ratio of the number of DRAM accesses to the total number of memory accesses. $DramR$ is calculated by

$$DramR = \frac{\sum_{i=1}^{T} DramV[i]}{\sum_{i=1}^{T} MemV[i]}, \tag{2.5}$$

where $DramV[i]$ is the size of data in DRAM potentially accessed by task i. A communication-aware mapping method will have higher impacts on the memory congestion of parallel applications that have higher values of $CommR$, $CommC$ and $DramR$.

For communication-aware task mapping methods that aim to improve the locality of communication, it is necessary to have a high variance in the amount of communication per task pair. It is necessary because the locality-based mapping focuses on mapping the tasks that have a larger amount of communication than the other tasks. The communication locality metric $CommLoc$ is defined to describe the variance. A related work [21] is adopted to formulate this metric. First, the amount of communication of each task pair is normalized to the largest amount of communication among all task pairs. This normalization is shown by

$$S\text{comm}_{norm}[i][j] = \frac{S\text{comm}[i][j]}{max(S\text{comm})}. \tag{2.6}$$

Then, $CommLoc$ is calculated by

$$CommLoc = \frac{\sum_{i=1}^{T} var(Scomm_{norm}[i][1..T])}{T}, \tag{2.7}$$

where max and var are the functions that calculate the maximum and variance, respectively. A locality-based task mapping method will gain higher performance improvements for parallel applications that have higher values of $CommLoc$.

In a parallel application that has a low or zero communication-to-memory ratio, task mapping can still affect the memory access behavior if the application performs a substantial number of memory accesses. In this case, distributing the non-communicating memory accesses can reduce the memory congestion because it will improve the balance of memory accesses among the NUMA nodes. However, analyzing the communicating memory accesses is necessary to reduce the amount of remote memory accesses and the memory congestion because of two reasons. First, as shown in related work [2, 13], the cost of communication remains a performance-limiting factor in modern NUMA systems. Second, in many parallel applications, improving the communication locality also significantly affects the balance of memory accesses [26, 27], indicating that tasks with larger amounts of communication also perform substantial amount of memory accesses. In computation-intensive applications [54], where the ratio of memory accesses to computation is low, task mapping cannot significantly affect performance because the impact of memory access latency on the execution time is negligible.

All the proposed metrics, except *DramR* and *CommC*, are application-dependent, which means that the value of each metric is affected by the communication behaviors of the application. For *DramR* and *CommC*, the value of the metric also depends on the system used for obtaining the metrics. The sizes of the processor caches will affect the number of DRAM accesses of an application. A smaller last-level cache may increase the number of DRAM accesses because the application needs to fetch more data from DRAM. As previously discussed in Chapter 1, a high number of processor cores can induce a large number of concurrent communications. Thus, the value of *CommC* will increase with the number of parallel tasks and the number of processor cores.

2.3.5 Static and Dynamic Communication Behaviors

For offline and online task mapping methods, it is necessary to evaluate if an application changes its communication behavior during the execution. The communication behavior is *static* if it does not change during the execution, while it is *dynamic* if it changes during the execution. If an application has a static communication behavior, offline mapping is sufficient to improve the performance of the application. Moreover, for a target NUMA system, an offline mapping method only needs to run profiling and analysis once, and the mapping result can be reused for different executions. Unlike offline mapping, online mapping incurs an overhead to every execution of the application from continuous monitoring and repeated calculations of the mapping.

In the case of dynamic communication behavior, offline mapping can still give a performance benefit if the number of changes in the communication behavior is low. However, if an application frequently changes its communication behavior during the execution, offline mapping will not significantly impact the performance of the application because the static mapping is not sufficient to take into account the temporal changes of the communication behavior. Moreover, for different executions, an offline method may need to rerun the profiling and analysis steps to update the mapping. In contrast, online mapping can give a higher performance benefit to the application by dynamically changing the mapping to adapt to changes in the communication behavior.

To describe the static and dynamic communication behaviors of an application, the communication dynamic metric *CommDyn* is defined as the number of changes of the communication behavior during the execution of the application. Given a set of time intervals $\{it_1, it_2, ..., it_n\}$, *CommDyn* is calculated by

$$CommDyn = \sum_{i=1}^{Nit} compare(M_i, M_{i+1}), \tag{2.8}$$

where M_i is the communication matrix of interval it_i, Nit is the number of intervals, and *compare* is the function that calculates the difference between two communication matrices of different intervals. The output of the function is 0 or 1. To calculate the difference, the function first generates a task sequence for each matrix by sorting the tasks of the matrix by their amount of communication. Then, it compares the two sequences of the two matrices. The function will return 1 if the two sequences are different. Otherwise, it will return 0. Parallel applications that have higher values of *CommDyn* are expected to gain a higher performance improvement from online mapping.

2.4 Evaluation

This section presents the experimental evaluation of DeLocProf. First, it presents the methodology of the evaluation. Then, it presents the communication behaviors of a set of parallel applications and characterizes the communication behaviors using the proposed metrics. The results of the metrics are also used to classify the applications based on the suitability of offline and online task mapping methods to improve the performance of the applications. Finally, the effectiveness of the proposed metrics is discussed by examining the performance results of the applications.

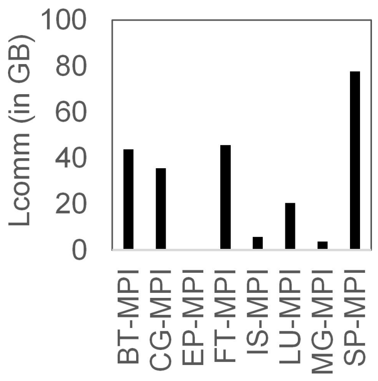
2.4.1 Evaluation Methodology

The experiments have been conducted on an Intel-based NUMA system, named Xeon56, that has two Intel Xeon E5-2680v4 processors. The system consists of two NUMA nodes, and is operated with Linux OS kernel v4.4. The NUMA nodes are connected with Quick-path Interconnect (QPI), and each node has 28 logical cores, 64 GB main memory, and an Integrated Memory Controller (IMC) [34]. Each NUMA node or processor has private L1 and L2 caches, and an L3 cache as last-level cache that is shared among all cores of the node. Thus, the system has 56 logical cores in total. The sizes of L1, L2, and L3 caches are 64 KB, 256 KB, and 35 MB, respectively.

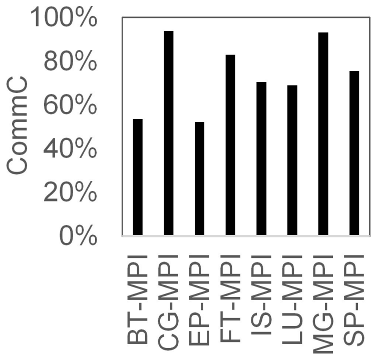
The evaluation uses three sets of parallel benchmarks as workloads: the MPI and OpenMP implementations of the NPB [53] v3.3.1, and the PARSEC benchmark suite [55] v2.1. The PARSEC benchmarks consist of parallel applications based on OpenMP and Pthreads implementations. All the NPB applications are executed with the class C input size, while PARSEC applications are executed with the native input size, which is the largest size available. The applications of PARSEC and the OpenMP implementation of NPB (NPB-OMP) are executed with 56 threads. For the MPI implementation (NPB-MPI), the number of processes executing the CG-MPI, FT-MPI, IS-MPI, and MG-MPI must be the power of two. Thus, 32 processes are launched for executing these four applications. The number of processes of BT-MPI and SP-MPI is required to be a square, and thus 49 processes are launched for executing BT-MPI and SP-MPI. EP-MPI and LU-MPI are executed with 56 processes using all the cores. For the NPB-MPI applications, Open MPI v3.1.4 [43] is used as the MPI implementation.

To analyze the communication behaviors, each application is preliminarily executed using the Scatter mapping, which is also the default mapping of the runtime system. This mapping is used because it does not consider the communication behaviors of the application, and thus, it does not require any information about the communication behaviors prior to the execution. This mapping is also used as the baseline for the performance evaluation. To detect the communications in the multithreaded applications, the timestamp of a communication event is traced in ns and μs time resolutions. The μs time resolution is only used for the NPB-OMP due to the tracing time constraints. However, the results of analysis of the communication behaviors of the NPB-OMP applications show that the method can still effectively analyze the communication behaviors of the NPB-OMP applications using μs time resolution. These results are presented in Section 2.4.2.

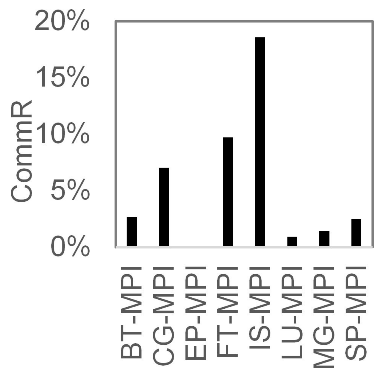
To discuss the effectiveness of the proposed metrics, this evaluation compares the performance results of four mapping methods and discusses the results with the metrics. The mapping methods used for the evaluation are Random, Scatter, Balance and Locality. Both Scatter and Random do not consider the communication behaviors of the application. In Random, the task mapping is randomly generated for each execution, and this mapping is used to evaluate the importance of task mapping. In contrast to Scatter and Random, both Balance and Locality consider the spatial communication behavior of the application. The differences among Scatter, Balance and Locality are shown in Figure 2.5. An existing method, called TreeMatch [12], is used to calculate the mapping for Locality. Balance and Locality use the communication matrix to calculate the mapping, and the matrix is obtained by using the method described in Section 2.3.3.



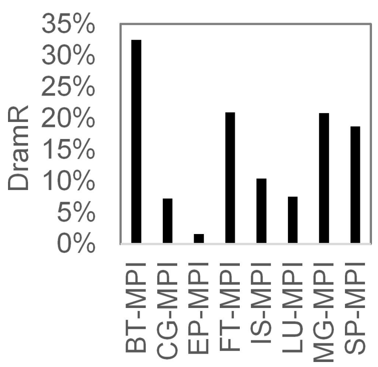
(a) *L*comm.



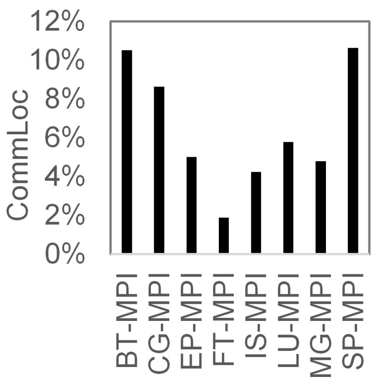
(b) *CommC*.



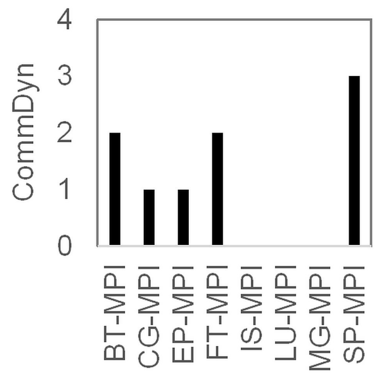
(c) *CommR*.



(d) *DramR*.



(e) *CommLoc*.



(f) *CommDyn*.

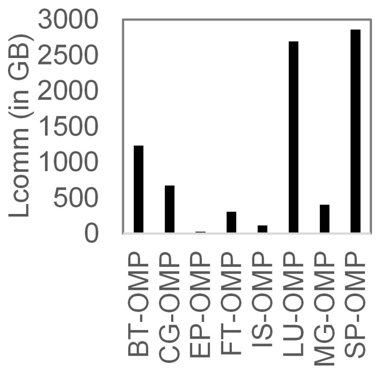
Figure 2.7: Communication behaviors of NPB-MPI described with the metrics.

2.4.2 Communication Behaviors of the Benchmarks

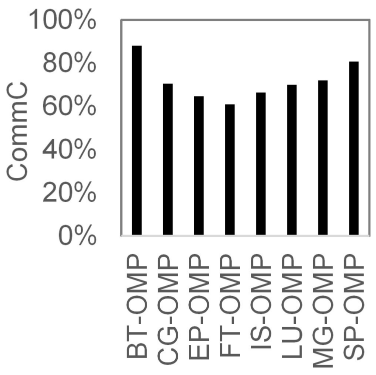
In this subsection, the communication behaviors of the benchmarks are analyzed using the metrics introduced in Section 2.3.4. All the metrics, except DRAM-to-memory ratio, are obtained using techniques described in Section 2.3.3. The DRAM-to-memory ratio is obtained by measuring performance counters with Linux perf tool [56]. To obtain the communication dynamic metric, the communication matrix is obtained for every 1-second interval. Figures 2.7, 2.8, and 2.9 show the values of the metrics for the NPB-MPI, NPB-OMP, and PARSEC applications, respectively. The vertical axis of each figure represents the values of the metric shown by the figure. The values of *CommC*, *CommR*, *DramR* and *CommLoc* are shown in percentages, and the values *L*comm are shown in gigabyte.

EP-MPI has a low communication load (Figure 2.7(a)) and communication-to-memory ratio (Figure 2.7(c)), indicating that it cannot gain improvement from communication-aware task mapping. CG-MPI, FT-MPI, IS-MPI, MG-MPI, and SP-MPI have higher communication concurrency (Figure 2.7(b)) and DRAM-to-memory ratio (Figure 2.7(d)) compared with the other applications. These results indicate that they can gain a higher performance improvement from a task mapping method that reduces memory congestion than that from the other applications. In BT-MPI, CG-MPI, LU-MPI, and SP-MPI, the communication locality is higher than that of the other applications (Figure 2.7(e)). However, LU-MPI has low communication-to-memory ratio and DRAM-to-memory ratio, indicating that it cannot gain significant performance improvement from communication-aware task mapping. On the other hand, BT-MPI, CG-MPI, and SP-MPI can gain a higher performance improvement from locality-based mapping compared with the other applications. Most of the NPB-MPI applications have dynamic communication behaviors (Figure 2.7(f)), indicating that most of the applications can gain a higher performance improvement from online task mapping. The results in Figure 2.7 indicate that all NPB-MPI applications, except EP-MPI and LU-MPI, are expected to benefit from the task mapping method that focuses on reducing memory congestion.

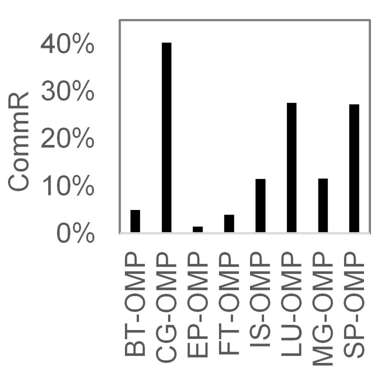
For NPB-OMP applications, the results of communication-to-memory (Figure 2.8(c)) and communication locality (Figure 2.8(e)) indicate that all the applications, except EP-OMP and FT-OMP, can benefit from locality-based mapping. In EP-OMP and FT-OMP, the load of communication (Figure 2.8(a)) and communication-to-memory ratio are low, indicating that these two applications cannot gain performance improvement from communication-aware task mapping. The results of communication concurrency (Figure 2.8(b)), communication-to-memory ratio and DRAM-to-memory ratio (Figure 2.8(d)) show that all NPB-OMP applications, except EP-OMP and FT-OMP, have a high risk of memory congestion. In CG-OMP, although the DRAM-to-memory ratio is low, the communication-to-memory ratio is the highest among the NPB-OMP applications. It means that CG-OMP has much more memory accesses to cache memory than that to DRAM, and it can benefit from task mapping to reduce the congestion of memory access to the shared caches. The results of communication dynamic (Figure 2.8(f)) show that most of the applications can benefit from online task mapping. The results in Figure 2.8 suggest that all NPB-OMP applications, except EP-OMP and FT-OMP, are expected



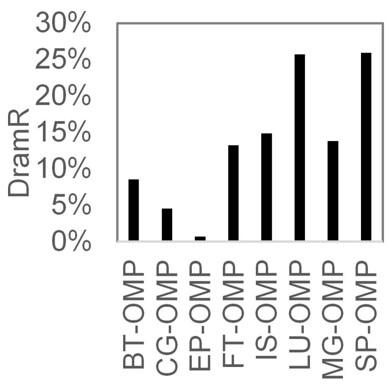
(a) *Lcomm*.



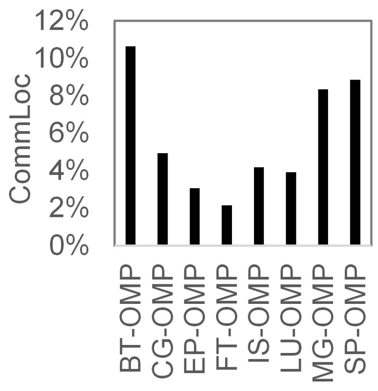
(b) *CommC*.



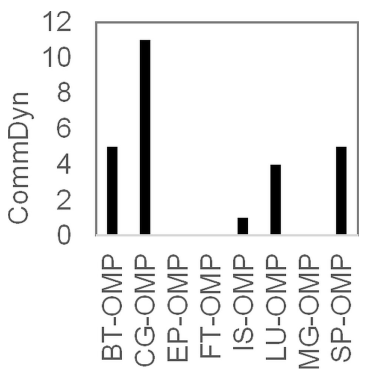
(c) *CommR*.



(d) *DramR*.



(e) *CommLoc*.

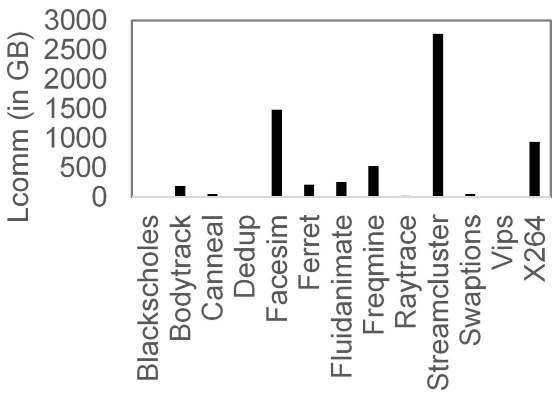


(f) *CommDyn*.

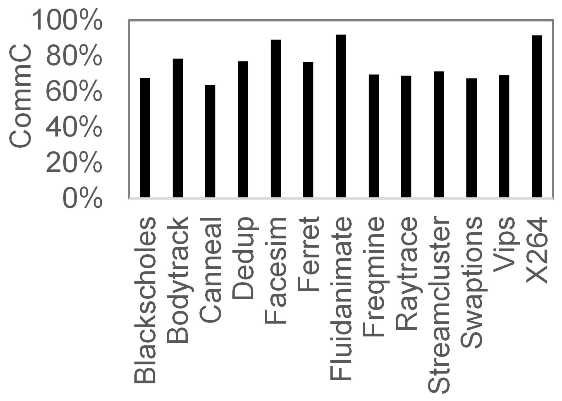
Figure 2.8: Communication behaviors of NPB-OMP described with the metrics.

to benefit from task mapping that reduces the number of remote accesses and memory congestion.

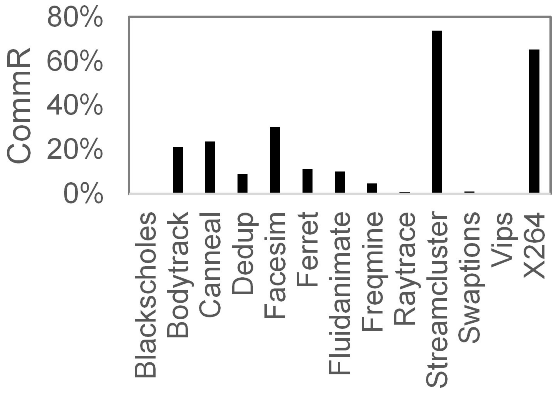
Although most PARSEC applications have a high communication concurrency (Figure 2.9(b)), some applications have a low communication-to-memory ratio (Figure 2.9(c)). It means that not all PARSEC applications will benefit from communication-aware task mapping. Most of the applications have dynamic communication behaviors (Figure 2.9(f)), indicating that these applications will also benefit from online task mapping. In Blackscholes, Swaptions and Vips, the load of communication (Figure 2.9(a)) and communication-to-memory ratio are negligible, indicating that these three applications cannot gain performance improvement from communication-aware task mapping. In Freqmine, although the communication load is higher than those of other PARSEC applications, the communication-to-memory ratio is low. It means that Freqmine cannot gain significant improvement from communication-aware task mapping. Although Canneal, Dedup, Fer-



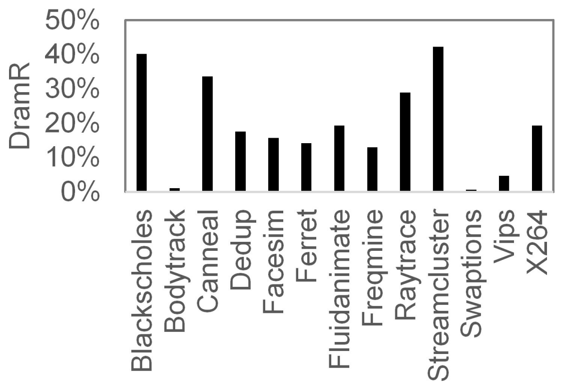
(a) *Lcomm*.



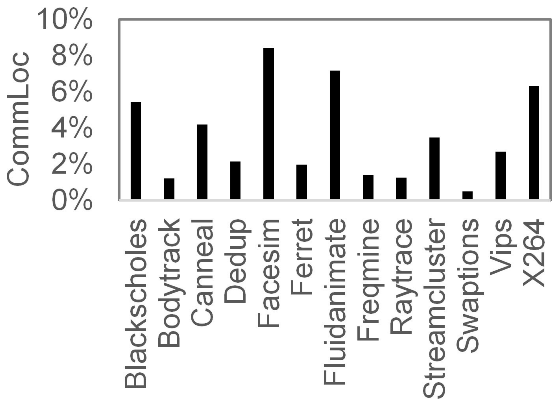
(b) *CommC*.



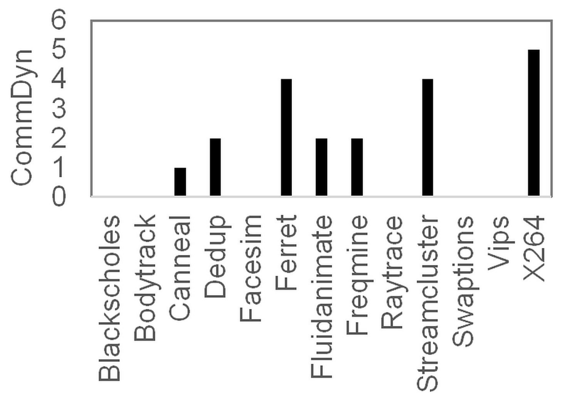
(c) *CommR*.



(d) *DramR*.



(e) *CommLoc*.



(f) *CommDyn*.

Figure 2.9: Communication behaviors of PARSEC described with the metrics.

ret and Fluidanimate have a lower communication load compared with Freqmine, these four applications have higher communication-to-memory ratio and DRAM-to-memory ratio (Figure 2.9(d)). Thus, Canneal, Dedup, Ferret and Fluidanimate can still gain performance improvements from memory-congestion aware task mapping.

In Bodytrack, the DRAM-to-memory ratio is low. However, it has a higher communication-to-memory ratio compared with most of the other applications. It means that Bodytrack has much more memory accesses to cache memory than those to DRAM, and thus it can benefit from the proposed method to reduce the congestion of memory access to the shared caches. In contrast to Bodytrack, Raytrace has a higher DRAM-to-memory ratio than most of the PARSEC applications, which means that it can benefit from the proposed method to reduce the congestion on memory controllers. On the other hand, Facesim, Streamcluster and X264 have higher loads of communication, communication concurrency, communication-to-memory ratio and DRAM-to-memory ratio compared with most of the other applications, indicating that these three applications will gain significant performance improvements from a task mapping method that reduces memory congestion. In Facesim, the communication locality (Figure 2.9(e)) is the highest among the PARSEC applications, indicating that it will gain a higher performance improvement from locality-based mapping compared with the other applications. The results in Figure 2.9 indicate that all PARSEC applications, except Blackscholes, Freqmine, Swaptions, and Vips, are expected to benefit from the proposed method.

2.4.3 Classification of the Applications

In this subsection, the benchmarks are classified using the benchmarking results presented in Section 2.4.2, which are measured on the Xeon56 system. Three classes are used for the classification: *Non-mapping*, *Static*, and *Dynamic*. Non-mapping is a class for applications that cannot benefit from task mapping to improve performance, while Static and Dynamic are classes for applications that have static and dynamic communication behaviors, respectively. Figure 2.10 shows the decision tree that is used to classify a

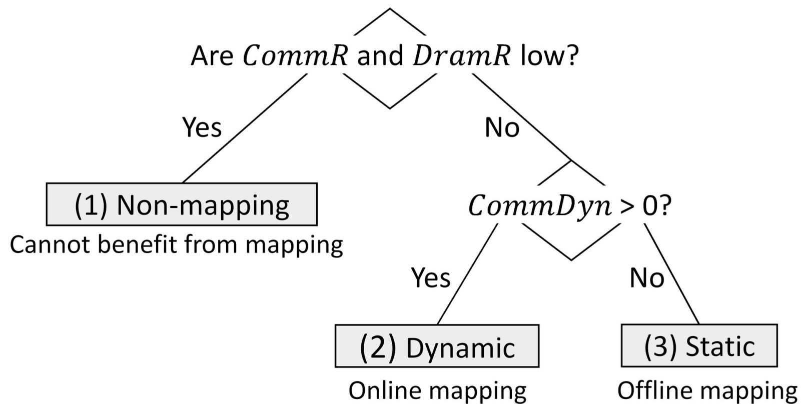


Figure 2.10: The decision tree for classifying a parallel application.

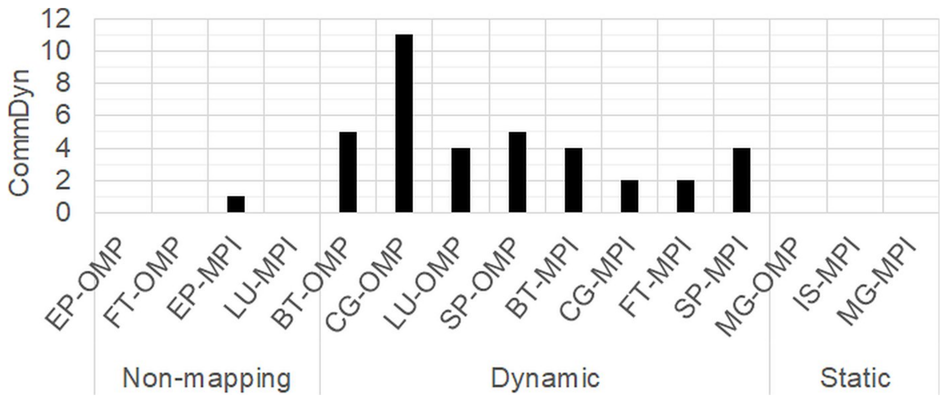


Figure 2.11: Classification results of the NPB.

parallel application. The decision tree uses *CommR*, *DramR*, and *CommDyn* metrics to classify the application.

First, the decision tree determines whether the application belongs to the Non-mapping class by evaluating the values of *CommR* and *DramR* metrics. As discussed in Section 2.3.4, a parallel application that has a low communication-to-memory ratio cannot gain a significant performance improvement from communication-aware task mapping. However, if the application has a high DRAM-to-memory ratio, task mapping can still give a performance benefit by reducing the congestion on the memory controllers. Since the *DramR* metric is system-dependent, the results of Non-mapping class may be different for different target systems. Then, if the application can benefit from task mapping, the decision tree determines if the application has static or dynamic communication behaviors by evaluating the value of *CommDyn*. As previously described in Section 2.3.5, offline

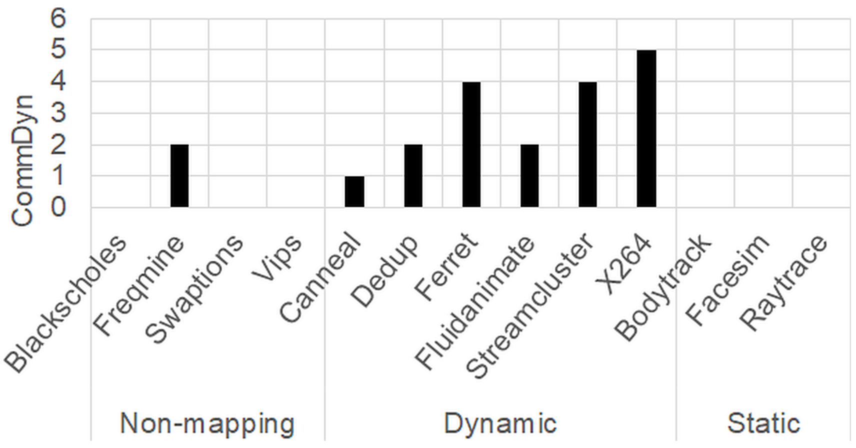


Figure 2.12: Classification results of the PARSEC.

and online task mappings are suitable for improving the performance of applications that belong to Static and Dynamic classes, respectively.

Figures 2.11 and 2.12 show the classification results for the NPB and PARSEC applications. The y-axis represents the value of the *CommDyn* metric for each application, which also represents the number of changes in the communication behavior during the execution of the application. Most of the NPB applications are classified as Dynamic, which means that most of the applications will gain a higher performance improvement from online task mapping than that from offline mapping. MG-OMP, IS-MPI, and MG-MPI belong to the Static class, suggesting that offline task mapping is suitable for improving the performance of these particular applications. The other NPB applications are classified as Non-mapping, indicating task mapping cannot improve the performance of these applications.

Most of the PARSEC applications are classified as Non-mapping and Static. As also described in Section 2.4.2, Blackscholes, Freqmines, Swaptions, and Vips cannot gain a performance improvement from task mapping. Thus, these four applications are classified as Non-mapping. Bodytrack, Facesim, and Raytrace are classified as Static, indicating that these three applications will gain a higher performance improvement from offline mapping than that from online mapping. The other six applications are classified as Dynamic, suggesting that online task mapping will give a higher performance benefit to these applications. The classification results show that both offline and online methods are

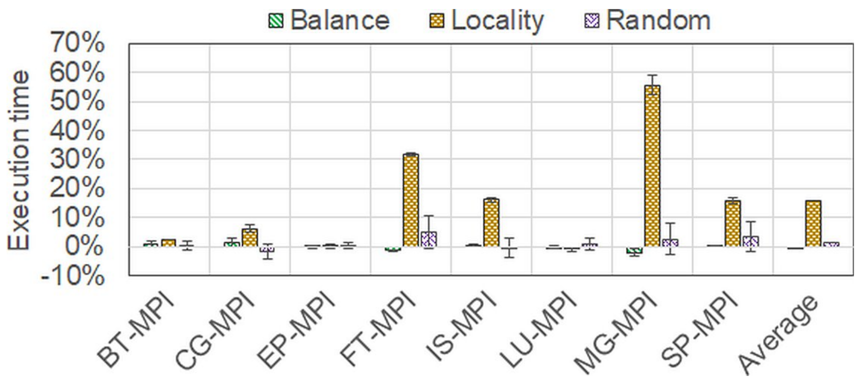


Figure 2.13: Performance results of NPB-MPI with the four mapping methods.

important in increasing performance, and the proposed metrics can be used to evaluate if a particular application can gain a performance benefit from these mapping methods.

2.4.4 Performance Results

Figures 2.13 and 2.14 show the performance results on the Xeon56 system. The performance results are obtained by measuring the execution time of the applications with each mapping method. The results are the averages obtained from 10 sample executions, which are normalized to the results of Scatter mapping. The figure also shows the 95% confidence interval calculated with Student's t-distribution [57]. The error line of the bar represents the confidence intervals of the samples.

Figure 2.13 depicts the execution time of the NPB-MPI applications. The results of Random mapping show higher margin of errors in most of the applications, indicating that most of the applications are affected by the task mapping. As predicted by the analysis of the communication behaviors of the NPB-MPI applications, Scatter and Balance can achieve a higher improvement than Locality for all NPB-MPI applications, except EP-MPI and LU-MPI. Compared with Locality, Balance gains the highest performance improvements for FT-MPI and MG-MPI by 33% and 57%, respectively.

In BT-MPI, CG-MPI and SP-MPI, the difference in execution times between Locality and Balance is smaller than that among FT-MPI, IS-MPI and FT-MPI because BT-MPI, CG-MPI and SP-MPI have the highest communication locality among the NPB-MPI applications (Figure 2.7(e)). However, in most of the NPB-MPI applications, Balance

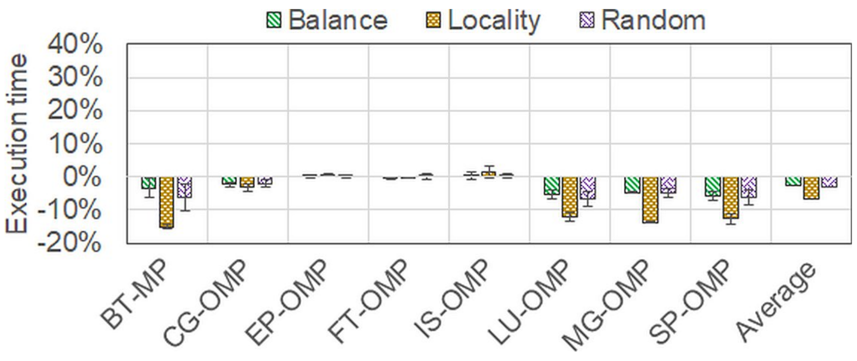


Figure 2.14: Performance results of NPB-OMP with the four mapping methods.

and Scatter outperform Locality, indicating that locality-based mapping cannot achieve the best performance of the applications. These results suggest that in the Xeon56 system, the impact of the memory congestion on the performance of the NPB-MPI applications is higher than that of the locality. Moreover, in the case when the number of tasks is less than the number of processor cores available, Locality can map more tasks to one NUMA node to reduce the number of remote accesses. Since the number of concurrent communications on one NUMA node becomes higher, the memory congestion increases on that particular node.

For NPB-OMP applications, task mapping also affects most of the applications. However, as shown in Figure 2.14, Scatter shows the lowest performance among the methods in most of the NPB-OMP applications, which is contrast to the results of NPB-MPI applications. As expected by previous analysis, Locality can achieve a higher average improvement than Balance and Scatter. These results indicate that most of the NPB-OMP applications gain more benefit from locality-based mapping. As also predicted by the previous analysis of the communication behaviors of NPB-OMP applications, Balance can improve the performance of BT-MP, LU-MP, MG-OMP and SP-OMP. As shown in the results of communication concurrency, communication-to-memory ratio and DRAM-to-memory ratio, these four applications have the highest risk of memory congestion among the NPB-OMP applications. By reducing the communication load imbalance, Balance also reduces memory congestion in these four applications.

In the case of PARSEC applications, Balance shows the highest average improvement

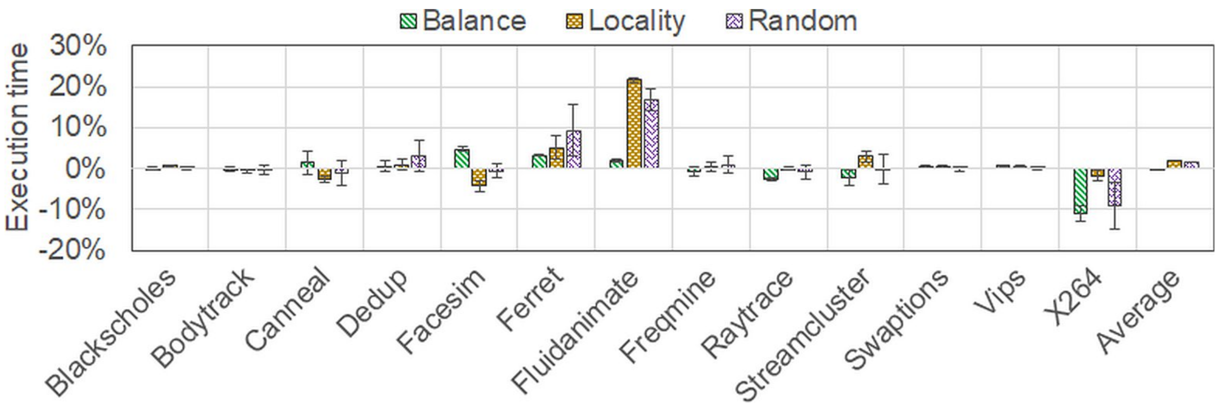


Figure 2.15: Performance results of PARSEC with four mapping methods.

among the four methods. As described by the metrics of the PARSEC applications, most of the applications gain performance improvements from task mapping that reduces the memory congestion. In Fluidanimate, Streamcluster and X264, Locality shows the lowest performance because these applications are susceptible to memory congestion. Maximizing the locality degrades the performance of these three applications. Locality increases the performance of Facesim because this application has a high communication locality (Figure 2.8(e)). The performance results of NPB-MPI, NPB-OMP and PARSEC applications suggest that the proposed metrics can effectively describe the communication behaviors that affect the locality and memory congestion on NUMA systems.

2.5 Conclusions

This chapter has discussed a method, DeLocProf, to analyze and characterize the spatial and temporal communication behaviors of parallel applications. The method consists of two techniques to obtain communication events of applications based on distributed-memory and shared-memory parallel processing methods, a data clustering method to identify concurrent communications, and a set of metrics to characterize the communication behaviors. To obtain the communication events of a multithreaded application, the proposed method detects implicit communications from the memory accesses among the threads of the application.

The experimental evaluation has been conducted on a real NUMA system with three sets of parallel benchmarks. The evaluation results demonstrate that the proposed method can effectively analyze and characterize the spatial and temporal communication behaviors of all the applications. The experimental results show that the impacts of task mapping on the execution time of the parallel applications vary depending on the communication behaviors of the applications, and the proposed metrics can describe the communication behaviors that affect the locality and memory congestion. The results of performance and analysis of the communication behaviors demonstrate that the metrics are effective in evaluating the impacts of a task mapping method on the performance of a particular application. In addition, the proposed metrics have been used to classify the parallel applications based on the suitability of offline and online task mapping methods to improve the performance of the applications.

Chapter 3

Offline Task Mapping for Coordinating Locality and Memory Congestion

3.1 Introduction

In NUMA systems, task mapping has a significant impact on the performance of a parallel application because it affects the cost of communication between tasks of the application. However, in recent NUMA systems, task mapping becomes more challenging because a large number of processor cores in a system can induce a large number of accesses to the memory devices, causing memory congestion. Furthermore, maximizing the locality can degrade performance of the applications because it potentially increases the memory congestion. In the case of many concurrent communications, the memory access traffic will be concentrated more on particular NUMA nodes.

A parallel application consists of multiple *tasks*, each of which is executed on a processor core as a thread or a process. An unit of executing each task is a thread in shared-memory parallel processing usually expressed with OpenMP directives, while it is a process in distributed-memory parallel processing usually programmed with MPI. Thus, in MPI and multithreaded applications, *process mapping* and *thread mapping* are also referred to as task mapping, respectively [39]. In NUMA systems, a communication between tasks will access the local memory device if it is performed by tasks that are executed by the processor 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 tasks that are executed by different cores of different NUMA nodes. Thus, the remote access needs a longer latency than the local access.

In the second chapter of this dissertation, a comprehensive evaluation has been conducted to analyze and characterize the communication behaviors of parallel applications, and to investigate the impacts of communication behaviors on the performance results of different task mapping methods. The evaluation results show that all the applications are susceptible to the memory congestion, and task mapping can improve performance by reducing the memory congestion. However, some applications can still gain a performance improvement from the locality-based mapping, indicating that the remote access penalty still has a significant impact on the performance of the applications. These results suggest two important facts. First, task mapping is crucial to improve the performance of parallel applications on NUMA system. Second, considering both locality and memory congestion is necessary to achieve a scalable performance on NUMA systems.

The objective of this chapter is to obtain task mapping for coordinating locality and memory congestion on NUMA systems. To achieve this objective, this chapter proposes an offline mapping method, called *decongested locality (DeLoc)*, that analyzes the communication behaviors and calculates the mapping before the execution of the application. The method consists of a task mapping algorithm and several techniques to gather the NUMA node topology of the target NUMA system and to identify tasks that potentially cause the memory congestion. To obtain the task mapping, DeLoc first retrieves information about the NUMA node topology of the target system and the communication behaviors of the target application. Then, DeLoc uses the retrieved information to calculate the mapping by using an algorithm, called *DeLocMap*. Finally, the mapping result is applied to the execution of the application.

To retrieve the information about the communication behaviors, DeLoc utilizes De-

LocProf, which is proposed in the second chapter of this dissertation. Thus, this chapter also highlights the contribution of DeLocProf to task mapping. For the evaluation, the experiments have been conducted on a real NUMA system and a multicore simulator with various applications from three sets of parallel benchmarks. The effectiveness of the proposed method is discussed by comparing the results of performance and energy consumption with a dynamic mapping method and five static mapping methods, including the current state-of-the-art locality-based method. A detailed analysis of the sources of performance and energy consumption improvements is also presented to discuss the effectiveness of DeLoc in improving the locality and reducing the memory congestion.

The main contribution of this chapter is the DeLoc method that can address the locality and memory congestion problems on NUMA systems. It includes the DeLocMap algorithm that uses information about the NUMA node topology and the communication behaviors to calculate the mapping that can coordinate the locality and memory congestion.

The rest of this chapter is organized as follows. Section 3.2 reviews the related work. The proposed method is presented in Section 3.3. The evaluation results are discussed in Section 3.4. Finally, Section 3.5 summarizes the conclusions of this chapter.

3.2 Related Work

This section reviews the related work of this chapter. It first reviews existing methods for MPI process mapping. Then, it reviews existing methods for thread mapping. Finally, it reviews existing methods for thread and data mapping that consider the memory congestion on NUMA systems.

3.2.1 MPI Process Mapping

Various MPI process mapping 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 [11]. The task mapping problem is NP-hard [58], and thus it is necessary to use heuristic algorithms to calculate the mapping. Hendrickson and Leland proposed graph-based partitioning algorithms to optimize the process mapping [59]. Zhang et al. [24] and Ma et al. [60] proposed process placement strategies for MPI collective operations. A more recent method was proposed by Jeannot et al. with a tree-based algorithm called TreeMatch [12]. It achieves a better performance than the previous graph-based partitioning algorithms by taking into account the spatial communication behavior of the application and the hardware topology of the system. All of these related work focused only on improving the locality. As demonstrated in the second chapter of this dissertation, considering only the locality is not sufficient to achieve the best performance of the applications. In contrast to these related work, DeLoc focuses on both improving the locality and reducing the memory congestion. In the evaluation of this chapter, the benefits of DeLoc are discussed, with a comparison of the performance results of DeLoc and TreeMatch.

3.2.2 Thread Mapping

A thread mapping method was proposed by Diener et al. [26]. The method analyzes the spatial communication behavior of multithreaded applications to improve the locality and the balance of communication. In contrast to this method, DeLoc considers not only the spatial communication behavior, but also the temporal communication behavior of the application. As discussed in Chapter 2, analyzing the temporal communication behavior is necessary to effectively reduce the memory congestion. An online-based communication detection method, called CDSM, was proposed by Diener et al. [2]. During the execution of an application, it continuously monitors and detects communications among the threads of the application. Then, it periodically analyzes the communication behavior and migrates threads to improve the locality. In contrast to CDSM, DeLoc focuses on both improving the locality and reducing the memory congestion. Moreover, DeLoc does not introduce the runtime overhead caused by the continuous monitoring and thread migrations.

3.2.3 Thread and Data Mapping

Thread mapping and *data mapping* are commonly used to improve the locality of memory accesses on NUMA systems [29]. Recent thread mapping mechanisms [2, 13, 14, 61] work in the system level to dynamically migrate the threads to improve the locality and balance of memory accesses among NUMA nodes. The main advantage of the system-level mechanisms is that they can take into account the behavior of multiple applications running at the same time. However, these mechanisms incur runtime overhead when migrating tasks during the execution, which can significantly affect performance and energy consumption. The related work [2, 27] has shown that the migration overhead can increase cache misses and interconnect traffic. To reduce the migration overhead, thread mapping and data mapping can be performed in an integrated way [21, 29]. By mapping the data instead of threads, a thread mapping mechanism can prevent unnecessary thread migrations between NUMA nodes.

Dashti et al. [13] proposed a data mapping method, called Carrefour, to reconcile the data access locality and memory congestion on NUMA systems. Carrefour works as a Linux kernel policy to dynamically migrate memory pages between NUMA nodes to avoid the congestion. Lepers et al. [14] proposed a thread and data mapping method, called AsymSched, that takes into account the bandwidth asymmetry of asymmetric NUMA systems to minimize congestion on interconnect links and memory controllers on recent NUMA systems. It relies on continuous monitoring of the communications, thread migration, and memory migration. As shown in [21, 62], task mapping is a prerequisite of data mapping, and the primary benefit of data mapping is that it can prevent unnecessary task migrations. In contrast to these related work, DeLoc applies task mapping when the target application is launched, and thus it does not need to migrate tasks during the execution of the application.

CDSM, Carrefour and AsymSched introduce monitoring and migration overheads to the execution of the parallel application. In contrast to these methods, DeLoc does not suffer from the migration overhead. The overhead of DeLoc is incurred by the offline profiling and analysis. However, a rerun of the profiling and analysis is required only when the communication behaviors of the application have changed. Moreover, unlike these methods, DeLoc works on the application level and does not rely on a specific operating system or hardware. Compared with AsymSched, DeLoc focuses on reducing not only memory congestion but also the amount of remote accesses. As shown in the evaluation of this chapter, Facesim and most of the NPB-OMP applications can gain significant improvements from reducing the amount of remote accesses.

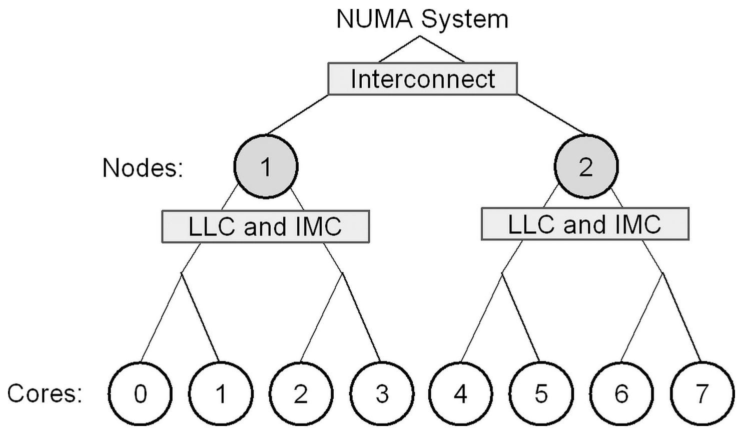


Figure 3.1: An example of the NUMA node topology with eight cores.

3.3 DeLoc: A Task Mapping Method for Coordinating Locality and Memory Congestion

This section describes DeLoc that can address both the locality and memory congestion problems. The procedure of DeLoc is summarized as follows.

1. Gather the NUMA node topology information of the target system.
2. Analyze the spatial and temporal communication behaviors of the target application.
3. Compute a mapping between tasks and the processor cores using the DeLocMap algorithm.

The following three subsections describe Steps 1, 2, and 3, respectively. Then the scalability of DeLoc is discussed in Section 3.3.4.

3.3.1 Gathering the NUMA Node Topology Information of the Target System

The first step is to retrieve information about the NUMA node topology of the target NUMA system. To obtain all of the information required in this step, the use of a specific tool is not mandatory. Some tools, such as Hwloc [63] and numactl [64], are available for the information retrieval. The topology is modeled as a tree to express the information on the locations of shared last-level caches, memory controllers, and interconnect links. This information is required because DeLoc focuses on reducing the amount of remote accesses through interconnects and reducing the congestion on the shared caches and memory controllers.

In the NUMA systems considered in this chapter, each NUMA node is physically associated with a shared last-level cache (LLC) and an integrated memory controller (IMC), such as in Intel-based and AMD-based NUMA systems [4]. Thus, the location of memory controllers also represents the location of the last-level caches. Figure 3.1 depicts an example of the model of a two-node NUMA system that consists of eight processor cores. The topology information also includes the identity information of the NUMA nodes and processor cores. The identity information is used later by the mapping algorithm to match the tasks with the processor cores.

3.3.2 Analyzing the Communication Behaviors of the Application

The main purpose of this step is to identify groups of tasks that perform concurrent communications by analyzing the spatial and temporal communication behaviors of the target application. To analyze the spatial and temporal communication behaviors, this step first needs to obtain a time-series dataset of communication events among the parallel tasks. This dataset is obtained by tracing the communication events while preliminarily executing the application on the target system. In multithreaded applications, this step includes detecting the implicit communication between the threads. For the tracing, DeLoc uses DeLocProf, which is proposed in Chapter 2. With this tool, DeLoc can be applied to applications based on message-passing and shared-memory parallel processing. The tracing tool generates the time-series dataset of communication events by recording the IDs of the sender and receiver, the timestamp, and the data size of each event.

After the time series dataset is obtained, they are analyzed to identify the concurrent communications by using the data clustering method that is also proposed in Chapter 2.

Algorithm 1 The DeLocMap Algorithm.

```
Input: $T$ {The NUMA node topology}
Input: $G$ {The groups of task pairs}
Output: $M$ {The map of processor core IDs and task IDs}
 1: $M \leftarrow initializeMap(T)$
 2: $weightedG \leftarrow calculateLoads(G)$ {Calculate $L$pair and $L$group}
 3: $sortedG \leftarrow sortByLg(weightedG)$ {Sort the groups by their $L$group}
 4: $current\_node \leftarrow firstNode(T)$
 5: $i \leftarrow 0$
 6: while $i < count(sortedG)$ and $countUnmappedCores(M) > 0$ do
 7:   $currentPairs \leftarrow sortedG[i]$
 8:   $sortedPairs \leftarrow sortByLp(currentPairs)$ {Sort the pairs by their $L$pair}
 9:   $N_{pairs\_c} \leftarrow count(sortedPairs)$
10:   for $j = 0$ to $N_{pairs\_c}$ do
11:     if not $mapped(sortedPairs[j])$ then
12:       $mapPair(sortedPairs[j], current\_node, M)$
13:       $current\_node \leftarrow nextNode(T)$
14:     end if
15:   end for
16:   $i \leftarrow i + 1$
17: end while
```

The result of the clustering method is a clustered dataset of communication events, where each event is associated with a cluster identifier. In the dataset, multiple communication events that belong to the same cluster are identified as concurrent communications. This clustered dataset represents the spatial and temporal communication behaviors of the application. Finally, groups of task pairs are generated from this dataset by aggregating the communication events for each cluster. The task pairs of the communication events that belong to the same cluster are considered as a group. Thus, each group consists of task pairs that perform the concurrent communications.

3.3.3 Computing the Task Mapping

The final step is to compute the mapping between tasks and processor cores using the DeLocMap algorithm. This algorithm, depicted in Algorithm 1, can calculate a match between task IDs and processor core IDs using the node topology information and the groups of task pairs obtained from the previous steps.

The algorithm works as follows: first, DeLocMap uses the topology model to construct the map of processor core IDs and task IDs (Line 1). The keys of the map represent the IDs of processor cores available in the system, and each value represents the ID of the task associated with the key. At the beginning of the algorithm, each value is set to empty. A pair of tasks can belong to multiple groups because the pair can communicate at different times. Since the algorithm calculates only one mapping for the whole execution, groups that have a higher amount of communication must take precedence to be mapped over the other groups. However, in such a case, avoiding congestion may increase the amount of remote accesses. This case is discussed in Section 3.4.2.

To determine the order of the groups, DeLocMap calculates two load metrics (Line 2). It first calculates the load of a task pair $L\text{pair}$ by normalizing the data size of the communication event of the pair $S\text{comm}$ to its highest value, as defined by Equation (3.1). $S\text{comm}$ represents the amount of communication of the task pair. Then the load of a group, $L\text{group}$, is calculated by Equation (3.2), where $N_{\text{all-pairs}}$ is the total number of pairs in all groups, and $N_{\text{pairs-g}}$ is the total number of task pairs in group g.

$$L\text{pair} = \frac{S\text{comm}}{\sum_{i=1}^{N_{\text{all-pairs}}} S\text{comm}_i}, \tag{3.1}$$

$$L\text{group} = \sum_{i=1}^{N_{\text{pairs-g}}} L\text{pair}_i. \tag{3.2}$$

After calculating the load metrics, the algorithm selects a task pair that has not been mapped to processor cores sequentially from the groups with the highest to the lowest $L\text{group}$, and from the pairs with the highest to the lowest $L\text{pair}$. This selection is achieved by the sorting steps in the algorithm (Lines 3 and 8). The algorithm then maps each task pair to the processor cores that are currently unmapped in the current NUMA node (Line 12). The current NUMA node is obtained by traversing the NUMA nodes in the topology tree (Lines 4 and 13). DeLocMap aims to improve the locality by mapping two tasks of a pair to the same NUMA node, while also reducing the memory congestion by mapping the task pairs in the same group to the different NUMA nodes.

As shown in Algorithm 1, the applicability of DeLoc depends on the node topology of the target NUMA system. A NUMA system can have a more complex topology than the topology shown in Figure 3.1. However, DeLoc uses a tree model to express the NUMA node topology of the target system. This tree model is a prerequisite for the DeLocMap algorithm. Thus, DeLoc can apply to any NUMA node topologies that can be expressed as a tree.

3.3.4 Scalability of DeLoc

The scalability of DeLoc is determined by the complexities of its steps. Each of the step, except the mapping computation, can be executed in parallel. The complexity of the analysis step is determined by the time complexity of the clustering method, which is $O(N_{events} \log k)$ [52], where k is the number of clusters evaluated. The complexity of the mapping computation is determined by the time complexity of the DeLocMap algorithm, which is $O(N_{\text{all-pairs}} \log N_{\text{all-pairs}})$ on average. The complexity of the tracing step depends on the parallel processing method of the target application. For MPI applications, the time complexity of the tracing step is $O(N_{events})$, where N_{events} is the number of communication events. In the case of multithreaded application, the communication events are traced from the memory accesses among threads. Thus, in this case, the time complexity of the tracing step is $O(N_{mem})$, where N_{mem} is the number of memory accesses.

3.4 Evaluation

To evaluate the effectiveness of the proposed method, two experiments have been conducted on a real system and a simulation environment. This section first describes the experimental setup, and then discusses the performance results.

3.4.1 Performance Evaluation on a Real System

The main experiments have been conducted on the Xeon56 system, which is also used in Chapter 2. Three sets of parallel benchmarks are used as workloads: the MPI and OpenMP implementations of the NPB [53] v3.3.1, and the PARSEC benchmark suite [55] v2.1. The input size and the number of tasks used for executing each application are the same as used in the evaluation of Chapter 2. For the NPB-MPI applications, Open MPI v3.1.4 [43] is used as the MPI runtime system. By default, Open MPI uses the *vader* BTL component to optimize the data transfer between NUMA nodes.

For detecting the communications in the multithreaded applications, the timestamp of communication event is traced using the same time resolution as used in Section 2.4. The mapping results of DeLoc is applied to the execution of the application by assigning tasks to processor cores when the application is launched. For the MPI applications, the tasks are assigned to processor cores by specifying the mapping between MPI ranks and processor core IDs in the rank file. For the multithreaded applications, the tasks are assigned to processor cores by setting the processor affinity for each thread according to the mapping result. Likwid-pin [65] is used to set the processor affinity of the threads.

In this evaluation, DeLoc is compared to a dynamic mapping method and five static mapping methods. The dynamic mapping method is called AutoNUMA [13,29,61], which is the default thread and data mapping mechanisms used in the Linux kernel. It can be enabled and disabled through the sysctl interface by setting `kernel.numa_balancing` to 1 and 0, respectively. AutoNUMA uses information about page faults of parallel applications to dynamically detect memory accesses, and performs thread and data mapping. During the application runtime, it migrates memory pages and threads to improve the locality and balance of memory access among the NUMA nodes, and thus incurs a certain overhead from the migration.

The five static methods used for the evaluation are Packed, Scatter, Balance, Locality, and Random mapping. The differences among Packed, Scatter, Balance and Locality are described in Section 2.2. Both Packed and Scatter do not consider the communication behaviors of the application. Packed maps the neighboring tasks to the same NUMA node, while Scatter maps the neighboring tasks to the different NUMA nodes. In the case where neighboring tasks have a larger amount of communication than the other tasks, Packed will increase the locality of communication, while Scatter will reduce the communication load imbalance among the NUMA nodes. This case is discussed in more detail in Section 3.4.1.

In contrast to Packed and Scatter, Balance and Locality consider the spatial communication behavior of the application. Balance aims to maximize the balance of communication among the NUMA nodes, while Locality focuses on improving the locality of communication. The mapping for Locality is obtained by using the TreeMatch algorithm [12], which is the current state-of-the-art algorithm for maximizing the locality of communication. Random generates a task mapping for each execution. To avoid the effects the thread and data migrations on the results of the static methods, the AutoNUMA is disabled when executing the benchmarks with the static mapping methods. In addition, the *first-touch* mapping [7] is used as the data mapping policy, which is the default mapping policy of the Linux kernel. In first-touch data mapping, a memory page is allocated on the same node with the task that first uses the page, and the page is not migrated during the execution.

As discussed in Section 3.2, data mapping can also affect the memory accesses on NUMA systems, and is used to prevent unnecessary task migrations. However, DeLoc and the other static mapping methods apply the task mapping only when the target application is launched, and thus these methods do not need to migrate tasks during the execution of the application. Furthermore, the first-touch policy allocates a memory page according to the tasks that first uses the page. Therefore, in the case of DeLoc and the other static methods, the task mapping also determines the data mapping. However, in contrast to the other static mapping methods, DeLocMap algorithm computes the task mapping that can both improve the memory access locality and reduce the memory congestion.

3.4.1.1 Analyzing the Communication Behaviors of the Benchmarks

In Section 2.4, the communication behaviors of NPB and PARSEC applications have been thoroughly analyzed. The analysis results suggest that all the NPB-MPI applications, except EP-MPI and LU-MPI, gain performance improvements from task mapping that reduces the memory congestion. It means that most of the NPB-MPI applications are expected to benefit from DeLoc and Balance. For the NPB-OMP applications, only EP-OMP and FT-OMP cannot gain performance improvements from task mapping because these two applications have a low communication-to-memory ratio (Figure 2.8(c)). Most NPB-OMP applications have a high communication-to-memory ratio, DRAM-to-memory ratio (Figure 2.8(d)), and communication locality (Figure 2.8(e)), indicating that these applications will gain performance improvements from task mapping that improves the locality and reduces the memory congestion. Thus, DeLoc is expected to improve the performance of these applications.

Most of the PARSEC applications are also expected to gain performance improvements from DeLoc. Facesim has a high communication locality (Figure 2.9(e)), indicating that it will gain a performance improvement from task mapping that improves the locality, such as DeLoc and Locality. However, Blackscholes, Freqmine, Swaptions and Vips cannot benefit from task mapping because they show a low load of communication (Figure 2.9(a)) and communication-to-memory ratio (Figure 2.9(c)). For the other PARSEC applications, the results of communication concurrency (Figure 2.9(b)), communication-to-memory ratio and DRAM-to-memory ratio (Figure 2.9(d)) show that these applications will gain performance improvements from task mapping that reduces the memory congestion, such as DeLoc and Balance.

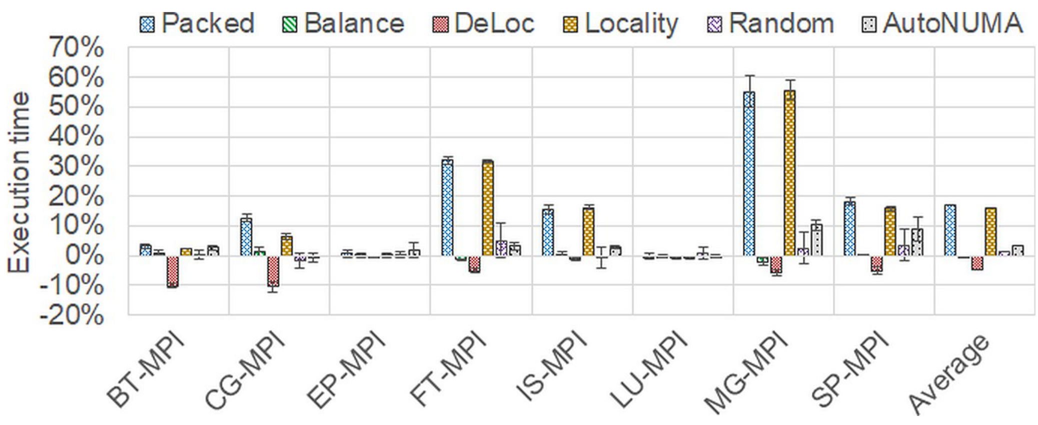


Figure 3.2: Performance results of NPB-MPI on Xeon56

3.4.1.2 Performance Results

Figures 3.2, 3.3, and 3.4 show the performance results obtained on the Xeon56 system. The performance results are obtained by measuring the execution time of the applications with each mapping method. The results are the averages obtained from 10 sample executions, which are normalized with the results of Scatter mapping. The figures also show 95% confidence interval calculated with Student's t-distribution. The error line of the bar represents the confidence intervals of the samples. Scatter is used as the baseline because, as shown in the related work [13,37,49,66], the memory access imbalance among the NUMA nodes can increase the memory congestion, and Scatter can reduce the memory access imbalance among the NUMA nodes without the need of information about the communication behaviour of the application.

Figure 3.2 depicts the execution time of the NPB-MPI applications. The results of Random mapping show that most of the NPB-MPI applications are affected by the task mapping. On average, DeLoc shows the highest improvements among the methods, by 4.8% compared with Scatter. As predicted by the previous analysis of the communication behaviors of the NPB-MPI applications, DeLoc can achieve the highest improvements for all NPB-MPI applications, except EP-MPI and LU-MPI. Compared with Locality, DeLoc gains the highest performance improvements for FT-MPI and MG-MPI by 36.8% and 61%, respectively.

In BT-MPI, CG-MPI and SP-MPI, Locality shows a shorter execution time than

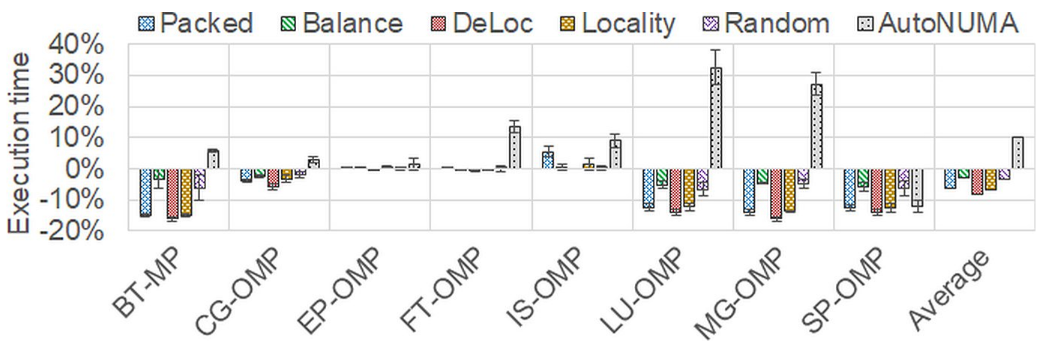


Figure 3.3: Performance results of NPB-OMP on Xeon56

Packed because these three applications have the highest communication locality among the NPB-MPI applications (Figure 2.7(e)). However, in most of the NPB-MPI applications, DeLoc, Balance, AutoNUMA, and Scatter outperform Locality, indicating that locality-based mapping cannot achieve the best performance of the applications. These results suggest that in the Xeon56 system, the impact of memory congestion on the performance of the NPB-MPI applications is higher than that of the locality. Furthermore, in the case where the number of tasks is less than the number of processor cores available, Locality can map more tasks to one NUMA node to reduce the number of remote accesses. Since the number of concurrent communications on one NUMA node becomes higher, the memory congestion increases on that particular node. However, the results of Balance and DeLoc also show that minimizing the communication load imbalance itself is not sufficient to achieve the best performance, and considering both the locality and the memory congestion is still crucial to achieving the best performance.

For NPB-OMP applications, task mapping also affects most of the applications. However, as shown in Figure 3.3, Scatter shows the lowest performance among the methods in most of the NPB-OMP applications. These results are contrast to those of NPB-MPI applications. Moreover, on average, Locality achieves higher performance improvements than Packed, Balance and Scatter. These results indicate that most of the NPB-OMP applications gain more benefit from locality-based mapping. On the other hand, DeLoc achieves the highest performance improvements among the methods in most of the NPB-OMP applications, by up to 16.1% in the cases of BT-OMP and MG-OMP (8.3% on

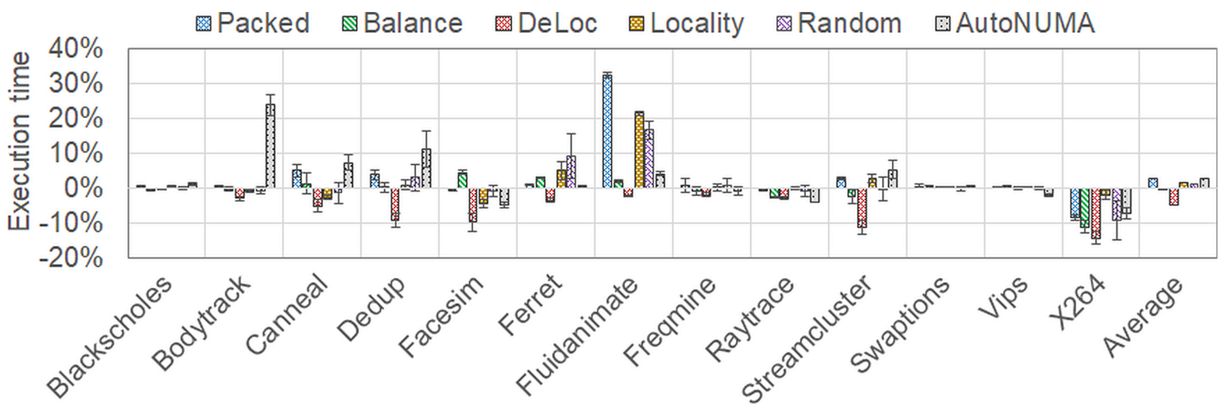


Figure 3.4: Performance results of PARSEC on Xeon56.

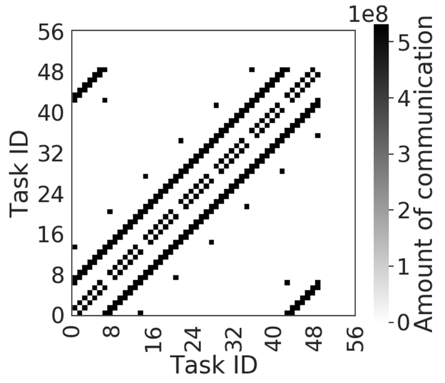
average). As predicted by the analysis of the communication behaviors of NPB-OMP applications, DeLoc can achieve the highest performance improvements in BT-OMP, LU-OMP, MG-OMP and SP-OMP. As shown in the results of communication concurrency, communication-to-memory ratio and DRAM-to-memory ratio, these four applications have the highest risk of memory congestion among the NPB-OMP applications. This fact indicates that only considering the locality is not sufficient to achieve the best performance for these applications.

As predicted in Chapter 2, DeLoc can reduce the execution times of most of the PARSEC applications. Moreover, on average, DeLoc can achieve the highest improvements among the methods. Among the PARSEC applications, DeLoc can achieve the highest performance improvements in Facesim, Streamcluster, and X264 by 9.7%, 11% and 14.1%, respectively. Compared with Balance, DeLoc shows the highest improvement in Facesim by 14.3%. These results suggest the importance of increasing locality to improve the performance of Facesim. In Fluidanimate, DeLoc can achieve a 23.7% shorter execution time than that of Locality. Moreover, compared with Packed, Balance and DeLoc, Locality shows the lowest performance in Fluidanimate, Streamcluster and X264, indicating that maximizing the locality degrades the performance of these three applications.

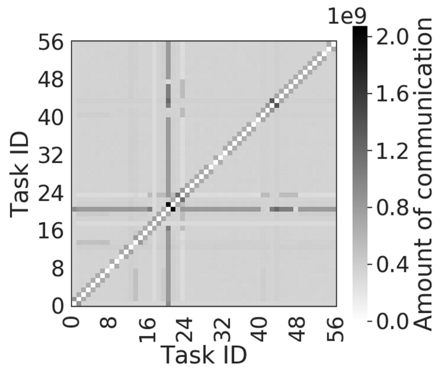
In most of the NPB-OMP applications and some PARSEC applications, AutoNUMA shows the longest execution time among the methods. In LU-OMP, AutoNUMA shows the highest performance degradation by 46.5% and 32.4% compared with DeLoc and Scatter, respectively. Furthermore, in FT-OMP and Bodytrack, although most of the static methods show a similar execution time, AutoNUMA shows a much longer execution time than the other methods. In contrast to the static mapping methods, AutoNUMA suffers overhead from migrating memory pages and threads during the application runtime. These results show that the migration overhead significantly degrades the performance of the applications. In most of the NPB-OMP applications, AutoNUMA shows the lowest performance among the methods. These results are in contrast to those of NPB-MPI applications. It is because NPB-OMP applications have much more memory accesses than NPB-MPI applications. The migration overhead has higher impacts on the performance of NPB-OMP applications. The impacts of the migration overhead are discussed in more detail in Section 3.4.1.3.

The performance results on Xeon56 show that DeLoc can consistently achieve the highest performance among the methods. By taking into account both spatial and temporal communication behaviors of the applications, it can effectively reduce the amount of remote accesses and memory congestion. These results also show the effectiveness of the method proposed in Chapter 2 to analyze the communication behaviors of all the tested applications.

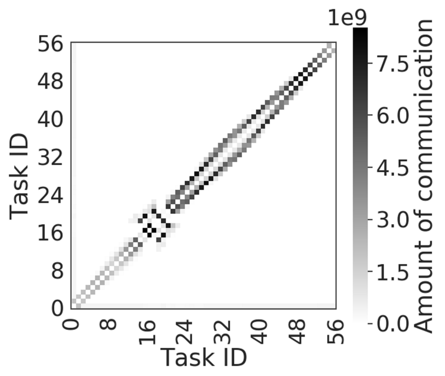
The performance results of Scatter show that it achieves shorter execution times than Locality and Packed for most NPB-MPI applications, and Packed can achieve shorter execution times than Balance for most NPB-OMP applications. Although Packed and Scatter do not consider the communication behaviors of applications, these two mapping methods can effectively improve the performance of applications that have a communication behavior, in which neighboring tasks have a larger amount of communication than the other tasks. However, as shown in the results of NPB-MPI, NPB-OMP and PARSEC applications, both Packed and Scatter cannot consistently improve performance. Furthermore, as shown in [49], when the number of NUMA nodes in the system becomes larger, Scatter may suffer from high latencies of remote accesses and can not effectively reduce the memory congestion. These results show that to effectively reduce the amount of remote accesses and memory congestion, it is necessary to consider the communication



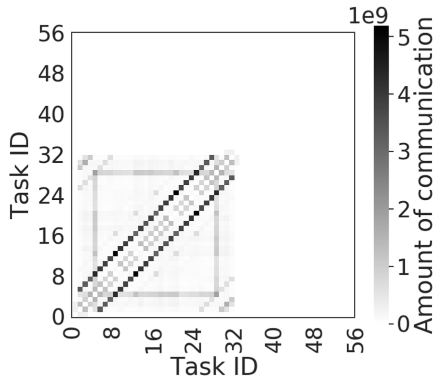
(a) SP-MPI.



(b) CG-OMP.



(c) MG-OMP.



(d) Fluidanimate.

Figure 3.5: Communication behaviors of the applications that benefit from Packed and Scatter mappings.

behaviors of the application.

Figure 3.5 shows examples of the communication behavior that can benefit from the Packed and Scatter mappings. Figures 3.5(a), 3.5(b), 3.5(c) and 3.5(d) show the spatial communication behaviors of SP-MPI, CG-OMP, MG-OMP and Fluidanimate, respectively. In the figures, the x-axis and y-axis show the task ID, and each cell represents the amount of communication (Scomm) between a task pair of the corresponding axes. The values of communication amount are in bytes and shown in E notation [67]. The darker cells indicate a larger amount of communication. As shown in the figures, SP-MPI, CG-OMP, MG-OMP and Fluidanimate show a similar communication behavior, with a large amount of communication between neighboring tasks, such as task pair (0,1). These results show that in these four applications, Packed will reduce the amount of remote accesses, and Scatter will reduce the communication load imbalance among the

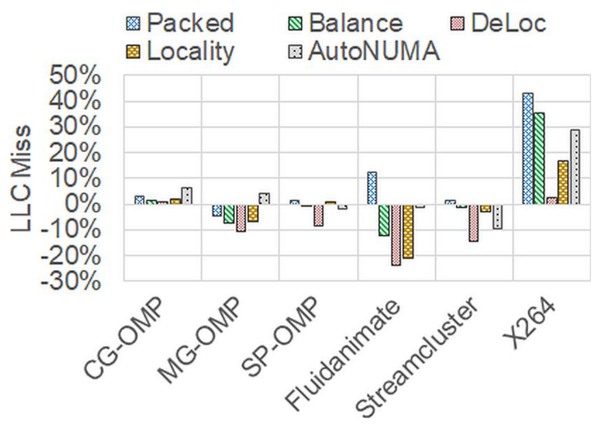
NUMA nodes. Moreover, SP-MPI and Fluidanimate show a similar amount of communication between tasks that are further apart, such as task pairs (1,7) and (2,8) in SP-MPI, and task pairs (1,5) and (2,6) in Fluidanimate. These results show that in SP-MPI and Fluidanimate, Scatter can also reduce the amount of remote accesses.

3.4.1.3 Performance Analysis

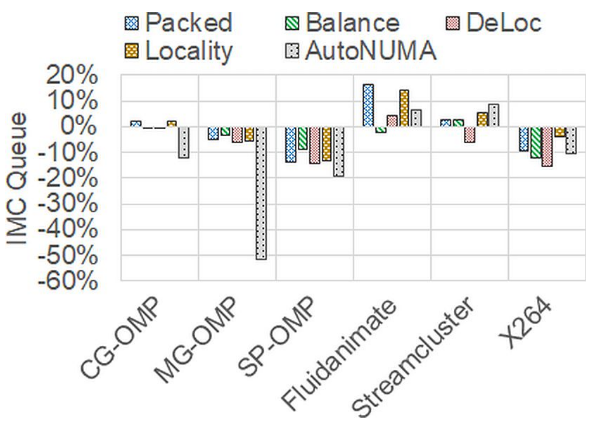
The sources of performance improvements are investigated by analyzing the performance characteristics of six applications selected from NPB-OMP and PARSEC. These applications are CG-OMP, MG-OMP and SP-OMP of the NPB, and Fluidanimate, Streamcluster and X264 of the PARSEC. Three metrics are used to quantitatively compare the performance characteristics of these applications: LLC miss, IMC queue, and QPI volume. These metrics are obtained by measuring the Intel performance counters [68] with Linux perf tool.

LLC miss represents the number of last-level cache misses across all NUMA nodes. IMC queue is the total duration of memory accesses to wait in the queue of the memory controllers. A higher value of this metric indicates a longer queuing delay caused by the congestion on memory controllers. The number of cache misses is also used to evaluate the impact of memory congestion because the congestion of memory access to LLC will increase the cache misses [69]. QPI volume is the volume of data sent through interconnects, which also represents the amount of remote accesses. A higher value of this counter indicates longer latencies from remote accesses. The random mapping is not included in this evaluation because the performance monitoring results of this mapping can significantly change for different executions.

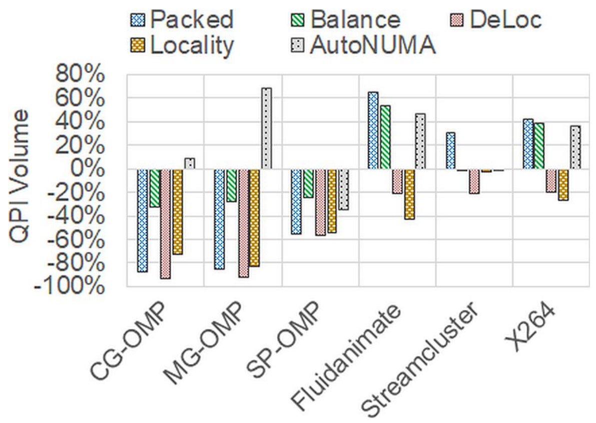
Figure 3.6 shows the performance monitoring results of the six applications, which are normalized with the results of Scatter mapping. In MG-OMP and SP-OMP, DeLoc can achieve the highest improvement by reducing LLC misses, IMC queue and the amount of remote accesses. The results of MG-OMP and SP-OMP show that DeLoc increases the locality of communication. Furthermore, by distributing the communication load over the NUMA nodes, DeLoc can reduce not only the congestion on memory controllers but also



(a) LLC misses.



(b) IMC queue.



(c) QPI volume.

Figure 3.6: Performance monitoring results of the NPB and PARSEC applications.

the congestion of memory access to LLCs. In CG, Packed shows a significant reduction in QPI volume because, as shown in Figure 3.5(b), CG has the communication behavior that can benefit from the Packed mapping. The results of IMC queue show a small difference among the methods. It means that in CG, the locality has a higher impact than memory congestion, and thus Packed and Locality show a higher performance improvement than Balance. On the other hand, DeLoc shows the lowest QPI volume and IMC queue, thus it can achieve the highest performance improvement among the methods.

In Fluidanimate, DeLoc and Locality show a lower LLC miss than that of the other methods because both methods improve the locality of communication. Scatter shows a lower IMC queue than Packed and DeLoc, and a lower QPI volume than Packed and Balance. It is because, as shown in Figure 3.5(d), this application has the communication behavior that can benefit from the Scatter mapping. In the case of Locality, although the number of cache misses is lower than that of Balance and Scatter, the IMC queue is much higher than that of Balance and Scatter. Thus, Locality shows a lower performance than Balance, DeLoc and Scatter. On the other hand, DeLoc can achieve the highest performance improvements by simultaneously reducing the memory congestion and the amount of remote accesses.

In Streamcluster, DeLoc gains the highest performance improvements by simultaneously reducing cache misses, IMC queue and QPI volume. Locality shows a lower LLC miss than Packed, Balance and Scatter. However, Locality shows the highest IMC queue. On the other hand, Balance and Scatter show a lower IMC queue and execution time than Locality. As discussed in Section 3.4.1.1, in Streamcluster, the impact of memory congestion is higher than that of the locality. The performance monitoring results show that maximizing the locality can degrade the performance of this application because it will significantly increase the memory congestion.

In X264, Locality shows the lowest QPI volume among the methods. However, Balance and DeLoc show the lowest IMC queue and the shortest execution time among the methods. As shown in Figure 2.9(d), X264 has a higher DRAM-to-memory ratio than most of the PARSEC applications. These results show that both methods can achieve a higher performance improvement than the other methods by significantly reducing the congestion on memory controllers. Note that Locality can achieve a lower IMC queue than Scatter, indicating that, in X264, improving the locality of communication can also reduce the communication load imbalance among the NUMA nodes. On the other hand, DeLoc can achieve the highest performance improvements from the reductions in IMC queue and QPI volume. It means that DeLoc can effectively reduce the congestion on memory controllers and the amount of remote accesses.

In CG-OMP, MG-OMP, and SP-OMP, AutoNUMA shows the lowest IMC queue, indicating that this method effectively reduces the memory congestion in these three applications. However, in all the six applications, AutoNUMA shows a higher QPI volume than those of DeLoc and Locality. The highest QPI volume is shown in MG-OMP, by 67.9% compared with Scatter. By migrating memory pages and threads to a different

Table 3.1: Simulation configuration.

Parameter	Value
NUMA Nodes (processors)	4x 16-core processors; L1I/L1D cache per core; L2 cache per core; L3 cache shared between 16 cores; 4x memory controllers; 5x bidirectional interconnects
Processor cores	2.4 GHz; Nehalem performance model
L1I/L1D caches	256 KB; 8-way; 64-byte line size; LRU policy
L2 caches	2 MB; 8-way; LRU policy
L3 caches	20 MB; 16-way; LRU policy
Memory controllers	60 ns latency; 36 GB/s bandwidth; 14-way interleave; DRAM directory model
Interconnects	25.6 GB/s bandwidth; network bus model

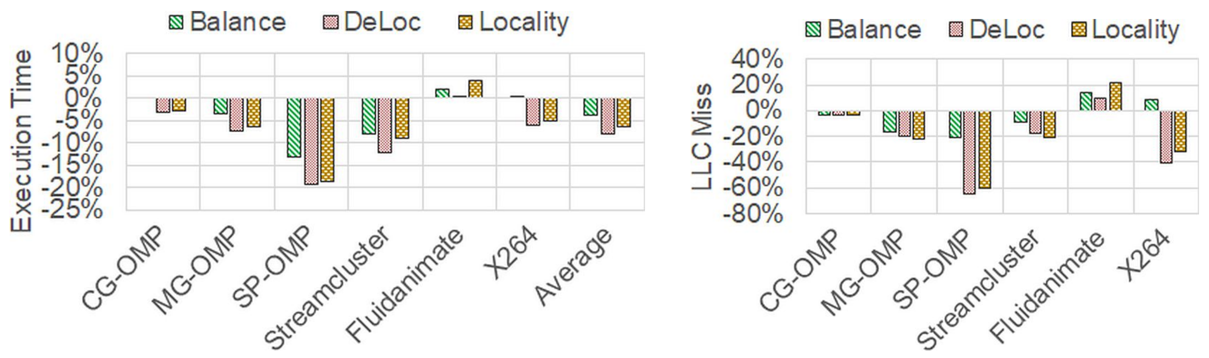
NUMA node, AutoNUMA potentially increases data traffic on interconnects because the threads may need to access data that reside in a remote NUMA node. These results suggest that the migration overhead has a significant impact on the volume of data traffic on interconnects. On the other hand, DeLoc and Locality do not suffer from the migration overhead, and thus show a lower QPI volume than that of AutoNUMA.

The performance monitoring results show that DeLoc gains performance improvements from the reductions in the LLC misses, IMC queue and QPI volume. Higher improvements are shown by the applications that have higher communication concurrency, communication-to-memory ratio and DRAM-to-memory ratio.

3.4.2 Performance Evaluation with a Simulator

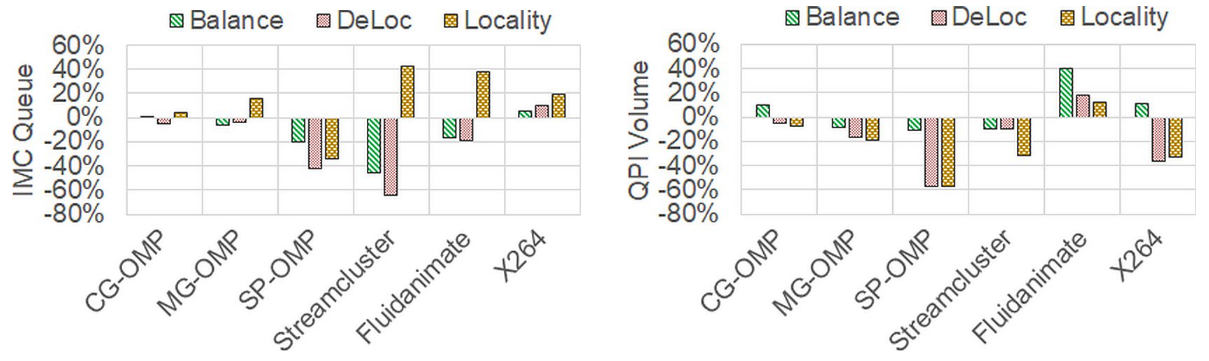
To evaluate the effectiveness of the proposed method on a larger-scale system, the experiments have been conducted in a multicore simulator, called Sniper [70]. A 4-node NUMA system is used for the simulation configuration, and the specifications of each processor, memory controllers, and QPI are set according to the hardware specifications of the Xeon56 system.

Table 1 shows the configuration parameters for the simulation. The simulation uses six applications that are used in the performance analysis of the real system evaluation (Figure 3.6). These applications are executed with 64 threads to use all the processor cores of the simulated system. The simlarge input sizes are used for the PARSEC applications.



(a) Execution time.

(b) LLC misses.



(c) IMC queue.

(d) QPI volume.

Figure 3.7: Performance results in the simulator.

For the NPB-OMP applications, class B input sizes are used for CG-OMP and MG-OMP, and the class A input size only for SP-OMP. The simulation uses smaller input sizes than the evaluation with a real system in Section 3.4.1 due to simulation time constraints. The simulation time drastically increases with the input size. For one CG-OMP execution, the simulation time with the class B input size is slower than that with the class A input size by two orders of magnitude.

In this evaluation, DeLoc is compared with Locality, Balance, and Scatter. These three methods are chosen because Locality and Balance consider the spatial communication behavior of the application, and in the real system evaluation, Scatter shows a higher performance improvement than Packed, Balance, Locality and AutoNUMA in the case of Fluidanimate. Moreover, in Fluidanimate, the performance difference between Scatter and Packed is the largest among the NPB-OMP and PARSEC applications.

Figure 3.7 shows the results of executing the six applications in the simulator, which are also normalized with the results of Scatter mapping. This evaluation uses the same performance metrics as in the real system evaluation. These metrics are obtained from the simulation output of the Sniper. The IMC queue and QPI volume metrics are obtained by measuring the counters of DRAM queuing delay and network packets in the simulator. On average, DeLoc can achieve the highest performance improvement among the methods, by up to 19.4% in the case of SP-OMP. In all the tested applications, except Fluidanimate, DeLoc and Locality show higher performance improvements than Scatter and Balance. The performance improvements are mainly obtained from the reductions in LLC misses and QPI volume, with the highest reductions exhibited in SP-OMP and X264.

In X264, the reductions of QPI volume shown by DeLoc and Locality are 36.4% and 32.9%, respectively. In CG-OMP, the QPI volume of Balance is higher than that of Scatter. In Streamcluster, the QPI volume of Balance is higher than that of Locality. These results are contrary to the results of the real system evaluation. These results suggest that on the simulated system, the impact of communication locality on the execution time is higher than that of the real system. Thus, reducing the amount of remote accesses becomes more effective in improving the performance of most of the applications.

In SP-OMP, DeLoc shows not only the lowest LLC misses and QPI volume but also the lowest IMC queue, and thus achieves the highest performance improvement among the methods. Moreover, the reductions of IMC queue and execution time are higher than those in the real system evaluation. This fact shows that on the simulated system, the impact of memory congestion on the execution time of SP-OMP is also higher than that in the real system. It is because the number of cores of each NUMA node in the simulated system is higher than that in the Xeon56 system.

In Fluidanimate, DeLoc and Scatter show the highest improvement among the methods. Scatter can achieve a comparable performance with DeLoc because this application has the communication behavior that can benefit from the Scatter mapping, which is also exhibited in the real system evaluation. These results show that the communication behavior of Fluidanimate does not change, even if the input size is changed. In this application, Locality show the highest IMC queue and LLC misses among the methods. By minimizing the amount of remote accesses, Locality increases the congestion of memory access to the LLC and memory controllers. On the other hand, DeLoc can achieve a shorter execution time than Balance and Locality due to the reductions in both the memory congestion and the amount of remote accesses.

As discussed in Section 3.3.3, reducing memory congestion may increase the amount of remote accesses. In Streamcluster and Fluidanimate, DeLoc shows a higher QPI volume than Locality because, to reduce memory congestion, DeLoc distributes the concurrent communications over the NUMA nodes. However, as also discussed in the real system evaluation, the memory congestion has a high impact on the execution time of Streamcluster and Fluidanimate. Thus, in these applications, DeLoc can still achieve shorter execution times than those of Locality.

The simulation results show that, in most of the tested applications, the impact of communication locality on the execution time increases with the number of NUMA nodes. The applications that have higher communication-to-memory ratio and communication locality, such as SP-OMP and X264, will gain a higher performance improvement from locality-based task mapping. DeLoc can achieve the highest performance in most of the applications by simultaneously reducing the amount of remote accesses and memory congestion.

3.5 Conclusions

This chapter has proposed a task mapping method, DeLoc, to address both the locality and the memory congestion problems. DeLoc uses information about the NUMA node topology, and also spatial and temporal communication behaviors of the applications to optimize the task mapping. To analyze the communication behaviors of parallel applications, it uses DeLocProf, which is proposed in Chapter 2. In addition, a mapping algorithm, called DeLocMap, has been introduced to calculate the mapping that can simultaneously improve the locality and reduce memory congestion. The algorithm can effectively reduce memory congestion by identifying the concurrent communications that potentially cause the memory congestion.

DeLoc has been extensively evaluated on a real NUMA system and multicore simulator. For the discussion, DeLoc has been compared with a dynamic mapping method, two greedy-based methods, a random method, a balance-based method, and a locality-based method. On the real system, it has been evaluated with a wide range of parallel applications from three sets of parallel benchmarks based on MPI, OpenMP and Pthreads implementations. The experiments have also been conducted on the simulator to evaluate DeLoc on a larger number of NUMA nodes.

The evaluation results show that DeLoc consistently outperforms the other methods, suggesting that the mapping method that considers both locality and memory congestion is more effective in improving the performance of parallel applications on NUMA systems. DeLoc achieves performance improvements by up to 46.5%, 61%, and 14.3% compared with the dynamic mapping, locality-based mapping, and the balance-based mapping methods, respectively. Higher performance improvements have been shown in the applications that have higher communication concurrency, communication-to-memory ratio and DRAM-to-memory ratio. The performance improvements are obtained from the reductions in LLC misses, queuing delay in memory controllers, and data traffics in interconnects.

Chapter 4

Online Task Mapping for Coordinating Locality and Memory Congestion

4.1 Introduction

Currently, multicore processors are widely used not only for shared-memory parallel processing but also for distributed-memory parallel processing, such as MPI [31]. In parallel applications based on MPI, a task is executed by a processor core as an MPI process, and thus MPI process mapping is also referred to as task mapping [39]. 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 its corresponding processes of a sender and a receiver. This pair is also referred to as a process pair or a task pair [12, 29].

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. Since the latency of remote memory access is higher than that of local memory access, the cost of the later communication higher than that of the former communication. MPI provides extensions that enable faster intra-node communication through the use of shared memory, such as Nemesis for MPICH2 [71] and KNEM [72] for Open MPI [43]. 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 and reduce energy consumption. Such process mapping methods are called *communication-aware process mapping* [20, 29].

Related work on MPI process mapping mostly focuses only on improving the locality to minimize the overall cost of communication. However, as shown in Section 3.4, considering both the locality and the memory congestion is necessary to achieve the best performance on NUMA systems. To optimize the process mapping, it is necessary to analyze the communication behavior of the MPI applications. Most of existing 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 [73].

The objective of this chapter is to realize task mapping for coordinating locality and memory congestion on NUMA systems without the extensive profiling and analysis. This chapter focuses on task mapping for parallel applications based on MPI, where task mapping is represented by process mapping. To achieve the objective, this chapter proposes a process mapping method, called *online decongested locality for MPI* (OnDeLoc-MPI), that can address the locality and the memory congestion problems on recent NUMA systems without the needs of offline profiling and analysis. It consists of a mechanism that dynamically performs the process mapping for adapting to changes in the communication behaviors, and a mapping algorithm, called OnDeLocMap+, to calculate the process mapping that can simultaneously reduce the amount of remote memory accesses and the memory congestion. OnDeLoc-MPI works online during the execution of an MPI application and does not require prior knowledge of the communication behaviors, modifications to the application, or changes to the hardware and operating system.

There are two key differences between DeLoc and OnDeLoc-MPI. First, OnDeLoc-MPI analyzes the spatial communication behavior by monitoring the communication events during the execution of the application. It does not need to trace the communication events by preliminarily running the application. Second, it dynamically updates the mapping to adapt to the temporal changes of the communication behavior. Thus, it does not need to analyze the temporal communication behavior prior to the execution. Because of these two differences, OnDeLoc-MPI does not impose overheads from the offline profiling and analysis.

The main contribution of this chapter is the OnDeLoc-MPI that can address the locality and memory congestion problems on NUMA systems. It includes the OnDeLocMap+ algorithm to calculate the mapping that can coordinate the locality and memory congestion without extensive analysis of the communication behaviors. Moreover, this chapter clarifies the impacts of task mapping on the performance and energy consumption of parallel applications on NUMA systems.

The rest of the chapter is organized as follows. Section 4.2 reviews the related work. Then, the procedure and implementation of OnDeLoc-MPI are described in Section 4.3. Experimental setup and results are presented in Section 4.4. This section also discusses the overhead of OnDeLoc-MPI. Finally, conclusions and future work are summarized in Section 4.5.

4.2 Related Work

This section reviews the related work of this chapter. First, it reviews existing methods for MPI process mapping. Then, it reviews existing online methods for thread and process mapping. Finally, this section reviews existing online methods for thread and data mapping.

4.2.1 MPI Process Mapping

Various MPI process mapping methods have been proposed in related studies. Most of the methods rely on offline profiling to trace communication among processes and to analyze the communication behaviors of the applications [11, 12, 24, 49]. 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.

4.2.2 Online Thread and Process Mapping

An online communication detection method, called CDSM, has been proposed in a related study [2]. It works at the operating system level during the execution of a parallel application. It detects the communication behavior from page faults and uses this information to perform the process and thread mapping dynamically. CDSM has focused on improving the locality to minimize the communication cost. The evaluation results of the related work had 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 any communication detection mechanisms 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 4.4, the benefits of OnDeLoc-MPI are discussed through performance comparison of the two methods.

4.2.3 Online Thread and Data Mapping

A memory placement method, called Carrefour, has been proposed in [13]. It improves performance on recent 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. [14] 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 recent NUMA systems. It relies on a continuous sampling of the memory accesses to analyze the communication among threads.

Both Carrefour and AsymSched require a sampling mechanism that is available only on AMD processors. In contrast to these methods, OnDeLoc-MPI analyzes the communication behavior directly from MPI communication events and does not require the communication detection and memory sampling mechanisms. Moreover, OnDeLoc-MPI works at the runtime system level and does not rely on a specific hardware or operating system. Compared with AsymSched, OnDeLoc-MPI focuses on reducing not only memory congestion but also the amount of remote accesses. As discussed in Section 3.4.1, reducing the amount of remote accesses can substantially improve performance.

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

This section proposes OnDeLoc-MPI that can address the locality and memory congestion problems on NUMA systems. It first explains the procedure of OnDeLoc-MPI, and then

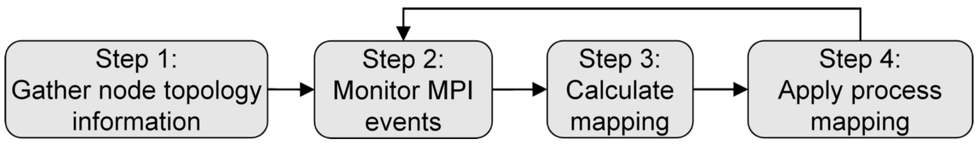
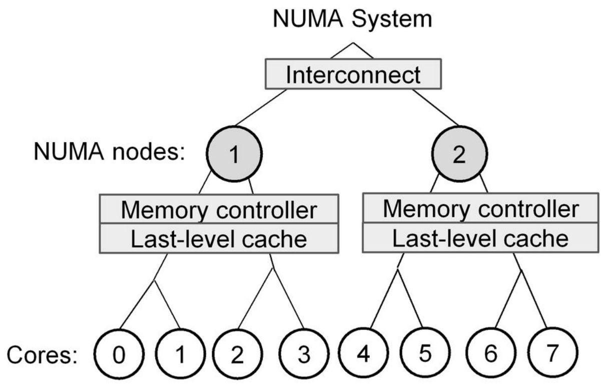
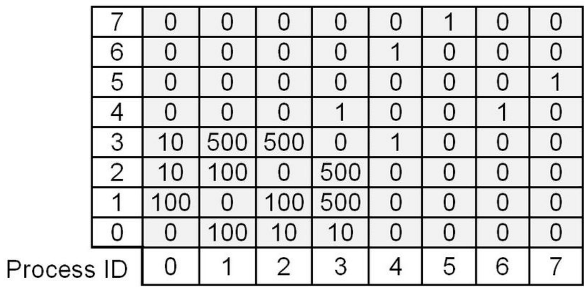


Figure 4.1: The procedure of OnDeLoc-MPI.



(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 4.2: Examples of the NUMA node topology and the communication matrix.

describes the implementation of the method in the MPI runtime system.

4.3.1 Procedure of OnDeLoc-MPI

Figure 4.1 shows the procedure of OnDeLoc-MPI, which consists of four steps:

Step 1 Gather the node topology information of the NUMA system.

Step 2 Monitor MPI communication events during the execution.

Step 3 Calculate the MPI process mapping.

Step 4 Apply the process mapping.

In Step 1, 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 LLC and an integrated memory controller, such as Intel-based and AMD-based NUMA systems [4]. Thus, the location of the NUMA node represents the locations of both 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 4.2(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.

In Step 2, during the execution of the application, the communication behavior of the application is analyzed by monitoring the communication events among MPI processes. The communication matrix is used to model the communication behavior, which is described in Section 2.3. 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 4.2(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, OnDeLoc-MPI updates 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 Steps 3 and 4, 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 4.3.2. Then, in Step 4, the calculated mapping is applied to the execution by assigning processor cores to

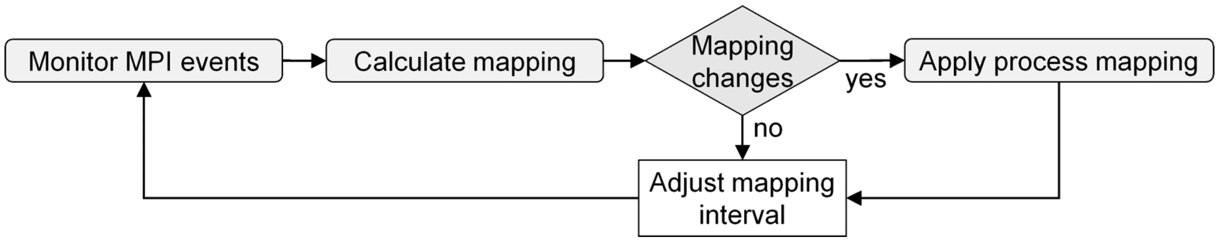


Figure 4.3: The mechanism of mapping interval adjustment.

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 Figure 4.1, Steps 2 to 4 are performed iteratively during the execution of the application. Since the mapping interval determines the number of iterations, it can significantly affect the performance results and overhead of OnDeLoc-MPI. 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. The overhead of OnDeLoc-MPI is discussed in more detail in Section 4.4.4.

To reduce the overhead, OnDeLoc-MPI dynamically adjusts the mapping interval, which is shown in Figure 4.3. A slope parameter, v, is used 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 (4.1). 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 (4.2). The mapping interval is shortened so that the mapping can adapt more quickly to changes in the communication behavior.

$$Interval_{curr} = Interval_{prev} \times v, \tag{4.1}$$

$$Interval_{curr} = \frac{Interval_{prev}}{v}. \tag{4.2}$$

4.3.2 Implementation of OnDeLoc-MPI

OnDeLoc-MPI has been implemented as a module for Open MPI runtime system [43]. The implementation consists of four parts:

Part 1 A modification to the monitoring layer of the runtime system.

Part 2 Data consolidation among MPI processes.

Part 3 OnDeLocMap+ algorithm.

Part 4 Data structures to store the communication matrices and process mapping.

4.3.2.1 Modification of the Runtime System

For the implementation, a module has been added to 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, two functions have been implemented in the module. The first function obtains the node topology information of the system by using Hwloc library [63]. This library is used 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 [42] that is built on top of the PML of the Open MPI stack [43]. OnDeLoc-MPI uses 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 P2P and collective communications.

For Steps 3 and 4, a thread, called *mapper thread*, is created to periodically calculate and apply 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 [43, 47]. 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

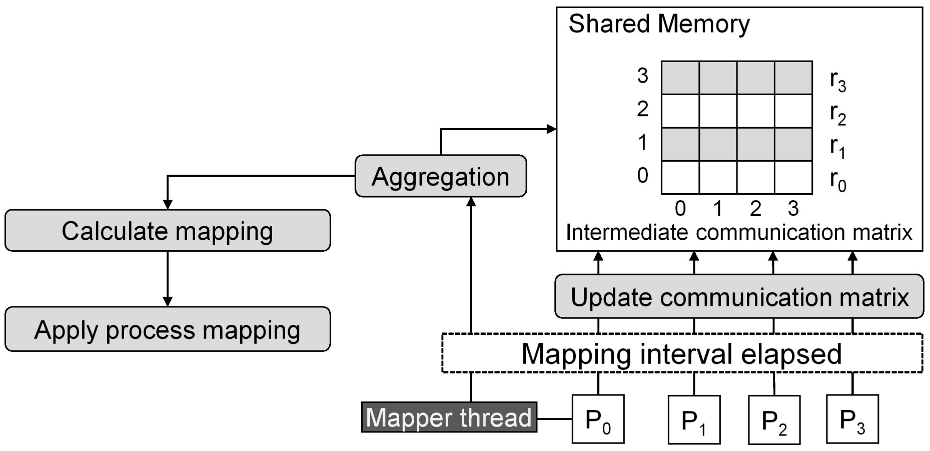


Figure 4.4: The consolidation mechanism with four MPI processes.

process mapping, OnDeLoc-MPI assigns the target processor cores to processes according to the mapping by using the `sched_setaffinity()` function call of the Linux system. For the implementation, the use of this function is not mandatory. Alternatively, some libraries such as Hwloc-bind [63] and Likwid-pin [65] can be used to assign processor cores to processes.

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 task mapping because migrating MPI processes across systems potentially incurs a significant network overhead [74].

4.3.2.2 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 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 Figure 4.4. For the consolidation purpose, OnDeLoc-MPI stores 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. OnDeLoc-MPI uses the POSIX shared memory API [45] 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, OnDeLoc-MPI also uses 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, a cell (x, y) is updated 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.

4.3.2.3 OnDeLocMap+ Algorithm

The OnDeLocMap+ algorithm is depicted in Algorithm 2. As shown in the evaluation results of Chapter 3 and the related work [27], the migration overhead has a significant impact on the performance of the applications. To reduce this overhead, the OnDeLocMap+ algorithm prevents unnecessary process migrations by considering the current mapping to calculate the next mapping. It gives a higher priority to processes that have a larger amount of communication to be mapped to the same NUMA node of the current mapping. The algorithm is detailed as follows.

First, OnDeLocMap+ uses a topology model to construct the map between processor

Algorithm 2 The OnDeLocMap+ Algorithm.

```
Input: $T$ {The node topology tree}
Input: $A$ {The communication matrix}
Input: $CurrentM$ {The current mapping}
Output: $M$ {The map of processor core IDs and process IDs}
1: $M \leftarrow createMap(T)$
2: $Pairs \leftarrow generatePairs(A)$
3: $sortedPairs \leftarrow sortByAcomm(Pairs)$
4: $i \leftarrow 0$
5: while $i < num(Pairs)$ and $numUnmappedCores(M) > 0$ do
6:    $pair \leftarrow sortedPairs[i]$
7:    $nodesOfCurrentM \leftarrow getNodesOfCurrentM(pair, CurrentM)$
8:    if $isNodesAvailable(nodesOfCurrentM)$ then
9:       $mapPair(pair, nodesOfCurrentM)$
10:   else
11:      $mapPair(pair, nextNodeAvailable(T))$
12:   end if
13:   $i \leftarrow i + 1$
14: end while
```

core IDs and process IDs (Line 1). The keys of the map represent the IDs of processor cores available 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. The `mapPair()` function maps the processes to the processor cores of the NUMA node that is available in a round-robin fashion (Line 11). The algorithm aims 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, when selecting the target NUMA nodes for a process pair, OnDeLocMap+ first evaluates the NUMA nodes that are currently mapped for the same pair (Lines 7-8). The function `getNodesOfCurrentM()` will return two NUMA nodes, where each node is associated with each process of the pair. If the current NUMA nodes are available, it will map each process of the pair to the processor cores of the current 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 for offline mapping in Section 3.3.3.

Compared with the DeLocMap algorithm (Section 3.3), OnDeLocMap+ is simpler and faster because it does not need to process the result of the data clustering method in the mapping calculation. In the case of DeLoc, the mapping is static, which is that there is only one mapping during the whole execution of the application. Thus, to obtain the temporal communication behavior of the whole execution of an application, DeLoc statically analyzes the communication traces of the application using a data clustering method. Then, the result of the data clustering method is used by the mapping algorithm to represent the temporal communication behavior of the application. In contrast, OnDeLoc-MPI works at runtime during the execution of an MPI application and needs to dynamically change the process mapping to adapt to the changes in the communication behavior of the application. Thus, OnDeLocMap+ can focus only on optimizing the process mapping for the latest communication behavior of the application. Both DeLocMap and OnDeLocMap+ use a tree structure to represent the NUMA node topology of the system. Thus, OnDeLocMap+ can also be applied to any NUMA node topologies that can be represented as a tree.

4.3.2.4 Data Structures

For each MPI application, OnDeLoc-MPI allocates 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, a key-value map is allocated 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.

4.4 Evaluation

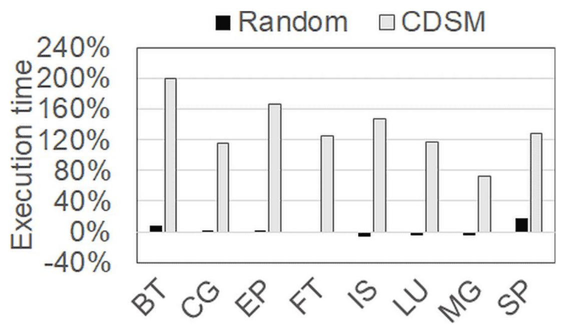
This section presents the experimental evaluations of OnDeLoc-MPI. The main evaluation consists of three parts: performance, energy consumption, and overhead. In addition, this section discusses the evaluation with a larger NUMA system.

4.4.1 Experimental Setup

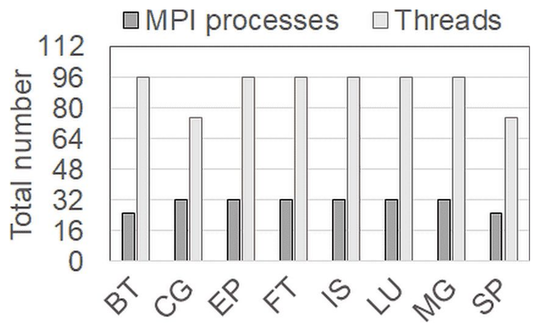
The main experiments have been conducted on a NUMA system, named Xeon2, that consists of two NUMA nodes and one Intel Xeon E5-2690 processor per NUMA node. Each processor has private L1 and L2 caches and an L3 caches as a LLC that is shared among all cores of of the NUMA node. The sizes of L1, L2, and L3 caches are 64 KB, 256 KB, and 20 MB, respectively. The system has 32 logical cores in total and runs Linux OS kernel v3.2. The NUMA nodes are connected with QuickPath Interconnect (QPI) [8], and each NUMA node has 16 logical cores and an Integrated Memory Controller (IMC). As workloads, eight applications of the NPB-MPI v3.4 are executed with the class C input size. All the applications except BT-MPI and SP-MPI are executed with 32 processes using all cores in the system. BT-MPI and SP-MPI require a square number of processes, and thus these two applications are executed with 25 processes. Open MPI v3.1 is used as the MPI runtime system for the main experiments.

In all the experiments, the mapping interval is kept 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. 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. This analysis step will incur a high overhead [73], and thus is not applied in this evaluation.

In the main experiments, OnDeLoc-MPI is compared with an online-based thread mapping method and two static mapping methods. In static mapping [39], one process



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



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

Figure 4.5: Evaluation results with CDSM.

mapping is used for the whole execution of the application. The mapping is obtained prior to the execution, and is not changed during the execution of the application. The static mapping methods used in the experiments are Default and Static-best. Default is the original mapping of the MPI runtime system, and represents the baseline of these experiments. Default corresponds to the Scatter mapping used in the evaluation of Chapter 3. It maps the neighboring processes to processor cores of different NUMA nodes in a round-robin fashion. As discussed in Section 3.4, this mapping method can improve the balance of communication when most communications are performed among neighboring processes.

Static-best mapping is obtained by comparing the performance results of Balance, Locality and DeLoc mapping methods. These three methods are described and examined in Chapter 3. As shown in the previous chapter, DeLoc achieves the highest performance among the static mapping methods. Thus, in this evaluation, DeLoc is used for obtaining the Static-best mapping. DeLoc first collects communication traces by profiling the target application. Then, it analyzes the spatial and temporal communication behaviors of the application and calculates the mapping using the DeLocMap algorithm. The profiling and analysis are performed offline prior to the execution of the application. Thus, Static-best mapping imposes a high overhead caused by the offline profiling and analysis.

The online-based mapping method used for the evaluation is CDSM-mod, which is a modified version of CDSM. CDSM is modified because the evaluation of this chapter shows that it significantly degrades the performance of all the tested applications, which are contrast with the results shown in [2]. Figure 4.5(a) shows the performance results of CDSM, compared with the baseline and the Random mapping method. For the Random mapping, a process mapping is randomly generated before each execution. As shown in the figure, CDSM shows longer execution times for all the applications. In BT, CDSM can increase the execution time by a factor of two compared with Default. It is 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 [75]. Thus, the total number of threads spawned by the runtime system can be higher than the number of MPI processes.

Figure 4.5(b) shows the total number of threads executed for the NPB applications with 32 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 numbers 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, the method is modified by including only the parent threads of the MPI processes in the steps of detecting communication and calculating the thread mapping. These parent threads are detected from the first n threads created at the beginning of the execution, where n is equal to N_p.

To compare the performance results of OnDeLoc-MPI with the evaluation results of Chapter 3, the experiments have also been conducted on the Xeon56 system, which is the same system as used in the previous evaluation. In these experiments, the input sizes, the number of processes, and the runtime system are the same as used in the previous evaluation. The discuss the experimental results on Xeon56, OnDeLoc-MPI is compared with all the methods used in the previous evaluation.

The Xeon56 system is not used for the main experiments because of two reasons. First, CDSM cannot be executed on the Xeon56 system because the method relies on a previous

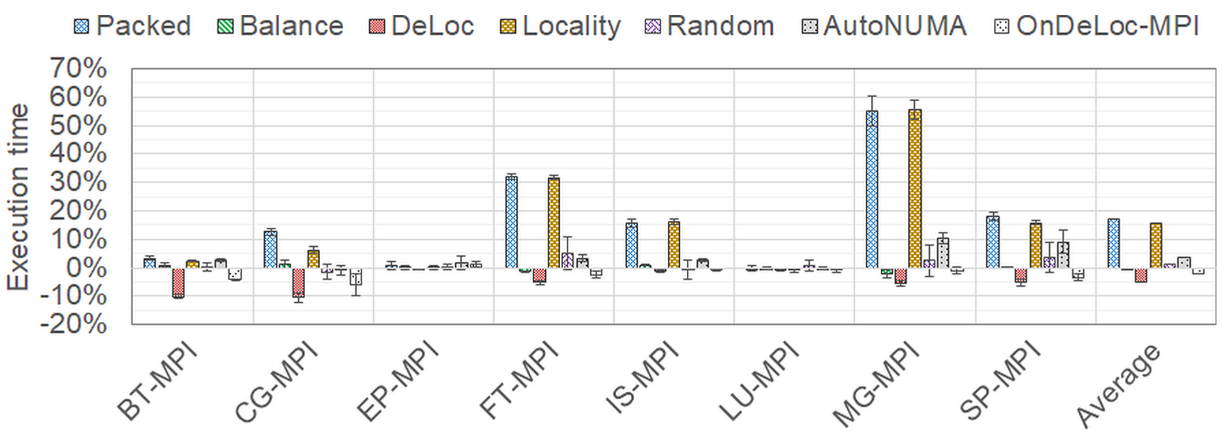


Figure 4.6: Performance results on Xeon56.

version of the Linux kernel. Second, core and Uncore energy consumptions cannot be measured due to the limitation of the Running Average Power Limit (RAPL) hardware counters [76] on the Xeon56 system. Measuring the core and Uncore energy is necessary to evaluate the impacts of online task mapping methods on the energy consumption of processor core, memory controllers, and interconnects. The results of core and Uncore energy consumptions are discussed in more detail in Section 4.4.3.

4.4.2 Performance Evaluation

Figures 4.6 and 4.7 show the performance results on the Xeon56 and Xeon2 systems, respectively. The performance results are obtained by measuring the execution time of the applications with each mapping method. All the experimental results are the averages obtained from 10 sample executions, which are normalized with the results of the baseline mapping. The figure also shows the 95% confidence interval calculated with Student's t-distribution The error line of the bar represents the confidence intervals of the samples.

On the Xeon56 system, OnDeLoc-MPI shows a higher average performance than all the other methods, except DeLoc. Compared with the locality-based and Default methods, OnDeLoc-MPI achieves 18% and 2% average performance improvements, respectively. In most of the applications, OnDeLoc-MPI shows a lower performance than DeLoc. However, the performance differences between the two methods are small, which is 2.6% on average. These results suggest that both DeLoc and OnDeLoc-MPI can achieve the high-

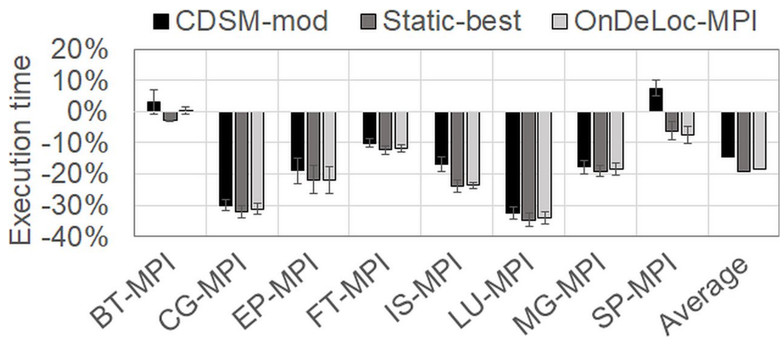


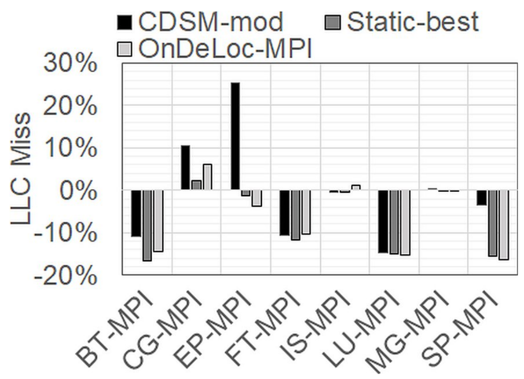
Figure 4.7: Performance results on Xeon2.

est performance improvement in most of the applications by coordinating locality and memory congestion.

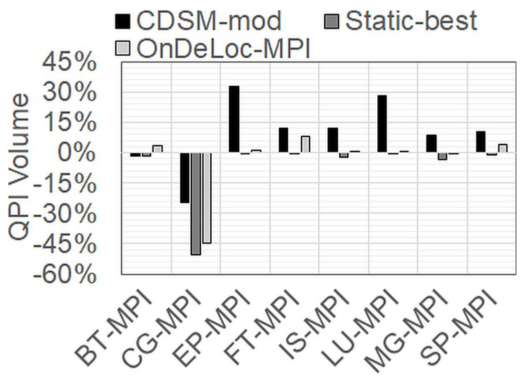
On the Xeon2 system, OnDeLoc-MPI shows a higher performance than Default and CDSM-mod in most of the applications. Compared with Default, the average performance improvement of OnDeLoc-MPI is 18.5%. The highest performance improvements are exhibited in CG-MPI and LU-MPI by 31.2% and 34%, respectively. 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-MPI and SP-MPI, 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 the proposed method can achieve comparable performance to Static-best even without any kind of extensive profiling and analysis. Moreover, in SP-MPI, OnDeLoc-MPI achieves the highest performance improvement among the methods. These results indicate that in the case of SP-MPI, the static mapping is not sufficient to take into account the temporal changes of the communication behavior.

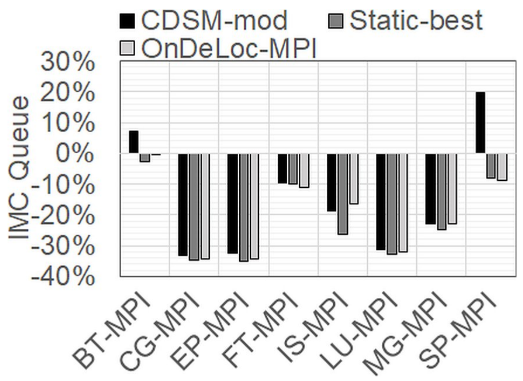
To investigate the sources of performance improvements, the performance characteristics of the NPB applications are evaluated. Three metrics are used for the evaluation: LLC miss, QPI volume and IMC queue metrics. These metrics are obtained by monitoring the Intel performance counters [68]. LLC miss represents the number of last-level cache



(a) LLC miss.



(b) QPI volume.

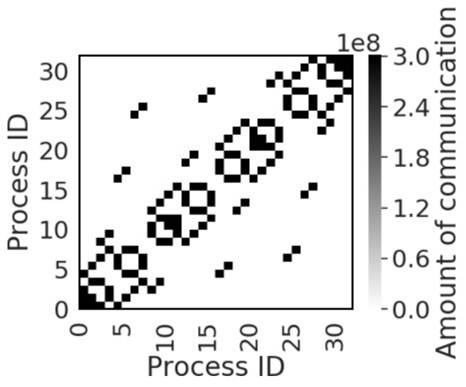


(c) IMC queue.

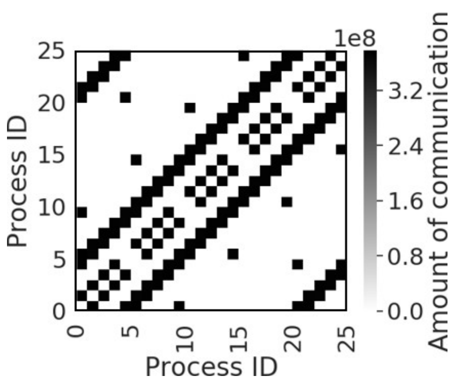
Figure 4.8: Performance monitoring results on Xeon2.

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.

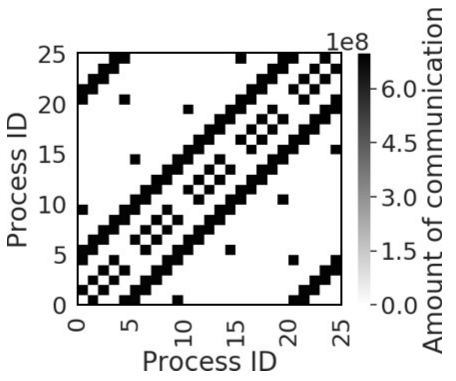
Figures 4.8(a), 4.8(b) and 4.8(c) show the results of LLC misses, QPI volume and IMC queue, respectively. These figures show that most of the applications gain a substantial performance improvement from reductions in the caches misses and IMC queuing delay. It means that, on the Xeon2 system, the memory congestion has a significant impact on the performance of most of the NPB-MPI applications. Moreover, in BT-MPI and SP-MPI, CDSM-mod increases the IMC queuing delay, and in most of the applications, it shows a higher IMC queue than OnDeLoc-MPI and Static-best. This fact suggests that only considering the locality is not sufficient to achieve the best performance for these



(a) CG-MPI.



(b) BT-MPI.



(c) SP-MPI.

Figure 4.9: Communication behaviors of CG-MPI, BT-MPI and SP-MPI.

applications.

Figures 4.9(a), 4.9(b), and 4.9(c) show the communication matrices of CG-MPI, BT-MPI and SP-MPI, respectively. The x-axis and y-axis show the process ID, and each cell represents the amount of communication between two processes of the corresponding axes. The values of amount of communication are in bytes, and the darker cells indicate a larger amount of communication. In CG-MPI, OnDeLoc-MPI shows significant reductions in interconnect traffic and IMC queuing delay, indicating that both the locality and the memory congestion have a significant impact to this application. As shown in Figure 4.9(a), groups of processes have a higher amount of communication compared with processes outside the group. These results show the effectiveness of the OnDeLocMap+ algorithm to simultaneously reduce the amount of remote accesses and the memory congestion.

In the cases of BT-MPI and SP-MPI, the performance differences among the methods are smaller than those in the other applications. It is because these two applications have the communication behavior that can benefit from the Default mapping. As shown in their communication matrices (Figures 4.9(b) and 4.9(c)), most communication events are performed by the neighboring processes. 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 (Figure 4.8(b)) and a longer execution time (Figure 4.7) than 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 by Static-best. Furthermore, even for the applications that cannot gain a significant performance improvement from the Static-best mapping, such as BT-MPI, 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, and OnDeLoc-MPI can effectively reduce this overhead.

On the Xeon2 system, most of the NPB-MPI applications gain a significant performance improvement from task mapping. These results are contrast with the performance results on Xeon56, where some applications, such as EP-MPI and LU-MPI, cannot gain a significant performance improvement from task mapping. The results on the two systems are different because the size of LLC of Xeon2 is smaller than that of Xeon56. As a result, the applications access DRAM more frequently on Xeon2 than that on Xeon56. As shown by the performance monitoring results on Xeon2, OnDeLoc-MPI and Static-best significantly reduce the LLC misses and queuing delay in the memory controllers for most of the applications. This fact shows the importance of DRAM-to-memory metric to evaluate the impacts of task mapping on the performance of the applications.

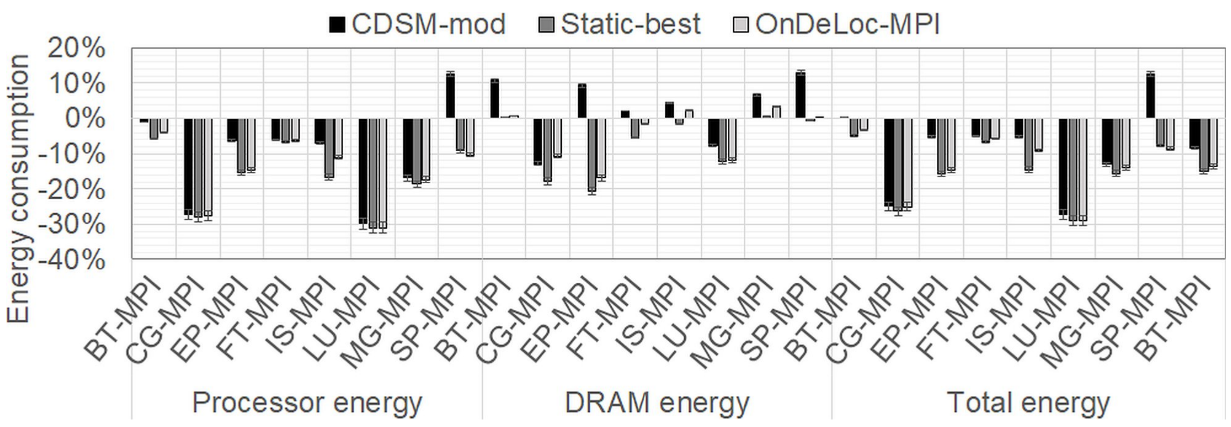


Figure 4.10: Energy consumption results on Xeon2.

4.4.3 Energy Consumption Evaluation

This section presents and discusses the energy consumption results of the NPB applications on the Xeon2 system. The processor energy and DRAM energy are measured by using the RAPL hardware counters. 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 a processor core, which include QPI and memory controllers [77]. To evaluate the energy consumption of processor core and Uncore components, the processor energy consumption is decomposed into core and Uncore energy consumptions. The core and Uncore energy are also measured using the RAPL counters.

Figure 4.10 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 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-MPI and MG-MPI, 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

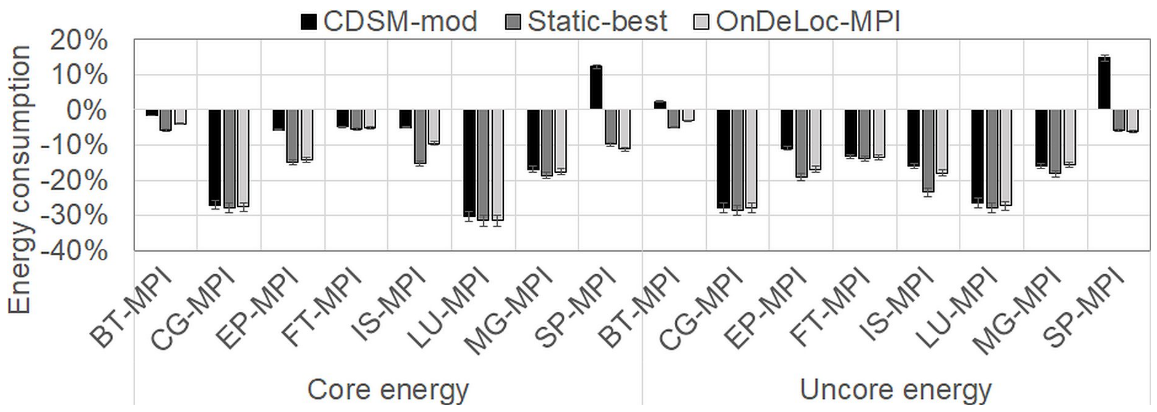


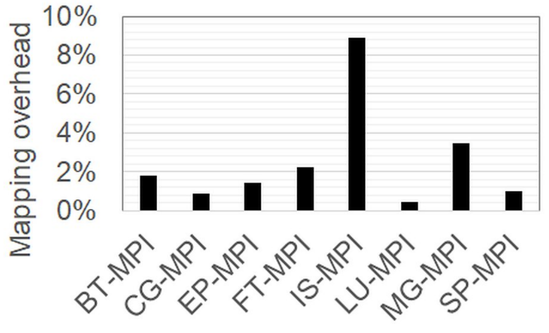
Figure 4.11: Core and Uncore energy consumptions on Xeon2.

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.

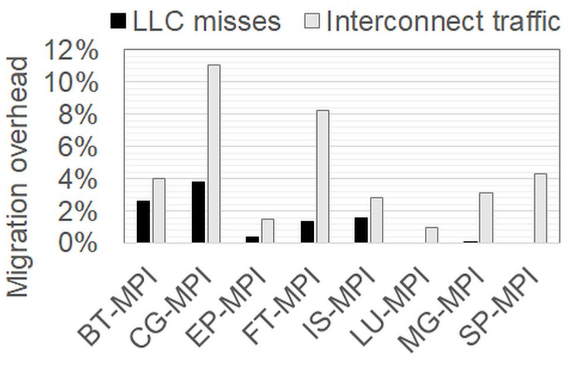
Figure 4.11 shows the results of core and Uncore energy consumptions. In all the NPB applications, OnDeLoc-MPI shows lower core and Uncore energy consumptions than CDSM-mod and Default. The core energy consumption is reduced most in LU-MPI, with a reduction of 31.5%, and the Uncore energy consumption is reduced most in CG-MPI, 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 are caused by the increases in execution time and interconnect traffic, respectively. Moreover, in BT-MPI and SP-MPI, CDSM-mod shows the highest Uncore energy because, as shown in Figures 4.8(b) and 4.8(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-



(a) Mapping overhead.



(b) Migration overhead.

Figure 4.12: Overhead of OnDeLoc-MPI.

mod, and Static-best reduce the static energy consumption in most of the applications. However, as shown in BT-MPI and SP-MPI, OnDeLoc-MPI can achieve lower core and Uncore energy consumptions than Default and CDSM-mod without significantly changing 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.

4.4.4 Overhead of OnDeLoc-MPI

OnDeLoc-MPI incurs runtime overhead because it works during the execution of the MPI application. 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, the overhead of OnDeLoc-MPI is evaluated on the Xeon2 system.

The mapping overhead is shown in Figure 4.12(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 percentages of the execution times. For all the applications, the mapping overhead is less than 9%, and the average overhead of the mapping is low of only about 2.5%. IS-MPI 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-MPI for updating the communication matrices and calculating the mapping is less significant than those in the other applications. The results of evaluating the 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. Note that this evaluation does not include the mapping overhead. The functions of OnDeLoc-MPI that update the communication matrix and compute the mapping are disabled. The process mapping for each interval is provided offline prior to the execution, and obtained by preliminarily running each application with OnDeLoc-MPI. Therefore, in this evaluation, only the migration overhead affects the execution of each application.

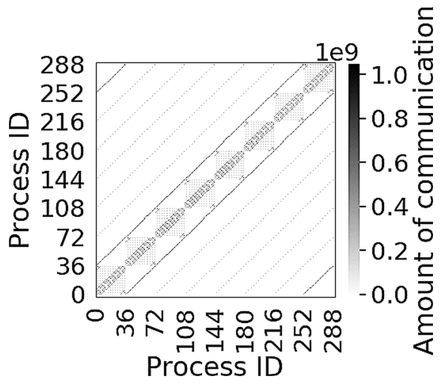
Figure 4.12(b) shows the migration overhead on the cache misses and interconnect traffic. The number of cache misses is obtained by aggregating the cache misses of all cache levels across the NUMA nodes. For all the applications, the migration overhead is less than 11%, and the highest overheads on the interconnect traffic are imposed in CG-MPI and FT-MPI. As shown in the communication matrix of CG-MPI (Figure 4.9(a)), it 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-MPI and CG-MPI show the highest overhead to LLC misses, 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-MPI and FT-MPI. OnDeLoc-MPI performs less migrations 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-MPI, CG-MPI, and FT-MPI 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 than the 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-MPI, the benefit of online mapping surpasses the overhead of OnDeLoc-MPI, and thus OnDeLoc-MPI can achieve a higher performance than Static-best.

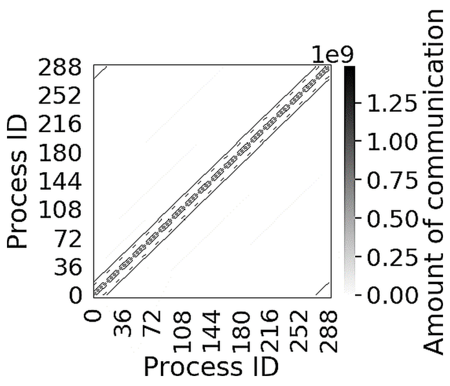
4.4.5 Evaluation on a Larger System

The evaluation results on the Xeon2 system have demonstrated the effectiveness of OnDeLoc-MPI in improving the performance and energy consumption of the applications. However, when the number of NUMA nodes becomes larger, the performance and energy consumption improvements of an online mapping method can decrease significantly because of two main reasons. First, as shown in the evaluation of Chapter 3, the impacts of locality on the performance of the application increases with the number of NUMA nodes. Second, the increase in the number of NUMA nodes may also increase the migration overhead.

To evaluate OnDeLoc-MPI with larger numbers of NUMA nodes and processor cores, the experiments have been conducted on a NUMA system, named KNL4. The system is based on Intel Xeon Phi Knights Landing (KNL) processor [78]. It has 288 logical cores in total, and is running Linux kernel v4.4. It is configured 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. The Open MPI v3.1 is also used as the MPI runtime system for this evaluation.



(a) Test Case A.

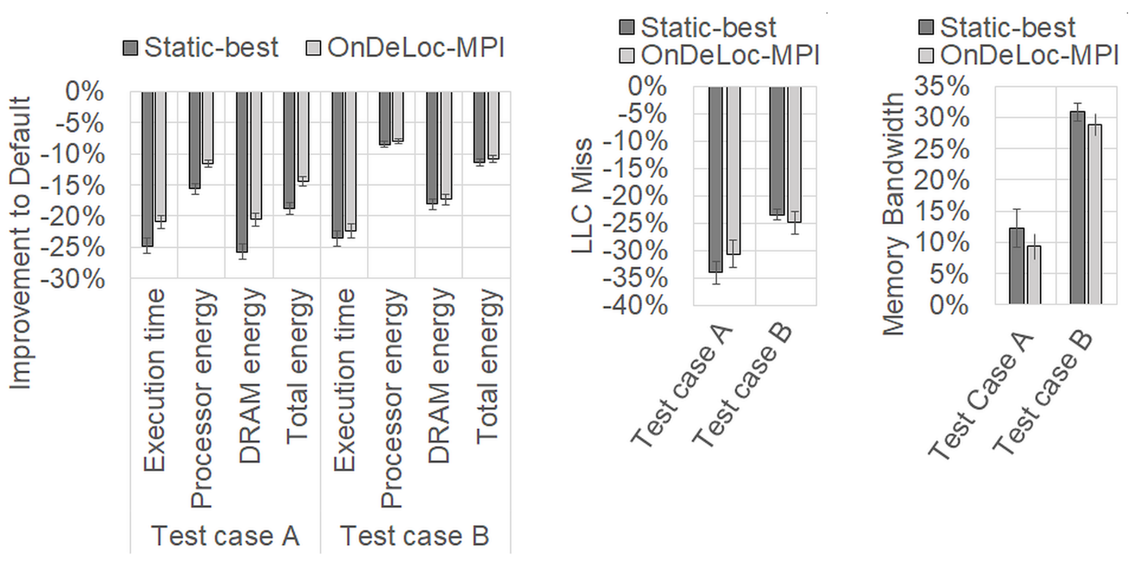


(b) Test Case B.

Figure 4.13: Communication behaviors of the GROMACS applications.

As the workloads, this evaluation uses two biomolecular applications of the UAEBS benchmarks [79,80]. The applications are Test Case A and Test Case B of the benchmarks that are based on GROMACS simulation [81]. The communication behaviors of these applications are shown by the communication matrices in Figure 4.13. The values of amount of communication are in bytes and shown in E notation [67]. The darker cells indicate a larger amount of communication. These matrices are obtained by OnDeLoc-MPI at the end of application execution. These two applications are executed with 288 MPI processes to use all the processor cores of the system. This evaluation does not use NPB applications because all the applications except EP-MPI cannot be executed with this number of processes. OnDeLoc-MPI is compared with Default and Static-best mapping. CDSM-mod cannot be evaluated because its limitation to the previous version of Linux kernel. This fact highlights the advantage of implementing OnDeLoc-MPI at an application level: it does not depend on specific features of the hardware and operating system.

Figure 4.14(a) shows the results of performance and energy consumption on the KNL4 system. The Uncore energy cannot be measured in this evaluation because of the limitation of the RAPL counters on the KNL. However, the impacts of the mapping methods on the total energy consumption can still be evaluated 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



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

Figure 4.14: Performance and energy consumption results on KNL4.

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 than Static-best. However, the performance and energy consumption results of these two methods are almost the same. 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 a 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 numbers of processes and NUMA nodes become larger.

For further evaluation, the LLC misses and memory bandwidth metrics are measured for each execution of the applications. Memory bandwidth is the bandwidth of memory accesses to the memory controllers. This evaluation 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 a higher memory congestion leads to a lower memory access bandwidth.

Figures 4.14(b) and 4.14(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 LLC 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 average difference in cache misses between the two methods is 3.37% in Test Case A, while the average difference in memory bandwidth is 2.9% and 2.1% in Test Cases A and B, respectively.

4.5 Conclusions

In this chapter, a process mapping method, called OnDeLoc-MPI, has been proposed to address the locality and the memory congestion problems on NUMA systems. OnDeLoc-MPI 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 Static-best 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. The 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 an overhead from the mapping calculation and process migration. To reduce the overhead, OnDeLoc-MPI employs mechanisms to prevent unnecessary process migrations and to 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% of the total execution time, and the migration overhead to interconnect traffic and cache misses is less than 11%.

Chapter 5

Conclusions

In NUMA systems, the communication among tasks of parallel applications have a significant impact on performance and energy consumption. The mapping of tasks to processor cores, called task mapping, is crucial to achieving the scalable performance on NUMA systems. By exploiting the communication behavior of the application, task mapping can improve the locality of communication, and thus reduce the overall cost of communication. However, in recent NUMA systems, considering only the locality is not sufficient to achieve the best performance and energy efficiency due to the memory congestion. Furthermore, maximizing the locality can degrade the performance because it increases the congestion on the particular NUMA nodes. Thus, it is necessary to consider both locality and the memory congestion on NUMA systems.

In order to perform task mapping that can address the locality and memory congestion, analyzing both the spatial and temporal communication behaviors of the applications is mandatory. It is because the communication behaviors can be different for different applications, and the communication behaviors of a particular application determine the impacts of a task mapping method on the execution of the application. These differences are caused by the diversity of the amount and time of communication events among parallel tasks of the applications.

The objective of this dissertation is to improve the performance and energy efficiency of parallel applications on NUMA systems by introducing task mapping methods for coordinating the locality and the memory congestion. Since analyzing the communication behaviors is a mandatory step in task mapping, this dissertation first discusses a method to analyze and characterize the spatial and temporal communication behaviors of parallel applications. Then, this dissertation discusses offline and online task mapping methods for coordinating locality and memory congestion on NUMA systems.

Chapter 2 discusses the analysis and characterization of the communication behaviors. The problem of this chapter is that considering only the spatial communication behavior is not sufficient for coordinating locality and memory congestion. Analyzing both the spatial and temporal communication behaviors is necessary to identify the concurrent communications that cause the memory congestion. This problem presents two challenges that need to be addressed. The first challenge is to obtain the explicit and implicit communications events among tasks of parallel applications. The second challenge is to analyze and describe the communication behaviors. To address these two challenges, Chapter 2 proposes a method, DeLocProf, to analyze and characterize the communication behaviors. The method employs two techniques to obtain communication events of two different parallel processing methods, a data clustering method to detect concurrent communications, and a set of metrics to characterize the communication behaviors.

The evaluation of Chapter 2 demonstrates that the proposed method can effectively analyze and characterize the spatial and temporal communication behaviors of all the applications. The experimental results have shown that the impacts of task mapping on the performance of the applications may vary depending on the communication behaviors of the applications, and the proposed metrics are effective in characterizing the communication behaviors that affect the locality and memory congestion. Moreover, the results of performance and analysis of the communication behaviors have shown that the metrics can be used to evaluate the suitability of a task mapping method on a particular application. DeLocProf will be useful for computer application and system design which needs to evaluate the impacts of the communication behaviors of parallel applications and task mapping on performance and energy consumption of NUMA systems.

Chapter 3 discusses offline task mapping for coordinating locality and memory congestion. The problem of this chapter is that considering only the locality is not sufficient to achieve a scalable performance on NUMA systems. Task mapping for coordinating locality and memory congestion is necessary to improve performance on NUMA systems. The objective of this chapter is to realize task mapping that can address the locality and the memory congestion problems. To achieve this objective, this chapter proposes an offline task mapping method, DeLoc, that uses information about the NUMA node topology and the communication behaviors to improve the locality and reduce the memory congestion. To perform the task mapping, DeLoc analyzes the communication behaviors and calculates the mapping prior to the execution of the application. The evaluation of Chapter 3 has demonstrated that DeLoc achieves the highest performance for most of the tested applications by simultaneously reducing the amount of remote accesses and memory congestion. The performance improvements are obtained from the reductions in the cache misses, interconnect traffic, and memory access delays in memory controllers. The evaluation results also clarify the impacts of task mapping on the performance of the applications on NUMA systems.

Chapter 4 discusses online task mapping for coordinating locality and memory congestion. The problem of this chapter is that analyzing the communication behaviors is a mandatory step in task mapping for coordinating locality and memory congestion on NUMA systems. However, the offline profiling and analysis of communication behaviors impose a high overhead and is not applicable if the application changes its communication behavior between executions. The objective of Chapter 4 is to obtain task mapping that can coordinate locality and memory congestion without the needs of offline profiling and analysis. To achieve this objective, Chapter 4 proposes an online mapping method, OnDeLoc-MPI, to address the locality and memory congestion problems. It works online during the execution of an MPI application, and dynamically performs the task mapping to adapt to changes in the communication behavior of the application. In contrast to DeLoc and most of the related work, OnDeLoc-MPI does not require offline profiling and analysis of the communication behaviors, and does not rely on communication detection mechanisms and specific hardware or operating system. Alternatively, it analyzes the communication behaviors by monitoring the MPI communication events during the execution of the application. The evaluation of Chapter 4 has shown that OnDeLoc-MPI can achieve performance and energy consumption improvements close to the best static mapping method with low overhead. Moreover, Chapter 4 clarifies the impacts of task mapping on energy consumption of the applications on NUMA systems.

Chapter 4 has demonstrated the effectiveness of OnDeLoc-MPI in reducing both the amount of remote accesses and the memory congestion without the needs of offline profiling and analysis. However, the performance and energy consumption improvements of OnDeLoc-MPI in most of the tested applications are still lower than those of DeLoc, which also show the advantage of DeLoc over OnDeLoc-MPI. By analyzing the communication behaviors and computing the mapping offline prior to the execution, DeLoc does not introduce the migration overhead to the execution. Because of this advantage, DeLoc is more suitable for the application that has a static communication behavior, while OnDeLoc-MPI is suitable for the application that has a dynamic communication behavior. As shown in the evaluation of Chapter 2, by using the proposed metrics, parallel applications can be classified based on the suitability of offline and online task mapping methods to improve the performance of the applications. The evaluation of Chapters 2, 3 and 4 suggest that both DeLoc and OnDeLoc-MPI are necessary to improve the performance and energy efficiency on NUMA systems.

Although this dissertation has shown the significance of the proposed methods, some limitations of the methods also need to be considered. DeLocProf detects implicit communications among threads by monitoring all memory accesses using binary dynamic instrumentation. If the number of memory accesses is very large, the overhead caused by the dynamic instrumentation may surpass the benefits of task mapping. One way to reduce this overhead is by using sampling mechanisms to limit the number of memory accesses traced during the execution. However, this approach may reduce the accuracy of the communication detection. Further studies can use DeLocProf to investigate the impacts of the sampling mechanisms on the results of analysis and characterization of the communication behaviors.

DeLoc and OnDeLoc-MPI consider tasks in a parallel application as processes or threads, not both. The mapping results may not be optimal if the application consists of tasks implemented as processes and threads, such as in hybrid MPI/OpenMP implementations. Furthermore, DeLoc and OnDeLoc-MPI use a tree model to represent the NUMA node topology of the system. One model arises in upcoming architectures with different kinds of memory device. For instance, the KNL system features high-bandwidth memory, called MCDRAM, inside the package while also using standard off-package DDR memory. It means that each core may have two distinct local NUMA nodes, and thus, it has two parents. In this case, a different model, such as a graph, is required to express the NUMA node topology. To make DeLoc and OnDeLoc-MPI applicable to this case, further studies can extend the mapping algorithms for different models of the NUMA node topology.

Future work has been identified to extend the methods proposed in this dissertation. DeLoc and OnDeLoc-MPI have been thoroughly evaluated on different scales of NUMA systems and a simulator. However, on a large cluster of NUMA systems, the impacts of task mapping may change because the latency and bandwidth of internetwork links will also affect performance and energy consumption. Thus, extending the methods for the large cluster is one topic for the further work. For this future work, extending the methods for parallel applications with hybrid programming of MPI and multithreading, such as OpenMP and Pthreads, will also be necessary.

This dissertation has shown that the migration overhead significantly affects performance and energy consumption. Furthermore, on a cluster of NUMA systems, migrating tasks between systems will incur a higher overhead due to the higher latency of internetwork links. One approach to reduce the overhead is by analyzing the impacts of the migration overhead offline prior to the execution of the application. This analysis requires quantitative metrics to estimate the impacts of the migration overhead on a particular application. Thus, further studies targeting task mapping for large-scale NUMA systems can extend the mapping methods of this dissertation to a hybrid method of offline and online mechanisms.

Bibliography

[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," *Commun. ACM*, vol. 58, no. 12, p. 59–66, Nov. 2015.

[2] M. Diener, E. H. Cruz, P. O. Navaux, A. Busse, and H.-U. Heiíß, "Communication-aware process and thread mapping using online communication detection," *Parallel Comput.*, vol. 43, no. C, pp. 43–63, Mar. 2015.

[3] F. Broquedis, N. Furmento, B. Goglin, P.-A. Wacrenier, and R. Namyst, "Forest-GOMP: An efficient OpenMP environment for NUMA architectures," *International Journal of Parallel Programming*, vol. 38, no. 5, pp. 418–439, Oct 2010.

[4] 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.

[5] T. Brecht, "On the importance of parallel application placement in NUMA multiprocessors," in *USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems - Volume 4*, ser. Sedms'93. Berkeley, CA, USA: USENIX Association, 1993, pp. 1–1.

[6] K. C. Sevcik and S. Zhou, "Performance benefits and limitations of large numa multiprocessors," *Performance Evaluation*, vol. 20, no. 1, pp. 185 – 205, 1994, performance '93.

[7] C. Lameter, "NUMA (Non-Uniform Memory Access): An overview," *Queue*, vol. 11, no. 7, pp. 40:40–40:51, Jul. 2013.

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

[9] P. Conway and B. Hughes, "The amd opteron northbridge architecture," *IEEE Micro*, vol. 27, no. 2, pp. 10–21, Mar. 2007.

[10] J. Feliu, J. Sahuquillo, S. Petit, and J. Duato, "Understanding cache hierarchy contention in cmps to improve job scheduling," in *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, ser. IPDPS '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 508–519.

[11] 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.

[12] 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.

[13] 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.

[14] B. Lepers, V. Quéma, and A. Fedorova, "Thread and memory placement on NUMA systems: Asymmetry matters," in *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '15, Berkeley, CA, USA, 2015, pp. 277–289.

[15] J. Lenis and M. A. Senar, “A performance comparison of data and memory allocation strategies for sequence aligners on NUMA architectures,” *Cluster Computing*, vol. 20, no. 3, pp. 1909–1924, Sep 2017.

[16] S. Borkar and A. A. Chien, “The future of microprocessors,” *Commun. ACM*, vol. 54, no. 5, pp. 67–77, May 2011.

[17] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach*, 5th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[18] A. Kagi, J. R. Goodman, and D. Burger, “Memory bandwidth limitations of future microprocessors,” in *23rd Annual International Symposium on Computer Architecture (ISCA’96)*, May 1996, pp. 78–78.

[19] J. E. Boillat and P. G. Kropf, “A fast distributed mapping algorithm,” in *CONPAR 90 — VAPP IV*, H. Burkhart, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1990, pp. 405–416.

[20] J. M. Orduña, F. Silla, and J. Duato, “On the development of a communication-aware task mapping technique,” *J. Syst. Archit.*, vol. 50, no. 4, pp. 207–220, Mar. 2004.

[21] M. Diener, E. H. Cruz, L. L. Pilla, F. Dupros, and P. O. Navaux, “Characterizing communication and page usage of parallel applications for thread and data mapping,” *Performance Evaluation*, vol. 88-89, pp. 18 – 36, 2015.

[22] S. Chodnekar, V. Srinivasan, A. S. Vaidya, A. Sivasubramaniam, and C. R. Das, “Towards a communication characterization methodology for parallel applications,” in *Proceedings Third International Symposium on High-Performance Computer Architecture*, Feb 1997, pp. 310–319.

[23] N. Barrow-Williams, C. Fensch, and S. Moore, "A communication characterisation of splash-2 and parsec," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, Oct 2009, pp. 86–97.

[24] J. Zhang, J. Zhai, W. Chen, and W. Zheng, "Process mapping for mpi collective communications," in *Euro-Par 2009 Parallel Processing*, H. Sips, D. Epema, and H.-X. Lin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 81–92.

[25] E. R. Rodrigues, F. L. Madruga, P. O. A. Navaux, and J. Panetta, "Multi-core aware process mapping and its impact on communication overhead of parallel applications," in *2009 IEEE Symposium on Computers and Communications*, July 2009, pp. 811–817.

[26] M. Diener, E. H. M. Cruz, M. A. Z. Alves, M. S. Alhakeem, P. O. A. Navaux, and H.-U. Heiß, "Locality and balance for communication-aware thread mapping in multicore systems," in *Euro-Par 2015: Parallel Processing*, J. L. Träff, S. Hunold, and F. Versaci, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 196–208.

[27] M. Agung, M. A. Amrizal, R. Egawa, and H. Takizawa, "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)*, Oct. 2019, pp. 17–24.

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

[29] M. Diener, E. H. Cruz, M. A. Alves, P. O. Navaux, and I. Koren, "Affinity-based thread and data mapping in shared memory systems," *ACM Computing Surveys (CSUR)*, vol. 49, no. 4, p. 64, 2017.

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

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

[32] J. Kim and D. J. Lilja, “Characterization of communication patterns in message-passing parallel scientific application programs,” in *Network-Based Parallel Computing Communication, Architecture, and Applications*, D. K. Panda and C. B. Stunkel, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 202–216.

[33] A. Faraj and X. Yuan, “Communication characteristics in the nas parallel benchmarks,” in *14th IASTED International Conference on Parallel and Distributed Computing Systems (PDCS 2002)*, 2002, pp. 724–729.

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

[35] J. P. Singh, E. Rothberg, E. Rothberg, and A. Gupta, “Modeling communication in parallel algorithms: A fruitful interaction between theory and systems?” in *Proceedings of the Sixth Annual ACM Symposium on Parallel Algorithms and Architectures*, ser. SPAA ’94. New York, NY, USA: ACM, 1994, pp. 189–199.

[36] C. McCurdy and J. Vetter, “Memphis: Finding and fixing NUMA-related performance problems on multi-core platforms,” in *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, March 2010, pp. 87–96.

[37] R. Lachaize, B. Lepers, and V. Quéma, “Memprof: A memory profiler for NUMA multicore systems,” in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC’12. Berkeley, CA, USA: USENIX Association, 2012, pp. 5–5.

[38] A. Giménez, T. Gamblin, I. Jusufi, A. Bhatele, M. Schulz, P. Bremer, and B. Hamann, “Memaxes: Visualization and analytics for characterizing complex memory perfor-

mance behaviors," *IEEE Transactions on Visualization and Computer Graphics*, vol. 24, no. 7, pp. 2180–2193, July 2018.

[39] J. Diaz, C. Munoz-Caro, and A. Nino, "A survey of parallel programming models and tools in the multi and many-core era," *IEEE Transactions on parallel and distributed systems*, vol. 23, no. 8, pp. 1369–1386, 2012.

[40] R. L. Graham and G. Shipman, "MPI Support for Multi-core Architectures: Optimized Shared Memory Collectives," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, A. Lastovetsky, T. Kechadi, and J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 130–140.

[41] H. Zhu, D. Goodell, W. Gropp, and R. Thakur, "Hierarchical Collectives in MPICH2," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, M. Ropo, J. Westerholm, and J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 325–326.

[42] G. Bosilca, C. Foyer, E. Jeannot, G. Mercier, and G. Papauré, "Online Dynamic Monitoring of MPI Communications," in *European Conference on Parallel Processing*. Berlin, Heidelberg: Springer, 2017, pp. 49–62.

[43] 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*. Berlin, Heidelberg: Springer, 2004, pp. 97–104.

[44] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, Jan 1998.

[45] IEEE, "IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(R)) Base Specifications, Issue 7," *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)*, pp. 1–3951, Jan 2018.

[46] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Acm sigplan notices*, vol. 40, no. 6. ACM, 2005, pp. 190–200.

[47] W. Gropp, "MPICH2: A new start for MPI implementations," in *Proceedings of the 9th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface.* London, UK, UK: Springer-Verlag, 2002, pp. 7–7.

[48] A. E. Eichenberger, C. Terboven, M. Wong, and D. an Mey, "The design of openmp thread affinity," in *Proceedings of the 8th International Conference on OpenMP in a Heterogeneous World*, ser. IWOMP'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 15–28.

[49] M. Agung, M. A. Amrizal, K. Komatsu, R. Egawa, and H. Takizawa, "A memory congestion-aware MPI process placement for modern NUMA systems," in *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*, Dec 2017, pp. 152–161.

[50] M. Ackerman, S. Ben-David, S. Brânzei, and D. Loker, "Weighted clustering," in *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence*, ser. AAAI'12. AAAI Press, 2012, pp. 858–863.

[51] G. Schwarz *et al.*, "Estimating the dimension of a model," *The annals of statistics*, vol. 6, no. 2, pp. 461–464, 1978.

[52] D. Pelleg and A. W. Moore, "X-means: Extending k-means with efficient estimation of the number of clusters," in *Proceedings of the Seventeenth International Conference*

on Machine Learning, ser. ICML '00. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000, pp. 727–734.

[53] 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.

[54] M. Kumar, "Measuring parallelism in computation-intensive scientific/engineering applications," *IEEE Transactions on Computers*, vol. 37, no. 9, pp. 1088–1098, Sep. 1988.

[55] C. Bienia and K. Li, "PARSEC 2.0: A new benchmark suite for chip-multiprocessors," in *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*, June 2009.

[56] A. C. De Melo, "The new Linux perf tools," in *Slides from Linux Kongress*, vol. 18. Nuremberg, Germany: Georg Simon Ohm University, 2010.

[57] G. W. Hill, "Acm algorithm 395: Student's t-distribution," *Commun. ACM*, vol. 13, no. 10, p. 617–619, Oct. 1970.

[58] Bokhari, "On the mapping problem," *IEEE Transactions on Computers*, vol. C-30, no. 3, pp. 207–214, March 1981.

[59] B. Hendrickson and R. Leland, "The chaco user's guide: Version 2.0," Technical Report SAND95-2344, Sandia National Laboratories, Tech. Rep., 1995.

[60] T. Ma, T. Herault, G. Bosilca, and J. J. Dongarra, "Process distance-aware adaptive mpi collective communications," in *Cluster Computing (CLUSTER), 2011 IEEE International Conference on.* IEEE, 2011, pp. 196–204.

[61] J. Corbet. (2012) AutoNUMA: the other approach to NUMA scheduling. Retrieved Oct 1, 2019 from https://lwn.net/Articles/488709.

[62] C. Terboven, D. an Mey, D. Schmidl, H. Jin, and T. Reichstein, “Data and thread affinity in OpenMP programs,” in *Proceedings of the 2008 Workshop on Memory Access on Future Processors: A Solved Problem?*, ser. MAW ’08. New York, NY, USA: ACM, 2008, pp. 377–384.

[63] 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 *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on.* IEEE, 2010, pp. 180–186.

[64] A. Kleen, “A NUMA API for Linux,” *Novel Inc*, 2005.

[65] J. Treibig, G. Hager, and G. Wellein, “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*, San Diego CA, 2010.

[66] M. Awasthi, D. W. Nellans, K. Sudan, R. Balasubramonian, and A. Davis, “Handling the problems and opportunities posed by multiple on-chip memory controllers,” in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT ’10. New York, NY, USA: ACM, 2010, pp. 319–330.

[67] K. Brodlie, *Visualization Techniques.* Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 37–85.

[68] Intel, “Intel Xeon Processor E5 and E7 v4 Product Families Uncore Performance Monitoring Reference Manual,” Apr 2016. [Online]. Available: https://www.intel.com/content/www/us/en/products/docs/processors/xeon/xeon-e5-e7-v4-uncore-performance-monitoring.html

[69] A. Fedorova, S. Blagodurov, and S. Zhuravlev, “Managing contention for shared resources on multicore processors,” *Queue*, vol. 8, no. 1, pp. 20:20–20:35, Jan. 2010.

[70] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 11, no. 3, p. 28, 2014.

[71] D. Buntinas, G. Mercier, and W. Gropp, "Implementation and shared-memory evaluation of MPICH2 over the Nemesis communication subsystem," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, B. Mohr, J. L. Träff, J. Worringen, and J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 86–95.

[72] B. Goglin and S. Moreaud, "KNEM: A generic and scalable kernel-assisted intra-node MPI communication framework," *Journal of Parallel and Distributed Computing*, vol. 73, no. 2, pp. 176 – 188, 2013.

[73] J. Zhai, T. Sheng, J. He, W. Chen, and W. Zheng, "Efficiently acquiring communication traces for large-scale parallel applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 11, pp. 1862–1870, Nov 2011.

[74] A. Barak, A. Margolin, and A. Shiloh, "Automatic resource-centric process migration for MPI," in *Recent Advances in the Message Passing Interface*, J. L. Träff, S. Benkner, and J. J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 163–172.

[75] G. Dózsa, S. Kumar, P. Balaji, D. Buntinas, D. Goodell, W. Gropp, J. Ratterman, and R. Thakur, "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*, ser. EuroMPI'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 11–20.

[76] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le, "RAPL: Memory power estimation and capping," in *2010 ACM/IEEE International Symposium on Low-Power Electronics and Design (ISLPED)*, Aug 2010, pp. 189–194.

[77] J. Hofmann, D. Fey, J. Eitzinger, G. Hager, and G. Wellein, “Analysis of intel’s haswell microarchitecture using the ecm model and microbenchmarks,” in *Proceedings of the 29th International Conference on Architecture of Computing Systems – ARCS 2016 - Volume 9637*. New York, NY, USA: Springer-Verlag New York, Inc., 2016, pp. 210–222.

[78] A. Sodani, “Knights landing (KNL): 2nd Generation Intel® Xeon Phi processor,” in *2015 IEEE Hot Chips 27 Symposium (HCS)*, Aug 2015, pp. 1–24.

[79] PRACE, “Unified European Applications Benchmark Suite,” 2013, www.prace-ri.eu/ueabs.

[80] D. Zivanovic, M. Pavlovic, M. Radulovic, H. Shin, J. Son, S. A. Mckee, P. M. Carpenter, P. Radojković, and E. Ayguadé, “Main memory in HPC: Do we need more or could we live with less?” *ACM Trans. Archit. Code Optim.*, vol. 14, no. 1, pp. 3:1–3:26, Mar. 2017.

[81] M. J. Abraham, T. Murtola, R. Schulz, S. Páll, J. C. Smith, B. Hess, and E. Lindahl, “GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers,” *SoftwareX*, vol. 1-2, pp. 19 – 25, 2015.

Acknowledgments

This dissertation would not have been possible without the inspiration and support of a number of wonderful individuals. My thanks and appreciation to all of them for making this dissertation possible. I express my deepest gratitude to my supervisor, Professor Hiroyuki Takizawa. Without his enthusiasm, encouragement, support, and continuous optimism, this dissertation would not have been completed. I express my sincere gratitude to Associate Professor Ryusuke Egawa for providing insightful comments and advices during my study, and thoughtful review of this dissertation. I would also like to express my gratitude to Professor Hiroaki Kobayashi and Professor Kentaro Sano, for being the committee members, and for their valuable comments and suggestions that contribute significantly to this dissertation. I express my gratitude also to Associate Professor Kazuhiko Komatsu, Assistant Professor Masayuki Sato, and Dr. Muhammad Alfian Amrizal for their help and support during my study.

I would also like to thank all other members in Takizawa and Egawa Laboratory, for their comments, discussions and support during my study. I would also like to express my gratitude to the Ministry of Education, Culture, Sports, Science and Technology of Japan for the financial support during my study. Finally, my deep and sincere gratitude to my family for their continuous and unparalleled love, help and support.

Mulya Agung

January, 2020