A Framework for Task Placement on Multicore Architectures

by

Pirah Noor Soomro

A Dissertation Submitted to the
Graduate School of Sciences and Engineering
in Partial Fulfillment of the Requirements for
the Degree of

Master of Science

in

Computer Science and Engineering

KOÇ
UNIVERSITY

January 19, 2018

A Framework for Task Placement on Multicore Architectures

Koç University

Graduate School of Sciences and Engineering

This is to certify that I have examined this copy of a master's thesis by

Pirah Noor Soomro

and have found that it is complete and satisfactory in all respects,

and that any and all revisions required by the final

examining committee have been made.

Committee Members:

Asst. Prof. Didem Unat

Assoc. Prof. Öznur Özkasap

Asst. Prof. Ayşe Yilmazer

Date:

I dedicate this work to my Mom and Dad

ABSTRACT

Current multicore machines have a large number of cores and the number of cores is expected to increase in upcoming exascale multicore machines. Binding parallel tasks to cores according to a placement policy is one of the key aspects to achieve good performance in multicore machines because it can reduce on-chip communication among parallel threads. Binding also prevents operating system from migrating threads, which improves data locality. However, there is no single mapping policy that works best among all different kinds of applications and machines because each machine has a different topology and each application exhibits different communication pattern. Determining the best policy for a given application and for a given machine requires extra programming effort. To relieve the programmer from that burden, we argue the need for an automated task binding tool that generates mapping policy specific to the machine topology and application behaviour.

We present BindMe, a thread binding library, that assists programmer to bind threads to underlying hardware. BindMe incorporates state-of-the- art mapping algorithms which use communication pattern in an application to formulate an efficient task placement policy. We also introduce ChoiceMap, a communication aware mapping algorithm that generates a mapping sequence by respecting mutual priorities of parallel tasks. ChoiceMap performs a fair mapping by reducing communication volume among cores. The algorithm can be used both in shared memory and distributed memory systems. ChoiceMap is incorporated in BindMe and can be used as one of the mapping options. We have tested BindMe with various applications from NAS parallel benchmark. Our results show that choosing a mapping policy that best suits the application behavior can increase its performance and no single policy gives the best performance across different applications.

ÖZETÇE

Mevcut çok çekirdekli sistemler çok sayıda işlemci çekirdeğine sahiptir ve yakın zamanda kullanılmaya başlanacak olan exascale sistemlerde çekirdek sayısının daha da artması beklenmektedir. Paralel işleri işlemcilere yerleştirmek çip içindeki islemciler arasındaki iletişim miktarını azalttığından çok çekirdekli makinelerde performansı arttırıcı bir etkendir. Ayrıca bu yöntem işletim sisteminin iş parçacıklarının göç etmesini önleyerek veri yerelliğinin iyileştirilmesi de sağlamaktadır. Ancak, farklı uygulamalar ve makineler arasında en iyi performansı sağlayacak tek bir paralel işleri işlemcilere yerleştirme algoritması bulunmamakta çünkü her makinenin farklı bir topolojisi bulunmakta ve her uygulama farklı bir iletişim örüntüsü sergilemektedir. Bir uygulama veya bir makine için en uygun algoritmayı belirlemek, fazladan programlama eforu gerektirmektedir. Yazılımcının bu yükünü ortadan kaldırmak için, uygulama örüntüsüne ve makine topolojisine özel eşleme algoritması üretecek otomatik bir iş atama aracının gerekliliğini savunmaktayız.

Bu amaç doğrultusunda bu tezde yazılımcıya iş parçacıklarını donanıma atamasında yardımcı olacak iş parçacığı atama kütüphanesini, BindMe aracını, sunuyoruz. BindMe, etkili bir iş yerleştirme ilkesini formülize etmek için uygulamadaki iletişim örüntülerini kullanır ve en yeni eşleme algoritmalarını bünyesinde bulundurmaktadır. Buna ilave olarak, paralel iş parçacıklarının ortak önceliklerini değerlendirerek eşleme dizisi üreten iletişim bilinçine dayalı bir eşleme algoritması olan ChoiceMap'i sunuyoruz. ChoiceMap, çekirdekler arası iletişim hacmini azaltarak dengeli bir eşleme gerçekleştirmektedir. Algoritma hem dağıtık, hem paylaşımlı sistemlerde kullanılır. BindMe, bünyesinde ChoiceMap algoritmasını da eşleme seçeneklerinden biri olarak bulundurmaktadır. BindMe aracını, NAS paralel benchmark kapsamındaki bir çok uygulamayla değerlendirdik. Bulgularımız gösteriyor ki, uygulama örüntülerine en

iyi uyacak eşleme ilkesini seçmek, uygulamanın verimini arttırmakta ve tek bir çeşit eşleme algoritması farklı uygulamalarda en iyi performansı sağlayamamaktadır.

ACKNOWLEDGMENTS

I would like to express my gratitude to my supervisor Dr. Didem Unat for being a great mentor. Dr. Didem has been a great motivator and inspiration for me to grow as a researcher. I would also like to thank Higher Education Commission Pakistan for supporting my expanses for my Masters in Sceince. I am also grateful to Graduates School ofS cience and Engineering, Koç University for providing a nice environment to focus on my studies and supporting my research expanses.

Last but not least, I am thankful to my family and friends from core of heart to be with me through all tough times during my studies.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

NOMENCLATURE

- PU : Processing Unit
- MPI : Message Passing Interface
- OMP : Open Multi-Processing
- GNU : GNU is Not Unix
- TIG : Task Integration Graph
- OS : Operating System
- GCC : GNU Compiler Collection
- ICC : Intel C Compiler
- NAS : NASA Advanced Supercomputing
- NPB : NAS Parallel Benchmark
- NUMA : Non-Uniform Memory Access
- SNC : Single NUMA Cluster
- KNL : KNights Landing
- API : Application Programming Interface
- CFD : Computational Fluid Dynamics

Chapter 1

INTRODUCTION

1.1 Motivation

As the number of cores increases in multicore architectures, task placements on cores become an important performance parameter as optimal placement reduces the execution time and power consumption on the chip. In early years of multicore, there were only few cores thus, mapping threads to cores did not play a significant role in performance. However, today's chips are equipped with 10s of cores [Sodani et al., 2016] and it is expected that the number of cores on a chip will double every two to three years [Ang et al., 2014]. Having large number of cores on one chip causes variance in communication latency among cores due to distance between them [Das et al., 2013]. As a result, to avoid execution delays by communication, placement of tasks should be done in a way to maximize the memory bandwidth and minimize the latency.

When an application is executed in parallel there is an affinity among its parallel tasks. Tasks that communicate the most in terms of amount of data would have high affinity with each other. Existence of such affinity-based relation among tasks can be utilized to place threads to reduce the communication time.

Current state-of-the-art shared memory programming models such as OpenMP [Dagum and Menon, 1998] provide process affinity options. Even though these affinity options facilitate thread binding, binding is performed without considering application's communication behavior. One solution is to device a binding sequence based on the communication pattern of the application and bind the task according to that sequence. This solution is not portable because the programmer has to discover the details of machine topology and bind the tasks accordingly.

1.2 BindMe Framework

We develop BindMe framework, which aids programmer in thread binding by discovering the machine topology. It combines the information about the communication pattern of an application and hardware topology and generates a binding sequence by using various mapping algorithms. BindMe leverages existing tools and combines them under a single framework. It uses the Numalize tool [Diener et al., 2016] to extract communication information from an application and leverages hwloc [Broquedis et al., 2010] to explore underlying hardware topology tree. Based on this information it generates a mapping sequence using state of the art algorithms, such as *TreeMatch* [Jeannot et al., 2014] and *EagerMap* [Cruz et al., 2015]. Tasks are then automatically bound to cores according to the mapping sequence. Programmer has the choice to set a mapping algorithm for generating the mapping sequence. BindMe reduces this efforts with a single library call.

We have tested the BindMe library with the applications from NAS benchmark [Jin et al., 1999]. Performance of the applications is evaluated in terms of execution time by binding applications with five different mapping policies with the help of the BindMe library. Our results show that choosing a mapping policy that best suits the application behavior can increase its performance and no single policy gives the best performance across different applications and architectures.

1.3 ChoiceMap Algorithm

We propose a mapping algorithm that utilizes application behavior and machine topology to generate a mapping sequence. We model the mapping problem as a matching problem where tasks are paired based on their priorities for partner tasks. Our algorithm is based on the core concept of roommate matching algorithm [Irving, 1985] in which every person has a priority list of all the other people as a potential roommate.

People are paired based on their mutual priorities. Since the machine has a hierarchy of resources, we apply this algorithm recursively and compose groups out of subgroups of threads. Our approach accurately captures task affinities in an application and generates a fair mapping sequence.

Treematch [Jeannot et al., 2014] and Eagermap [Cruz et al., 2015] also analyze communication pattern of an application and topology of the underlying machine to generate a mapping sequence. However, these mapping algorithms do not consider mutual priorities of tasks to be paired. We propose an algorithm that maps tasks by pairing most mutually desired tasks together. Unlike other mapping algorithms, our algorithm is more fair.

1.4 Contributions

This thesis has following contributions:

- We examined current binding options and ways to automate binding task.
- We developed BindMe that automatically binds tasks to execution units based on a mapping policy by combining machine's topology and application's communication information.
- We examined existing mapping solutions and their problems in detail.
- We designed ChoiceMap, a mapping algorithm that does fair mapping by analyzing communication matrix and machine topology.

1.5 Organization of Thesis

Chapter 2 provides necessary background on task placement. Chapter 3 discusses the existing mapping solutions with examples. Chapter 4 discusses the ChoiceMap algorithm. The BindMe tool is discussed in Chapter 5 along with its API, usage and supporting tools. In Chapter 6 we evaluate mapping policies and compare choicemap with existing solutions.Overall work of this thesis is concluded in Chapter 7.

Chapter 2

BACKGROUND

2.1 Introduction

Current parallel programming frameworks provide binding options that can be set to bind tasks to cores in order to maximize cache usage and load balance among cores. In this chapter we will discuss available binding options and requirements for automatic binding.

Runtime libraries provide options to bind OpenMP threads. The interface is controlled by environment variables. For example, Intel offers KMP_AFFINITY which is used to set binding policy for an application.

In GNU compilers, OpenMP thread binding can be done by setting an environment variable, GOMP_CPU_AFFINITY. For binding threads of an application, binding list should be assigned to GOMP_CPU_AFFINITY prior to execution. This requires programmer to explore machine topology and analyze application behavior.

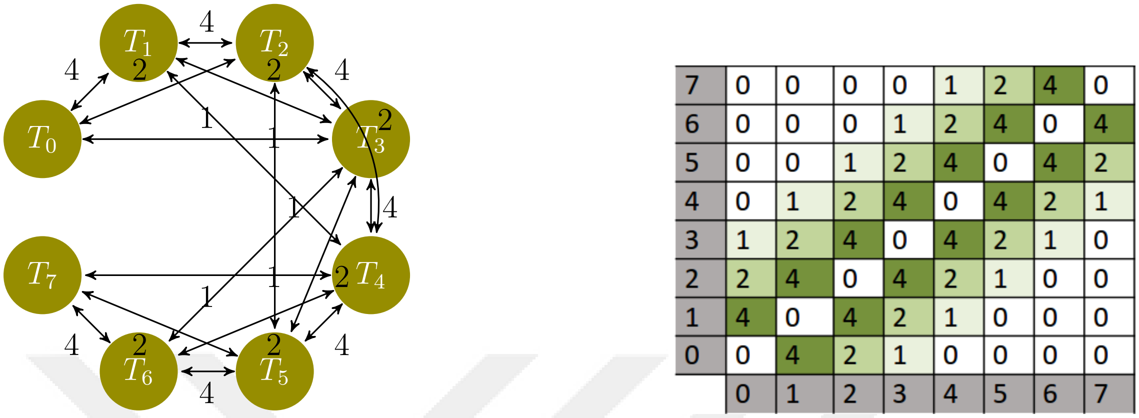
MPI [Gropp et al., 1996], another common parallel programming framework also provides options to bind processes to cores. MPI process launcher (e.g. mpirun or mpiexec) support various binding switches. *-bind-to-X*, where X can be a node, core or socket, binds the processes to said topology level and signals operating system not to migrate the process out of that level X. *-bind-to-X* switch is used with combination of *-byX* switch (X refers to node or core or socket). This switch represents the binding granularity by following a round robin [Shreedhar and Varghese, 1996] mapping policy. User can specify a custom binding strategy by using a *-hostfile* option. The file contains more detailed and machine specific information, such as number of processes to be assigned to each node with a mapping policy represented by *-map-by-X* switch. Although MPI process launcher provides a detailed interface to bind the processes according to intuition of the user. Similar to OpenMP, it requires expertise of user that s/he must know machine topology and applications behavior.

The main purpose of task mapping is to get benefit from data locality and shared memory bandwidth. The execution behavior of a parallel application provides an insight of task affinities for determining a good mapping strategy. For this purpose application is executed once to record communication behavior which is used by mapping algorithm to generate mapping sequence. Mapping algorithms also require information about machine topology to determine total processing units for mapping sequence. We further explain these two inputs in order to understand how they can be used for designing a more complex mapping algorithm.

2.2 Communication Matrix

In a parallel application, task affinity is represented as the amount of data communicated between tasks. More data communicated between two tasks indicates a strong affinity between those tasks. In a shared memory environment, we consider that communication between two threads occur if they share the same cache line.

Communication behavior of an application can be represented as a communication graph or task integration graph [Long and Clarke, 1989]. As shown in Figure 2.1, for a parallel application using 8 threads, the vertices represents threads and edges between two vertices, represent communication. The weight on edges represents the amount of communication between the two corresponding threads. Communication graph can be converted into a communication matrix, which would contain amount of communication between thread pairs where the indices represent the thread IDs. The communication matrix is symmetric and the zero diagonal matrix, as shown in Figure 2.1. Communication aware mapping algorithms analyze communication matrix to generate a mapping sequence for a parallel application.



7	0	0	0	0	1	2	4	0
6	0	0	0	1	2	4	0	4
5	0	0	1	2	4	0	4	2
4	0	1	2	4	0	4	2	1
3	1	2	4	0	4	2	1	0
2	2	4	0	4	2	1	0	0
1	4	0	4	2	1	0	0	0
0	0	4	2	1	0	0	0	0
	0	1	2	3	4	5	6	7

Figure 2.1: Weighted task interaction graph (TIG) of a parallel application with 8 threads (left). Communication matrix of TIG (right)

2.3 Machine Topology

In order to map tasks to processing units, machine architecture should be analyzed first. It gives an insight about the number of tasks to be grouped at each level of topology tree.

Computing resources in a machine can be represented as a tree, where the root represents the machine and the leaf nodes represent logical processing units. Typically a topology tree is symmetric but it does not have to be. Figure 2.2 shows an example machine topology for only its processing units, omitting storage units. The machine topology tree can be converted into an arity sequence, which represents the number of objects at each level of the topology tree. The topology tree of 4 levels in Figure 2.2 is represented by an arity sequence of $1, 2, 2, 2$. Mapping algorithm uses this arity sequence to determine the number of tasks in a single group at a given level of topology tree.

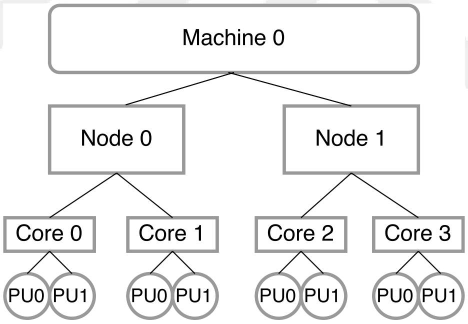


Figure 2.2: An example machine topology, Arity sequence = 1,2,2,2

Chapter 3

EXISTING MAPPING SOLUTIONS

3.1 Introduction

There are various mapping solutions available and the main focus in all mapping policies is, *"How to arrange threads on cores such that the arrangement reduces inter-core communication, improve cache locality and overall execution time. "*. Mapping solutions are categorized into two domains. One of the domain includes such policies that simply binds threads to cores as in OpenMP (Intel compiler) without taking the application behavior into account. On the other hand, second domain includes those mapping algorithms that analyze a specific application on a particular architecture and generate a mapping sequence based on case specific features such as communication behavior and machine topology. For the purpose of reference we name them as general mapping and application specific policies respectively. This chapter covers these two approaches with their examples in detail and discusses short comings which need to be considered for better mapping.

3.2 General Mapping Policies

The main purpose of such binding is to prevent OS from migrating threads between different processing units during execution of an application since thread migration can also lead to page migration which results into lowering the application performance. Next we will consider three binding options supported by Intel compiler.

***Scatter*:** Figure 3.1 shows an example of *scatter* mapping supported in OpenMP, where tasks are equally distributed on resources. An application with 4 threads and affinity set to the *scatter* mode will get equal number of tasks on both NUMA nodes.

Compact: This policy supports data locality with an assumption that there exists more communication and data sharing between neighboring threads. Figure 3.1 shows an example of *compact* policy. The threads are placed so that all processing units of node 0 are occupied first and then next node is utilized for additional threads.

Balanced Balanced is yet another mapping policy which lies in between compact and scatter in terms of benefits. In *balanced* threads are equally distributed among all cores but threads in the same cores are neighboring threads. Thus balanced policy gets benefit of load balancing as in *scatter* and also improves cache utilization by placing neighbor threads together, as in *compact*.

OMP_PROC_BIND: Another mapping policy designed for OpenMP nested tasks based applications is provided by OpenMP 4.5, OMP_PROC_BIND. OMP_PROC_BIND specifies whether processes can migrate throughout the execution or remain at there binding sites. It also specifies thread affinity policy for the corresponding nesting level. *Master* means that nested threads should be placed at the same place where master thread resides. *Close* specifies that nested threads should be bound close to master threads. In *spread*, nested threads are distributed sparsely across the binding sites. In this thesis we do not consider nested parallelism in applications so we have focused only on scatter and compact policies for evaluations, which are most commonly used.

3.3 Communication Aware Mappings

A more case specific mapping can be formulated by analyzing application behavior and the topology of the machine on which application executes. There are various algorithms available in literature which analyze communiaction matrix of an application which is generated beforehand and an information of hardware hierarchy. Two of the latest algorithms are discussed below in the context of communication aware mapping:

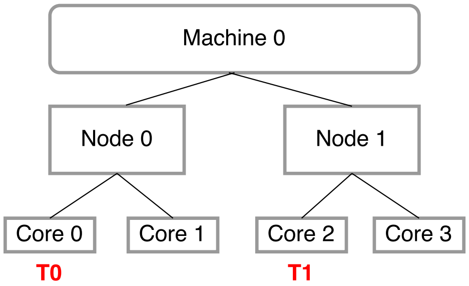
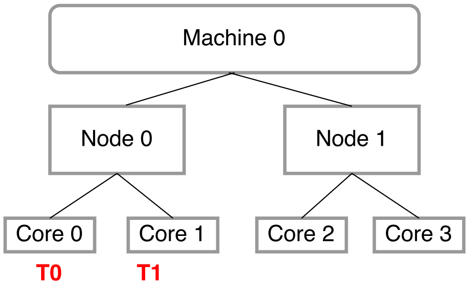


Figure 3.1: On the left, scatter placement policy is shown; task 9 goes to core 0 on socket 0, task 1 goes to core 2 on socket 1. On the right, compact placement policy is shown; both task 0 and 1 are placed on the same socket on core 0 and core 1.



3.3.1 TreeMatch Algorithm

TreeMatch applies two main steps to generate a mapping sequence. Firstly all the possible combinations for given set of tasks are generated. The generated combinations contain pairs with common tasks, for example (1,5), (2,5), (3,5) and so on. TreeMatch converts these redundant pairs into a graph of incompatibilities. The vertices represents task pairs and there is an edge between two pairs if the corresponding pairs contain a common task. For example there will be an edge between (1,5) and (2,5) because 5 is a common task between the two pairs, hence the two pairs are incompatible. This graph is referred to as complement of Kneser Graph [Poljak and Tuza, 1987] in literature. Independent sets of task pairs are generated by processing the graph. An independent set contains task pairs with no common task. In the second step the quality of independent sets is determined by picking the best combination of pairs. In this step the vertices are ranked by the value that represents the amount of communication reduced by pairing two tasks, where smaller is better. An independent set of task pairs with smaller values is generated by applying heuristics such as; ranking vertices by smallest values first and greedily building a maximal independent set. Once groups are generated, topology information is used to map tasks to processing units. Generated sequence represents binding site of threads corresponding

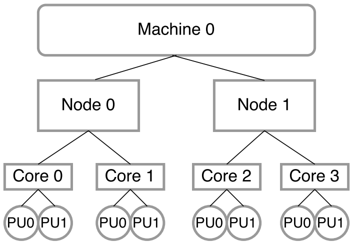


Figure 3.2: An example machine topology, Arity sequence = 1,2,2,2 at level 0,1,2,3 respectively

to an index of sequence.

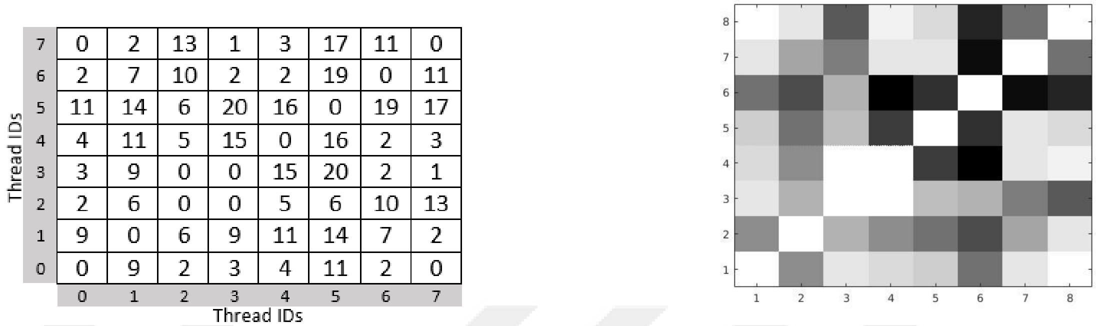
3.3.2 EagerMap Algorithm

EagerMap on the other hand deals with the mapping problem with a different approach. It implements a greedy strategy. For a given task, it searches for a winner task which is the most communicating task with the given task. Pairs are generated by searching for winner tasks. Once all pairs are generated, a topology tree is used to determine group of task pairs. EagerMap applies the same procedure iteratively to generate a mapping sequence for a given machine topology.

3.3.3 Examples

In order to understand the logic of mapping algorithms let us consider a dummy topology shown in Figure 3.2 and an example of communication matrix (Figure 3.3). We will use this example to demonstrate working of mapping algorithms.

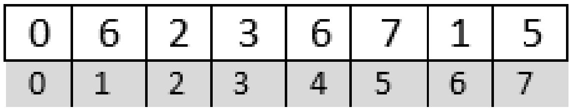
TreeMatch first generates all combination $\binom{p}{n}$ of tasks for a given level. Here n represents total tasks and p represents number of tasks per combination. For example for given case, pairs for level 3 are generated. First of all possible combinations of tasks are created. In our case $\binom{2}{8} = 28$ possible pairs are generated. Out of these combination, next step is to find n/p pairs that do not have any task in common. In order to generate independent tasks, a graph of incompatibilities is generated in which vertices represent task pairs and an edge between two pairs represent that



7	0	2	13	1	3	17	11	0
6	2	7	10	2	2	19	0	11
5	11	14	6	20	16	0	19	17
4	4	11	5	15	0	16	2	3
3	3	9	0	0	15	20	2	1
2	2	6	0	0	5	6	10	13
1	9	0	6	9	11	14	7	2
0	0	9	2	3	4	11	2	0
Thread IDs	0	1	2	3	4	5	6	7

Figure 3.3: Left: Communication matrix, Right: Communication pattern

there is a task in common. An independent set of task pairs can be generated by processing the graph by any greedy algorithm. An independent set of pairs does not means that it is good in quality as well. For example in the matrix pairing 0 with 5 is better than pairing 0 with 2 as 0 is communicating most with 5 instead of 2. Placing most communicating tasks together reduces communication. Therefore pairs in independent sets generated are given a value that represents communiaction reduction if respective tasks get paired. For example for pair (0,5) total communication volume of 0 is 31 and that of 5 is 103. Sum of their communication is 134. if paired together the communication would be reduced by 22, so the total communication of the pair would be 112 which is assigned as value of pair(It is represented as V^* in Figure 3.1). Smaller value is better. Once all pairs are annotated with their values, independent pairs with smaller values are selected for final pairing sequence. In Figure 3.1 best and worst combinations of each task is shown. It can be seen TreeMatch managed to be in between best and worst combinations. Complete sequence for level 3 is shown in Table 3.4



0	6	2	3	6	7	1	5
0	1	2	3	4	5	6	7

Figure 3.4: TreeMatch mapping sequence for level 3

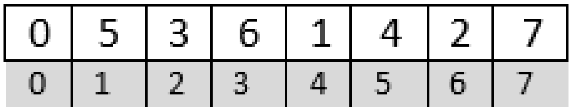
Table 3.1: TreeMatch algorithm executed for example case

0		**1**		**2**		**3**		**4**		**5**		**6**		**7**	
Pair	***V****	**Pair**	***V****	**Pair**	**V***	**Pair**	**V***	**Pair**	**V***	**Pair**	**V***	**Pair**	**V***	**Pair**	**V***
0-1	*71*	1-0	*71*	2-0	*69*	3-0	*75*	4-0	*79*	5-0	*112*	6-0	*80*	7-0	*78*
0-2	*69*	1-2	*88*	2-1	*88*	3-1	*90*	4-1	*92*	5-1	*133*	6-1	*97*	7-1	*101*
0-3	*75*	1-3	*90*	2-3	*92*	3-2	*92*	4-2	*88*	5-2	*133*	6-2	*75*	7-2	*63*
0-4	*79*	1-4	*92*	2-4	*88*	3-4	*76*	4-3	*76*	5-3	*113*	6-3	*99*	7-3	*95*
0-5	*112*	1-5	*133*	2-5	*133*	3-5	*113*	4-5	*127*	5-4	*127*	6-4	*105*	7-4	*97*
0-6	*80*	1-6	*97*	2-6	*75*	3-6	*99*	4-6	*105*	5-6	*118*	6-5	*118*	7-5	*116*
0-7	*78*	1-7	*101*	2-7	*63*	3-7	*95*	4-7	*97*	5-7	*116*	6-7	*78*	7-6	*78*
B(0,2)	69	**B(1,0)**	*71*	**B(2,7)**	*63*	**B(3,0)**	*75*	**B(4,3)**	*76*	**B(5,0)**	*112*	**B(6,2)**	*75*	**B(7,3)**	*63*
W(0,5)	112	**W(1,5)**	*133*	**W(2,5)**	*133*	**W(3,5)**	*113*	**W(4,5)**	*127*	**W(5,1)**	*133*	**W(5,6)**	*118*	**W(7,5)**	*116*
TM(0,6)	80	**TM(1,7)**	*101*	**TM(2,3)**	*92*	**TM(3,2)**	*92*	**TM(4,5)**	*127*	**TM(5,4)**	*127*	**TM(6,0)**	*80*	**TM(7,1)**	*101*
B = Best, W = Worst, TM = TreeMatch															
V* = Pairing value															

Table 3.2: EagerMap algorithm executed for example

Tas number	Task list	Winner task	Pair
0	1,2,3,4,5,6,7	5	(0,5)
1	2,3,4,6,7	4	(1,4)
2	3,6,7	7	(2,7)
3	6	6	(3,6)

EagerMap applies a greedy approach to look for the best partner for a given task. EagerMap picks a task and searches for its winner task. Winner task is another task which communicates the most with the given task. For example as shown in Table 3.2, EagerMap first looks for task 0. By looking at communication matrix, most communicating task with 0 is task 5. Both tasks are paired and removed from tasks list. It next looks for winner of task 1. From the available tasks list task 4 is the most communicating one and gets paired with 1. The process goes on until task list is empty. Complete sequence is shown in Figure 3.5. Different mapping solutions solve the mapping problem with a different approach. TreeMatch generates a mapping sequence with an exhaustive search approach, where all communications



0	5	3	6	1	4	2	7
0	1	2	3	4	5	6	7

Figure 3.5: EagerMap mapping sequence for level 3

are considered to pick the best ones. EagerMap searches for the winner task without considering communication priorities of winning task. For example winning task might have another most communicating task, but since winning task is excluded from the search, it is no longer considered for its own winner task. Considering these shortcomings, we designed our own algorithm for communication aware mapping discussed in next chapter.

Chapter 4

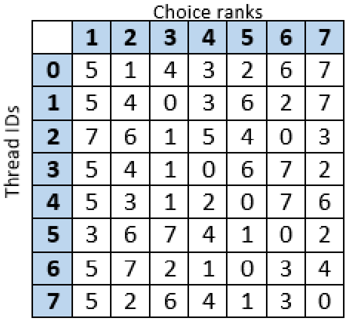
CHOICEMAP: A FAIR TASK MAPPING ALGORITHM

4.1 Introduction

ChoiceMap leverages the communication matrix and machine topology for determining a mapping. However, our algorithm creates a fair mapping and treats each thread equally. Every thread has its own choice list based on the amount of communication it does with other threads. This chapter discusses ChoiceMap in detail.

Pairing threads according to their choice lists results in a mapping sequence that reduces the communication distances. This problem resembles genderless stable marriage problem [McVitie and Wilson, 1971] or roommate matching problem [Irving, 1985] in literature, where a person is paired with another person by considering preferences of both sides. We model the task affinities with a choice matrix, thus creating more opportunities for the tasks to get paired with the best possible task by considering mutual priorities.

A choice matrix is an (n x $(n-1)$) matrix, where n represents total number of parallel tasks. The row index i represents the task number and column index j represents priority of a given task with respect to task i. Choice matrix is suitable to use for pairing since it provides a straightforward understanding of task affinity. For example in EagerMap a winner task is paired with the given task which is similar to pairing a task with its first choice. But that results in an unfair pairing because winner task might have another task as its first choice. Assume that task t communicates the most with task t'. On the other hand, task t' communicates the most with task t'', where $t'' \neq t$. Our algorithm pairs tasks based on their nearest and mutually prioritized choices. As an example, Figure 4.1 shows choice matrix of the communication matrix shown in Figure 3.3.



Choice ranks

Thread IDs	1	2	3	4	5	6	7
0	5	1	4	3	2	6	7
1	5	4	0	3	6	2	7
2	7	6	1	5	4	0	3
3	5	4	1	0	6	7	2
4	5	3	1	2	0	7	6
5	3	6	7	4	1	0	2
6	5	7	2	1	0	3	4
7	5	2	6	4	1	3	0

Figure 4.1: Choice matrix

4.2 ChoiceMap Algorithm

Algorithm 1 presents the ChoiceMap algorithm, which takes two inputs: $aritySequence$ and $commMatrix$. $AritySequence$ represents arity of nodes at each level of machine topology and $commMatrix$ represents the amount of communication between threads in an application. The algorithm is iterated for every level in the machine hierarchy. In line 6, choice matrix ($chMatrix$) is generated from the communication matrix. We assume that total number of tasks are equal to the number of processing units in the machine. Starting from line 10, for every $task$, we first compute the nearest choice from $chMatrix$ and name it as $candidate$. We check, if the $task$ is also nearest choice of $candidate$, then we pair both tasks ($task$ and $candidate$). After pairing, both tasks are removed from search space, thus invalidating them to be a candidate of any other tasks which are not paired yet. Removing tasks from priority lists of all other tasks results in a change in choices of the remaining tasks. Thus there are more chances for the less desirable tasks to be paired since the strong competitors are gone. In this way it guarantees that all the tasks will get paired on the basis of some priority. The process repeats until all the tasks get paired.

4.2.1 Hierarchy Mapping

Pairing tasks only at the deep most level of hierarchy does not fully satisfies task affinity. As in a parallel application, communication is not limited between two tasks.

Algorithm 1 ChoiceMap's Algorithm

```
procedure ChoiceMap($aritySequence, commMatrix$)
    $mappingSequnce := \{\}$    ▷ set of pairs
    $depth \leftarrow getDepth(aritySequence)$    ▷ total levels of machine topology
    for $level\ in\ \{depth - 1 ... 0\}$ do
        $n\_sites \leftarrow getTotalSites(aritySequence, level)$
        $chMatrix \leftarrow generateChoiceMatrix(commMatrix, n\_sites)$
        $paired := 0$    ▷ number of tasks paired in a pass
        while $paired \leq n\_sites$ do    ▷ loop untill all tasks get paired
            $prev\_paired \leftarrow paired$
            for $task\ in\ \{0 ... n\_sites - 1\}$ do
                $candiddate \leftarrow getNearestChoice(task, chMatrix)$
                if $task == getNearestChoice(candidate, chMatrix)$ then
                    $mappingSequence := mappingSequence \cup \{pair(task, candidate)\}$
                    $invalidate(task, candidate, chMatrix)$
                    $paired := paired + 2$
            if $prev\_paired == paired$ then    ▷ if no tasks get paired during the pass
                $solveCycle(commMatrix, chMatrix, mappingSequence)$
        $aggregate(commMatrix, mappingSequence, level + 1)$
    return $mappingSequence$
```

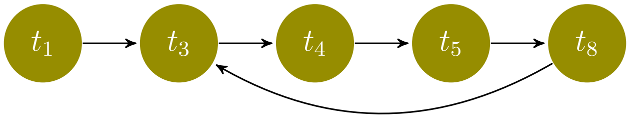


Figure 4.2: Loop in choice matrix

In order to map task pairs to upper level nodes in the machine topology, the task pairs are grouped and treated as a single task. The process is repeated until the root of the topology tree. The communication matrix is also aggregated to represent communication volume between task pairs as shown in Algorithm 1, line 18. In a communication matrix, $C(i, j)$ represents communication volume of task i with task j. Aggregated matrix A of order $(n/2, n/2)$ is defined as:

$$A(i', j') = C(i'(i), j'(i)) + C(i'(i), j'(j)) + C(i'(j), j'(i)) + C(i'(j), j'(j)) \tag{4.1}$$

Where i' and j' represent i^{th} and j^{th} task pair from previous level respectively. Note that pairing tasks is a bottom up procedure. Once aggregated, the choice matrix is generated from aggregated matrix and mapping is performed. Mapping is iteratively performed until the root node of the topology. A mapping sequence corresponding to the nodes of a given level is generated in every iteration.

4.2.2 Cycles in Choice Matrix

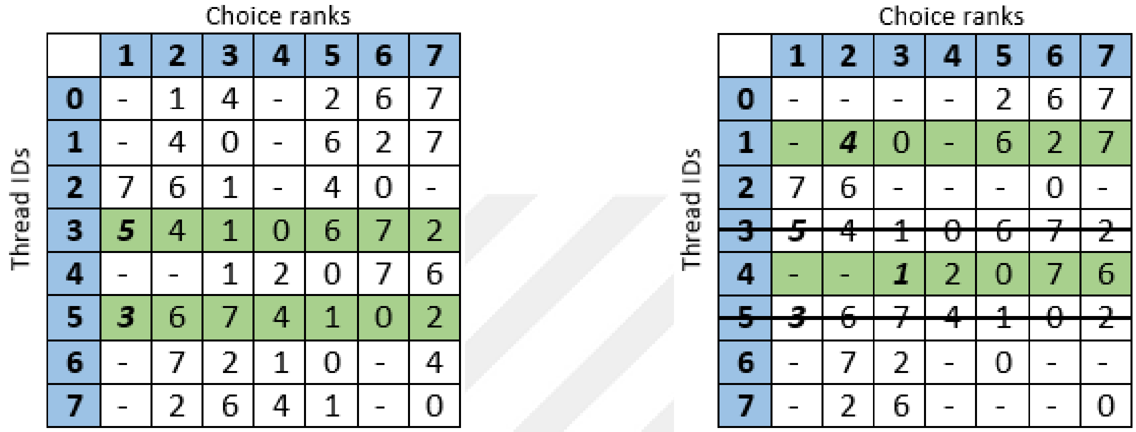
It may happen at any stage of pairing that no tasks get paired at the end of a pairing loop. This means that the choice matrix has a cycle in it. For example as shown in Figure 4.2a task 3 chooses 4, 4 chooses 5, 5, chooses 8 and 8 chooses 3. That means there are no tasks that are the nearest choices of each other. In order to proceed further we need to detect cycle and break the loop, as shown in Listing 2. For solving the loop, we first select a task from the cycle (say, α). We select α as the smallest task number in the cycle, just for the sake of simplicity. To break the cycle we pair the task α to one of its desirers that communicates the most with it (max_comm_α). After pairing we invalidate the paired tasks. Choice matrix is now cycle free and remaining tasks can be paired.

Algorithm 2 Cycle Solving

```
procedure SOLVECYCLE($chMatrix, commMatrix, mappingSequence$)
    $\alpha \leftarrow gettask(chMatrix)$        ▷ select a task from cycle to break it
    $\alpha\_desirers \leftarrow getDesirersOf(\alpha, chMatrix)$
    $max\_comm\_\alpha \leftarrow getMostCommunicatingTask(\alpha, \alpha\_desirers, commMatrix)$
    $pair(\alpha, max\_comm\_\alpha, mappingSequence)$
    $invalidate(\alpha, max\_comm\_\alpha, choiceMatrix)$
    $paired+ = 2$        ▷ paired 2 tasks
    return $mappingSequence$
```

4.3 Example

We now explain working of ChoiceMap with the help of an example case. Referring to Figure 3.2 shows machine topology of our example case. An example of communication matrix shown in Figure 2.1b will be used. ChoiceMap first generates choice matrix from communication matrix as shown in Figure 4.1. The pairing procedure starts from the deepest most level. According to our example we have 8 parallel tasks and for level 3 we need to group them in pairs. During first pass Figure 4.3, task 3 and task 5 are the nearest (first) choice of each other. ChoiceMap saves them as a pair and removes them from the choice matrix. In this way paired tasks are not considered again and other tasks get an opportunity to get paired with their most communicating task from available list of tasks. Four tasks get paired in first pass, and four in second pass as shown in Figure 4.4. The algorithm repeats the same procedure for every level of the topology tree. The pairing sequence of ChoiceMap for level 3 is shown in Figure 4.5. ChoiceMap produces the best possible pairs unlike TreeMatch and EagerMap. Pairing comparison for real applications is discussed in Chapter 6.



Choice ranks

Thread IDs

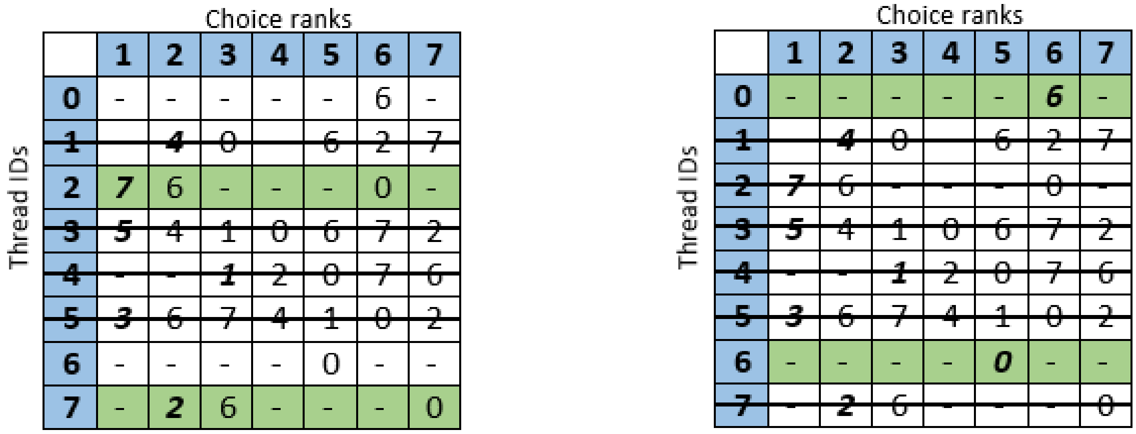
	1	2	3	4	5	6	7
0	-	1	4	-	2	6	7
1	-	4	0	-	6	2	7
2	7	6	1	-	4	0	-
3	***5***	4	1	0	6	7	2
4	-	-	1	2	0	7	6
5	***3***	6	7	4	1	0	2
6	-	7	2	1	0	-	4
7	-	2	6	4	1	-	0

Choice ranks

Thread IDs

	1	2	3	4	5	6	7
0	-	-	-	-	2	6	7
1	-	***4***	0	-	6	2	7
2	7	6	-	-	-	0	-
~~3~~	~~***5***~~	~~4~~	~~1~~	~~0~~	~~6~~	~~7~~	~~2~~
4	-	-	***1***	2	0	7	6
~~5~~	~~***3***~~	~~6~~	~~7~~	~~4~~	~~1~~	~~0~~	~~2~~
6	-	7	2	-	0	-	-
7	-	2	6	-	-	-	0

Figure 4.3: Level 3: Pass 1: four processes got paired.



Choice ranks

Thread IDs

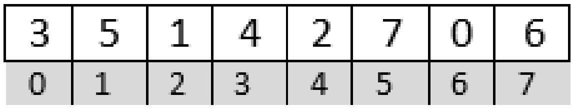
	1	2	3	4	5	6	7
0	-	-	-	-	-	6	-
~~1~~		~~***4***~~	~~0~~		~~6~~	~~2~~	~~7~~
2	***7***	6	-	-	-	0	-
~~3~~	~~***5***~~	~~4~~	~~1~~	~~0~~	~~6~~	~~7~~	~~2~~
~~4~~	~~-~~	~~-~~	~~***1***~~	~~2~~	~~0~~	~~7~~	~~6~~
~~5~~	~~***3***~~	~~6~~	~~7~~	~~4~~	~~1~~	~~0~~	~~2~~
6	-	-	-	-	0	-	-
7	-	***2***	6	-	-	-	0

Choice ranks

Thread IDs

	1	2	3	4	5	6	7
0	-	-	-	-	-	***6***	-
~~1~~		~~***4***~~	~~0~~		~~6~~	~~2~~	~~7~~
~~2~~	~~***7***~~	~~6~~	~~-~~	~~-~~	~~-~~	~~0~~	~~-~~
~~3~~	~~***5***~~	~~4~~	~~1~~	~~0~~	~~6~~	~~7~~	~~2~~
~~4~~	~~-~~	~~-~~	~~***1***~~	~~2~~	~~0~~	~~7~~	~~6~~
~~5~~	~~***3***~~	~~6~~	~~7~~	~~4~~	~~1~~	~~0~~	~~2~~
6	-	-	-	-	***0***	-	-
~~7~~	~~-~~	~~***2***~~	~~6~~	~~-~~	~~-~~	~~-~~	~~0~~

Figure 4.4: Level 3: Pass 2: other four processes got paired.



3	5	1	4	2	7	0	6
0	1	2	3	4	5	6	7

Figure 4.5: ChoiceMap mapping sequence for level 3

Table 4.1: Comparison of different mapping algorithms based on reduction in communication volume by pairing

0		**1**		**2**		**3**		**4**		**5**		**6**		**7**	
Pair	***V****	**Pair**	***V****	**Pair**	**V***	**Pair**	**V***	**Pair**	**V***	**Pair**	**V***	**Pair**	**V***	**Pair**	**V***
0-1	*71*	1-0	*71*	2-0	*69*	3-0	*75*	4-0	*79*	5-0	*112*	6-0	*80*	7-0	*78*
0-2	*69*	1-2	*88*	2-1	*88*	3-1	*90*	4-1	*92*	5-1	*133*	6-1	*97*	7-1	*101*
0-3	*75*	1-3	*90*	2-3	*92*	3-2	*92*	4-2	*88*	5-2	*133*	6-2	*75*	7-2	*63*
0-4	*79*	1-4	*92*	2-4	*88*	3-4	*76*	4-3	*76*	5-3	*113*	6-3	*99*	7-3	*95*
0-5	*112*	1-5	*133*	2-5	*133*	3-5	*113*	4-5	*127*	5-4	*127*	6-4	*105*	7-4	*97*
0-6	*80*	1-6	*97*	2-6	*75*	3-6	*99*	4-6	*105*	5-6	*118*	6-5	*118*	7-5	*116*
0-7	*78*	1-7	*101*	2-7	*63*	3-7	*95*	4-7	*97*	5-7	*116*	6-7	*78*	7-6	*78*
B(0,2)	69	**B(1,0)**	*71*	**B(2,7)**	*63*	**B(3,0)**	*75*	**B(4,3)**	*76*	**B(5,0)**	*112*	**B(6,2)**	*75*	**B(7,3)**	*63*
W(0,5)	112	**W(1,5)**	*133*	**W(2,5)**	*133*	**W(3,5)**	*113*	**W(4,5)**	*127*	**W(5,1)**	*133*	**W(5,6)**	*118*	**W(7,5)**	*116*
TM(0,6)	80	**TM(1,7)**	*101*	**TM(2,3)**	*92*	**TM(3,2)**	*92*	**TM(4,5)**	*127*	**TM(5,4)**	*127*	**TM(6,0)**	*80*	**TM(7,1)**	*101*
EM(0,5)	112	**EM(1,4)**	*92*	**EM(2,7)**	*63*	**EM(3,6)**	*99*	**EM(4,1)**	*92*	**EM(5,0)**	*112*	**EM(6,3)**	*99*	**EM(2,7)**	*63*
CM(0,6)	80	**CM(1,4)**	*92*	**CM(2,7)**	*63*	**CM(3,5)**	*113*	**CM(4,1)**	*92*	**CM(5,3)**	*113*	**CM(6,0)**	*80*	**CM(7,2)**	*63*
B = Best, W = Worst, TM = TreeMatch, EM = EagerMap, CM = ChoiceMap															
V* = Pairing value															

4.4 Mapping Comparison

Based on communication reduction formula discussed in Section 3.3.3, we compare the ability of ChoiceMap in terms of picking better pairs for a mapping sequence. This is shown in Table 4.1. For every task we generate pairs, for example for N tasks, there are $N-1$ pairs for every task as a task can not be paired with itself. A pair value is calculated for each pair which represents the quality of the pair. If a pair value is small then its quality is good because it means it performs less communication with the rest of the tasks. In the Table 4.1 Best refers to a pair with smallest pair value (V^*) and Worst refers to a pair with largest pairing value (V^*). It can be observed from the table that ChoiceMap picks the best possible pair candidate for every task.

Chapter 5

BINDME: A THREAD BINDING TOOL

5.1 Introduction

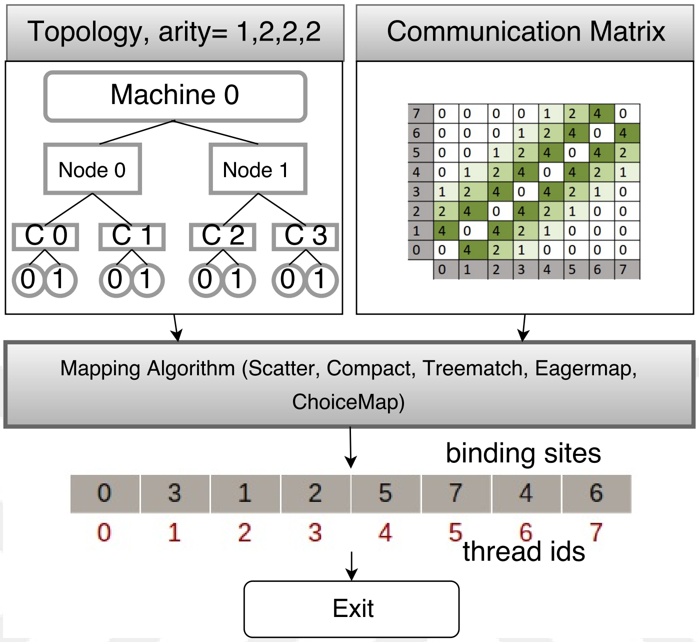
Binding an application according to its communication behavior and matching with the machine topology requires extra efforts from the programmer. The BindMe library encapsulates all the complexities of the binding process such as discovering machine hierarchy, generating a mapping sequence from available mapping policies and then binding parallel tasks. This chapter discusses the implementation and usage of BindMe.

BindMe automatically discovers machine topology and generates a mapping sequence according to the policy specified by the programmer. Figure 5.1 gives an overview of the BindMe library. BindMe generates machine topology by using the *hwloc* tool [Broquedis et al., 2010]. A Mapping sequence is generated by the policy specified by the programmer. Some mapping policies require communication matrix for mapping. For those policies, BindMe utilizes the *Numalize* tool, which captures the communication between threads.

5.2 Programming Interface of BindMe

BindMe packs all the complexities of a binding process in just one function call, namely *bindme()*. An overview of the function is given in Listing 1. The function should be called at the initialization phase of an application program. All threads will be placed on their respective processing units before the main computation of the application starts.

In the internals of the function, we first create the topology tree of underlying ma-



7	0	0	0	0	1	2	4	0
6	0	0	0	1	2	4	0	4
5	0	0	1	2	4	0	4	2
4	0	1	2	4	0	4	2	1
3	1	2	4	0	4	2	1	0
2	2	4	0	4	2	1	0	0
1	4	0	4	2	1	0	0	0
0	0	4	2	1	0	0	0	0
	0	1	2	3	4	5	6	7

0	3	1	2	5	7	4	6
0	1	2	3	4	5	6	7

Figure 5.1: Overview of BindMe

chine using the hwloc [Broquedis et al., 2010] tool, which discovers all the memory and execution units. An arity sequence is generated from the machine topology. In line 12, a mapping sequence is generated by providing the arity sequence, communication matrix and type of the mapping algorithm, which are specified by the programmer. Communication matrix is ignored if the type of mapping algorithm is simple (e.g. *scatter* or *compact*). Once a mapping sequence is generated, an OpenMP parallel region is created, and then every thread binds itself on an execution unit with respect to its thread

```
boolean bindme(Policy mapPolicy, Granularity g){

  boolean status = true;
  Topology topo = getTopology();
  Sequence arSeq = generateAritySequence(topo);
  Sequence mapSeq; // mapping sequence
  if(mapPolicy == COMPACT || mapPolicy == SCATTER)
    mapSeq = generateMapSeq(mapPolicy, arSeq);
  else //if mapPolicy = TREEMATCH or EAGERMAP or CHOICEMAP
  {
    CommMatrix cm = getCommunicationMatrix();
    mapSeq = generateMapSeq(mapPolicy, arSeq, cm);
  }
  int threadID;
  int bindingSite;
  #pragma omp parallel shared(topo, mapSeq, status) private(threadID, bindingSite)
  {
    threadID = omp_get_thread_num();
    bindingSite = mapSeq[threadID];
    //bind each thread to its respective binding site with given granularity(CORE or FINE)
    bindThread( topo, bindingSite, g, &status);
  }
  return status;
}
```

Listing 5.1: main steps of bindme(). Some details are omitted for clarity.

bindme() is called by specifying a mapping policy and granularity. Mapping policies available in BindMe are :

- SCATTER
- COMPACT
- EAGERMAP
- TREEMATCH
- CHOICEMAP

Binding granularity options available in BindMe are:

- CORE: *binds to cores*
- FINE: *binds to logical processing units (for example Hyperthreads)*

For binding an application according to ChoiceMap with fine granularity we call *bindme()* as: **bindme(CHOICEMAP, FINE);** The default mapping option is COMPACT and default binding granularity is FINE.

5.3 Supporting Tools

BindMe leverages different tools to automate the task placement. BindMe uses the *hwloc* tool for exploring machine topology and binding threads/processes to execution units. Communication matrices are generated by *Numalize* and *CommMonitor*. We will now discuss them one by one.

5.3.1 Hwloc

It is an open source and portable **H**ard**W**are **Loc**ality project, in short *hwloc*. It provides a set of command line tools and APIs to examine hardware topology. It provides details of processor, memory and channel type in a machine. *lstopo*, a command line tool of hwloc provides graphical representation of the hardware resources, an example is shown in Figure 5.2. It shows the machine being examined has 2 NUMA nodes, 4 cores per node with 2 logical processors (hyperthreads) in each core. Hwloc provides an interface to bind tasks (thread or process) to a desired processing unit. BindMe uses the hwloc interface for examining machine topology, binding tasks and checking site of execution of task (where a thread or process is running).

5.3.2 Numalize

Numalize [Diener et al., 2016] is a tool for detecting communication in shared memory. It is a cache simulator based on the Pin dynamic binary instrumentation tool [Luk et al., 2005]. The tool traces all memory accesses of an application at the granularity of a cache line, we set the cache line to 64 bytes in our study. *Numalize* detects

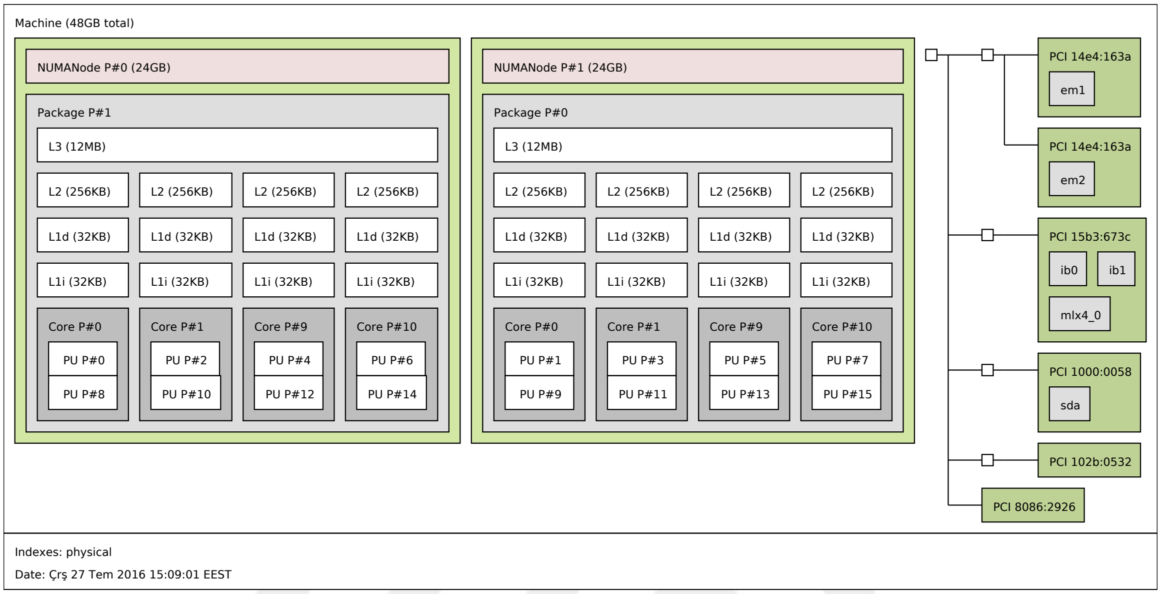


Figure 5.2: Example output of a machine by lstopo.

communication when different threads access the same cache line. The output of *Numalize* is a communication matrix of size *(total threads X total threads)*. When a large application is executed with *Numalize*, the runtime is slowed down by about 100 times the normal execution time. This is because *Numalize* utilizes binary instrumentation tool and monitors every load and store in memory. In some cases *Numalize* crashes because of the large memory requirements of an application in addition to its own internal data structures.

5.3.3 CommMonitor

The communication matrices of several applications cannot be extracted by using *Numalize*. This is due to the large memory consumption overhead by *Numalize*, which causes it to crash and a huge slowdown in execution time (almost 100 times of the normal execution time of the application).

To cope with this problem, one of our group mates, Sasongko developed a tool - *CommMonitor* for detecting communication between threads. *CommMonitor* is based on perf_event_open system call. By using this tool, we can extract communication matrices even for large applications. The memory consumption and execution time overhead of this tool is much smaller than *Numalize*, it adds less than 10% of the overhead to the application since this tool makes use of hardware performance counters for event-based sampling.

In order to capture communication matrix from an application, we set perf_event_open in our tool to perform sampling with sample frequency ten thousands. Furthermore we also set *CommMonitor* to monitor two events; all loads micro operation and all stores micro operations. For each sampled event, i.e. memory load or memory store, we obtain the thread id of the thread which causes the event and the memory address that is accessed. By using the thread id and the address information of memory access events, our tool is able to generate the communication matrix which represents the communication intensity among the threads.

For our evaluation, we used both tools. The cases where *Numalize* fails, we used *CommMonitor* instead.

Chapter 6

EVALUATION

6.1 Introduction

We evaluate BindMe using applications from various scientific domains. BindMe supports five mapping policies to bind an application according to the mapping policy selected by the programmer. In this chapter we perform evaluation of mapping policies using BindMe. We also compare *ChoiceMap* with other mapping algorithms.

Application execution behavior depends on various parameters such as underlying machine topology, programming model, problem size, number of parallel tasks and cache line size. There is no single mapping policy that is best in all cases that is why different mapping policies are tested given a particular set of parameters. BindMe facilitates picking a best mapping policy for the application on a particular machine.

6.2 Testbed

We used two platforms to test BindMe with different applications. One is Intel KNL (KNights Landing) and the other is Intel Broadwell. Figure 6.1 and 6.2 show the schematic diagrams of both machines. KNL is configured with SNC4 (Single NUMA Cluster) mode, which means 4 NUMA (Non-Uniform Memory Access) nodes. We have used 8 tiles per NUMA node. Each tile houses 2 cores and each core has 4 hyperthreads. In total we have 68 cores (272 hyperthreads) divided as 16 and 17 cores per node. Figure 6.2 shows machine topology of Broadwell. It has 2 NUMA nodes, 10 cores on each node. We have used 8 cores per NUMA node because our mapping algorithms works for the power of 2 nodes at every level of machine topology. Each core has two hyperthreads. Table 6.1 summarizes the topology specification of

Table 6.1: Machine topology specification

Machine	NUMA Nodes	Cores per NUMA node	Hyperthreads per core
KNL	4	16/17	4
Broadwell	2	8	2

both machines.

6.3 Evaluation

We chose test applications that perform high volume of communication across parallel threads. The test applications are kernels from various scientific domains such as image processing and computational fluid dynamics (CFD). We used OpenMP C version of NAS parallel benchmark version 3.0. We also tested BindMe with an image segmentation application.

6.3.1 NAS Parallel Benchmark (NPB)

We used five applications from NPB [Jin et al., 1999]; SP, LU, BT, CG and MG. SP (Scaler Pantadigonal solver) solves a synthetic CFD problem. It computes independent systems of non-diagonally dominant, scalar, pentadiagonal equations. LU (Lower Upper) solves regular-sparse lower and upper triangular systems from CFD. BT (Block Tridiagonal) also solves multiple, independent systems of non-diagonally dominant, block tridiagonal equations. MG (MultiGrid) is a simplified multigrid calculation. It processes highly structured long distance communication and tests both short and long distance data communications. In CG, conjugate gradient is used to compute an approximation to the smallest eigenvalues of a large, sparse and symmetric matrix. It performs unstructured grid computations. Table 6.3 summarizes the description of the benchmarks used for evaluation. We have used two classes of NAS applications, B and C. Classes represent the problem size of test applications. Table

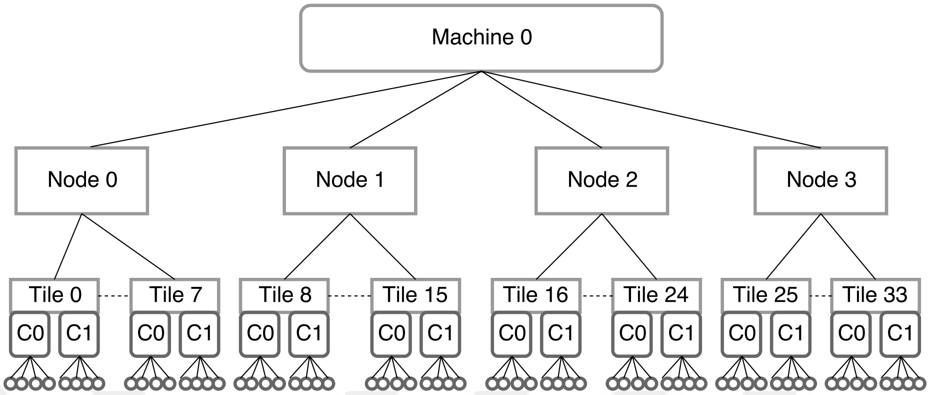


Figure 6.1: Schematic diagram of Intel KNL

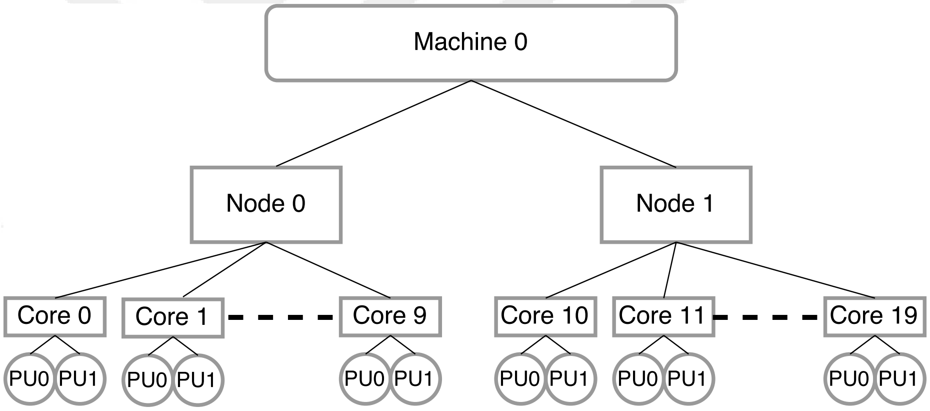


Figure 6.2: Schematic diagram of Intel Broadwell

6.4 shows input size of class B and C for respective applications. Figure 6.3 show the communication matrices of NAS applications for class B and C on Broadwell with 32 threads. Binding granularity is set as *fine*. Figure 6.4 shows communication matrices of NAS applications generated on KNL. We tested NAS applications (Class C) with 64 (granularity = *core*) and 256 (granularity = *fine*) threads on KNL. More details of parameters used for testing are summarized in Table 6.2. The communication matrices for class B are generated from default *Numalize* tool. For class C we modified the ganularity of memory monitoring to 2^4 in *Numalize* to support large input sizes.

Applications like SP, BT and MG perform near neighborhood communications (Figure 6.3 a, d, i and Figure 6.4 a, d, e). Such applications get benefit from the mapping policy in which neighboring threads are placed close to each other. Those applications in which communication is not between direct neighbors but between farthest threads require more careful mapping policies for binding. CG shows all to all communication which means every thread communicates with every other thread. Applications with irregular communication or non-neighbor communication pattern requires efficient mapping policy that reduces inter node communication and improves cache locality.

6.3.2 Image Segmentation Code

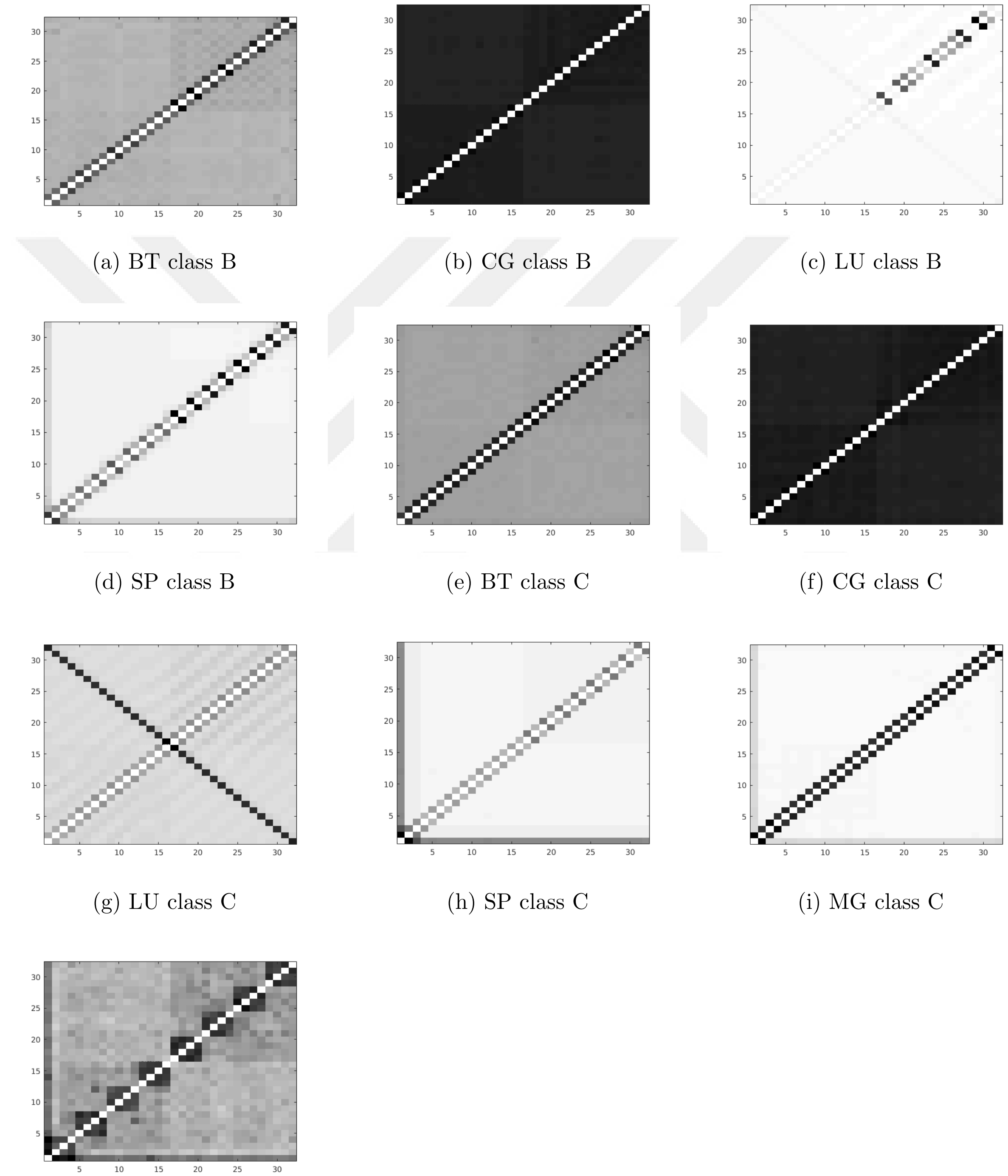
This is an application from Rodinia benchmarks [Che et al., 2009] which processes an image for segmentation. It generates clusters of same pixel colors in the image. The communication behavior changes by changing input image, but remains the same from run to run if provided the same input image. Figure 6.3 *j* and 6.4 *k* show communication matrix of the application by processing a large image of size 9000 x 7000 pixels on Broadwell and KNL respectively.

6.4 Evaluation Metrics

For evaluation of mapping policies we chose a set of metrics that shows runtime improvement of application performance as well as quality of mapping algorithms based on some analytical models.

6.4.1 Execution Time

A good mapping policy results in reduction of inter-core communication and better cache utilization thus reducing execution time. Figures 6.5, 6.6, 6.7 and 6.8 compare normalized execution time with respect to worst mapping policy. We tested five NAS benchmarks executed with five different mapping options available in BindMe; *Compact*, *Scatter*, *EagerMap*, *TreeMatch* and *ChoiceMap*, on both machines. Different mapping policies showed better results for different applications. Since *Compact* places neighbor threads close to each other, it performs better in those applications



(a) BT class B

(b) CG class B

(c) LU class B

(d) SP class B

(e) BT class C

(f) CG class C

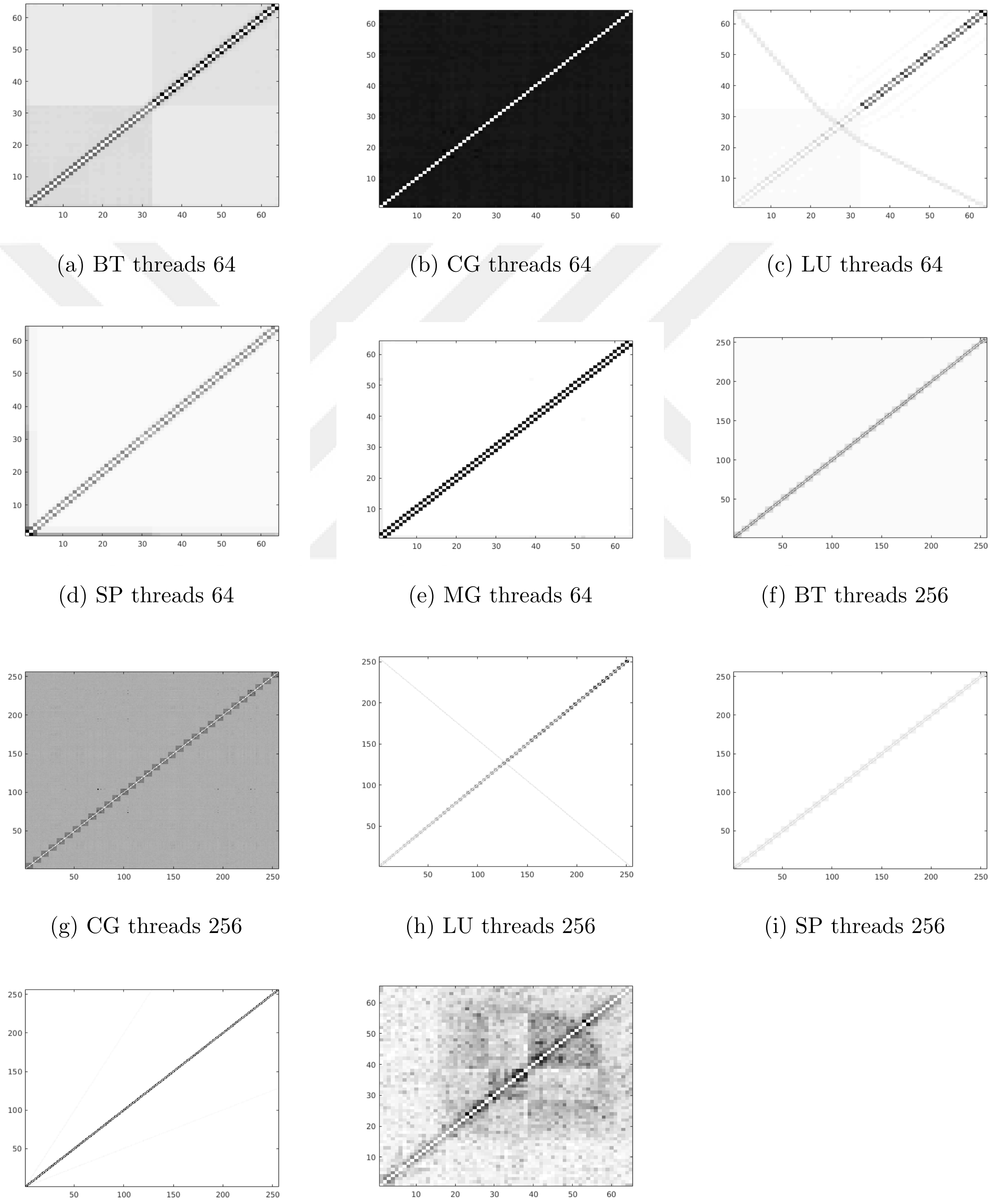
(g) LU class C

(h) SP class C

(i) MG class C

(j) Image segmentation on Broadwell

Figure 6.3: Communication matrices of BT, CG, LU, SP and MG class B and C on Broadwell with 32 threads. We did not include MG Class B here because its execution time is too short for a reliable evaluation



(a) BT threads 64

(b) CG threads 64

(c) LU threads 64

(d) SP threads 64

(e) MG threads 64

(f) BT threads 256

(g) CG threads 256

(h) LU threads 256

(i) SP threads 256

(j) MG threads 256

(k) Image segmentation on KNL

Figure 6.4: Communication matrices of BT, CG, LU, SP and MG on KNL with fine and core granularity

in which near neigborhood communication pattern is dominant. *Scatter* balances computation load, it performed better in case of BT because of a significant all to all communication behavior of BT. *TreeMatch*, *EagerMap* and *ChoiceMap* apply their own strategies to generate a mapping sequence, *ChoiceMap* performed better in many cases. In some cases *EagerMap* and *ChoiceMap* generated almost the same mapping sequence. *TreeMatch* did not perform well overall, but in some cases (SP with 64 threads on Intel KNL) it is better then other mapping policies.

The communication pattern of BT shows near neighborhood communication but BT also shows a significant volume of all to all communication. The all to all communication behavior of BT dominates over near-neighbor communication behavior thus *Scatter* performs better in this case since *Scatter* balances computation load. *ChoiceMap* performed better on KNL with 256 threads.

CG performs all to all communication, in this case compact and mapping algorithms perform better by realizing task affinities due to communication. However on Intel KNL with 256 threads,there is no clear winner policy.

LU does not perform near neighborhood communication, instead the communication is between farther threads, thus mapping by realizing communication pattern produces better results in the case of LU. The execution time of LU on both machines show that the mapping sequence generated by mapping algorithms is better than general mapping policies (*Compact, Scatter*). LU class C executed with 256 threads on KNL shows that *ChoiceMap* performs the best in capturing right pairs for mapping sequence.

SP also shows near neighbor communication along with thread 0's communication with all threads. Figures 6.5 and 6.6 shows execution times of SP, *ChoiceMap* performs better in SP with class C and B. However on Intel KNL, with 64 threads *TreeMatch* performs better than other policies. This shows *Compact* cannot be always the best choice for an application that apparently shows near-neighborhood communication, in some cases *ChoiceMap* successfully detected the best pairs to be grouped together. This point is discussed in more detail in next section.

MG exhibits near-neighbor communication behavior thus in this case *Compact* and *ChoiceMap* perform better on KNL, since *ChoiceMap* is based on mutual task priorities, it capturs the affinities in the case of MG. However on Broadwell there is no clear winner for MG. Figuer 6.9 shows execution time of MG class C executed on Intel Broadwell and Intel KNL. It can be concluded from the figure that the winner mapping policy for the same application on different machines can be different. Therefore a mapping policy must be selected by trying all options. With BindMe it is relatively easy to test different mapping options in order to pick the best performing one.

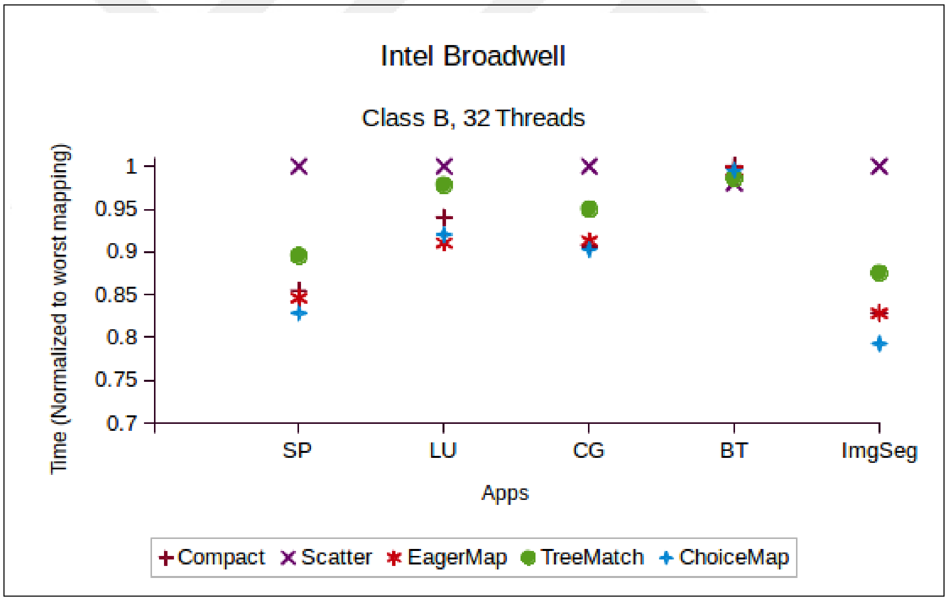


Figure 6.5: Execution time of NAS applications, class B and Image segmentation application executed on Broadwell, granularity = core

6.4.2 Communication Reduction between NUMA Nodes

To compare the quality of pairs generated by the mapping algorithm we calculated the communication reduction value. This formula is previously introduced in *TreeMatch* algorithm to generate pairs, as explained in Section 3.3.3. Communication reduction

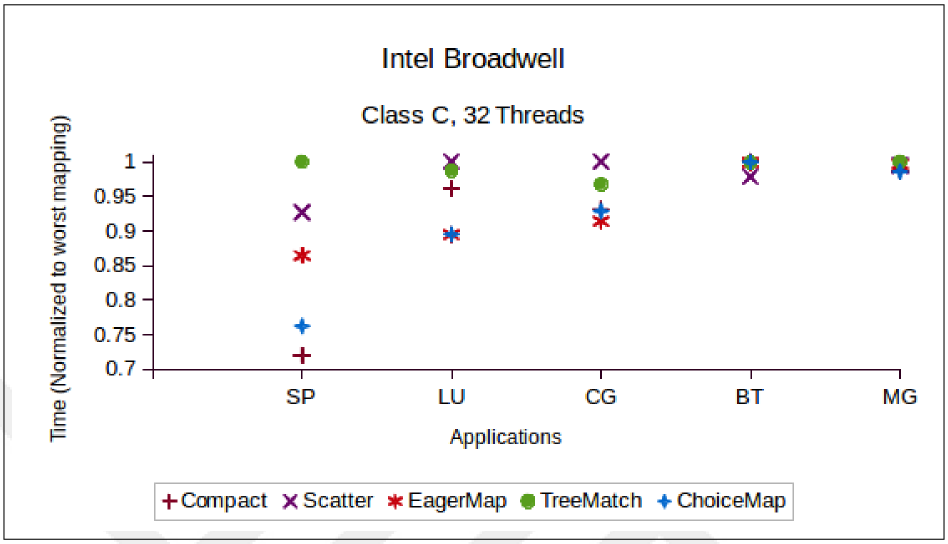


Figure 6.6: Execution time of NAS application, class B on Broadwell, granularity = core

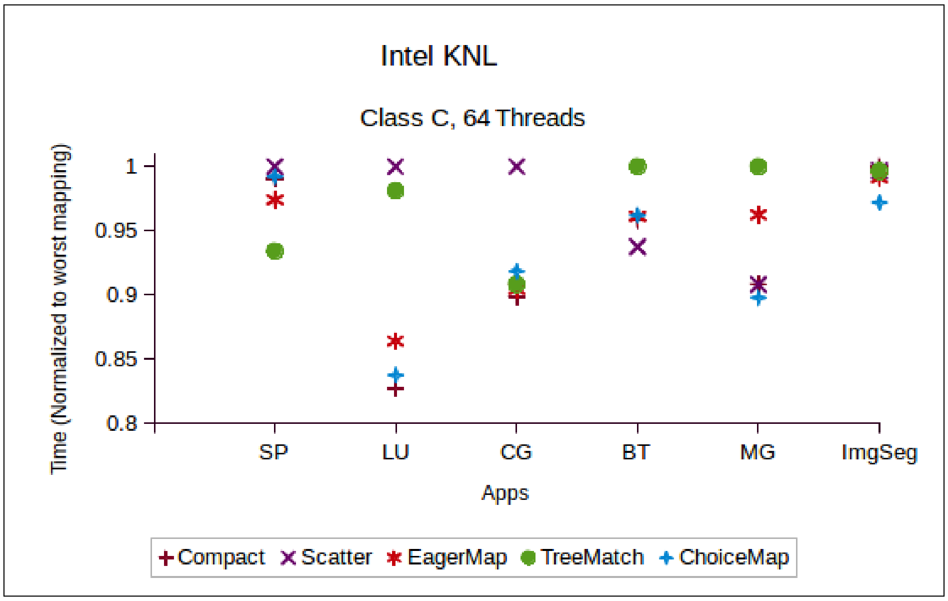


Figure 6.7: Execution time of NAS application, class C on KNL, granularity = core

value C_{rv} of a pair (i, j) is calculated by the equation Figure 6.1.

$$C_{rv}(i,j) = \sum_{t=0}^{N} C(i,t) + \sum_{t=0}^{N} C(j,t) - 2C(i,j) \tag{6.1}$$

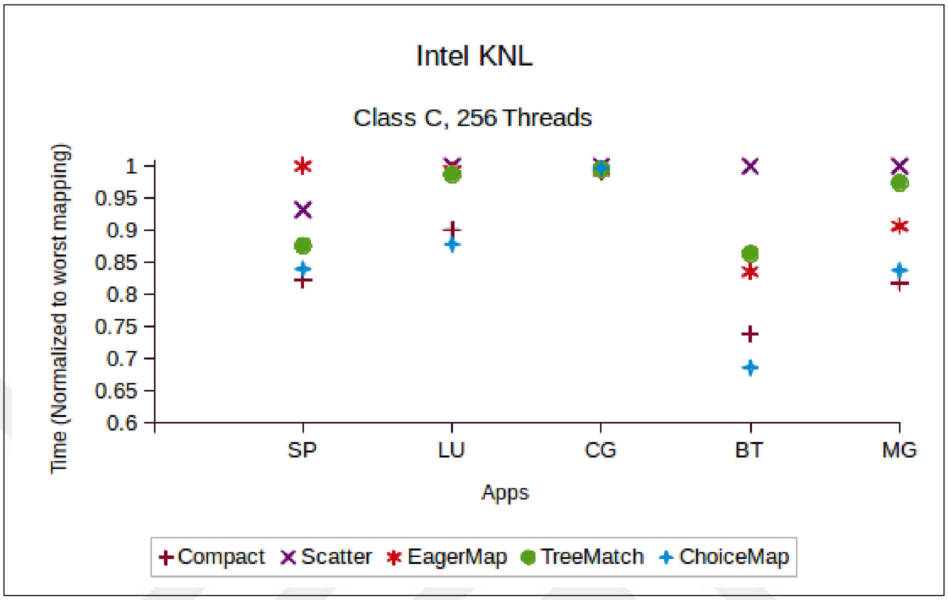


Figure 6.8: Execution time of NAS application, class C on KNL, granularity = fine

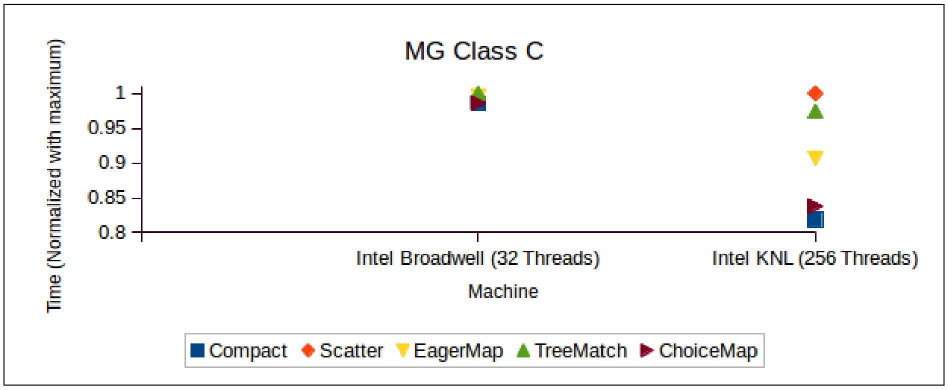


Figure 6.9: Execution time of MG Class C executed on Intel Broadwell and Intel KNL with fine granularity

where i and j are tasks in a pair and C is communication matrix of order $(N * N)$. N refers to total number of tasks.

The best pair is the one with smallest C_{rv} value. The worst pairs is the one with largest C_{rv} value. We generate all possible combinations of N tasks and pick the best and worst pair for every task. We then compare pair values generated by the mapping algorithm. A good mapping policy should generate pairs which are as close as possible to the best pairs, however best and worst pairing for all tasks is not possible. Because a sequence of best or worst pairs contains multiple pairs with common task, which is not possible for mapping.

Mapping with better communication reduction value results in improved data locality. As it reflects that large possible volume of communication is done inside the node at a given level. Thus most of the shared data is available in the shared memory of communicating threads, this results in improved data locality. Figure 6.10 shows that for every task *ChoiceMap* is almost the same as the best pairing. *EagerMap* and *ChoiceMap* generated almost the same pairs for SP on Broadwell at leaf level, therefore both algorithms resulted into better mapping sequence, it is also reflected from the execution time of SP class C on Broadwell (6.6) . The difference in execution time is caused by mapping done at upper levels of topology. However it is not always true, as in Figure 6.10 image segmentation application shows a slight difference in *ChoiceMap* and *EagerMap* pairing. The irregular communication pattern of the application resulted into difference in pairing across mapping algorithms. The better pairing of *ChoiceMap* resulted into reduced execution time of the application as shown in Figure 6.5.

6.4.3 Communication Volume Across NUMA Nodes

A good mapping reduces communication volume across the NUMA nodes or sockets, thus improving locality. To test the quality of mapping policies we compare communication amount between the nodes after mapping tasks according to a mapping policy. A good mapping policy should reduce the amount of communication across the NUMA nodes, that means most of the communication should be performed inside the NUMA node resulting in improved data locality.

To compare communication volume across the nodes, we tested communication matrix of LU (class C) generated on KNL with 64 threads. KNL has 4 NUMA nodes (Figure 6.12) with 16 cores per node. Figure 6.11 compares the average, minimum

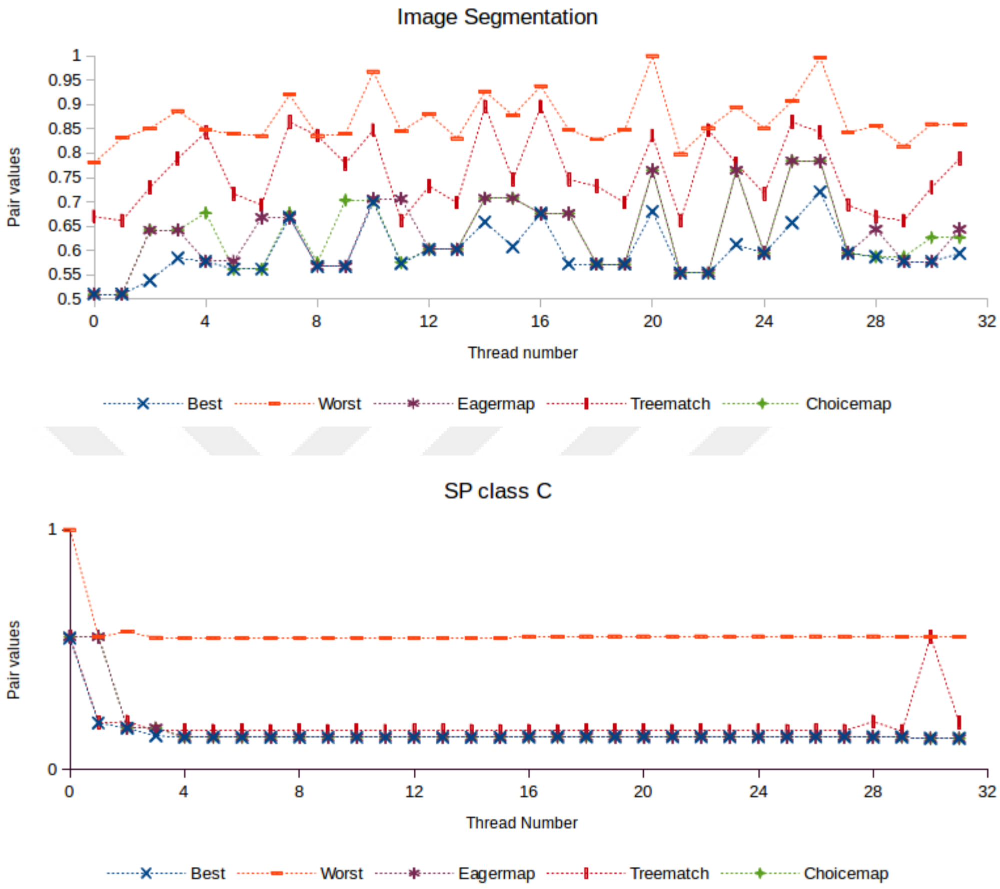


Figure 6.10: Comparison of pair values generated by *EagerMap*, *TreeMatch* and *ChoiceMap* with respect to best and worst pair values

and maximum inter-NUMA node communication with respect to applied mapping. It can be observed that *ChoiceMap* results in the least communication volume for inter-NUMA node communication. This improvement is also reflected from execution time of LU on KNL, representing *ChoiceMap*, and *Compact* as the better mapping policies. Table 6.7 shows the over all communication across the NUMA nodes with respective mappings.

Table 6.2: Configuration of test applications, granularity indicates cache line size in bytes

Benchmark	Class	Iterations	Threads	Machine	Tool
BT	C	30	32	Broadwell	Numalize (Granularity = 2^6)
LU	C	60	32	Broadwell	Numalize (Granularity = 2^6)
SP	C	25	32	Broadwell	Numalize (Granularity = 2^6)
CG	C	75	32	Broadwell	Numalize (Granularity = 2^6)
MG	C	20	32	Broadwell	Numalize (Granularity = 2^6)
BT	B	200	32	Broadwell	Numalize (Granularity = 2^6)
LU	B	250	32	Broadwell	Numalize (Granularity = 2^6)
SP	B	400	32	Broadwell	Numalize (Granularity = 2^6)
CG	B	75	32	Broadwell	Numalize (Granularity = 2^6)
ImgSeg	9K X 7K	N/A	32	Broadwell	CommMonitor
BT	C	10	64	KNL	Numalize (Granularity = 2^4)
LU	C	60	64	KNL	Numalize (Granularity = 2^4)
SP	C	25	64	KNL	Numalize (Granularity = 2^4)
CG	C	75	64	KNL	Numalize (Granularity = 2^4)
MG	C	20	64	KNL	Numalize (Granularity = 2^4)
BT	C	5	256	KNL	Numalize (Granularity = 2^4)
LU	C	10	256	KNL	Numalize (Granularity = 2^4)
SP	C	2	256	KNL	Numalize (Granularity = 2^4)
CG	C	75	256	KNL	Numalize (Granularity = 2^4)
MG	C	20	256	KNL	Numalize (Granularity = 2^4)
ImgSeg	9K X 7K	N/A	64	KNL	CommMonitor

Table 6.3: Test applications

Benchmarks	**Input class**
LU: Solves regular-sparse lower and upper triangular systems from CFD	B,C
SP: It computes independent systems of non-diagonally dominant, scalar, pentadiagonal equations	B,C
CG: Conjugate gradient method used to find smallest eigenvalues of a large, sparse and symmetric matrix	B,C
BT: solves multiple, independent systems of non diagonally dominant, block tridiagonal equations	B,C
MG: a simplified multigrid calculation. It processes highly structured long distance communication and tests both short and long distance data communications	C
Image Segmentation: It generates clusters of same pixel colors in the image	Image of 9000 X 7000 pixels

Table 6.4: Input size of class B and C for NAS applications

Application	B	C
SP	102^3	162^3
BT	102^3	162^3
LU	102^3	162^3
MG	256^3	512^3
CG	75000	150000

Table 6.7: Communication volume (in Trillion communication events in terms of cache line sharing) of LU (class C) between four NUMA nodes of KNL with 64 threads

Nodes	**Compact**	**Scatter**	**Eagermap**	**Treematch**	**ChoiceMap**
0-1	9.83	11.16	7.57	22.08	8.51
0-2	7.66	8.81	6.54	12.31	7.76
0-3	9.10	10.89	6.95	7.51	6.63
1-2	8.25	11.02	8.21	7.89	8.18
1-3	6.18	7.76	7.12	12.84	6.16
2-3	6.52	11.20	8.84	21.11	8.67
Sum	47.53	60.83	45.22	83.73	45.90
Average	7.92	10.14	7.54	13.96	7.65
Max	9.83	11.20	8.84	22.08	8.67
Min	6.18	7.76	6.54	7.51	6.16

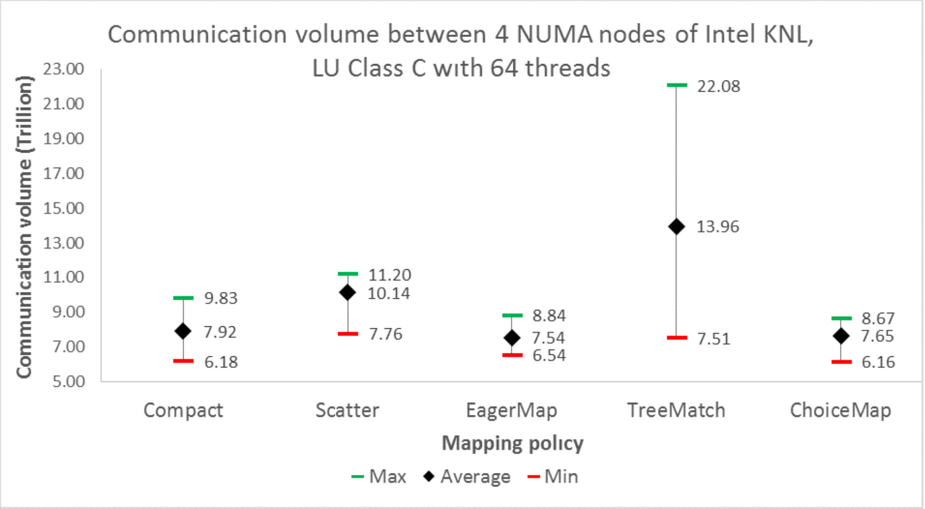


Figure 6.11: Comparison of inter NUMA node communication

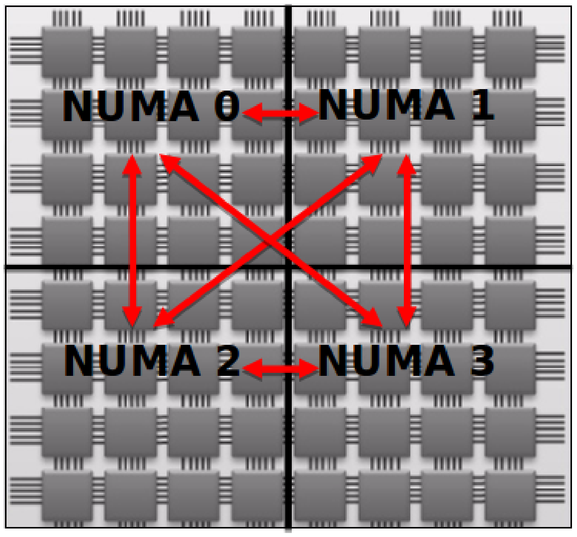


Figure 6.12: Communication between four NUMA nodes

Chapter 7

CONCLUSION

In a multicore machine, binding parallel tasks to cores according to a mapping policy that satisfies the application behavior is an important performance boosting factor. We introduce the BindMe tool that assists programmer to bind threads to cores for better performance. BindMe automatically explores machine topology and binds threads to processing units according to a mapping policy which is selected by the programmer. BindMe has a user friendly interface to bind threads to hardware with a granularity defined by the programmer. Our experiments show that there is no one best mapping policy for all kinds of applications. Therefore BindMe facilitates in selecting a mapping policy which performs better for an application on a given machine. We also present a mapping algorithm, ChoiceMap that does mapping more fairly by considering mutual priorities of tasks. Our experiments show that ChoiceMap generates pairs of good quality by reducing overall communication of resulting pairs and it also reduces inter-node communication volume. ChoiceMap is implemented in BindMe as one of the mapping policies.

BIBLIOGRAPHY

[Ang et al., 2014] Ang, J., Barrett, R., Benner, R., Burke, D., Chan, C., Cook, J., Donofrio, D., Hammond, S. D., Hemmert, K., Kelly, S., Le, H., Leung, V., Resnick, D., Rodrigues, A., Shalf, J., Stark, D. andŨnat, D., and Wright, N. (2014). Abstract Machine Models and Proxy Architectures for Exascale Computing. In *2014 Hardware-Software Co-Design for High Performance Computing*, pages 25–32. IEEE.

[Broquedis et al., 2010] Broquedis, F., Clet-Ortega, J., Moreaud, S., Furmento, N., Goglin, B., Mercier, G., Thibault, S., and Namyst, R. (2010). 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*, pages 180–186. IEEE.

[Che et al., 2009] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H., and Skadron, K. (2009). Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54. Ieee.

[Cruz et al., 2015] Cruz, E. H., Diener, M., Pilla, L. L., and Navaux, P. O. (2015). An efficient algorithm for communication-based task mapping. In *Parallel, Distributed and Network-Based Processing (PDP), 2015 23rd Euromicro International Conference on*, pages 207–214. IEEE.

[Dagum and Menon, 1998] Dagum, L. and Menon, R. (1998). Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55.

[Das et al., 2013] Das, R., Ausavarungnirun, R., Mutlu, O., Kumar, A., and Azimi, M. (2013). Application-to-core mapping policies to reduce memory system interference in multi-core systems. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 107–118. IEEE.

[Diener et al., 2016] Diener, M., Cruz, E. H., Alves, M. A., and Navaux, P. O. (2016). Communication in shared memory: Concepts, definitions, and efficient detection. In *Parallel, Distributed, and Network-Based Processing (PDP), 2016 24th Euromicro International Conference on*, pages 151–158. IEEE.

[Gropp et al., 1996] Gropp, W., Lusk, E., Doss, N., and Skjellum, A. (1996). A high-performance, portable implementation of the mpi message passing interface standard. *Parallel computing*, 22(6):789–828.

[Irving, 1985] Irving, R. W. (1985). An efficient algorithm for the stable roommates problem. *Journal of Algorithms*, 6(4):577–595.

[Jeannot et al., 2014] Jeannot, E., Mercier, G., and Tessier, F. (2014). Process placement in multicore clusters: Algorithmic issues and practical techniques. *IEEE Transactions on Parallel and Distributed Systems*, 25(4):993–1002.

[Jin et al., 1999] Jin, H.-Q., Frumkin, M., and Yan, J. (1999). The openmp implementation of nas parallel benchmarks and its performance.

[Long and Clarke, 1989] Long, D. L. and Clarke, L. A. (1989). Task interaction graphs for concurrency analysis. In *Proceedings of the 11th international conference on Software engineering*, pages 44–52. ACM.

[Luk et al., 2005] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. (2005). Pin: building customized program analysis tools with dynamic instrumentation. In *Acm sigplan notices*, volume 40, pages 190–200. ACM.

[McVitie and Wilson, 1971] McVitie, D. G. and Wilson, L. B. (1971). The stable marriage problem. *Communications of the ACM*, 14(7):486–490.

[Poljak and Tuza, 1987] Poljak, S. and Tuza, Z. (1987). Maximum bipartite subgraphs of kneser graphs. *Graphs and Combinatorics*, 3(1):191–199.

[Shreedhar and Varghese, 1996] Shreedhar, M. and Varghese, G. (1996). Efficient fair queuing using deficit round-robin. *IEEE/ACM Transactions on networking*, 4(3):375–385.

[Sodani et al., 2016] Sodani, A., Gramunt, R., Corbal, J., Kim, H.-S., Vinod, K., Chinthamani, S., Hutsell, S., Agarwal, R., and Liu, Y.-C. (2016). Knights landing: Second-generation intel xeon phi product. *IEEE Micro*, 36(2):34–46.