Received: 7 February 2018 | Revised: 12 April 2018 | Accepted: 1 May 2018
DOI: 10.1002/cpe.4692

WILEY

SPECIAL ISSUE PAPER

BindMe: A thread binding library with advanced mapping algorithms

Pirah Noor Soomro | Muhammad Aditya Sasongko | Didem Unat

Computer Science and Engineering, Koç University, Istanbul, Turkey

Correspondence
Pirah Noor Soomro, Computer Science and Engineering, Koç University, Istanbul, Turkey.
Email: psoomro16@ku.edu.tr

Didem Unat, Computer Science and Engineering, Koç University, Istanbul, Turkey.
Email: dunat@ku.edu.tr

Funding information
The Scientific and Technological Research Council of Turkey, Grant/Award Number: 116C066; Higher Education Commission of Pakistan

Summary

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 platforms because each machine has a different topology and each application exhibits different communication pattern. Determining the best policy for a given application and machine requires extra programming effort. To relieve the programmer from that burden, we introduce 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 respects mutual priorities of parallel tasks and performs a fair mapping by reducing communication volume among cores. We have tested BindMe and ChoiceMap with various applications from NAS parallel benchmark and Rodinia bechmark. 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.

KEYWORDS

communication matrix, machine topology, mapping algorithms, multicore, task binding

1 | INTRODUCTION

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,[1-3] and it is expected that the number of cores on a chip will double every two to three years.[4] Having large number of cores on one chip causes variance in communication latency among cores due to distance between them.[5] As the number of cores increases in multicore architectures, task placement on cores becomes an important performance parameter as optimal placement would reduce the execution time and power consumption on the chip.

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.[6,7] Current state-of-the-art shared memory programming models such as OpenMP[8] provide thread 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 placement algorithm based on the communication pattern of the application. However, this solution is not portable because the programmer has to discover the details of machine topology and bind the tasks accordingly.

We develop the 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[9] to extract communication information from an application and leverages hwloc[10] to explore underlying hardware topology tree. Based on this information, it generates a mapping sequence

using state of the art algorithms, such as *TreeMatch*[11] and *EagerMap*.[12] 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.

Moreover, we propose a mapping algorithm, *ChoiceMap*, which models 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[13] 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.

We have tested the BindMe library and the *ChoiceMap* algorithm with the applications from NAS benchmark[14] and an image segmentation application from Rodinia benchmark.[15] 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. Moreover, *ChoiceMap* reduces the execution time of an application and improves the data locality by decreasing the off-core or off-node communication.

This paper has the following contributions.

- We have developed a framework, BindMe, that assists the programmer in thread binding on multicore architecture by combining machine topology and application characteristics.
- We propose a mapping algorithm, ChoiceMap, which pairs the threads based on their mutual preferences with respect to the communication frequency among them.
- We have tested BindMe and ChoiceMap with various applications from NAS and Rodinia benchmark suits. We have shown that no single mapping policy is best for all kinds of applications and machine topologies; however, BindMe makes it easy to perform different bindings to discover the best one.

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

2 | BACKGROUND

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 the mapping algorithm to generate a 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.

2.1 | 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.[16] As shown in Figure 1A, for a parallel application using 8 threads, the vertices represent threads and edges between two vertices represent

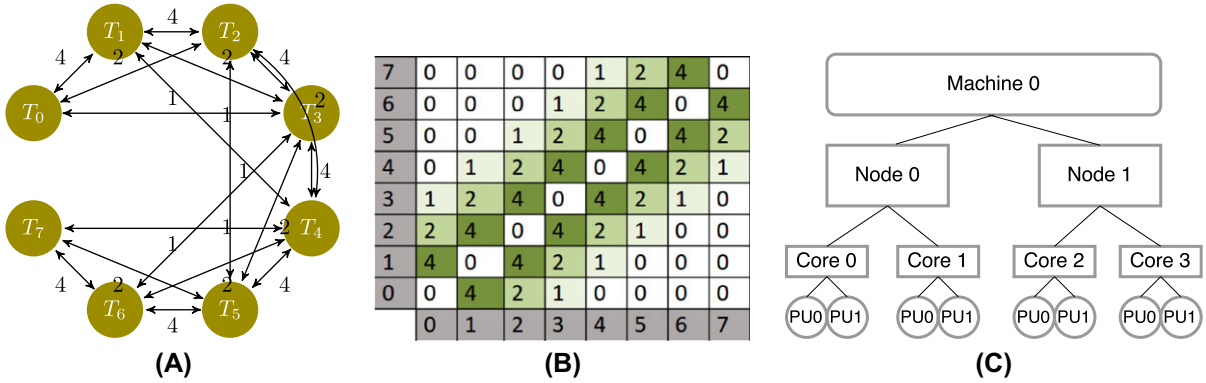

FIGURE 1 Communication matrix and machine topology examples. A, Weighted task interaction graph (TIG); B, Communication matrix of TIG; C, An example machine topology, Arity sequence = 1, 2, 2, 2

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 is shown in Figure 1B. Communication-aware mapping algorithms analyze communication matrix to generate a mapping sequence for a parallel application.

2.2 | 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 machine 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 1 shows an example machine topology for only its processing units, omitting storage units. The machine topology can be represented by an arity sequence. Arity refers to the number of children of a node. The arity sequence represents number of children of all the nodes in tree starting from the root. Mapping algorithm uses this arity sequence to determine the number of tasks in a single group at a given level of topology tree.

3 | EXISTING MAPPING SOLUTIONS

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 domains includes such policies that simply binds threads to cores as in OpenMP without taking the application behavior into account. The 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 communication-aware mapping, respectively.

3.1 | 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 in lowering the application performance. Next, we will consider three binding options supported by the Intel compiler.

Scatter: Figure 2 shows an example of the *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 2 shows an example of the *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 specifies whether processes can migrate throughout the execution or remain at their 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 paper, 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.

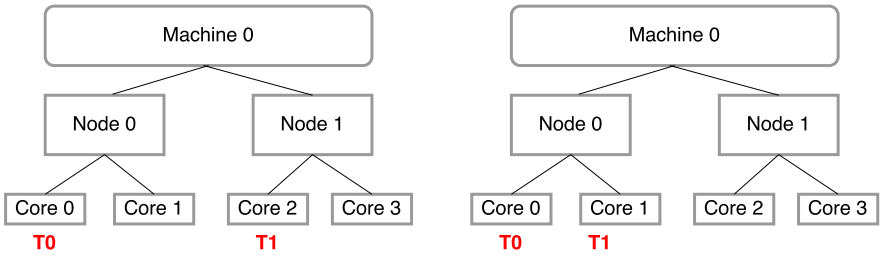

FIGURE 2 On the left, *Scatter* placement policy is shown; task 0 goes to core 0 on socket 0, task 1 goes to core 2 on socket 1. On the right, *Compact* placement policy is shown; both task 0 and 1 are placed on the same socket on core 0 and core 1

Thread binding can also be done by providing a list of cores to which threads should be bound. GOMP_CPU_AFFINITY environment variable is used to set cpu_list, which is a comma separated list of cores representing binding site of threads, respectively. For example, GOMP_CPU_AFFINITY= 0,2,1,3 means binding thread 0 to core 0, thread 1 to core 2, and so on. MPI[17] is another parallel programming framework provides options to bind processes to cores. Various binding switches are provided by the MPI process launcher (eg, mpirun or mpiexec). One of the switch is -bind-to-X, where X represents binding granularity such as node, core, or socket. MPI binds the processes to said granularity and signals operating system not to migrate the process out of that level. -bind-to-X switch can be used with combination of -byX switch, which represents the binding granularity by following a round robin mapping policy.[18] Another binding option in MPI is to specify custom binding strategy by using a -hostfile switch. 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. Similar to OpenMP, MPI requires the user to have some knowledge about machine topology and applications behavior.

3.2 | Communication-aware mapping algorithms

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 mapping algorithms available in literature, which analyzes communication 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.

3.2.1 | TreeMatch algorithm

TreeMatch applies two main steps to generate a mapping sequence. First, all the possible combinations for a 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 represent 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[19] in the work of Poljak and Tuza. 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 sites of threads corresponding to an index of sequence.

3.2.2 | EagerMap algorithm

EagerMap 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 in each level. *EagerMap* applies the same procedure iteratively to generate a mapping sequence for a given machine topology.

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. *TreeMatch* generates a mapping sequence with an exhaustive search approach, where all communications are considered to pick the best ones. Considering these shortcomings, we designed our own algorithm for communication-aware mapping discussed in the next section.

4 | CHOICEMAP: A FAIR TASK MAPPING ALGORITHM

ChoiceMap leverages the communication matrix and machine topology for determining 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. Pairing threads according to their choice list results in a mapping sequence that reduces the communication distances. This problem resembles genderless stable marriage problem[20] or roommate matching problem[13] in the work of Irving, 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 $n \times (n - 1)$ 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. However, 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.

4.1 | ChoiceMap algorithm

Algorithm 1 presents the *ChoiceMap* algorithm, which takes two inputs, ie, *aritySequence* and *commMatrix*. The *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 is generated from the communication matrix. We assume that total number of tasks is equal to the number of processing units in the machine. Starting from line 10, for every *task*, we first compute the nearest choice from *choiceMatrix* and name it as *candidate*. The nearest choice means the most desired choice of a given thread among remaining unpaired threads. The nearest choice of a thread changes over the course of the execution of the algorithm once its first choice gets paired with some other thread. We check if the *task* is also nearest choice of *candidate*, then we pair *task* and *candidate*. After pairing, both tasks are removed from the search space, thus invalidating them to be a candidate of 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.

Algorithm 1 ChoiceMap algorithm

```
1:  procedure ChoiceMap(aritySequence, commMatrix)
2:      mappingSequnce := { }                                   ▷ set of pairs
3:      depth ← getDepth(aritySequence)                         ▷ total levels of machine topology
4:      for level in {depth − 1..0} do
5:          n_sites ← getTotalSites(aritySequence, level)       ▷ get number of biding sites at given level
6:          choiceMatrix ← generateChoiceMatrix(commMatrix, n_sites)
7:          paired := 0                                         ▷ number of tasks paired in a pass
8:          while paired ≤ n_sites do                           ▷ loop untill all tasks get paired
9:              prev_paired ← paired
10:             for task in {0..n_sites − 1} do
11:                 candidate ← getNearestChoice(task, choiceMatrix)   ▷ get most desired unpaired task
12:                 if task == getNearestChoice(candidate, choiceMatrix) then
13:                     mappingSequence := mappingSequence ∪ {pair(task, candidate)}
14:                     invalidate(task, candidate, choiceMatrix)      ▷ remove this pair from search space
15:                     paired := paired + 2
16:                 end if
17:             end for
18:             if prev_paired == paired then                  ▷ if no tasks get paired during the pass, resolve cycle
19:                 resolveCycle(commMatrix, choiceMatrix, mappingSequence)
20:             end if
21:         end while
22:         aggregate(commMatrix, mappingSequence, level + 1)  ▷ advance to next level at machine hierarchy
23:     end for
24:     return mappingSequence
25: end procedure
```

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, task t_1 chooses t_2, t_2 chooses t_3, and t_3 chooses t_1 as its first priority. In order to proceed further, we need to detect cycle and break the loop, as shown in line 19 in Algorithm 1. For solving the loop, we first select a task from the cycle (say, α). For the sake of simplicity, we select α as the smallest task number in the cycle. To break the cycle, we pair the task α to one of its desirers that communicates the most with it. After pairing these two, we invalidate the paired tasks. This can be repeated until choice matrix is cycle-free.

Pairing tasks only at the deepest level of hierarchy does not fully satisfy task affinity. As in a parallel application, communication is not limited between two tasks. 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 line 22 in Algorithm 1. 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

$$A(i', j') = C\left(i'(i), j'(i)\right) + C\left(i'(i), j'(j)\right) + C\left(i'(j), j'(i)\right) + C\left(i'(j), j'(j)\right), \tag{1}$$

where i' and j' represent ith and jth task pair from previous level, respectively. Note that pairing tasks is a bottom up procedure. Once aggregated, the

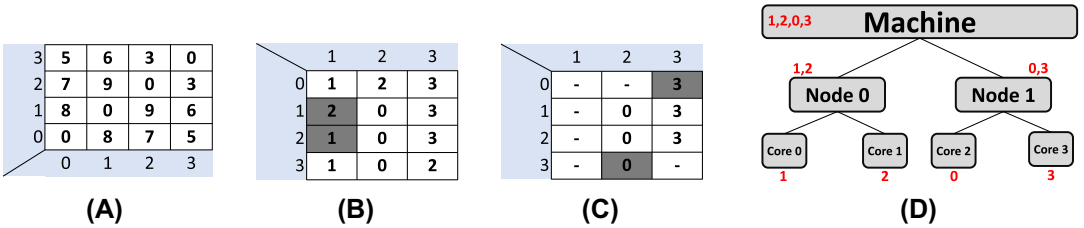

FIGURE 3 *ChoiceMap* example with four tasks. A, Communication matrix; B, Choice Matrix: Pair 1 and 2; C, Choice Matrix: Pair 0 and 3; D, Mapping on machine topology

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 | Example

We now explain *ChoiceMap* with an example shown in Figure 3. There are four tasks to be mapped on example topology. We first generate choice matrix (Figure 3B) from communication matrix (Figure 3A). The first choice of task 0 is task 1; however, task 1's first choice is not task 0. Therefore, task 0 can not be paired with task 1 as they do not mutually prioritize each other at the same rank. We further find that tasks 1 and 2 are first choices of each other so they are paired and removed from the choice matrix. After pairing tasks 1 and 2, the choice matrix now contains only two threads, and they are also nearest choice of each other; thus, tasks 0 and 3 are paired together. Once all tasks are paired for a given level, the algorithm repeats itself for upper levels of machine topology. Note that, except at last level, a group of tasks is considered as a single virtual task at each level of machine topology. The communication matrix is also aggregated for the upper levels according to the groups of tasks.

5 | BINDME FRAMEWORK

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 into a single library call. Figure 4 gives an overview of the BindMe library. BindMe generates machine topology by using the *hwloc* tool.[10] 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.

BindMe packs all the complexities of a binding process in just one function call, namely, *bindme()*. An overview of the function is given in Figure 4. 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.

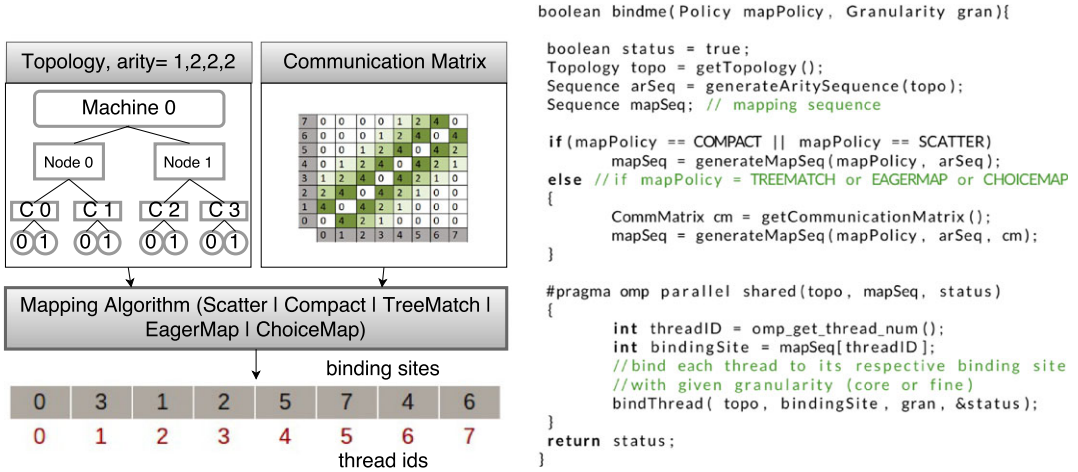

FIGURE 4 Overview of BindMe and source code of BindMe (details are omitted for clarity)

In the internals of the function, we first create the topology tree of underlying machine using the hwloc tool, which discovers all the memory and execution units. An arity sequence is generated from the machine topology. 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 (eg, *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 ID.

Bindme() is called by specifying a mapping policy and granularity. Mapping policies can be *Scatter*, *Compact*, *EagerMap*, *TreeMatch*, or *ChoiceMap*. Binding granularity options available in BindMe are core and fine, where core binds threads to cores and fine binds threads to logical processing units (ie, hyperthreads). Since the communication matrix is generated beforehand, this overhead is not applied to every run. The overhead of generating mapping sequence is negligible and independent of application execution time because it depends on the number of threads and depth of machine topology. In the worst case, the complexity of mapping algorithms is $O(L * N^2)$,[13] where N is the number of threads and L is the number of levels in the topology tree, both of which are relatively small in current multicore architectures.

5.1 | Supporting tools

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. Istopo, a command line tool of hwloc, provides graphical representation of the hardware resources. 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.

Numalize

Numalize[9] is a tool for detecting communication in shared memory. It is a cache simulator based on the Pin dynamic binary instrumentation tool.[21] 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 communication when different threads access the same cache line. Since the communication is captured on the granularity of cache line, communication matrix should be collected by executing the application on the same platform for which mapping sequence is being generated. 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 slows down by about 100 times the normal execution time because *Numalize* monitors every load and store in memory. Unfortunately, in some cases, *Numalize* crashes because of the large memory requirements of an application in addition to its own internal data structures.

6 | EVALUATION

We evaluate BindMe using applications from various scientific domains with mapping policies supported in BindMe. We used two platforms in our experiments. One is Intel KNL (KNights Landing) and the other is Intel Broadwell. Figures 5A and 5B 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. In total, there are 68 cores (272 hyperthreads) divided as 16 and 17 cores per node. Two cores are organized as a single tile, and each core has 4 hyperthreads. Figure 5B shows machine topology of Broadwell. It has 2 NUMA nodes, with 10 cores on each node. We have used 8 cores per NUMA node on Broadwell and 16 cores per NUMA on KNL because our mapping algorithms works with power of 2 execution units.

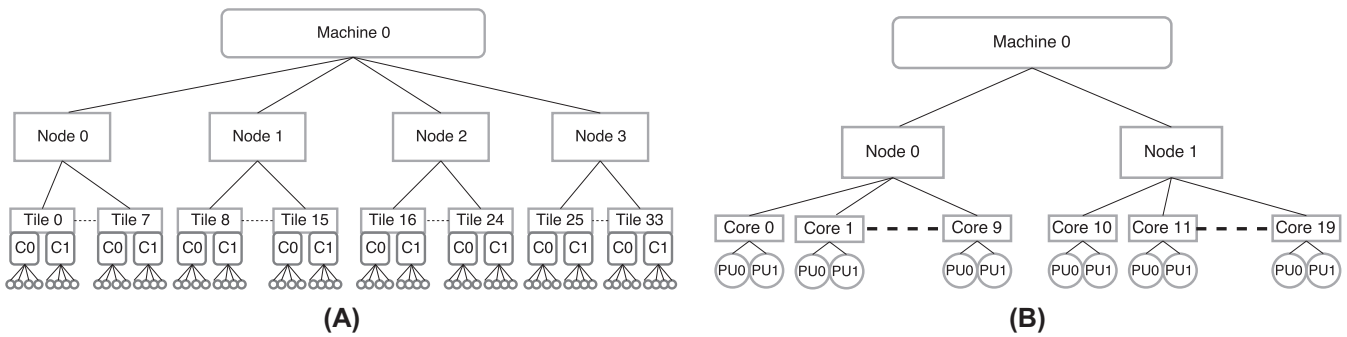

FIGURE 5 Schematic diagrams of test machines. A, Intel KNL; B, Intel Broadwell

TABLE 1 Brief description and input sizes of test applications

Benchmarks	Input class
LU: Solves regular-sparse lower and upper triangular systems from CFD	B (102^3), C (162^3)
SP: It computes independent systems of non-diagonally dominant scalar pentadiagonal equations	B (102^3), C (162^3)
CG: Conjugate gradient method used to find smallest eigenvalues of a large, sparse, and symmetric matrix	B (75000),C (150000)
BT: solves multiple independent systems of non diagonally dominant block tridiagonal equations	B (102^3), C (162^3)
MG: a simplified multigrid calculation. It processes highly structured long distance communication and tests both short and long distance data communications	C (512^3)
Image Segmentation: generates clusters of same pixel colours in the image	Image of 9000 × 7000 pixels

6.1 | Test applications

We used five applications from NAS Parallel Benchmarks,[14] ie, SP, LU, BT, CG, and MG. SP (Scalar Pentadiagonal 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.

Moreover, we use an image segmentation application from the Rodinia benchmarks.[15] The application generates clusters of same pixel colours in an image. The communication behavior changes based on the input image but remains the same from run to run if provided the same input image. Table 1 summarizes the description of the benchmarks used for evaluation. We have used B and C classes of NAS applications, which represent the problem size of test applications. On KNL, we tested NAS applications with Class C with 64 (granularity = *core*) and 256 (granularity = *fine*) threads. Appendix provides the communication matrices for all the applications on Broadwell and KNL (Figures A1 and A2).

6.2 | Performance of mapping policies

A good mapping policy results in reduction of inter-core communication and better cache utilization, thus reducing execution time. Figure 6 compares normalized execution time with respect to the worst mapping policy. We tested five NAS benchmarks executed with five different mapping

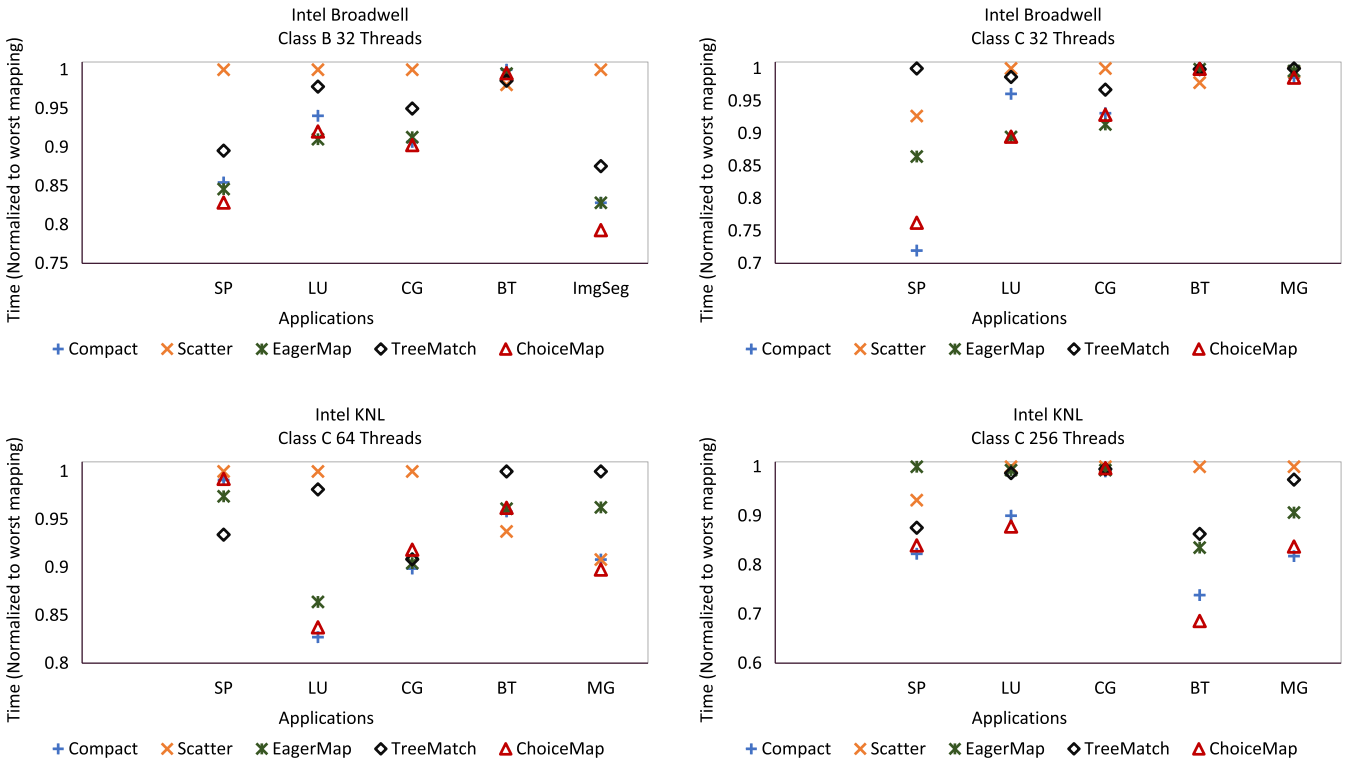

FIGURE 6 Execution time of applications tested on Broadwell and KNL

options available in BindMe, ie, *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 in which near neighborhood communication pattern is dominant. *Scatter* balances computation load; it performs 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 mapping sequence. *ChoiceMap* performs better in many cases. Applications with a significant all to all communication may not get benefit from *ChoiceMap* since every thread performs similar amount of communication with all threads. However, in applications with a more distinctive communication pattern, *ChoiceMap* results in better performance. Image segmentation code is one of the examples, which shows such a communication pattern, and *ChoiceMap* generates a best mapping sequence as in Figure 6, ie, Intel Broadwell Class B 32 Threads. In some cases, *EagerMap* and *ChoiceMap* generated almost the same mapping sequence, thus performing the same. *TreeMatch* did not perform well overall, but in some cases, (SP with 64 threads on Intel KNL) it is better than other mapping policies.

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

CG performs an intense all to all communication along with an irregular pattern, which highlights the most communicating thread pairs. 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 shows that the mapping sequence generated by mapping algorithms is better than general mapping policies (*Compact* and *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 6A and 6B show 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 captured the affinities in the case of MG. However, on Broadwell, there is no clear winner for MG. By comparing execution time of MG class C executed on Intel Broadwell and Intel KNL, it can be concluded that the winner mapping policy for the same application on different machines can be different (Figure 6). Therefore, a mapping policy must be selected by trying all the options. With BindMe, it is relatively easy to test different mapping options in order to pick the best performing one.

6.3 | Locality measures

To compare the quality of pairs generated by the mapping algorithms, we calculated the communication reduction value. This formula is previously introduced in *TreeMatch* algorithm to generate pairs. Communication reduction value C_{rv} of a pair (i, j) is calculated by the Equation 2.

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

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 the smallest C_{rv} value since C_{rv} represents the amount of communication that the thread pair has to perform with all the other threads. The worst pair is the one with the largest C_{rv} value. We generate all possible combinations of N tasks, and pick the best and worst pairs 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 task can be paired only once.

Mapping with better communication reduction value results in improved data locality[22] as a large volume of communication is done inside the node at a given level and reduces data movement across farther nodes. Figure 7 shows that, for every task, *ChoiceMap* is almost the same as best pairing. *EagerMap* and *ChoiceMap* generated almost the same pairs for SP on Broadwell at leaf level; therefore, both algorithms resulted in a better mapping sequence. It is also reflected from the execution time of SP class C on Broadwell (Figure 6B). The difference in execution time is caused by mapping done at upper levels of the machine topology. In Figure 7, image segmentation application shows a slight difference in *ChoiceMap* and *EagerMap* pairing. The irregular communication pattern of the application led to difference in pairing across mapping algorithms. The better pairing of *ChoiceMap* gives reduced execution time of the application, as shown in Figure 6A. In some cases, however, *ChoiceMap* performs worst than *TreeMatch* and *EagerMap*. For example, in Figure 7A, *ChoiceMap* does not generate the best pair for threads 4, 9, and 30, whereas *EagerMap* generates the best pair for these threads because the goal of *ChoiceMap* is to perform a fair pairing policy, which may not result

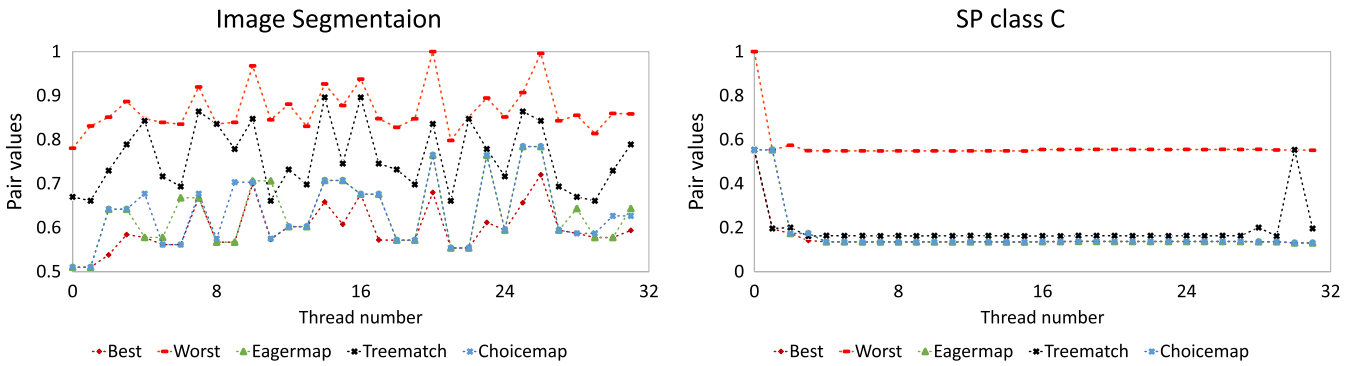

FIGURE 7 Comparison of pair values generated by *EagerMap*, *TreeMatch*, and *ChoiceMap* with respect to best and worst pair values

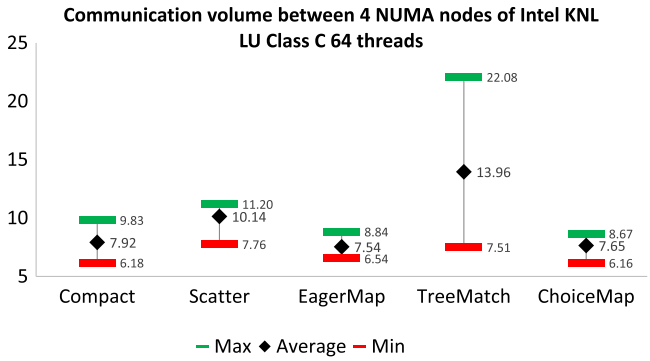

FIGURE 8 Comparison of inter-NUMA node communication. Volume represents the cache line sharing events

in best performance. Other mapping algorithms, due to the difference in the nature of mapping policy (greedy or exhaustive search), may succeed to generate a better pair resulting into better performance than *ChoiceMap*. It is also important to note that performance can be affected by thread private memory accesses. Current mapping algorithms take the locality into account but not the total memory accesses performed by the sockets, which may result in imbalanced memory bandwidth usage between sockets. We plan to implement a balanced strategy in our future work.

To test the quality of mapping policies, we compare communication amount between the NUMA nodes after mapping is performed at the lower levels of the topology. A good mapping policy should reduce the amount of communication across the NUMA nodes, which means most of the communication should be performed inside the NUMA node, resulting in improved data locality. To compare communication volume across the NUNA nodes, we tested communication matrix of LU (class C) generated on KNL with 64 threads using 4 NUMA nodes and 16 cores per node. Figure 8 compares the average, minimum, and maximum inter-NUMA node communication with respect to the applied mapping. It can be observed that *ChoiceMap* results in the least communication volume for inter-NUMA node communication. This improvement is also reflected at the execution time of LU on KNL.

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 have introduced 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 presented 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.

ACKNOWLEDGMENTS

Unat is supported by the Scientific and Technological Research Council of Turkey under project 116C066. Soomro is supported by Higher Education Commission of Pakistan under the HRDI-UESTP scholarship for MS leading to PhD.

ORCID


Pirah Noor Soomro http://orcid.org/0000-0001-8654-3249

REFERENCES

1. Sodani A, Gramunt R, Corbal J, et al. Knights landing: second-generation Intel Xeon Phi product. *IEEE Micro*. 2016;36(2):34-46.
2. Shalf J, Dosanjh S, Morrison J. Exascale computing technology challenges. Paper presented at: International Conference on High Performance Computing for Computational Science; Springer; 2010; Berkeley, CA.
3. Duran A, Klemm M. The Intel® many integrated core architecture. Paper presented at: 2012 International Conference on High Performance Computing and Simulation (HPCS); 2012; Madrid, Spain.
4. Ang JA, Barrett RF, Benner RE, et al. Abstract machine models and proxy architectures for exascale computing. Paper presented at: 2014 Hardware-Software Co-Design for High Performance Computing; 2014; New Orleans, LA.
5. Das R, Ausavarungnirun R, Mutlu O, Kumar A, Azimi M. Application-to-core mapping policies to reduce memory system interference in multi-core systems. Paper presented at: 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA); 2013; Shenzhen, China.
6. Diener M, Cruz EHM, Navaux POA. Communication-based mapping using shared pages. Paper presented at: 2013 IEEE 27th International Symposium on Parallel & Distributed Processing (IPDPS); 2013; Boston, MA.
7. Diener M, Cruz EHM, Alves MAZ, Alhakeem MS, Navaux POA, Heiß H-U. Locality and balance for communication-aware thread mapping in multicore systems. Paper presented at: European Conference on Parallel Processing; 2015; Vienna, Austria.
8. Dagum L, Menon R. OpenMP: an industry standard API for shared-memory programming. *IEE Comput Sci Eng*. 1998;5(1):46-55.
9. Diener M, Cruz EHM, Alves MAZ, Navaux POA. Communication in shared memory: Concepts, definitions, and efficient detection. Paper presented at: 2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP); 2016; Heraklion, Greece.
10. Broquedis F, Clet-Ortega J, Moreaud S, et al. hwloc: A generic framework for managing hardware affinities in HPC applications. Paper presented at: 2010 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP); 2010; Pisa, Italy.
11. Jeannot E, Mercier G, Tessier F. Process placement in multicore clusters: algorithmic issues and practical techniques. *IEEE Trans Parallel Distrib Syst*. 2014;25(4):993-1002.
12. Cruz EHM, Diener M, Pilla LL, Navaux POA. An efficient algorithm for communication-based task mapping. Paper presented at: 2015 23rd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP); 2015; Turku, Finland.
13. Irving RW. An efficient algorithm for the stable roommates problem. *J Algorithms*. 1985;6(4):577-595.
14. Jin H-Q, Frumkin M, Yan J. The OpenMP Implementation of NAS Parallel Benchmarks and its Performance. Technical Report. Moffett Field, CA: NASA; 1999.
15. Che S, Boyer M, Meng J, et al. Rodinia: A benchmark suite for heterogeneous computing. Paper presented at: 2009 IEEE International Symposium on Workload Characterization (IISWC); 2009; Austin, TX.
16. Long DL, Clarke LA. Task interaction graphs for concurrency analysis. In: Proceedings of the 11th International Conference on Software engineering; 1989; Pittsburgh, PA.
17. Gropp W, Lusk E, Doss N, Skjellum A. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Comput*. 1996;22(6):789-828.
18. Shreedhar M, Varghese G. Efficient fair queuing using deficit round-robin. *IEEE/ACM Trans Netw*. 1996;4(3):375-385.
19. Poljak S, Tuza Z. Maximum bipartite subgraphs of Kneser graphs. *Graphs Comb*. 1987;3(1):191-199.
20. McVitie DG, Wilson LB. The stable marriage problem. *Commun ACM*. 1971;14(7):486-490.
21. Reddi VJ, Settle A, Connors DA, Cohn RS. PIN: a binary instrumentation tool for computer architecture research and education. In: Proceedings of the 2004 Workshop on Computer Architecture Education: Held in Conjunction with the 31st International Symposium on Computer Architecture; 2004; Munich, Germany.
22. Unat D, Dubey A, Hoefler T, et al. Trends in data locality abstractions for HPC systems. *IEEE Trans Parallel Distrib Syst*. 2017;28(10):3007-3020.

How to cite this article: Soomro PN, Sasongko MA, Unat D. BindMe: A thread binding library with advanced mapping algorithms. *Concurrency Computat Pract Exper*. 2018;30:e4692. https://doi.org/10.1002/cpe.4692

APPENDIX

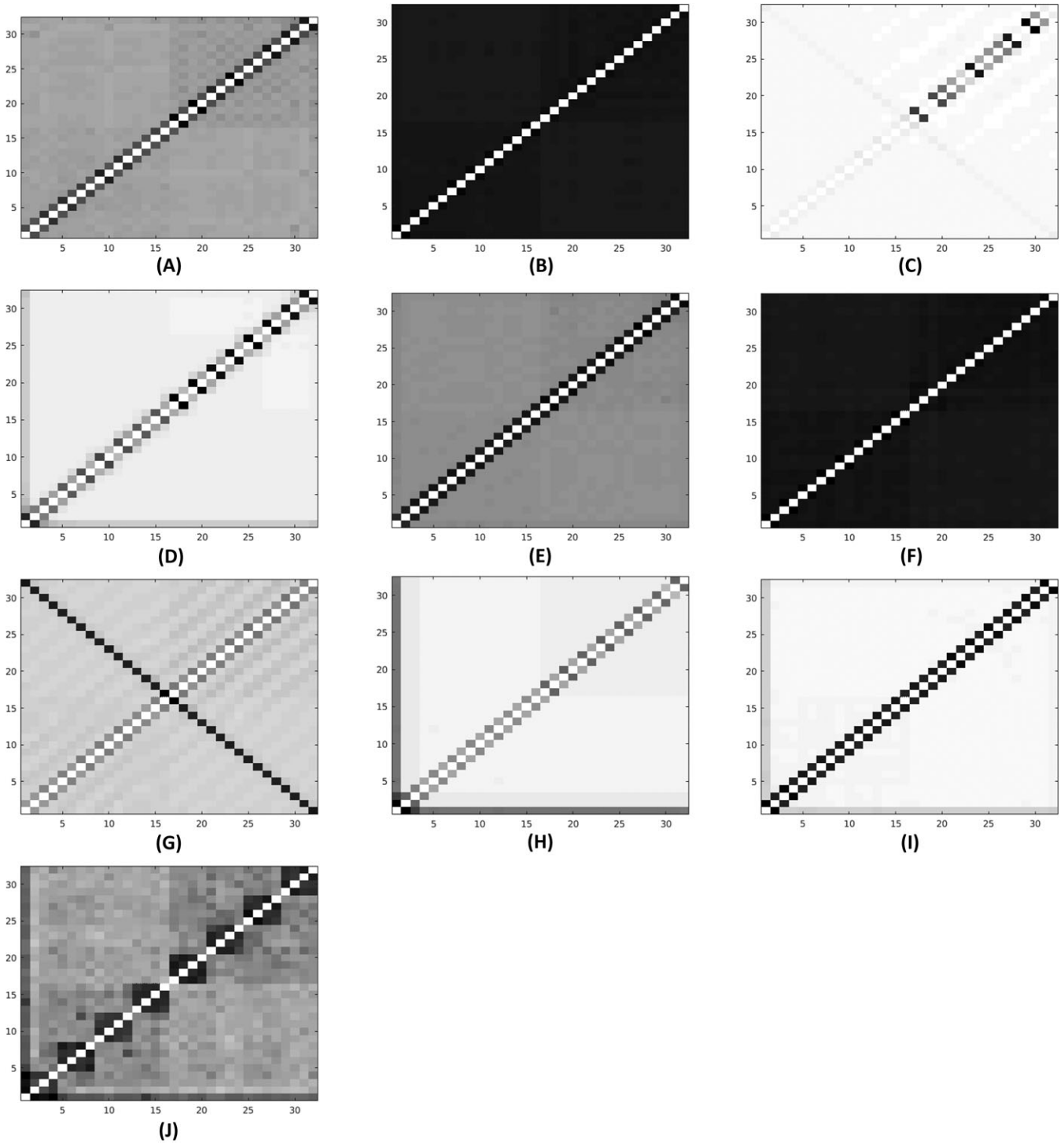

FIGURE A1 Communication matrices of BT, CG, LU, SP, and MG class B and C on Broadwell with 32 threads. We did not include MG Class B here because its execution time is too short for a reliable evaluation

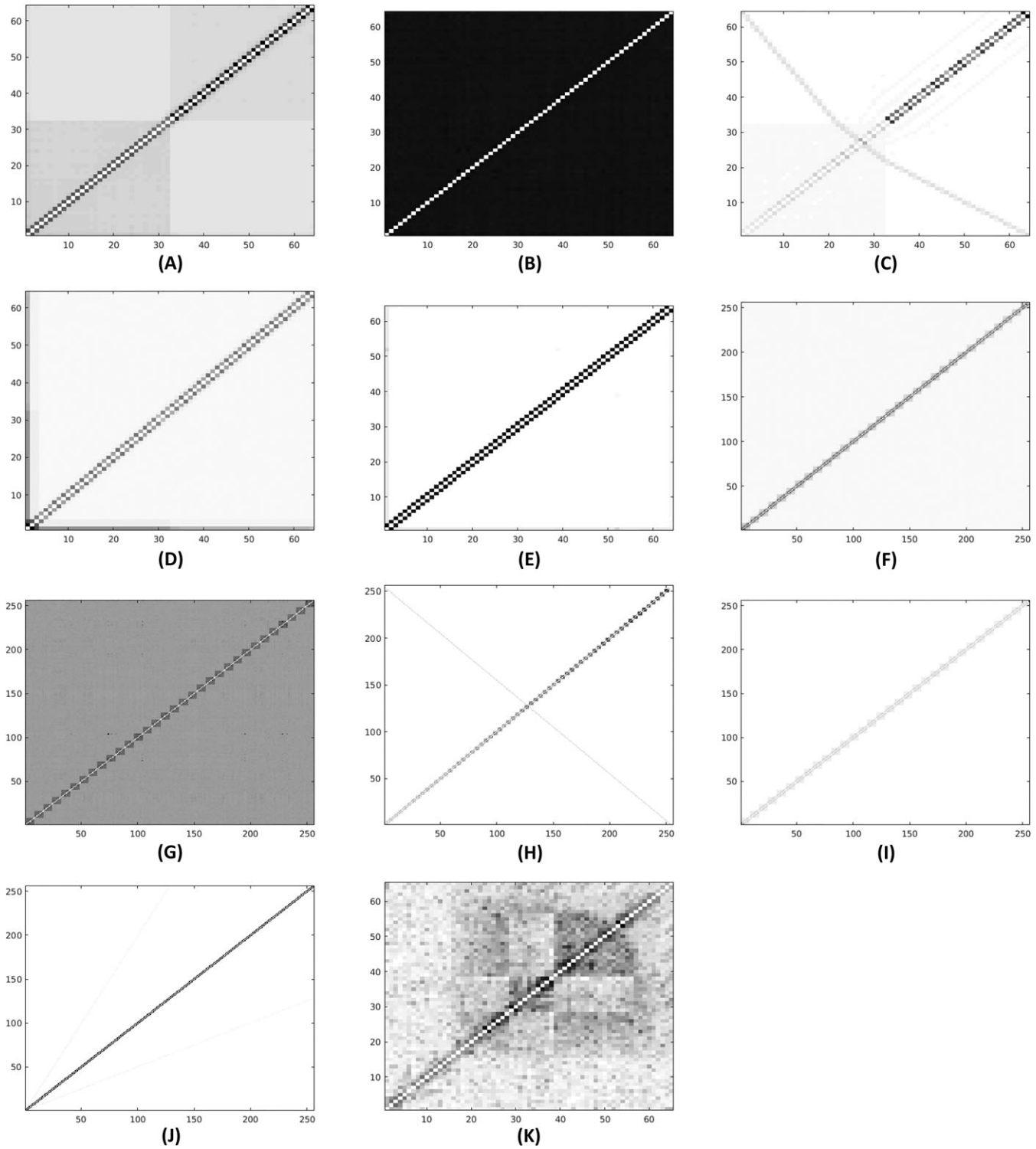

FIGURE A2 Communication matrices of BT, CG, LU, SP, and MG on KNL with fine and core granularity