EagerMap: A Task Mapping Algorithm to Improve Communication and Load Balancing in Clusters of Multicore Systems

Eduardo Cruz, Matthias Diener, Laércio Lima Pilla, Philippe Navaux

EagerMap: A task mapping algorithm to improve communication and load balancing in clusters of multicore systems

Eduardo H. M. Cruz[1], Matthias Diener[2], Laércio L. Pilla[3], and Philippe O. A. Navaux[4]

[1]Federal Institute of Parana (IFPR) – Paranavai, Brazil
[2]University of Illinois at Urbana-Champaign – Champaign, USA
[3]Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG – Grenoble, France
[4]Informatics Institute, Federal University of Rio Grande do Sul (UFRGS) – Porto Alegre, Brazil

Abstract

Communication between tasks and load imbalance have been identified as a major challenge for the performance and energy efficiency of parallel applications. A common way to improve communication is to increase its locality, that is, to reduce the distances of data transfers, prioritizing the usage of faster and more efficient local interconnections over remote ones. Regarding load imbalance, cores should execute a similar amount of work. An important problem to be solved in this context is how to determine an optimized mapping of tasks to cluster nodes and cores that increases the overall locality and load balancing. In this paper, we propose the EagerMap algorithm to determine task mappings, which is based on a greedy heuristic to match application communication patterns to hardware hierarchies and which can also consider the task load. Compared to previous algorithms, EagerMap is faster, scales better, and supports more types of computer systems, while maintaining the same or better quality of the determined task mapping. EagerMap is therefore an interesting choice for task mapping on a variety of modern parallel architectures.

1 Introduction

Optimizing the execution of parallel applications has become an important research topic in recent years [5, 19]. Modern parallel systems consist of many interconnected cluster nodes that themselves constitute a parallel shared-memory machine with a deep memory hierarchy. In such architectures, communication and load imbalance can have a higher impact on application performance than computation [39, 18]. A common way to reduce the impact of communication is to improve the *locality* of communication, by increasing the use of local interconnections and reducing the use of remote ones [42, 39]. Load imbalance can

be improved by using dedicated load balancing algorithms. These optimizations can be performed in parallel applications that use message passing libraries such as MPI [12], but also in applications that use the shared memory paradigm, such as OpenMP or UPC [2, 1].

In both types of parallel applications, the communication performance between the tasks of a parallel application are influenced by the levels of the hierarchy [43]. Communication through a shared cache memory or intra-chip interconnection is faster than communication between processors due to the slower inter-chip interconnections [35, 14]. Likewise, communication within a machine is faster than the communication between nodes in a cluster or grid. The network speeds also vary, increasing the difference in the communication latencies and bandwidths. Clusters and grids impose an additional challenge, since each node can have a different machine configuration. Tasks that communicate intensely should be mapped to PUs close together in the hierarchy. To improve load balance, we need to analyze the load of each task of the parallel application and map them to the cores in such a way that the loads of all cores are similar. In this context, the mapping of tasks to processing units (PUs) plays a key role in the performance of parallel applications [44].

Four main steps are required in the process of mapping tasks to the architecture. The first step is to detect the communication between the tasks. In message passage environments, the messages sent between the tasks need to be monitored. The second step is to detect the topology of the hierarchy, which is also highly dependent on the type of architecture. In shared memory architectures, tools such as hwloc [11] can be used, while most cluster environments have vendor-specific tools to manage topologies. The third step is to use a mapping algorithm to generate a mapping of tasks to PUs, combining communication, load, and topology information. The final step is to execute the application with the determined mapping, or migrate tasks to their assigned PUs.

This paper focuses on the third step: the mapping algorithm. The task mapping problem can be defined as follows [14]. Consider two graphs, one representing the parallel application, and one representing the parallel architecture. In the application graph, vertices represent tasks and edges represent the amount of communication between them. In the architecture graph, vertices represent the machine components, including the PUs, cache memories, NUMA nodes, network routers, switches, and others organized hierarchically, while edges represent the links' bandwidth and latency. The task mapping problem consists of finding a mapping of the tasks in the application graph to the PUs in the architecture graph, such that the total communication cost is minimized and the load on the cores is even.

The complexity of finding an optimal mapping is NP-Hard [9]. Due to the high number of tasks and PUs, finding an optimal mapping for an application is unfeasible. Heuristics are therefore employed to compute an approximation of the optimal mapping. However, current mapping algorithms still present a high execution time, since they were developed focusing on static mappings and are mostly based on complex analysis of the graphs. This reduces their applicability, especially for online mapping, since their overhead may harm performance.

In this paper, we propose *EagerMap*, an efficient algorithm to generate task mappings. It coarsens the application graph by grouping tasks that communicate intensely. This coarsening follows the topology of the architecture hierarchy. Based on observations of application behavior, we propose an efficient greedy

strategy to generate each group. It achieves a high accuracy and is faster than other approaches of the state of the art. After the coarsening, we map the group graph to the architecture graph.

EagerMap was initially proposed in [15]. The main extensions and improvements presented in this paper are the following.

- The original algorithm supported symmetric tree topologies only, which limited its applicability mostly to shared memory machines. We extended it to also support clusters and grids, where the topology of the network that connects the cluster/grid is arbitrary and the topology internal to each compute node is a symmetric tree.
- We also designed a parallel version of EagerMap, allowing it to calculate mappings faster.
- We added load balancing support to EagerMap that can handle oversubscribed scenarios (more threads/processes than processing units).

2 Related Work

Previous studies evaluate the impact of task mapping considering communication [43], showing that it can influence several hardware resources. In shared memory environments, communication-based task mapping reduces execution time, cache misses and interconnection traffic [16]. In the context of cluster and grid environments, mapping tasks that communicate to the same computing node reduces network traffic and execution time [10, 24]. Communication costs can also be minimized in virtualized environments [40], demonstrating its importance for cloud computing.

Several mapping algorithms have been proposed to optimize communication and load balancing. Most traditional algorithms are based on graph partitioning, such as Scotch [38]. The algorithm is based on the idea of divide and conquer, recursively allocating subsets of processes to subsets of PUs. The algorithm bipartitions an unprocessed set of PUs into two disjoint subsets, and calls a graph bipartitioning algorithm to split the subset of tasks associated with the PUs across the two subsets. As bipartitionings are performed recursively, the set sizes decrease, until all sets have only one element. A disadvantage of Scotch is that the topology description requires information about the communication latency between the PUs, and such values have a high influence in the final mapping. This presents a challenge for Scotch, since it is difficult to determine an accurate value for its parameters. On the other hand, EagerMap does not depend on such information, which makes it easier to use and can cause more reliable results. Other works that follow the same basic idea of Scotch are Zoltan [20], METIS [33, 31], and Chaco [23]. The work described in [22] also uses graph partitioning to group tasks, but it uses a greedy technique to map the groups to the topology. Some of these algorithms have been parallelized, such as PT-Scotch [13] and Par-METIS [32].

Tree representations are used in TreeMatch [26], which can lead to more optimized algorithms in architectures that can be represented in this way. However, the algorithm used in TreeMatch to group tasks has an exponential complexity because it generates all possible groups of tasks for each level of the memory

hierarchy. This can lead to unreasonable mapping calculation time depending on the number of tasks. Another limitation of TreeMatch is that it is not able to calculate mapping for most clusters or grids, as their topology may not be represented by a tree. TreeMatch has also been ported to the Charm++ environment [27], where support for load balancing was included. [4] also proposes a load balancer for Charm++ with a focus on the integrated OpenMP runtime.

MPIPP [12] is a framework to find optimized mappings for MPI-based applications. MPIPP initially maps each task to a random PU. At each iteration, MPIPP selects pairs of tasks to exchange PUs to reduce communication cost as much as possible. The communication costs depend on chosen parameters, and can be related to the number of messages or amount of data transferred between the MPI processes. This process is repeated several times, improving the quality of the mapping on each iteration, and can go on until no gains are achieved. One of the main drawbacks of MPIPP is that it depends too much on the initial random mapping, which does not have any guarantees regarding the mapping quality. Due to this, MPIPP can generate bad mappings. Other works that are based on the concept of refining an initial mapping at each iteration are [8] and [30].

Our previous work [14] uses Edmonds' graph matching algorithm [21] to calculate mappings, which solves the maximum weight perfect matching for complete weighted graphs to generate mappings. The idea of the proposal is to call the graph matching algorithm several times to generate the groups of threads that should be mapped together. Each time the matching algorithm is called, it generates pairs of threads that have a high degree of communication. Since it always generates pairs of threads, the solution is limited to environments where the number of tasks and PUs is a power of two. Another problem is that the proposed technique must be coded for a particular number of threads and architecture, which does not work automatically.

3 EagerMap: Greedy Hierarchical Mapping

EagerMap receives three pieces of input: a communication matrix containing the amount of communication between each pair of tasks, the load of each task and a description of the architecture hierarchy. It outputs which PU executes each task. To represent the architecture hierarchy, we use a tree, in which the vertices represent objects such as PUs and cache memories, and the edges represent the links between them. Our task grouping is performed with an efficient greedy strategy that is based on an analysis of the communication pattern of parallel applications. Fig. 1 depicts the different communication patterns of several parallel benchmark suites [3, 28, 36, 7], which we obtained using the methodology described in Section 4.2. We observe three essential characteristics in the communication behavior of the applications that need to be considered for an efficient mapping strategy:

1. There are two types of communication behavior: structured and unstructured communication. In applications with structured communication, each task communicates more with a subgroup of tasks, such that mapping these subgroups to PUs nearby in the hierarchy can improve performance. In Fig. 1, all applications except Vips show structured communication patterns. Our mapping algorithm is designed to handle structured communication patterns, be-

cause in applications with unstructured communication, there may not be a task mapping that can improve performance.

2. In applications with structured communication patterns, the size of the subgroups with intense internal communication is usually small when compared to the total number of tasks in the parallel application. For instance, in the communication pattern of CG-MPI (Fig. 1b), subgroups of 8 tasks communicate intensely, out of 64 tasks in total.

3. The amount of communication within each subgroup is much higher than the amount of communication between different subgroups.

In this section, we describe EagerMap in detail, give an example of its operation and discuss its complexity. We first describe the algorithm only considering communication, and then we extend it to consider both communication and load.

3.1 Description of the EagerMap Algorithm

The algorithm requires two previously initialized variables: *nLevels* and *execElInLevel*. *nLevels* is the number of shared levels of the architecture hierarchy plus two. This addition is required to create a level to represent application tasks (level 0) and another level to represent the processing units (level 1).

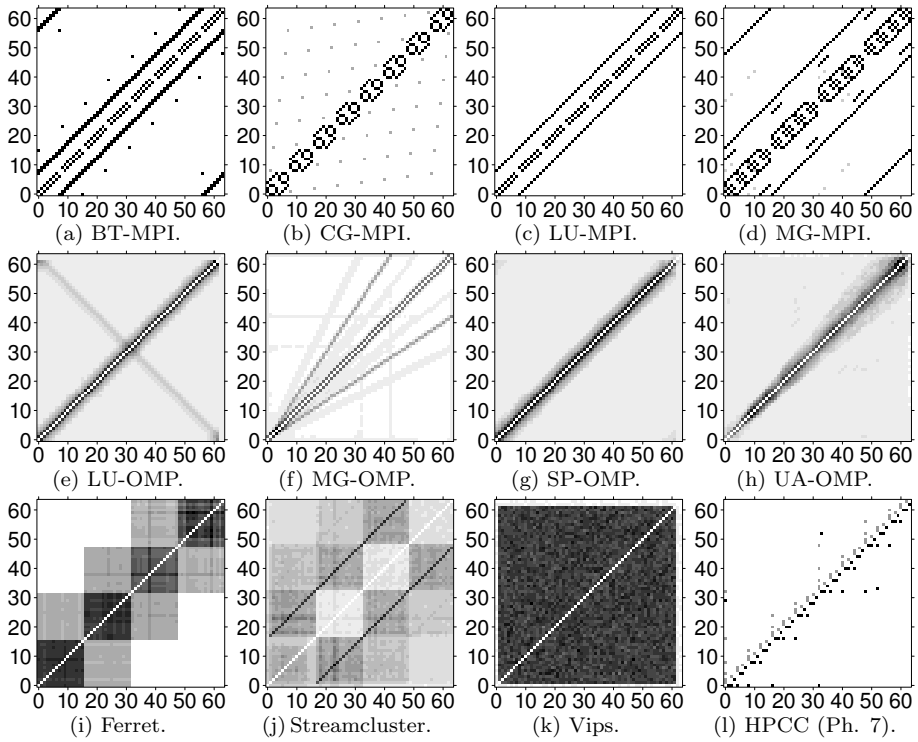


Figure 1: Example communication matrices for parallel applications consisting of 64 tasks. Axes represent task IDs. Cells show the amount of communication between tasks. Darker cells indicate more communication. The communication matrices were generated as explained in Section 4.2.

Algorithm 1: *MapAlgorithm*: The top level algorithm of EagerMap.

```
   Input: commMatrixInit[][], nTasks
   Output: map[]
   LocalData: nElements, i, nGroups, rootGroup, commMatrix[][], groups[],
              previousGroups[]
   GlobalData: nLevels, execElInLevel[], hardwareTopologyRoot
 1 begin
 2     for i←0 ; i<nTasks ; i←i+1 do
 3         groups[i].id ← i;
 4         groups[i].nElements ← 0;
 5     end
 6     nElements ← nTasks;
 7     commMatrix ← commMatrixInit;
 8     for i←1 ; i<nLevels ; i←i+1 do
 9         previousGroups ← groups;
           /* GenerateGroupsForLevel is implemented in Algorithm 2.        */
10         [nGroups, groups] ← GenerateGroupsForLevel(commMatrix, nElements, i,
             previousGroups, execElInLevel[i]);
11         if i < nLevels-1 then
               /* RecreateMatrix is implemented in Algorithm 4.            */
12             commMatrix ← RecreateMatrix(commMatrix, groups, nGroups);
13         end
14         nElements ← nGroups;
15     end
16     rootGroup.nElements ← nElements;
17     for i←1 ; i<nElements ; i←i+1 do
18         rootGroup.elements[i] ← groups[i];
19     end
       /* MapGroupsToTopology is implemented in Algorithm 5.               */
20     MapGroupsToTopology(archTopologyRoot, rootGroup, map);
21     return map;
22 end
```

$execElInLevel$ is a vector with $nLevels$ positions. $execElInLevel[0]$ is not used. $execElInLevel[1]$ contains the number of processing units. For positions i, such that $1 < i < nLevels$, the value is the number of hardware objects on the respective architecture hierarchy level. Hardware objects are cores, caches, processors and NUMA nodes, among others. For instance, $execElInLevel[2]$ can be the number of cores, $execElInLevel[3]$ the number of last level caches, and $execElInLevel[nLevels-1]$ the number of NUMA nodes. Since private levels of the architecture hierarchy are not important for our mapping strategy, we only consider the shared levels when preparing both $nLevels$ and $execElInLevel$.

3.1.1 Top Level Algorithm

The top level mapping algorithm is shown in Algorithm 1. It calculates the mapping for each level on the architecture hierarchy. The *groups* variable represents the groups of elements for the level being processed. The *previousGroups* variable represents the groups of elements of the previous level. First, it initializes *groups* with the application tasks (loop in line 2). Afterwards, it iterates over all levels on the architecture hierarchy, in line 8. After generating the groups of tasks for a level (line 10) (explained in Section 3.1.2), the algorithm generates a new communication matrix (line 12, discussed in Section 3.1.3). This step is necessary since we consider each group of tasks as the base element for mapping

Algorithm 2: *GenerateGroupsForLevel*: Generates the groups for a level of the architecture hierarchy.

```
   Input: commMatrix[][], nElements, level, previousGroups[], avlGroups
   Output: nGroups, groups[]
   LocalData: chosen[], elPerGroup, leftover, gi, inGroup, i, newGroup
 1 begin
 2     nGroups ← min(nElements, avlGroups);
 3     elPerGroup ← nElements / nGroups;
 4     leftover ← nElements % nGroups;
 5     for i←0 ; i<nElements ; i←i+1 do
 6         chosen[i] ← 0;
 7     end
 8     gi ← 0;
 9     for i←0 ; i<nElements ; i←i+inGroup do
10         inGroup ← elPerGroup;
11         if leftover > 0 then
12             inGroup ← inGroup + 1;
13             leftover ← leftover - 1;
14         end
           /* GenerateGroup is implemented in Algorithm 3.                  */
15         newGroup ← GenerateGroup(commMatrix, nElements, inGroup, chosen,
             previousGroups);
16         newGroup.nElements ← inGroup;
17         newGroup.id ← gi;
18         groups[gi] ← newGroup;
19         gi ← gi + 1;
20     end
21     return [nGroups, groups];
22 end
```

on the next hierarchy level.

The *groups* variable implicitly generates a tree of groups. Level 0 represents the tasks. Level 1 represents groups of tasks. Level 2 represents groups of groups of tasks. In other words, on each level a new application graph is generated by coarsening the previous level. After the loop in line 8 finishes, the *groups* variable represents the hierarchy level $nLevels - 1$ and contains $nElements$ elements. We set up *rootGroup* to point to these elements of the highest level (for loop in line 17). Finally, the algorithm maps the tree that represents the tasks, *rootGroup*, to the tree that represents the architecture topology, *archTopologyRoot*. This procedure is explained in Section 3.1.4.

3.1.2 Generating the Groups for a Level of the Architecture Hierarchy

The *GenerateGroupsForLevel* algorithm, described in Algorithm 2, handles the creation of all groups for a given level of the architecture hierarchy. It expects that the levels of hierarchy up to the previous processed level to be already grouped in *previousGroups*. The maximum number of groups is the number of hardware objects of that level. The selection of which elements belong to each group is performed by *GenerateGroup*.

The *GenerateGroup* algorithm, in Algorithm 3, groups elements that present a large amount of communication among themselves. For the grouping, the algorithm adopts a greedy strategy as follows. Each iteration of the loop in line 2 adds one element to the group. The added element, expressed by the

Algorithm 3: *GenerateGroup*: Generates one group of elements that communicate.

```
   Input: commMatrix[][], totalElements, groupElements, chosen[], previousGroups[]
   Output: group
   LocalData: i, j, k, w, wMax, winners[], winner
 1 begin
 2     for i←0 ; i<groupElements ; i←i+1 do
 3         wMax ← -1;
 4         for j←0 ; j<totalElements ; j←j+1 do
 5             if chosen[j]=0 then
 6                 w ← 0;
 7                 for k←0 ; k<i ; k←k+1 do
 8                     w ← w + commMatrix[j][winners[k]];
 9                 end
10                 if w > wMax then
11                     wMax ← w;
12                     winner ← j;
13                 end
14             end
15         end
16         chosen[winner] ← 1;
17         winners[i] ← winner;
18         group.elements[i] ← previousGroups[winner];
19     end
20     return group
21 end
```

winner variable, is the one that presents the largest amount of communication relative to the elements already in the group. The *chosen* variable is used to avoid selecting the same element more than once. *GenerateGroup* can be parallelized in the loop of line 4, where each thread would compute its local *winner* in parallel, as we will explain in Section 3.5.

3.1.3 Computing the Communication Matrix for the next Level of the Architecture Hierarchy

RecreateMatrix, described in Algorithm 4, regenerates the communication matrix to be used for the next level of the architecture hierarchy. The new communication matrix has an order of *nGroups*. It contains the amount of communication between the groups. It is calculated by summing up the amount of communication between the elements of different groups.

3.1.4 Mapping the Group Tree to the Architecture Topology Tree

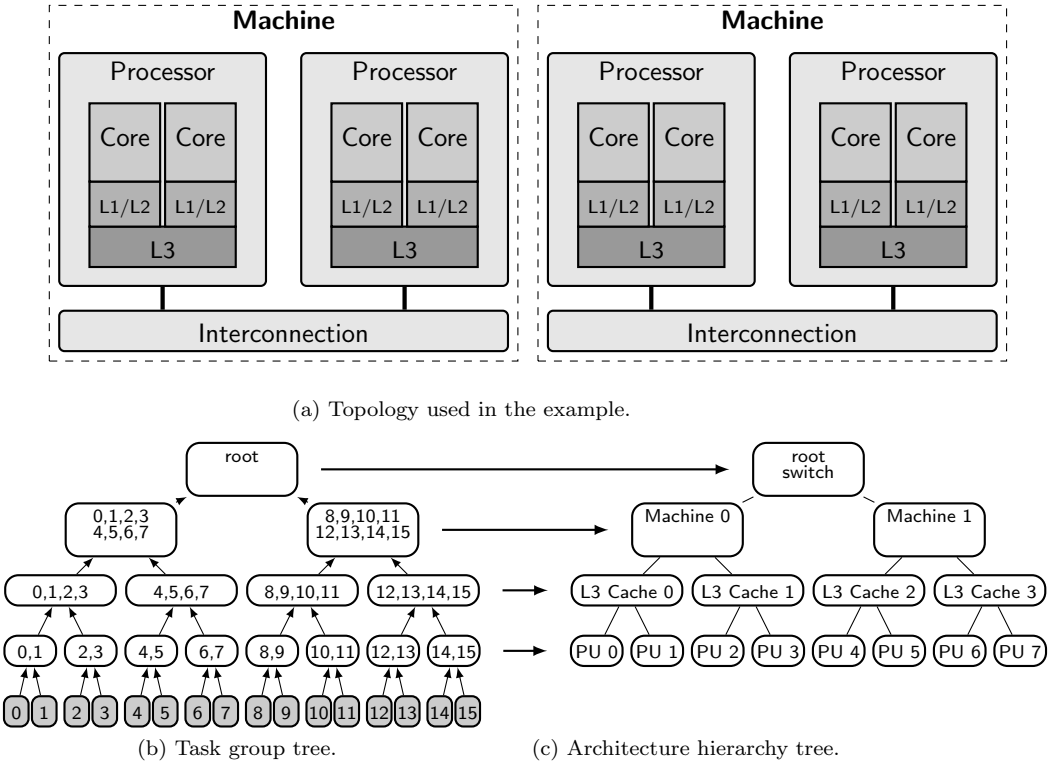
The algorithm to map the group tree to the architecture topology tree is *MapGroupsToTopology*, detailed in Algorithm 5. It performs a recursion over the levels of the architecture hierarchy. The recursion stop condition happens when it reaches the lowest level of the architecture topology, the processing unit (PU) (line 2). If the stop condition is not fulfilled, the algorithm already knows that the maximum number of groups per level never exceeds the number of hardware objects of that level, as explained in *GenerateGroupsForLevel*. Therefore, if the level of the architecture hierarchy in the recursion is shared (line 6), the algorithm only assigns one hardware object of the following level

Algorithm 4: *RecreateMatrix*: Calculates the communication matrix for the next level.

```
   Input: commMatrix, groups[], nGroups
   Output: newCommMatrix[][]
   LocalData: i, j, k, z, w
 1 begin
 2     for i←0 ; i<nGroups-1 ; i←i+1 do
 3         for j←i+1 ; j<nGroups ; j←j+1 do
 4             w ← 0;
 5             for k←0 ; k<groups[i].nElements; k←k+1 do
 6                 for z←0 ; z<groups[j].nElements; z←z+1 do
 7                     w ← w + commMatrix[ groups[i].elements[k].id ][
                         groups[j].elements[z].id ];
 8                 end
 9             end
10             newCommMatrix[i][j] ← w;
11             newCommMatrix[j][i] ← w;
12         end
13     end
14     return newCommMatrix;
15 end
```

Algorithm 5: *MapGroupsToTopology*: Maps the group tree to the hardware topology tree.

```
   Input: hardwareObj, group, map[]
   LocalData: i
 1 begin
 2     if hardwareObj.type = ProcessingUnit then
 3         for i←0 ; i<group.nElements ; i←i+1 do
 4             map[ group.elements[i].id ] ← hardwareObj.id;
 5         end
 6     else if hardwareObj.nSharers > 1 then
 7         for i←0; i<group.nElements; i←i+1 do
 8             MapGroupsToTopology(hardwareObj.linked[i], group.elements[i], map);
 9         end
10     else
11         MapGroupsToTopology(hardwareObj.linked[0], group, map);
12     end
13 end
```



(a) Topology used in the example.

(b) Task group tree.

(c) Architecture hierarchy tree.

Figure 2: Mapping 16 tasks in an architecture with 8 PUs, L3 caches shared by 2 PUs, 2 processors per machine, and 2 machines.

to each group and recursively calls itself for each element of the following level. Otherwise, the level is private and is not considered for mapping (line 10).

3.2 Mapping Example in a Multi-level Hierarchy

For a better understanding of how our mapping algorithm works, we present an example of mapping 16 tasks in a cluster of 2 machines, each consisting of 8 cores, with L3 caches shared by 2 cores and 2 processors per machine. Fig. 2a illustrates the hierarchy of the machine, which needs to be transformed in the topology tree shown in Fig. 2c. The topology tree can be generated either automatically, using tools such as hwloc [11], or manually by the programmer. Private levels, with arity 1, such as L1 and L2 caches, as well as the processor, are not relevant. The global variable *nLevels* has the value of 4, since there are 4 levels in the hierarchy. Therefore, *execElInLevel* has 4 positions. Position 0 represents the tasks and is not used. Position 1 represents the PUs (the cores in this case) and its value is 8. Position 2 represents the L3 caches and its value is 4. Finally, position 3 represents the machines and its value is 2.

Regarding Algorithm 1 (*MapAlgorithm*), the loop in line 8 is executed 3 times. In the first iteration, it generates the groups of tasks that execute in each core. Since the number of tasks is 16 and the number of cores is 8, it generates 8 groups of 2 tasks each. The communication matrix used in the first

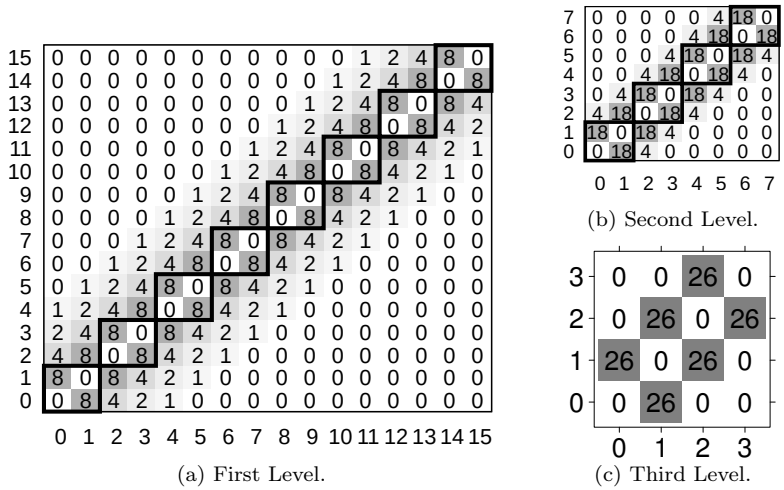


Figure 3: Communication matrices used in our example. A box is drawn around the groups of tasks generated by the algorithm.

i	winners		winner	wmax	w_0	w_1	w_2	w_3	w_4	w_{5-15}
0			0	0	0	0	0	0	0	0
1	0		1	8	-	8	4	2	1	0
2	0	1	-	-	-	-	-	-	-	-

Figure 4: Example of the *GenerateGroup* algorithm, when generating the first group in our example.

iteration is shown in Fig. 3a. We demonstrate how the first group is generated in Fig. 4 (*GenerateGroup*, Algorithm 3). When i is 0, the *winners* vector is empty and task 0 is selected to be the first one of the group. When i is 1, the task that presents the most communication to the tasks in the *winners* vector is task 1. Therefore, task 1 is selected as *winner* and enters the group. Algorithm *GenerateGroup* then returns a group formed by tasks [0, 1], implicitly stored in the *group.elements* vector.

GenerateGroupsForLevel (Algorithm 2) repeats the same procedure until all groups of the level are formed. Then, the communication matrix for the next level, in Fig. 3b, is calculated by *RecreateMatrix* (Algorithm 4). The second iteration of the loop in line 8 of *MapAlgorithm* behaves similarly, but selects which groups of tasks from the previous iteration share each L3 cache. Since the number of groups was 8 and there are 4 L3 caches, it generates 4 new groups with 2 elements from the previous level in each.

In the third iteration of the loop in line 8 of *MapAlgorithm*, it selects which groups of groups of tasks share each machine. The communication matrix illustrated in Fig. 3c is used. Since the number of groups generated in the previous iteration was 4 and there are 2 machines, it generates 2 new groups of 2 elements each. To complete the group tree, the *rootGroup* variable points to these 2 groups. Afterwards, in line 19 of *MapAlgorithm*, the task group tree shown in Fig. 2b is finished.

MapGroupsToTopology (Algorithm 5) maps the task group vertices to ver-

tices of the architecture hierarchy. The mapping begins from the root node. Tasks $[0-7]$ are mapped to Machine 0. Tasks $[0-3]$ are mapped to L3 cache 0. Tasks $[0-1]$ are mapped to PU 0 (core 0). The recursion continues until all groups of tasks are mapped to a PU (core). In this example, tasks x and $x+1$, where x is even, will be mapped to core $x/2$.

3.3 Complexity of EagerMap

For the analysis of the complexity of EagerMap, we first introduce the variables used in the equations. E is the number of elements to be mapped in the current level of the architecture hierarchy (tasks or groups of tasks), which is different for each level. G is the number of elements per group. P is the number of processing units. N is the number of tasks to be mapped. L is the number of levels on the architecture hierarchy. To make it easier to calculate the complexity, we consider that $P \leq N$.

The complexity of *GenerateGroup* is:

$$\sum_{i=1}^{G}\sum_{j=1}^{E} i = \sum_{i=1}^{G} E \cdot i \leq E \cdot G^2 \tag{1}$$

The complexity of *GenerateGroupsForLevel* is shown in Equation 2. The number of groups is $\frac{E}{G}$. Also, E is an upper bound for G.

$$\sum_{i=1}^{E/G} GenerateGroup \leq \sum_{i=1}^{E/G} E \cdot G^2 \leq \frac{E}{G} \cdot E \cdot G^2 \leq E^3 \tag{2}$$

The complexity of *RecreateMatrix* is $O(E^2)$. The complexity of *MapGroupsToTopology* is the same as performing a depth-first search, $O(V + C)$, where V is the number of vertices and C is the number of edges. Since our *groups* variable implicitly forms a tree, we know that $C = V - 1$. The last level of this tree has N vertices, and the penultimate level has P vertices. The number of vertices of the other levels is the number of the previous level, divided by the number of sharers. The number of sharers is greater than 1 since we only keep track of shared levels. Therefore, $V \leq N + P + 2 \cdot P = O(N)$. With this, the complexity of *MapGroupsToTopology* is $O(N)$.

The complexity of the top level algorithm, *MapAlgorithm*, depends on all previous algorithms, as shown in Equation 3. To calculate the complexity, we have to take into account that the value of E changes on each level of the architecture hierarchy.

$$\sum_{i=0}^{L} \Big(GenerateGroupsForLevel + RecreateMatrix \Big) + MapGroupsToTopology \tag{3}$$

$$= \sum_{i=0}^{L} \Big(O\left(E_i^3\right) + O\left(E_i^2\right) \Big) + O(N) \tag{4}$$

We can rewrite Equation 4 as Equation 5 by considering that the algorithm iterates $L+1$ times (L levels of the architecture topology plus one level for the

Algorithm 6: *LbGenerateGroupsForLevel*: Generates the groups for a level of the architecture hierarchy.

```
   Input: commMatrix[][], nElements, level, previousGroups[], avlGroups
   Output: nGroups, groups[]
   LocalData: chosen[], gi, i, newGroup, totalLoad
 1 begin
 2     totalLoad ← 0;
 3     for i←0 ; i<nElements ; i←i+1 do
 4         chosen[i] ← 0;
 5         totalLoad ← totalLoad + previousGroups[i].load;
 6     end
 7     gi ← 0;
 8     for i←0 ; i<nElements ; i←i+newGroup.nElements do
 9         loadThreshold ← totalLoad / avlGroups;
           /* LbGenerateGroup is implemented in Algorithm 7.          */
10         newGroup ← LbGenerateGroup(commMatrix, nElements, chosen,
             previousGroups, loadThreshold, i);
11         newGroup.id ← gi;
12         totalLoad ← totalLoad - newGroup.load;
13         avlGroups ← avlGroups - 1;
14         groups[gi] ← newGroup;
15         gi ← gi + 1;
16     end
17     return [nGroups, groups];
18 end
```

tasks) and that only shared topology levels are represented, which means that, in the worst case scenario, the topology will be a binary tree with P leaves and a maximum number of vertices equal to $2P - 1$.

$$\leq \sum_{i=1}^{L} \left[O\left(\left(\frac{P}{2^{i-1}} \right)^3 \right) + O\left(\left(\frac{P}{2^{i-1}} \right)^2 \right) \right] + \big(O(N^3) + O(N^2) \big) + O(N) \quad (5)$$

$$\begin{aligned} &\leq 2\big(O(P^3) + O(P^2)\big) + \big(O(N^3) + O(N^2)\big) + O(N) \\ &= 3\big(O(N^3) + O(N^2)\big) + O(N) = O\left(N^3\right) \end{aligned} \quad (6)$$

With the simplifications shown in Equation 6, we find that EagerMap has a polynomial complexity of $O(N^3)$.

3.4 Considering the Task Load

In a parallel application, it is very common to have its tasks use the CPU for different amounts of time and thereby have different computational power requirements. We call the CPU power a task requires to complete its execution the *task load*. In order to consider the task load when mapping tasks, and thereby balancing the load, the algorithms presented in the previous sections need slight modifications. Most modifications are done in the algorithms *GenerateGroupsForLevel* (Algorithm 2) and *GenerateGroup* (Algorithm 3). In these algorithms, the total number of elements is related to the number of

Algorithm 7: *LbGenerateGroup*: Generates one group of elements that communicate.

```
   Input: commMatrix[][], totalElements, chosen[], previousGroups[], loadThreshold,
          done
   Output: group
   LocalData: i, j, w, wMax, winners[], winner
 1 begin
 2     group.load ← 0;
 3     group.nElements ← 0;
 4     for i←0 ; done<totalElements AND group.load<loadThreshold; i←i+1 do
 5         wMax ← -1;
 6         for j←0 ; j<totalElements ; j←j+1 do
 7             if chosen[j]=0 then
 8                 w ← 0;
 9                 for k←0 ; k<i ; k←k+1 do
10                     w ← w + commMatrix[j][winners[k]];
11                 end
12                 if w > wMax then
13                     wMax ← w;
14                     winner ← j;
15                 end
16             end
17         end
18         chosen[winner] ← 1;
19         winners[i] ← winner;
20         group.elements[i] ← previousGroups[winner];
21         done ← done + 1;
22         group.nElements ← group.nElements + 1;
23         group.load ← group.load + previousGroups[winner].load;
24     end
25     return group
26 end
```

tasks, and they try to keep the number of tasks per group as close as possible to $nElements/nGroups$. To consider the task load, we modify the concept of an element to represent the task load. In this way, instead of the task, the algorithm uses the task load. And instead of the total number of elements, the algorithm uses the total task load. The idea is to group tasks that communicate in the same groups, but limit the group sizes by the load, not number of tasks.

The modified algorithms to consider the load can be found in Algorithms 6 (*LbGenerateGroupsForLevel*) and 7 (*LbGenerateGroup*). The key idea is that the maximum load of a group (*loadThreshold*) is recalculated on every iteration of the loop (line 9 in Algorithm 6). We do this to get a more even balance of the load for the next groups, since the actual load of the group rarely is exactly $totalLoad/avlGroups$ due to the different loads between the tasks. If this is not done, the loads of the last groups are usually very low. Regarding Algorithm 7, the main difference is that the group size is limited by the average load *loadThreshold*, in line 4, while in the non load balancing version of the algorithm the group size was limited to the average number of tasks per group.

3.5 Parallel EagerMap

The function *GenerateGroup* is the only function in the critical path of EagerMap that can be parallelized without changing the serial version behavior.

Algorithm 8: *ParallelGenerateGroup*: Generates one group of elements that communicate.

```
   Input: commMatrix[][], totalElements, groupElements, chosen[], previousGroups[]
   Output: group
   LocalData: i, j, wMax, nt, winners[], winner, threadResult[]
 1 begin
 2     for i←0 ; i<groupElements ; i←i+1 do
 3         wMax ← -1;
 4         #pragma omp parallel
               ThreadData: id, j, w, k
 5             id ← getThreadID();
 6             #pragma omp master
 7             nt ← getNumberOfThreads();
 8             threadResult[id].wMax ← -1;
 9             #pragma omp for
10             for j←0 ; j<totalElements ; j←j+1 do
11                 if chosen[j]=0 then
12                     w ← 0;
13                     for k←0 ; k<i ; k←k+1 do
14                         w ← w + commMatrix[j][winners[k]];
15                     end
16                     if w > threadResult[id].wMax then
17                         threadResult[id].wMax ← w;
18                         threadResult[id].winner ← j;
19                     end
20                 end
21             end
22         end
23         wMax ← -1;
24         for j←0 ; j<nt ; j←j+1 do
25             if threadResult[j].wMax>wMax then
26                 wMax ← threadResult[j].wMax;
27                 winner ← threadResult[j].winner;
28             end
29         end
30         chosen[winner] ← 1;
31         winners[i] ← winner;
32         group.elements[i] ← previousGroups[winner];
33     end
34     return group
35 end
```

RecreateMatrix and *MapGroupsToTopology* could also be parallelized. However, the overhead of parallelizing them surpasses the benefits.

The parallel version of EagerMap consists of parallelizing the loop of line 4 of Algorithm 3. This loop is responsible for finding the next task that will belong to a group. The idea of the parallel algorithm is that we can search for the next task in parallel, where each thread evaluates a subset of the total tasks and finds its local task that maximizes the group communication.

The parallel algorithm is shown in Algorithm 8. We express the parallelization in a notation similar to the OpenMP standard. In the beginning of the parallel block, in line 4, we initialize the vector *threadResult* containing the local task that maximizes the group communication. The local task is searched in parallel in the for loop in line 10. After that, the task that maximizes communication is selected from the local task found by each thread, in line 24. Although there are no critical regions, we do not expect a linear speedup because the parallel region has a short duration. The speedup should be higher with a larger number of tasks.

The function calls to *GenerateGroup* related to the first hierarchy level are the most expensive procedures, since they are processing the grouping of tasks. Thereby, for the first hierarchy level, we use the parallel version. For the subsequent levels, the cost of *GenerateGroup* falls dramatically, such that is better to use the sequential version.

3.6 Running EagerMap in Non-Tree Topologies

The previous algorithms presented in this section work only in symmetric tree topologies. This makes them applicable only to shared memory machines, or clusters where all machines have the same topology. In this section, we explain how to extend EagerMap to work on any kind of cluster or grid. The idea is to make a pre-computation of which tasks will run on each machine of the cluster, and then call the previously presented algorithms for each machine separately. We consider that the communication within each machine is much faster than the communication between different machines.

The algorithms that perform this pre-computation are very similar to the others and are also based on a greedy grouping strategy. The first thing the algorithm does is to calculate the number of tasks that each machine will execute. For that, it counts the total number of available PUs in all machines, and spreads the tasks in such a way that every machine will have the closest possible number of tasks per PU. For instance, if the number of tasks is equal to the total number of PUs, each machine will have 1 task per PU. After that, it generates the group of tasks that each machine will run in a greedy way similarly to Algorithm 3, where the next task to be added to a group is the one that maximizes the communication to the elements already within the group.

The final step, as already explained, is to call EagerMap separately for each group. In this way, we can handle any kind of cluster/grid regardless of the topology or if the machines have different configurations. The load balance can be taken into account in this step in a very similar way to what is explained in Section 3.4. The division of the tasks that will run on each group can be parallelized as explained in Section 3.5.

4 Evaluation Methodology of EagerMap

In this section, we show how we evaluate our proposal. We discuss the benchmarks and architecture used in our evaluation, how we obtain the communication matrices, and other mapping strategies used for comparison.

4.1 Benchmarks

For the evaluation of the mapping algorithms, we use applications from the MPI implementation of the NAS parallel benchmarks (NPB) [3], the OpenMP NPB implementation [28], the High Performance Computing Challenge (HPCC) benchmark [36] and the PARSEC benchmark suite [7]. The NAS benchmarks were executed with input size *B*, HPCC with an input matrix with 4000^2 elements and PARSEC was executed with the *native* input size.

Most experiments are based on applications with a static communication behavior. HPCC, which has several communicative phases, was used to show that efficient online algorithms, such as EagerMap, are important for applications with a dynamic behavior. All applications were configured to create a number of threads equivalent to the number of PUs of the machine.

4.2 Generating the Communication Matrices

For the MPI-based benchmarks, we used the eztrace framework [41] to trace all MPI messages sent by tasks and built a communication matrix based on the number of messages sent between tasks. We also generated the communication matrices using the number of bytes and, although the value of each individual cell was different, the overall pattern was the same. For the HPCC benchmark, which has a different communication pattern for each step, we also use eztrace to generate the matrices statically, since the communication pattern does not change between executions. For the benchmarks that use shared memory for implicit communication using memory accesses, we built a memory tracer based on the Pin binary analysis tool [34], similarly to [6]. This tool traces all memory accesses from the tasks. We build a communication matrix by comparing memory accesses to the same memory address from different tasks and increment the matrix on every match.

4.3 Generating the Task Load

To measure the task load, we count the amount of instructions executed by each thread, which better represents the load than the traditional CPU time [17]. Therefore, the load between the tasks are imbalanced when the amount of instructions executed by each thread is different.

4.4 Hardware Architecture

The hardware architecture used in our experiments regarding shared memory consists of 4 Intel Xeon X7550 processors for a total of 64 PUs. In this architecture, there are three possibilities for communication between tasks. Tasks running on the same core can communicate through the fast L1 or L2 caches and have the highest communication performance. Tasks that run on different cores

have to communicate through the slower L3 cache, but can still benefit from the fast intra-chip interconnection. When tasks communicate across physical processors, they need to use the slow inter-chip interconnection.

For the cluster experiments, we used OpenMPI 1.6.5 and two types of machines: (A) 2 Intel Xeon E5530, with a total of 16 PUs, and (B) 2 Intel Xeon CPU E5-2640v2, with a total of 32 PUs. We used only half of the PUs of the machines because it presented better performance than using all PUs due to SMT and poor support from OpenMPI. We evaluated 6 different clusters with these machines:

4×8	4 machines of type A, 32 tasks in total
2×16	2 machines of type B, 32 tasks in total
4×16	4 machines of type B, 64 tasks in total
8×16	8 machines of type B, 128 tasks in total
Mixed-32	2 machines of type A and 1 of type B, 32 tasks in total
Mixed-64	4 machines of type A and 2 of type B, 64 tasks in total

To evaluate the load-aware EagerMap, we used a Xeon Phi Coprocessor model 3120P, with the Knights Corner architecture. We used this architecture because, due to the large number of cores (57 cores, each 4-way SMT), load balancing plays a key role in its performance.

4.5 Comparison

We compare EagerMap to (i) a Random mapping, which is an average result of 30 different random mappings; (ii) TreeMatch [25, 26]; (iii) Scotch [38]; and (iv) MPIPP [12]. Most performance results are normalized to the performance of the default operating system mapping, the Linux Completely Fair Scheduler [29], which focuses on keeping the load balanced between the tasks and cores.

5 Results

We evaluate the performance and quality of our algorithm, as well as the performance improvements obtained when mapping the tasks of the applications.

5.1 Performance of the Mapping Algorithms

Fig. 5 shows the execution time of the four mapping algorithms for all benchmarks. For 128 tasks, EagerMap is about 10 times faster than Scotch, 1000 times faster than TreeMatch, and 100,000 times faster than MPIPP. TreeMatch has an exponential complexity, so a much higher execution time is expected for it. Due to this, the difference in execution time between TreeMatch and EagerMap increases together with the number of tasks. MPIPP is much slower than the other mechanisms because it needs to perform several iterations to refine its initial random mapping. It did not finish executing when the number of tasks was higher than 128.

5.2 Quality of the Mapping

The quality of the calculated mapping determines the benefits that can be achieved. Quality is measured by the amount of locality achieved. We use

Equation 7 to calculate the quality, which is provided in the source code of TreeMatch. In this equation, n is the number of tasks, $M[i][j]$ is the amount of communication between tasks i and j, $map[x]$ is the processing unit (PU) mapped, and $lat[a][b]$ is the latency of the PUs in the hierarchy. We calculated the latencies using LMbench [37].

$$MappingQuality = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{M[i][j]}{lat\big[map[i]\big]\big[map[j]\big]} \tag{7}$$

Fig. 6 presents the mapping quality results for the communication matrices previously illustrated in Fig. 1. Applications with more structured communication patterns presented the highest improvements, as expected. CG-MPI, LU-MPI and HPCC (Phase 7) presented the best improvements because their communications can be easily optimized by mapping neighboring tasks together.

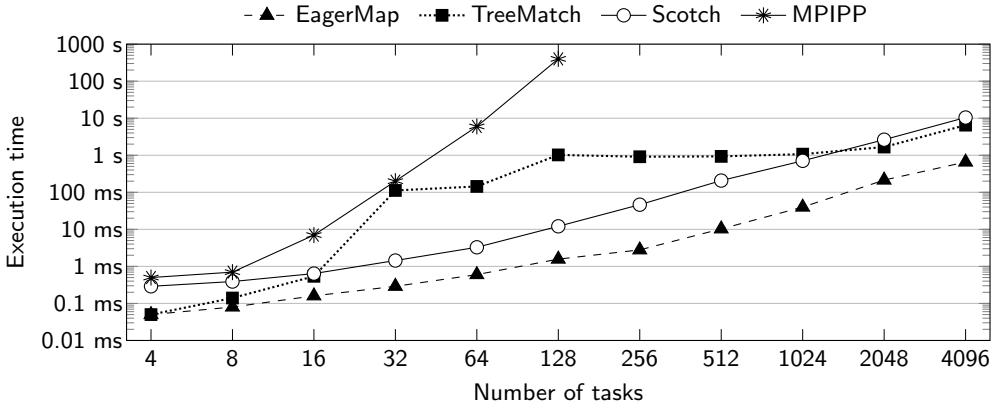


Figure 5: Execution time (in ms) of the mapping algorithms, for different numbers of tasks to be mapped.

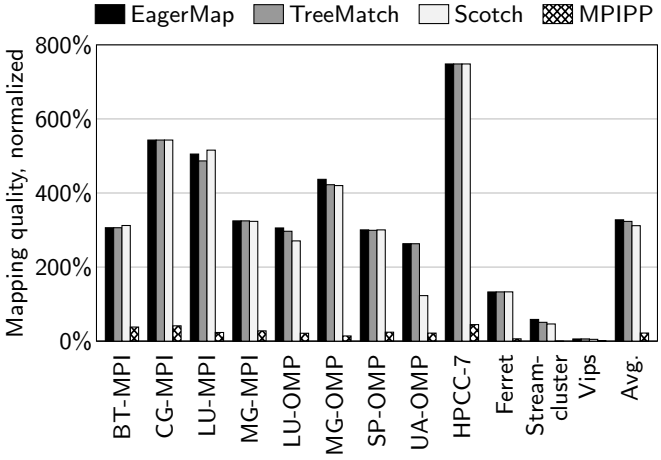


Figure 6: Comparison of the mapping quality, normalized to the random mapping. Higher values are better.

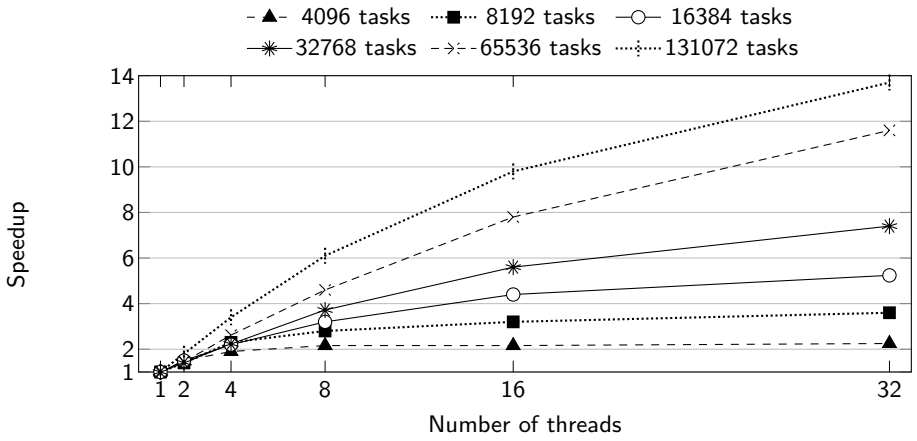


Figure 7: Speedup of the parallel version of EagerMap with different number of tasks to be mapped, varying the number of execution threads.

In BT-MPI and MG-MPI, near and distant tasks communicate, presenting more challenges for the mapping algorithm. The reason is that if neighboring tasks are mapped together, the communication between distant tasks does not improve. Likewise, mapping distant tasks together does not improve communication between neighboring tasks. In applications with less structured patterns, such as Streamcluster, a lower improvement is expected because the ratio of communication between tasks that communicate more and tasks that communicate less is lower. Vips is the only application with unstructured communication, such that no task mapping was able to improve its communication.

The quality obtained with MPIPP is similar to the random mapping, since MPIPP is based on refining an initial random mapping. Although MPIPP can work with few tasks, when the number of tasks increases, the possibilities of permutation increase exponentially. This makes it more difficult to find new combinations to improve the initial solution. EagerMap, TreeMatch and Scotch presented similar results for all applications. This result demonstrates that EagerMap is able to achieve results as good as more complex algorithms due to the characteristics of the communication patterns we observed in Section 3.

5.3 Speedup of the Parallel Version of EagerMap

The speedup of the parallel version of EagerMap when mapping different numbers of tasks can be found in Fig. 7. We analyze the speedup relative to the number of tasks because it is the parameter with most influence in execution time. The task tree structure has little impact on the speedup, since the parallel algorithm is applied only to the first levels of the hierarchy (usually only to the first level), as explained in Section 3.5. As we explain in Section 3.5, we do not expect a linear speedup because the duration of the parallel phase is short. The parallel phase is restarted several times during the mapping calculation, imposing overhead in the parallelization. On the other hand, the impact of the overhead is decreased when the number of tasks to be mapped increases, as can be seen in the figure. This happens because, with more tasks, the duration of

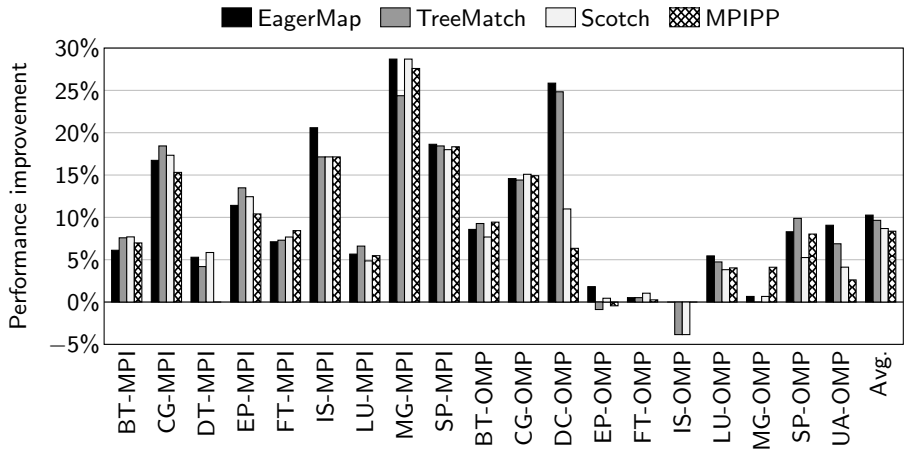


Figure 8: Performance improvement compared to the operating system mapping of the NAS-MPI and NAS-OMP benchmarks in a shared memory environment.

the parallel phase increases. With 131072 tasks, the speedup using 32 threads was 13.7, and should be higher for more tasks.

It is important to note that, when the number of tasks is large, the usage of the parallel version of EagerMap is encouraged. This happens because the execution time grows fast with the increase of the number of tasks. For instance, with 32768 tasks, the single threaded execution required 43 minutes. With 131072 tasks, the single threaded execution time grew to 25 hours. Using the parallel version with 32 threads, the execution time was reduced to 5.8 minutes and 1.8 hours, respectively.

5.4 Performance Improvements in a Shared-Memory Environment

We executed applications using the mapping obtained with the algorithms in the machine with Intel Xeon X7550 processors. The execution time results for the MPI and OpenMP NAS Benchmarks are shown in Fig. 8. For these applications, since their communication pattern is stable, we calculated the mapping statically. As a case study for online mapping, we used the HPCC Benchmark, shown in Fig. 9, since it contains 16 phases with different communication behaviors. For each phase, we call the mapping algorithm and migrate tasks accordingly. Execution time results correspond to the average of 30 executions, and are normalized to the execution time of the original policy of the operating system. The confidence interval is smaller than 1% for all algorithms.

In general, applications with more structured communication patterns present higher performance improvements. For instance, CG-MPI's performance was significantly improved when compared to the OS mapping, since its communication pattern shows high potential for mapping, as discussed in Section 5.2. On average, EagerMap improved performance by 10.3%, while TreeMatch, Scotch and MPIPP showed improvements of 9.6%, 8.7% and 8.4%, respectively. The results of the HPCC Benchmark (Fig. 9) show a larger improvement by EagerMap than related work. Also, the lower overhead of our

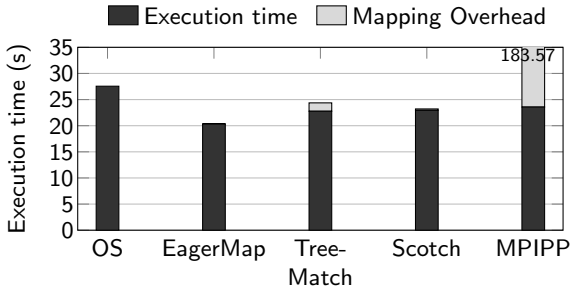


Figure 9: Execution time of HPCC using online mapping.

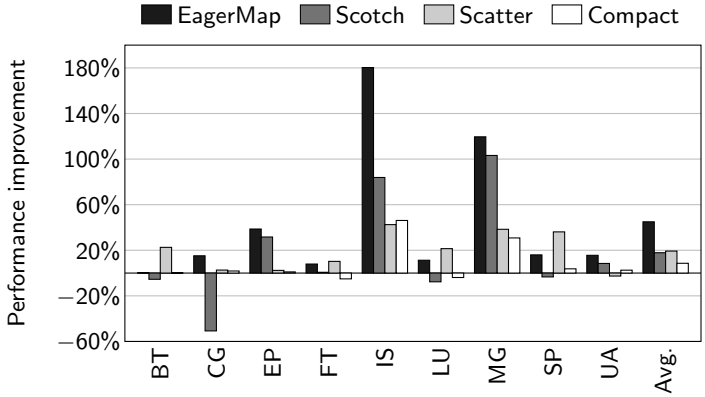
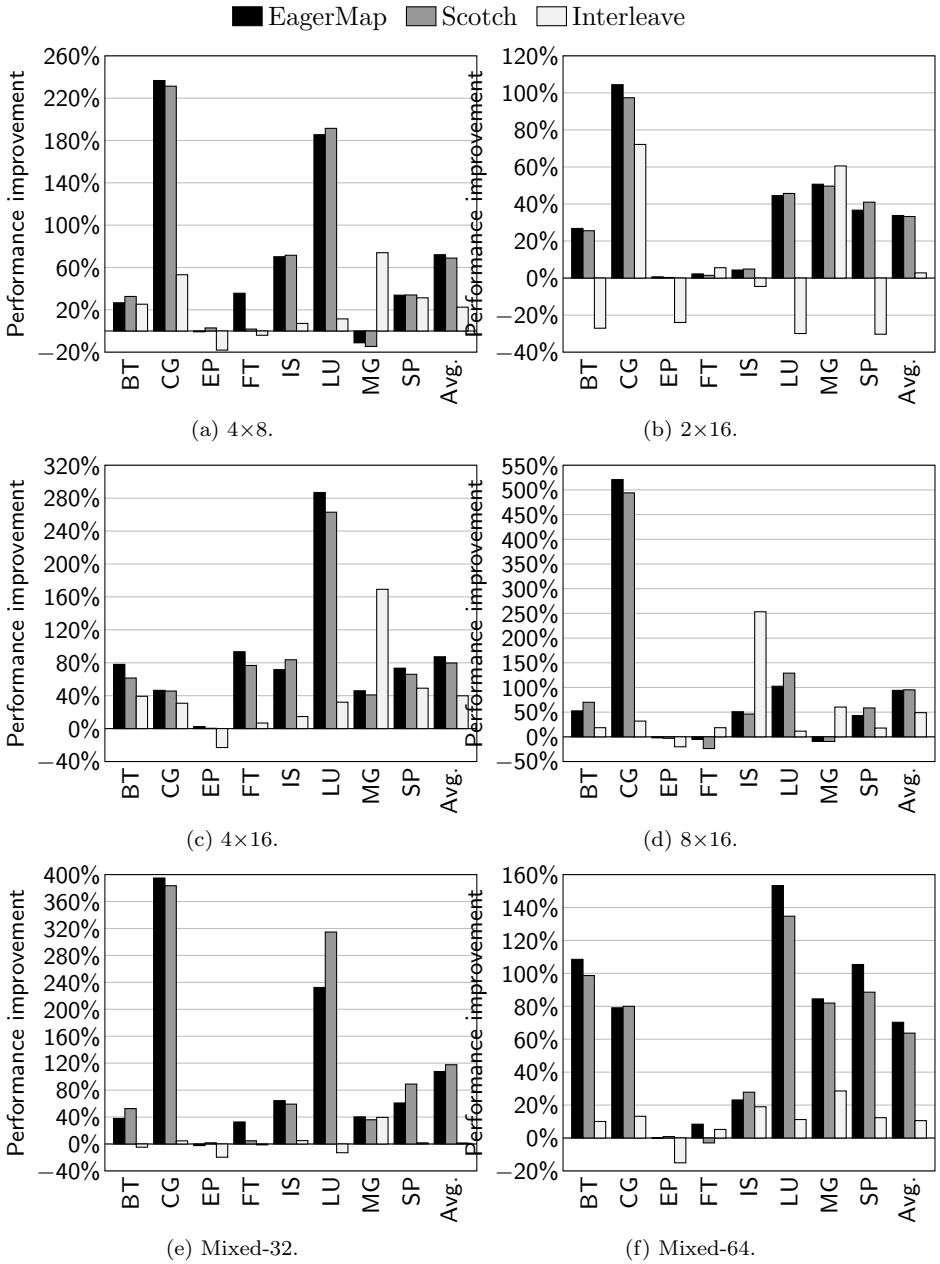


Figure 10: Performance improvements from the load-aware EagerMap in Xeon Phi for the NAS OpenMP benchmarks compared to the operating system mapping.

algorithm does not harm the application performance as much as TreeMatch and MPIPP.

5.5 Performance Improvements from the Load-Aware Algorithm

The performance improvements from the load-aware EagerMap in Xeon Phi is shown in Fig. 10. On average, EagerMap improved the performance by 44.9%, while Scotch, Scatter and Compact improved by an average of 17.8%, 19.3% and 9.6%, respectively. Scatter and Compact do not perform any analysis on the communication and load, following only pre-defined rules. This shows that EagerMap, by actually considering the communication and load aspects of the applications, can provide a better performance improvement. On the other hand, Scotch also takes into account both the load and communication to calculate the mapping. This indicates that EagerMap can make a better use of this information than Scotch.



(a) 4×8.

(b) 2×16.

(c) 4×16.

(d) 8×16.

(e) Mixed-32.

(f) Mixed-64.

Figure 11: Performance improvement compared to the operating system mapping of the NAS-MPI benchmarks in the clusters.

5.6 Performance Improvements in Clusters

The results of the cluster experiments with applications from the MPI NAS benchmarks can be found in Fig. 11. The results are normalized to the random mapping. The mapping was performed only statically because OpenMPI does not support task migration between nodes.

EagerMap achieves performance results similar to Scotch. However, as shown in Section 5.1, the time EagerMap requires to calculate the mapping is about 10 times faster than Scotch. In the experiments with 32 tasks, in the 4×8, 2×16 and Mixed-32 clusters, the application CG was the one that achieved the best results, where the performance was improved by up to 394% in Mixed-32. In 4×16, with 64 tasks, LU was the application with the best improvements, of 286%. BT and SP use less cores than the total in all clusters except 4×16 and Mixed-64 (the applications use a quadratic number of tasks), and Scotch presents better results. In 4×16 and Mixed-64, EagerMap was better. This shows that EagerMap is better at optimizing for communication, while Scotch is better at balancing the load between the machines. Regarding MG, the Interleave policy achieves good results because, coincidentally, its mapping is very close to the best mapping possible for MG.

6 Conclusions

The mapping of tasks to PUs in parallel architectures influences the communication performance and load balance. The communication performance can be improved by mapping tasks that communicate to PUs nearby in the memory hierarchy or to the same node in clusters or grids, making use of faster interconnections. The load balance can be improved by mapping the tasks in such a way that the load is evenly distributed among the PUs. The mapping algorithm selects which PU will execute each task and plays a key role in these types of mapping. When the mapping algorithm considers both metrics, more challenges arise, since these metrics can lead to contradictory decisions.

In this paper, we proposed EagerMap, that originally worked only in shared-memory architectures and only considered the communication pattern. We added support for load balancing, mapping in clusters and grids, and proposed a parallel version to accelerate its execution. In contrast to previous work, it adopts a more efficient method to select which tasks should be mapped together, based on an analysis of the communication pattern of parallel applications. We performed experiments with a large set of benchmarks with different communication characteristics. Results show that EagerMap calculated better task mappings than the state of the art, with a drastically lower overhead and better scaling, which makes EagerMap more suitable for online mapping.

As future work, we intend to study EagerMap to allow it to execute in a cluster in a distributed way.

EagerMap is licensed under the GPL and is available at `http://github.com/ehmcruz/eagermap`.

Acknowledgment

This research received funding from the EU H2020 Programme and from MCTI/RNP-Brazil under the HPC4E project, grant agreement n.º 689772. It was also supported by Intel.

References

[1] A. Anbar, O. Serres, E. Kayraklioglu, A.-H. A. Badawy, and T. El-Ghazawi. Exploiting Hierarchical Locality in Deep Parallel Architectures. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(2):1–25, 2016.

[2] R. Azimi, D. K. Tam, L. Soares, and M. Stumm. Enhancing Operating System Support for Multicore Processors by Using Hardware Performance Monitoring. *ACM SIGOPS Operating Systems Review*, 43(2):56–65, apr 2009.

[3] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. *International Journal of Supercomputer Applications*, 5(3):66–73, 1991.

[4] S. Bak, H. Menon, S. White, M. Diener, and L. Kale. Multi-level load balancing with an integrated runtime approach. In *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2018.

[5] G. Ballard, E. Carson, J. Demmel, M. Hoemmen, N. Knight, and O. Schwartz. Communication lower bounds and optimal algorithms for numerical linear algebra. *Acta Numerica*, 23(May):1–155, 2014.

[6] N. Barrow-Williams, C. Fensch, and S. Moore. A Communication Characterisation of Splash-2 and Parsec. In *IEEE International Symposium on Workload Characterization (IISWC)*, pages 86–97, 2009.

[7] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 72–81, 2008.

[8] J. E. Boillat and P. G. Kropf. A Fast Distributed Mapping Algorithm. In *Joint International Conference on Vector and Parallel Processing (CONPAR 90 – VAPP IV)*, pages 405–416, 1990.

[9] S. Bokhari. On the Mapping Problem. *IEEE Transactions on Computers*, C-30(3):207–214, 1981.

[10] B. Brandfass, T. Alrutz, and T. Gerhold. Rank reordering for MPI communication optimization. *Computers & Fluids*, 80(July):372–380, 2013.

[11] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. hwloc: A Generic Framework for Managing Hardware Affinities in HPC Applications. In *Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 180–186, 2010.

[12] H. Chen, W. Chen, J. Huang, B. Robert, and H. Kuhn. MPIPP: An Automatic Profile-guided Parallel Process Placement Toolset for SMP Clusters and Multiclusters. In *ACM/IEEE International Conference for High*

Performance Computing, Networking, Storage and Analysis (SC), pages 353–360, 2006.

[13] C. Chevalier and F. Pellegrini. PT-Scotch: A Tool for Efficient Parallel Graph Ordering. *Parallel Computing*, 34(6-8):318–331, July 2008.

[14] E. H. M. Cruz, M. Diener, M. A. Z. Alves, and P. O. A. Navaux. Dynamic thread mapping of shared memory applications by exploiting cache coherence protocols. *Journal of Parallel and Distributed Computing (JPDC)*, 74(3):2215–2228, mar 2014.

[15] E. H. M. Cruz, M. Diener, L. L. Pilla, and P. O. A. Navaux. An Efficient Algorithm for Communication-Based Task Mapping. In *International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 207–214, 2015.

[16] E. H. M. Cruz, M. Diener, L. L. Pilla, and P. O. A. Navaux. Hardware-assisted thread and data mapping in hierarchical multicore architectures. *ACM Trans. Archit. Code Optim.*, 13(3):28:1–28:28, Sept. 2016.

[17] E. H. M. Cruz, M. Diener, M. S. Serpa, P. O. A. Navaux, L. Pilla, and I. Koren. Improving communication and load balancing with thread mapping in manycore systems. In *Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 93–100, 2018.

[18] W. J. Dally. GPU Computing to Exascale and Beyond. Technical report, nVidia, 2010.

[19] M. Deveci, K. Kaya, B. Ucar, and U. V. Catalyurek. Fast and High Quality Topology-Aware Task Mapping. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 197–206, 2015.

[20] K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, and U. V. Catalyurek. Parallel hypergraph partitioning for scientific computing. In *IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, pages 124–133, 2006.

[21] J. Edmonds. Maximum matching and a polyhedron with 0,1-vertices. *Journal of Research of the National Bureau of Standards – Section B. Mathematics and Mathematical Physics*, 69B(1 and 2):125, 1965.

[22] R. Glantz, H. Meyerhenke, and A. Noe. Algorithms for Mapping Parallel Processes onto Grid and Torus Architectures. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 236–243, mar 2015.

[23] B. Hendrickson and R. Lelandy. The chaco users guide version 2.0. Technical report, Sandia National Laboratories, 1995.

[24] S. Ito, K. Goto, and K. Ono. Automatically optimized core mapping to subdomains of domain decomposition method on multicore parallel environments. *Computers & Fluids*, 80:88–93, jul 2013.

[25] E. Jeannot and G. Mercier. Near-optimal placement of MPI processes on hierarchical NUMA architectures. In *Euro-Par Parallel Processing*, pages 199–210, 2010.

[26] E. Jeannot, G. Mercier, and F. Tessier. Process Placement in Multicore Clusters: Algorithmic Issues and Practical Techniques. *IEEE Transactions on Parallel and Distributed Systems*, 25(4):993–1002, apr 2014.

[27] E. Jeannot, G. Mercier, and F. Tessier. Topology and Affinity Aware Hierarchical and Distributed Load-balancing in Charm++. In *Workshop on Optimization of Communication in HPC (COM-HPC)*, pages 63–72, 2016.

[28] H. Jin, M. Frumkin, and J. Yan. The OpenMP implementation of NAS Parallel Benchmarks and Its Performance. Technical Report October, NASA, 1999.

[29] M. T. Jones. Inside the linux 2.6 completely fair scheduler. `https://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler/`, 2009. [Online; accessed June-2018].

[30] Z. Jovanovic and S. Maric. A heuristic algorithm for dynamic task scheduling in highly parallel computing systems. *Future Generation Computer Systems*, 17(6):721–732, apr 2001.

[31] G. Karypis and V. Kumar. Metis – unstructured graph partitioning and sparse matrix ordering system, version 2.0. Technical report, University of Minnesota, Department of Computer Science, 1995.

[32] G. Karypis and V. Kumar. Parallel Multilevel K-way Partitioning Scheme for Irregular Graphs. In *ACM/IEEE Conference on Supercomputing*, pages 1–21, 1996.

[33] G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, Dec. 1998.

[34] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 190–200, 2005.

[35] H. Luo, P. Li, and C. Ding. Thread data sharing in cache: Theory and measurement. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 103–115, New York, NY, USA, 2017. ACM.

[36] P. Luszczek, J. J. Dongarra, D. Koester, R. Rabenseifer, B. Lucas, J. Kepner, J. Mccalpin, D. Bailey, D. Takahashi, J. Jack, and R. Rabenseifner. Introduction to the HPC Challenge Benchmark Suite. Technical report, 2005.

[37] L. McVoy and C. Staelin. Lmbench: Portable Tools for Performance Analysis. In *USENIX Annual Technical Conference (ATC)*, pages 23–38, 1996.

[38] F. Pellegrini. Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs. In *Scalable High-Performance Computing Conference (SHPCC)*, pages 486–493, 1994.

[39] J. Shalf, S. Dosanjh, and J. Morrison. Exascale Computing Technology Challenges. In *High Performance Computing for Computational Science (VECPAR)*, pages 1–25, 2010.

[40] J. Sonnek, J. Greensky, R. Reutiman, and A. Chandra. Starling: Minimizing Communication Overhead in Virtualized Computing Platforms Using Decentralized Affinity-Aware Migration. In *International Conference on Parallel Processing (ICPP)*, pages 228–237, sep 2010.

[41] F. Trahay, F. Rue, M. Faverge, Y. Ishikawa, R. Namyst, and J. Dongarra. EZTrace: a generic framework for performance analysis. In *International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 618–619, 2011.

[42] D. Unat, A. Dubey, T. Hoefler, J. Shalf, M. Abraham, M. Bianco, B. L. Chamberlain, R. Cledat, H. C. Edwards, H. Finkel, K. Fuerlinger, F. Hannig, E. Jeannot, A. Kamil, J. Keasler, P. H. J. Kelly, V. Leung, H. Ltaief, N. Maruyama, C. J. Newburn, and M. Pericás. Trends in data locality abstractions for hpc systems. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 28(10):3007–3020, Oct 2017.

[43] W. Wang, T. Dey, J. Mars, L. Tang, J. W. Davidson, and M. L. Soffa. Performance Analysis of Thread Mappings with a Holistic View of the Hardware Resources. In *IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, 2012.

[44] J. Zhai, T. Sheng, and J. He. Efficiently Acquiring Communication Traces for Large-Scale Parallel Applications. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 22(11):1862–1870, 2011.