2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing

An Efficient Algorithm for Communication-Based Task Mapping

Eduardo H. M. Cruz*, Matthias Diener*, Laércio L. Pilla†, Philippe O. A. Navaux*
*Informatics Institute, Federal University of Rio Grande do Sul, Porto Alegre, Brazil
{ehmcruz, mdiener, navaux}@inf.ufrgs.br
†Department of Informatics and Statistics, Federal University of Santa Catarina, Florianópolis, Brazil
laercio.pilla@ufsc.br

***Abstract*—The communication between tasks of a parallel application is an important characteristic to consider when mapping tasks to computing cores due to possible differences in communication performance. Within a machine, performance differences are introduced by the memory hierarchy, in which cache memories can be shared by groups of cores and intra-chip interconnections are faster than inter-chip interconnections. In cluster and grid systems, the network imposes an additional communication latency. By mapping tasks that communicate to cores nearby on the memory hierarchy, or to the same nodes in clusters or grids, the communication of parallel applications is optimized, leading to increased performance and energy efficiency.**

In the task mapping context, one of the most important aspects to be considered is the mapping algorithm, as it determines the improvements that can be achieved. Since the problem of finding the best mapping is NP-Hard, heuristics must be employed to find an approximate solution in feasible time. In this paper, we present EagerMap, a new algorithm to perform communication-based mapping that is based on a greedy grouping strategy applied hierarchically. Experimental evaluation indicates that the execution time of our algorithm is 10 times faster than the state-of-the-art, and presents higher performance improvements. Due to its low execution time and high stability, EagerMap is also suitable for online task mapping, where tasks are migrated during execution.

***Keywords*-task mapping; communication; hardware topology; memory hierarchy;**

I. Introduction

Parallel architectures introduce a complex hierarchy to allow efficient memory accesses and communication among tasks. Within a machine, the hierarchy consists of several cache memory levels (private or shared), as well as Non-Uniform Memory Access (NUMA) behavior, in which memory banks are divided into NUMA nodes. In clusters and grids, the hierarchy introduces routers, switches and network links with different latencies, bandwidths and topologies.

The different levels of the hierarchy influence the communication performance between the tasks of a parallel application [1]. Communication through a shared cache memory or intra-chip interconnection is faster than communication between processors due to the slower inter-chip interconnections [2]. Likewise, communication within a machine is faster than the communication between nodes in a cluster or grid. In this context, the mapping of tasks to processing units (PUs) plays a key role in the performance of parallel applications [3]. Tasks that communicate intensely should be mapped to PUs close together in the hierarchy.

The communication-based task mapping problem can be defined as follows [2]. 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.

The complexity of finding an optimal mapping is NP-Hard [4]. 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 communication-based task mappings. EagerMap is designed to work with symmetric tree hierarchies. 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 current approaches. After the coarsening, we map the group graph to the architecture graph.

II. Related Work

Previous studies evaluate the impact of task mapping considering the communication [1], showing that it can influence several hardware resources. In shared memory environments, communication-based task mapping reduces execution time, cache misses and interconnection traffic [2]. In the context of cluster and grid environments, mapping

1066-6192/15 $31.00 © 2015 IEEE
DOI 10.1109/PDP.2015.25

tasks that communicate to the same computing node reduces network traffic and execution time [5], [6]. Communication cost can also be minimized in virtualized environments [7], demonstrating its importance for cloud computing.

Several mapping algorithms have been proposed to optimize communication. Most traditional algorithms are based on graph partitioning, such as Zoltan [8] and Scotch [9]. They use generic graphs to represent the topology, and have a complexity of $O(N^3)$ [10]. Tree representations are used in TreeMatch [11], which can lead to more optimized algorithms. 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. Furthermore, TreeMatch does not support mapping multiple threads to the same PU.

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 accuracy of MPIPP depends on the initial random mapping. Our previous work [2] uses Edmonds' graph matching algorithm to calculate mappings, but is limited to environments where the number of tasks and PUs is a power of two.

III. EagerMap – Greedy Hierarchical Mapping

EagerMap receives two pieces of input, a communication matrix containing the amount of communication between each pair of tasks as well as a description of the architecture hierarchy, and 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 [13], [14], [15], [16], which we obtained with the methodology described in Section IV-A2. We observe three essential characteristics in the communication behavior of the applications that need to be considered for an efficient mapping strategy:

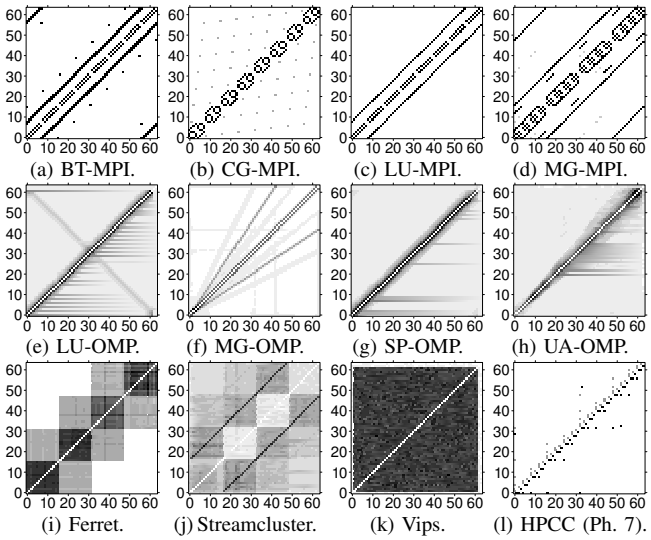


Figure 1. Example communication matrices for parallel applications consisting of 64 tasks. Axes represent task IDs. Cells show the amount of communication between tasks. Darker cells indicate more communication.

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, because 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.

A. Description of the EagerMap Algorithm

The algorithm requires two variables to be initialized previously: $nLevels$ 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). $execElInLevel$ 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 $execElInLevel$.

1) Top Level Algorithm: The top level mapping algorithm is shown in Algorithm 1. The algorithm 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

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      nElements ← nTasks;
6      commMatrix ← commMatrixInit;
7      for i←1 ; i<nLevels ; i←i+1 do
8          previousGroups ← groups;
9          [nGroups, groups] ←
           GenerateGroupsForLevel(commMatrix, nElements, i,
           previousGroups, execElInLevel[i]);
10         if i < nLevels-1 then
11             commMatrix ← RecreateMatrix(commMatrix, groups,
               nGroups);
12         nElements ← nGroups;
13     rootGroup.nElements ← nElements;
14     for i←1 ; i<nElements ; i←i+1 do
15         rootGroup.elements[i] ← groups[i];
16     MapGroupsToTopology(archTopologyRoot, rootGroup, map);
17     return map;
```

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      if nElements > avlGroups then
3          nGroups ← avlGroups;
4      else
5          nGroups ← nElements;
6      elPerGroup ← nElements / nGroups;
7      leftover ← nElements % nGroups;
8      for i←0 ; i<nElements ; i←i+1 do
9          chosen[i] ← 0;
10     gi ← 0;
11     for i←0 ; i<nElements ; i←i+inGroup do
12         inGroup ← elPerGroup;
13         if leftover > 0 then
14             inGroup ← inGroup + 1;
15             leftover ← leftover - 1;
16         newGroup ← GenerateGroup(commMatrix, nElements,
           inGroup, chosen, previousGroups);
17         newGroup.nElements ← inGroup;
18         newGroup.id ← gi;
19         groups[gi] ← newGroup;
20         gi ← gi + 1;
21     return [nGroups, groups];
```

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 7. After generating the groups of tasks for a level (line 9) (explained in Section III-A2), the algorithm generates a new communication matrix (line 11, discussed in Section III-A3). This step is necessary since we consider each group of tasks as the base element for mapping on the next hierarchy level.

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

```
   Input: commMatrix[][], totalElements, groupElements, chosen[],
          previousGroups[]
   Output: group
   LocalData: i, j, 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                  if w > wMax then
10                     wMax ← w;
11                     winner ← j;
12         chosen[winner] ← 1;
13         winners[i] ← winner;
14         group.elements[i] ← previousGroups[winner];
15     return group
```

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 7 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 14). 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 III-A4.

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, which are defined 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. The strategy works as follows. Each iteration of the loop in line 2 adds one element to the group. The added element, expressed by the $winner$ 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. After the loop, the master thread would select the $winner$ among the ones calculated by each thread.

3) Computing the Communication Matrix for the next Level of the Architecture Hierarchy: $RecreateMatrix$, described in Algorithm 4, regenerates the communication

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              newCommMatrix[i][j] ← w;
9              newCommMatrix[j][i] ← w;
10     return newCommMatrix;
```

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      else if hardwareObj.nSharers > 1 then
6          for i←0; i<group.nElements; i←i+1 do
7              MapGroupsToTopology(hardwareObj.linked[i],
               group.elements[i], map);
8      else
9          MapGroupsToTopology(hardwareObj.linked[0], group,
           map);
```

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.

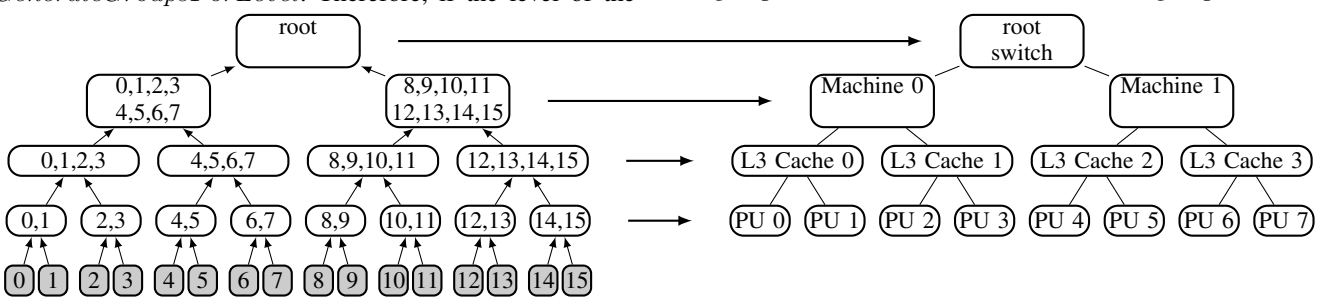
4) Mapping the Group Tree to the Architecture Topology Tree: The algorithm to map the group tree to the architecture topology tree is $MapGroupsToTopology$, detailed in Algorithm 5. It is a recursive algorithm that performs a recursion for each level of the architecture hierarchy. The stop condition of the recursion is when it reaches the lowest level of the architecture topology, the processing unit (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 5), the algorithm only assigns one hardware object of the following level to each group and and recursively calls itself for each element of the following level. Otherwise, the level is private and is not considered for mapping (line 8).

B. 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. 2b illustrates the hierarchy of the machine. 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, its value is not used. Position 1 represents the processing units (the cores in this case), its value is 8. Position 2 represents the L3 caches, its value is 4. Finally, position 3 represents the machines, its value is 2.

Regarding Algorithm 1 ($MapAlgorithm$), the loop in line 7 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 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 amount of 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 7 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 groups of groups of tasks with 2 elements in each group.



(a) Task group tree. (b) 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.

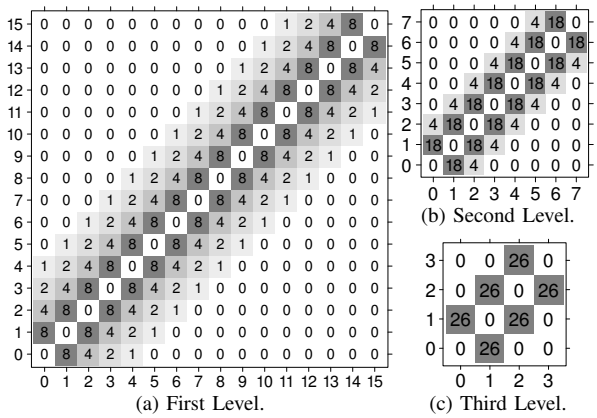


Figure 3. Communication matrices used in our example.

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.

In the third iteration of the loop in line 7 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 groups of groups of groups of tasks, 2 elements in each group. To complete the group tree, the $rootGroup$ variable points to these 2 groups of groups of groups of tasks. Afterwards, in line 14 of $MapAlgorithm$, the task group tree shown in Fig. 2a is finished.

$MapGroupsToTopology$ (Algorithm 5) maps the task group vertices to vertices 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 $x/2$.

C. Complexity of the Algorithm

For the analysis of the complexity of the algorithm, 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.

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

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 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 $2P-1$.

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

$$\leq 2\big(O(P^3) + O(P^2)\big) + \big(O(N^3) + O(N^2)\big) + O(N) = 3\big(O(N^3) + O(N^2)\big) + O(N) = O\left(N^3\right) \tag{6}$$

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

IV. Evaluation of EagerMap

In this section, we first show how we evaluate our proposal. Afterwards, we present the results obtained on its empirical evaluation.

A. Methodology of the Mapping Comparison

We discuss the benchmarks and architecture used in our evaluation, how we obtain the communication matrices, and other mapping strategies to which we compare EagerMap.

1) Benchmarks: For the evaluation of the mapping algorithms, we use applications from the MPI implementation of the NAS parallel benchmarks (NPB) [13], the OpenMP NPB implementation [14], the High Performance Computing Challenge benchmark [15] and the PARSEC benchmark suite [16]. The NAS benchmarks were executed with the *B* input size, HPCC with an input matrix with 4000^2 elements and PARSEC was executed with the *native* input size.

2) Generating the Communication Matrices: For the MPI based benchmarks, we used the eztrace framework [17] to trace all MPI messages sent by tasks and built a communication matrix based on the number of messages sent between tasks. 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 [18], similarly to [19]. 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.

3) Hardware Architecture: The hardware architecture used in our experiments is illustrated in Fig. 5, with 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 (case A) can communicate through the fast L1 or L2 caches and have the highest communication performance. Tasks that run on different cores (case B) have to communicate through the slower L3 cache, but can still benefit from the fast intra-chip interconnection. When tasks communicate across physical processors (case C), they need to use the slow inter-chip interconnection. Hence, the communication performance in case C is the slowest in this architecture.

4) Comparison: We compare EagerMap to (i) a Random mapping, which is an average result of 30 different random mappings; (ii) TreeMatch [11] version 0.2.3; (iii) Scotch [9] version 6.0; and (iv) MPIPP [12].

B. Results

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

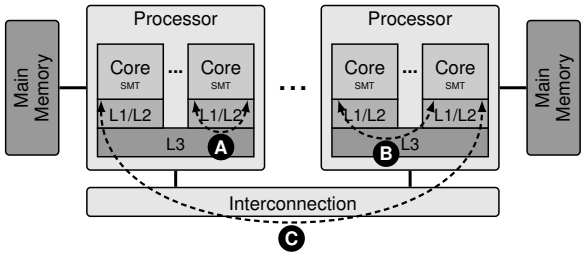


Figure 5. Hardware architecture used in our experiments.

1) Performance of the Mapping Algorithms: Fig. 6 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.

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 [20].

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

Fig. 7 presents the mapping quality results for the communication matrices previously illustrated in Fig. 1. Applications with more structured communication patterns

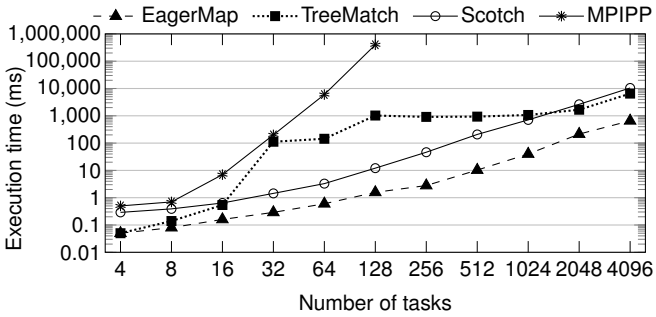


Figure 6. Execution time (in ms) of the mapping algorithms, for different numbers of tasks to be mapped.

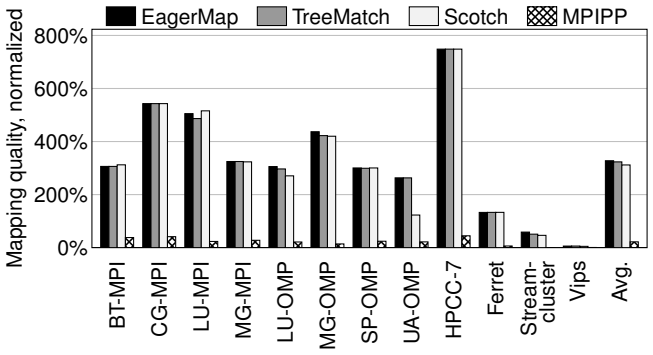


Figure 7. Comparison of the mapping quality, normalized to the random mapping. Higher values are better.

presented the highest improvements, as expected. CG-MPI, LU-MPI and HPCC (Phase 7) presented the highest improvements in accuracy because their communication can be easily optimized by mapping neighboring tasks together. In BT-MPI and MG-MPI, near and distant tasks communicate, presenting more challenges for the mapping algorithm. This happens because 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 in accuracy 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 optimize 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 are endless, and due to this it is 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 III.

3) Mapping Stability: Another important metric to consider for mapping algorithms is the numbers of migrations performed due to changes in the communication matrix, which we call stability. A high stability is important for online mapping techniques. In algorithms with low stability, small changes in the communication lead to unnecessary task migrations. When the stability of the algorithm is higher, more differences in the communication matrix are required to migrate tasks.

The stability results are shown in Fig. 8 for the BT-MPI benchmark running with 64 tasks. The x-axis corresponds to the percentage difference of a matrix relative to a base matrix. A value of $k\%$ indicates that all values of the matrix were changed randomly up to a limit of $k\%$ of the maximum of the matrix. Since the communication pattern itself does not change, ideally there should be no migrations. The y-axis shows the number of task migrations resulting from the changes. Lower numbers of migrations are better. The results show that our proposed algorithm has a higher stability than previous mechanisms. In this way, EagerMap introduces less overhead than the other algorithms, since they introduce unnecessary task migrations.

4) Performance Improvements: We executed applications using the mapping obtained with the algorithms. The execution time results for the MPI and OpenMP NAS Benchmarks are shown in Fig. 9. For these applications, since their communication pattern is stable, we calculated the mapping

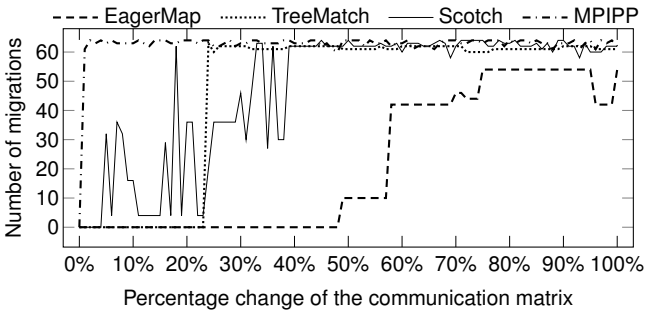


Figure 8. Comparison of the mapping stability. Number of migrations depending on the percentile change of the communication matrix for an application consisting of 64 tasks. Less migrations are better.

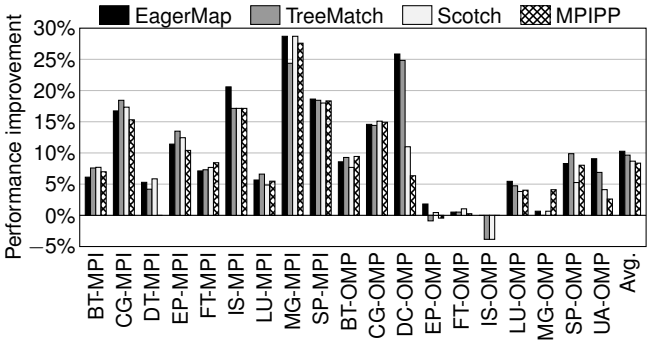


Figure 9. Performance improvement compared to the OS mapping of the NAS-MPI and NAS-OMP benchmarks.

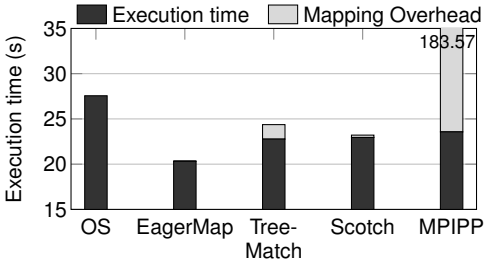


Figure 10. Execution time of HPCC using online mapping.

statically. As a case study for online mapping, we used the HPCC Benchmark, shown in Fig. 10, 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 represented less 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 IV-B2. 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. 10) show a larger improvement by EagerMap than related work. Also,

the lower overhead of our algorithm does not harm the application performance as much as TreeMatch and MPIPP.

V. Conclusions

In parallel applications, the mapping of tasks to cores influences the communication performance. By mapping tasks that communicate to cores nearby in the memory hierarchy or to the same node in clusters or grids, performance is improved by making use of faster interconnections. The mapping algorithm selects which core will execute each task and plays a key role in this type of mapping.

In this paper, we proposed a new mapping algorithm, EagerMap. 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. Furthermore, its high stability makes EagerMap more suitable for online mapping.

For the future, we will extend EagerMap to support arbitrary hardware hierarchies, not only those based on trees. EagerMap is licensed under the GPL and is available at `http://github.com/ehmcruz/eagermap`.

Acknowledgment

This work was supported by CNPq, Capes and FAPERGS.

References

[1] 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.

[2] 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*, vol. 74, no. 3, 2014.

[3] J. Zhai, T. Sheng, and J. He, "Efficiently Acquiring Communication Traces for Large-Scale Parallel Applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 11, pp. 1862–1870, 2011.

[4] S. Bokhari, "On the Mapping Problem," *IEEE Transactions on Computers*, vol. C-30, no. 3, pp. 207–214, 1981.

[5] B. Brandfass, T. Alrutz, and T. Gerhold, "Rank reordering for MPI communication optimization," *Computers & Fluids*, pp. 372–380, Jan. 2012.

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

[7] 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)*, Sep. 2010, pp. 228–237.

[8] 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)*, 2006.

[9] F. Pellegrini, "Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs," in *Scalable High-Performance Computing Conference (SHPCC)*, 1994.

[10] T. Hoefler and M. Snir, "Generic topology mapping strategies for large-scale parallel architectures," in *International Conference on Supercomputing (ICS)*, 2011.

[11] E. Jeannot, G. Mercier, and F. Tessier, "Process Placement in Multicore Clusters: Algorithmic Issues and Practical Techniques," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 4, pp. 993–1002, Apr. 2014.

[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 *International Conference on Supercomputing*, 2006.

[13] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga, "The NAS Parallel Benchmarks," *International Journal of High Performance Computing Applications*, vol. 5, no. 3, pp. 66–73, 1991.

[14] H. Jin, M. Frumkin, and J. Yan, "The OpenMP implementation of NAS Parallel Benchmarks and Its Performance," 1999.

[15] 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," 2005.

[16] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[17] 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)*, May 2011.

[18] C. Luk, R. Cohn, R. Muth, and H. Patil, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[19] N. Barrow-Williams, C. Fensch, and S. Moore, "A Communication Characterisation of Splash-2 and Parsec," in *IEEE International Symposium on Workload Characterization (IISWC)*, 2009.

[20] L. McVoy and C. Staelin, "lmbench: Portable Tools for Performance Analysis." in *USENIX annual technical conference*, no. January, 1996.