CMLB: A Communication-aware and Memory Load Balance Mapping Optimization for Modern NUMA Systems

1st Jingbo Li
High Performance Computing Lab
Xi'an Jiaotong University
Xi'an, China
lijingbo17@stu.xjtu.edu.cn

2nd Yuxin Zhang
High Performance Computing Lab
Xi'an Jiaotong University
Xi'an, China
zhangyuxinxhu@163.com

3rd Xingjun Zhang
High Performance Computing Lab
Xi'an Jiaotong University
Xi'an, China
xjzhang@xjtu.edu.cn

***Abstract*—For parallel applications, mapping parallel threads to cores according to the access behavior plays an important role to optimize the applications performance. The imbalance between thread communication and memory bandwidth will severely increase the average latency and the execution time of the application when running on modern non-uniform memory access (NUMA) architecture. Previous studies on thread mapping mostly focus on the locality of memory accesses to improve the communication efficiency. However, maximizing the locality may cause memory congestion because of the imbalance on memory bandwidth between nodes. In this paper, a communication-aware and memory load balance mapping algorithm (CMLB) for modern NUMA systems is propose which works on improving the locality of communication as well as avoiding memory congestion problem. To verify the effectiveness of the algorithm, the rotor35-omp program and applications from NAS parallel benchmark and Parsec benchmark are used. Experimental results show that CMLB could greatly balance the memory bandwidth between nodes to reduce the memory latency and also improve the locality of communication, get the better performance than the state-of-the-art mapping methods.**

***Index Terms*—High performance computing, Parallel applications, Memory congestion, Thread mapping, Non-uniform memory access**

I. Introduction

Modern non-uniform memory access (NUMA) systems consist of numerous processor cores and complex hierarchy of memory. Within a NUMA multiprocessors system, each processor consists of a group of processor cores, which is associated with one or more memory controllers and memory devices [1], [2]. And the group of processor cores is referred to as a NUMA node [3], [4]. Although there are QuickPath Interconnect (QPIs) between NUMA nodes [5], [6], accessing a remote NUMA node has a longer latency than accessing the memory of the local NUMA node.

A parallel application usually has multiple tasks, each task is executed on a processor core as a thread or a process [7]. In NUMA systems, there are two types of communication. Local communication means tasks communicate with each other through the shared cache memory or intra-chip interconnection. And remote communication indicates tasks communicate with each other through QPIs. Due to the feature of the memory access on NUMA systems, local communication is faster than remote communication. In this context, the mapping of tasks to processor cores plays a key role in the performance of parallel applications [8], [9].

Previous works [9], [10] attempted to optimization parallel performance via communication-aware task mapping, which is based the communication matrix and the architecture graph. Specifically, the symmetric communication matrix represents the amount of communication between threads, and the architecture graph stands for the machine components topology include cores, cache memories, NUMA nodes. These methods map the tasks which communicate intensely to cores close together at the same NUMA node, has a good work on improving the locality of communication. However, recent works [3], [11] show that maximizing locality does not always improve the performance of applications, because the locality-based mapping may cause memory congestion which means that most of the memory access events will occur in a particular node, it will take longer average access latency.

In this paper, the communication-aware and memory load balance mapping algorithm (CMLB) is proposed, which can efficiently improve the locality of communication as well as avoiding memory congestion problem. The main contributions are as follows. 1) an efficient memory access detecting and analyzing method is designed. Specially, based on the slide window, an access phases division algorithm is proposed which can effectively deal with the sequential relationship of continuous access. 2) a thread grouping algorithm is proposed which can ensure load balance when increasing locality. 3) to verify the effectiveness of the algorithm, rotor35-omp, NAS Parallel Benchmark (NPB) and Parse benchmark applications with different access behavior are verified from execution time, QPI improvement, imbalance of memory bandwidth and

This work was funded by the National Key Research and Development Program of China (2016YFB0200902).

Xingjun Zhang is the corresponding author.

average latency four aspects.

The remainder of this paper is organized as follows: Section II introduce the review of the mapping algorithm and closely related studies; Section III presents the proposed CLMB algorithm in detail; Section IV provides a demonstration its performance; and Section V conclude this paper and discuss the potential future research.

II. Related work

Previous studies [8], [12] found that in shared memory environments a communication-aware task mapping can reduce execution time, cache misses and interconnection traffic. Some task mapping methods considering the communication between tasks have been proposed. Mapping the tasks which communicate intensely to cores close together at the same NUMA node, has a good work on improving the locality of communication and can improve the performance of applications [12].

As the dependencies between tasks can be represented by a direct acyclic graph (DAG) diagram, somg traditional algorithms are based on graph partitioning, such as Zoltan [13] and Scotch [14]. In recent years, there are some algorithms based on a communication matrix which is symmetric and represents the amount of communication between threads. Jeannot et al. [15] proposed TreeMatch algorithm which generates all possible group of tasks. However, this algorithm have a high complexity. To improve the efficiency of this algorithm, Cruz et al. [2] proposed the EagerMap in which a greedy policy was used to group tasks. By simplifying calculations, EagerMap got a lower complexity of $O(N^3)$ than TreeMatch. Soomro et al. [16] proposed ChoiceMap algorithm, in which a fair policy to pair threads was adopted. As a result, this algorithm got a better performance on some benchmark applications than EagerMap.

There is no doubt that the above of the mapping algorithm both need to detect communicate behaviors between tasks. EagerMap and ChoiceMap use Numalize [17] tool based on Intel Pin tools [18] to detect communication. And ChoiceMap also use CommDetective [1] which used Linux Perf tools to detect communication. Besides, CommDetective has a lower overhead than Numalize.

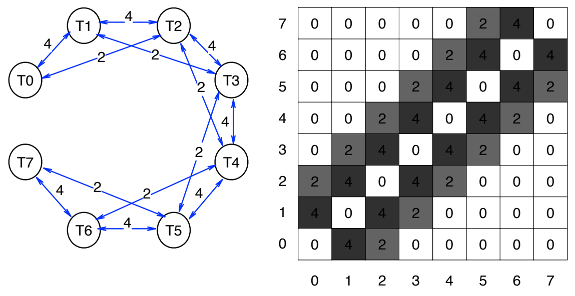
However, these studies focus on the locality of memory accesses to improve the communication efficiency which only considering features of communication between tasks and maximizing the locality may cause memory congestion. And the imbalance between thread communication and memory bandwidth will severely increase the average latency and the execution time of the application when running on modern NUMA architecture [3]. Therefore, in this papre, CMLB is proposed to solve this issue, which can efficiently improve the locality of communication as well as avoiding memory congestion problem.

III. CMLB: Communication-aware and memory load balance mapping

A. Detecting and analyzing the memory access of threads

The memory access behaviors of threads are the important basis for the thread grouping algorithm. The CommDetective [1] is adopted to generate communication matrix to improve the locality of communication. To avoid the memory congestion, a memory access load detection method is proposed. By combining the communication matrix and memory access load vector, CMLB can identify the current application and machine state in time.

1) Detecting communication between threads: In parallel applications, based on shared memory programming such as OpenMP and Pthreads, data are exchanged and shared through the shared memory between threads. Inter-thread communication is defined as the operation of different threads that read and write the same cache line in the shared memory.



(a) Communication graph (b) Communication matrix

Fig. 1: Illustration of communication graph and communication matrix.

Communication behavior of an application can be represent as a communication graph or task integration graph [19]. Communication graph can be converted into a matrix, which contains the amount of communication between thread pairs where the indices represent the thread IDs, the communication matrix is symmetric with zero diagonal, as shown in Fig. 1.

To build the matrix, CommDetective is adopted which is based on Linux Perf tools, and use the perf_event_open() call to extract the memory access events from hardware performance monitoring unit (PMU). A PMU interrupt is referred to a "sample". The Intel's Precise Event-Based Sampling (PEBS) [20] facility offers the ability to inspect the effective address accessed by the instruction on an event overflow for certain kinds of events such as loads and stores. When two access events have the same access address, the two events are considered as occur a communication, and update it in the communication matrix.

In this paper, the set perf_event_open() in CommDetective is set to perform sampling with sample frequency 1000 that means this tool will extract a sample from PMU every 1000 access events elapse. Besides, CommDetective to monitor two type of events for sampling are set to monitor: all loads micro operations and all stores micro operations.

2) Analyzing the memory access load of each thread: Diffierend from communication matrix, access load vector represents the memory access load of each thread. A high value in the access load vector imply the thread will make lots of pressure on memory. The whole process of analyzing the access load is divided into two parts: select and record all the DRAM access events in an access table.

Firstly, the events which access DRAM due to L3 cache miss when Perf extracts the samples of access events in CommDetective are recorded. For each event, the related information include access address, thread id and the timestamp of accessing are extracted.

Secondly, the timestamp of each access record is converted to the time in nanoseconds (ns) according to the CPU clock frequency, the new timestamp is calculated by Equation 1.

$$tsc = \frac{cycles \times 1000000}{feq} \tag{1}$$

where $cycles$ represents the old timestamp in CPU clock cycles, and feq means the CPU clock frequency in khz. And then a time slice (1ms) needs to be set to merge the records whose timestamp interval is within it. The merged record includes the timestamp, the thread id list, and the number of the original records which indicates how many memory accesses are performed in this time slice. The higher this value, the higher the memory access data traffic in this time slice. Therefore, it is necessary to focus on extracting the memory access feature of the merged records with a high number of original records, because these records will have a large impact on DRAM.

It is obvious that the number of memory access changed regularly over time, as shown in Fig. 2, in which four applications in NPB are tested, and the area between the red lines represents an access phase. It can been seen that the time interval which has the large amount of memory access is likely to cause memory congestion. Besides, there are some phases where the amount of access first increases and then decreases over the whole execution.

Therefore, an access phases division algorithm based on the slide window algorithm is proposed, as shown in Algorithm 1. When using Perf to extract the access events, there are some noises to influence the data of time series. Therefore, linear smooth are used to denoise the time series of the access amount. According to the method in Rocka [21], the outliers in the time series data are less than 5%, so the data with the top 5% of the value in the series are deleted, and replaced by linear interpolation vaule. After smoothing the data, Algorithm 1 are used to divide the access into phases. Two pointers, $left$ and $right$, are set in line 3, and a minimum window width are defined in line 4 which represents the minimum time interval of an access phase. After that, the $right$ pointer are moved forward, when the $right$ pointer meet the point with the low value, record the phase from $left$ pointer to $right$ pointer, and move the $left$ pointer to the position of the $right$ pointer for the next search.

Algorithm 1 Access phases disvsion algoirthm

```
Input: the time series of the access amount $s$
Output: the phases of the memory access $pha$
 1: num ← len(s) ×0.05
 2: low_value ← mean(sorted($s$)[:num])
 3: $left \leftarrow 0$ ; $right \leftarrow 0$
 4: min_width ← 100
 5: while $right \leq$ len($s$) do
 6:     if $s[right] \leq$ low_value and ($right$ - $left$) $\geq$ min_width then
 7:         append the $s$[$left$:$right$+1] to $pha$
 8:         $left \leftarrow right$
 9:     $right$+=1
10: return $pha$
```

For practical considerations, there exits some small phases due to continuous points with low value in the series, so minimum window width (100) is set to avoid existing too many small phases which will influence the result. By using Algorithm 1, all the access phases from an application are identified, as shown in Fig. 2.

B. Computing and Grouping Threads

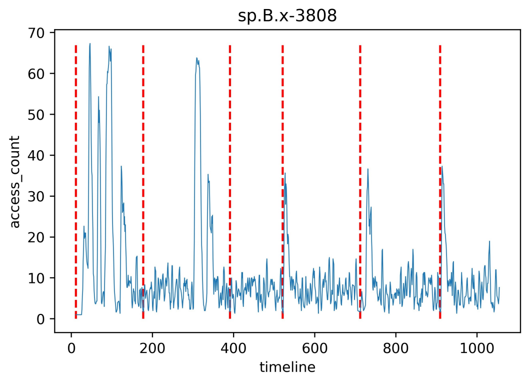
1) Thread Grouping Algorithm: The design principle of the grouping algorithm is to improve the locality of the memory access and ensure that the memory access load of each NUMA node is balanced.

Firstly, the number of groups of all threads is determined. In order to make full use of the computing resources, all the running threads are distributed on all NUMA nodes. Therefore, the number of the groups of threads is the number of NUMA nodes. With hwloc [22] tool, the machine components topology including the number of nodes, the number of cores and memory caches in each node are obtained.

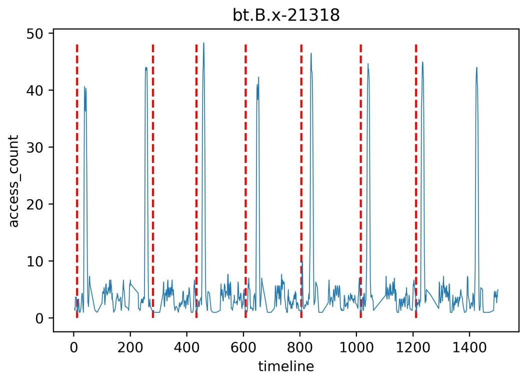
To select one thread for grouping, the greedy policy in EagerMap is adopted. For example, according to the list of grouped threads $t = [t_1, t_2, t_3]$ and the communication matrix, the amount of communication between the remaining threads and t_i are calculated. Then, the communication from high to low are sorted. Each thread t_w from the remaining threads are selected. The sum of communication with the threads in t_i is calculated, as shown in Equation 2.

$$C = \sum_{i=1}^{n} CommMatrix[t_w][t_i] \tag{2}$$

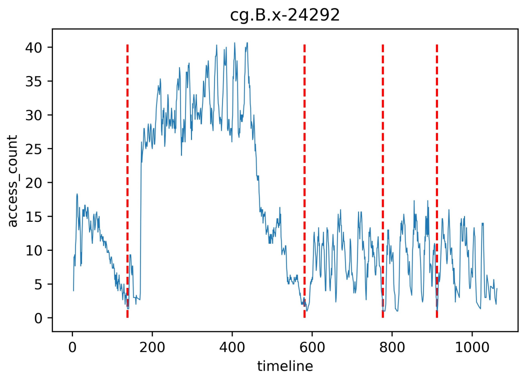
Where $CommMatrix$ represents the communication matrix, n represents the number of grouped threads in t_i, and C represents the sum of the communication between one remaining thread t_w and all the threads in t_i. And then all the sum of communication between the remaining threads and t_i are calculated. By visiting the communication rank list and



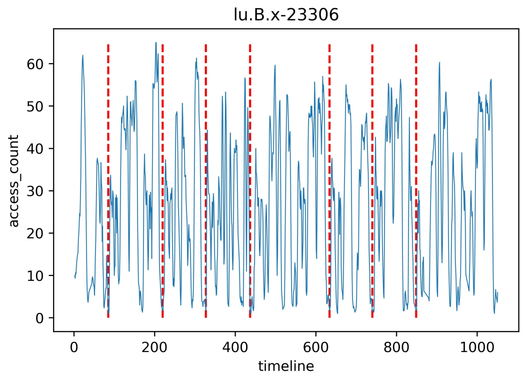
(a) SP-OMP



(b) BT-OMP



(c) CG-OMP



(d) LU-OMP

Fig. 2: The time series of access amount in NPB benchmark

choosing the first element where the thread has the largest communication with all the grouped threads, a thread can be selected to insert in the group. And then check that if inserting the t_{w1} will destroy the memory access balance in this group according to access load vector. If not, insert t_{w1} in the group, otherwise, choose the next thread t_{w2} in the communication rank list and repeat the above steps until select a thread to insert in the group.

2) Load Balance Judgement: After determining the number of the groups, it is necessary to calculate the average memory load of each group according to the access load vector, as shown in Equation 3.

$$Aml = \frac{1}{groups} \sum_{i=1}^{n} \frac{1}{m} \sum_{j=1}^{m} d_j p_i \tag{3}$$

Where $groups$ represents the number of the core groups, n means the length of the access load vector, and Aml represents the average memory load of each group. d_j stands for the access amount of the j^{th} time slice. p_i is the access load vector of the i^{th} phase. m stands for the number of the phases.

As Algorithm 2 shown, from line 4 to line 6, the current access load of this group are calculated, and $lastLoad$ and n are known from line 5 to line 6. From line 12 to line 18, $maxLoad$ and $minLoad$ which represent the sum of the access load of

Algorithm 2 JudgeLoadBalance algorithm

Input: the candidate thread tid, the list of grouped threads $cur\_grouped$, the list of remaining threads $last\_threads$
Output: bool type

```
1:  cur_load ← AccVector[$tid$]
2:  for grouped_t in $cur\_grouped$ do
3:      cur_load += AccVector[grouped_t]
4:  last_load ← sum(AccVector) / num_nodes - cur_load
5:  last_num_tids ← per_group_tids – len($cur\_grouped$)
6:  if last_num_tids == 1 and $tid$ in overLoad_list then
7:      return False
8:  else
9:      return True
10: f ← sorted(last_threads.remove($tid$),reverse=True)
11: max_load ← sum(f[ : last_num_tids])
12: min_load ← sum(f[-last_num_tids:])
13: if last_load $\geq$ min_load and last_load $\leq$ max_load then
14:     return True
15: else
16:     return False
```

the top n threads and last n threads after sorted are calculated. When selecting a thread from the rank list, the thread t_w insert the group at first. All the remaining threads are sored according to the value in the $AccVector$ from high to low. If $lastLoad$ is between $minLoad$ and $maxLoad$, the thread t_w will not destroy the balance in this group. Otherwise, t_w will destroy the balance.

C. Theoretical Evaluation of the Algorithm

In this section, the theoretical performance of CMLB is verified compared with the state-of-art mapping methods including EagerMap, TreeMatch and ChoiceMap from the amount of remote communication and the access balance of NUMA nodes.

After getting the mapping result from a mapping method, remote communication and the load balance of nodes can be measured according to the communication matrix and access load vector. Due to both of the mapping methods are the communication-based methods, the remote communication can be calculated using the communication matrix, as shown in Equation 4 and 5.

$$RemoteComm = \sum_{i=1}^{n} \sum_{j=i+1}^{n} CommVal(map[i], map[j]) \tag{4}$$

$$CommVal(g1, g2) = \sum_{m=1}^{n1} \sum_{n=1}^{n2} CommMat[g1[m]][g2[n]] \tag{5}$$

In Equation 4, n represents the number of nodes. The threads in a node are regarded as a group. $CommVal$ means the communication between two groups and map is the mapping result of all the groups. In Equation 5, $g1$, $g2$ are the two groups and m, n stand for the number of threads in the two groups.

The access load balance of the nodes is also a criterion to judge the performance of mapping method, shown sa Equation 6.

$$Load_{std} = \sqrt{\frac{\sum_{i=1}^{n}(L_i - L)}{n}} \tag{6}$$

Where, $Load_{std}$ represents the standard deviation of the access load between groups, n means the number of the groups. And L_i is the access load in the $i^{th} group$, L stands for the means of the access load of all the groups.

SP and BT applications are tested by the two metrics to compare the theoretical performance of CMLB, as shown in table I and II.

It can been seen that, although CMLB is slightly higher than EagerMap, TreeMatch and ChoiceMap in $RemoteComm$ of the two applications, they are all at an order of magnitude. As for $Load_{std}$ of the applications, CMLB is significantly lower than the other methods. CMLB could greatly reduce the difference of memory access load between nodes when the remote communication is not much higher than the other methods. Therefore, CMLB is better than the other mapping methods from the theoretical evaluation.

TABLE I: The result of SP-OMP

Method	$RemoteComm$	$Load_{std}$
CMLB	1.99×10^8	139.37
EagerMap	1.90×10^8	33455.01
TreeMatch	2.13×10^8	29343.56
ChoiceMap	1.92×10^8	33214.47

TABLE II: The result of LU-OMP

Method	$RemoteComm$	$Load_{std}$
CMLB	3.51×10^7	462.69
EagerMap	3.40×10^7	29310.60
TreeMatch	3.62×10^7	28913.45
ChoiceMap	3.54×10^7	28712.68

IV. Experiment and analysis

A. Experimental environment

In this paper, the platform which has two NUMA nodes and each node has 8 cores which share a L3 cache with 20 MB and a DRAM with 16 GB are used, as shown in Table III. The platform use hyper-threading technology, each core could run 2 threads at the same time.

The rotor35-omp program, NPB benchmark and Parsec benchmark are used to verify the performance of CMLB. The rotor35 is an parallel program of a compressor rotor square cavity flow model which simulates the 36-channel axial compressor rotor model. The numerical solution process constructs the control equations, as shown in Equation 7.

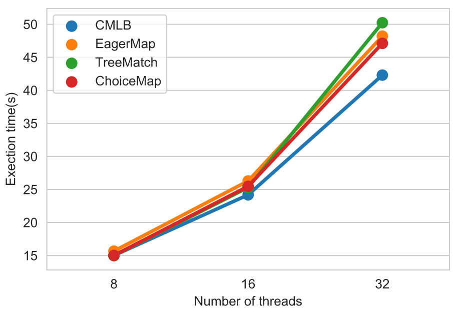
$$\frac{\partial Q}{\partial t} + \frac{\partial F_c}{\partial x} + \frac{\partial G_c}{\partial y} + \frac{\partial H_c}{\partial z} = \frac{\partial F_v}{\partial x} + \frac{\partial G_v}{\partial y} + \frac{\partial H_v}{\partial z} + I \tag{7}$$

Where Q is a conservation vector, F_c, G_c, and H_c are convection vectors along the three coordinate directions of x, y, and z respectively. F_v, G_v, and H_v means the viscosity vector along the three coordinate directions of x, y, and z respectively and I represents the source term.

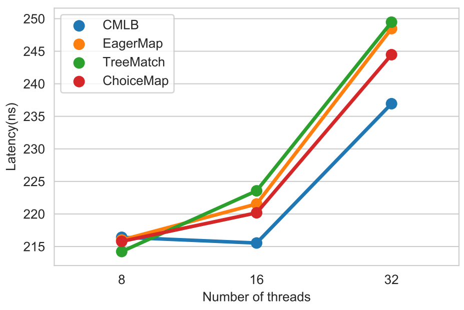
Except rotor35-omp program, four applications, SP, BT, LU and CG from NPB and three applications, Streamcluster, Facesim and Fluidanimate in Parsec benchmark are chosen to verify the effectiveness of the CMLB algorithm.

TABLE III: Platform configuration

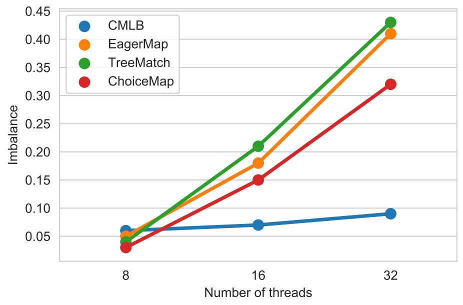
Property	Parameters
Architecture	2 nodes, 1 socket/node, 8 processors/socket
Processors	Intel Xeon E7-4809
Cache	8 ×(32 KB+32 KB) L1, 8 ×256 KB L2, 20MB L3
Memory	32GB DDR3, page size 4KB



(a) Execution time

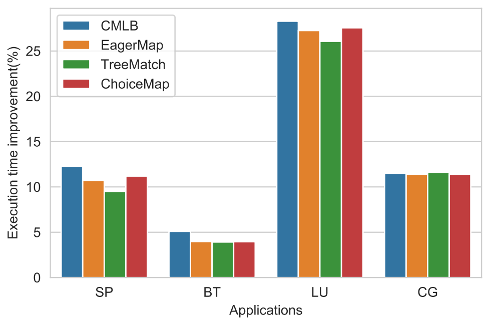


(b) Average latency

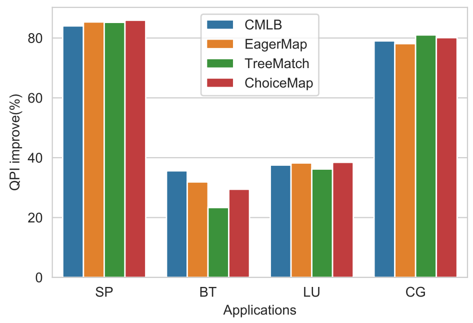


(c) Imbalance of memory bandwidth

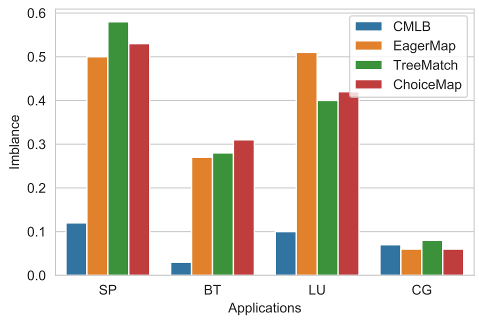
Fig. 3: Experimental test on rotor35-omp



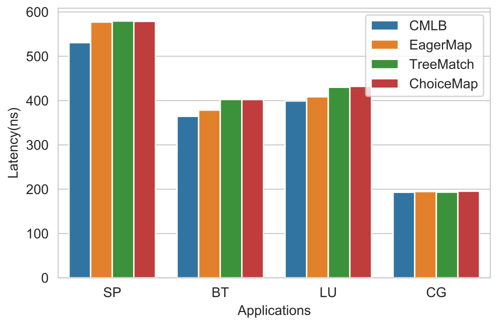
(a) Execution time improvement



(b) QPI improve



(c) Imbalance of memory bandwidth



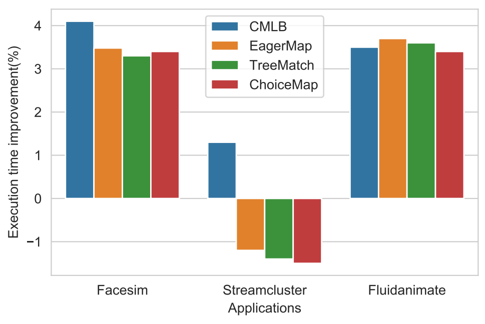
(d) Average latency

Fig. 4: Experimental test on NPB-OMP benchmark

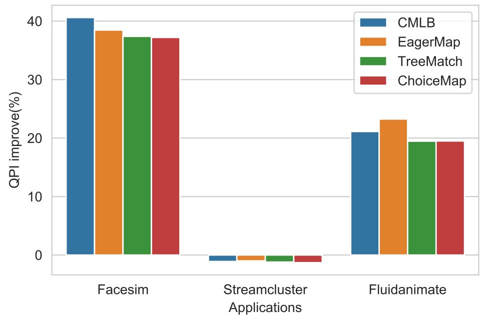
B. Test on rotor35-omp

Shown as Fig. 3, the optimization performance of CMLB is more obvious when the number of threads is larger. In Fig. 3(a), rotor35-omp with 32 threads have an obvious decrement about 6.25% on execution time when comparing with the other mapping algorithms. For 8 and 16 threads of rotor35-omp, CMLB has the similar execution time with others. This is because the data size of rotor35-omp increases as the number of threads increases. Therefore, the number of memory accesses of the program also increases, resulting in an increasing in the remote communication and the imbalance of memory bandwidth between nodes. The imbalance of bandwidth between all nodes is also tested, which is measured by the variance of all the nodes memory bandwidth. As shown in Fig. 3(c), CMLB has the lowest value of the imbalance in the 16 and 32 threads rotor35, which indicates CMLB could balance the memory access load between all nodes better than the other methods, so that CMLB has the lowest average latency of rotor35 as shown in Fig. 3(d).

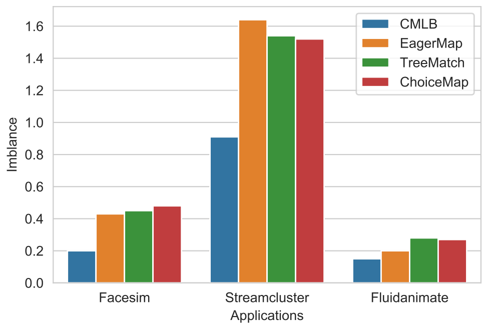
The above experiment shows that CMLB could balance the memory bandwidth between all the nodes and relieve the memory congestion problem which makes the average latency is significantly reduced, and finally make the execution time of rotor35-omp reduce. CMLB get an better performance on rotor35-omp than the other methods.



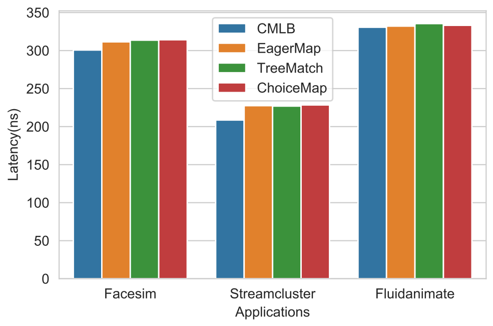
(a) Execution time improvement



(b) QPI improve



(c) Imbalance of memory bandwidth



(d) Average latency

Fig. 5: Experimental test on Parsec benchmarks

C. *Test on NPB benchmark*

In this section, SP, BT, LU and CG applications from NPB are used to verity the performance of CMLB from 4 aspects including execution time, QPI, the imbalance of memory bandwidth between nodes and the average latency. Besides, these applications are the Class B size and 32 threads, as shown in Fig. 4.

CMLB obtain an obvious optimization performance in SP, BT, LU, and CG applications. The highest execution time improvement reaches about 28.29%, and the highest QPI reduces is about 84.03%. Although CMLB is similar to EagerMap, TreeMatch and ChoiceMap in terms of QPI performance gains in SP, BT and LU applications, the imbalance of memory bandwidth is greatly reduced compared to other methods. BT has reduced significantly compared with EagerMap, TreeMatch and ChoiceMap by about 88%, 89% and 92%. What's more, CMLB also reduce the average memory latency. The average memory latency of SP has reduced significantly, which is 8.1%, 8.4% and 8.3% respectively compared with EagerMap, TreeMatch and ChoiceMap. The performance difference of the above metrics finally make the execution time improvement of the three applications SP, BT, and LU increase by 2.03%, 2.20% and 1.93% on average compared to EagerMap, TreeMatch and ChoiceMap respectively.

As for CG, CMLB has not achieved any improvement compared to other mapping methods. That is because the amount of memory access by each thread of CG is roughly the same, and the memory bandwidth is always in a relatively balanced state, which make the average latency is also the same as other mapping methods.

Therefore, CMLB has a better optimization performance on the SP, BT, LU applications compared to EagerMap, TreeMatch and ChoiceMap. CMLB could greatly reduce the imbalance of memory bandwidth between all nodes and also reduce the amount of remote communication. CMLB gain the best performance on execution time improvement which is also prove the theoretical evaluation in Section III-C.

D. *Test on Parsec benchmark*

As shown in Fig. 5, the metrics are the same as NPB benchmark. It is clear that CMLB has a certain optimization effect in Facesim, Streamcluster, and Fluidanimate compared to EagerMap, TreeMatch and ChoiceMap. The execution time improve is about 4.1%, and the QPI is reduced about 40.1% on average. What's more, CMLB has the better performance on Facesim, Streamcluster applications than others. As for Streamcluster, CMLB reduce the imbalance of memory bandwidth about 2 times comparing with other methods, and also reduce about 8.3% on average latency, so that make the execution time speed up about 1.3% than other three mapping algorithms.

For Fluidanimate application, it's similar to CG. CMLB obtains the similar performance on QPI as well as the imbalance of memory bandwidth. Therefore, the execution time of Fluidanimate has little difference among the these mapping methods.

Therefore, CMLB mapping method has a better optimization performance on the Facesim, Streancluster applications compared to EagerMap, TreeMatch and ChoiceMap, and has the similar performance on Fluidanimate.

V. Conclusion

In NUMA architecture systems, mapping parallel tasks to cores according to the access behavior plays an important role to optimize the parallel applications performance. Only maximizing the locality may cause memory congestion because of the imbalance on memory bandwidth between nodes. To solve this issue, a new mapping algorithm, CMLB, is proposed in this paper whose goal is to improve the locality of communication as well as avoide memory congestion problem. The main contributions are as follows. 1) an efficient memory access detecting and analyzing method is designed. Specially, based on the slide window, an access phases division algorithm is proposed which can effectively deal with the sequential relationship of continuous access. 2) a thread grouping algorithm is proposed which can ensure load balance when increasing locality. 3) to verify the effectiveness of the algorithm, rotor35-omp, NPB and Parse benchmark applications with different access behavior are test. Results show that CMLB could improve the locality of communication between threads and balance the memory bandwidth between nodes, greatly relieve the memory congestion problem and get the better performance than the state-of-the-art, including EagerMap, TreeMatch and ChoiceMap. For the future, we will extend CMLB to map tasks on a heterogeneous platform including both CPUs and GPUs.

Acknowledgment

We would like to thank the all reviewers, whose insightful comments greatly improved the quality of this paper. The work was funded by the National Key Research and Development Program of China (2016YFB0200902).

References

[1] M. A. Sasongko, M. Chabbi, P. Akhtar, and D. Unat, "Comdetective: a lightweight communication detection tool for threads," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2019, Denver, Colorado, USA, November 17-19, 2019*, M. Taufer, P. Balaji, and A. J. Peña, Eds. ACM, 2019, pp. 18:1–18:21.

[2] E. H. M. Cruz, M. Diener, L. L. Pilla, and P. O. A. Navaux, "Eagermap: A task mapping algorithm to improve communication and load balancing in clusters of multicore systems," *ACM Trans. Parallel Comput.*, vol. 5, no. 4, pp. 17:1–17:24, 2019.

[3] F. Gaud, B. Lepers, J. R. Funston, M. Dashti, A. Fedorova, V. Quéma, R. Lachaize, and M. Roth, "Challenges of memory management on modern NUMA systems," *Commun. ACM*, vol. 58, no. 12, pp. 59–66, 2015.

[4] M. Diener, E. H. M. Cruz, M. A. Z. Alves, P. O. A. Navaux, and I. Koren, "Affinity-based thread and data mapping in shared memory systems," *ACM Comput. Surv.*, vol. 49, no. 4, pp. 64:1–64:38, 2017.

[5] D. Molka, D. Hackenberg, and R. Schöne, "Main memory and cache performance of intel sandy bridge and AMD bulldozer," in *Proceedings of the workshop on Memory Systems Performance and Correctness, MSPC '14, Edinburgh, United Kingdom, June 13, 2014*, J. Singer, M. Kulkarni, and T. Harris, Eds. ACM, 2014, pp. 4:1–4:10.

[6] D.Ziakas, A.Baum, R.A.Maddox, and R.J.Safranek, "Intelquickpath interconnect architectural features supporting scalable system architectures," in *Proceedings of IEEE Symposium on High Performance Interconnects, Aug 2010*. IEEE, 2010, pp. 1–6.

[7] J. Li, X. Zhang, J. Zhou, X. Dong, and C. Zhang, "swHPFM: Refactoring and optimizing the structured grid fluid mechanical algorithm on the sunway taihulight supercomputer." *Appl. Sci.*, vol. 10, no. 1, pp. 72–93, 2020.

[8] J. Li, X. Zhang, L. Han, Z. Ji, X. Dong, and C. Hu, "OKCM: improving parallel task scheduling in high-performance computing systems using online learning," *J. Supercomput.*, vol. 77, no. 6, pp. 5960–5983, 2021.

[9] E. H. M. da Cruz, M. Diener, L. L. Pilla, and P. O. A. Navaux, "An efficient algorithm for communication-based task mapping," in *23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2015, Turku, Finland, March 4-6, 2015*. IEEE Computer Society, 2015, pp. 207–214.

[10] E. H. Cruz, M. Diener, M. A. Alves, and P. O. Navaux, "Dynamic thread mapping of shared memory applications by exploiting cache coherence protocols," *Journal of Parallel and Distributed Computing*, vol. 74, no. 3, pp. 2215–2228, 2014.

[11] M. Dashti, A. Fedorova, J. R. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quéma, and M. Roth, "Traffic management: a holistic approach to memory placement on NUMA systems," in *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, Houston, TX, USA, March 16-20, 2013*. ACM, 2013, pp. 381–394.

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

[13] K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, and Ü. V. Çatalyürek, "Parallel hypergraph partitioning for scientific computing," in *20th International Parallel and Distributed Processing Symposium (IPDPS 2006), Proceedings, 25-29 April 2006, Rhodes Island, Greece*. IEEE, 2006.

[14] F. Pellegrini, "Static mapping by dual recursive bipartitioning of process architecture graphs," in *Proceedings of IEEE Scalable High Performance Computing Conference*, 1994, pp. 486–493.

[15] E. Jeannot, G. Mercier, and F. Tessier, "Process placement in multicore clusters: Algorithmic issues and practical techniques," *IEEE Trans. Parallel Distributed Syst.*, vol. 25, no. 4, pp. 993–1002, 2014.

[16] P. N. Soomro, M. A. Sasongko, and D. Unat, "Bindme: A thread binding library with advanced mapping algorithms," *Concurr. Comput. Pract. Exp.*, vol. 30, no. 21, 2018.

[17] M. Diener, E. H. M. da Cruz, M. A. Z. Alves, and P. O. A. Navaux, "Communication in shared memory: Concepts, definitions, and efficient detection," in *24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, Heraklion, Crete, Greece, February 17-19, 2016*. IEEE Computer Society, 2016, pp. 151–158.

[18] V. J. Reddi, A. Settle, D. A. Connors, and R. S. Cohn, "PIN: a binary instrumentation tool for computer architecture research and education," in *Proceedings of the 2004 workshop on Computer architecture education - Held in conjunction with the 31st International Symposium on Computer Architecture, WCAE@ISCA 2004, Munich, Germany, June 19, 2004*. ACM, 2004, p. 22.

[19] D. L. Long and L. A. Clarke, "Task interaction graphs for concurrency analysis," in *Proceedings of the 11th International Conference on Software Engineering, Pittsburg, PA, USA, May 15-18, 1989*, L. E. Druffel, D. Fairley, and D. Bjørner, Eds. IEEE Computer Society / ACM Press, 1989, pp. 44–52.

[20] D. Rice, L. Biller, J. Glick, and C. Sandifer, "Intel microarchitecture codename nehalem performance monitoring unit programming guide," https://software.intel.com/sites/default/files/m/5/2/c/f/1/30320-Nehalem-PMU-Programming-Guide-Core.pdf, 2010, accessed June 16, 2021.

[21] Z. Li, Y. Zhao, R. Liu, and D. Pei, "Robust and rapid clustering of kpis for large-scale anomaly detection," in *26th IEEE/ACM International Symposium on Quality of Service, IWQoS 2018, Banff, AB, Canada, June 4-6, 2018*. IEEE, 2018, pp. 1–10.

[22] 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 *Proceedings of the 18th Euromicro Conference on Parallel, Distributed and Network-based Processing, PDP 2010, Pisa, Italy, February 17-19, 2010*. IEEE Computer Society, 2010, pp. 180–186.