CMLB: A Balanced Thread Mapping Method for Modern NUMA Systems

\*Note: Sub-titles are not captured in Xplore and should not be used

line 1: 1st Given Name Surname   
line 2: *dept. name of organization   
(of Affiliation)*  
line 3: *name of organization   
(of Affiliation)*line 4: City, Country  
line 5: email address or ORCID

line 1: 4th Given Name Surname  
line 2: *dept. name of organization*  
*(of Affiliation)*  
line 3: *name of organization   
(of Affiliation)*line 4: City, Country  
line 5: email address or ORCIDline 1: 2nd Given Name Surname  
line 2: *dept. name of organization   
(of Affiliation)*  
line 3: *name of organization   
(of Affiliation)*line 4: City, Country  
line 5: email address or ORCID

line 1: 5th Given Name Surname  
line 2: *dept. name of organization   
(of Affiliation)*  
line 3: *name of organization   
(of Affiliation)*line 4: City, Country  
line 5: email address or ORCIDline 1: 3rd Given Name Surname  
line 2: *dept. name of organization   
(of Affiliation)*  
line 3: *name of organization   
(of Affiliation)*line 4: City, Country  
line 5: email address or ORCID

line 1: 6th Given Name Surname  
line 2: *dept. name of organization   
(of Affiliation)*  
line 3: *name of organization   
(of Affiliation)*line 4: City, Country  
line 5: email address or ORCID

*Abstract*—In parallel applications, mapping parallel threads to cores according to the access behavior plays an important role to optimize the applications performance. When a parallel application runs on modern NUMA architecture, the efficiency of communication between threads and the memory bandwidth between nodes are not balanced, which indirectly increase the average latency and the execution time of the application. Previous work 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, we propose a thread mapping method that works on improving the locality of communication as well as avoiding memory congestion problem. Experiments 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.

Keywords—parallel applications, NUMA, shared memory , memory bandwidth, memory latency, thread mapping

# Introduction

Modern NUMA (non-uniform memory access) systems consist of numerous processor cores and complex hierarchy of memory. Within a NUMA multiprocessors computer, each processor consists of a group of processor cores, which is associated with one or more memory controllers and memory devices [1]. And the group of processor cores is referred to as a NUMA node [2], [3]. Although there are QPIs [4] (QuickPath Interconnect) between NUMA nodes, 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. In NUMA systems, when tasks communicate with each other through a shared cache memory or intra-chip interconnection, that is called local communication. And if tasks communicate with each other through QPI, which is called remote communication. 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 [9].

Mapping tasks of an application to processor cores according to the communication between tasks is called communication-aware task mapping [8]. Communication-aware task mapping use a communication matrix and an architecture graph to complete the whole mapping process, communication matrix which is symmetric represents the amount of communication between threads, and architecture graph represents the machine components topology include cores, cache memories, NUMA nodes [9]. The above mapping method map the tasks which communicate intensely to cores close together at the same NUMA node, it has a good work on improving the locality of communication. However, recent works [10], [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, we propose *CMLB*, a communication-aware and memory load balance task mapping method. *CMLB* is designed to work on improving the locality of communication as well as avoiding memory congestion problem. Due to a task can include a thread and a process, and *CMLB* focus on the threads mapping. *CMLB* can improve the performance of applications by gathering and analyzing the memory access behaviors and then using a thread grouping algorithm to map a group of threads to a NUMA node.

# Related Work

Previous studies propose several task mapping method considering the communication between tasks. In shared memory environments a communication-aware task mapping reduces execution time, cache misses and interconnection traffic [12]. In the context, mapping tasks that communicate to the same NUMA node can improve the performance of applications.

Several mapping methods have been proposed to reduce the cost of communication. Most 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. Jeannot et al. [15] proposed Treematch algorithm which generates all possible group of tasks, and it have a high complexity. To improve the complexity of the algorithm, Cruz et al. [16] proposed Eagermap which used a greedy policy to group tasks, and it have a lower complexity of O(N3) than Treematch. Soomro et al. [17] proposed Choicemap algorithm, it use a fair policy to pair threads and has a better performance of some applications than Eagermap. However, these methods that only consider the feature of communication between tasks may cause memory congestion.

The above of the mapping methods both need to detect the communicate behaviors between tasks. Eagermap and Choicemap use the Numalize [18] tool based on Intel Pin tools [19] to detect communications. And Choicemap also use CommDetective [20] which used the Linux Perf tools to detect communications, CommDetective has a lower overhead than Numalize.

# CMLB – Communication-Aware And Memory Load Balance Mapping

In this section, we describe CMLB in detail, which contains detecting and analyzing the memory access behaviors of threads, computing and grouping threads. At last, we use two metrics to evaluate CMLB theoretically.

## Detecting and Analyzing the Memory Access Behaviors of Threads

CMLB requires the memory access behaviors of threads, which is an important basis for the thread grouping algorithm to map the threads to cores. We use the CommDetective [20] tool to generate the communication matrix to improve the communication efficiency, and in order to solve the memory congestion problem, we propose the memory access load detection and use a vector to record the memory access load of each thread. By combining the communication matrix and memory access load vector, CMLB could improve the locality of the communication as well as make the memory access load balance on NUMA nodes.

### Detecting communication between threads: In parallel applications based on shared-memory such as OpenMP and Pthreads, data are exchanged and shared through shared memory between threads. We refer to the operation of different threads that read and write the same cache line in shared memory as inter-thread communication.

Communication behavior of an application can be represent as a communication graph or task integration graph [21]. 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 and has a zero diagonal. CMLB and other communication-based mapping methods analyze the matrix to group threads and map them to machine.

We use the CommDetective tool to detect the communication and generate the communication matrix. CommDetective is based on Linux Perf tools, and use the perf\_event\_open() system call to extract the memory access events from hardware performance monitoring unit (PMU). PMUs can be configured to trigger an overflow interrupt once a threshold number of events elapse. A PMU interrupt is be referred to a “sample”, and Intel’s Precise Event-Based Sampling (PEBS) [21] 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, we consider the two enents occur a communication, and update in the communication matrix.

In this paper, we set perf\_event\_open in CommDetective to perform sampling with sample frequency 1000 that means this tool will extract a sample from PMU every 1000 access events elapse, and set Commdetective to monitor two type of events for sampling: all loads micro operations and all stores micro operations.

### Analyzing the memory access load of each thread: Diffiered from communication matrix, access load vector represents the memory access load of each thread. If a thread access the memory many times in the whole execution, it will has a high value in the access load vector. Therefore 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, analyze the access table using the data analyze methods to genetate the access load vector.

We firstly select the events which access DRAM due to L3 cache miss when Perf extracts the samples of access events in CommDetective. That is because the access events which really access the DRAM could show the memory load features accurately, and the types of the events which selected are MEM\_LOAD\_UOPS\_LLC\_MISS\_RETIRED.LOCAL\_DRAM, LLC\_MISS\_RETIRED.REMOTE\_DRAM. For each event we extract the related information include access address, thread id and the timestamp of accessing, and then make up these pieces of information to a record which will be stored at an access table.

Secondly, we use the Pandas tool supported by python to analyze the access table. 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

 (1)

where *cycles* represents the old timestamp in CPU clock cycles, and *feq* represents the CPU clock frequency in khz. After the conversion is completed, a time slice needs to be set to merge the records whose timestamp interval is within the time slice, and we set the time slice to 1ms in this paper. The merged record includes the timestamp, the thread id list, and the number of the original records. The number of the original records indicates how many memory accesses are performed in a time slice, and the higher this value, the higher the memory access data traffic in this time slice, and the higher the memory load. Therefore, it is necessary to focus on extracting the memory access features of the merged records with a high number of original records, because these merged records will have a large impact on DRAM and are likely to cause memory congestion problem.

And than we can find the number of memory access changed regularly over time, as shown in Fig. 2. We tested four applications, and drew the timestamp, the amount of the memory access of each merged record for every application in pictures. It’s easy to see that the time interval which has the large amount of memory access is likely to cause memory congestion, so we should focus on extracting each thread access load feature in that time interval. And there are some phases where the amount of access first increases and then decreases over the whole execution , so we need to divide the execution into several phases and combine the access load features from all the phases.

We proposed an access phases division algorithm based on the slide window algorithm, as shown in Algorithm 1. Before dividing the access phases, we first process the time series of the access amount. When using Perf to extract the access events, there may have noise to influence the data of time series. Therefore, we should linear smooth the time series of the access amount. According to the method in Rocka [21], the outliers in the time series data are usually less than 5%, so we delete the data with the top 5% of the value in the series, and use linear interpolation to fill in and replace. After smoothing the data, we use Algorithm 1 to divide the access phases. As algorithm 1 shown, we input the time series of the access amount which are linear smoothed, and use the means of the last 5% of the value in the series as the low value of the access amount, which is shown from line 1 to line 2. And then, we set two pointers in line 3, and also set a minimum window width in line 4 which represents the minimum time interval of an access phase. After that, we move forward the right pointer, 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.

For practical considerations, there may have some small phases due to many continuous points with low value in the series, so minimum window width needs to be set to avoid existing too many small phases which will influence the result. In this paper, we set the minimum window width to 100, that means a phase has at least 100 time slices. By using Algorithm 1, we finally get all the access phases from an application. As shown in Figure 2, we tested four application in NAS Parallel Benchmark (NPB), and the area between the red lines represents an access phase.

Due to each access phase has its own access load feature, we should extract the access load feature of each phase and combine them together. The access load feature of each phase could represent by a vector which include the access load of each thread. In order to get the access load feature of a phase, we use the thread list in a merged record which represents the thread ids accessing the memory. And a phase includes at least 100 merged records, we merge all the thread list in a phase and count the access amount of each thread. By doing this, we get the access load feature of each phase.

Each phase has its own feature, so we need to combine all the phases together. Due to phases have the different access amount, and the phase which has the large access amount will have a significant impact on memory access and be likely to cause memory congestion, so we use the average of access amount of a time slice in a phase to represent the weight of this phase. And then we use the weight of each phase to combine them, as shown in Equation 2-3.

Equation 2 calculate the weight of a phase, where n represents the number of time slices in this phase and di represents the access amount of the i-th time slice, so w represents the average of access amount of a time slice in this phase. And Equation 3 calculate the final access load vector, where n represents the number of the phases, wi  represents the weight of the i-th phase, and pi represents the access load vector of the i-th phase.

## Computing and Grouping Threads

### Description of Thread Grouping Algorithm: According to section I, CMLB is designed to work on improving the locality of communication as well as avoiding memory congestion problem. Therefore, 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. And we design the thread grouping algorithm using a top to bottom mode.

We first determine the number of groups of all threads, and generate all the groups in a loop. 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 the Hwloc tool, we can get the machine components topology including the number of nodes, the number of cores and memory caches in each node. As Algorithm 2 shown, we input the communication matrix, access load vector and the configmap which record the machine topology, and then calculate the number of threads in each group and initialize variables such as the list of threads to be grouped and the list of grouping results from line 1 to line 5. From line 6 to line 8, generate each group and add the result of each group to the final result list. At last, return the result of threads grouping in line 10.

The function *GenerateOneGroup* which represents the process to generate a group of threads is described in Algorithm 3. According to Algorithm 2, we get the number of threads for a group, and then select a thread from the list of the remaining threads. And repeat the above step until the number of threads in the group reaches the upper limit. As Algorithm 3 shown, we first input the number of threads in each group and the list of the remaining threads to be grouped, then initialize the group list and update the remaining list to be grouped from line 1 to line 2. And select one of the remaining threads to be grouped to join and update the remaining list from line 3 to line 6. At last, return a group of threads and the remaining threads list in line 8.

In order to implement the function SelectOneThread in line 4 of Algorithm 3, we reference the greedy policy in EagerMap. According to the list of grouped threads *t* = [t1,t2,t3] and the communication matrix, we calculate all the amount of communications between all the remaining threads and *t*, and sort the communications from high to low. We select each thread tw from all the remaining threads and calculate the sum of communication with all the threads in t, as shown in Equation 4.

Where commmatrix represents the communication matrix, n represents the number of grouped threads in t, and C represents the sum of the communications between one remaining thread tw and all the threads in t. And then we can calculate all the sum of communications between the remaining threads and t, and sort them from high to low. At last, we get the communication rank list:[[C1, tw1], [c2,tw2], …]. When selecting a thread to join in group, we can visit the communication rank list and choose the first element where the thread has the largest communication amount with all the grouped threads, and then check that if joining the tw1 will destroy the memory access balance in this group according to access load vector. If it not, joining tw1 in the group will be safe, otherwise, choose the next thread tw2 in the communication rank list and repeat the above steps until select a thread to join in the group. The method of justify the balance of the access load in a group will described in Algorithm 5.

As the Algorithm 4 shown, we input the list of grouped threads and the list of remaining threads, and calculate the sum of the communication between each remaining thread and all the grouped threads and record in the rank list from line 1 to line 8. And then visit the rank list and choose the thread from line 11 to line 16. At last, return the grouped thread. Particularly, if all the threads in rank list destroy the balance of the group, the thread which is the first element in rank list will be selected.

According to the Algorithm 4, before we select thread to join the group, we should check it whether destroying the load balance of the group, and the function in line 11 of Algorithm 4 implement that. After determining the number of the groups in Top-Level Algorithm, we need to calculate the average memory load of each group according to the access load vector, as shown in Equation 5.

Where groups represents the number of the groups, n represents the length of the access load vector, and aml represents the average memory load of each group.

When selecting a thread from the rank list, the thread tw join the group at first, calculate the current load of this group and the *lastLoad* which is the difference between Aml and current load, and calculate *n* which represents the number of the remaining threads in this group at the same time. After that, we sort all the remaining threads according to the value in the AccVector from high to low, and calculate the maxLoad and minLoad which represent the sum of the access load of the top n threads and last n threads after sorted. If lastLoad is between minLoad and maxLoad, the thread tw will not destroy the balance in this goup. Otherwise, tw will destroy the balabce. That’s because we can not find n remaining threads to make the load reach lastLoad and the total load of this group will also not reach Aml.

As Algorithm 5 shown, we input the candidate thread, the list of grouped threads and the list of remaining threads. From line 4 to line 6, calculate the current access load of this group, and then calculate latsLoad and n from line 5 to line 6. From line 12 to line 18, calculate maxLoad, minLoad and check if lastLoad is between maxLoad and minLoad.

### An example of Grouping Threads:We now expalin the thread grouping algorithm with an example shwon in Figure 3. At the beginning, we input a communication matrix and a access load vector, and there are 8 threads in total. And all the threads will be grouped into 2 groups due to our machine has 2 NUMA nodes, so each group has 4 threads. As shown in Figure3b, the group 1 now has 2 threads(0 and 1), when selecting another thread to insert the group, we should justify the access load of the group. Firstly, we sort all the remaining threads from high to low according to the communication with thread 0 and thread 1, and record them in the rank list. And then, choose the first element(thread 2) from the rank list and try to insert in the group. According to the access load vector, we can calculate Aml is 18 ((1+2+ .. +8)/2) and lastLoad is 12 due to the current load including thread 0,1,2 is 6. And we calculate minLoad which the remaining threads can provide is 4 (provided by thread 3), and we can also calculate maxLoad is 8 (provided by thread 7). After that, we find lastLoad is larger than maxLoad, if thread 2 insert the group, the totol load of this group will larger than Aml and will make imbalance on two groups. Therefore, we choose next therad in rank list until find a proper thread. And finally we select the thread 6 that will not destory the balance of groups. At last, thread 0,1,6 and 7 are grouped in group 1, and last threads are grouped in group 2, and we can find the total access loads of 2 groups are the same.

## Theortical Evaluation of CMLB

In this section, we will test the theoretical performance of CMLB and compare with the state-of-art mapping algorithms include 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, we can measure remote communication and the load balance of nodes according to the communication matrix and access load vector. Due to both of the mapping methods which we compare with are the communication-based methods, we can calculate the remote communication using the communication matrix, as shown in Equation 6 and Equation 7.

In Equation 6, where n represent the number of nodes and we regard the threads in a node as a group. CommVal represents the function to calculate the communication between two groups and map represents the mapping result all the groups. Equation 7 is the process of calculate CommVal, where g1, g2 represent the two groups and n1, n2 represent the number of threads in the two groups. RemoteComm corresponds to the QPI in section 4, the lower this value, the higher the performance gains.

The access load balance of the nodes is also a criterion to judge the performance of mapping method. It is measured by Equation 8 and Equation 9,

In Equation 8, where Loadstd represents the standard deviation of the access load between groups, n represents the number of the groups. And Li is the access load in the ith group, L represents the means of the access load of all the groups. In Equation 9 is the process of calculate the Li, m represents the number of the threads in one group and Accvector is the access load vector. Loadstd corresponds to the imbalance in section 4, the lower this value, the higher the performance gains.

And then we test SP, BT and LU applications using the two metrics to compare the theoretical performance of CMLB with the other mapping methods. As shown in table1-3. From the tables we can see that, although CMLB is slightly higher than EagerMap, TreeMatch and ChoiceMap in RemoteComm of 3 applications, they are all at an order of magnitude. As for Loadstd of 3 applications, CMLB is significantly lower than the other methods. On the whole, CMLB can greatly reduce the difference in memory access load between nodes and maintain the memory access load balance of nodes as well as the amount of remote communication is not much different from the other methods. Therefore, CMLB is better than the other mapping methods from the theoretical evaluation.

In section 4, we will test these mapping methods from the experiment evaluation which will includes more applications and more metrics, and then reach the conclusion according to the evaluation results.

## Some Common Mistakes

* The word “data” is plural, not singular.
* The subscript for the permeability of vacuum **0, and other common scientific constants, is zero with subscript formatting, not a lowercase letter “o”.
* In American English, commas, semicolons, periods, question and exclamation marks are located within quotation marks only when a complete thought or name is cited, such as a title or full quotation. When quotation marks are used, instead of a bold or italic typeface, to highlight a word or phrase, punctuation should appear outside of the quotation marks. A parenthetical phrase or statement at the end of a sentence is punctuated outside of the closing
* parenthesis (like this). (A parenthetical sentence is punctuated within the pa

b

* A graph within a graph is an “inset”, not an “insert”. The word alternatively is preferred to the word “alternately” (unless you really mean something that alternates).
* Do not use the word “essentially” to mean “approximately” or “effectively”.
* In your paper title, if the words “that uses” can accurately replace the word “using”, capitalize the “u”; if not, keep using lower-cased.
* Be aware of the different meanings of the homophones “affect” and “effect”, “complement” and “compliment”, “discreet” and “discrete”, “principal” and “principle”.
* Do not confuse “imply” and “infer”.
* The prefix “non” is not a word; it should be joined to the word it modifies, usually without a hyphen.
* There is no period after the “et” in the Latin abbreviation “et al.”.
* The abbreviation “i.e.” means “that is”, and the abbreviation “e.g.” means “for example”.

An excellent style manual for science writers is [7].

# Experimental Evaluation of CMLB

In this section, we will test the performance of CMLB by using some applications.

## Experimental enviroment

In this paper, we used 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, the information of the machine is shown in table 4. The platform use hyper-threading technology, each core could run 2 threads at the same time.

We used rotor35-omp program, NPB benchmark and Parsec benchmark to test the performance of CMLB.

rotor35 is an example program of a compressor rotor square cavity flow model in our computational fluid dynamics (CFD) project. The rotor35 program is aimed at the 36-channel axial compressor rotor model, and sequentially executes the three modules of CFD engineering application: pre-processing, numerical solution, and post-processing [18]. The numerical solution process constructs the control equations shown in Equation 10 for the field physical quantities between discrete points of the fluid model.

Where Q is a conservation vector, Fc, Gc, and Hc are convection vectors along the three coordinate directions of x, y, and z respectively. Fv, Gv, and Hv are the viscosity vector along the three coordinate directions of x, y, and z respectively and I represents the source term. The calculation example solves the 6 physical quantities of the vector Q in the Equation 10.

Using the Full Multi-Grid (FMG) triple network loop, the residual limit and interpolation between the coarse grid, the medium grid and the fine grid are carried out according to the standard of the multi-grid method [19]. The process is shown in Figure 5. Until the residual error in the program converges or reaches the maximum number of iterative steps, the numerical solution part of the program ends.

The rotor35 program used in this paper is a modified version of the original MPI version. The original rotor program uses processes to calculate 36 channels in parallel, which means one channel is calculated by one process, and communication occurs between processes. Since the computer platform used in this paper is a shared memory environment, and only one MPI process is allowed to run. Therefore, the original rotor35 program needs to be modified into a single MPI process program that only calculates one channel. When the rotor35 is executed in a single process, it uses OpenMP multi-threading for parallelism, and the multi-process parallel MPI program of the rotor35 was modified to a multi-threaded parallel OpenMP program under a single process, which referred to as rotor35-omp program.

Except rotor35-omp program, we select several applications in NPB and Parsec benchmark. In NPB benchmark, SP, BT, LU and CG are chosen for testing, and Streamcluster, Facesim and Fluidanimate are chosen from Parsec.

## Test on rotor35-omp

We used rotor35-omp program to test the performance of CMLB, and select the 8 threads, 16 threads and 32 threads versions of rotor35-omp during the test. The data size of the rotor35 program with different thread versions is also different.

Firstly, we analyze the access behavior of rotor35, and get the time series of access amount as shown in Figure 5. We can see that there are 3 obvious access phases corresponding to the 3 stages in Figure 4. And then we use the grouping algorithm to divide threads into groups and map each group to a NUMA node. We execute rotor35-omp after mapping threads and test it with different threads versions as shown in Figure 6.

The optimization performance of CMLB is more obvious when the number of threads is larger. As shown in Figure 6a, rotor35-omp with 32 threads have an obvious decrement about 6.25% on execution time when comparing CMLB with the other mapping methods. For 8 and 16 threads versions of rotor35, CMLB has the similar execution time with others. This is because the data size of rotor35 increases as the number of threads increases, and the number of memory accesses of the program also increases, resulting in an increasing imbalance of communication between threads and memory bandwidth between nodes. We also tested the imbalance of bandwidth from all nodes, which is calculated by the variance of all the nodes memory bandwidth. As shown in Figure 5c, 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 CMLB has the lowest average latency of rotor35 as shown in Figure 5b.

The above experiment shows that CMLB can balance the memory bandwidth between all the nodes so that the average latency is significantly reduced, and ultimately make the execution time of rotor35 reduce. Therefore, CMLB get the best performance on rotor35-omp than the other methods.

## Test on NPB benchmark

#### In this section, we use NPB benchmark to test the performance of CMLB. We selected SP, BT, LU and CG applications from NPB and test the performance from 4 metrics including execution time, QPI valume, the imbalance of memory bandwidth between nodes and the average latency.

We tested the 4 applications with the Class B size and 32 threads. As shown in Figure 7.

CMLB has obvious optimization effects in SP, BT, LU, and CG programs compared to before mapping. The highest execution time improvement reaches about 28.29%, and the highest QPI reduces about 84.03%. In addition, through comparison with other mapping methods, for SP, BT, and LU applications, the CMLB is similar to Eagermap, TreeMatch and ChoiceMap in terms of QPI performance gains, and the imbalance of memory bandwidth is greatly reduced compared to other methods, and BT has reduced significantly in that, compared with Eagermap, TreeMatch and ChoiceMap by about 88%, 89% and 92%. And it also reduces 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.

For CG, the CMLB has not achieved any improvement in these metrics compared to other mapping methods. That is because the amount of memory access by each thread of CG is also 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 mapping method 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 the average latency when QPI improvement is similar than others. So that, CMLB gained the best performance on execution time improvement which is also prove the theoretical evaluation in section 3.3.

# Conclusions

In parallel applications, mapping parallel tasks to cores according to the access behavior plays an important role to optimize the applications performance. By gathering and extracting the features of access behavior, and grouping threads according to that, the applications performance will get improved.

In this paper, we proposed a new mapping method, CMLB. In contrast to previous work, it considers the access load of each thread, based on an analysis of the time series of access amount in an application. We performed experiments with rotor35 and several public applications with different access behavior. Results show that CMLB could balance the memory bandwidth between nodes and reduce the average latency, greatly relieve the memory congestion problem and get the better performance than the state-of-the-art.

For the future, we will extend CMLB to map tasks on a heterogeneous platform including both CPUs and GPUs.

##### Acknowledgment

The preferred spelling of the word “acknowledgment” in America is without an “e” after the “g”. Avoid the stilted expression “one of us (R. B. G.) thanks ...”. Instead, try “R. B. G. thanks...”. Put sponsor acknowledgments in the unnumbered footnote on the first page.

## D. Test on Parsec benchmark

We also used Parsec benchmark to test CMLB. And we selected Facesim, Streancluster and Fluidanimate applications with 32 threads from Parsec. As shown in Figure 6.

The test metrics are the same as NPB benchmark, CMLB has a certain optimization effect in Facesim, Streamcluster, and Fluidanimate compared to before mapping. The execution time improve is about 4.1%, and the QPI is reduced about 40.1%. When comparing with other methods, CMLB has the better performance on Facesim, Streamcluster applications than others. For Streamcluster, CMLB reduced the imbalance of memory bandwidth about 2 times comparing with other methods, and also reduced about 8.3% on average latency, so that make the execution time speed up about 1.3% than before mapping and others make the execution time higher than others about 1% on average.

For Fluidanimate application, it’s similar to CG. CMLB has the similar performance on QPI and the imbalance of memory bandwidth, so that the execution time of Fluidanimate has little difference among the 4 mapping methods.

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

##### References

1. 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, ser. MSPC ’14. New York, NY, U SA: ACM, 2014, pp. 4:1–4:10.
2. F. Gaud, B. Lepers, J. Funston, M. Dashti, A. Fedorova, V. Quéma, R. Lachaize, and M. Roth, “Challenges of Memory Management on Modern NUMA Systems,” Queue, vol. 13, no. 8, p. 70, 2015..
3. M.Diener,E.H.Cruz,M.A.Alves,P.O.Navaux,andI.Koren,“Affinity- based thread and data mapping in shared memory systems,” ACM Com- puting Surveys (CSUR), vol. 49, no. 4, p. 64, 2017.
4. D.Ziakas,A.Baum,R.A.Maddox,andR.J.Safranek,“IntelQuickPath Interconnect Architectural Features Supporting Scalable System Architec- tures,” in 2010 18th IEEE Symposium on High Performance Interconnects, Aug 2010, pp. 1–6.
5. R. Nicole, “Title of paper with only first word capitalized,” J. Name Stand. Abbrev., in press.
6. Y. Yorozu, M. Hirano, K. Oka, and Y. Tagawa, “Electron spectroscopy studies on magneto-optical media and plastic substrate interface,” IEEE Transl. J. Magn. Japan, vol. 2, pp. 740–741, August 1987 [Digests 9th Annual Conf. Magnetics Japan, p. 301, 1982].
7. M. Young, The Technical Writer’s Handbook. Mill Valley, CA: University Science, 1989.

[8] E. H. M. Cruz, M. Diener, M. A. Z. Alves, and P. O. A. Navaux, “Dynamic thread mapping of shared memory appli- cations by exploiting cache coherence protocols,” Journal of Parallel and Distributed Computing, vol. 74, no. 3, 2014.

[9] E. H. M. Cruz, M. Diener, LL. Pilla, P. O. A. Navaux, An efficient algorithm for communication-based task mapping. Paper presented at: 2015 23rd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP); 2015; Turku, Finland.

[10] F. Gaud, B. Lepers, J. Funston, M. Dashti, A. Fedorova, V. Quéma, R. Lachaize, and M. Roth, “Challenges of Memory Management on Modern NUMA Systems,” Queue, vol. 13, no. 8, p. 70, 2015.

[11] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers,

V. Quema, and M. Roth, “Traffic management: a holistic approach to memory placement on NUMA systems,” in ACM SIGPLAN Notices, vol. 48, no. 4. 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 & Fluids, vol. 80, pp. 88–93, Jul. 2013.

[13] K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bissel- ing, and U. V. Catalyurek, “Parallel hypergraph partitioning for scientific computing,” in IEEE International Parallel & Distributed Processing Symposium (IPDPS), 2006.

[14] F. Pellegrini, “Static Mapping by Dual Recursive Bipartition- ing of Process and Architecture Graphs,” in Scalable High- Performance Computing Conference (SHPCC), 1994.

[15] E. Jeannot, G. Mercier, and F. Tessier, “Process Placement in Multicore Clusters: Algorithmic Issues and Practical Tech- niques,” IEEE Transactions on Parallel and Distributed Sys- tems, vol. 25, no. 4, pp. 993–1002, Apr. 2014.

[16] E. H. M. Cruz, M. Diener, LL. Pilla, P. O. A. Navaux, EagerMap: A Task Mapping Algorithm to Impromove Communication and Load Balancing in Clusters of Multicore Systems. ACM Transactions on Parallel Computing, 2019, 5(4), pp. 1-24.

[17] P. N. Soomro, M. A. Sasongko, D. Unat. BindMe: A thread binding library with advanced mapping algorithms. Concurrency Computat Pract Exper (cpe), vol.30, no. 21, p.e4692, 2018.

[18] M. Diener, E. H. M. Cruz, LL. Pilla, P. O. A. Navaux, Communication in shared memory: Concepts, definitions, and efficient detection. Paper presented at: 2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP); 2016; Heraklion, Greece.

[19] V. J. Reddi, A. Settle, D. A. Connors, 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; 2004; Munich, Germany.

[20] M. A. Sasongko, M. Chabbi, P. Akhtar, and D. Unat. 2019. ComDetective: A Lightweight Communication Detection Tool for Threads . In Proceedings of ACM Supercomputing (SC’19). ACM, New York, NY, USA, 12 pages.

[21]

**IEEE conference templates contain guidance text for composing and formatting conference papers. Please ensure that all template text is removed from your conference paper prior to submission to the conference. Failure to remove template text from your paper may result in your paper not being published.**