LHMapping: A Dynamic Thread-to-core Mapping Method Using Loop-based Hierarchical Grouping Algorithm

Abstract: **Mapping optimization methods have often been used to obtain reasonable matching between parallel computing tasks and computing resources. However, conventional static mapping methods can only be used for specific applications and cannot be adapted for applications that involve high computational load, frequent memory access, and significant changes during runtime (e.g., computational fluid dynamics programs). Furthermore, in computers with non-uniform memory access architecture, the efficiency of multithread communication through shared memory is unbalanced, which degrades the performance of applications. In this regard, this paper proposes a dynamic thread-to-core mapping mechanism for multithreaded parallel applications and introduces key controls to implement the mechanism as a kernel module in Linux. A grouping algorithm referred to as the loop-based hierarchical mapping (LHMapping) algorithm, to ensure that the mapping is overall optimal, is proposed and implemented; its calculation accuracy is comparable to that of existing algorithms at a lower computational cost. A maximum performance gain of greater than 20% was achieved for the NAS Parallel Benchmark suite on our Intel 80-core parallel cluster.**

Keywords: **Multithreaded Parallel Applications; Communication in Shared Memory; Dynamic Mapping Optimization; Grouping Algorithm;**
Doc

1 Introduction

Highly parallel applications have attained the characteristics of large scale and strong scalability. The most representative applications comprise parallel computational fluid dynamics (CFD) programs. They are typically used to solve complicated hydrodynamic equations that simulate complex fluid systems with tens of billions of grids after domain discretization. Consequently, the computational load and parallel scale of these applications are extremely large.

In addition, with the introduction of high-performance computers, the requirement to match computing tasks with computing resources has become an important problem in high-performance computing. Mapping optimization has become the main method of solving this problem. Therefore, significant improvement in the performance of the programs, and even a reduced energy consumption, can be achieved by developing a

mapping mechanism for complex parallel applications such as large-scale CFD programs.

Furthermore, to improve the optimization effect, we can use not only message passing interface for parallel optimization between processes but also more fine-grained multithread parallel optimization approaches such as open multi-processing (OpenMP). In the OpenMP environment, data are exchanged and shared through shared memory between threads. We refer to the operation of different threads that read and write the same data in shared memory as inter-thread communication. In parallel computers with non-uniform memory access (NUMA) architecture, the use of this implicit communication method results in unbalanced communication overhead [1]. If two threads that need to communicate are distributed among central processing unit (CPU) cores sharing the same cache, they need to read or write data only on the shared cache to complete the communication, and memory access has little effect on the cache hit ratio. However, if two threads that need to communicate are on different processors, one of the threads must read the data in the local memory of the other thread. Thus, the two threads must communicate with each other through on-chip interconnects, and the communication overhead is much higher. Therefore, further study of mapping optimization methods operating at multithread level is necessary to effectively address the problem of communication imbalance between threads. Such an approach can lead to flexible and multi-level mapping optimization techniques for highly parallel applications [2].

Most conventional mapping methods are static, although they use a variety of performance indexes to compare all possible mapping choices comprehensively and have significant optimization effects for specific applications. However, static mapping can be applied only to a small number of cores, and the programs must be run multiple times in advance; thus, static mapping methods are less versatile, scalable, and portable. Therefore, the communication imbalance between threads can be eliminated and the computing tasks of the program can be matched with the computing and storage resources of the system by designing a runtime mechanism for dynamic mapping optimization of multithreaded parallel applications. This mechanism should flexibly adapt to the characteristics of programs whose computing and memory access change dynamically during runtime.

To this end, we developed and implemented a dynamic thread-to-core mapping optimization method combined with the platform architecture for large OpenMP multithreaded parallel applications.

2 Related Work

There have been many studies of mapping optimization. They can be classified into several categories according to the mapped objects: mapping of tasks (where jobs are

assigned to different computing nodes), mapping of application data (where data are distributed to the memory of different processors), mapping of processes (where processes are mapped to different processors or computing nodes), and mapping of threads (where threads are mapped to processor cores). They can also be classified according to the problems solved by mapping optimization: mapping to maintain load balance, to reduce competition between hardware resources, or to avoid unbalanced communication overhead.

A framework for thread-to-core mapping, Autopin [3], selects the thread mapping that presents the highest instructions per cycle (IPC) as the optimal choice from many possible mapping choices. However, it is evidently not comprehensive to evaluate the runtime performance using only IPC as the metric. Thus, this method is only effective for applications in Autopin, but not for other applications. BlackBox [4] uses a similar concept; however, it is intended mainly for parallel computers with a symmetric multiprocessing architecture, which use simultaneous multithreading (SMT) technology. Furthermore, it can be used only for two-threaded producer–consumer programs. Owing to the unique platform architecture, competition for hardware resources will still occur if threads are mapped to the same physical core, despite the low communication overhead. In addition, Wang et al. [5] used machine learning methods to predict the optimal number of threads and optimal thread scheduling strategy for programs running on multicore processors, and they implemented mapping using a portable automation method based on compiler platforms.

Researchers subsequently began studying dynamic mapping methods. Suleman et al. [6] designed a feedback-adjustable dynamic thread number mapping framework by extracting runtime information. They also designed mechanisms for synchronization-aware threading and bandwidth-aware threading to overcome the limits on data synchronization and off-chip bandwidth; finally, they predicted the optimal number of threads. Ju et al. [7] changed the kernel control register in Linux and used the read performance-monitoring counter (rdpmc) instruction to read the system performance information in user mode. In addition, they implemented a thread mapping mechanism based on data locality analysis of Intel MIC many-core systems. Acosta et al. [7] proposed a thread allocation algorithm for chip multiprocessors (CMPs) with SMT technology. The algorithm considers not only the workload characteristics but also the underlying instruction fetching strategy. Ştirb et al [9] focused on NUMA and proposed two algorithms that achieved thread classification and thread type-aware mapping to improve runtime performance and reduce energy consumption. Cruz et al. [9] used the cache coherence protocol to accurately detect and count communication events between threads by setting hardware counters on the private L1 or L2 cache of each core. However, users cannot read the contents of a specific cache line, and the consistency messages are passed directly to the processor; thus, a cache simulator was needed for their experimental test.

As for threads grouping method, Osiakwan and Akl [11] adopted the classical Edmonds method in their graph division algorithm; however, the algorithm can be used

only when both the number of threads and the number of processing units are multiples of two. In addition, an open source parallel software package was developed by Sandia Nation Laboratories [12]. They used hypergraphs to model the communication and then perform the division. The communication modeling is more accurate than that offered by other packages; however, the tedious calculation and modeling make it unsuitable for dynamic mapping methods, and the optimization effect of the parallel hypergraph partitioning library is not notably better than that of the parallel sparse matrix-vector multiplication algorithm. Another open source grouping algorithm is Scotch [13], which uses a bipartition recursive strategy based on a heuristic method to group threads layer by layer. Its calculation process still has high latency; however, the concept of hierarchical grouping corresponds exactly to the hierarchical distribution of hardware objects in parallel computers. Therefore, this method is valuable for mapping optimization.

3 Dynamic Thread Mapping Mechanism

In thread mapping, threads are mapped to processing units to optimize the performance of a multithreaded parallel program according to the running characteristics and computer architecture. The performance can be optimized through various approaches such as balancing the load, reducing competition for hardware resources, or reducing the communication overhead.

As mentioned in Section 1, our mechanism is used to solve the problem of unbalanced communication between threads in a shared memory communication environment. Therefore, threads are mapped so that different threads that frequently communicate with each other are mapped to CPU cores that are physically close to each other and can share the same memory. Here, we define a communication event between two threads as a process in which two threads sequentially access the same data in the memory when the program is running. The amount of communication between two threads increases with the number of communication events between them.

Based on our definition of thread mapping, we consider three problems in our study on the dynamic mapping mechanism: detection and counting of communication events between threads in runtime, allocation of threads to specific CPU cores according to the amount of communication, and scheduling and migration of threads in the system. Subsequently, we design and implement a mechanism that combines the solutions to these problems. In this section, we describe each of these aspects separately.

3.1 Overall Flow of the Mapping Mechanism

We designed and implemented three modules that solve the aforementioned three problems: the communication detection module, grouping computation module, and

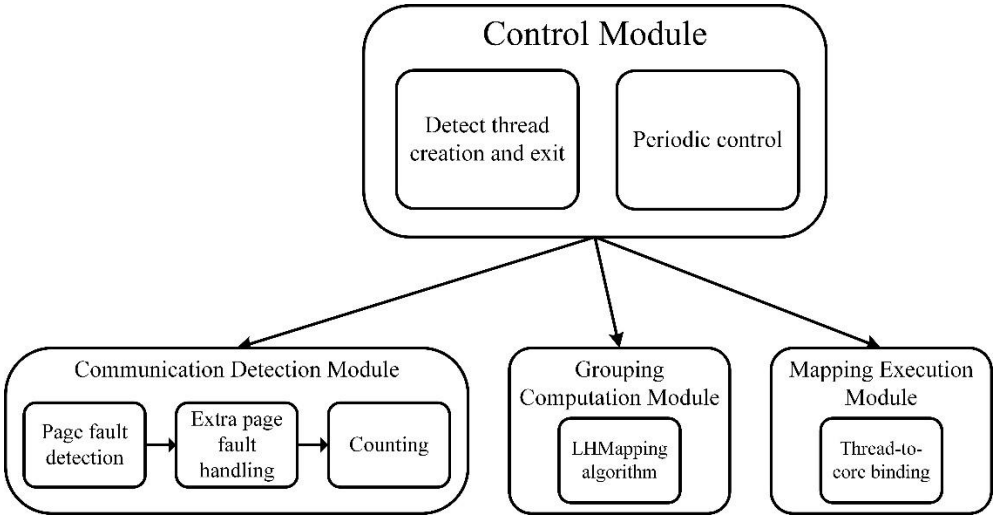


Fig. 1 Modules and flow chart of the dynamic thread-to-core mapping mechanism

mapping execution module. Together with a control module, they form a complete mapping optimization mechanism.

The modules and a flow chart of our dynamic thread-to-core mapping optimization mechanism are shown in Fig. 1 Modules and flow chart of the dynamic thread-to-core mapping mechanismand Fig. 2 Flow of the dynamic thread-to-core mapping mechanismThe processes are described in the following paragraphs.

When the program starts, we monitor thread creation in real time. When the program starts to run in a multithreaded environment, our mapping mechanism begins to continuously perform step 1, while periodically repeating steps 2 and 3 at certain intervals. This continues until all threads are observed to exit, at which time the program terminates or transitions to a single-threaded state. If it is an OpenMP program, it executes in the Fork-Join mode, and our mapping mechanism operates only in the multithreaded state. Steps 1–3 are described as follows:

1) Detection and counting of communication between threads in multithreaded state. We use the kMAF mechanism [14], which is based on page faults, for communication detection. We continuously obtain the information for each page fault on each thread in runtime. We maintain the communication matrix $A \in R^{n \times n}$, where n is the total number of threads, and the matrix elements $a(i, j)$ $(i \in n, j \in n)$ are communication events between threads *i* and *j*. We regard the page faults caused by different threads on the same memory page for a period of time as communication events between these threads and constantly update the matrix *A*. The subsequent execution of grouping and mapping is based on the values in the communication matrix. The implementation of communication detection and counting is described in detail in Section 3.2.

2) Thread grouping. The loop-based hierarchical mapping (LHMapping) algorithm designed and implemented in this study is used to group all threads in applications according to the values in the communication matrix and the hardware architecture. The grouping result determines the CPU core to which each thread is mapped. The algorithm inputs the communication matrix *A* and hardware architecture information of the system on which the program is running (obtained by hwloc [15]) and outputs a

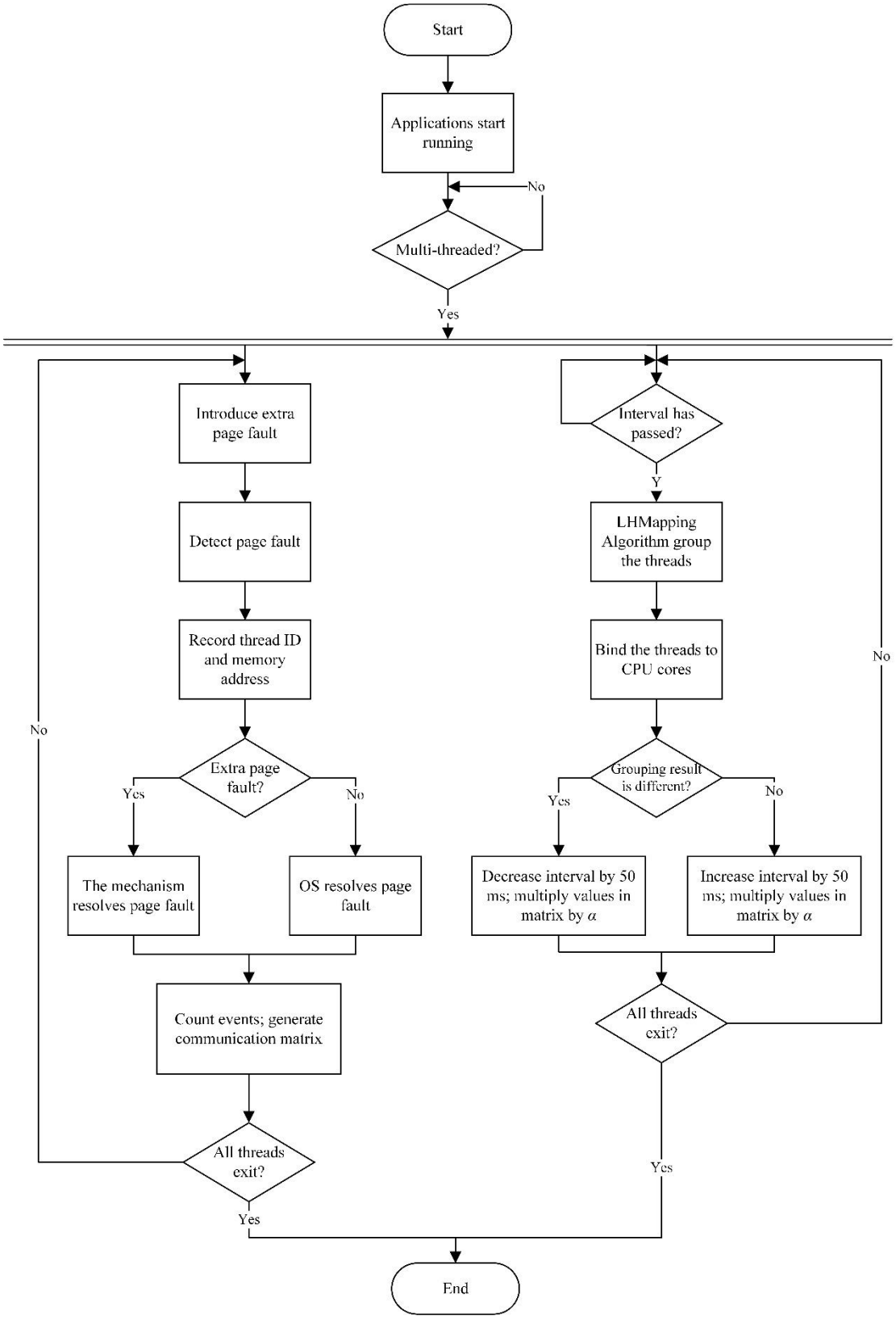


Fig. 2 Flow of the dynamic thread-to-core mapping mechanism

data structure representing the mapping of threads to processing units. The number of groups and number of elements per group are also determined according to the hardware architecture. The LHMapping algorithm is described in detail in Section 3.3.

3) Performing the mapping. Because the grouping procedure in step 2 yields a one-

to-one correspondence between threads and CPU cores, we can call the Linux kernel function *sched_setaffinity ()* [or the function *pthread_setaffinity_np ()* provided by the Linux Pthreads library] directly to set the CPU affinity of the threads. Thus, threads are temporarily bound to their respective CPU cores. The operating system does not migrate threads in the program before the next grouping occurs and the mapping result changes. The mapping execution method is described in detail in Section 3.4.

3.2 Communication Detection

We use a communication detection and counting method based on page faults. The details are presented in this subsection.

First, the Linux kernel provides groups of Kprobe functions, which can detect page faults in the system whenever our mechanism is applied, and the program is running. The dynamic mapping mechanism needs to record the thread ID and the physical address of the memory page in each detected page fault.

In addition, the mechanism introduces additional page faults. A page table entry in the application process page table is periodically changed. If the present bit of a page table entry is 1, it is changed to 0 to invalidate the memory page corresponding to the page table entry. As a result, if a thread needs to access this page the next time, it generates a page fault interruption. The introduction of additional page faults causes more overhead for processing interruptions in runtime; however, more cases of different threads accessing the same memory data can be observed, and the accuracy of communication detection increases.

Second, we define and initialize a hash table before counting the communication events. Each hash table entry corresponds to a memory block that is divided to obtain a certain granularity, and the division-remainder method is used as the hash function. Each hash table entry is a queue, and each entry in the queue records the thread IDs of page faults on the same memory block. The size of these queues can be a variable that controls the accuracy of communication detection. In our mechanism, the queue size is set to 4. When a new thread ID enters the queue, this thread and the other threads still in the queue are counted as communication events. We continuously record the information and update the communication matrix.

There are two types of page faults. One type is generated by the operating system (OS) itself; the other is introduced by our detection mechanism. The interrupt recovery process of the former is completed by the OS interrupt processing program; that of the latter is completed by the detection mechanism itself. The operation is as follows: the data are passed to the thread that needs to read them, and the page table entry is restored; after the interruptions are processed, applications can continue to run; and the counting of communication events and applications can be executed in parallel.

3.3 LHMapping Algorithm

3.3.1 Description of LHMapping

The algorithm provided by the Scotch mapping library has high complexity, which is not suitable for the dynamic mapping mechanism proposed in our study. Therefore, we propose the LHMapping algorithm, which has the same accuracy as the Scotch mapping library and a lower computational overhead. The LHMapping algorithm uses the multi-layer grouping method of the Scotch mapping library. We use the greedy strategy for grouping. The algorithm inputs the communication matrix data and the topology tree of the hardware objects in the system and outputs a multidimensional array representing the grouping result.

The LHMapping algorithm comprises the following:

1) The *Top-Level* algorithm represents the overall execution flow of the LHMapping algorithm.

First, we define the variables *nlevels* and *level*, which represent the total number of levels in all groups and the current level, respectively; both variables are initialized to

Algorithm 1: *Top-Level* Algorithm

Input: *CommMatrixInit[][], nThreads*
Output: *map[]*
Local Variables: *nElements, i, nGroups, rootGroup [], CommMatrix[][], groups[], pregroups[]*
Global Variables: *HardwareTopoRoot, nlevels, nObjects[]*

```
1  begin
2      for i = 0 ; i < nThreads ; i = i + 1 do
3          groups[i].nElements = 0;
4      nElements = nThreads;
5      CommMatrix = CommMatrixInit;
6      for i = 1 ; i <= nlevels ; i = i + 1 do
7          pregroups = groups;
8          [nGroups, groups[i]] = GenerateOneLevel(CommMatrix, nElements,
                                    nObjects[i], i, pregroups);
9          if i < nlevels then
10             CommMatrix = UpdateMatrix(CommMatrix, groups, nGroups);
11         nElements = nGroups;
12     rootGroup.nElements = nElements;
13     for i = 1 ; i < nElements ; i = i + 1 do
14         rootGroup.element[i] = groups[nlevels][i];
15     GroupsToTopology(HardwareTopoRoot, rootGroup, map);
16     return map;
17 end
```

Algorithm 2: *GenerateOneLevel*

Input: *CommMatrix[][], nElements, nobject, level, pregroups[]*
Output: *nGroups, groups[level]*
Local Variables: *chosen[], elpergroup, nel, leftover, newgroup, i, j*

```
1   begin
2     if nElements > nobject then
3       nGroups = nobject;
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    j = 0;
11    for i = 0 ; i < nElements ; i = i + nel do
12      nel = elpergroup;
13      if leftover > 0 then
14          nel = nel + 1;
15          leftover = leftover − 1;
16      newgroup = GenerateGroup(CommMatrix, nElements, nel, chosen,
                               pregroups);
17      newgroup.elements = nel;
18      newgroup.id = j;
19      groups[level][j] = newgroup;
20      j = j + 1;
21    return [nGroups, groups[level]];
22  end
```

0. We then define and initialize two arrays, *groups[]* and *pregroups[]*. *Groups[]* stores all the grouping results, where *groups[i]* is the grouping result of level *i*. *Pregroups[]* stores the grouping results of the previous level. It is a value in *groups [i]* (0 <= *i* <= *nlevels*). These variables are all initialized to 0, as shown in lines 3 and 7 of the algorithm.

We then call *GenerateOneLevel ()* to the groups starting from the bottom level (that is, we first divide all the threads into many small thread groups according to the number of threads that can be shared by hardware objects). We must call *GenerateOneLevel ()* at each level. If the current grouping level is not the top one, *UpdateMatrix ()* is called to update the communication matrix in preparation for the grouping of the next level, as shown in lines 8–10 of the algorithm, and update the variables *pregroups[]* and *nElements*, which are passed as actual parameters to the next *GenerateOneLevel ()* function. After the grouping of the top layer is completed, *GroupsToTopology ()* is

Algorithm 3: ***GenerateGroup***

Input: *CommMatrix[][], nElements, nel, chosen[], pregroups[]*

Output: *newgroup*

Local Variables: *i, j, m,* W_j, W_{max}

```
1  begin
2      for i = 0 ; i < nel; i = i + 1 do
3          W_max = −1;
4          for j = 0; j < nElements; j = j + 1 do
5              if chosen[j] = 0 then
6                  W_j = 0;
7                  for k = 0; k < i; k = k + 1 do
8                      W_j = W_j + CommMatrix[j][newgroup.element[k]];
9                  if W_j > W_max then
10                     W_max = W_j;
11                     m = j;
12         chosen[m] = 1;
13         newgroup.element[i] = pregroups[m];
14     return newgroup;
15 end
```

called to map the array *groups[]* to specific hardware objects in the topology tree, and the mapping results are stored in the variable *map[]*. The top-level algorithm returns the variable *map[]*, which marks the end of LHMapping algorithm.

2) The *GenerateOneLevel* algorithm calculates the grouping on each level.

Before grouping at the current level, we calculate the number of groups, *nGroups*, and the number of elements, *nElements*, allocated to each group. Generally, *nGroups* is equal to the number of hardware objects that can be matched in the current layer. However, if *nElements* is less than the number of hardware objects, *nGroups* is equal to *nElements*, as shown in lines 2–5 of the algorithm. If elements remain, they are distributed among the last few groups, as shown in lines 6 and 7. In addition, we define an integer array variable *chosen[]*, which is a flag array that indicates whether the traversed element has been grouped. All the items in *chosen[]* are initialized to 0.

GenerateGroup () is called to generate each group cyclically. We first handle the leftovers. The resulting group is stored in the structure variable *newgroup*. The structure contains the group ID, total number of elements in the group, and *element[]*, which stores the ID of every element in the group. After a new group is generated, we save the *newgroup* to *groups[level]*. The GenerateOneLevel algorithm returns a tuple *[nGroups, groups[level]]*.

Algorithm 4: ***UpdateMatrix***

Input: *CommMatrix[][], groups[level], nGroups*
Output: *newCommMatrix[][]*
Local Variables: *i, j, k, t, 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[level][i]; k = k + 1 do
6                  for t = 0; t < groups[level][j]; t = t + 1 do
                       W = W +
7                      CommMatrix[groups[level][i].element[k].id]
                       [groups[level][j].element[t].id];
8              newCommMatrix[i][j] = W;
9              newCommMatrix[j][i] = W;
10     return newCommMatrix;
11 end
```

Algorithm 5: ***GroupsToTopology***

Input: *HardwareRoot, RootGroup, map*
Local Variable: *i*

```
1  begin
2      if HardwareRoot.type = CPUCore then
3          for i = 0; i < rootGroup.nElements; i = i + 1 do
4              map[rootGroup.element[i].id] = HardwareRoot.id;
5      else if HardwareRoot.shares > 1 then
6          for i = 0; i < rootGroup.nElements; i = i + 1 do
7              GroupsToTopology(HardwareRoot.linked[i],
               RootGroup.element[i], map);
8      else
9          GroupsToTopology(HardwareRoot.linked[0], RootGroup, map);
10 end
```

3) The *GenerateGroup* algorithm generates each group.

We select each element in the group using a loop. In each round of selection, we first define a local variable W_{max} (the maximum number of communication events between the element to be traversed and the elements in the group) and initialize it to 0 or −1. The greedy strategy is used to select all the elements in one group (lines 4–11 of the algorithm). If *chosen[j]* = 1 during the traversal, it indicates that the element has been allocated, and the element is skipped; otherwise, we calculate the communication W_j between element *j* and the existing elements in the group. If W_j is greater than W_{max}, we let $W_{max} = W_j$, record the current *j* with W_{max}, and let $m = j$. After one traversal, there is

always an element with the largest W_m; we select element *m* into the group, and store it in *newgroup.element[i]*. Because all elements at this level are arranged according to the sequence in *pregroups[]*, *m* denotes the thread or thread group represented by *pregroups[m]*. The *GenerateGroup* algorithm returns the grouping result *newgroup*.

4) The *UpdateMatrix* algorithm is a subfunction called by the *Top-Level* algorithm, which modifies the communication matrix after grouping is completed. We transform the communication volume between threads or smaller thread groups into the communication volume between larger thread groups.

We use a simple algebraic summation method to update the communication matrix. For example, the algorithm divides threads 0 and 1 into thread group 0, and threads 2 and 5 into thread group 1. The communication between group 0 and 1 is the sum of the original communication of thread 0 and threads 2 and 5, plus the sum of the communication of thread 1 and threads 2 and 5. We implement this summation using a two-layer loop, and the *UpdateMatrix* algorithm returns a new matrix, *newCommMatrix[][]*.

5) The *GroupsToTopology* algorithm maps the tree of the grouping results to the topology tree of the system hardware objects.

We implement a recursive method that starts from the root node and traverses down the hardware objects. If we traverse to the CPU core (the bottom layer), our recursion reaches the end. If we traverse to an intermediate layer (other sharable hardware objects), *GroupsToTopology ()* is called again. If the hardware object is neither a CPU core nor a sharable object, mapping is not considered. The *GroupsToTopology* algorithm returns *map[]*.

3.3.2 Complexity Analysis of LHMapping

1) Time Complexity

The LHMapping algorithm adds overhead mainly while grouping the threads, updating the matrix, and mapping the topology tree. The algorithm uses a hierarchical grouping method. The *Top-Level* algorithm calls *GenerateOneLevel()* to generate the groups on every level, and in each level we call *GenerateGroup ()* to generate a group cyclically. We assume that *N* is the total number of elements to be grouped on a certain level, *G* is the total number of elements in each group, and *n* is the total number of threads to be grouped in parallel programs. In the function *GenerateGroup ()*, whenever we select an element to enter the group, it must traverse all the other elements and calculate the communication between the elements in the group and the currently traversed element. Therefore, the time complexity of the function *GenerateGroup ()* can be expressed as

$$\sum_{i=1}^{G}\sum_{j=1}^{N} i = N \cdot \sum_{i=1}^{G} i = N \cdot \frac{G(G+1)}{2} \leq N \cdot G^2 \quad (1)$$

Thus, the time complexity of the function *GenerateOneLevel* is

$$\sum_{i=1}^{N/G} GenerateGroup \leq \sum_{i=1}^{N/G} N \cdot G^2 \leq \frac{N}{G} \cdot N \cdot G^2 \leq N^3 \quad (2)$$

The time complexity of *UpdateMatrix ()*, which has two loops, is $O(N^2)$. We denote

the total number of levels as L. According to the execution flow of the *Top-Level* algorithm, the time complexity of the grouping computation and matrix update is

$$\sum_{i=1}^{L} GenerateOneLevel + \sum_{i=1}^{L-1} UpdateMatrix$$
$$\leq \sum_{i=1}^{L} (GenerateOneLevel + UpdateMatrix)$$
$$= \sum_{i=1}^{L} (O(N_i^3) + O(N_i^2)) \quad (3)$$

In our overall grouping process, the worst time complexity is that at which the hardware objects at each level can share only two units of the upper level. Thus, we can regard the grouping topology tree as a binary tree, i.e., $N_i = \frac{N_{i-1}}{2}$. Therefore, Equation (3) can be rewritten as

$$\sum_{i=1}^{L} \left(O(N_i^3) + O(N_i^2)\right)$$
$$= \sum_{i=1}^{L} \left(O\left(\left(\frac{n}{2^{i-1}}\right)^3\right) + O\left(\left(\frac{n}{2^{i-1}}\right)^2\right)\right) < 2\left(O(n^3) + O(n^2)\right) \quad (4)$$

The function *GroupToTopology ()* is a recursive depth-first traversal method; therefore, the complexity can be expressed as $O(V + E)$, where V is the vertex of the graph, and E is the edge. In the hardware object topology tree structure, $E = V - 1$, and the total complexity is $O(V)$. Because the number of elements that can be shared during grouping must be greater than 1, the number of nodes at other levels must be less than or equal to the number of the lowest nodes, n. Thus $n \geq V/2$, and n and V have the same magnitude. The complexity of *GroupToTopology ()* can finally be expressed as $O(n)$.

In summary, the time complexity of the LHMapping algorithm is

$$\sum_{i=1}^{L} GenerateOneLevel + \sum_{i=1}^{L-1} UpdateMatrix + GroupToTopology$$
$$\leq 2\left(O(n^3) + O(n^2)\right) + O(n) = O(n^3) \quad (5)$$

2) Space Complexity

In our algorithm, array *groups[]*, in which the grouping results are saved, is a two-dimensional array. The first dimension is the level of grouping, and the total number is *levels*; the second dimension is the number of groups at each level, and the total number of elements can be expressed as $\sum_{i=1}^{levels} nGroups_i$. Because the hardware objects are all shareable, the worst space complexity is that at which each group has only two elements. Therefore, the total space occupied by all groups can be expressed as

$$2\sum_{i=1}^{levels-1} \frac{n}{2^i} = \sum_{i=0}^{levels-2} \frac{n}{2^i} \leq 2n \quad (6)$$

where n is the total number of threads. Therefore, the overall space complexity of all variables representing the grouping result is $O(n)$.

In *UpdateMatrix ()*, the worst case of the space complexity can be expressed as

$$\sum_{i=0}^{levels-1} O((\frac{n}{2^i})^2) = O(n^2) \quad (7)$$

In summary, the space complexity of the LHMapping algorithm is $O(n^2) + O(n) = O(n^2)$.

3.4 Other Controls

3.4.1 Controls on Communication Matrix

To improve the overall optimization effect, we must update the matrix *A* after each grouping computation. All the elements in the matrix, *a*(*i*,*j*) (*i*$\in n, j \in n$), are multiplied by a reduction coefficient, α ($0.5 < \alpha < 1$). Because the communication matrix is updated throughout the life cycle of the threads, this process reduces the impact of the amount of earlier communication on the subsequent matrix data, and reduces the impact on the accuracy of grouping computation.

To facilitate the calculations, we temporarily set $\alpha = 0.75$ to avoid additional overhead. Hence, we let

$$a_{new}[i][j] = a_{old}[i][j] - (a_{old}[i][j] \gg 2) \quad (8)$$

3.4.2 Periodic Controls

Our dynamic thread-to-core mapping mechanism uses a periodic method to sequentially execute the grouping computation and mapping execution modules. Therefore, a time interval is set between two adjacent grouping computation and mapping execution processes. The *interval* variable must have a reasonable range to maximize the optimization effect and minimize the occurrence of excessive overhead.

We temporarily set *interval* to 200 ms. For two adjacent grouping computations, if the result of the second is the same as that of the first or differs only slightly, the running characteristics of the programs and the thread communication pattern do not change significantly. Therefore, we can consider increasing *interval* by 50 ms. Conversely, if the communication pattern changes significantly and must be adjusted, *interval* is decreased by 50 ms.

When programs are running in multiple threads, *interval* is always greater than 50 ms and less than 1 s.

3.4.3 Mapping Execution Module

We map threads to CPU cores using the CPU affinity setting [16]. The steps involved in the mapping execution module are as follows:

1) Input the grouping computation result *map*, which is a topology tree. Initialize the variable *i* to 0.

2) Access the *i*th leaf node at the bottom layer of *map* and store the information for the thread ID and the corresponding CPU core ID. Select thread *tid [i]*, where the corresponding CPU core ID is *cpu [i]*.

3) Call void CPU_ZERO (cpu_set_t * *mask*), which is defined in the Linux kernel, to clear every bit in the mask.

4) Call void CPU_SET (int *cpu [i]*, cpu_set_t * *mask*), which is defined in the Linux kernel, where the *cpu [i]* bit of the mask is set to 1.

5) Call int pthread_setaffinity_np (pthread_t *tid [i]*, size_t *cpusetsize*, const cpu_set_t * *mask*) from the Linux multithread interface library to limit thread *tid [i]* to run on the CPU that was set in steps 3 and 4. Alternatively, call the process CPU affinity setting function, int sched_setaffinity (pid_t *pid*, size_t *cpusetsize*, cpu_set_t * *mask*), to make the same settings. Here pid uses *tid [i]*. So for the binding for thread *tid [i]*is completed.

6) Let $i = i + 1$. Repeat steps 2–5 until all the threads are completely bound and mapping execution is completed.

3.4.4 Implementation of the Mapping Mechanism

Our mapping mechanism is required to modify the Linux interruption handling routine during execution and call the CPU affinity function provided by the Linux kernel. Therefore, we cannot simply optimize the mapping in user mode.

We borrowed the kernel module mechanism provided by the Linux system, implemented our mechanism as a kernel module, and inserted it into the kernel. If a multithreaded application is running in a system that has the kernel module, our mechanism is performed.

We performed our experiment on different platforms; therefore, we must ensure that all of the operating systems have the same kernel version. In this study, kernel version 3.10.0 was used in the experiment.

4 Evaluations

We tested the performance of the proposed LHMapping method and the overall mapping mechanism on a multicore processor cluster with NUMA architecture.

4.1 Experimental Environment

We performed our experiments on an 80-core parallel computer (referred to herein as C80) consisting of an Intel Xeon E7-8850 processor and on a virtual machine built on an 8-core Intel Core i7-6700 computer (referred to herein as PC). The parameters of the experimental platform are shown in **TABLE 1** Overview of the Two Platforms Used in the Evaluation.

The C80 machine is a typical NUMA computer. It contains two NUMA nodes; each node contains four sockets, and each socket has 10 Intel Xeon E7-8850 processor cores. Each processor core has a 256 KB private L1 instruction and data cache and an 8 MB private L2 cache; 10 CPU cores share an L3 cache. Simultaneous multithreading

technology is not used in this system.

TABLE 1 Overview of the Two Platforms Used in the Evaluation

Name	Property	Parameters
C80	Architecture	2 nodes, 4 sockets/node, 10 processors/socket
	Processor	Intel Xeon E7-8850, 2.0 GHz
	Cache	10 × (256 KB + 256 KB) L1, 10 × 8 MB L2, 16 MB L3
	Memory	128 GB DDR3, page size 4 KB
PC	Architecture	2 nodes, 4 processors/node
	Processor	Intel Core i7-6700, 3.4 GHz
	Cache	4 × (16 KB + 16 KB) L1, 4 × 256 KB L2, 4 MB L3
	Memory	8 GB DDR4, page size 4 KB

C80 can run parallel applications with up to 80 threads, whereas the PC can run only 8-thread programs. The CentOS and Ubuntu operating systems are installed on C80 and PC, respectively, and their kernel version are 3.10.0.

We used the OpenMP implementation [17] of the NAS parallel benchmark (NPB) [18], which includes eight applications with communication characteristics: integer sort (IS), embarrassingly parallel (EP), Fourier transform (FT), multi-grid (MG), conjugate gradient (CG), lower-upper Gauss–Seidel solver (LU), scalar penta-diagonal solver (SP), and block tri-diagonal solver (BT) benchmarks.

4.2 Evaluations of LHMapping Algorithm

4.2.1 Evaluation of Overhead and Accuracy

We first evaluated the performance of the LHMapping algorithm itself. We used two indexes: the running time and mapping accuracy. We compared our algorithm with the Scotch algorithm. The target application was run while inter-thread communication events were counted to obtain the overall communication matrix. The matrix and hardware topology information were passed to the two algorithms, which were run to obtain the grouping results and record the running time.

We used the 4-thread, 8-thread, 16-thread, and 64-thread programs of OpenMP implementation of SP (SP-OMP) to evaluate our grouping algorithm on the C80 machine. We compared the grouping time of the two algorithms, as shown in Fig. 3 Grouping time of LHMapping and Scotch algorithms.

For all the programs, the running time of the LHMapping algorithm is 2–4 times shorter than that of the Scotch algorithm. This result indicates that the LHMapping algorithm has greater grouping efficiency. In addition, according to Fig. 3, the running time of the two algorithms grows almost exponentially with increasing thread number. This result is consistent with the results of our time complexity analysis.

For all the programs, the running time of the LHMapping algorithm is 2–4 times

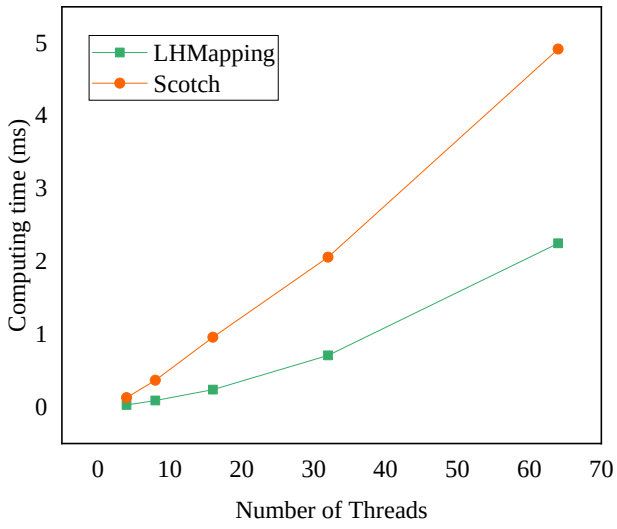


Fig. 3 Grouping time of LHMapping and Scotch algorithms

shorter than that of the Scotch algorithm. This result indicates that the LHMapping algorithm has greater grouping efficiency. In addition, according to Fig. 3, the running time of the two algorithms grows almost exponentially with increasing thread number. This result is consistent with the results of our time complexity analysis.

In addition, to evaluate the accuracy and demonstrate the effectiveness of the LHMapping algorithm, we introduce an index called *MappingAccuracy* to evaluate the mapping accuracy of TreeMatch [19].

$$MappingAccuracy = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{Comm[i][j]}{latency[map[i]][map[j]]} \tag{9}$$

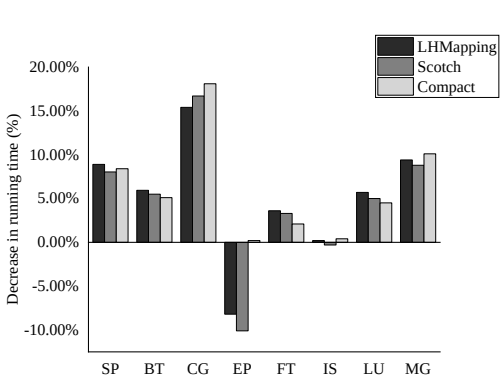
where n is the number of threads, $Comm[i][j]$ is the value of the communication between thread i and thread j, and $latency[map[i]][map[j]]$ is the data transmission delay between the core to which thread i is mapped and the core to which thread j is mapped. The latency can be calculated using lmbench [20]. Theoretically, a higher value of *MappingAccuracy* indicates a better mapping effect. We chose the 64-thread SP-OMP programs and calculated the *MappingAccuracy* value of the two mapping algorithms according to their grouping results. The results are shown in **TABLE 2** MappingAccuracy values of the Two Algorithms.

TABLE 2 MappingAccuracy values of the Two Algorithms

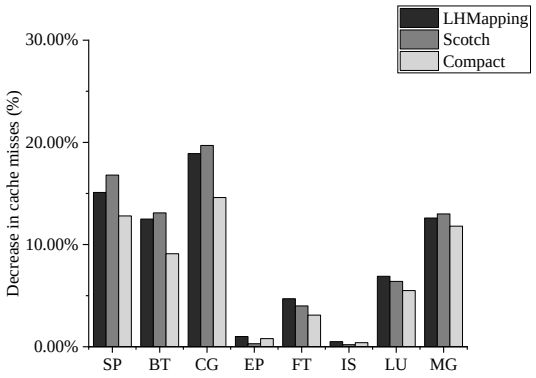
Algorithm	MappingAccuracy
LHMapping	1.21×10^7
Scotch	1.44×10^7

The data in TABLE 2 MappingAccuracy values of the Two Algorithmsindicate that the LHMapping algorithm has slightly lower mapping accuracy than the Scotch algorithm. However, the results have the same order of magnitude, and the difference between

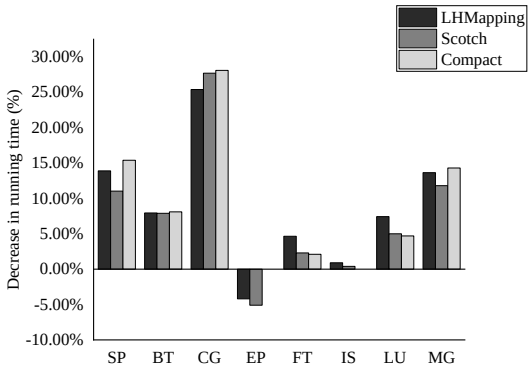
them is not large; thus, there is no substantial impact on the mapping result. The comparative evaluation reveals that the LHMapping algorithm outperforms the Scotch algorithm.



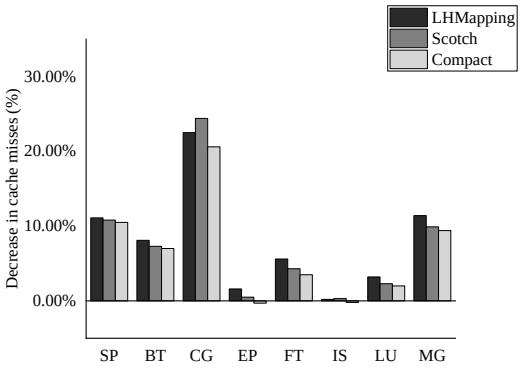
(a) Decrease in running time on PC



(b) Decrease in cache misses on PC



(c) Decrease in running time on C80



(d) Decrease in cache misses on C80

Fig. 1 Comparison of optimization effects of using LHMapping algorithm for thread mapping

4.2.2 Evaluation of the Overall Optimization Effect of the Mapping Mechanism

Following the aforedescribed evaluation, we combined our LHMapping algorithm and the Scotch algorithm into an overall mapping mechanism to generate two mapping optimization mechanisms with different grouping algorithms. Both mechanisms were implemented as kernel modules in Linux. We compared the performance gains of two indicators: the program running time and cache misses. The LHMapping algorithm, Scotch algorithm, and Compact pattern provided by OpenMP were compared. We ran the 8-thread A-Class OpenMP implementation of NPB (NPB-OMP) benchmarks on PC and 64-thread B-Class benchmarks on C80. The results are shown in Fig. 1 Comparison of optimization effects of using LHMapping algorithm.

Dynamic mapping using the LHMapping algorithm has clear optimization effects on the SP, BT, CG, LU, and MG programs. The maximum decrease in the program running time is approximately 25%, and the maximum decrease in cache misses is approximately 23%. In addition, mapping optimization using the LHMapping algorithm has better optimization effects on the FT, LU, and MG programs and other

programs with more balanced communication and computation. For the computationally intensive program EP, which has no inter-processor communication, the mapping mechanism has no optimization effect but instead introduces additional overhead, degrading its performance. For the full-switch communication program IS, the mapping mechanism has almost no optimization effect. In fact, there is no good mapping method that solves the communication imbalance in these applications.

4.3 Evaluations of Variables

4.3.1 Reduction Coefficient α

On the C80 machine, we set α to 0, 0.25, 0.5, 0.75, or 1 with a fixed *interval* value of 200 ms. We also compared the performance gains of the mapping mechanism using the 64-thread SP-OMP programs for these values of α. The results are shown in Fig. 2 Effect of α on optimization effect.

When α = 0.75, the dynamic mapping mechanism has the best optimization effect on the SP-OMP program. When α = 0.5, the optimization effect is also significant and is similar to the effect at α = 0.75. Because the calculation is also simple when α = 0.75, we selected α = 0.75 as the optimal value.

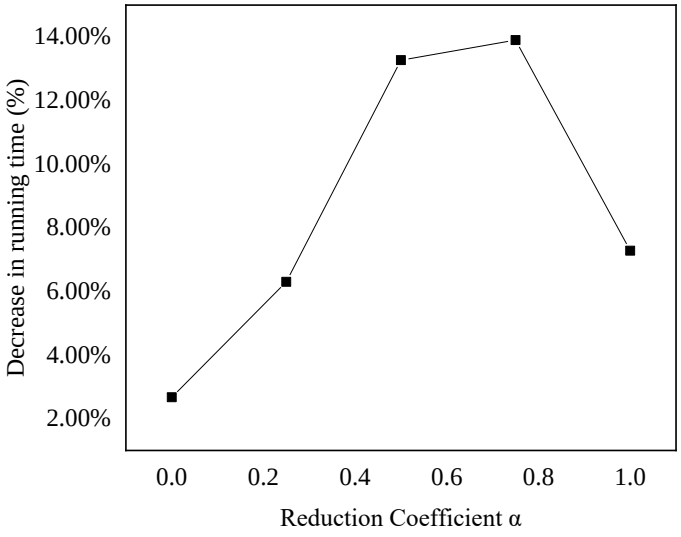


Fig. 2 Effect of α on optimization effect

4.3.2 Interval

We initially set the interval to 100, 200, 300, 400, or 500 ms, with a fixed reduction coefficient α of 0.75. If a grouping result was similar to the previous one, the interval was increased by 50 ms; otherwise, the interval was decreased by 50 ms. The interval was always maintained between 50 ms and 1 s. The decreases in the running time of the 64-thread SP-OMP program on C80 at these intervals were compared, as shown in Fig. 3 Effect of interval on optimization effect.

When the interval is 200 ms, the mapping mechanism has the best optimization effect on the SP-OMP program. The added control over the interval range ensures that the interval gradually reaches its optimal value during the dynamic mapping execution. Therefore, the difference in performance among these intervals is not large. We ultimately selected an interval of 200 ms.

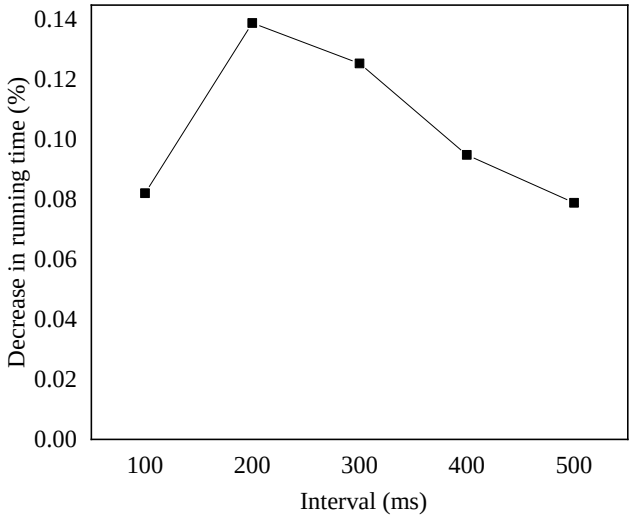


Fig. 3 Effect of interval on optimization effect

5 Conclusions

Mapping optimization is usually an important way to improve application performance based on computer architecture. In this paper, we designed and implemented a dynamic thread-to-core mapping mechanism for OpenMP multithread parallel applications that are running on NUMA machines to avoid unbalanced communication overhead. Our mechanism includes inter-thread communication counting, periodic thread grouping computation, and setting of the thread–CPU affinity, etc. To further improve the effect of mapping optimization, we proposed a grouping algorithm, LHMapping. The algorithm is based on the topology tree of the hardware objects and performs grouping level by level in a bottom-up manner. The breadth-first traversal strategy is adopted for processing each level's grouping, and the elements with the highest communication value are sequentially added to the group. The grouping results are finally mapped to a hardware object. In an evaluation with NPB suites on a 80-core NUMA cluster, both of our mapping machenism and LHMapping algorithm showed a good optimization effect on the two performance indexes of running time and cache misses.

Our study provides a solution for optimizing the matching of application programs and computing resources, and also realizes a basis for further research on system-level

mapping optimization and mapping optimization on heterogeneous platforms. In the future, we will aim to improve the scalability of our dynamic mapping mechanism and optimize the performance of the application considering multiple performance measures.

REFERENCES

[1] M. Awasthi, D. Nellans, K. Sudan, R. Balasubramonian and A. Davis. Handling the problems and opportunities posed by multiple on-chip memory controllers[C]// 2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT), Vienna, 2010, pp. 319-330.

[2] S.H. Bokhari. On the Mapping Problem[J]. IEEE Transactions on Computers, 2006, C-30(3):207-214.

[3] Klug, Tobias, et al. autopin–automated optimization of thread-to-core pinning on multicore systems[J]. Transactions on high-performance embedded architectures and compilers III. Springer, Berlin, Heidelberg, 2011. 219-235.

[4] Radojkovic, Petar, et al. Thread assignment of multithreaded network applications in multicore/multithreaded processors[J]. IEEE Transactions on Parallel and Distributed Systems. (2012): 2513-2525.

[5] Wang Z , O'Boyle M F P . Mapping parallelism to multi-cores: a machine learning based approach[C]// Acm Sigplan Symposium on Principles & Practice of Parallel Programming. ACM, 2009.

[6] Suleman MA, Qureshi MK, Patt YN. Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on CMPs[C]//Proceedings of the ACM 13th International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS). 2008:227-286.

[7] Ju, Tao, et al. Energy-Efficient Thread Mapping for Heterogeneous Many-Core Systems via Dynamically Adjusting the Thread Count[J]. .Energies (2019): 1346.

[8] Acosta C , Cazorla F J , Alex Ramírez, et al. Thread to Core Assignment in SMT On-Chip Multiprocessors[C]// International Symposium on Computer Architecture & High Performance Computing. IEEE, 2009.

[9] Ştirb I. Improving runtime performance and energy consumption through balanced data locality with NUMA-BTLP and NUMA-BTDM static algorithms for thread classification and thread type-aware mapping[J]. International Journal of Computational Science and Engineering. 2020;22(2-3):200-10.

[10] Cruz E H M, Diener M, Alves M A Z, et al. Dynamic thread mapping of shared memory applications by exploiting cache coherence protocols[J]. Journal of Parallel & Distributed Computing, 2014, 74(3):2215-2228.

[11] Osiakwan K, AKL Selimg. The Maximum Weight Perfect Matching Problem for Complete Weighted Graphs IS in PC[C]// IEEE Second Symposium on Parallel & Distributed Processing. IEEE Computer Society, 1990.

[12] Devine K D , Boman E G , Heaphy R T , et al. Parallel hypergraph partitioning for scientific computing[C]// International Parallel & Distributed Processing Symposium. IEEE, 2006.

[13] F. Pellegrini. Static mapping by dual recursive bipartitioning of process architecture graphs[C]// Proceedings of IEEE Scalable High Performance Computing Conference, Knoxville,

TN, USA, 1994, pp. 486-493.
[14] Diener M , Cruz E , Alves M , et al. Kernel-Based Thread and Data Mapping for Improved Memory Affinity[J]. IEEE Transactions on Parallel and Distributed Systems, 2015, 27(9):1-1.
[15] François Broquedis, Jérôme CletOrtega, Stéphanie Moreaud, et al. hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications[C]// 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing. IEEE, 2010.
[16] Love R . CPU Affinity[J]. Linux Journal, 2003(111):p.18-22.
[17] Jin, H. & MA, Frumkin. The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance. 1999.
[18] Bailey, David H, Barszcz, Eric, Barton, John T, et al. The Nas Parallel Benchmarks.[J]. international journal of supercomputer applications, 2010, 2(4):158 - 165.
[19] Jeannot E , Mercier G , Tessier F . Process Placement in Multicore Clusters:Algorithmic Issues and Practical Techniques[J]. IEEE Transactions on Parallel and Distributed Systems, 2014, 25(4):993-1002.
[20] Mcvoy L W , Staelin C . lmbench: Portable Tools for Performance Analysis[C]// Proceedings of the 1996 annual conference on USENIX Annual Technical Conference. 1996.