energies

MDPI

Article

DagTM: An Energy-Efficient Threads Grouping Mapping for Many-Core Systems Based on Data Affinity

Tao Ju [1,2,*], Xiaoshe Dong [1,*], Heng Chen [1] and Xingjun Zhang [1]

[1] School of Electronics and Information Engineering, Xi'an Jiaotong University, Xi'an 710049, China; hengchen@mail.xjtu.edu.cn (H.C.); xjzhang@mail.xjtu.edu.cn (X.Z.)

[2] School of Electronics and Information Engineering, Lanzhou Jiaotong Universtiy, Lanzhou 730370, China

* Correspondence: jutao2011@stu.xjtu.edu.cn (T.J.); xsdong@xjtu.edu.cn (X.D.); Tel.: +86-158-0290-1805 (T.J.); +86-29-8266-3951 (X.D.)

Academic Editor: Hua Li

Received: 18 July 2016; Accepted: 7 September 2016; Published: 20 September 2016

Abstract: Many-core processors are becoming mainstream computing platforms nowadays. How to map the application threads to specific processing cores and exploit the abundant hardware parallelism of a many-core processor efficiently has become a pressing need. This work proposes a data affinity based threads grouping mapping strategy Data Affinity Grouping based Thread Mapping (DagTM), which categorizes threads into different groups according to their data affinity and the hardware architecture feature of many-core processors. After that, the thread groups are mapped to the specific processing cores to be energy efficiently executed. More specifically, first, the intra-thread data locality is analyzed by computing the data reuse distance, and the inter-thread data affinity is quantified by affinity matrix. Second, the threads are categorized into different groups via affinity subtree spanning algorithm. Finally, the thread groups are assigned to different processing cores through static binding. DagTM is able to reduce conflicts of the shared memory access and additional data transmission, increase utilization of the computing resources, and reduce entire system energy consumption. Experimental results show that DagTM obtains a nearly 14% improvement in computing performance, and a nearly 10% reduction in energy consumption compared with the traditional thread mapping mechanism under the condition of not introducing additional runtime overhead.

Keywords: many-core system; thread mapping; data affinity; data reuse distance; thread grouping

1. Introduction

Improving computing performance and reducing energy consumption remain key problems in the high-performance computing domain [1]. The heterogeneous many-core system has emerged as a promising solution in energy-efficient computing. In the emerging heterogeneous many-core systems composed of a host processor and co-processor, the host processor is used to deal with complex logical control tasks (i.e., task scheduling, task synchronizing, and data allocating), and the co-processor is used to compute large-scale parallel tasks with high computing density and simple logical branch. These two processors cooperate to compute different portions of a program to improve the program energy efficiency. The host processor generally adopts chip multi-processors that contain a limited number of processor cores, and the co-processor generally adopts an emerging many-core processor, such as graphics processing unit (GPU) and Intel many integrated core (MIC), which integrates many processing cores (generally tens or even hundreds of cores) in a single chip, and these processing cores are connected via interconnection network and employs simultaneous multithreading

(SMT). For instance, Intel MIC [2] is a new type of many-core processor architecture. It extends the traditional microprocessor vector, integrating many microprocessors onto a single chip. All processing cores within each microprocessor share the same last level cache and operate at the same frequency. These microprocessors are connected via a ring interconnecting network in order to further improve computing energy efficiency.

With the development of many-core technology, the increasing number of processing cores deteriorates the contention for shared resources (such as shared cache and shared bandwidth). In addition, hierarchical non-uniform memory access (NUMA) memories lead to more complex memory access. If the threads with frequent data interaction are mapped to different hardware threads on different processing cores, higher memory access latency and higher data transmission overhead will occur. On the contrary, if the threads without data affinity are mapped to a same processing core, higher memory access conflicts and higher cache misses will occur. These higher memory access latency, higher data transmission overhead, higher memory access conflicts, and higher cache misses will result in a computing performance decrease and energy consumption increase. Therefore, memory architecture features and data access pattern should be simultaneously considered in the process of mapping the application threads to processing cores in many-core systems.

State-of-the-art techniques for thread mapping which rely on the data locality can be classified into static and dynamic mapping. Static mapping generally profiles and analyzes offline the data, accesses program information, and partitions the workloads into different threads. Then the threads are directly mapped to the processing cores according to user-defined mapping, e.g., using the Open Multi-Processing (OpenMP) affinity clause. This static mapping is simple and does not introduce additional overhead. However, it is unable to practically reflect the data locality between threads running on a specific processor. Dynamic mapping can assign threads to processing cores by dynamically profiling and thread migration, but this will introduce high runtime overhead and impact the program performance. Since the traditional multi-core system has a limited number of processing cores, it can obtain acceptable computing performance by using the existing mapping mechanisms. However, the existing mapping mechanisms cannot meet the higher requirements of performance and efficiency for many-core systems. The reason is that the many-core systems have more processing cores and more complex memory hierarchy.

In order to handle the above challenges, we first analyze the data locality of a thread by computing the data reuse distance, which is a rigorous and hardware-independent data locality metric [3]. To efficiently compute data reuse distance of a thread, we design a data reuse distance computation algorithm by constructing a balanced binary tree. We then propose the data locality pattern to determine the data affinity between threads, and use the affinity matrix to quantify the data affinity. On the basis of data affinity, the threads are categorized into different groups by an affinity subtree spanning (ASS) algorithm combined with the many-core system hardware architecture feature. The ASS algorithm can ensure that data correlations between threads within the same group are as large as possible, and data correlations between threads in different group as small as possible. After that, the threads grouping mapping strategy Data Affinity Grouping based Thread Mapping (DagTM) is implemented. DagTM maps the different threads groups to different processing cores based on the data affinity between threads and the architecture features of a many-core system, in order to minimize the data interaction between processing cores, fully exploit the computing power of a processing core, and reduce the system energy consumption. Furthermore, DagTM will not produce additional runtime overhead, because the thread grouping is conducted before the actual execution of a program.

Experimental results show that DagTM improves the application performance by nearly 14%, and decreases energy consumption by 10% on average compared with the traditional Operating System (OS) thread mapping mechanism for PARSEC [4] benchmark programs running on an Intel MIC heterogeneous many-core system. The key contributions of this work are as follows:

(1) We design an algorithm for calculating data reuse distance by constructing a balanced binary tree. The algorithm can efficiently implement the data reuse distance calculation by inserting, traversing, and deleting data nodes during the process of constructing a balanced binary tree.

(2) We propose data locality patterns that reflect the different data correlations, and use an affinity matrix to quantify the data affinity between threads.

(3) We design an affinity subtree spanning algorithm based on an affinity graph to implement thread grouping.

(4) We implement the thread grouping mapping strategy DagTM based on the data affinity on the Intel MIC heterogeneous many-core system.

2. Related Work

A large number of efforts have been done for mapping threads to processing cores based on the data locality. Jiang et al. [5] introduced the concurrent data reuse distance concept by extending the traditional data reuse distance, connected concurrent reuse distance with the data locality of each individual thread by using a probabilistic model, and presented a solution to collect and apply the concurrent reuse distance on Chip Multi-Processor (CMP) platforms. Zhang et al. [6] designed a novel data locality optimization strategy for multicores, which is able to balance both inter and intra-core reuses. The strategy is essentially an exhaustive comparison method, which needs to calculate the data reuse weights, construct a data dependence graph and data sharing graph in advance, and tradeoff performance and overhead. In addition, the strategy mainly focuses on the multi-core single threading without considering simultaneous multi-threading. Drebes et al. [7] proposed a resource-aware approach combined with topology-aware work stealing, dependence-aware memory allocation, and work pushing. The approach can significantly improve the performance of some memory-bound applications. However, for computation-bound applications, the approach may not achieve the ideal performance due to additional run-time overhead. Lu et al. [8] proposed a software framework that partitions the cache at the data object level to reduce cache misses. Unlike our strategy, the work focuses mainly on reducing cache misses to improve performance without considering the data interference between different processing cores. Moreover, it needs to modify the Linux kernel to implement the proposed framework, which restricts its generality. Muralidhara et al. [9] proposed a cache hierarchy-aware application grouping algorithm to find an application-to-core mapping. The work mainly analyzes the memory access relationship between different applications according to the sampling reuse distance distribution on the simulator, and groups the workload on the coarse-grained program level, which could not fully reflect the data interaction characteristics between different threads in the same application. Diener et al. [10] proposed a mechanism to improve memory access locality by reducing accesses to remote caches and memories. In the process of program execution, sharing threads are migrated to the same memory level processing core, and memory pages accessed by a thread are migrated to a node that runs the thread to improve memory access performance. However, the mechanism will introduce additional runtime overhead. Ding et al. [11] proposed a cache hierarchy-aware code mapping and scheduling strategy for multicore architectures. In the process of mapping, the loop iteration, data reuse, and processor memory hierarchy are abstracted as iteration vector, reuse vector, and core vector, for implementing the loop iterations to processing cores mapping by using the algebraic method. The method calculates the data locality by comparing the vectors. For the computation intensive loop iteration, this method can achieve good computing performance, but for the storage density and communications intensive applications, simply comparing the vector is not able to effectively extract relative complex data correlation between threads. Cruz et al. [12] proposed a mechanism to detect the communication patterns of shared memory applications by monitoring cache coherence protocols, and proposed an algorithm to dynamically migrate the threads. This mechanism could implement the dynamic thread mapping to reduce communication overhead supported by the certain hardware. Tousimojarad et al. [13] proposed an extended lowest load technique by using a heuristic to find the optimal target core for each thread. The work aims mainly to provide load balancing in a multithreaded multiprogramming environment. Marongiu et al. [14] considered a representative template of a modern multi-cluster embedded Multiprocessor System-on-chip (MPSoC), and presented an extensive evaluation of the cost

associated with supporting OpenMP. They adopted a hierarchical barrier algorithm to improve the performance of global synchronization, and introduced extensions for data distribution at the cluster level to implement data sharing in an effective manner. Poovey et al. [15] proposed four novel, hybrid hardware/software, pattern-based thread mapping predictors, which aims to provide load balancing to improve the performance. Our work mainly utilizes the data locality between threads to map the thread to the appropriate processing cores to improve energy efficiency. However these works are not orthogonal to our work, they can mutual combine to further improve the energy efficiency.

The different thread mapping strategies mentioned above, either introduce extra runtime overhead, or need customized support from compilers or special hardware, which limit their versatility and effectiveness. However, DagTM simultaneously takes into account the memory access characteristics of application threads and hardware architecture features of many-core processors, analyzes the data affinity of application threads, and then divides the threads into different groups. DagTM can reduce shared memory access conflicts and unnecessary data transmission, improve program computing performance, and reduce system energy consumption without requiring additional runtime overhead and special hardware support.

3. Threads Grouping Mapping Framework

The thread grouping mapping framework of DagTM is shown in Figure 1. First, the application program is divided into the corresponding number of application threads according to the maximum number of hardware threads supported by the many-core processor. The computing tasks are evenly allocated to different application threads. The data locality of computing tasks is not considered during task partition, and will be taken into account in the subsequent process of threads grouping. We develop the parallelism based mainly on the loop portions of benchmark programs. Given that the major computing task is focused on the loop portions for most of the programs, inserting the OpenMP directive statement # *pragma omp parallel for* in the loop parts of benchmark program can realize the parallelization.

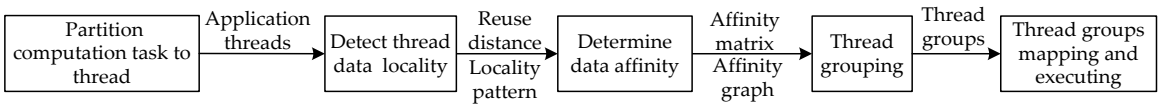

Figure 1. Data Affinity Grouping based Thread Mapping (DagTM) thread grouping mapping framework.

Second, DagTM detects the data locality of thread by computing the data reuse distance. The different threads are merged into the different pattern classes by data locality pattern classification. Third, DagTM analyzes the data affinity between different threads through calculating the number of shared data. The data affinity reflects the inherent data correlation of program, and is independent of the specific running platform. DagTM uses the affinity matrix to quantify the data affinity. Fourth, threads are categorized into different thread groups relying on the data affinity matrix and data affinity graph. Finally, thread groups are mapped to different processing cores to execute by considering the hardware architecture feature of many-core processor.

4. Detecting Thread Data Locality

After the computation tasks are partitioned into different application threads, DagTM detects the data locality of thread by collecting the memory access data of each thread, and calculating the data reuse distance.

4.1. Calculating Data Reuse Distance

Data reuse distance is an ideal metric for detecting program data locality [2,5], which refers to the number of distinct data elements referenced between current and the previous access to the same data

element. The reuse distance is inherent to a program, and is independent of hardware configuration. A small reuse distance means the accessed data has good data locality, which can be accessed at high frequency; on the contrary, it means a poor data locality, and a low access frequency. We design the Pin tool by using Intel Pin Application Program Interface (API) [16,17] to collect the memory access data and calculate the data reuse distance, in parallel. After that, the average data reuse distance of each thread is calculated.

The traditional data reuse distance calculation is realized by using stack- or tree-based algorithms. Stack-based algorithms are inefficient due to a need to sequentially traverse all of the data sequences from the stack top to the bottom. However, the tree based algorithm has high computing efficiency, because it is able to utilize some special properties of the tree to reduce the redundant traversal. In order to improve the speed of traversal data nodes in the tree, Niu et al. [18] used a hash table to assist the data reuse distance calculation. Because the hash table must be constructed before calculating the data reuse distance, it will incur additional overhead in both time and space.

In order to reduce the additional overhead in both time and space, we calculate the data reuse distance when inserting the memory access data into a balanced binary tree, and simultaneously record the data reuse distance into the corresponding data node. In this way, the calculation of data reuse distance and collection of thread access data can be simultaneously completed, which does not need the support of other auxiliary data structure (e.g., a hash table). Therefore it could reduce additional overhead in both time and space. When the whole memory access data scanning is finished, the balanced binary tree containing the memory access data and reuse distance is also generated. The balanced binary tree node is used to record the memory access data. It is defined as a struct. The node data structure definition is shown as follows:

```
struct Node
{int TS; //time stamp that records the access order of data.
float Element; //records the accessed data.
int Frequency; //records the access times of memory access data.
int Weight; //records the number of sub-node contained in the current node.
int RD; //records the data reuse distance of the current node.
}.
```

The whole process of calculating the data reuse distance includes inserting nodes, deleting nodes, and traversing nodes in the balanced binary tree. The average data reuse distance of a whole thread is calculated by traversing the balanced binary tree, which is used to quantify the data locality of a thread.

4.1.1. Collecting Thread Access Data

Collecting memory access data is implemented through inserting data nodes in the balanced binary tree. The data node insertion operation adopts the in-order inserting algorithm of a balanced binary tree based on the time stamp as a primary key.

Before inserting a new data node, the algorithm first traverses the current binary tree, and judges whether the new data has been recorded in the current binary tree. If the data has been recorded, the data reuse distance will be calculated. The existing data node that contains the new data is deleted from the current binary tree, and the current binary tree is adjusted to maintain the balance. Then the new data node is inserted into the current balanced binary tree. If new data is not recorded in the current binary tree, the new data node will be inserted into the binary tree. The algorithm constantly iterates according to the above steps until all the memory access data and corresponding information are recorded. A complete algorithm of collecting access data consists of the following five procedures: scanning memory access data, inserting data node, deleting the original data node, counting data reuse frequency, and calculating data reuse distance. The concrete realization process is as follows:

(1) Initialization: define the data structure of new data node: *Node* (*TS*; *Element*; *Frequency*; *Weight*; *RD)*, an empty binary tree.
(2) Call Pin tool to scan threads access variables, and record time stamp t_i and data element d_i.
(3) Assign initial values to data item variables of new data node: $TS = t_i$; $Element = d_i$; $Frequency = 1$; $Weight = 1$; $RD = \infty$.
(4) Judge whether the data element of new data node is included into the current binary tree, and inorder traverse the binary tree.
(5) If the data element has been contained in the data node of the current binary tree:

    a Call the data reuse distance calculation function (shown in Algorithm 1) to calculate current data reuse distance *RD*.
    b Count the data reuse frequency of new data node: the data reuse frequency equals to the data frequency value of current node plus one.
    c Delete the current data node in the current binary tree.
    d Adjust the balanced binary tree. The conventional AVL algorithm (balanced binary tree algorithm) is used to adjust the balanced binary tree based on the time stamp as a primary key.
    e Insert the new node into the current balanced binary tree.

(6) If the node is not contained in the current balanced binary tree, then directly insert the new data node into the binary tree.
(7) Repeat steps (2)–(4), until all the access data of thread have been scanned. Generate the new balanced binary tree which contained all the memory access data, corresponding data access frequency, and corresponding data reuse distance information.
(8) Adjust the data reuse distance which equals ∞ in the data node. The values ∞ are replaced with the *M* (refers to the number of data nodes in the current binary tree). The *M* is the value of Weight of root node, and it also refers to the max data reuse distance of current thread.
(9) Finish the collecting access data of thread.

4.1.2. The Algorithm of Calculating Data Reuse Distance

The data reuse distance calculation process is encapsulated into an isolated function, which is directly called during collecting thread access data. The algorithm realization is as follows:

(1) Search the data node which contains the data to be inserted into the current binary tree. If the data node is not included, which means that the data is firstly accessed, and its data reuse distance is set as ∞.
(2) If the data node is found in current binary tree, then the data reuse distance calculating process is as follows:

    (a) If the time stamp of the target node (N_{target}) is smaller than the time stamp of the root node (N_{root}), it indicates that N_{target} is the left subtree node of N_{root}. The data reuse distance equals the number of right subtree nodes plus the number of nodes which are included in the left subtree of N_{root} and their time stamps are larger than the time stamp of N_{target}.
    (b) If the time stamp of N_{target} is larger than the time stamp of N_{root}, it indicates that N_{target} is the right subtree node of N_{root}. The data reuse distance is the number of nodes which are included in the right subtree of N_{root} and their time stamps are larger than the time stamp of N_{target}.

(3) Compare the data reuse distance of the current data with the original node in the binary tree, and set the smaller value as the final data reuse distance *RD* of the data node to be inserted in the binary tree in order to ensure the validity of data reuse distance.

(4) In the left or right subtree of N_{root}, the calculation process of the number of nodes in which time stamp is larger than the time stamp of N_{target} is as follows. (*rd* refers to the final number of node).

(a) Set the initial value of *rd* as 0.
(b) Assign the weight value of the right child node of N_{target} to *rd*, and set N_{target} as the current node $N_{current}$.
(c) Backtrack to the parent node of $N_{current}$.
(d) If the $N_{current}$ is not the root node, and its time stamp is larger than the $N_{current}$, the *rd* equals to the weigh value of $N_{current}$ subtracts its left child weigh value. Set the current parent node as the $N_{current}$, and go to the step (c).
(e) If the time stamp of $N_{current}$ is smaller than $N_{target,}$ then set the current parent node as the $N_{current}$ and go to the step (c).
(f) If the $N_{current}$ is N_{root}, then calculation is completed.

The algorithm details are shown in Algorithm 1.

Algorithm 1: Calculating data reuse distance algorithm.

```
1 Input: N_root, N_target
2 Output: RD
3 Begin
4 RD = 0;
5 if ((N_target ->TS) < (N_root ->TS)) then
  //Target node time stamp is smaller than the root node time stamp,
  //the target node is the left subtree node.
6 RD = (N_root -> right_child.weight) + 1 + (N_target ->right_child.weight);
7 p = N_target -> parent;
8 while (p! = N_root)
9 if ((p-> TS) > (N_target -> TS)) then
10 RD + = (p->weight)-(p->left_child.weight);
11 p = p-> parent;
12 else
13 p = p-> parent;
14 end if
15 end while
16 RD = (RD < N_target ->RD)? RD: N_target ->RD;
   //Set the smaller value of data reuse distance between target node
   // and current to be inserted node as the current data reuse distance.
17 return RD;
18 else
   //The target node is in the right subtree of root node.
19 RD = N_target -> right_child.weight;
20 p = p->parent;
21 while (p! = T)
22 if ((p->TS) > (N_target -> TS)) then
23 RD+ = (p-> weight)-(p-> left_child.weight);
24 p = p->parent;
25 else
26 p = p-> parent;
27 end if
28 end while
   //Set the smaller value of data reuse distance between target node
   //and current to be inserted node as the current data reuse distance.
29 RD = (RD < N_target -> RD)? RD: N_target ->RD;
30 return RD;
31 end if
32 End
```

4.2. The Instance of Calculating Data Reuse Distance

The following concrete instance explains the process of collecting the memory access data and calculating the data reuse distance. Table 1 shows the data sequence accessed by the thread, the values of frequency and reuse distance are obtained by executing Algorithm 1.

Table 1. The memory access data sequence.

Time Stamp	1	2	3	4	5	6	7	8	9	10
data element	d	b	c	e	a	b	e	c	f	a
frequency	1	1	1	1	1	2	2	2	1	2
reuse distance	∞	∞	∞	∞	∞	3	2	3	∞	4

Figure 2 shows the process of inserting data elements of the Table 1 into a balanced binary tree, and calculating the data reuse distance and frequency as well. The process mainly includes the searching, deleting, and inserting operation. Finally, it generates a balanced binary tree which contains the unique memory access data *d*, *b*, *c*, *e*, *f*, *a*, and their corresponding data reuse distance information.

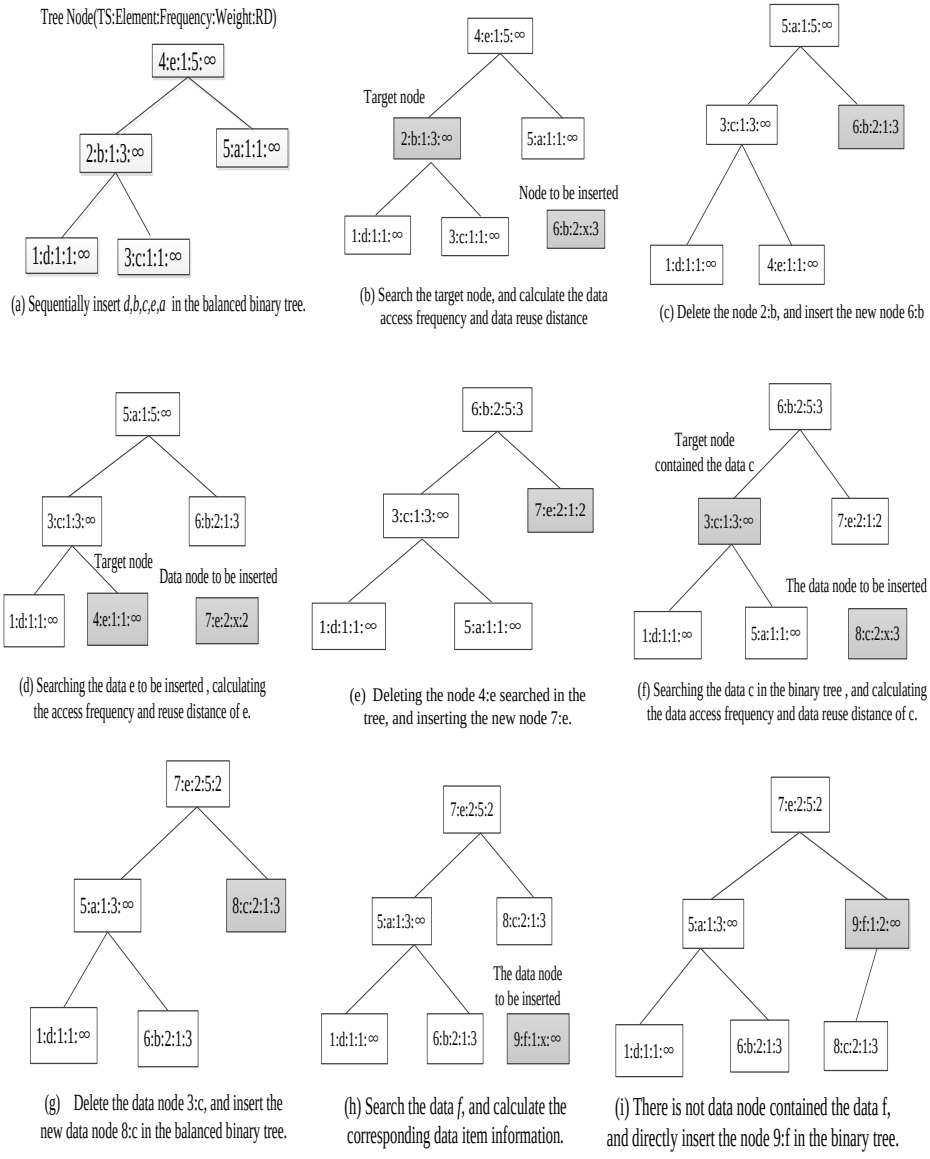

Figure 2. *Cont.*

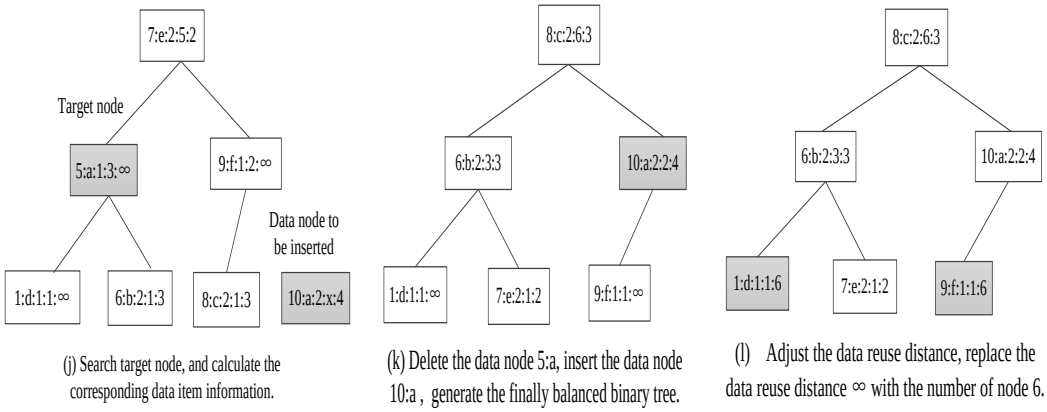

Figure 2. The instance of collecting access data.

4.3. The Average Data Reuse Distance of Thread

After the balanced binary tree is generated, the average data reuse distance of every thread is computed by traversing the corresponding binary tree. Let K refer to the total number of threads, and RD_j (j = 1, 2, . . . , K) be the average data reuse distance of every thread, the data reuse distance of every data is rd_i, and the number of unique data is M. RD_j can be calculated as follows:

$$RD_j = \frac{\sum_{i=1}^{M} rd_i}{M} \tag{1}$$

The average data reuse distance reflects the internal data locality of a thread. The average data reuse distance is greater which means the data reuse rate is low and data locality is poor, and otherwise the high data reuse rate and better data locality.

4.4. The Algorithm Complexity Analysis

Let N refer to the total number of access data of a thread, and the M be the number of unique access data. The whole algorithm implementation mainly includes the insertion, traversal, and deletion of nodes in the balanced binary tree. The main computation overhead is spent on searching the data and calculating data reuse distances. The time complexity of searching for target data nodes is O(M), and the time complexity of calculating data reuse distances is O(log(M/2)), so the total time complexity of the whole algorithm is O(N(M + log(M/2))). The space complexity of the whole algorithm is O(N).

5. Determining Data Affinity

We merge all the threads into different data locality pattern classes based on the data reuse distance. We then analyze and quantify the data affinity between threads. First, we determine the data locality pattern according to the average data reuse distance of every thread. After that, the threads are merged into different pattern classes. Lastly, we analyze the data affinity between threads in every pattern class, and use the data affinity matrix to quantify the data affinity between different threads.

5.1. Definition of Data Locality Pattern

Data locality patterns are classified into three types: data sharing patterns, data dependency patterns, and data isolation patterns. The different data locality patterns are quantified by the data reuse distance. We set the data reuse distance threshold values as $D_{\min}$ and $D_{\max}$, which reflect different data access characteristics, and divide the data reuse distances into three different ranges, each of which corresponds to one of the data locality patterns. Finally, we identify the data locality pattern of each thread by comparing its average data reuse distance with the threshold values $D_{\min}$ and $D_{\max}$. The data locality pattern definitions are as follows:

Definition 1: *Data Sharing Pattern* (*DSP*): $RD_j < D_{min}$. Under this pattern, the data accessed by the thread has strong temporal locality. Threads that belong to this pattern should be assigned to different hardware threads on the same processing core, and the data accessed by the thread should be allocated to the same memory location of thread.

Definition 2: *Data Isolation Pattern* (*DIP*): $RD_j > D_{max}$. Under this pattern, the data accessed by the thread has poor temporal locality. Threads that belong to this pattern should be assigned to different hardware threads of different processing cores of different processors.

Definition 3: *Data Dependency Pattern* (*DDP*): $D_{min} \leq RD_j \leq D_{max}$. Under this pattern, the data accessed by the thread has partially temporal locality. Threads that belong to this pattern should be assigned to different hardware threads of different processing cores on same processor.

The specific threshold values of $D_{\min}$ and $D_{\max}$ are obtained by empirical observation. By measuring the different benchmark programs, we calculate the data locality, and analyze the relationship between data locality and data reuse distance. Comparing the data reuse distance values of programs with strong data sharing, let the max value be $D_{\min}$; and comparing the data reuse distance values of programs with isolated data access characteristic, let the min value be $D_{\max}$. In this article, the obtained values of $D_{\min}$ and $D_{\max}$ are 35% and 85% of the amount of data access in certain interval, respectively.

5.2. Quantifying Data Affinity among Threads

We analyze the data affinity between threads in every pattern class by calculating the number of identical accessed data between different threads, and use the data affinity matrix to quantify the data affinity between different threads. The data affinity matrix reflects the data sharing characteristics between different threads. The matrix row and column label respectively represent different thread Identifies (IDs). Every element value of the matrix represents the number of sharing data between threads marked by the corresponding row and column ID. A greater element value means the data sharing is better between corresponding threads, and the data affinity is also better between them.

We calculate the number of identical accessed data between different threads, and set the number as the corresponding data element value of the data affinity matrix. In order to improve the calculation speed, we compare in parallel the same accessed data between threads that belong to the same data locality pattern class. By comparing the corresponding balanced binary tree of different threads, we compute the number of same data nodes between different binary trees, and the number is the sharing data volume of the corresponding two threads. In addition, the number is recorded into the corresponding element of the data affinity matrix. Finally, a complete data affinity matrix is constructed, which reflects the data affinity between threads. After that, the data affinity matrix is transformed to the data affinity graph which can intuitively reflect the data affinity between threads. The data affinity graph is an undirected weighted connected graph, whose vertex refers to the thread ID, and edge weight refers to the data sharing volume between threads.

6. Threads Grouping Mapping

DagTM implements mapping threads to processing cores in two stages based on the data affinity combined with the memory hierarchy feature of many-core system. The first stage is threads grouping; the second stage is assigning thread groups to different processing cores.

6.1. Threads Grouping

Threads are divided into *K* different thread groups based on the data affinity graph combined with the max number of hardware threads supported by a processing core. The threads grouping needs to ensure the good data affinity between threads in every similar group. It is essentially a combinatorial optimization problem to divide the threads into different groups and ensure a better data sharing in

the same thread group. In the process of thread grouping, the impact of the current thread on other threads should be considered, similarly, the impact of other threads on the current thread should also be considered. In this article, the thread grouping is abstracted as a graph decomposition problem. By designing an affinity sub-tree spanning algorithm, the data affinity graph is decomposed as K subtrees to meet the above requirements. The threads with high data sharing are merged into the same thread group, and the threads with strong memory access conflicts are merged into different thread groups.

6.1.1. Affinity Subtree Spanning Algorithm

The detailed execution process of the affinity subtree spanning algorithm is as follows:

(1) The $G = (V, E)$ is a weighted undirected connected graph (i.e., affinity graph). The vertex V refers to the set of threads, and edge E refers to the set of data affinity between different threads. Each edge $(T_i, T_j) \in E$ has a weight value $\omega\ (T_i, T_j)$, which refers to the sharing data volume between the corresponding threads (as shown in Figure 3a). The total number of vertexes of graph G is N_t, which refers to the total number of threads; N_p refers to the number of nodes of every subtree (i.e., the number of threads of every thread group), the number corresponds to the max number of hardware threads supported by the specific many-core processor; K refers to the number of finally generated subtrees.

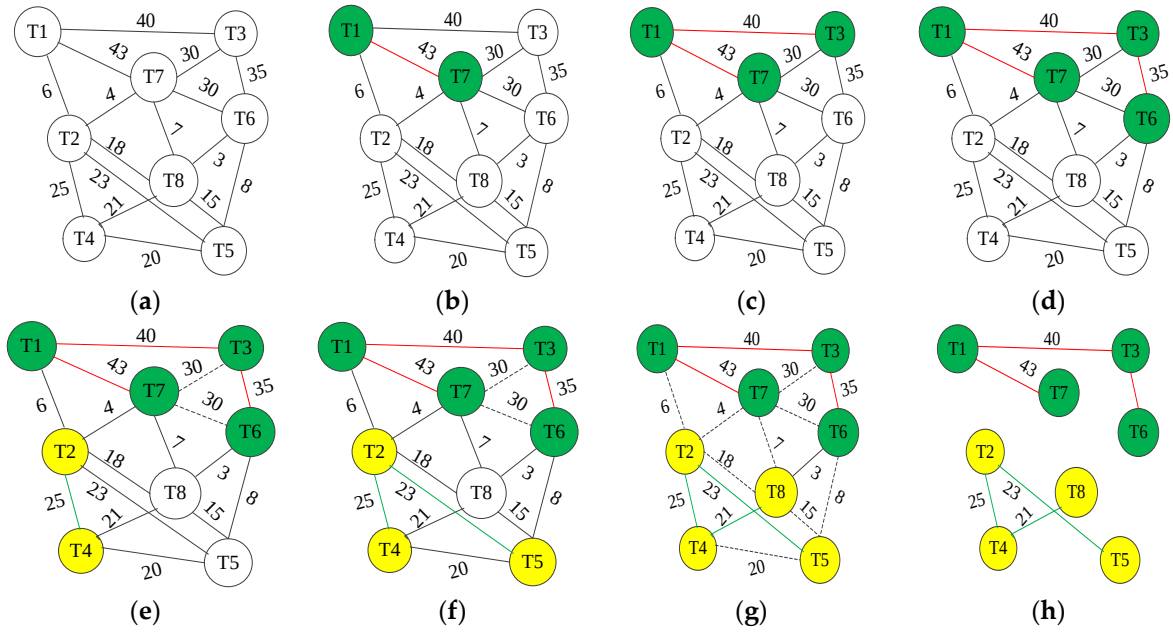

Figure 3. Affinity sub-tree spanning process instance. (**a**) The affinity graph; (**b**) Find the edge (T1, T7); (**c**) Find the edge (T1, T3); (**d**) Find the edge (T3, T6); (**e**) Generate the subtree (T1, T7, T3, T6); (**f**) Find the edge (T2, T4), (T2, T5); (**g**) Find the edge (T4, T8); (**h**) Generate the subtree (T2, T4, T5, T8).

(2) Generating the K subtrees from the weighted undirected connected graph G. ST_k refers to the different subtrees. Each generated subtree contains N_p nodes at most, and it must ensure the sum of weight values of subtree edges is the largest, so it should satisfy following constraint conditions:

i. $K = \left\lceil \frac{N_t}{N_p} \right\rceil$

ii. $\omega\left(ST_k\right) = \max \sum_{(T_i, T_j) \in ST_k} \omega\left(T_i, T_j\right)$

(3) The sub-tree spanning algorithm is shown as Algorithm 2.

Algorithm 2: Affinity Sub-tree Spanning Algorithm.

```
1 Input: G = (V, E), N_p.
2 Output: ST_k (k = 1, 2, . . . , n).
3 Begin
4 K = 1;
5 Search_max_weight_edge (G, E, ST_k);
  //Searching the max weight values edge, and add it into the current subtree ST_k.
6 while (V! = NULL)
7 {
8 do
9 {
10 Search_adjacent_max_weigh_edge (G, E_i, ST_k );
   //Searching the max weight edge connecting the current edge E_i, and add //it into the current
subtree ST_k.
11 } while (V_i < N_p && V ! = NULL)
12 end do
13 Delete_adjacent_edge (G, ST_k );
   //Deleting adjacent edges are not to be included in the current subtree ST_k.
14 Generate_subtree (ST_k);
   //Generate the new subtree ST_k.
15 K++;
16 Search_max_weight_remain_edge (G, E-E_{k-1}, ST_k);
   //Searching the max weight values edge in the remained graph G.
17 }
18 end while
19 End.
```

6.1.2. The Algorithm Complexity Analysis

The complexity of the affinity subtree spanning algorithm is related to the number of vertexes and edges of the affinity graph. The time complexity is mainly related to the number of edges of the data affinity graph. The initial comparison needs to compare the weight values of all edges. Subsequently, the number of comparisons will be reduced gradually. If the number of edges is n, the number of comparisons will be n, $n-1$, $n-2$, . . . , 1, the total time complexity of algorithm is $O(n^2/2 + n/2)$. In order to reduce the space complexity of the algorithm, the adjacency matrix is used to store the affinity graph, and the upper diagonal information of the adjacency matrix is only stored. So, the space complexity of the algorithm is $O(V^2/2)$.

6.1.3. Instance of Threads Grouping

A concrete instance of the affinity sub-trees generating procedure is shown in Figure 3. The graph contains eight threads, and the max number of hardware thread supported by the processing core is four. Finally, the two sub-trees are generated, i.e., the eight threads are merged into two thread groups ST_1 (T1, T7, T3, T6) and ST_2 (T2, T4, T5, T8). The generated each thread group contains four threads at most, and the sum of weight value of sub-tree edge is the largest, which ensures the data correlation between threads within the same group as large as possible.

6.2. Mapping Rules

Combined with the memory hierarchy graph, the threads in the affinity sub-tree are mapped to the different processing cores of many-core processor. Referring to the mapping algorithm in the [9,10],

the data affinity sub-trees and memory hierarchy graph are used as input, we realize the thread mapping by static binding of threads to processing cores. The mapping rules are as follows:

Rule 1: The application threads in the same thread group should be assigned to different hardware threads in the same processing core as far as possible. If the hardware threads in the same processing core are all allocated, the application threads should be assigned to the adjacent processing cores. The aim is to reduce the additional data replication and memory access latency, and improve the utility of the sharing cache.

Rule 2: The application threads in the different thread groups should be assigned to different processing cores. Let the isolated threads be dispersed among different processing cores with isolated cache space. The aim is to avoid a high data transmission latency and shared cache contention caused by the great number of different data replications of application threads.

6.3. DagTM Implementation

DagTM implementation includes the data locality detecting of thread, data affinity quantifying, thread grouping, and thread to processing core mapping and executing. The concrete implementation is shown in Figure 4.

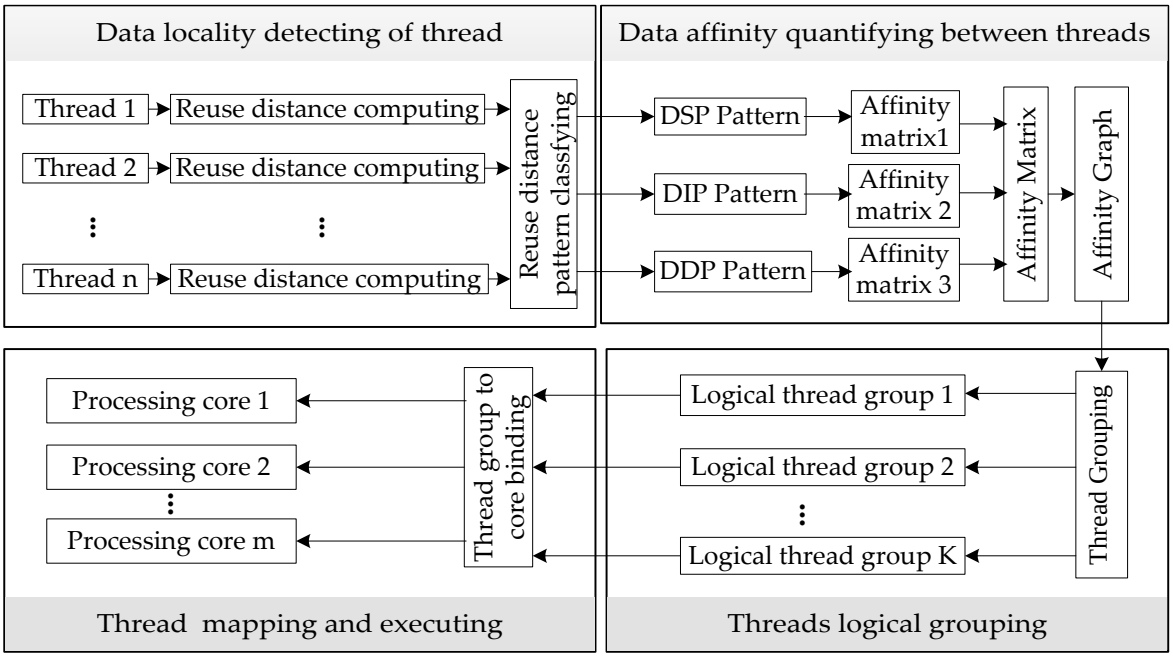

Figure 4. Implementation framework of DagTM.

We utilize an eight-thread parallel application to explain the complete DagTM mapping process to the Intel MIC processor. The task model and target platform model can refer to our previous article [19]. The detailed mapping process is as follows:

(1) DagTM first computes the average data reuse distance of each thread, identifies the locality pattern of different threads, and merges the threads into different pattern classes on the basis of the locality pattern [20–22]. After that, DagTM constructs the data affinity matrix (as shown in Figure 5) of threads by counting the sharing data volume between threads in different pattern classes, and transforms the data affinity matrix to the data affinity graph (as shown in Figure 6).

	T1	T2	T3	T4	T5	T6	T7	T8
T1	0	0	40	0	0	0	43	0
T2	6	0	0	25	23	0	4	18
T3	40	0	0	0	0	35	30	0
T4	0	25	0	0	20	0	0	21
T5	0	23	0	20	0	0	0	15
T6	0	0	35	0	8	0	30	3
T7	43	4	30	0	0	30	0	7
T8	0	18	0	21	15	3	7	0

Figure 5. Data affinity matrix.

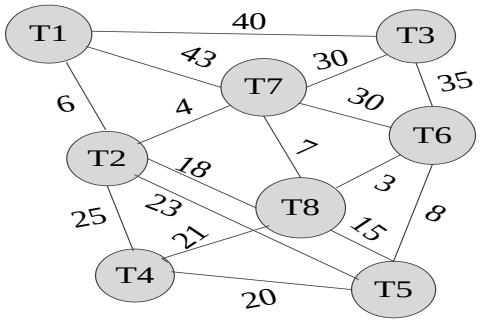

Figure 6. Thread affinity graph.

(2) The threads are categorized into different groups via the affinity subtree spanning algorithm. The presented example is based on the Intel MIC many-core system. For the specific Intel MIC heterogeneous system architecture readers can consult reference [19], and the memory architecture is shown as Figure 7. The MIC processor supports four hardware threads in each processing core.

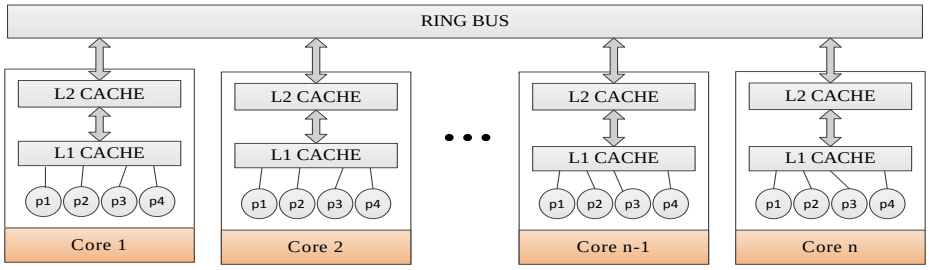

Figure 7. Intel MIC processor memory hierarchy.

(3) After the thread grouping is completed, the thread groups are assigned to the different processing cores. We need to make sure that the threads assigned to the same processing core have better data locality, and threads assigned to the different processing cores have the smallest data affinity. The finally mapping result is shown in Figure 8c.

Figure 8 compares the mapping results between traditional OpenMP mapping and DagTM mapping for the same application threads on the same Intel MIC many-core system. The OpenMP Compact mapping mechanism mainly considers making full use of every processing core, and the data locality between threads is not considered. It will assign the threads with high sharing data to the different processing cores, and result in a high additional memory access.

As shown in Figure 8a, the threads T1, T3, T6, and T7 with high data affinity are assigned to different processing cores 1 and 2. The same data copy needs to be stored to the chip cache of core 1 and core 2, respectively, which added the additional memory overhead. The OpenMP Scatter mapping

in Figure 8b mainly considers the load balance, it evenly assigns threads to the different processing cores, and also it does not consider the data locality between threads. Therefore, the scatter mapping method will also introduce high additional data memory access, moreover it is unable to make full use of the processing cores source and will cause high system energy consumption. However, as shown in Figure 8c, the DagTM mapping considers the data locality between different threads, and divides the threads into different groups according to the hardware architecture features of the processing core and data affinity, and then maps the different thread groups to the specific processing cores of a many-core processor, so it could utilize the data locality between threads to improve the data sharing between hardware threads, and reduce the additional data access and data transmission. In addition, it could make full use of the processing core sources to improve utilization of every processing core and reduce the whole system energy consumption.

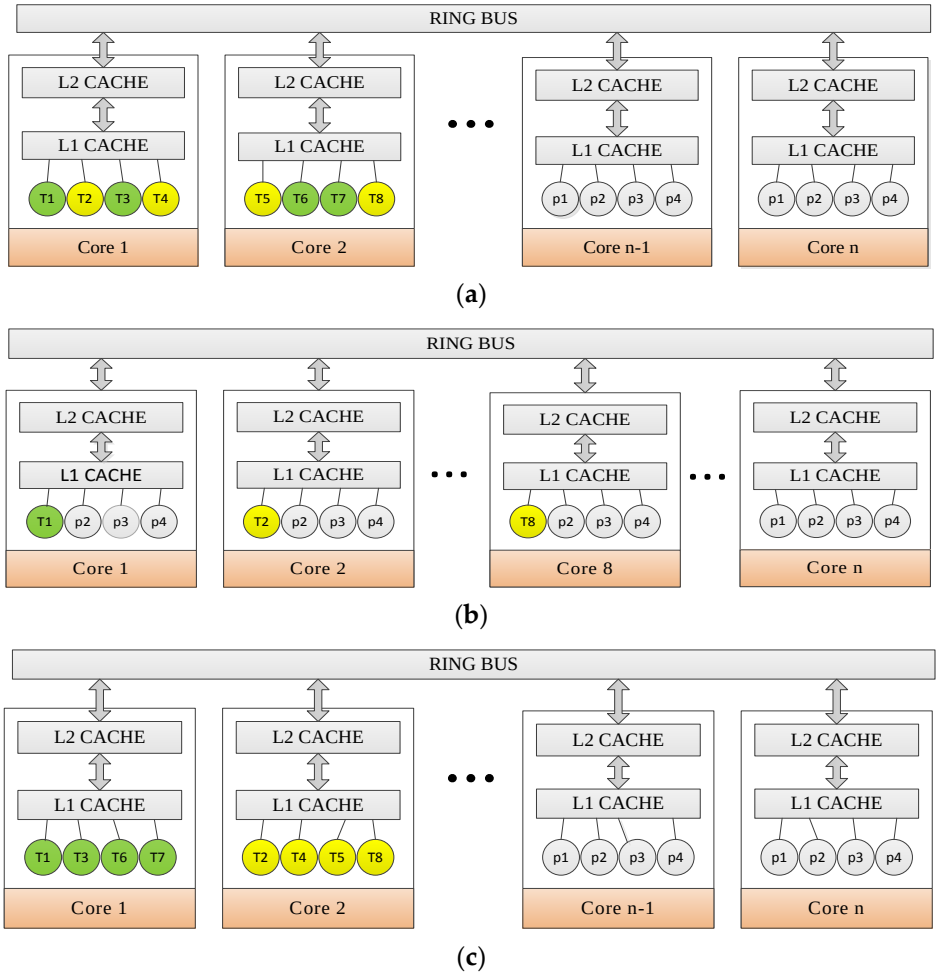

Figure 8. Different thread mapping results. (**a**) OpenMP Compact mapping result; (**b**) OpenMP Scatter mapping result; (**c**) DagTM mapping result.

7. Experimental Evaluation

7.1. Experimental Methodology

We used the PARSEC [4] benchmark suite to evaluate the DagTM. The benchmark programs were executed by using the native input size based on the OpenMP APIs. The experiment was conducted on an Intel MIC heterogeneous many-core system that consists of two eight-core E5-2670 CPUs and one Xeon Phi 7110P MIC co-processor with 64 GB memory and a 300 GB hard disk. The MIC co-processor contains 61 processing cores, and every processing core supports four hardware threads.

The PCI-E x16 bus that connects the CPU and MIC co-processor can transfer data at a maximum transmission speed of up to 16 GB/s. The OS is Red Hat Enterprise Linux Server release 6.3, the performance metrics were obtained by the PAPI_5.4.1 performance measurement tool [16,17]. The soft development environment is Intel parallel_studio_xe_2013_update3_intel64.

The DagTM, OpenMP Compact, OpenMP Scatter, Oracle (the ideal optimized thread mapping for the application obtained by empirical observation), and Kernel Memory Affinity Framework (kMAF) [10] mapping mechanisms were used in benchmark programs, respectively, to compare their performance from the following three aspects: computing performance, energy consumption, and extra overhead.

7.2. Experimental Results and Discussion

7.2.1. Computing Performance

Figure 9 shows the relative improvement of computing performance of different mapping mechanisms for different benchmark programs. The normalized performance improvement ratio was computed via the relative reduce ratio of application execution time of different mapping mechanisms compared to the baseline that OS default mapping mechanism (first-touch policy).

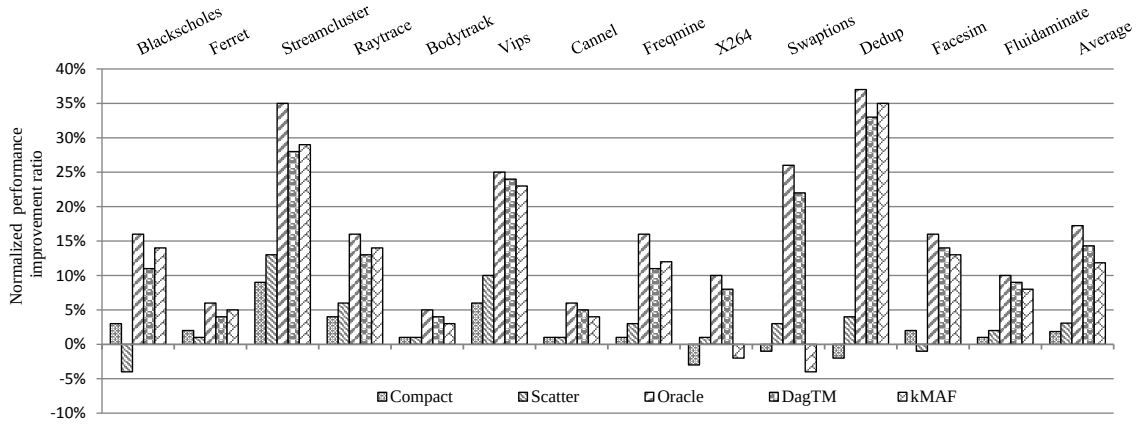

Figure 9. Computing performance comparison.

As shown in Figure 9, the average computing performance was increased by 2%, 3%, 17%, 14%, and 12% compared to the baseline (OS) by Compact, Scatter, Oracle, DagTM, and kMAF, respectively. The DagTM computing performance amounted to 82.4% of the Oracle and was better than the other three mapping mechanisms. The computing performances of the Compact and Scatter were lower than the DagTM and kMAT. In some cases, their computing performances were even lower than the OS default mechanism (e.g., *Blackscholes*, *x264*, *Dedup*, and *Facesim*). The main reason is that the Compact and Scatter do not completely consider the data locality between threads, which will cause the sharing source contention and additional data transmission delay. Furthermore, the DagTM computing performance in some cases was lower than the kMAF (e.g., *Streamcluster*, *Raytrace*, *Freqmine*, and *Dedup*). The reason is that kMAF can dynamically adjust threads according to the runtime data affinity between different threads. kMAF can achieve a better computing performance when running behavior of application has significant changes and the performance benefits obtained by dynamic adjustment is greater than the additional overhead. However, for the applications whose running behaviors have no significant changes, the additional runtime overhead introduced by the kMAF will offset the performance benefits, so their computing performances are lower than the DagTM.

Figure 10 shows the reduction ratio of the last level cache misses normalized to the OS default mapping mechanism. The smaller the normalized value is, the better. Overall, the reduction ratio of the L2 level cache misses of DagTM was superior to the Scatter and Compact. The reduction of the

cache misses of kMAF was the closest to the Oracle, and superior to the others. The main reason is that kMAF is able to monitor the cache status in real-time and dynamically adjust threads to reduce the cache line replication. However, real-time monitoring data affinity will introduce additional runtime overhead, which will offset the part of performance benefits obtained from the reduction cache misses, and impact the whole computing performance. So, as shown in Figure 9, the kMAF average computing performance is not superior to the DagTM.

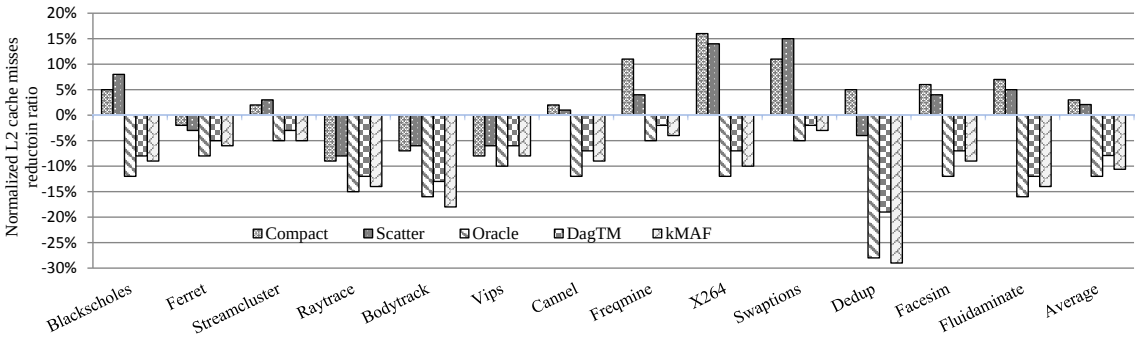

Figure 10. The reduction of the last level cache misses.

7.2.2. Energy Consumption

Apart from performance, reducing energy consumption is another important goal of thread mapping. On the one hand, the energy consumption can be reduced by hardware approaches (e.g., DVFS [23,24]); on the other hand, it can be reduced by efficiently exploiting the computing sources and reducing the execution time. DagTM relatively reduces the whole system static energy consumption by making more efficient use of the computing resources and reducing the execution time. The system energy consumption was measured during the execution of each application by using PAPI components, which provides access to the energy and power values returned by the Intel RAPL interface [17].

As shown in Figure 11, the average energy consumption was reduced by 2.3%, 3.2%, 12.4%, 10.3%, and 8.5% compared to the baseline (OS) by Compact, Scatter, Oracle, DagTM, and kMAF, respectively. Because the DagTM considered the data affinity before the thread mapping, which is able to reduce the memory access contention, improve the sharing sources utilization, reduce data transmission overhead, and reduce the whole execution time, so it could relatively reduce the whole system energy consumption.

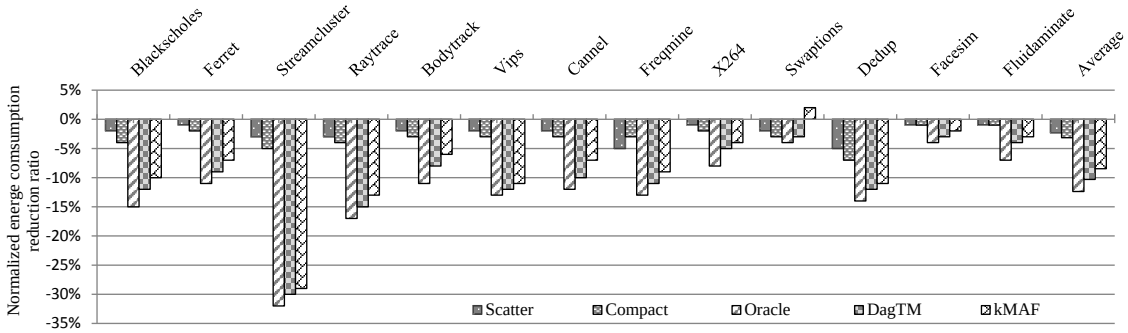

Figure 11. The reduction of the system energy consumption.

7.2.3. Overhead

DagTM will introduce extra overhead before the execution of the program. The extra overhead is mainly attributed to the data locality detection, data affinity determination, and thread grouping.

To approximate the extra overhead for the benchmark program, we measured the time spent mostly on data locality detection. Figure 12 shows the extra overhead of DagTM, which was measured by the execution time ratio of data locality detection compared to the whole program execution time of the different benchmark programs. The average extra overhead introduced by DagTM is nearly 11%.

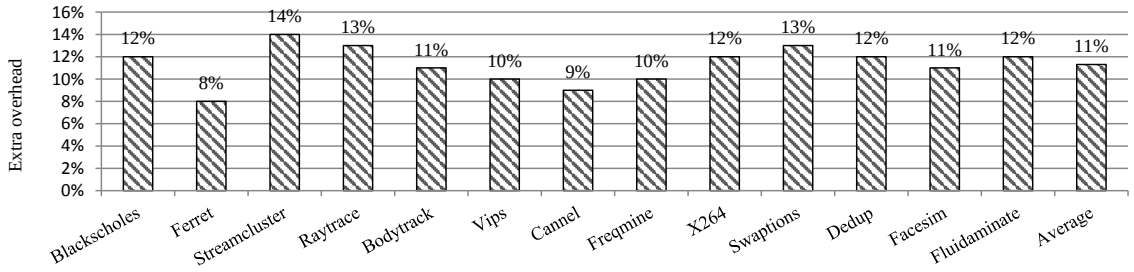

Figure 12. Extra overhead of DagTM.

The Compact and Scatter mappings do not consider the program itself data locality, and directly map the thread to the processing core, which will not introduce additional overhead before and during the program execution. The Oracle mapping obtained the best performance by exhaustively comparing, and introduced the largest additional overhead, which only serves as an ideal mapping standard, and does not serve as a practical mapping approach. The kMAF is able to dynamically adjust the thread according to the running status of a program, which will introduce a certain additional runtime overhead and impact on the computing performance of a program. It needs to trade off the performance benefits and additional runtime overhead. However, DagTM directly implements thread mapping after completing threads grouping. Due to the fact that threads grouping is implemented before the program execution, it will not introduce additional runtime overhead. DagTM is able to realize thread grouping at the cost of a negligible preprocessing overhead. It could make up the shortcoming of the Scatter and Compact approaches, and obtain similar performance improvements without introducing additional runtime overhead compared to the kMAF.

8. Conclusions and Future Work

In this article, we have investigated the mapping problem of thread to processing core based on data affinity. The mapping of threads to the different processing cores of a many-core processor was implemented based on the data affinity between threads considering the memory hierarchy architecture features. The ultimate purpose of this work is to improve the whole system energy efficiency by reducing sharing memory access contention, increasing sharing resource utilization, and reducing data transmission overhead. Specifically, the data locality is detected by computing the data reuse distance; the data affinity is quantified via an affinity matrix; the threads are divided into different thread groups via an affinity sub-tree spanning algorithm. Finally, the thread groups are assigned to the processing cores by static binding. The benchmark programs evaluation results show that the DagTM is effective for improving program computing performance and reducing energy consumption. DagTM is able to reasonably map the threads to different processing cores relying on the data affinity between threads, and improve the whole system energy efficiency without introducing additional runtime overhead.

For the future, we will extend DagTM and combine with the dynamic detecting the phase changes of the running program to realize the hybrid static and dynamic thread mapping based on the data affinity. In addition, we will combine the DagTM with the other thread mapping strategies to adapt multithreaded multiprogramming environment and cluster architecture.

Acknowledgments: We would like to thank the anonymous reviewers for their useful suggestions. This work is supported by the National Natural Science Foundation of China under Grant No. 61572394, the National Key Research and Development Program of China under Grand No. 2016YFB0200902.

Author Contributions: Tao Ju and Xiaoshe Dong conceived and designed the experiments; Tao Ju and Heng Chen performed the experiments; Tao Ju and Xingjun Zhang analyzed the data; Tao Ju wrote the paper.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Brodtkorb, A.R.; Dyken, C.; Hagen, T.R.; Hjelmervik, J.M.; Storaasli, O.O. State-of-the-art in heterogeneous computing. *Sci. Programm.* **2011**, *18*, 1–33. [CrossRef]
2. Liu, X.; Smelyanskiy, M.; Chow, E.; Dubey, P. Efficient sparse matrix-vector multiplication on x86-based many-core processors. In Proceedings of the 27th International ACM Conference on International Conference on Supercomputing, Eugene, OR, USA, 27 March 2013; pp. 273–282.
3. Zhong, Y.; Shen, X.; Ding, C. Program Locality Analysis Using Reuse Distance. *ACM Trans. Programm. Lang. Syst.* **2009**, *31*, 341–351. [CrossRef]
4. Bienia, C.; Kumar, S.; Singh, J.P.; Li, K. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques, Toronto, ON, Canada, 25–29 October 2008; pp. 72–81.
5. Jiang, Y.; Zhang, E.Z.; Tian, K.; Shen, X. Is Reuse Distance Applicable to Data Locality Analysis on Chip Multiprocessors? In Proceedings of the International Conference on Compiler Construction, Paphos, Cyprus, 20–28 March 2010; pp. 264–282.
6. Zhang, Y.; Kandemir, M.; Yemliha, T. Studying inter-core data reuse in multicores. *ACM Sigmetrics Perform. Eval. Rev.* **2011**, *39*, 25–36.
7. Drebes, A.; Heydemann, K.; Drach, N.; Pop, A.; Cohen, A. Topology-Aware and Dependence-Aware Scheduling and Memory Allocation for Task-Parallel Languages. *ACM Trans. Arch. Code Optim. (TACO)* **2014**, *11*, 1–25. [CrossRef]
8. Lu, Q.; Lin, J.; Ding, X.; Zhang, Z.; Zhang, X.; Sadayappan, P. Soft-olp: Improving Hardware Cache Performance through Software-Controlled Object-Level Partitioning. In Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques (PACT), Raleigh, NC, USA, 12–16 September 2009; pp. 246–257.
9. Muralidhara, S.P.; Kandemir, M.; Kislal, O. Reuse Distance Based Performance Modeling and Workload Mapping. In Proceedings of the 9th Conference on Computing Frontiers, Caligari, Italy, 15–17 May 2012; pp. 193–202.
10. Diener, M.; Cruz, E.; Navaux, P.; Busse, A.; Heiβ, H. kMAF: Automatic Kernel-Level Management of Thread and Data Affinity. In Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques (PACT), Edmonton, AB, Canada, 23–27 August 2014; pp. 277–288.
11. Ding, W.; Zhang, Y.; Kandemir, M.; Srinivas, J.; Yedlapalli, P. Locality-Aware Mapping and Scheduling for Multicores. In Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO), Shenzhen, China, 23–27 February 2013; pp. 1–12.
12. Cruz, E.; Diener, M.; Alves, M.; Navaux, P.O.A. Dynamic Thread Mapping of Shared Memory Applications by Exploiting Cache Coherence Protocols. *J. Parallel Distrib. Comput.* **2014**, *74*, 2215–2228. [CrossRef]
13. Tousimojarad, A.; Vanderbauwhede, W. An efficient thread mapping strategy for multiprogramming on manycore processors. *Parallel Comput.* **2013**, *25*, 63–71.
14. Marongiu, A.; Burgio, P.; Benini, L. Supporting OpenMP on a multi-cluster embedded MPSoC. *Microprocess. Microsyst.* **2011**, *35*, 668–682. [CrossRef]
15. Poovey, J.A.; Rosier, M.C.; Conte, T.M. *Pattern-Aware Dynamic Thread Mapping Mechanisms for Asymmetric Manycore Architectures*; Technical Report; Georgia Institute of Technology: Atlanta, GA, USA, 2011.
16. Weaver, V.; Johnson, M.; Kasichayanula, K.; Ralph, J. Measuring Energy and Power with PAPI. In Proceedings of the IEEE International Conference on Parallel Processing Workshops, Pittsburgh, PA, USA, 2012; pp. 262–268.
17. Terpstra, D.; Jagode, H.; You, H.; Dongarra, J. Collecting performance data with PAPI-C. In *Tools for High Performance Computing*; Springer: Berlin/Heidelberg, Germany, 2010; pp. 157–173.
18. Niu, Q.; Dinan, J.; Lu, Q.; Sadayappan, P. PARDA: A Fast Parallel Reuse Distance Analysis Algorithm. In Proceedings of the IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS), Shanghai, China, 21–25 May 2012; pp. 1284–1294.

19. Ju, T.; Zhu, Z.; Wang, Y.; Dong, X. Thread Mapping and Parallel Optimization for MIC Heterogeneous Parallel Systems. In Proceedings of the 14th International Conference on Algorithms and Architectures for Parallel Processing, Dalian, China, 24–27 August 2014; pp. 300–311.
20. Xiang, X.; Ding, C.; Luo, H.; Bao, B. HOTL: A Higher Order Theory of Locality. In Proceedings of the 18th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Houston, TX, USA, 16–20 March 2013; pp. 343–356.
21. Bach, M.M.; Charney, M.; Cohn, R.; Demikhovsky, E.; Devor, T. Analyzing parallel programs with pin. *IEEE Comput.* **2010**, *43*, 34–41. [CrossRef]
22. Schuff, D.L.; Kulkarni, M.; Pai, V.S. Accelerating Multicore Reuse Distance Analysis with Sampling and Parallelization. In Proceedings of the 19th International conference on Parallel Architectures and Compilation Techniques (PACT), New York, NY, USA, 11–15 September 2010; pp. 53–64.
23. Cochran, R.; Hankendi, C.; Coskun, A.; Reda, S. Pack & Cap: Adaptive DVFS and Thread Packing under Power Caps. In Proceedings of the 44th Annual International Symposium on Microarchitecture, MICRO, Porto Alegre, RS, Brazil, 4–7 December 2011; pp. 175–185.
24. Rodrigues, R.; Koren, I.; Kundu, S. Does the Sharing of Execution Units Improve Performance/Power of Multicores? *ACM Trans. Embed. Comput. Syst.* **2015**, *14*, 17. [CrossRef]