2009 International Conference on Parallel Processing

A Partition-Merge based Cache-Conscious Parallel Sorting Algorithm for CMP with Shared Cache*

Song Hao[1], Zhihui Du[1+] ,David A. Bader [2], Yin Ye[1]
[1]*Tsinghua National Laboratory for Information Science and Technology*
Department of Computer Science and Technology, Tsinghua University, 100084, Beijing, China
[2]*College of Computing, Georgia Institute of Technology, Atlanta, GA, 30332, USA*
[+]Corresponding Author's Email: duzh@tsinghua.edu.cn

Abstract **To explore chip-level parallelism, the PSC (Parallel Shared Cache) model is provided in this paper to describe high performance shared cache of Chip Multi-Processors (CMP). Then for a specific application, parallel sorting, a cache-conscious parallel algorithm, PMCC (Partition-Merge based Cache-Conscious) is designed based on the PSC model. The PMCC algorithm consists of two steps: the partition-based in-cache sorting and merge-based k-way merge sorting. In the first stage, PMCC first divides the input dataset into multiple blocks so that each block can fit into the shared L2 cache, and then employs multiple cores to perform parallel cache sorting to generate sorted blocks. In the second stage, PMCC first selects an optimized parameter k which can not only improve the parallelism but also reduce the cache missing rate, then performs a k-way merge sorting to merge all the sorted blocks. The I/O complexity of the in-cache sorting step and k-way merge step are analyzed in detail. The simulation results show that the PSC based PMCC algorithm can out-performance the latest PEM based cache-conscious algorithm and the scalability of PMCC is also discussed. The low I/O complexity, high parallelism and the high scalability of PMCC can take advantage of CMP to improve its performance significantly and deal with large scale problem efficiently.**

Keywords Parallel Sorting; Cache-conscious Algorithm; Chip Multi-Processors (CMP)

I. INTRODUCTION

The "memory wall" is the growing disparity of speed between CPU and the off-chip memory. CPU speed improves much faster than the memory speed. Given this trend, it was expected that memory latency would become an overwhelming bottleneck in computer performance. At present, the improvements of CPU frequency have been slowed due to memory wall and some other physical barriers. CPU manufacturers have turned to more efficient architectures to improve CPU performance. One of the most important innovations in the architecture is Chip Multi-Processors (CMP), which integrate multiple processor cores in a single chip.

In order to overcome the memory wall, we first provided a CMP model, the *PSC* model in this paper, in which the L_2 on-chip caches are shared by the processor cores in some degree. The *PSC* model can take advantage of the high performance shared cache and explore chip-level parallelism, which will reduce the *I/O complexity (main memory access times)* of an application. Then for a specific problem, the parallel sorting, we designed a cache-conscious parallel algorithm based on the *PSC* model.

Parallel sorting has been studied extensively during the past thirty years [7, 8, 9]. However most of the previous algorithms are implemented on SMP machines. With the emergence of CMP, a new kind of parallel platform with different characteristics has become reality. The design of parallel sorting algorithms which can take advantage of the characteristics of CMP is now a challenging task. Cache-efficient algorithms reduce the execution time of an application by exploiting data parallelisms inherent in the transfer of useful data blocks between adjacent memory levels. By increasing locality in their memory access patterns, these algorithms try to keep the block transfer times small. And the *I/O complexity* of an algorithm is the number of *I/Os* (i.e. block transfers) performed between main memory and on-chip cache. As we introduced, the growing disparity of speed between CPU and main memory makes it more disastrous to access main memory for CPU performance. So reducing the *I/O complexity* is an effective method to overcome the memory wall.

We explore the effect of shared cache in CMP on sorting performance and demonstrate a cache-conscious parallel sorting algorithm with low parallel *I/O complexity*, PMCC (Partition-Merge based Cache-Conscious). The PMCC approach consists of two main steps, the partition-based in-cache sorting and merge-based k-way merge sorting. This algorithm mainly has two advantages relative to other parallel sorting algorithms: (1) The PMCC algorithm can make full use of the on-chip shared cache. In the partition-based in-cache sorting stage, the input dataset is divided into blocks. The size of each block can fit into the shared cache. Then each block can be loaded into the cache and sorted without any memory access. In the merge-based k-way merge sorting stage, the algorithm can calculate the optimized value of k to make the shared cache to be fulfilled. Both of the two stages can make full use of the capacity of the shared cache; (2) The PMCC algorithm can explore the on-chip parallelism and reduce the *I/O complexity* significantly. In the partition-based in-cache sorting stage, each block is sorted in parallel by all the processor cores, and in the merge-based k-way merge sorting stage, each processor core performs a k-way merge sorting until the number of sorted subsets is less than the number of processor cores. During the two stages, because of the existence of the shared cache, most of the synchronization operations and data exchanges among the processor cores can be performed without accessing main memory, which can

* This paper is partly supported by National Natural Science Foundation of China (No. 60773148 and No.60503039), Beijing Natural Science Foundation (No. 4082016), China's National Fundamental Research 973 Program (No. 2004CB217903), NSF Grants CNS-0614915 and an IBM Shared University Research (SUR) award.

0190-3918/09 $26.00 © 2009 IEEE
DOI 10.1109/ICPP.2009.26

IEEE computer society

reduce the *I/O complexity* significantly.

The notations used throughout this paper are summarized in Table 1 for clarity.

TABLE I. SUMMARY OF NOTATIONS

NOTATION	DESCRIPTION
N	number of elements in the input dataset
n	number of elements in each data block which can fit into the shared cache
C_1, C_2	capacity of L_1 and the shared L_2 cache
B	width of each cache line
p	number of processor cores
D	number of processor cores to perform a SBPS algorithm
SD	sharing degree, number of processor cores sharing the same L_2 cache
PD	parallel degree of I/O operations
e_i	the i th input element
$list_i$	the i th sorted list
$Core_i$	the i th processor core

The rest of this paper is organized as follows. Section 2 will introduce the background and the related work. We will model the CMP with a shared cache in Section 3. In Section 4, the proposed algorithm is introduced and the complexity analysis of our algorithm will be presented in Section 5. The simulation experiments are presented in Section 6 and we conclude in Section 7.

II. BACKGROUND AND RELATED WORK

2.1 Parallel sorting

Significant research has been performed in the area of parallel sorting. But most of these studies are aimed at specific problems or machines. Generally, most parallel sorts suitable for multiprocessor computers can be placed into one of two general categories: merge-based sorts and partition-based sorts. Merge-based sorts consist of multiple merge stages across processors. And partition-based sorts consist of two phases: partitioning the data set into smaller subsets such that all elements in one subset are no greater than any element in another, and sorting each subset in parallel. The performance of partition-based sorts primarily depends on how well the data can be evenly partitioned into smaller ordered subsets. Shi et al. [7] proposed a partitioning method based on regular sampling. Helman et al. [8, 12] propose two algorithms respectively for partitioning the input dataset. These parallel sorting algorithms are aiming at traditional SMP machines, but our PMCC algorithm is for CMP machines with shared cache. The PMCC algorithm can take advantage of the shared on-chip cache but the previous algorithms just employ the off-chip DRAM.

2.2 Cache-conscious sorting

The influence of caches on sorting performance has been discussed in several papers. In [2], M. Goodrich et al. presented a private-cache CMP model and studied two fundamental sorting algorithms in this model. And in [1], V. Ramachandran et al. considered three types of caching systems for CMPs, and derived results for three classes of problems in these systems. These two documents only covered two extreme kinds of cache structures, private cache and full-shared cache. And the algorithms and applications they examined were limited in these two cache structures. But in our *PSC* model, we introduced the concept *Sharing Degree (SD)*, which indicates the number of processor cores sharing the same L_2 cache. For different applications, we can verify the specific values of *SD* to get the best performances.

Besides, LaMarca et al. [6] restructures some classic sorting algorithms by optimizing their cache behavior. Rahman et al. [5] analyzed the cache effect on distribution sorting. Wickremesinghe et al. [4] explored the effect of memory system features on sorting performance and proposed a new cache-conscious sorting algorithm. However, these researches mainly focus on traditional single core processor and serial sorting algorithms, and our algorithm is a parallel sorting algorithm and aims at multi-core processors.

Inoue et al. [3] proposed a parallel sorting algorithm, AA-Sort, for shared-memory multi-processors, which can take advantage of SIMD instructions. AA-Sort can exploit the feature of processor cores' private memory, L1 cache or local storage of core, and divide the input data set into subsets whose size can fit into the private memory. Then the cores sort them with SIMD instructions in parallel and merge all the subsets with an improved merge sort algorithm. The difference between PMCC and AA-Sort is that, AA-Sort aims at the private cache (or other kind of local storage) and sorts each subset with serial algorithms, but PMCC makes use of the shared cache and performs parallel algorithms to sort each subsets.

III. MODEL THE CMP WITH SHARED CACHE

Current CMP chips typically include two or three levels of cache memory arranged in a hierarchy, and the sizes of on-chip L_2 and L_3 cache memories are expected to continue increasing. For example, the Alpha 21364 contains a 1.75MB L_2 cache, and Intel Itanium2 contains 3MB of on-chip L_3 cache, and the IBM Power6 contains 8MB of L_2 cache.

In this section we propose a CMP model which can describe the characteristics of CMPs with on-chip shared cache. We call it the *Parallel Shared Cache (PSC)* model to underline the parallelization in accessing the shared cache. The *PSC* model is a computational model with p processor cores and a three-level memory hierarchy. The memory hierarchy consists of the main memory shared by all the processor cores and the two-level on-chip cache, in which the L_1 cache is private for each processor core and the L_2 cache is shared by a fixed number of processor cores. The size of each L_1 cache is C_1 and size of each L_2 cache is C_2. Both the L_1 and L_2 caches are partitioned into cache lines of size B. So the data is transferred between main memory and L_2 cache in cache lines of size B.

We define the *sharing degree (SD)* as number of processor cores sharing the same L_2 cache. When the value of *SD* is 1, L_2 cache becomes the private cache of each processor core, and if *SD* equals to p, L_2 cache is shared by all the processor cores. Figure 1 shows a *PSC* example with $SD = 2$. And we defined the *parallel degree (PD)* as the parallel degree of *I/O* operations, which indicates the cache line number that can be transferred between each L_2 cache and the main memory in one *I/O* operation. We adopt the same method with *PEM* model [2] to measure *PSC* model's *I/O* complexity. That is, to

measure the number of parallel line transfers between the main memory and the L_2 cache, and each time p/SD (number of L_2 caches) lines can be read from/write into the main memory. For example, an algorithm with each of the p processor cores simultaneously reading one (different) line from the main memory would have an *I/O* complexity of *O(SD/PD)*, not *O (p)*.

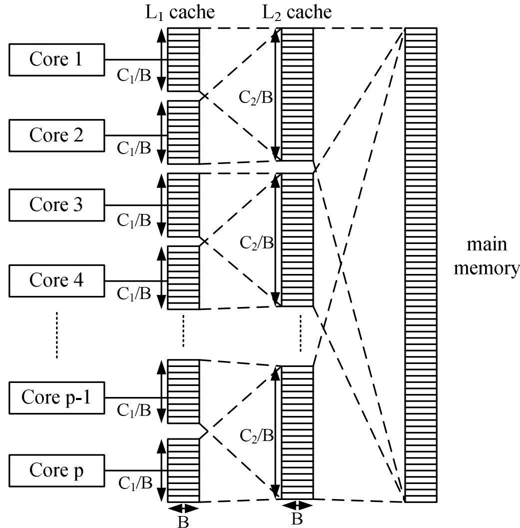


Figure 1. PSC Model (SD = 2)

In *PSC* model, we disregard on-chip networks and assume that all the communications among the processor cores is conducted by writing/reading to/from the shared memory. This shared memory can be main memory or shared cache, and it's easy to find that communications through the shared cache won't increase I/O complexity. When it comes to the main memory, the *I/O* complexity of communications depends on whether the multiple processor cores will access the same block. If processor cores access distinct blocks, the *I/O* complexity is *O(1)*, and if they access the same block, according to [2], we can differentiate three variations:

- Concurrent Read, Concurrent Write (CRCW)
- Concurrent Read, Exclusive Write (CREW)
- Exclusive Read, Exclusive Write (EREW)

And we only consider the CREW model and leave the other two models for future work. In the case of CREW the p writes from p processor cores to the same block result in p I/Os. But this can be improved to *O(log p)* with some auxiliary blocks.

Algorithm 1: in_cache_sort (array, N, p, C_2, SD)

\# The **in_cache_sort** function sorts each block with a
\# Sample-Based Parallel Sorting algorithm
\# array[0:N-1] : array to be sorted, N : size of array,
\# p : number of processor cores, SD : sharing degree,
\# C_2 : capacity of each L_2 cache
1: begin:
2: $Round = \lceil (N \times SD)/(C_2 \times p) \rceil$
3: $GroupNum = p/SD$;
4: **for** i = 0 to $Round$-1 **do**
5: **for each** *group j* **in parallel do**
\# j [0,$GroupNum$ - 1]
6: $blockBase = i \times C_2 \times p/SD + j \times C_2$
7: **Read** array[$blockBase$: $blockBase + C_2$ -1] (one block) into L_2 cache
8: *SBPS* (array[*blockBase*] , C_2, SD)
\# sort one block of C_2 elements with
\# Sample-Based Parallel Sorting algorithm
\# by SD processor cores
9: **end for**
10: **end for**
11:end in_cache_sort

IV. PMCC ALGORITHM

In this section, we present our new PMCC parallel sorting algorithm. PMCC consists of two sub-algorithms, the partition-based in-cache sorting and merge-based k-way merge sorting. The overall PMCC executes the following phases using the two algorithms: (1) Divide all of the data into blocks that fit into the L_2 cache; Sort each block with the in-cache sorting algorithm; (2) Merge the sorted blocks with the merge sorting algorithm.

4.1 In-cache sorting algorithm

We adopt the Sample-Based Parallel Sorting (*SBPS*) to sort each block (Line 8). The idea behind *SBPS* algorithm is to find a set of d (= SD) -1 splitters to partition the input data set containing n=C_2 elements into d groups indexed from 1 up to d such that every element in the i-th group is less than or equal to each of the elements in the (i+1)-th group. The d groups can be turned over to the correspondingly indexed processor core, after which the n elements will be arranged in sorted order.

Assume that there are n elements in each block which can be indexed from 0 up to n-1, and without loss of generality, we assume that d divides n evenly. Generally, the procedure of Sampling-Based Parallel Sorting can be described as follows (as shown in Figure 2):

Step 1. Divide the block into d subsets. Each processor core $core_i$ ($0 \leq i \leq d-1$) sorts one subset composed of n/d elements $\{e_{i\times(n/d)}, e_{i\times(n/d)+1}, ..., e_{(i+1)\times(n/d)-1}\}$ using a serial sorting algorithm;

Step 2. $core_0$ selects a sample set $\{y_1, y_2, ..., y_{d-1}\}$ which is used as the pivots to split each subset, and then broadcast the sample set;

Step 3. Each processor core splits its subset into d lists with the sample set, where $\forall e \in list_{i,j}$, it meets $y_{j-1} < e < y_j$;

Step 4. For each $core_i$, it performs a mergeSort algorithm to merge all the i-th sorted list $\{list_{0,i}, list_{1,i}, ..., list_{d-1,i}\}$.

The ability to partition the data evenly into ordered subsets is essential for partition-based sorts. If the distribution statistics of the data are known, it becomes easy to divide the data into equal-sized subsets. Unfortunately, in general, we do not know the data distribution. To overcome this difficulty, most partition-based sorts apply sampling-based method. That is, to select a sample of the input dataset and partition the data according to the distribution of the sample. The effectiveness

of sampling depends largely on the distribution of the original data, the choice of proper sample size, and the way in which the sample is drawn.

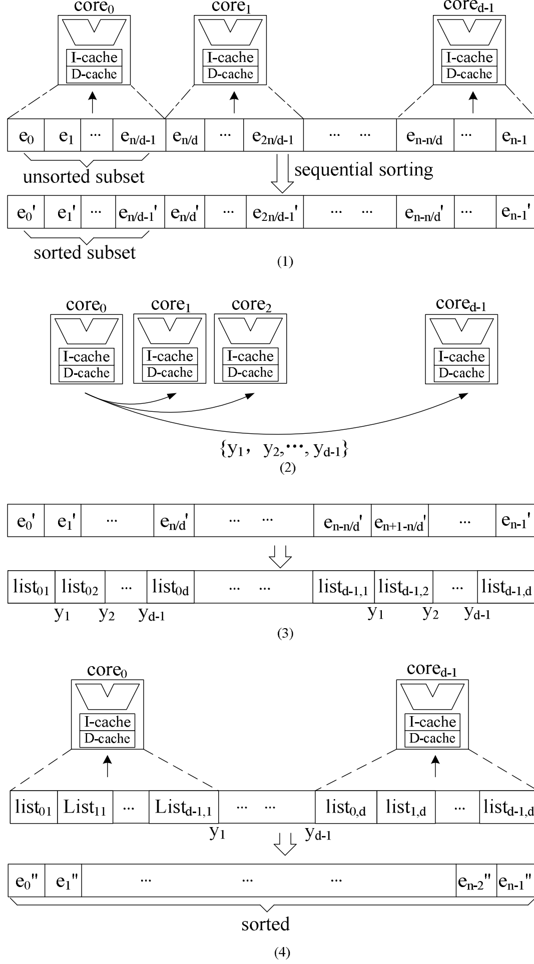


Figure 2. Sampling-based Parallel Sorting

In Step 2, we use the sample based method to select the pivots and split each ordered subset. Obviously, the more evenly we divide the input data, the higher the algorithm's performance is. So the sample should be able to reflect the distribution of input. A lot of work has been done on how to select the sample of input data. We compare the effectiveness of some typical sample-selection approaches including Random Sampling (RS) [9], Parallel Sorting by Regular Sampling(PSRS) [7], and Deterministic Regular Sampling (DRS) [8]. We introduce the three approaches briefly below, and the implementation details can be found in [7, 8, 12].

In Random Sampling (RS), elements in each subset are assigned into d buckets randomly. Then the processor cores exchange data by sending the contents of bucket j to $Core_j$, which will form d new subsets. After each processor core sorts its subset, $Core_0$ selects the pivots which can divide its subset evenly and broadcasts the pivots as splitters of Step3.

In PSRS, after sorting the subsets in Step1, each processor core chooses d-1 samples which are evenly spaced throughout the subset and sends them to $Core_0$. After gathering samples from other cores, $Core_0$ sorts them into an ordered list with a serial sorting algorithm and choose pivots from the list, by which the list can be divided evenly. Then the pivots are broadcasted as splitters of Step3.

In DRS, each processor core divides the subset into d subsequences after Step 1. Then the cores exchange the subsequences with each other. From each of the received subsequences, $Core_0$ selects some samples and merges them into an ordered list. And the pivots will be selected from the list just like PSRS.

We don't compare the performance of the three approaches here, but focus on the *I/O* complexity of each approach, which is essential to overcome the memory wall.

4.2 Merge sorting algorithm

The input dataset is divided into N/C_2 sorted equal-size subsets in the in-cache sorting stage. And in the merge sorting stage, we need to merge all the subset into one sequential data set.

Algorithm 2: *k*_merge_sort (array, *N*, *p*, C_2, *SD*)

```
# The k_merge_sort function merges N/C2 ordered blocks
# into one block, 2≤k≤C2/B
# array[0:N-1] : array to be sorted, N : size of array,
# p : number of processor cores, SD : sharing degree,
# C2 : capacity of each L2 cache
1: begin:
2:   UsedCoreNum = min {SD, ⌊C2 / (B×k)⌋}
3:   Round = (N/C2)/(UsedCoreNum×k×p/SD)
4:   BlockSize = C2
5:   for core 0 to UsedCoreNum-1 in each group g
        in parallel do
6:      coreIndex = getIntraGroupIndex( )
        # get each core's intra-group index
7:      for i = 0 to Round-1 do
8:         blockBase = (UsedCoreNum×(p×i/SD+g) +
                    coreIndex) ×k×BlockSize
9:         if blockBase < N-1 do
10:           from array[blockBase:
                         blockBase+k×BlockSize]
              read the first line (B elements) of each block into
              L2 cache
11:           while not all the k blocks are merged do
12:              merge the k lines until one of them is finished
13:              read the next line from the original block
14:              continue
15:           end while
16:        end if
17:     end for
18:     Round = Round / k
19:     BlockSize = BlockSize ×k
20:     if BlockSize ≥ N do
21:        return
22:     end if
23:  end for
24:end k_merge_sort
```

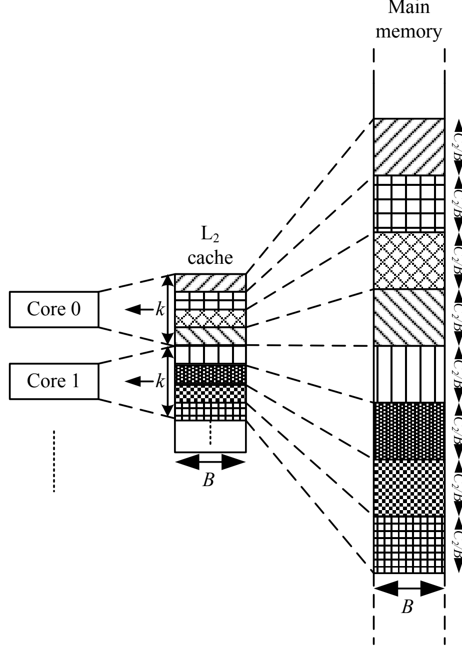


Figure 3. k-way merge sorting (k = 4)

The most popular algorithm for merging multiple subsets into one is 2-way MergeSort. Its time complexity is $O(N \log_2 N)$ and space complexity can be $O(1)$. However, in 2-way MergeSort, each element needs to be accessed $\log_2 N$ times. Because in each recursion round the element is accessed at least once. From the view of cache, if the size of input is larger than capacity of cache, each element will cause $\log_2 N / B$ cache misses, which will significantly increase the *I/O* complexity.

The pseudo-code of the *PSC* *k*-way merge sorting algorithm is shown in Algorithm 2. First, given a value of *k*, we need to decide how many processor cores can be used to perform the merge sorting in each group. We define *UsedCoreNum* as $\min\{SD, \lfloor C_2/(B \times k) \rfloor\}$ (Line 2) to make sure that at least one line of the block to be merged is kept in the L_2 cache during the merging operation (as shown in figure 3). Then in each round $UsedCoreNum \times k \times p/SD$ blocks are merged in parallel into $UsedCoreNum \times p/SD$ ones, so after $(N/C_2)/(UsedCoreNum \times k \times p/SD)$ rounds, the N/C_2 blocks are merged into $N/C_2 \times k$ ones. Then we recursively perform this merging procedure until only one block is left.

V. I/O COMPLEXITY ANALYSIS

5.1 In-cache sorting algorithm

In this part, we discuss the *I/O* complexity of in-cache sorting algorithms with different *SBPS* algorithms respectively. We don't focus on the implementation details of each *SBPS* algorithms, but focus on the *I/O* complexity only.

THEOREM 1. *If the input array is located in contiguous main memory, the I/O complexity of PSRS algorithm in PSC model is* $O(C_2/B \cdot PD + d^2/B \cdot PD)$.

PROOF. The pseudo-code of PSRS algorithm can be found in [7]. The *PSC* PSRS algorithm consists of four phases: (1). each of the *d (=SD)* processor cores performs a serial sorting on its subset in parallel; (2). each core select a sample vector and send it to $core_0$; (3). $core_0$ selects a splitter vector from the sample vectors and broadcasts it; (4). each core splits it subsets and performs a merge sorting in parallel.

Before the serial sorting, the block needs to be read into the L_2 cache. Since the input array is located in contiguous main memory. The *I/O* complexity of reading a block is $O(C_2/(B \cdot PD))$. Then the serial sorting doesn't contribute more I/O complexity. In the second phase, each of the *d* processor cores writes a sample vector of *d-1* elements into a contiguous memory. It needs $d(d-1)/B \cdot PD$ *I/O*s in *PSC* model. In the third phase, $Core_0$ sorts the samples and selects *d-1* elements as the splitter vector. Since we only consider CREW model in this paper, the *d* processor cores can read the splitter vector concurrently. So this phase contributes no *I/O* complexity. In the last phase, each of the *d* processor cores performs a *d*-way merge sorting. In *PSC* model, the merge operation and data movement intra the block can be accomplished with in the L_2 cache. So it doesn't contribute *I/O* complexity either.

So in total, the *PSC* PSRS algorithm's *I/O* complexity is $O(C_2/B \cdot PD + d(d-1)/B \cdot PD) = O(C_2/B \cdot PD + d^2/B \cdot PD)$.

□

COROLLARY 1. *If the input array is located in contiguous main memory, the I/O complexity of PSRS-based in-cache sorting algorithm is*

$$O(N \cdot (SD + SD^3/C_2)/(p \cdot B \cdot PD))$$

PROOF. If the *SBPS* of the in-cache sorting algorithm is implemented with PSRS algorithm, in each round *p/SD* blocks are sorted. And there are totally $(N/C_2)/(p/SD)$ rounds. So the *I/O* complexity of PSRS in-cache sorting algorithm is

$$O(((N/C_2)/(p/SD)) \cdot (C_2/B \cdot PD + d^2/B \cdot PD))$$
$$= O(N \cdot SD \cdot (1 + d^2/C_2)/(p \cdot B \cdot PD))$$

Since $d = SD$,

$$O(N \cdot SD \cdot (1 + d^2/C_2)/(p \cdot B \cdot PD))$$
$$= O(N \cdot (SD + SD^3/C_2)/(p \cdot B \cdot PD))$$ □

THEOREM 2. *If the input array is located in contiguous main memory, the I/O complexity of RS algorithm in PSC model is* $O(C_2/B \cdot PB)$.

PROOF. The difference between RS and PSRS is that the cores don't gather the sample vectors in the second phase of PSRS, but do a data exchange before the serial sorting. So RS algorithm consists the following four phase: (1). each core assigns its subset into *d* buckets randomly and sends the elements in *bucket j* to $core_j$; (2). each of the *d* cores sorts the received elements in phase one; (3). $core_0$ select a splitter vector and broadcasts it; (4). each core performs a *d*-way merge sorting.

In the first phase, one additional auxiliary block is needed to place the reassigned block. Since each element is referenced only once, the assign operation counts for $2 \times C_2/B \cdot PD$ *I/O*s. Then after the reassigned block is read into the L_2 cache, no more *I/O*s are needed to exchange data.

The rest phases are similar to PSRS except that the d cores don't gather the sample vectors in RS. So the *I/O* complexity of RS in PSC model is $O(2\times C_2/B) = O(C_2/B\cdot PD)$. □

COROLLARY 2. *If the input array is located in contiguous main memory, the I/O complexity of RS-based in-cache sorting algorithm is* $O(N\cdot SD/(p\cdot B\cdot PD))$

PROOF. The proof the COROLLARY 2 is the same to COROLLARY 1. The *I/O* complexity of running $(N/C_2)/(p/SD)$ rounds of RS algorithm is

$$(N/C_2)/(p/SD)\times O(C_2/B) = O(N\cdot SD/(p\cdot B\cdot PD)).$$ □

THEOREM 3. *If the input array is located in contiguous main memory, the I/O complexity of DRS in PSC model is* $O(C_2/B\cdot PD)$.

PROOF. The *PSC* DRS algorithm consists of four phases: (1). each of the d cores sorts its subset with a serial sorting algorithm in parallel; (2). each core sends every i-th ($0\le i\le d-1$) element of its subset to $core_i$; (3). $core_0$ selects a splitter vector and broadcasts it; (4). each core performs a merge sorting algorithms.

In phase 1, the block is read into L_2 cache with the *I/O* complexity $O(C_2/B\cdot PD)$ before the serial sorting. The phase 2 needs an auxiliary block to reorganize the input block, so the I/O complexity of data exchange is $O(C_2/B\cdot PD)$. The rest phases are similar to THEOREM 2 and contribute no *I/O* complexity. So in total the *I/O* complexity of DRS in-cache sorting algorithm is $O(2\times C_2/B\cdot PD) = O(C_2/B\cdot PD)$. □

COROLLARY 3. *If the input array is located in contiguous main memory, the I/O complexity of DRS-based in-cache sorting algorithm is* $O(N\cdot SD/(p\cdot B\cdot PD))$

PROOF. The proof the COROLLARY 3 is the same to COROLLARY 1. The *I/O* complexity of running $(N/C_2)/(p/SD)$ rounds of DRS algorithm is

$$(N/C_2)/(p/SD)\times O(C_2/B\cdot PD) = O(N\cdot SD/(p\cdot B\cdot PD))$$ □

We can find that the *I/O* complexity of DRS-based and RS-based in-cache sorting algorithm is lower than the PSRS-based one. In our simulation experiments, we applied the DRS-based in-cache sorting algorithm.

5.2 Merge sorting algorithm

THEOREM 4. *If the input array is located in contiguous main memory, the I/O complexity of the k-way merge sorting algorithm in PSC model is* $O((N\cdot SD/PD\cdot p\cdot B)\log_k(N/C_2))$.

PROOF. In the k-way merge sorting algorithm of PMCC, the depth of the k-way merge tree is $\log_k(N/C_2)$. That means there are totally $\log_k(N/C_2)$ rounds of k-way merge sorting to merge N/C_2 blocks into one.

In each round, each of the N input elements is read into L_2 cache at least once. And in *PSC* model each time $PD\cdot p/SD$ lines can be read into L_2 cache. So the *I/O* complexity of each round is $O((N/B)/(PD\cdot p/SD))$. So in total the $\log_k(N/C_2)$ rounds of merge sorting counts for the *I/O* complexity:

$$\log_k(N/C_2)O((N/B)/(PD\cdot p/SD)) = O((N\cdot SD/PD\cdot p\cdot B)\log_k(N/C_2))$$ □

According to the analysis above, the total *I/O* complexity of PMCC is

$$O(N\cdot SD/(p\cdot B\cdot PD)+N\cdot SD\cdot\log_k(N/C_2)/(PD\cdot p\cdot B))$$
$$=O(N\cdot SD\cdot(1+\log_k(N/C_2))/(PD\cdot p\cdot B))$$
$$=O(\frac{N\cdot SD}{PD\cdot p\cdot B}\log_k\frac{N}{C_2}) \quad (1)$$

According to THEOREM 4, for a given implementation of the *PSC* model, the *I/O* complexity of the merge sorting algorithm is decided by the way number k. By increasing k, the I/O complexity can be reduced. Since $2\le k\le C_2/B$, the low bound of total *I/O* complexity is $O(\frac{N\cdot SD}{PD\cdot p\cdot B}\log_{C_2/B}\frac{N}{C_2})$.

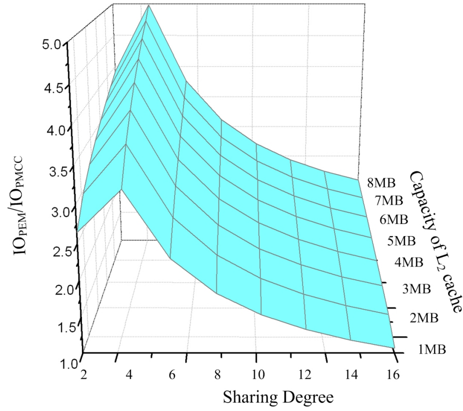
VI. SIMULATION EXPERIMENTS

In order to examine the *I/O* complexity, we simulated the execution procedure of PMCC and the latest cache-conscious parallel sorting algorithm, *PEM* Distribution sort in the CMP environments.

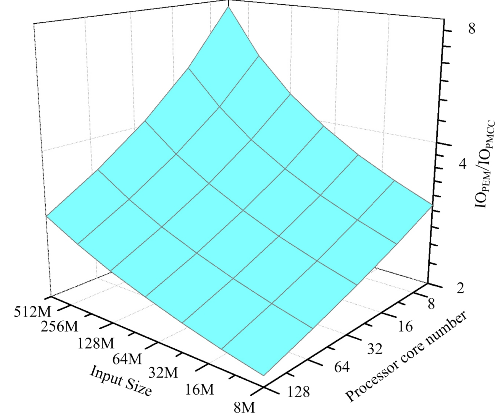
6.1 Comparison of I/O Complexity

We explored the cache behavior of PMCC and *PEM* Distribution sort (*PEM-D*) when executed in CMP environments and compared the I/O times of the two algorithms. Figure 4 shows the ratio of *PEM-D*'s I/O times to PMCC's.

As we can see in figure 4 (a), for most observations, *PEM-D* requires more I/Os than PMCC. In the 16-core CMP environment, a sharing degree of four gives PMCC the greatest advantage over *PEM-D* when the sharing degree equals to the parallel degree. The value of IO_{PEM-D}/IO_{PMCC} is over 3.2, which means the *I/O* times of PMCC is only about 31.3% of *PEM-D*'s. On another side, with the growth of L_2 cache's capacity, the value of IO_{PEM-D}/IO_{PMCC} doesn't change obviously. This means that PMCC has similar scalability with *PEM-D* with the growth of L_2 cache's capacity.

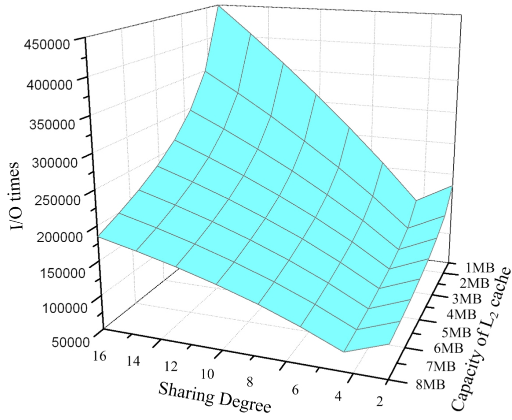


(a) p = 16, N = 64M, PD = 4

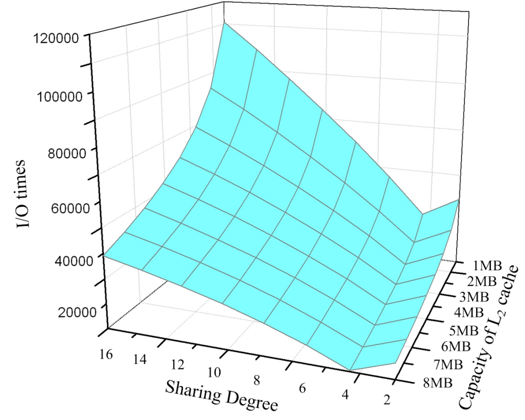


(b) SD = 4, C_2= 4MB/SD

Figure 4. Comparison of I/O times between PMCC and PEM Distribution sort (PEM-D)

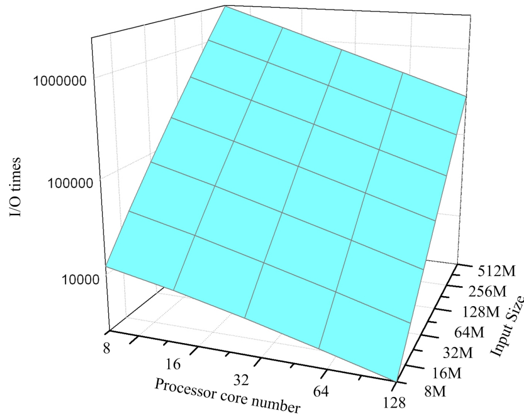


(a) p = 16, N = 64M, PD = 4

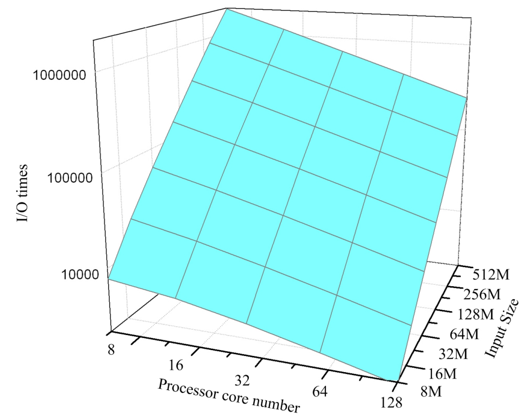


(b) p = 32, N = 32M, PD = 4

Figure 5. Scalability of PMCC with the growth of L2 cache's capacity



(a)SD = 4, C_2 = 4MB/SD



(b) SD = 8, C_2 = 8MB/SD

Figure 6. Scalability of PMCC with the growth of processor core number and input size

In figure 4 (b), we can see that, first, with the growth of processor core number, the value of $IO_{PEM\text{-}D}/IO_{PMCC}$ becomes smaller. This means that, given the sharing degree and L_2 cache's capacity, a smaller processor core number makes PMCC's advantage more obvious. Second, with the input size becoming bigger, the advantage of PMCC becomes greater and when the input size is 512M and processor core number is 8, the value of $IO_{PEM\text{-}D}/IO_{PMCC}$ is 6.75, which means the *I/O* times of PMCC is only about 14.8% of *PEM-D*'s.

6.2 Scalability

We explored the scalability of PMCC by increasing the capacity of L_2 cache, number of processor cores and the input size in the simulation.

We can see in figure 5 that with the growth of L_2 cache's capacity, the I/O times are reduced significantly. In the in-cache sorting stage of PMCC, the input is divided into small sorted blocks, which will be merged in the next stage. By increasing the capacity of L_2 cache, we can increase the block size and reduce the block number, which will reduce the total rounds of the merge sorting and in turn reduce the times of *I/O*s. And we can also see that in both of the 16-core and 32-core CMP environments, a sharing degree of four gives the PMCC least I/O times when the sharing degree equals to the parallel degree.

Figure 6 shows the reduction of *I/O*s with growths of processor core numbers and the *I/O* growth mode with the growth of input size. As we discussed in section 6.1, given a *SD*, the growth of core number will increase the parallelism degree of the *I/O* operations. So when the core number increases, the *I/O* times decreases accordingly. Figure 6 also shows the processor core number grows in a *log* scale in our simulation and the *I/O* times reduce in a *log* scale too. And the overall tendencies obey a linear law. In figure 6, we can also see that with the growth of input size, the *I/O* time grows in a linear law too.

VII. CONCLUSION

Chip Multi-Processor has become a new platform for parallel computing. In this paper, we explored the characteristics of CMP and provided a computational CMP model, *PSC*, which consists of *p* processor cores and a three-level memory hierarchy. Based on the *PSC* model, we designed a new parallel sorting algorithm, PMCC, which consists of two parts, the in-cache sorting and merge sorting algorithm. PMCC first divides all of the data into blocks that fit into the shared cache and sorts each block with the in-cache sorting algorithm. Then the sorted blocks are merged together with merge sorting.

PMCC can take advantage of shared L2 cache and minimize total number of data transfers. According to our analysis, the PSRS in-cache sorting algorithm has an *I/O* complexity of $O(N \cdot (SD + SD^3 / C_2)/(PD \cdot p \cdot B))$ and the RS and DRS in-cache sorting algorithm's I/O complexity is $O(N \cdot SD/(PD \cdot p \cdot B))$. The I/O complexity of *k*-way merge sorting algorithm is $O((N \cdot SD / PD \cdot p \cdot B)\log_k(N/C_2))$. And the low bound of the *I/O* complexity of PMCC algorithm is $O(\frac{N \cdot SD}{PD \cdot p \cdot B}\log_{C_2/B}\frac{N}{C_2})$.

According to the simulation experiments, PMCC can significantly reduce the I/O complexity relative to *PEM* Distribution sort and has considerable scalability.

REFERENCES

[1] R. Chowdhury, V. Ramachandran. "Cache-efficient Dynamic Programming Algorithms for Multicores". In the 20th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2008), Munich, Germany, 2008, Page(s): 207-216.

[2] L. Madalgo, M. Goodrich, M. Nelson. "Fundamental Parallel Algorithms for Private-Cache Chip Multiprocessors". In the 20th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2008), Munich, Germany, 2008, Page(s): 197-206

[3] H. Inoue, T. Moriyama, H. Komatsu, T. Nakatani. "AA-Sort: A New Parallel Sorting Algorithm for Multi-Core SIMD Processors". 16th International Conference on Parallel Architecture and Compilation Techniques (PACT2007). Page(s): 189-198.

[4] R. Wickremesinghe, L. Arge, J. S. Chase, J. S. Vitter. "Efficient sorting using registers and caches". Lecture notes in Computer Science. Volume 1982/2001, Page(s): 51-62.

[5] N. Rahman, R. Raman. "Analyzing Cache Effects in Distribution Sorting". ACM Journal of Experimental Algorithms, 2000. Volume: 5.

[6] A. LaMarca, R. E. Ladner. "The influence of caches on the performance of sorting". Journal of Algorithms archive, Volume 31 , Issue 1 (April 1999), Page(s): 370-379

[7] H. Shi, J. Schaeffer. "Parallel Sorting by Regular Sampling". Journal of *Parallel and Distributed Computing*, Volume 14 , Issue 4 (April 1992) Page(s): 361 – 372, ISSN:0743-7315

[8] D. R. Helman, J. JaJa, D. A. Bader. "A new deterministic parallel sorting algorithm with an experimental evaluation". ACM *Journal of Experimental Algorithmics (JEA)*, September 1998, Volume 3.

[9] D. R. Helman, D. A. Bader, J. JaJa. "A Randomized Parallel Sorting Algorithm with an Experimental Study". ACM Journal of Parallel and Distributed Computing , July 1998, Volume 52