A Tool to Detect Performance Problems of Multi-threaded Programs on NUMA Systems

Liang Zhu, Hai Jin, Xiaofei Liao
Service Computing Technology and System Lab
Cluster and Grid Computing Lab
School of Computer Science and Technology
Huazhong University of Science and Technology, Wuhan, 430074, China
Email: hjin@hust.edu.cn

***Abstract*—The multi-core architectures are nowadays characterized by *Non-Uniform Memory Access* (NUMA). Efficiently exploiting such architectures is extremely complicated for programmers. Multi-threaded programs may encounter high memory access latency if the mapping of data and computing is not considered carefully on such systems. Programmers need tools to detect performance problems if high memory access latency occurs. To address this need, we present a profiling tool called *LaProf*, which uses memory access latency information to detect performance problems on NUMA systems. This tool can be used to detect three performance problems of multi-threaded programs, which are: 1) data sharing. Shared data will cause remote memory access if threads which access the shared data are not allocated on the same node of NUMA systems; 2) shared resource contention. High memory access latency will influence the performance severely if contention happens on shared resources, such as last-level caches, inter-connect links and memory controllers; 3) remote access imbalance. The thread which has the most number of remote data access becomes the critical thread which lags down the overall performance of multi-threaded program. After the detection done by *LaProf*, using simple and general NUMA optimization techniques, the performance improvement for each problem is 88%, 32%, 99% respectively.**

I. Introduction

In modern multi-core systems, some memory is locally-joint to the processor, while the other memory is remotely-joint to the processor. This type of combination is known as *Non-Uniform Memory Access* (NUMA) architectures. It is faster for a CPU core to access the memory which is locally joint than the memory which is remotely joint. We regard the memory and threads which access the memory with uniform latency as a *memory field*. There are totally four *memory fields* on 2-node NUMA system (each node has 4 cores). Based on the values of Intel Nehalem architecture [1], the access latencies of four *memory fields* are as shown in Figure 1. The *memory fields* on local node has lower access latency than the *memory fields* on remote node. Systems with multiple *memory fields* are challenging to program efficiently. Without careful consideration, the data and threads which use the data will not be mapped just in the local *memory fields* which has lower memory access latency and higher bandwidth. Multi-threaded programs may experience significant performance losses when their memory accesses are distributed on remote *memory fields*. Based on a review of previous research work,

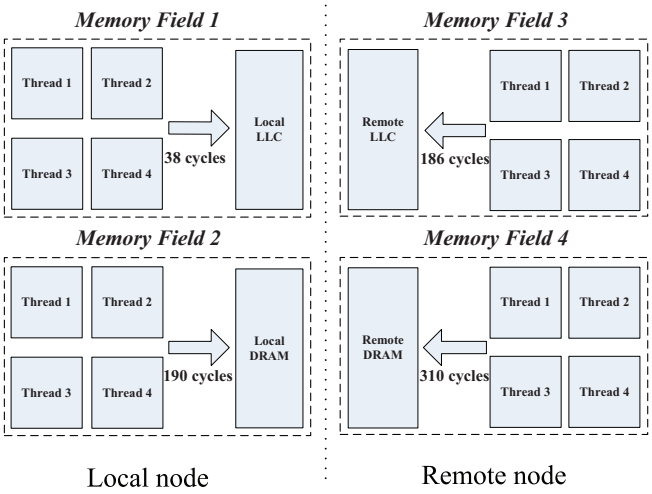


Fig. 1: An example of *memory fields* on NUMA architectures

there will be three performance problems if the mis-allocation of the data and computing appears on such systems.

Firstly, for shared data, it is very difficult to avoid remote memory access [2], [3], [4], [5]. There will be accesses to these shared data from remote *memory fields* no matter at which node a shared data is allocated. This is due to the fact that the shared data and threads which access the shared data can not be allocated exactly in the local *memory field*. The data sharing exists in the form of shared data structures which are accessed simultaneously by some or all threads of the multi-threaded program. Some threads have to access the shared data structure by crossing the inter-connect links if they are not allocated on the local *memory fields* of the shared data. For example, in *streamcluster* program (from *Parsec* benchmark [6]), the *points[]* data structure is shared by all threads. Due to threads which access *points[]* have to be allocated on all nodes of the system, we can not distribute this shared data and all threads which access it only on the local *memory fields* [2].

Secondly, contention on shared resources (e.g., last level cache, inter-connect links and memory controllers) can lead to extremely high data access latency [7]. The contention is caused by a large amount of data going in and out a

2324-9013/16 $31.00 © 2016 IEEE
DOI 10.1109/TrustCom.2016.185

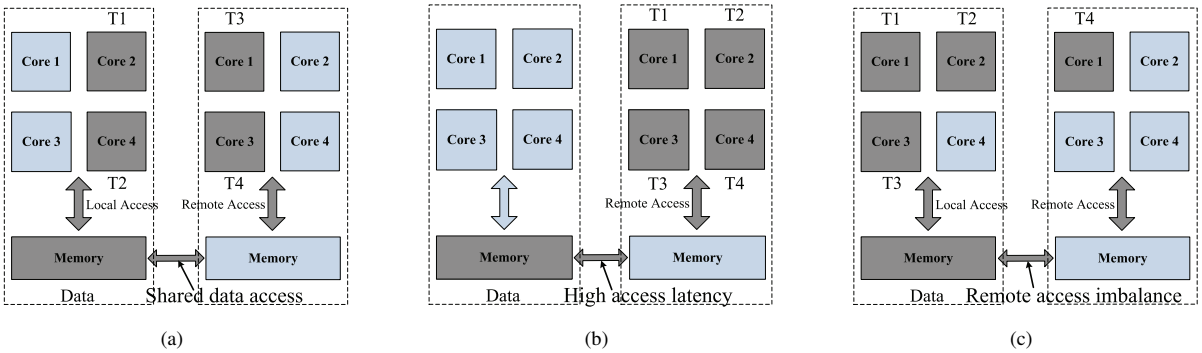


Fig. 2: Examples of three performance problems when multi-threaded programs run on NUMA architectures

single *memory field*. Memory access requests are generated very quickly by high-speed multi-core processors and thus contention will be caused by them. If a majority of data accesses generated by the multi-threaded program are not local, contention may occur due to the restricted bandwidth of remote *memory fields*. The bandwidth of a remote *memory field* is determined by the inter-connect link by which data can transfer between CPU cores and remote memory. This problem can be particularly severe if the data and threads which access the data are allocated on a single remote *memory field*. This is because threads will contend for the restricted bandwidth of that remote *memory field*. If the contention occurs, data access latency can become as high as 5X normal data access latency [7].

Thirdly, remote access distribution imbalance has a large influence on the performance of multi-threaded programs on NUMA systems [8], [9]. Remote data access normally takes more time than local data access. If the number of remote data access is extremely different among all threads of one multi-thread program, the thread which has the highest number of remote data access becomes the critical thread and will run the slowest among all threads. In a multi-threaded program, threads which run at quicker speeds have to wait threads which run at slower speeds at the synchronization points (e.g., barriers). Therefore, all threads from different *memory fields* can not continue execution until a delayed thread catches up. Moreover, the synchronization overhead becomes unacceptable if a multi-threaded program has a mass of synchronization due to the fact that threads have to wait for the critical thread at every synchronization point.

These three performance problems can not be distinguished easily from each other. They all show the phenomenon that they have high data access latency, but the reasons which cause high data access latency are different for each of them and the optimization methods are also different. We present a tool used to detect which specific factor causes the performance degradation. Then programmers can use the information offered by *LaProf* to guide them to do performance optimization. At present, detection tools which can find the performance problems of multi-threaded programs on NUMA systems have been rarely proposed. Especially, rare tools have been proposed to find out the problem of remote access distribution imbalance.

II. Motivation

This section shows three examples of performance problems when the multi-threaded program runs NUMA systems, which are shown in Figure 2(a), Figure 2(b), and Figure 2(c) individually. It is necessary to present a tool which can detect these three performance problems.

Data sharing problem [2], [3], [10] is shown in Figure 2(a). The shared data between threads *T1*, *T2* and *T3*, *T4* causes remote data access which has to across the inter-connect link. Two threads are allocated on node 0, two threads are allocated on node 1. The threads allocated on node 0 and 1 have shared data. The shared data is allocated on node 0. Threads on node 1 have to access shared data which is allocated on node 0 by across the inter-connect link. The shared data influences the performance of multi-threaded programs more severely than the private data of multi-threaded program. This is because the private data is always allocated on the local *memory field*. However, the shared data is accessed by many threads and it is impossible to allocate shared data only on the local *memory field*. Therefore, the coherence requirement of shared data will be completed by data transmission through the costly inter-connect links.

Shared resource contention problem [7], [11] is shown in Figure 2(b). The program data needed by the multi-threaded program is initialized on node 0. All worker threads *T1*, *T2*, *T3* and *T4* are allocated on node 1. All worker threads will contend for the restricted bandwidth of the inter-connect link if they simultaneously access the program data. In normal circumstances, the shared resource contention is reflected by extremely high data access latency. In this situation, it will lead to extremely high data access latency due to the contention of inter-connect links. The performance of multi-threaded program is lagged down severely by shared resource contention on the inter-connect link.

Remote access distribution imbalance problem [8], [9] is shown in Figure 2(c). The program data is initialized on node 0 and three worker threads *T1*, *T2* and *T3* are also

allocated on node 0. They all access the program data locally through local memory controller which has low data access latency. However, the worker thread *T4* is allocated on node 1 and accesses the program data allocated on node 0 only through the inter-connect link and remote memory controller, which has high data access latency. As a consequence, the thread allocated on node 1 has much larger number of remote data access than the threads allocated on node 0. Thread *T4* becomes the critical thread and delays the overall program process significantly because threads *T1*, *T2* and *T3* have to wait it at barriers. In order to improve performance, multi-threaded applications with inter-thread synchronization (e.g., barriers) need to balance the number of costly remote data accesses for all threads.

Different optimization methods have to be adopted if different problems influence the performance. There are three performance optimization methods which correspond to the three problems: 1) For data sharing, we should cluster threads which have shared data onto the same node, so as to eliminate the cross-node shared data access and coherence overhead; 2) For high data access latency, we should alleviate or eliminate shared resource contention. For example, memory-intensive threads should be allocated onto different nodes so as to alleviate memory controller or LLC contentions. What is more, the data and computing threads should be allocated onto the same node if the inter-connect contention causes the performance degradation; 3) For remote access imbalance, we can allocate these threads and the program data symmetrically on all nodes so as to make all threads have the same number of local and remote data access.

Especially, our tool can find out the problem of remote access imbalance, which is rarely discussed by previous research work. In order to demonstrate the importance of this problem, we show more performance optimization methods for this problem. For threads which have higher number of remote data access, there are three optimization methods: 1) These threads can be given higher execution priorities in SMT processors so as to avoid them waiting in the ready-to-run queues; 2) Memory migration policy can be used to migrate the data of these threads from remote node to local node so as to reduce the number of remote data access for them; 3) *Dynamic voltage and frequency scaling* (DVFS) policy can be used to accelerate the execution of these threads. The performance improves due to the fact that these threads will run faster and arrive at the synchronization point (e.g., barriers) simultaneously with other threads.

III. Implementation

The realization of *LaProf* is divided into two parts: the sampling part and the analysis part. The relationship between these two parts is shown in Figure 3. The sampling part gets the sample result. The analysis part analyzes the private/shared data access latency, high data access latency and remote access distribution according to the sample result. Using the analysis result, we can explicitly point out which factors affect the performance of multi-threaded program.

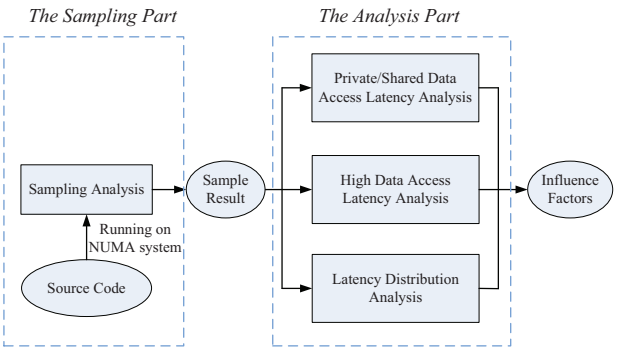


Fig. 3: The relationship of the main components of *LaProf*

A. The Sampling Part

Figure 4 shows the sampling realization of *LaProf*. At first, we use *vfork()* system call to create a child thread, which is used to run the multi-threaded program which we intend to sample. The parent thread is used to sample the multi-threaded program during its execution phase. Then the child thread invokes the system call *execvp()* to start the execution of the multi-threaded program. The parent thread is blocked until the child thread starts to invoke the system call *execvp()*. It means that only when the multi-threaded program starts execution, the sampling procedure starts to work.

When the child thread invokes the system call *execvp()*, the parent thread recovers execution. The parent thread acquires the *pid* of the child thread, which is returned by the system call *vfork()*. Then, we detect the number of threads by checking the number of file directories in Linux system path */proc/pid/task*. The parameter *NUM_THREADS* is used to pre-define the number of threads in parallel execution phase. Once the number of file directories equals to *NUM_THREADS*, we know that the multi-threaded program starts execution in parallel. We start sampling just as the multi-threaded program begins running in parallel stage.

We initiate the sampling procedure by using the function *sample_init*. This function invokes *perf_event_open* to create a file descriptor used to record the sampling result. It invokes *mmap()* to initiate the buffer *metadata_page* for storing the sampling result. After that, it uses *ioctl* to enable the counters of sampling events.

When the multi-threaded program finishes execution, the child thread is terminated. At this time, the parent thread stops sampling and invokes system call to get the *pid* and status of the child thread. Afterwards, the parent thread continues execution and invokes *sample_read* function to read the sample result recorded by *perf_event_open*. When the multi-threaded program finishes execution, the parent thread uses *ioctl* to disable the counters of sampling events. Then the parent thread reads the sampling data from buffer *metadata_page*. At last, these sampling data will be recorded in our sampling result file.

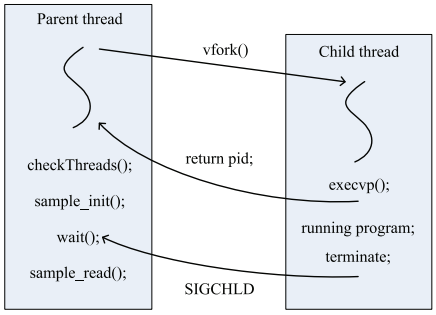


Fig. 4: The sampling realization of *LaProf*

B. The Analysis Part

From the sampling result, we intend to do data access latency analysis to find out the performance influence factors when multi-threaded programs run on NUMA systems. We analyze three performance influence factors, e.g., the data access latency of shared data, high data access latency and the unbalanced remote access distribution, which are illustrated below:

(1) Private/shared data access latency analysis. The sample result records that each data is accessed by which thread and the access latency. The private and shared data are classified by the number of threads which access it. Private data is accessed only by one thread, shared data is accessed by more than one. *LaProf* records the data address, the access latency associated with the address and the thread which accesses that data. Due to a lot of samples recorded during a single run, we write a script to help us analyze sample records. The script can analyze how many times each data is accessed and which threads access it. The analysis result is classified by data addresses and the access latency associated with each data address is summed up. At last, we compute the total/average data access latency of private/shared data and total number of private/shared data access.

(2) High data access latency analysis. *LaProf* only records the samples which satisfy the latency threshold requirement. By setting different data access latency threshold, *LaProf* only records the data accesses which have data access latency greater than the latency threshold. Shared resource contention can lead to extremely high data access latency which can not occur under normal circumstances that the contention does not happen. We can detect whether shared resource contention has happened by counting the number of recorded samples in high latency threshold setting. Furthermore, we can know the data access latency distribution by setting different data access latency thresholds.

(3) Unbalanced remote access distribution. The unbalanced remote access distribution is reflected by the number of recorded data for each thread. The imbalance of remote access distribution will be worse if the difference of recorded data number among all threads is larger. We use the standard deviation of the number of recorded data for all threads to represent the imbalance degree of remote access distribution.

TABLE I: Parameters of the NUMA platform

Processor	2 x Intel Xeon E5-2670
Cores per processor	8
Core number on Node 0	Core 0 $\sim$ 7
Core number on Node 1	Core 8 $\sim$ 15
L3 cache size	20MB
Main memory	64GB DDR3
Max IMC bandwidth	51.2GB/s
QPI bandwidth	8.0GT/s
Local DRAM access latency	62ns
Remote DRAM access latency	113ns

The standard deviation of the recorded data is associated with the runtime of multi-threaded program to show how the remote access distribution imbalance influences the performance of multi-threaded program.

IV. Experiments

The configuration of the NUMA platform is shown in Table I. When a multi-threaded program runs on this platform, *LaProf* acquires the data access latency, the *id* of thread which issues that data access and the memory hierarchy on which the accessed data is allocated (eg., the data can be located in local/remote L1, L2, L3 cache or memory). The configuration parameters of *LaProf*, such as the latency threshold and the sample period, can change in different runs. A data access will be recorded only if its latency is greater than the latency threshold. One record will write into our sample result each time when the number of records reaches to the value of the sample period.

The sample result acquired by *LaProf* is shown in Table II. The arabic numerals begin with 1 are used to stand for the actual thread *id* of multi-threaded programs (e.g., 1 stands for *tid* 8421, 2 stands for *tid* 8422). *Addr* means the linear data address accessed by the thread. Symbol *A* represents the address *2b348c100102*. It shows up in line 2, 5 and 7. It means that *A* is the shared data address accessed by thread 1, 2 and 3 simultaneously. In the meanwhile, other addresses are called private data addresses which are only accessed by one thread. The *cpu* means the core of the NUMA system that the thread is allocated on. The *weight* stands for the data access latency when the thread accesses the address denoted by *Addr*. The unit of all data latency and latency threshold values is CPU cycles.

Two commands *taskset* and *numactl* are used in our experiment. *taskset* is used to bind the threads to specific cores. *numactl* can be used to set thread scheduling strategy and memory allocation policy for NUMA systems. Using these two commands, we record the sample result in different configurations of memory and CPU affinities. As shown in Table I, Node 0 and Node 1 has Core 0 $\sim$ 7 and Core 8 $\sim$ 15 respectively. In our experiment, *cluster* stands for *taskset -c 0, 1, 2, 3, 4, 5, 6, 7*, which means threads are all allocated on Node 0. *distributed* stands for *taskset -c 1, 3, 5, 7, 9, 11, 13, 15*, which means threads are allocated on Node 0 and 1 symmetrically. Furthermore, *m 0 N 0* stands for *numactl -m 0 -N 1*, which means the memory is allocated on Node 0

TABLE II: The sample result acquired by *LaProf*

```
1: Thread_id = 1, Addr = 2b348ea54414, cpu = 2, weight = 93
2: Thread_id = 1, Addr = A           , cpu = 2, weight = 10d
3: Thread_id = 1, Addr = 2b348c576f50, cpu = 2, weight = d5
4: Thread_id = 2, Addr = 2b348bdb5810, cpu = 5, weight = e0
5: Thread_id = 2, Addr = A           , cpu = 5, weight = 148
6: Thread_id = 3, Addr = 2b3492190c98, cpu = 6, weight = 45
7: Thread_id = 3, Addr = A           , cpu = 6, weight = 165
8: Thread_id = 4, Addr = 2b348123ea12, cpu = 4, weight = c9
```

while threads are allocated on Node 1. *interleave* stands for *numactl –interleave=all*, which means the memory is allocated circularly on each node.

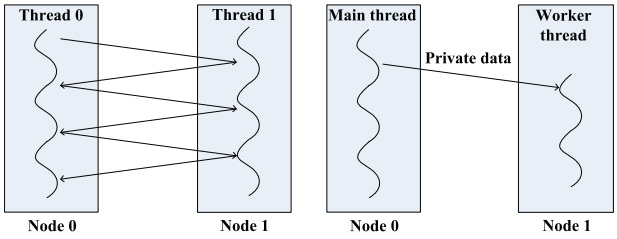
A. Private/Shared Data Access Latency Analysis

We use *streamcluster*, which is selected from *Parsec* benchmark [6], to illustrate the influence of private/shared data access latency. *Streamcluster* is one of data mining algorithms to solve the problem of online clustering. We get the analysis result after the execution of *streamcluster*, which is shown in Table III. $TL_{private}$ and TL_{shared} means total latency of private and shared data access, respectively. $TN_{private}$ and TN_{shared} means total number of private and shared data access, respectively. $AL_{private}$ and AL_{shared} means average latency of private and shared data access, respectively.

In Table III, $TL_{private}$, $TN_{private}$ and $AL_{private}$ of method *distributed* is 1.04X, 1.18X and 0.88X of method *cluster*. However, TL_{shared}, TN_{shared} and AL_{shared} of method *distributed* is 3.64X, 1.4X and 2.6X of method *cluster* respectively. Then, we compare the runtime of method *cluster* and *distributed*. The runtime of method *distributed* is 1.87X of method *cluster*. Through the above analysis, we find that the enormous increase of shared data access latency (AL_{shared}) is the main reason which causes the performance degradation of method *distributed*. Although the shared data access only accounts for 0.13% of total data access, the shared data access influences the performance more severely than the private data access, as shown in Table III. The same reason applies to method *m 0 N 0* and *interleave*. We also run *blackscholes* with 8 threads using *simlarge* input and *openmp* model. The conclusion we get is similar to the benchmark *streamcluster*, as shown in Table IV.

The explanation why shared data access influences performance more severely is given below: the cache coherence overhead caused by shared data access can largely increase the data access latency of shared data. If we use method *cluster* or method *m 0 N 0*, both shared and private data is accessed locally. However, if we use method *distributed* or method *interleave*, the shared data will migrate frequently between different nodes when it is alternately frequently accessed by two threads which are allocated on different nodes, as shown in Figure 5(a). Shared data can not always be allocated on local node and will be fetched from remote memory when its copy allocated on local memory is invalidated. When threads access the shared data, they may need across the inter-connect link. Nevertheless, the private data is always allocated on the local node and thread accesses private data locally. There exists no cache coherence overhead for private data. Private data will not migrate frequently between nodes of NUMA system. In our experiment, threads are bind to specific cores. There exists only one possible migrations for private data. It will migrate from the node on which it is initialized by main thread to the node on which it is accessed by worker thread, as shown in Figure 5(b). Therefore, the shared data is affected more from NUMA characteristics than the private data.

By using *cluster* strategy, we can reduce or eliminate the remote data access caused by shared data. Compared with *distributed* strategy, the performance of *streamcluster* benchmark improves 88%.



(a) Shared data migrates frequently (b) Private data migrates no more than once

Fig. 5: The difference between shared data and private data migration on NUMA systems

B. High Data Access Latency Analysis

We use *LU* benchmark selected from NPB benchmark [12], to study the influence of high data access latency. *LU* realizes the lower-upper symmetric gauss-seidel. We measure its data access latency characteristics at two configurations. The first configuration is that the latency threshold is equal to 30. The second configuration is that the latency threshold is equal to 300. It means that the samples will be recorded only if its data access latency is larger than 30 and 300 in first and second configuration respectively.

In Table V, TL means total latency of data access, TN means total number of data access, AL means average latency of data access. Figure 6 shows the ratio between TL, TN, AL of method *m 0 N 1* and that of method *m 0 N 0*. In the configuration that the latency threshold is greater than 30, the TL, TN and AL of method *m 0 N 1* is only 1.88X, 1.03X, 1.83X of method *m 0 N 0* respectively. However, in the configuration that the latency threshold is greater than

TABLE III: The Runtime and Access Latency of Private/Shared Data for *streamcluster*

Method	Time(s)	$TL_{private}$(cycles)	$TN_{private}$	$AL_{private}$(cycles)	TL_{shared}(cycles)	TN_{shared}	AL_{shared}(cycles)
cluster	16.03	13307523	67327	197.66	11616	85	136.34
distributed	30.11	13892938	79609	174.52	42281	119	**354.71**
m 0 N 0	16.25	13273781	67590	196.39	10134	68	149.04
interleave	22.37	12196883	65545	186.08	40106	106	**378.36**

TABLE IV: The Runtime and Access Latency of Private/Shared Data for *blackscholes*

Method	Time(s)	$TL_{private}$(cycles)	$TN_{private}$	$AL_{private}$(cycles)	TL_{shared}(cycles)	TN_{shared}	AL_{shared}(cycles)
cluster	3.66	43548	855	50.93	86307	582	148.29
distributed	5.56	42994	853	50.4	180982	544	**332.69**
m 0 N 0	3.67	43327	838	51.7	93399	602	155.15
interleave	6.16	40643	799	50.87	180351	576	**313.11**

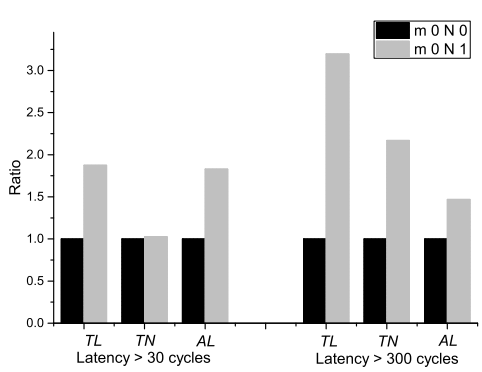


Fig. 6: The ratio between the parameters of *m 0 N 1* and *m 0 N 0* method

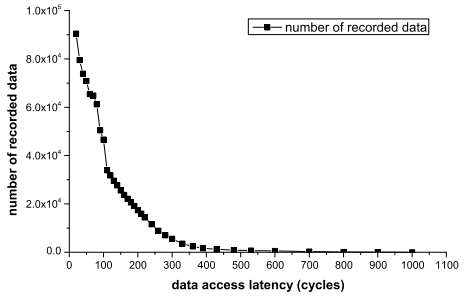


Fig. 7: The data access latency distribution of *streamcluster* benchmark

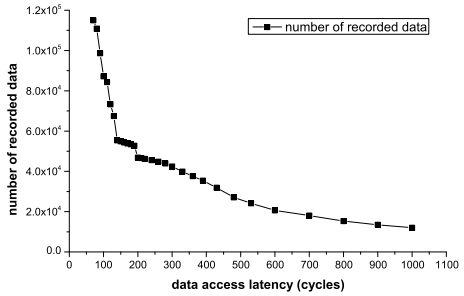


Fig. 8: The data access latency distribution of *lu* benchmark

300, the TL, TN and AL of method *m 0 N 1* is as large as 3.2X, 2.17X and 1.47X of method *m 0 N 0* respectively. From Table V, we can find out that the runtime of method *m 0 N 1* is 1.3X of method *m 0 N 0*. Through the above analysis, we can find out that the values (TL, TN and AL) with high latency threshold configuration (greater than 300) are more sensible and efficient indicators of the runtime of multi-threaded programs on NUMA systems.

TABLE V: The Runtime and Data Access Latency of *LU*

Method	Time(s)	Threshold	TL(cycles)	TN	AL(cycles)
m 0 N 0	27.29	30	3017960	23886	126.35
m 0 N 1	36.09	30	5664781	24521	231.02
m 0 N 0	27.27	300	12161837	18501	657.36
m 0 N 1	34.92	300	38888613	40230	966.66

We compare the data access distribution of *streamcluster* benchmark and *LU* benchmark. Figure 7 shows the data access latency distribution of *streamcluster* benchmark. 61192, 2384 and 9 samplings are recorded when the latency threshold is larger than 80, 360 and 1000, respectively. We can see the data access latency is distributed mainly in the low data access latency part. There are negligible number of data access which has latency greater than 1000 cycles. Figure 8 shows the data access latency distribution of *LU* benchmark when it runs with *m 0 N 1* method. From this figure, we can see that the high data access latency can not be negligible. 110898, 42404 and 12022 samplings are recorded when the latency threshold is larger than 80, 300 and 1000, respectively.

We give the illustration as below: The extremely high data access latency influences the performance of multi-threaded programs severely. If the extremely high data access latency occurs, it is very likely that shared resource contention happens. We give three circumstances where the shared resource contention will happen: 1) The shared last level cache can not hold all working set of threads running on this node; 2) The memory access request exceeds the processing capability of memory controllers; 3) The number of remote data access goes beyond the bandwidth of inter-connect links. When shared resource contention happens, the data access latency can as high as 1000 cycles, which is 5X larger than the data access latency caused by inter-connect wire delays [7]. Therefore, the higher number of extremely high data access latency appeared

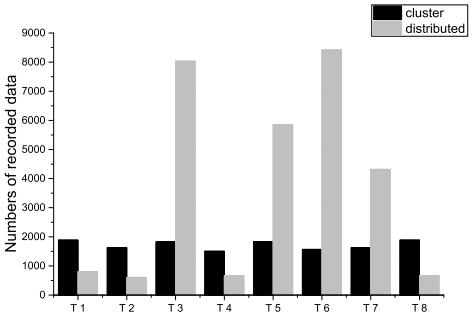


Fig. 9: The distribution unbalance of *cluster* and *distributed* method

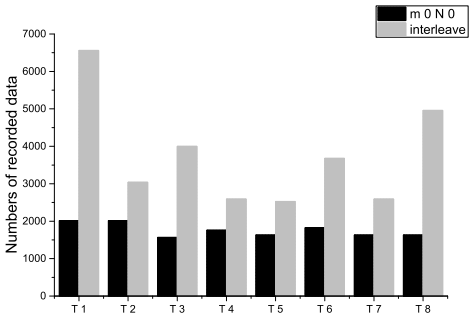


Fig. 10: The distribution unbalance of *m 0 N 0* and *interleave* method

in our sample result indicates the higher possibility that shared resource contention may occur. Our analysis tool *LaProf* can effectively detect whether or not and how often the extremely high data access latency occurs. With these information we can predict whether the shared resource contention has happened and thus how the performance of multi-threaded programs is affected by the shared resource contention.

By using *m 0 N 0* strategy, we can alleviate or eliminate the contention on inter-connect links. Compared with *m 0 N 1* strategy, the performance of *LU* benchmark improves 32%.

C. Remote Access Distribution Analysis

We use *freqmine* benchmark, which is also from *Parsec* benchmark [6], to explain the influence of remote access distribution. It is a data mining benchmark which is used for frequent itemset mining. The latency of local data access is assumed to be lower than 300 and the latency of remote data access is assumed to be higher than 300. If the latency threshold is set to 300, it means that only remote data accesses are recorded in our sample result.

We record the number of samples for each thread of *freqmine* to study the distribution of remote access. Figure 9 shows the distribution of recorded data for method *cluster* and method *distributed* respectively. The distribution of recorded data for method *cluster* is well balanced. The standard deviation of method *cluster* is 152.75. However, the distribution of recorded data for method *distributed* is very uneven. The standard deviation of method *distributed* is 3429.73. Figure 10 shows the distribution of recorded data for method *m 0 N 0* and method *interleave*. The distribution of recorded data for method *m 0 N 0* is quite symmetric. The standard deviation is 177.76. Nevertheless, the distribution of recorded data for method *interleave* is very asymmetric. The standard deviation is 1419.59.

Table VI displays the relationship between runtime and standard deviation of recorded data for different methods. We can find out that the runtime of multi-threaded programs is much longer when the standard deviation of recorded sample number for each thread is much higher. The runtime of multi-thread programs will be short if the standard deviation is low.

TABLE VI: The relationship between runtime and standard deviation of recorded data for *freqmine*

Method	Time(s)	Standard deviation
cluster	67.71	152.75
distributed	134.64	3429.73
m 0 N 0	64.67	177.76
interleave	136.4	1419.59

Our explanation for this phenomenon is described as below: If the recorded sample number for each thread distributes unbalanced, it means some threads have more number of remote data access than other threads. If the standard deviation of recorded data is higher, the number difference of remote data access will be larger among all threads. The threads with smaller number of remote data access have to wait for those threads with bigger number of remote data access at the synchronization point (e.g., barriers). The thread with the largest number of remote data access becomes the critical thread and lags down the overall performance, such as the thread *T6* in Figure 9 and the thread *T1* in Figure 10.

By using *cluster* strategy, we reduce or eliminate the distribution imbalance of remote data access. Compared with *distributed* strategy, the performance of *freqmine* benchmark improves 99%.

V. Related Work

Data sharing overhead has been discussed by work [2], [3], [4]. Data sharing could lead to remote data access on NUMA systems. Majo *et al.* [2] aimed at solving data sharing among threads in one multi-threaded program. Their work showed that a small change of the data structure used by the multi-threaded program could alleviate or eliminate data sharing.

Several work has focused on reducing remote memory accesses as much as possible when multi-threaded programs run on NUMA machines. Lachaize *et al.* [13] presented a profiler tool which could find interactions between threads and memory objects. The goal of their work was to find out solutions which could reduce the number of remote data access. By recording adequate and accurate information during the program execution phase, the tool could help programmers to find out which memory objects were accessed remotely and the reason why they caused remote data access.

Liu *et al.* [14] proposed a tool which could analyze memory access behaviors of multi-threaded program and provided sufficient information to solve performance problems of NUMA systems. They analyzed the recorded information and researched on how to map data and computing so as to increasing local accesses and reducing inter-connect contention. Their work aimed at solving two performance problems of NUMA systems: overburdened remote accesses and distribution imbalance of memory access requests on different nodes of NUMA systems.

Previous research works [15], [16], [17], [18] have presented that the performance of multi-threaded programs can be affected by NUMA architectures. However, they do not give a efficient method that can help the programmer detect which specific factors influence the performance of multi-threaded program on NUMA architectures. Our tool can be used to detect these performance influence factors.

VI. Conclusions

We propose a tool called *LaProf* which uses the data access latency information to detect performance problems of multi-threaded programs on NUMA systems. *LaProf* can be used to detect three performance problems: 1) Shared data access; 2) Contention on shared resources; 3) Remote access distribution imbalance. After the detection done by *LaProf*, the result can be used to guide the performance optimization of multi-threaded programs on NUMA systems. In our work, the performance of *streamcluster*, *LU*, *freqmine* is improved by 88%, 32%, 99% respectively.

Acknowledgment

This work was supported by National High-tech Research and Development Program of China (863 Program) under grant No.2015AA015303.

References

[1] D. Hackenberg, D. Molka, and W. E. Nagel, "Comparing cache architectures and coherency protocols on x86-64 multicore SMP systems," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'09)*. New York, NY, USA: ACM, 2009, pp. 413–422.

[2] Z. Majo and T. R. Gross, "(mis)understanding the NUMA memory system performance of multithreaded workloads," in *Proceedings of the 2013 IEEE International Symposium on Workload Characterization (IISWC'13)*, Sept. 2013, pp. 11–22.

[3] J. Rao, K. Wang, X. Zhou, and C. Z. Xu, "Optimizing virtual machine scheduling in NUMA multicore systems," in *Proceedings of the 2013 IEEE International Symposium on High Performance Computer Architecture (HPCA'13)*, Feb. 2013, pp. 306–317.

[4] D. Tam, R. Azimi, and M. Stumm, "Thread clustering: Sharing-aware scheduling on SMP-CMP-SMT multiprocessors," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys'07)*. New York, NY, USA: ACM, 2007, pp. 47–58.

[5] F. Gaud, B. Lepers, J. Decouchant, J. Funston, A. Fedorova, and V. Quéma, "Large pages may be harmful on NUMA systems," in *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC'14)*. Berkeley, CA, USA: USENIX Association, 2014, pp. 231–242.

[6] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT'08)*. New York, NY, USA: ACM, 2008, pp. 72–81.

[7] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth, "Traffic management: A holistic approach to memory placement on NUMA systems," in *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*. New York, NY, USA: ACM, 2013, pp. 381–394.

[8] L. Zhu, H. Jin, and X. Liao, "Syms: a symmetrical scheduler to improve multi-threaded program performance on NUMA systems," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 18, pp. 5810–5825, 2015.

[9] C. Su, D. Li, D. S. Nikolopoulos, M. Grove, K. Cameron, and B. R. de Supinski, "Critical path-based thread placement for NUMA systems," *SIGMETRICS Perform. Eval. Rev.*, vol. 40, no. 2, pp. 106–112, Oct. 2012.

[10] Q. Lu, C. Alias, U. Bondhugula, T. Henretty, S. Krishnamoorthy, J. Ramanujam, A. Rountev, P. Sadayappan, Y. Chen, H. Lin, and T.-F. Ngai, "Data layout transformation for enhancing data locality on NUCA chip multiprocessors," in *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques (PACT'09)*, Sept. 2009, pp. 348–357.

[11] T. Dey, W. Wang, J. Davidson, and M. Soffa, "Characterizing multi-threaded applications based on shared-resource contention," in *Proceedings of the 2011 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'11)*, Apr. 2011, pp. 76–86.

[12] R. F. Van der Wijngaart and P. Wong, "NAS parallel benchmarks version 2.4," NAS technical report, NAS-02-007, Tech. Rep., 2002.

[13] R. Lachaize, B. Lepers, and V. Quéma, "Memprof: A memory profiler for NUMA multicore systems," in *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC'12)*. Berkeley, CA, USA: USENIX Association, 2012, pp. 53–64.

[14] X. Liu and J. Mellor-Crummey, "A tool to analyze the performance of multithreaded programs on NUMA architectures," in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'14)*. New York, NY, USA: ACM, 2014, pp. 259–272.

[15] B. Lepers, V. Quema, and A. Fedorova, "Thread and memory placement on NUMA systems: Asymmetry matters," in *Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC'15)*. Santa Clara, CA: USENIX Association, July 2015, pp. 277–289.

[16] C. McCurdy and J. Vetter, "Memphis: Finding and fixing NUMA-related performance problems on multi-core platforms," in *Proceedings of the 2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS'10)*, Mar. 2010, pp. 87–96.

[17] Z. Majo and T. R. Gross, "A library for portable and composable data locality optimizations for NUMA systems," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'15)*. New York, NY, USA: ACM, 2015, pp. 227–238.

[18] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "Hpctoolkit: tools for performance analysis of optimized parallel programs," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 685–701, 2010.