How OpenMP Applications Get More Benefit from Many-Core Era

Jianian Yan, Jiangzhou He, Wentao Han, Wenguang Chen, and Weimin Zheng

Department of Computer Science and Technology,
Tsinghua University, China
{yanjn03,hejz07,hwt04}@mails.tsinghua.edu.cn,
{cwg,zwm-dcs}@tsinghua.edu.cn

Abstract. With the approaching of the many-core era, it becomes more and more difficult for a single OpenMP application to efficiently utilize all the available processor cores. On the other hand, the available cores become more than necessary for some applications. We believe executing multiple OpenMP applications concurrently will be a common usage model in the future. In this model, how threads are scheduled on the cores are important as cores are asymmetric. We have designed and implemented a prototype scheduler, SWOMPS, to help schedule the threads of all the concurrent applications system-widely. The scheduler makes its decision based on underlying hardware configuration as well as the hints of scheduling preference of each application provided by users. Experiment evaluation shows SWOMPS is quite efficient in improving the performance.

With the help of SWOMPS, we compared exclusive running one application and concurrent running multiple applications in term of system throughput and individual application performance. In various experimental comparisons, concurrent execution outperforms in throughput, meanwhile the performance slowdown of individual applications in concurrent execution is reasonable.

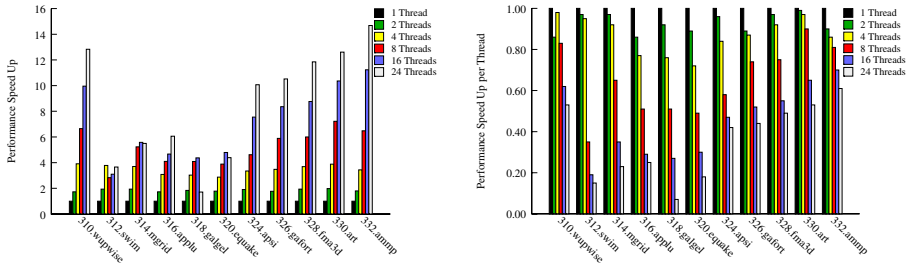
1 Introduction

Restricted by heating and power consumption, hardware vendors stopped increasing processor's performance by introducing complex circuit and increasing frequency. Instead, multiple cores are put into one chip. The effect of Moore's law has converted from increasing the performance of a single-core processor to the number of cores in a processor. Six-core general-purpose processor has been available in the market. Processors with more and more cores are coming soon. In the near future, one can easily have a computer with hundreds of cores by configuring multi-core processors in NUMA architecture.

In the multi-core era, OpenMP applications face the challenge of how to efficiently utilize the increasing computing power. Keeping the performance of OpenMP applications scaling with the number of processor cores is not trivial. Simply increasing the number of threads in an OpenMP applications does not guarantee the performance scaling. Figure 1 shows the result of scalability experiment with benchmarks in SpecOMP 2001. The experiment platform has 24 cores which detail configuration can be found in Sect. 4. In the test, the performance speedup per thread continues decreasing as the

M. Sato et al. (Eds.): IWOMP 2010, LNCS 6132, pp. 83–95, 2010.

number of threads grows, as showed in Fig. 1b. And for some benchmarks, for example *314.mgrid*, *318.galgel* and *320.equake* showed in Fig. 1a, performance with 24 threads is even worse than performance with 16 threads. Writing well scaling applications requires sophisticated techniques and wide range of knowledge from hardware architecture to application domain. And theoretically, according to Amdahl's law, even for perfectly written applications, the existence of serialized portion in the application limits the benefit that can be obtained by parallel execution, no matter how many cores are available.



(a) Performance Speed Up (b) Performance Speed Up per Thread

Fig. 1. Performance Scaling with Various OMP Threads

Single OpenMP application has difficulty to efficiently utilize the increasing processor cores. On the other side, for some applications, it is unnecessary to utilize as many as the available cores. We believe multiple OpenMP applications executing concurrently will be a common usage model in the future. In this model, which cores the threads of an application are executing on will be an important factor to the performance. In a computer with multi-core processors configured in NUMA architecture, the cores are asymmetric. Cores in the same processor share the last level cache and have the same local memory while cores in different processors do not. An improper scheduling of threads on the cores will harm the performance of not only the application itself, but also other applications concurrently running on the computer.

We have designed and implemented a prototype scheduler SWOMPS (**S**ystem-**W**ide **O**pen**MP** application **S**cheduler) to help schedule the threads of all concurrent applications system-widely. The scheduler makes its decision based on underlying hardware configuration as well as the hints of scheduling preference of each application provided by users. It will dynamically schedule the threads whenever an application comes to run or exits.

With the help of SWOMPS, we compared exclusive running one application and concurrent running multiple applications in term of system throughput and individual application performance. We take two application concurrently running for example. In various experimental comparisons, concurrent execution outperforms in throughput, meanwhile the performance slowdown of individual application in concurrent execution is reasonable.

The rest of the paper is organized as follows: Section 2 introduces two observations from the practical experience with OpenMP applications that guide the scheduler

design. Section 3 introduces the design and implementation of SWOMPS scheduler. Section 4 evaluates the scheduler and Sect. 5 compares the exclusive running model and the concurrent running model. Related work is introduced in Sect. 6 and Sect. 7 concludes the paper.

2 Practical Observations

2.1 Binding a Thread to a Core Will Improve Application Performance

Modern operating systems provide system calls with which user can specify a core that a thread is running on, which is also known as binding a thread to a core. OpenMP programmer usually binds each thread to a distinct core which will results in a better performance. Table 1 gives the execution time of the benchmarks with and without thread binding. Each benchmark is compiled with Open64 at O3 optimizing level, and running with 24 threads. The detail of the experiment environment can be found in Sect. 4. The last column of Table 1 shows the performance improvement due to thread binding. As we can see, the performance improvement ranging from 0.2% to 31.4%. There are two major sources of the performance improvement. Binding a thread avoids cache warming up when it is scheduled to a new core. And binding a thread keeps a core close to the data that it operates in the NUMA architecture.

Table 1. Execution Time Comparison of Thread Binding

Benchmark	Execution Time (s)		Perf. Impr.
	No Binding	Binding	
310.wupwise	190.13	170.36	11.6%
312.swim	405.21	401.94	0.8%
314.mgrid	660.11	523.93	26.0%
316.applu	228.76	227.93	0.4%
318.galgel	1192.54	939.25	27.0%
320.equake	148.59	121.95	21.8%
324.apsi	118.91	112.86	5.4%
326.gafort	311.10	310.53	0.2%
328.fma3d	222.95	199.97	11.5%
330.art	179.70	136.79	31.4%
332.ammp	320.40	302.94	5.8%

Table 2. Scheduling Preference of Each Benchmark

Benchmark	Execution Time (s)		Perf. Impr.
	Scatter	Gather	
310.wupwise	259.15	324.8	25.33%
312.swim	363.75	780.18	114.48%
314.mgrid	482.39	908.16	88.26%
316.applu	254.71	395.16	55.14%
318.galgel	616.53	676.9	9.79%
320.equake	129.54	179.15	38.30%
324.apsi	194.48	179.64	8.26%
326.gafort	479.53	641.55	33.79%
328.fma3d	290.59	377.77	30.00%
330.art	206.66	244.03	18.08%
332.ammp	510.3	490.78	3.98%

2.2 Different Applications Have Different Scheduling Preferences

Nowadays, most shared memory computers are configured in NUMA architecture. A processor can access its own *local memory* faster than non-local memory. A processor again contains several cores and cores in the same processor usually share the last level cache. The processor cores are asymmetric. Different OpenMP applications may have different thread scheduling preferences on the cores. Table 2 gives the results of a

scheduling preference experiment. The experiment runs on a computer with four processors, and each processor has six cores. A processor has 6M L3 cache that is shared amoung the 6 cores. In the experiment, each benchmark runs with 12 threads. In the *Scatter* scheduling, the 12 threads are scheduled onto 4 processors, each processor has 3 threads running on it. While in the *Gather* scheduling, the 12 threads are scheduled onto 2 processors, each processor has 6 threads running on it. In the scatter scheduling, the amount of shared cache per thread is larger and so are the memory bandwidth to local memory. In the gather scheduling, communication has lower cost as more threads are sharing the L3 cache, and more cores are sharing the same local memory that reduces the chance to access remote memory. Data in the *Perf. Impr.* column in Tab. 2 gives the performance improvements if an application is scheduled in favor of its preference versus against its preference. The observed performance improvements range from 3.98% to 114.48%. The experiment results indicate that different applications have different scheduling preferences. The performance improvement due to being properly scheduled varies from application to application.

3 SWOMPS: Design and Implementation

3.1 Scheduling Requirements

Learning from the experience introduced in Sect. 2, we believe a scheduler is needed to help improve application performance and the whole system's efficiency. The scheduler should fulfill the following requirements:

- **The scheduler should be system-wide that it could schedule multiple applications running currently**
  General OpenMP applications could not perfectly scale with the increasing of processor cores. When the number of available cores exceeds the requirements of a single OpenMP application, it is necessary to share the computer among multiple applications. A scheduler supporting multiple concurrently running applications is needed.
- **The scheduler should have respect for applications' scheduling preferences and resolve possible preference conflicts**
  Each OpenMP application has its specific scheduling preference. The scheduler should be aware of it and perform scheduling accordingly. When the scheduling preferences of concurrently running applications conflict, the scheduler should resolve the conflicts in favor of specific optimizing goal. For example, one way to optimize the system throughput is to weight each applications by the performance impact of its preferred scheduling, and satisfy the applications in the descending order.
- **Each thread should be bound to a core and this binding should be kept as long as possible**
  Scheduler should bind each thread to a specific core to favor its memory access. However, scheduler might need to reschedule the bindings when a new application comes to run or an application terminates. Rebinding a thread to a new core will need to warm up the cache again and may increase memory accesses to non-local memory. When the scheduler needs to adjust its previous scheduling, it should find a scheduling plan that rebinds previous threads as few as possible.

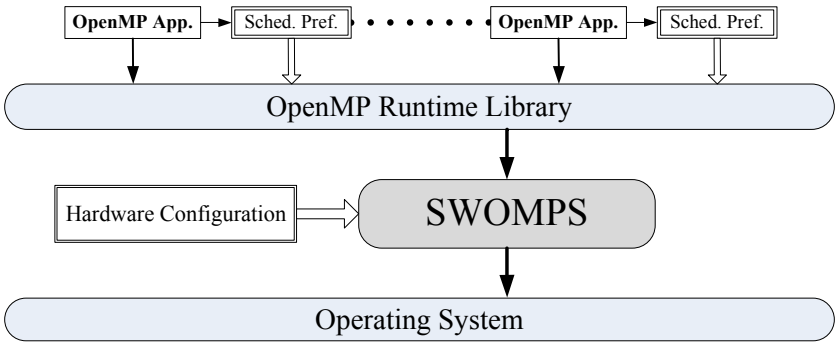


Fig. 2. SWOMPS Architecture

3.2 SWOMPS Work Flow

We have designed a prototype scheduler, SWOMPS (**S**ystem-**W**ide **O**pen**MP** application **S**cheduler) according to the observations introduced previously. Figure 2 shows how SWOMPS cooperates with other parts in the system. And its work flow is as following:

1. SWOMPS starts as a daemon. It inquiries the hardware configuration and initializes internal system model.
2. An application reports its scheduling preference to a database when it starts execution. The scheduling preference is described as an integer. The negative value indicates the application prefers scatter-scheduling, and the positive value indicates the application prefers gather-scheduling. The absolute value of the integer indicates the potential performance impact if the application is properly scheduled. The preference may vary with underlying system, it should not be hard coded. In our prototype implementation, the scheduling preference is stored in an environment variable. Moreover, for many real world applications, the preference may vary due to the alternation of different computational kernels. We use an overall preference for each application for simplicity.
3. OpenMP runtime library inquiries the scheduling preference after creating threads pool. It sends the preference as well as application process ID and thread IDs to SWOMPS.
4. When the application terminates, OpenMP runtime library sends the process ID to SWOMPS.
5. SWOMPS generates a scheduling plan whenever it receives a message. The scheduling plan for each process is calculated according to the current states of the system and the changes of the work load, either a group of threads start execution or they terminate. And the plan is carried out with the help of operating system.

3.3 Scheduling Algorithm

There are two phases in the scheduling algorithm of SWOMPS. The first phase assigns the core quota in each processor to the applications that they can bind their threads

onto, which is described in Algo. 1. The second phase decides the actual thread-core bindings, and it is described in Algo. 2.

Algorithm 1. Assign Core-Quota of Each Processor to Applications

Input: $Pref(app)$: scheduling preference of application app
Input: $PrevCoreQuota(app, p)$: In the previous scheduling, how many threads of application app had been bound to cores in processor p
Result: $CoreQuota(app, p)$: In the new scheduling, how many threads of application app could be bound to cores in processor p

```
1  m ←number of total processor cores;
2  foreach application, app, in descending order of |Pref(app)| do
3      CandidateCnt(app) ← min{m, number of threads in app};
4      m ← m − CandidateCnt(app);
5  foreach processor, p, do AvailCoreCnt(p) ← number of cores in p ;
6  foreach application, app, in descending order of |Pref(app)| do
7      foreach processor, p, do CoreQuota(app, p) ← 0 ;
8      n ← CandidateCnt(app);
9      while n > 0 do
10         p_0 ← arg max_p ⟨AvailCoreCnt(p), PrevCoreQuota(app, p)⟩;
11         if Pref(app) > 0 then
12             t ← 1;
13         else
14             t ← min{n, AvailCoreCnt(p_0)};
15         AvailCoreCnt(p_0) ← AvailCoreCnt(p_0) − t;
16         CoreQuota(app, p_0) ← CoreQuota(app, p_0) + t;
17         n ← n − t;
```

If the number of currently running threads in all applications is larger than the number of processor cores, SWOMPS will firstly satisfy applications with larger $|Pref(app)|$. The rest threads are not bound to any processor core and left to OS for scheduling. This is carried out by assigning $CandidateCnt(app)$ in the first loop (line 2 to 4) in Algo. 1. The loop iterates the applications from larger $|Pref(app)|$ to smaller ones and assigns core quota to each application. Next SWOMPS further assigns the core quota in each processor to the applications to determine how threads of an application spread among the processors. Again this assignment is carried out from larger $|Pref(app)|$ to smaller ones. For a specific application, as is described from line 9 to 17, SWOMPS will first find a processor that has the most cores unassigned. If there are multiple candidates, SWOMPS will choose one that the application has most thread bound to in the previous scheduling. When SWOMPS recalculats the scheduling plan of a process due to starting or terminating of other processes, it uses $PrevCoreQuota(app, p)$ to avoid unnecessary thread migration. Pseudo code in line 10 accomplishes this goal by finding the maximal tuple $\langle AvailCoreCnt(p), PrevCoreQuota(app, p)\rangle$ and return the relative processor p_0. SWOMPS will assign either one core or as many as possible to the application app according to the scheduling preference of the application and repeat until running out the application's core quota.

There are two steps in deciding the thread-core bindings. In the first step, as described from line 3 to 13 in Algo. 2, if an application has its threads bound to some core in the previous scheduling, these threads are bound to the same core as long as the application has core quota in the processor. In the second step, the rest of the threads are bound to cores where the application has core quota.

Algorithm 2. Generate Thread-Core Bindings

Input: $PrevThreadBind(t)$: the core that thread t bound to in the previous scheduling
Input: $CoreQuota(app, p)$: In the new scheduling, how many threads of application app could be bound to cores in processor p
Result: $ThreadBind(t)$: the core that thread t is bound to in the new scheduling

```
1  RestCoreQuota(.,.) ← CoreQuota(.,.);
2  UnboundCores(.) ← Cores(.);
3  foreach application, app, do
4      UnboundThreads(app) ← {threads of app} ;
5      foreach thread t of app do
6          if PrevThreadBind(t) ≠ ∅ then
7              c ← PrevThreadBind(t);
8              p ← processor that core c belongs to;
9              if RestCoreQuota(app, p) > 0 then
10                 RestCoreQuota(app, p) ← RestCoreQuota(app, p) − 1;
11                 UnboundThreads(app) ← UnboundThreads(app) − {t};
12                 UnboundCores(p) ← UnboundCores(p) − {c};
13                 ThreadBind(t) ← c;
14 foreach application, app, do
15     foreach processor, p, do
16         for i ← 1 to RestCoreQuota(app, p) do
17             t ← any element in UnboundThreads(app);
18             c ← any element in UnboundCores(p);
19             UnboundThreads(app) ← UnboundThreads(app) − {t};
20             UnboundCores(p) ← UnboundCores(p) − {c};
21             ThreadBind(t) ← c;
```

4 SWOMPS Evaluation

We have implemented our prototype scheduler, SWOMPS, with Open64 compiler and evaluated it on a SunFire X4440 server. The configuration of the server can be found in Table 3. The server is equipped with four processors. Each processor has six cores. Each core has separated L1 and L2 caches, and the 6 cores share the L3 cache. The four processors are configured in NUMA architecture. Each processor has 12G local memory. The operating system is Red Hat Enterprise Linux Server 5.4. The version of the Linux kernel is 2.6.18. The operating system allocates new page on the node where the task is running. The OpenMP runtime library is the default library used in Open64 of revision 2722.

We use benchmarks in SpecOMP 2001 test suite in our evaluation. Scheduling preference of each benchmark is set to the performance improvement when the application is properly scheduled. For example, as showed in Tab. 2, *324.apsi* prefers gather-scheduling, the performance improvement is 8.26%, so its scheduling preference is set to 8. While *312.swim* prefers scatter-scheduling, the performance improvement is 114.48%, so its scheduling preference is set to -114. Here we assume that users know application's running characteristic well when it is run exclusively and the scheduler can be guided by users.

4.1 Pairwise Execution

We firstly evaluate the scheduler by concurrently running two applications, each application with 12 threads. We sum up the execution time of the two benchmarks and compare it with the same test without SWOMPS' scheduling. We tested every pair of

Table 3. Experimenting Platform Configuration

Number of Sockets	4
Processor	AMD Opteron 8431
Number of Cores	6-core×4
L1 Cache Configuration	64K×6
L2 Cache Configuration	512K×6
L3 Cache Configuration	6M, Shared
Memory Configuration	12G×4

the 11 benchmarks in SpecOMP 2001. Table 4 lists the performance comparison of tests with and without SWOMPS' scheduling. For each cell in the table, the benchmark listed in the head of its row and its column are the benchmarks that are concurrently executed.

Table 4. Performance Comparison of Tests with and without SWOMPS in Pairwise Execution

Benchmark	310.wupwise	312.swim	314.mgrid	316.applu	318.galgel	320.equake	324.apsi	326.gafort	328.fma3d	330.art
312.swim	1.24									
314.mgrid	1.27	1.24								
316.applu	1.11	1.15	1.23							
318.galgel	1.15	1.08	1.24	1.04						
320.equake	1.17	1.31	1.36	1.32	1.11					
324.apsi	1.05	1.28	1.46	1.04	1.00	1.13				
326.gafort	1.17	1.11	1.05	1.08	1.07	1.02	1.12			
328.fma3d	1.15	1.11	1.31	1.13	1.09	1.26	1.06	1.19		
330.art	1.28	1.29	1.41	1.03	1.01	1.22	1.13	1.23	1.15	
332.ammp	1.12	1.40	1.13	1.16	1.11	1.13	1.07	1.11	1.04	1.17

Performance improvement can be observed in 54 out of the 55 testings, except the testing of <*324.apsi, 318.galgel*>. The maximum performance improvement is 46.3% (<*324.apsi, 314.mgrid*>). And the average improvement is 16.5%. SWOMPS showed its efficiency for two concurrently running applications.

4.2 Task Queue Simulation

To further evaluate SWOMPS, we test it in a more complicated running circumstance. We simulates a task queue. Benchmarks enter the queue in a random order, and the time interval between two successively entered benchmarks yields to exponential distribution. The expected value of the distribution is set to 360 with the intent of the queue being empty occasionally. There are at most three applications running currently. Each application has a random number of threads, either 6, 8 or 10. The benchmarks are also from SpecOMP 2001.

In addition to compare testing with and without SWOMPS' scheduling we also compare SWOMPS with a non system-wide scheduler implementation. We evaluate whether the lightweight, non system-wide implementation could be an alternative of SWOMPS. In the non system-wide implementation, the scheduler is linked to the application and becomes a part of it. There is no coordination between different applications. The scheduler can only schedule the application it belongs to. It schedules the

application according to the system state when the application starts execution. And that scheduling will not change till the application terminates.

We run each task queue test three times under different scheduling schemas: no scheduling, non system-wide scheduling and SWOMPS scheduling. We totally generated 10 task queue tests. Figure 3 shows the sum of execution time of the 11 benchmarks in each task queue test. Bars labeled with *No Scheduling* are the results of tests without any scheduling. Bars labeled with *Non SysWide* are the results of tests with non system-wide scheduling. And bars labeled with *SWOMPS* are the results of tests with SWOMPS scheduling.

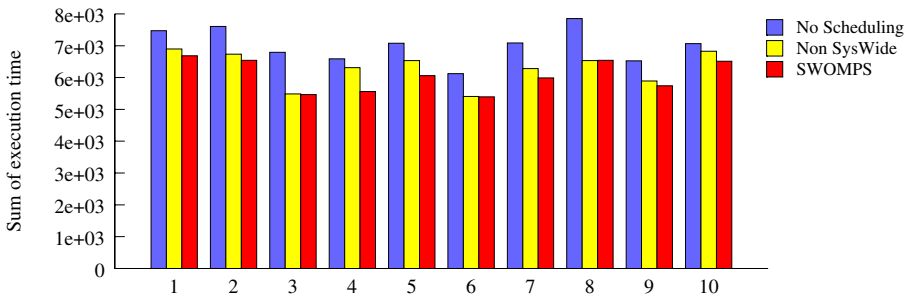


Fig. 3. Sum of Benchmark Execution Time in 10 Random Tests

Both the *Non SysWide* and *SWOMPS* scheduling outperform *No Scheduling* in all the 10 tests. The maximum time reduction is 24%, observed in test 3, for both *Non SysWide* and *SWOMPS*. The minimum time reduction is observed in test 10, 3% for *Non SysWide* and 8% for *SWOMPS*. The average time reduction are 12% for *Non SysWide* and 16% for *SWOMPS*. SWOMPS showed its efficiency in complicated circumstance too.

SWOMPS outperforms *Non SysWide* scheduling. Coordinating concurrently running application system-widely can make better use of hardware resources. Deeper comparisons indicate that system-wide scheduling is necessary in two circumstances, 1) when scheduling preferences of different application conflicts and 2) when an application terminates and more cores are available.

5 Comparison of Exclusive and Concurrent Running Model

In this section, we study the impact of concurrently running multiple applications. There are two concerns in our study, 1) throughput, measured by the reciprocal of the time needed to finish a group of task, and 2) performance of individual application in the concurrent execution.

We firstly enumerate every pair of the 11 benchmarks in SpecOMP 2001 as a task group. The two benchmarks first run exclusively, one after the other. Benchmarks *312.swim, 314.mgrid, 318.galgel* and *320.equake* are executed with 12 threads as their performances are better with 12 threads than with 24 threads. Then the two benchmarks are executed concurrently under SWOMPS' scheduling, each benchmark with

Table 5. Throughput and Performance Study of Pairwise Execution

(a) Throughput comparison between exclusively execution and pairwise execution with 12 threads for each application

Benchmark	310.wupwise	312.swim	314.mgrid	316.applu	318.galgel	320.equake	324.apsi	326.gafort	328.fma3d	330.art
312.swim	1.08									
314.mgrid	1.18	1.07								
316.applu	1.01	1.06	1.07							
318.galgel	1.09	1.00	1.18	0.97						
320.equake	1.02	1.13	1.13	1.07	0.91					
324.apsi	1.07	1.25	1.20	1.14	0.92	1.07				
326.gafort	0.99	1.01	1.04	0.96	1.11	0.81	0.94			
328.fma3d	1.22	1.20	1.22	1.15	1.07	1.08	1.11	0.95		
330.art	1.26	1.20	1.24	1.22	1.06	1.17	1.27	0.97	1.31	
332.ammp	1.05	1.06	1.25	0.93	1.27	0.89	0.90	1.14	1.12	1.03

(b) Performance reduction of individual benchmark due to concurrent execution

Benchmark	310.wupwise	312.swim	314.mgrid	316.applu	318.galgel	320.equake	324.apsi	326.gafort	328.fma3d	330.art
312.swim	17, 65									
314.mgrid	14, 58	40, 35								
316.applu	30, 54	58, 33	49, 26							
318.galgel	22, 42	45, 14	43, 16	35, 32						
320.equake	28, 36	68, 12	53, 7	52, 28	38, 27					
324.apsi	47, 34	61, 4	52, 3	55, 25	38, 27	45, 22				
326.gafort	39, 54	56, 31	56, 35	45, 41	43, 33	41, 50	32, 50			
328.fma3d	30, 44	57, 21	43, 15	43, 34	32, 27	28, 33	28, 44	44, 44		
330.art	33, 35	63, 8	54, 8	46, 25	35, 23	28, 26	23, 38	35, 39	36, 27	
332.ammp	34, 42	53, 1	52, 6	46, 21	34, 24	34, 36	34, 37	34, 44	34, 41	34, 30

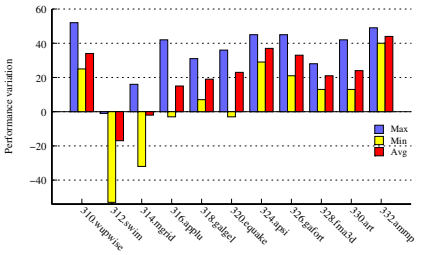
12 threads. We normalize the throughput of the concurrent run with respect to the exclusive run and the results are listed in Table 5a. In 43 of the 55 tests, concurrent run achieves better throughput. The best throughput improvement is 31% and the average throughput improvement is 9%.

We also tested concurrent run without SWOMPS' scheduling. Due to space limit, the detail of the result is not presented. In those tests, none of the concurrent run has better throughput. The average throuughput decreases by 32%. Simply running multiple applications concurrently does not guarantee throughput improvement.

We compare the performance of individual benchmark between exclusive run and concurrent run. The results are listed in Table 5b. For example, "17, 65" in the second row and second column indicates that, compared with exclusive run, when *312.swim*

Benchmark sequence in the task queue	N_T
332, 316, 326, 314, 328, 320, 330, 324, 310, 318, 312	1.56
328, 330, 326, 312, 320, 316, 332, 324, 318, 310, 314	1.50
312, 314, 328, 318, 310, 330, 320, 316, 326, 332, 324	1.54
326, 332, 330, 320, 314, 312, 310, 318, 324, 316, 328	1.49
330, 332, 318, 324, 310, 312, 320, 316, 314, 326, 328	1.49
330, 310, 324, 332, 328, 312, 314, 326, 316, 320, 318	1.35
312, 314, 330, 310, 324, 328, 316, 332, 320, 326, 318	1.39
332, 316, 314, 324, 328, 326, 312, 318, 310, 320, 330	1.54
316, 312, 320, 332, 326, 324, 310, 318, 330, 328, 314	1.50
326, 316, 320, 310, 314, 332, 318, 324, 328, 330, 312	1.56

(a) Throughput comparison



(b) Performance variation

Fig. 4. Throughput and Performance Study of Task Group with 11 Benchmarks

and *310.wupwise* run concurrently, performance of *312.swim* decreases 15%, and performance of *310.wupwise* decreases 65%. In the 110 comparisons, 90 of them are less than 50%, and the average performance decrease is 35.6%. This result is better than expected considering only half of the threads running and the L3 cache and memory bandwidth are shared among two benchmarks.

We next use all the 11 benchmarks as a task group. The 11 benchmarks enter a task queue in a random order. In the exclusive run, benchmarks *312.swim, 314.mgrid, 318.galgel* and *320.equake* are executed with 12 threads, others are executed with 24 threads. At the same time, only one benchmark is running. In the concurrently run, each benchmark runs with 12 threads. At the same time, there are two benchmarks running. When a benchmark terminates, the next benchmark in the task queue starts execution immediately. We tested 10 randomly generated testings and normalize the throughput of the concurrent run with respect to the exclusive run. Figure 4a lists the order of the 11 benchmarks in the queue in the first column. The second column lists the normalized throughput. On average, concurrently run has 49% improvement in throughput.

Figure 4b shows the performance variations of concurrent run compared with exclusive run. Bars labeled with *Max* give the maximum performance decrease of concurrent run in the 10 tests. Bars labeled with *Min* are the minimum performance decrease. Bars labeled with *Avg* are the average performance decrease. A negative value means benchmark runs faster in the concurrent run. Significant performance improvements have been observed in the concurrent run of benchmark *312.swim* and *314.mgrid*. A further study shows the major source of the improvement is thread binding. As we can see form the figure, the average performance decrease of the 11 benchmark are all less than 50%.

These experiments show that concurrent running model outperforms exclusive running model in throughput, especially if there are many tasks. Meanwhile, concurrent running will slow down the applications but they are reasonable.

6 Related Work

Many researches [1,2,3,4,5,6] have been carried out to improve the performance of OpenMP applications on multi-core system. Truong et al. [1] seek to improve scalability from implementation aspect. Their work introduces *thread subteams* to overcome the thread mapping problem and enhance modularity. Noronha et al. [2] study the benefits from using large page support for OpenMP applications. Terboven et al. [4] improve data and thread affinity of OpenMP programs on multi-core system by binding thread to thread cores and allocate memory with *next touch* strategy. A series of papers [3,5,6] from Broquedis et al. introduce a runtime system that transpose affinities of thread teams into scheduling hints. With the help of the introduced *BubbleSched* platform, they propose scheduling strategy suited to irregular and massive nested parallelism over hierarchical architectures. They also propose a NUMA-aware memory management subsystem to facilitate data affinity exploitation.

To the authors' knowledge, there was no study about improving concurrently running multiple OpenMP applications.

7 Conclusion

Restrictions of heating and power consumption has converted the effect of Moore's law from increasing the performance of a single-core processor to the number of cores in a processor. More and more cores will be available in one processor. However, a preliminary experimental study of the scalability of OpenMP applications shows that OpenMP applications cannot efficiently utilize the increasing processor cores in general. Concurrently running multiple OpenMP applications will become a common usage model.

In the NUMA architecture with multi-core processors, the processor cores are asymmetric. How the threads of the concurrently running OpenMP applications are distributed on the processor cores is important to the performance of all the current applications. In this paper, we proposed a system-wide scheduler, SWOMPS, to help schedule the threads on the processor cores. The scheduler makes its decision based on underlying hardware configuration as well as the hints of scheduling preference of each application. Experiment results shows that SWOMPS is efficient in improving the whole system performance. We also compared the exclusive running and concurrent running with SWOMPS. In various experimental comparisons, concurrent execution outperforms in throughput, meanwhile the performance slowdown of individual application in concurrent execution is reasonable.

References

1. Chapman, B.M., Huang, L.: Enhancing OpenMP and its implementation for programming multicore systems. In: Bischof, C.H., Bücker, H.M., Gibbon, P., Joubert, G.R., Lippert, T., Mohr, B., Peters, F.J. (eds.) PARCO. Advances in Parallel Computing, vol. 15, pp. 3–18. IOS Press, Amsterdam (2007)
2. Noronha, R., Panda, D.K.: Improving scalability of OpenMP applications on multi-core systems using large page support. In: IPDPS, pp. 1–8. IEEE, Los Alamitos (2007)
3. Thibault, S., Broquedis, F., Goglin, B., Namyst, R., Wacrenier, P.A.: An efficient OpenMP runtime system for hierarchical architectures. In: Chapman, B., Zheng, W., Gao, G.R., Sato, M., Ayguadé, E., Wang, D. (eds.) IWOMP 2007. LNCS, vol. 4935, pp. 161–172. Springer, Heidelberg (2008)
4. Terboven, C., an Mey, D., Schmidl, D., Jin, H., Reichstein, T.: Data and thread affinity in OpenMP programs. In: MAW '08: Proceedings of the 2008 workshop on Memory access on future processors, pp. 377–384. ACM, New York (2008)
5. Broquedis, F., Diakhaté, F., Thibault, S., Aumage, O., Namyst, R., Wacrenier, P.A.: Scheduling dynamic OpenMP applications over multicore architectures. In: Eigenmann, R., de Supinski, B.R. (eds.) IWOMP 2008. LNCS, vol. 5004, pp. 170–180. Springer, Heidelberg (2008)
6. Broquedis, F., Furmento, N., Goglin, B., Namyst, R., Wacrenier, P.A.: Dynamic task and data placement over numa architectures: An OpenMP runtime perspective. In: Müller, M.S., de Supinski, B.R., Chapman, B.M. (eds.) IWOMP 2009. LNCS, vol. 5568, pp. 79–92. Springer, Heidelberg (2009)
7. Hanawa, T., Sato, M., Lee, J., Imada, T., Kimura, H., Boku, T.: Evaluation of multicore processors for embedded systems by parallel benchmark program using OpenMP. In: Müller, M.S., de Supinski, B.R., Chapman, B.M. (eds.) IWOMP 2009. LNCS, vol. 5568, pp. 15–27. Springer, Heidelberg (2009)

8. Terboven, C., an Mey, D., Sarholz, S.: OpenMP on multicore architectures. In: Chapman, B., Zheng, W., Gao, G.R., Sato, M., Ayguadé, E., Wang, D. (eds.) IWOMP 2007. LNCS, vol. 4935, pp. 54–64. Springer, Heidelberg (2008)
9. Curtis-Maury, M., Ding, X., Antonopoulos, C.D., Nikolopoulos, D.S.: An evaluation of OpenMP on current and emerging multithreaded/multicore processors. In: Mueller, M.S., Chapman, B.M., de Supinski, B.R., Malony, A.D., Voss, M. (eds.) IWOMP 2005 and IWOMP 2006. LNCS, vol. 4315, pp. 133–144. Springer, Heidelberg (2008)