IPSJ SIG Technical Report

Vol.2013-HPC-140 No.10
2013/7/31

# A Performance Analyzer for Task Parallel Applications based on Execution Time Stretches

An Huynh[1,a)] Jun Nakashima[1,b)] Kenjiro Taura[1,c)]

**Abstract:** Performance loss in task parallel applications is contributed by 3 factors of thread idleness, parallelism overhead and work time stretch. Thread idleness is the time that threads have no work to do and parallelism overhead is the time that threads spend on extra instructions that would not be necessary in serial execution. The third factor, work time stretch, refers to the surplus time by which the same application-level code takes longer in parallel execution than in serial execution. We believe that work time stretch is the most important factor in future multi-core systems. Therefore, we have developed a profiler that analyzes work time stretch of task parallel applications. The profiler can clarify the contribution of work time stretch factor out of the other two, attributing stretched amount to specific code blocks so that programmers can know which parts of their programs are stretching. It also shows the surplus cache miss count that accompanies work time stretch.

**Keywords:** task parallelism, performance analyzer, execution time stretch, application instrumentation, library instrumentation, cache miss count

## 1. Introduction

Along with the development of computer system to multi-core multi-socket architectures, the need for an intuitive, high-level parallel programming model has risen. Task parallelism which is easy to use and very fit with divide-and-conquer algorithms has been adopted widely.

In task parallelism, a programmer's job is just to indicate the works that can be executed in parallel, grouping serial works into functions which are executed by tasks. Programmer can create as many tasks as they want to suit their algorithms. The other burdens of parallelism overhead and load balance are taken by runtime libraries. Runtime libraries take care of creating, synchronizing, terminating tasks with as low overhead as possible, and also does balancing load between all available cores by migrating tasks from busy cores to free cores. Most task parallel runtime libraries use work stealing [1] method which has been proved to be an efficient and scalable task scheduling strategy. In short, in work stealing a free core chooses a random victim core, then go to run queue of that victim, steal a work from that, bring it back and execute it. If the victim's run queue is empty, it would continue choosing other victims until it can succeed in stealing a work.

Modern task parallel libraries have done well regarding these parallelism overhead and thread idleness. A good implementation can keep parallelism overhead at least, and a suitable scheduling strategy can help achieve near-perfect load balancing, resulting a perfect scalability with a condition that applications have been well parallelized with fine-grained tasks.

However, there is one more factor that is usually unaware affecting task parallel performance. It is the fact that the same application-level code takes longer in parallel execution than in serial execution. We call this work time stretch, which implies the surplus amount of time that parallel execution takes compared with serial execution.

Along with the increasing number of cores on shared memory systems, and the unpredictable migrations of tasks done by work stealing scheduling strategy, work time stretch will become more and more important, but harder to analyze in future multi-core multi-socket systems. Therefore, we have created a profiler with the aim to analyze work time stretch factor of task parallel applications by comparing parallel execution with serial execution. The profiler can observe any specific code blocks of application and report that code block's stretching amount to users. It can also report counts of any hardware event that was occurred by that code blocks.

## 2. Related Work

There have been many works and products regarding performance analyzers. Two popular ones are the TAU performance system [2] and Intel VTune Amplifier software [3]. TAU is open source and has a powerful automatic instrumentation toolset. VTune Amplifier uses sampling method and does not need to instrument the executable. However, these tools analyze and evaluate only one execution of application. For example, they can pinpoint the most costly code

1 The University of Tokyo
a) huynh@eidos.ic.i.u-tokyo.ac.jp
b) nakashima@eidos.ic.i.u-tokyo.ac.jp
c) taura@eidos.ic.i.u-tokyo.ac.jp

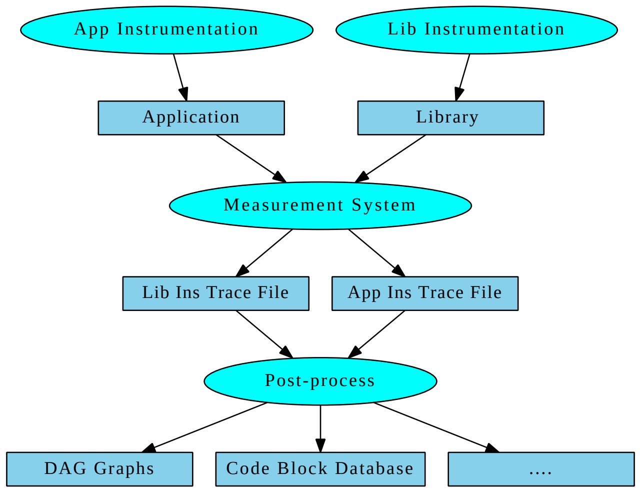


**Fig. 1** Profiler Structure

blocks in the application, which consume most of the execution time. Our approach is different, as we compare the executions on one and many cores, then we analyze performance basing on changes of work time in these 2 executions.

We have implemented our profiler in MassiveThreads [4] task parallel library which is a good implementation of a task parallel library (thread creation and destruction cost only 72 nanoseconds). We use PAPI [5] library to get hardware event counter values, and OTF (Open Trace Format) [6] library to store profile data in .otf format. OTF format has good scalability in dealing with huge data which are usually produced by any profiler. Besides, OTF also integrates Zlib [7] compression which will help to reduce the data size stored on disk.

The authors in [8] have identified the contributions of the 3 factors (overhead, idlness and work time stretch) in the performance loss of applications in BOTS [9] suite. They demonstrated that work time stretch accounted for a dominant part and proposed a locality-based scheduler which can mitigate this factor. While, our profiler can help identify work time stretch in all kinds of applications. We also go a step further that is indicating that increase in cache miss count is the reason causing work time stretch.

## 3. Profiler Design

The profiler includes 3 parts of instrumentation, measurement and post-process. Instrumentation's role is to put profiler's measurement functions to appropriate positions in library's code or application's code, so that measurement system can know occurences of events at runtime and carry out its job of collecting and storing profile data. The last part, post-process, investigates profile data stored by measurement system and extracts useful information. **Fig. 1** shows a general view about this profiler's design.

The profiler provides 2 kinds of instrumentations. One is library instrumentation and another one is application instrumentation. While library instrumentation can be done by profiler's developers, application instrumentation needs to be conducted by profiler's users.

```
void * profiler_task_begin(parent_node,
    spawn_index);
char  profiler_task_spawn(node_ptr);
void  profiler_task_sync(node_ptr, ...);
void  profiler_task_end(node_ptr);
void  profiler_task_resume(node_ptr);
void  profiler_task_pause(node_ptr);
```

**Fig. 2** App Instrumentation API

### 3.1 Library instrumentation

Library instrumentation is the instrumentation of measurement code done in task parallel libraries. When users run their application along with an instrumented library, the profiler's measurement system running behind-the-scene silently observes and collects profile data basing on the instrumentation codes that have been injected in the library. By this way, users don't need to change anything in their application code.

We have done this instrumentation in MassiveThreads library.

### 3.2 Application instrumentation

On the other hand, in application instrumentation we put measurement code to application code directly. This helps measurement system to have easy access to application code-level information, such as file name, function name, line number. Because this kind of instrumentation is relatively independent to runtime library, it is possible to use even with libraries which the profiler does not support.

Besides, in divide-and-conquer programming model, it is not always the case that applications call task creation for each recursive call. For example, imagining a binary task tree model in which each task spawns 2 child tasks, it is not rare that programmers write the code as such that each task calls library's task-creating function only for the first child, and it executes the second child task by itself. This helps reducing the number of tasks to be created, cutting tasks that do not do any real work, but just waits for synchronization. However, by doing this the library becomes unaware of the second child. Hence, task tree structure from the library's point of view becomes different from that of programmer's point of view. Of course, this will make profile data less meaningful to users.

Fortunately, this is not a problem with application instrumentation. By doing appropriate code insertions, users can make measurement system consider the second recursive call as a normal task spawning.

Application instrumentation provides a more flexible method, but with a tradeoff that users need to insert instrumentation codes by themself.

#### App instrumentation API

Application instrumentation API is shown in **Fig. 2**.

Its usage rule is quite simple: put *profiler_task_begin()* at the beginning of functions, put *profiler_task_spawn()* right before any recursive call (both calls that spawn a real task, or is just purely a function call), put *profiler_task_sync()* right after any synchronization instruc-

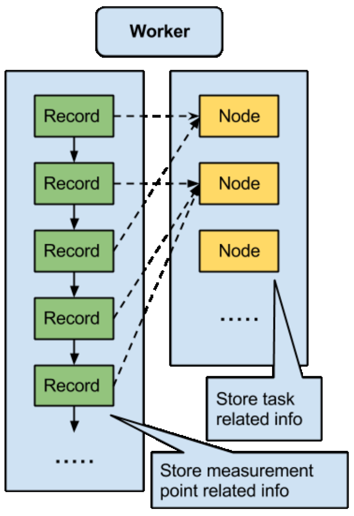


**Fig. 3** Measurement System

tion, put *profiler_task_end()* at the end of functions. *profiler_task_resume()* and *profiler_task_pause()* are used to instrument at right after a spawn and right before a sync respectively. These two functions can be abbreviated if the code block right after a spawn and/or right before a sync is too small or has nothing to measure.

It is also necessary to extend function arguments with two new arguments. One is a pointer returned by *profiler_task_begin()*, and one is a spawn index returned by *profiler_task_spawn()*.

The profiler considers *begin*, *sync*, *resume* as the beginnings of code blocks, and *spawn*, *pause*, *end* as the ends of code blocks.

Beside measuring measurement metrics that are associated with code blocks, the profiler's target is also to record **spawn** and **sync** events. Then basing on these data the profiler can generate a DAG graph that illustrates the relationship between not only tasks but also code blocks inside tasks.

## 3.3 Measurement system

Profiler's measurement system runs along with the execution of application. It gets time and collects other hardware event counters at points where measurement codes were inserted. It stores these data in memory as long as the data amount is not so large that memory thrashing may occur. If memory state gets bad, or execution has finished, it writes data from memory to file.

We pay attention to where to place time getting instructions (and other hardware counter getting instructions). It means that we put them at the end of measurement functions which measure the start point of code blocks, and at the beginning of measurement functions which measure the stop point of code blocks, so that the overhead of instrumentation functions is not included into measured data.

**Fig. 3** describes the structure of data that measurement system stores in memory.

The structure includes 2 lists of node list and record list. Each *node* holds information related to a particular task. The term *task* here means a recursive call of function in application instrumentation, and a task created in runtime system in library instrumentation. Information held by *node* are usually file name, function name, task level, *tree path*, etc. Especially, *tree path* is the most important. It is used to identify and distinguish a specific task from others. *tree path* is a string of continuous spawn indices seperated by underscore along which the current task has been created.

Each *record* holds data measured at a measurement point. Besides, *record* also holds pointer to the *node* of the task that it belongs to.

*Records* are deleted after they have been written to file. *Nodes* are deleted only when they have no *record* referring to them anymore and the task that it represents has finished.

Each worker thread (core) maintains 2 copies of this structure. One for library instrumentation and one for application instrumentation.

## 3.4 Post-process

Post-process' role is to analyze profile data, extract useful information and show it to users.

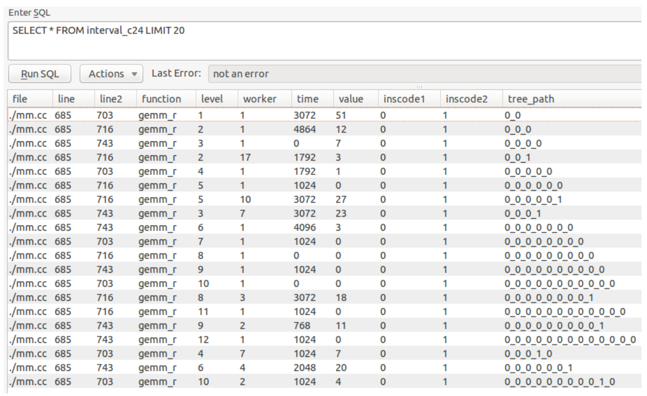
Profile data stored in OTF format by measurement system are a series of data (time, hardware event count) which are associated with points of time. Post-process first reads this series of point data from OTF format, pairing appropriately two adjacent point data to form an interval data.

An interval data represents **one execution of a code block**, it holds measurement metrics such as execution time of that code block, cache miss count that occurred in that execution of that code block. Besides, interval data also include information that help to trace back the positions of code blocks in application source code, such as source file name, function name, the beginning and ending line numbers of code blocks. One code block can be executed many times by many different tasks at many different task levels (task's depth). Therefore, in order to locate the right task where a code block was executed interval data also holds level and tree path of the task that has executed the code block.

In short, post-process transforms a series of point data to a series of interval data, and puts that series of interval data into an SQLite [10] database (**Fig. 4**) so that users can easily query necessary aggregated information. For example, users may want to get the sum execution time of all tasks in a particular level, or all tasks below a specific task (sub-tree). These requests can be done with ease using SQL queries.

One more feature of the profiler is that it can generate DAG graphs using dot language [11] which represent task tree structure along with its spawn-sync scenario. **Fig. 5** shows a spawn-sync scenario example.

This task first spawns 3 child tasks (0, 1, and 2). Next it synchronizes task 0 and task 1, leaving task 2 running. Then it spawns task 3 and task 4. There are now 3 child



Enter SQL

```
SELECT * FROM interval_c24 LIMIT 20
```

| file | line | line2 | function | level | worker | time | value | inscode1 | inscode2 | tree_path |
|---|---|---|---|---|---|---|---|---|---|---|
| ./mm.cc | 685 | 703 | gemm_r | 1 | 1 | 3072 | 51 | 0 | 1 | 0_0 |
| ./mm.cc | 685 | 716 | gemm_r | 2 | 1 | 4864 | 12 | 0 | 1 | 0_0_0 |
| ./mm.cc | 685 | 743 | gemm_r | 3 | 1 | 0 | 7 | 0 | 1 | 0_0_0_0 |
| ./mm.cc | 685 | 716 | gemm_r | 2 | 17 | 1792 | 3 | 0 | 1 | 0_0_1 |
| ./mm.cc | 685 | 703 | gemm_r | 4 | 1 | 1792 | 1 | 0 | 1 | 0_0_0_0_0 |
| ./mm.cc | 685 | 716 | gemm_r | 5 | 1 | 1024 | 0 | 0 | 1 | 0_0_0_0_0_0 |
| ./mm.cc | 685 | 716 | gemm_r | 5 | 10 | 3072 | 27 | 0 | 1 | 0_0_0_0_0_1 |
| ./mm.cc | 685 | 743 | gemm_r | 3 | 7 | 3072 | 23 | 0 | 1 | 0_0_0_1 |
| ./mm.cc | 685 | 743 | gemm_r | 6 | 1 | 4096 | 3 | 0 | 1 | 0_0_0_0_0_0_0 |
| ./mm.cc | 685 | 703 | gemm_r | 7 | 1 | 1024 | 0 | 0 | 1 | 0_0_0_0_0_0_0_0 |
| ./mm.cc | 685 | 716 | gemm_r | 8 | 1 | 0 | 0 | 0 | 1 | 0_0_0_0_0_0_0_0_0 |
| ./mm.cc | 685 | 743 | gemm_r | 9 | 1 | 1024 | 0 | 0 | 1 | 0_0_0_0_0_0_0_0_0_0 |
| ./mm.cc | 685 | 703 | gemm_r | 10 | 1 | 0 | 0 | 0 | 1 | 0_0_0_0_0_0_0_0_0_0_0 |
| ./mm.cc | 685 | 716 | gemm_r | 8 | 3 | 3072 | 18 | 0 | 1 | 0_0_0_0_0_0_0_0_1 |
| ./mm.cc | 685 | 716 | gemm_r | 11 | 1 | 1024 | 0 | 0 | 1 | 0_0_0_0_0_0_0_0_0_0_0_0 |
| ./mm.cc | 685 | 743 | gemm_r | 9 | 2 | 768 | 11 | 0 | 1 | 0_0_0_0_0_0_0_0_0_1 |
| ./mm.cc | 685 | 743 | gemm_r | 12 | 1 | 1024 | 0 | 0 | 1 | 0_0_0_0_0_0_0_0_0_0_0_0_0 |
| ./mm.cc | 685 | 703 | gemm_r | 4 | 7 | 1024 | 7 | 0 | 1 | 0_0_0_1_0 |
| ./mm.cc | 685 | 743 | gemm_r | 6 | 4 | 2048 | 20 | 0 | 1 | 0_0_0_0_0_0_1 |
| ./mm.cc | 685 | 703 | gemm_r | 10 | 2 | 1024 | 4 | 0 | 1 | 0_0_0_0_0_0_0_0_0_1_0 |

**Fig. 4** Execution interval database

```
void some_function()
{
        ..
        spawn_task(0)
        spawn_task(1)
        ..
        spawn_task(2)
        ..
        sync_tasks(0,1)
        ....
        spawn_task(3)
        ....
        spawn_task(4)
        ....
        sync_tasks(2,4)
        ......
        sync_tasks(3)
        ........
}
```

**Fig. 5** Spawn-sync scenario example

tasks (2, 3, 4) that are still running. Then it synchronizes task 2 and task 4. After that it continues to synchronize the last task 3.

Post-process divides a task into many parts by sync_tasks instructions, each part is mapped to one separate node in DAG graph. In this case, the above task is divided into 4 nodes, the first node contains 4 code blocks of

- before spawn_task(0)
- between spawn_task(0) and spawn_task(1)
- between spawn_task(1) and spawn_task(2)
- between spawn_task(2) and sync_task(0,1)

On graph, a node can display any information, for example, aggregate execution time of all code blocks that it contains.

**Fig. 6** shows a pseudo-DAG graph corresponding to the above code.

## 4. Case Study: Matrix Multiplication Application

We have used this profiler to evaluate our matrix multiplication (*mm*) application. Experimental environment settings is shown in **Table 1**.

This mm application multiplies 2 matrices of size 3200 which are generated with random values (A × B = C). It uses divide-and-conquer algorithm. At each computation step, it divides data into 2 halves, and call one child task to

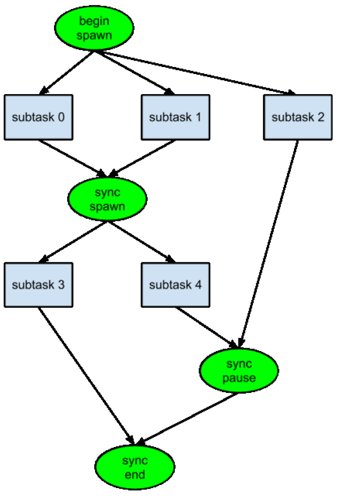


**Fig. 6** Spawn-Sync DAG graph

**Table 1** Experiment environment

| | |
|---|---|
| CPU | AMP Opteron 8354 (Barcelona) 2.2GHz<br>4 cores × 8 sockets = 32 cores |
| Memory | 32GB × 4 = 128GB |
| Cache | L1: Data:64KB/core, Instruction:64KB/core<br>L2: 512KB/core<br>L3: 2MB/socket |
| OS | Linux 2.6.32(Debian GNU/Linux) |

```
void gemm_r(...)
{
        ...
        spawn_task(...)
        spawn_task(...)
        sync_tasks()
        ...
}
```

**Fig. 7** Original recursive function gemm_r()

calculate each half. Specificly, it may divide matrix A vertically and multiply each half of A with B, otherwise it may divide matrix B horizontally and multiply A with each half of B. Therefore, task tree model of this application would look like a binary tree at which each tree node has 2 child nodes. A pseudo-code of its recursive function is shown in **Fig. 7**.

**Fig. 8** shows the scalability of this application. We can see that *mm* scales almost perfect at small numbers of cores. However, its scalability gets exhausted gradually from 16 cores when number of cores gets larger.

The reason of this degradation is what we want to know with this profiler. What makes a perfect performance on several cores degrade significantly when it comes to large number of cores?

We used the profiler's application instrumentation API to instrument this *mm* application. The instrumented code looks like in **Fig. 9**.

With problem size 3200, *mm* spawns tasks recursively until level 20, while level 0 has only one task which is the original running program. Leaf tasks, which do the real computation instead of dividing problem size and spawning tasks,

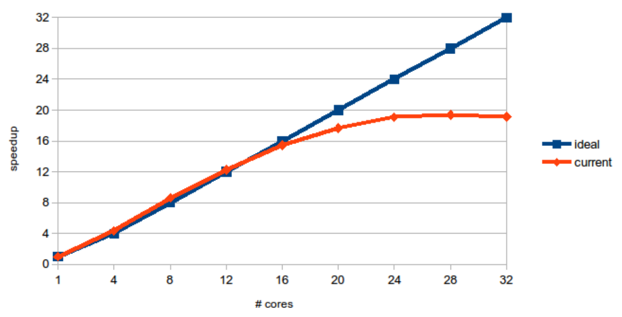


**Fig. 8** Scalability of *mm* (n=3200) on minnie machine

```
void gemm_r(..., void *parent_node, int
    spawn_index)
{
  void *node = profiler_task_begin(parent_node,
      spawn_index)
  ...
  int id_1 = profiler_task_spawn(node)
  spawn_task(..., node, id_1)
  int id_2 = profiler_task_spawn(node)
  spawn_task(..., node, id_2)
  sync_tasks()
  profiler_task_sync(node, id1, id2)
  ...
  profiler_task_end(node)
}
```

**Fig. 9** Instrumented recursive function gemm_r()

gather the most at level 19 and level 20.

We run *mm* on both 1 core and 32 cores with the profiler, setting the profiler to measure one hardware event of L3 cache miss. After running, the profiler generates some DAG graphs and a database containing all execution intervals of all tasks of *mm* application on 1-core and 32-core executions.

Because the number of tasks is very large, especially on high levels of task depth, it is not possible to always produce a DAG graph that contains all tasks. The profiler's DAG graph generator can be modified with user-specified level limitation.

**Fig. 10** and **Fig. 11** show such DAG graph where task level is limited at 3. All nodes with the same color belong to tasks at the same level. Each task has 2 nodes. The first node contains only one interval which is the code block from the beginning to the first spawn in *gemm_r()* function. The second node also contains only one interval which is the code block from after sync to the end. The task with gold color is the only task at level 1, which is the first call to recursive function *gemm_r*. Tasks of level 2 are in cyan color and tasks of level 3 are in orange color. Original task (level 0) is not drawn in the graph. The values on nodes are the sum execution time of all intervals belong to that node. However, green nodes represent aggregate values of all tasks below level 3.

Fig. 10 is the graph of serial execution on 1 core. And Fig. 11 is of the parallel execution on 32 core. By comparing these 2 graphs, we can grasp how much application's work

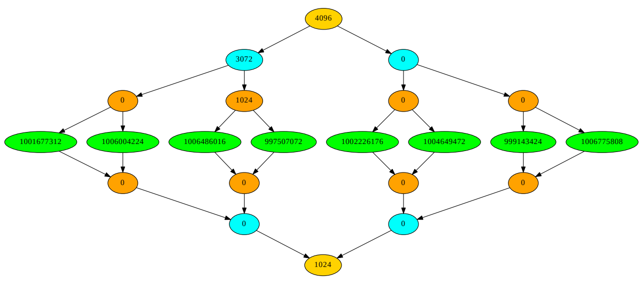


**Fig. 10** DAG with time (n=3200, cores=1)

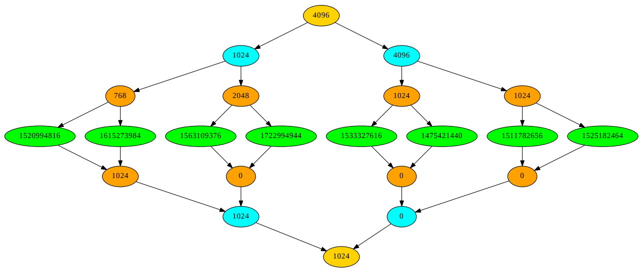


**Fig. 11** DAG with time (n=3200, cores=32)

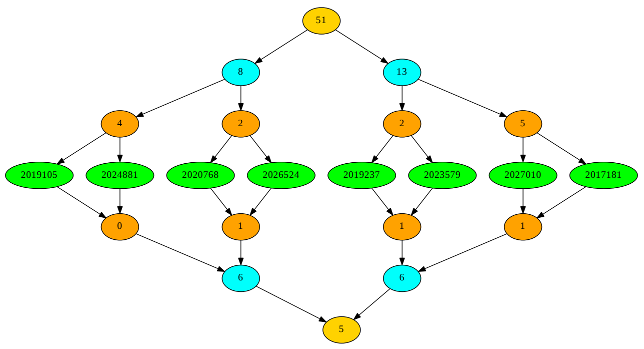


**Fig. 12** DAG with L3 cache miss count (n=3200, cores=1)

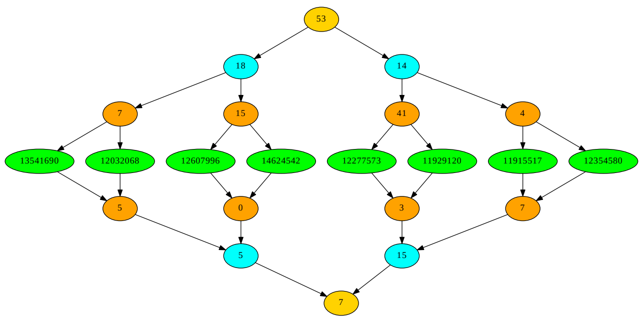


**Fig. 13** DAG with L3 cache miss count (n=3200, cores=32)

time has stretched.

Beside execution time, the profiler can also replace values on nodes to any other measured data. For example, **Fig. 12** and **Fig. 13** are such DAG graphs with L3 cache miss count shown on nodes. By comparing these 2 graphs, we can also understand that cache miss count increases along with work time stretch.

The database generated by the profiler contains two tables corresponding to 1 and 32-core executions.

Because *mm* application has only one recursive function, fields *file name* and *function name* are the same for all records, "mm.cc" and "gemm_r" respectively. Fields *begin-*

Vol.2013-HPC-140 No.10
2013/7/31

```
SELECT sum(time) FROM interval_c1 WHERE level
    =18;
```

**Fig. 14** Simple query

**Table 2** Work time of tasks at level 18, 19, 20 on 1 and 32 cores

| | Level 18 | Level 19 | Level 20 |
|---|---|---|---|
| 1 core | 42908416 | 6409045248 | 1527187456 |
| 32 cores | 50275072 | 9717456640 | 2637724928 |

(unit: nanosecond)

**Table 3** L3 cache miss count of tasks at level 18, 19, 20 on 1 and 32 cores

| | Level 18 | Level 19 | Level 20 |
|---|---|---|---|
| 1 core | 10764 | 8833358 | 7272947 |
| 32 cores | 667502 | 76261509 | 23482007 |

**Table 4** Comparing increased time and increased cache miss count

| | | Level 18 | Level 19 | Level 20 |
|---|---|---|---|---|
| R1 | Increased time | 7366656 | 3308411392 | 1110537472 |
| R2 | Increased CM | 656738 | 67428151 | 16209060 |
| R3 | R2 × 373ns | 244963274 | 25150700323 | 6045979380 |
| R4 | R3/R1 | 33.2 | 7.6 | 5.4 |

*ning line number* and *ending line number* (also *beginning instrument code* and *ending instrument code*) are either one of the four instrumented positions shown in Fig. 9, except the second *spawn* because it does not form any interval.

Let's try querying the total execution time of all tasks at level 18, 19 and 20 from the database (**Fig. 14**). Query result is shown in **Table 2**.

We can see that execution time of tasks at level 18, 19, 20 have stretched for 17.2% 51.6% and 72.7% respectively. **Fig. 3** shows the corresponding cache miss counts of tasks at level 18, 19 and 20.

In order to evaluate a bit more quantitatively about work time stretch and corresponding increase in cache miss count, we have measured the cost that one L3 cache miss takes (it is memory latency). Measurement result is 373 nanoseconds on our experiment machine. So by multiplying the increased amount of cache miss count with this latency we can acquire the time cost induced by L3 cache miss count increase (**Table 4**).

Surprisingly, the time cost calculated from cache miss count is many times larger than the real time stretch, 7.6 times and 5.4 times for level 19 and 20 respectively. Actually, this result expresses the parallelism in dealing with cache miss of computer processor. It is that one processor core can occur and wait for multiple cache misses at the same time, which is sometimes called outstanding cache miss.

## 5. Conclusion and Future Work

We have built a profiler that identifies work time stretch factor in task parallel applications. The profiler uses two kinds of instrumentations, library-level instrumentation and application-level instrumentation. to extract work time from applications' execution. It then compares work time of serial execution and parallel execution to identify the stretches. Beside work time, it also measures and compares other hardware event counters as an effort to find the cause of work time stretch.

The profiler provides insights of what code blocks stretch, how many hardware event counts increase. This may help programmers to find a way to improve their task parallel algorithm. Or either it may prove that we need a better scheduling strategy instead of the random manner of work stealing which potentially induces many surplus cache misses.

The profiler is now quite limited at analyzing measurement result. We will continue working to find better ways to express useful information to users. Currently we have known that work time stretch caused by cache miss count increase. Then what causes cache miss count to increase? This question requires us to go deeper in analyzing the task parallel executions.

We also intend to apply this profiler to analyze other applications like sorting and ExaFMM.

## References

[1] Blumofe, R. D. and Leiserson, C. E.: Scheduling multithreaded computations by work stealing, *J. ACM*, Vol. 46, No. 5, pp. 720–748 (online), DOI: 10.1145/324133.324234 (1999).

[2] Sameer S. Shende, A. D. M.: The Tau Parallel Performance System, *IJHPCA*, Vol. 20, No. 2, pp. 287–311 (online), available from ⟨http://www.cs.uoregon.edu/research/paracomp/papers/ijhpca05.tau/ijhpca$_t$*au.pdf*⟩ (2006).

[3] Intel: Intel VTune Amplifier, Intel Inc. (online), available from ⟨http://software.intel.com/en-us/intel-vtune-amplifier-xe⟩ (accessed 2013).

[4] Nakashima, J. and Taura, K.: Multithread Framework that Manages both Efficient I/O and Lightweight Thread Management, *Information Processing Society of Japan Transactions on Programming*, Vol. 4, No. 1, pp. 13–26 (2011).

[5] Browne, S., Dongarra, J., Garner, N., London, K. and Mucci, P.: A scalable cross-platform infrastructure for application performance tuning using hardware counters, *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '00, Washington, DC, USA, IEEE Computer Society, (online), available from ⟨http://dl.acm.org/citation.cfm?id=370049.370424⟩ (2000).

[6] Knüpfer, A., Brendel, R., Brunst, H., Mix, H. and Nagel, W. E.: Introducing the open trace format (OTF), *Proceedings of the 6th international conference on Computational Science - Volume Part II*, ICCS'06, Berlin, Heidelberg, Springer-Verlag, pp. 526–533 (online), DOI: 10.1007/11758525$_7$1(2006).

[7] Jean-loup Gailly, M. A.: zlib, zlib (online), available from ⟨http://www.zlib.net⟩ (accessed 2013).

[8] Olivier, S. L., de Supinski, B. R., Schulz, M. and Prins, J. F.: Characterizing and mitigating work time inflation in task parallel programs, *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, Los Alamitos, CA, USA, IEEE Computer Society Press, pp. 65:1–65:12 (online), available from ⟨http://dl.acm.org/citation.cfm?id=2388996.2389085⟩ (2012).

[9] Duran, A., Teruel, X., Ferrer, R., Martorell, X. and Ayguade, E.: Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP, *Parallel Processing, 2009. ICPP '09. International Conference on*, pp. 124–131 (online), DOI: 10.1109/ICPP.2009.64 (2009).

[10] Hipp, D. R.: SQLite, SQLite (online), available from ⟨http://www.sqlite.org⟩ (accessed 2013).

[11] Bilgin, A.: Graphviz - Graph Visualization Software, ATT Research (online), available from ⟨http://www.graphviz.org⟩ (accessed 2013).