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

An Huynh,† Jun Nakashima† and Kenjiro Taura†

1. Introduction

Speedup of parallel programs is rarely perfect, especially when an application executes on a large number of cores. As an example, **Fig. 1** shows the speedup of a recursive task parallel matrix multiply running on a node with 32 cores of AMD Opteron 8354. The speedup is nearly perfect until 16 cores, after which it begins to degrade.

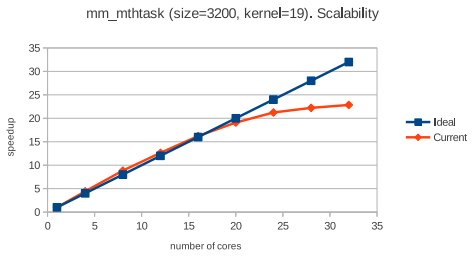

Fig. 1 Scalability

In general, a degradation of speedup is contributed by three factors: *idle time, overhead,* and *work time stretch.* The first factor, idle time, is the time that workers do not have any work to do. In task parallel systems, it is the time in which workers are trying to steal work from other workers. The second factor, overhead, refers to the time spent on extra instructions that would would not be necessary in serial execution. In task parallel systems, this includes the time to create/terminate tasks. The last factor, work time stretch, refers to the degree to which *the same application-level code* takes longer in parallel execution than in serial execution. A dominating cause of work time stretch is an increased cache miss ratio caused by task distribution (load balancing).

Among the three factors, we believe the work time stretch is most difficult to identify and most important for the programmer in future multi- and many-core systems. This is because the dynamic load balancing by the work stealing scheduler[1] changes which cores access which part of data, in a way not easy to predict by the programmer and analyzing/mitigating the effect of such changes in access pattern will be more important than ever in future. To this end, we are trying to develop a profiler that can clarify the contribution of work time stretch in application's slowdown and show the stretch factors of selected tasks.

† The University of Tokyo

2. Related Work

There have been many performance analysers. Two popular systems are the TAU system[2] and Intel VTune Amplifier[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 evaluate only one execution of application. For example, they analyse the most costly part of the application, figuring out the code blocks that consume most of the execution time. Our approach is different, we compare the executions on one and many cores, then we analyze performance basing on changes of work time.

3. Profiler Structure

The profiler's target is to collect information of individual runs of individual tasks so that it can distinguish work time out of idle time and runtime system's overhead. We have instrumented measurement code in MassiveThreads[4] task parallel runtime system. We instrumented at positions where a task is going to be started or stopped so that we can keep track on individual runs of tasks.

The profiler also pays attention to the length of linked lists. When it reaches a predefined limit (which is specified by environment variable), the profiler will pause worker thread, write data to file, releasing memory for new measured data, avoiding memory thrashing.

4. Detection of task execution time's stretch

We used the profiler to analyse the slowdown of matrix multiply application. The profiler has

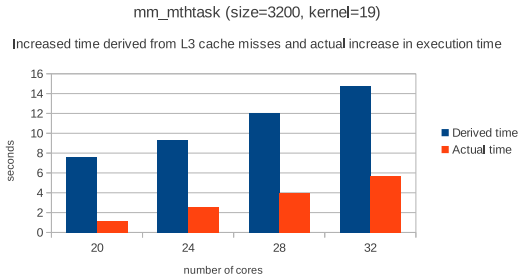

Fig. 4 Time amount corresponding to L3 cache miss count

shown that increase in work time (stretch factor) contributes the most in the slowdown. Particularly, it accounts for 78% on 20 cores, and increases gradually to 91% on 32 cores.

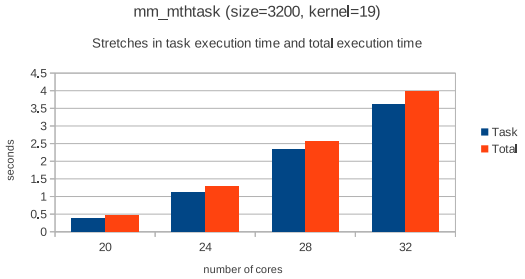

Fig. 2 Task Stretch's proportion

5. Cache Miss Count Observation

Fig. 3 shows L3 cache miss count (blue columns with left y-axis) together with the execution time of the matrix multiply application (red columns with right y-axis). We can realize that along with the stretch in execution time, cache miss count also increases.

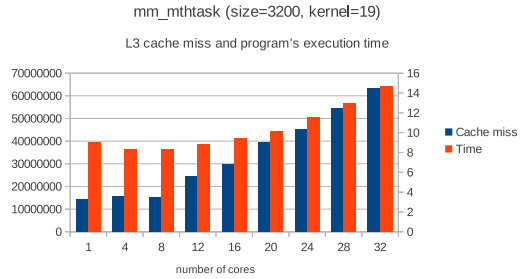

Fig. 3 L3 cache miss count

We have measured the latency of a single L3 cache miss on the machine, which was about 300 nanoseconds. Using this number we convert cache miss count into equivalent time amout. Then we redraw only the increased amount of time derived from cache miss count (blue columns) together with the increased amount of application's execution time (red column), we got **Fig. 4**.

In Fig. 4, the time caused by increase in cache miss count is much more than the actual increase in application's execution time. This can be understood by considering that processors can overlap several cache misses so each cache miss does not necessarily incur the full latency of 300 nanoseconds.

6. Conclusion and Future Work

Our work has demonstrated that work time stretch is the main factor that makes applications' performance degrade on large numbers of cores, beside the factors of idleness and runtime overhead. Moreover, it's also shown that work time stretch is accompanied with the increase in cache miss count. Indeed parallel execution is likely to cause a surplus amount of data loads when multiple worker threads load the same data to their different associated physical memory.

We have decomposed the work time stretch to task levels, but this is not useful enough. Programmers need to know specific code blocks that stretch so that they can know where they need to improve their programs. Therefore, in future work we would like to develop the profiler so that it can provide stretch information and other additional information like cache miss count regarding specific code blocks.

References

1) Robert D. Blumofe, Charles E. Leiserson. *Scheduling multithreaded computations by work stealing.* 35th Annual Symposium on Foundations of Computer Science, 1994 Proceedings.
2) Sameer S. Shende, Allen D. Malony. *The Tau Parallel Performance System.* International Journal of High Performance Computing Applications - IJHPCA , vol. 20, no. 2, pp. 287-311, 2006.
3) Intel VTune Amplifier *http://software.intel.com/en-us/intel-vtune-amplifier-xe.*
4) 中島潤，田浦健次朗．高効率な *I/O* と軽量性を両立させるマルチスレッド処理系．情報処理学会論文誌 プログラミング (PRO)．Vol.4，No.1，pp.13-26．2011 年 3 月．
5) S. Browne, J. Dongarra, N. Garner, K. London, P. Mucci. *A scalable cross-platform infrastructure for application performance tuning using hardware counters.* ACM/IEEE Conference on Supercomputing, 2000.
6) Omer Zaki, Ewing Lusk, William Gropp, Deborah Swider. *Toward Scalable Performance Visualization with Jumpshot.* International Journal of HPCA 1999.