Under submission to SOSP 2017 - do not cite or distribute

The case for [dynamically scalable parallelism]

Paper **#XX**

NN pages

Abstract

In recent years, economic and physical realities have pushed both chip designers and application developers to rely increasingly on parallelism to satisfy performance demands [8]. However, there is no free lunch. Even after an application developer has wrestled with several target architectures, she still does not know what resources the application will get at runtime. Often, it is easier to over-provision units of execution and hope for the best. Yet even mild contention can drastically reduce overall system throughput and wall-clock execution times of individual applications. On the other hand, under-provisioning units of execution means leaving resources unused and falling short of performance potential. In this paper, we propose a mechanism to coordinate CPU usage across applications and show that by selectively and dynamically changing application parallelism, we can avoid both pitfalls.

1. Questions

- How do developers typically provision number of threads? How do they determine the maximum?

2. To do

- DS Paper outline
- MS AC Understand difference in performance for different levels of parallelism
- ✓ Understand Intel TBB
- Design interface between library and coordinator
- Implement and integrate coordinator with library
- Integrate library into applications
    - http://wiki.cs.princeton.edu/index.php/PARSEC
- Evaluations
- Run ./bin/todo to find remaining todos in tex files

TODO: INSERT SMALL EXAMPLE

3. Introduction

Parallelism and cotenancy are the future. However, people program as though there is no cotenancy. This causes performance problems. Today, people statically partition. Utilization is not great and you don't get perfect isolation anyway.

Parallelism and cotenancy are here to stay

- Increasing number of cores [2]
- As a result, application developers must think about parallelism to get performance
- Cloud becoming increasingly popular; run many containers / VMs on a single host and want performance guarantees for tenants (sophisticated orchestration / scheduling)

Developers / OS have digested parallelism, but not cotenancy

- Developers: assume have full machine (over-provision) because not easy to know what resources we have.
- OS: schedule individual threads (Linux). Some control with cgroups to coordinate groups of threads. (is this fair? probably should write a different way. Don't want to offend OS people by implying OS can't handle multiuser...)

This can lead to poor utilization or excessive resource contention

- Over-provisioning may low utilization on many core. CPU share and quota (cgroup) not enough; need to change parallelism of applications to affect utilization
- Without coordinating resource usage, system may thrash since different applications have different overheads / slow-down in total CPU time

Contributions

- Observe that dynamically scaling parallelism can improve system performance (throughput, latency)
- Thread library that can dynamically scale application parallelism
- User-level coordinator to determine appropriate level of parallelism for registered applications

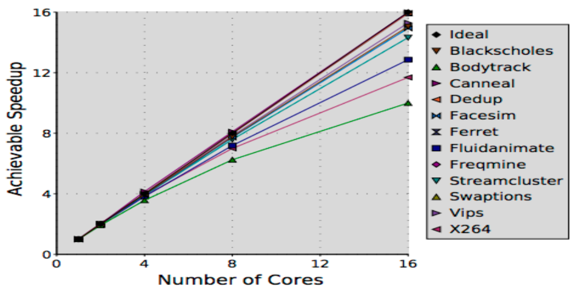

Figure 1. PLACEHOLDER: Wall-clock speed-up time. Upper bound for speedup of PARSEC workloads with input set simlarge based on instruction count. Limitations are caused by serial sections, growing parallelization overhead and redundant computations.

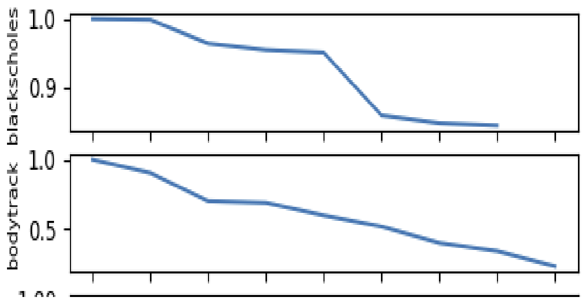

Figure 2. PLACEHOLDER: Although applications typically enjoy wall-clock speed-up times when they use more threads, often see corresponding increase in total CPU time [this graph might be confusing since we're talking about increase in total CPU time, but curve is going down]

4. Performance of parallel programs

4.1 More in-depth example showing utilization problems

Note: will cover PARSEC/Splash2 in more detail in later section [4] [3] [5]

4.2 Performance without contention

- speed-up curve (see Figure 1)
- slow-down curve (see Figure 2)

4.3 Performance under contention

- same application (see Figure 3)
- mixed group of applications (see Figure [])

4.4 Reasons for contention

- thread contention (manny runnable threads, but divided work up into small pieces; not overlapping well with i/o)
- lock contention (why would this be exacerbated under contention?)
- other resource contention

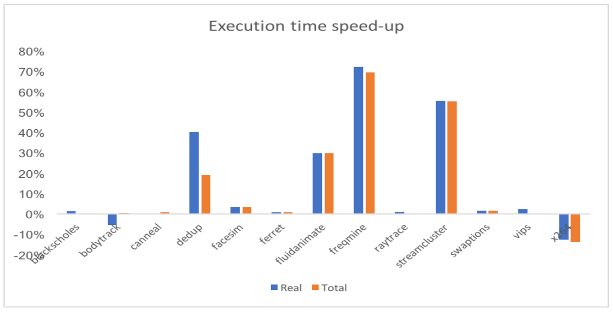

Figure 3. PLACEHOLDER: How groups of applications (same or mixed (need this fig)) do under contention at varying # of threads

Classes of application behavior

- Big improvement from reducing parallelism (dedup, facesim, freqmine, streamcluster)
- Slight improvement from reducing parallelism
- Performance hit from reducing parallelism (x264)

4.5 Common strategies for dealing with unknown resources

TODO: maybe delete this section?

- Overprovision number of threads and hope for the best. But this can cause problems (e.g., lock or resource contention)

5. Determining contention and who should change parallelism

TODO: change section title

TODO: maybe this shouldn't be its own section

5.1 System metrics for determining contention

- # of runnable threads
- CPU utilization

5.2 Application metrics for determining candidates for parallelism scaling

6. Design

Not sufficient to set parallelism just at application start, need to scale up and down. Machine load may change over time. Application requirements may change within a single run

6.1 Mechanism for communicating

1. Use kernel memory (must modify kernel)
   - OS: schedulers already quite complex. Must serve many masters; hard to make changes without harming certain workloads. Don't want to change the kernel if we can improve performance in user space [7]
2. User space

overhead.png

Figure 4. What is performance overhead of thread library over pthreads?

6.2 Thread library

Similar to Intel Thread Building Blocks [10], but dynamically scale number of workers across applications, rather than just across workers

- Types of blocks: iterative / non-iterative parallelized operations
- DAG
- Unstructured

Global queue vs. fixed partition + work stealing
Min/max number of threads set by application developer
I/O and CPU pools treated differently in most libs

6.3 Driver application

Track CPU utilization by application (any other metrics?) Change target parallelism (how to decide, when to decide, how much to change by, which to change) Change CPU share or quota

6.4 Client applications

7. Benchmark applications

- TODO: do we want a single large application (e.g., provisioned for KNL, but running on much smaller VM for some reason)?
- TODO: What about an unstructured application (e.g., web server)?
- TODO: What about long-lived application (e.g., streaming)? We can probably simluate this via streamcluster

7.1 PARSEC

7.2 SPLASH-2

TODO: maybe leave this out given timing

8. Evaluation

8.1 Workloads

Different classes of applications. Mixture of applications.

1. What workloads are helped a lot
2. What are not really affected
3. What are harmed

Should we have single large application?
How should we handle unstructured parallel programs (e.g., web server?)

8.2 Comparison with other thread libraries

OS (pthreads), OpenMP, Intel TBB

8.3 Varying guarantees (CPU share / quota)

8.4 Various machines

8.5 How much work for using?

9. Related work

9.1 Parallel programming libraries

9.2 Application / OS interface

- Intel TBB [6]
  - https://software.intel.com/en-us/blogs/2007/08/13/threading-building-blocks-scheduling-and-task-stealing-introduction
  - https://software.intel.com/en-us/node/506294
  - May not have full view (e.g., in virtualized environment)
- Microsoft PPL
  - https://msdn.microsoft.com/en-us/library/gg663539.aspx
- Scheduler activations [1]
- Cilk [**?**]
- Charm++
- OpenMP
- Chare
- Multikernel
- Exokernel
- Hierarchical schedulers
- Cache-aware / contention-aware / numa-aware schedulers
- Arachne
- Capriccio
- Wasted cores

10. Discussion and future work

Whatever we didn't finish
Making a case
Don't claim optimal

Acknowledgements

References

[1] ANDERSON, T. E., BERSHAD, B. N., LAZOWSKA, E. D., AND LEVY, H. M. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM*

Transactions on Computer Systems (TOCS) 10, 1 (1992), 53–79.

[2] BAUMANN, A., BARHAM, P., DAGAND, P.-E., HARRIS, T., ISAACS, R., PETER, S., ROSCOE, T., SCHÜPBACH, A., AND SINGHANIA, A. The multikernel: a new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles* (2009), ACM, pp. 29–44.

[3] BIENIA, C., KUMAR, S., SINGH, J. P., AND LI, K. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques* (2008), ACM, pp. 72–81.

[4] BIENIA, C., AND LI, K. *Benchmarking modern multiprocessors*. Princeton University, 2011.

[5] BIENIA, C., AND LI, K. Characteristics of workloads using the pipeline programming model. In *Computer Architecture* (2012), Springer, pp. 161–171.

[6] CONTRERAS, G., AND MARTONOSI, M. Characterizing and improving the performance of intel threading building blocks. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on* (2008), IEEE, pp. 57–66.

[7] LOZI, J.-P., LEPERS, B., FUNSTON, J., GAUD, F., QUÉMA, V., AND FEDOROVA, A. The linux scheduler: a decade of wasted cores. In *Proceedings of the Eleventh European Conference on Computer Systems* (2016), ACM, p. 1.

[8] MACK, C. A. Fifty years of moore's law. *IEEE Transactions on semiconductor manufacturing 24*, 2 (2011), 202–207.

[9] OUSTERHOUT, K., WENDELL, P., ZAHARIA, M., AND STOICA, I. Sparrow: distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (2013), ACM, pp. 69–84.

[10] REINDERS, J. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism.* " O'Reilly Media, Inc.", 2007.