PROGRESS IMBALANCE IN MULTI-PROCESS PERFORMANCE

Arghya Bhattacharya, Ph.D. Candidate, Algorithms Lab, CS@SBU

Stony Brook University
Computer Science

ABSTRACT

Most modern systems have multi-core, multi-threaded, and time-shared architecture and processes run on a shared cache. Understanding the behavior of a cache that several concurrent processes share is crucial for application designers. Multiple ***homogeneous threads***, threads running copies of the same program, may suffer from an imbalance of cache-residency [1].

We observe an interesting phenomenon: if we run multiple copies of the same program (***homogeneous instances***), each has private-data, and share a given shared memory, we observe a ***progress imbalance*** of the copies of the program; the program instances finish at different times.

We run up to six concurrent instances of two cache-oblivious divide-and-conquer and one cache-aware cubic matrix multiplication algorithms. We compare the differences in the running time of the program instances. We observe that the relative standard deviation of the running time increases with the number of concurrent instances. The more concurrent programs we run, the more progress imbalance among them we get to observe. One potential reason is an imbalance in the cache-sharing among the instances; indeed, the program instances with a transient stall suffer from a lesser share of cache throughout the run, leading to much higher running time.

KEY FINDINGS

> Multiple concurrent processes fail to share the memory gracefully.

> Some of the concurrent processes dominate others.

> The ones who dominate others are likely to have more share of the cache, leading to enjoy more progress, and ultimately causing a progress imbalance.

EXPERIMENTAL SETUP

Algorithms: We observe the running time of two cache-oblivious cubic matrix multiplication algorithms, one of which does in-place additions (***MM-INPLACE***), another does additions outside the recursion using extra space and linear scans (***MM-SCAN***) [2]. We also observe the running time for the cache-aware block matrix multiplication algorithm (***MM-BLOCK***) [4]. Each program multiplies two square matrices of width *2048*; the input matrices and the output matrix take a total of *48* MiB.

System: The input and output data of each program is stored in the hard-disk and mapped to the memory using a ***file-backed mmap***. We restrict the memory of a program by running it inside a Linux ***cgroup***.

Experiment: We run up to 6 concurrent instances of the matrix multiplication programs [3]; in each run, a fixed amount of memory is given ($k \times 10$ MiB memory for k concurrent instances), and the concurrent instances share this memory.

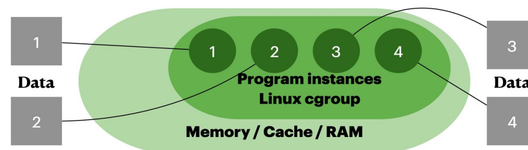
To know why: We run 6 concurrent instances, we employ a uniform transient stall to all the program instances; either all of them stall at the end (***homogeneous***) or a half stall in the middle and the other half stall at the end (***heterogeneous***).

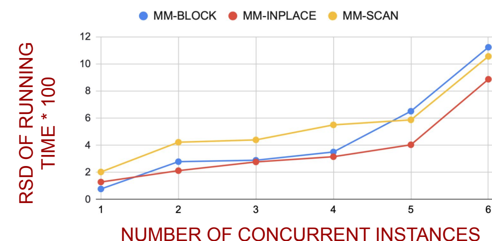
Quick review: For example, we run 3 concurrent instances of a program. The running-times of the instances are X1, X2, and X3.

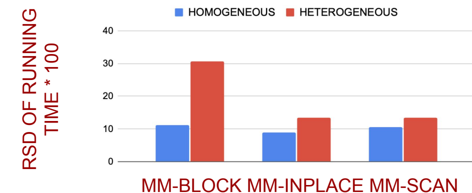
Mean running time = X = (X1 + X2 + X3) / 3

Standard Deviation, SD = $\sqrt{Var} = \sqrt{[(X1 - X)^2 + (X2 - X)^2 + (X3 - X)^2] / 3}$

Relative Standard Deviation, RSD = SD / X







CONCLUSION

Application designers and system researchers need to keep this phenomenon in mind while designing systems. In the future, we hope to explore some algorithmic solutions to deal with this phenomenon.

[1] Dice et al. Brief Announcement: Persistent Unfairness Arising from Cache Residency Imbalance. SPAA '14.
[2] Bender et al. Cache-adaptive Analysis. SPAA '16.
[3] Chowdhury et al. Autogen: Automatic Discovery of Efficient Recursive Divide-&-Conquer Algorithms for Solving Dynamic Programming Problems. ACM TPC '17.
[4] Hong and Kung. I/O Complexity: The Red-blue Pebble Game. STOC '81.