Analysis of Data Reuse in Task-Parallel Runtimes

Miquel Pericàs[1], Abdelhalim Amer[2], Kenjiro Taura[3], and Satoshi Matsuoka[1,2]

[1] Global Scientific Information and Computing Center
Tokyo Institute of Technology
pericas.m.aa@m.titech.ac.jp, matsu@is.titech.ac.jp
[2] Department of Mathematical and Computing Sciences
Tokyo Institute of Technology
amer@matsulab.is.titech.ac.jp
[3] Graduate School of Information Science and Technology
The University of Tokyo
tau@eidos.ic.i.u-tokyo.ac.jp

Abstract. This paper proposes a methodology to study the data reuse quality of task-parallel runtimes. We introduce an coarse-grain version of the reuse distance method called *Kernel* Reuse Distance (KRD). The metric is a low-overhead alternative designed to analyze data reuse at the socket level while minimizing perturbation to the parallel schedule. Using the KRD metric we show that reuse depends considerably on the system configuration (sockets, cores) and on the runtime scheduler. Furthermore, we correlate KRD with hardware metrics such as cache misses and work time inflation. Overall we found that KRD can be used effectively to assess data reuse in parallel applications. The study also revealed that several current runtimes suffer from severe bottlenecks at scale which often dominate performance.

1 Introduction and Background

Tasking has become an established technique to program multicore systems. This programming scheme supports many variations of parallel control, including nested, recursive and irregular parallelism. Task-parallel models, such as OpenMP [1], Threading Building Blocks [2] or Cilk [3], allow the developer to annotate functions or code blocks for asynchronous task execution and add synchronization points to process the children tasks' outputs. An underlying runtime tracks dependencies among tasks and *schedules* ready tasks to physical cores.

1.1 Scalability of Runtimes

Although the functionality of a runtime for homogeneous multicore systems may seem simple, developing efficient and scalable implementations is challenging. Design decisions can adversely affect execution time:

Runtime Overheads Operations such as task creation, synchronization or scheduling introduce *non-work* cycles that can considerably increase execution time. Runtime pressure grows with the number of workers and with finer task granularities. Contention can easily occur at scale. Runtimes should be as lightweight as possible to avoid such bottlenecks.

Scheduling Constraints Runtimes may place restrictions on task scheduling to simplify implementation or to set bounds on resource consumption. For example, some runtimes never migrate tasks once they have started. Some runtimes also limit the depth of nesting to avoid unlimited stack growth. Such constraints limit dynamic parallelism which manifests as non-work overheads in the form of processor idle time.

Resource Sharing Scheduling policies, such as *work-first* [4] or its dual *help-first*, and *work stealing* [5] techniques, determine the execution order of tasks. The resulting schedule defines the order of work *kernels* and their sharing of resources. A task order that ignores data locality issues can increase cache misses and generate work time inflation (WTI) [6].

In this work we use the term *non-work overheads* for any kind of processor activity that is not directly related to the program's main functionality, which is carried out by *work kernels* and the control necessary to setup their execution. The non-work overheads include runtime execution and *parallel idleness* [7]. Tasks may include several kernels, but the kernels themselves do not generate any new tasks. OVR_N and WTI_N (*Non-work Overheads* and *Work Time Inflation* at N cores) are two measurable scaling factors that describe the increase of execution time on N cores (T_N) relative to the *ideal* parallel execution time ($\frac{T_1}{N}$). OVR_N quantifies the increase in the total running time of all threads ($T_N \times N$) relative to the total time during which threads are performing work (Work_N). WTI_N quantifies the increase of the total work time at N cores (Work_N) relative to the work time of the serial execution (Work_1):

$$T_N = \frac{T_1}{N} \times \mathrm{OVR}_N \times \mathrm{WTI}_N \tag{1}$$

$$\mathrm{OVR}_N = \frac{T_N \times N}{\mathrm{Work}_N} \tag{2}$$

$$\mathrm{WTI}_N = \frac{\mathrm{Work}_N}{\mathrm{Work}_1} \tag{3}$$

Using this formulation, the speed-up on N cores becomes:

$$\text{Speed-Up}_N = \frac{T_1}{T_N} = \frac{N}{\mathrm{OVR}_N \times \mathrm{WTI}_N} \tag{4}$$

1.2 Performance Tools

Application developers are often unaware of such issues and are then surprised by the bad performance of their applications as they scale to many cores. Quality tools are needed to detect these problems. Profilers and tracers provide insight into *non-work overheads* [7–10] by quantifying load imbalance and runtime activity overhead. *Scheduling constraints* are more difficult to analyze, since they relate to algorithmic decisions inside the runtimes. Similarly, *caching problems* caused by scheduling decisions may be hard to identify. Low data locality exploitation in users' code, on the other hand, is a well known topic addressed by several tools [11–13].

This paper focuses on the problem of understanding caching problems introduced by the runtime scheduler in task-parallel applications. To analyze the impact of schedulers on data reuse we propose a methodology based on the concept of the reuse distance [14]. By analyzing the reuse distance observed at each last level cache, the metric allows to make a system-level assessment on the reuse performance of different runtimes.

1.3 Contributions

This paper makes the following contributions: 1) We describe the implementation of the Kernel Reuse Distance (KRD), a metric based on to the reuse distance targeting the analysis of temporal locality in task-parallel applications. 2) Using KRD we evaluate the temporal locality of two benchmarks using four schedulers. Our analysis reveals that differences in reuse increase with the number of cores and sockets. 3) We study the correlation between the KRD metric and hardware metrics such as cache misses and work time inflation. As part of this research we also observed that, at scale, performance and work time inflation are often dominated by runtime bottlenecks.

This paper is organized as follows: Section 2 sets the scenario by analyzing the scalability of two benchmark applications. Section 3 describes the KRD metric and its implementation. The metric is applied in Section 4 to observe how temporal locality is influenced by runtime schedulers and to study its correlation with performance metrics. We conclude in Sections 5 and 6 by discussing weaknesses of the approach and by summarizing the main conclusions.

2 Case Study: Matrix Multiplication and the Fast Multipole Method

The development of KRD is motivated with a scalability study of two codes: Matrix Multiplication (*MATMUL*) and the Fast Multipole Method (*FMM*).

2.1 Benchmarks

The *MATMUL* code is a SIMD-optimized divide-and-conquer implementation which includes a task parallel implementation based on Cilk-like `spawn` and `sync`

constructs [4]. The code recursively bisects the matrices until all three submatrices A, B and C fit in the L1 cache. For the experiment we use input matrices of size 4096×4096, which translates into 64MB per matrix (single precision). On our test environment (described below) the granularity of each task (kernel) is about 17 microseconds.

The *Fast Multipole Method* is based on the exa*FMM*-dev code developed by Rio Yokota [15]. The *FMM* algorithm contains multiple steps. We focus only on the dominant phase: the dual tree traversal, which includes the two main kernels: M2L (multipole-to-local) and P2P (particle-to-particle). We run one *FMM* timestep on 1 million particles organized as a plummer distribution. The multipole expansion coefficient is set to 5 and the number of particles per leaf box is 32. The tree traversal phase is also parallelized by a divide and conquer approach [16], and uses the same Cilk-like constructs as *MATMUL*. The *FMM* kernels are quite small, with each call to M2L only 500 nanoseconds. To avoid excessive overhead the recursion stops when less than 300 bodies remain under the current subtree, yielding multiple kernels per task. On our test system, the average size of one task is 3.25 microseconds.

2.2 Experimental Infrastructure

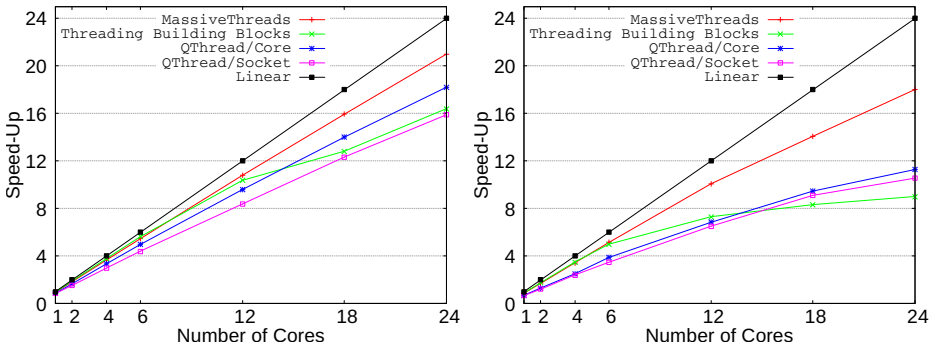
We benchmark the codes on a 4-socket x86-64 server featuring 4× Intel Xeon E7-4807 (Westmere-EX) processors, each with 6 cores clocked at 1.86GHz. The cores have a 32KB L1 data cache (8-way set associative) and a 256KB L2 cache (8-way). The six cores share a 18MB last level cache (L3) with 16 ways. Hyper-threading is not used. When scaling to multiple cores, we first allocate all the cores in one socket and then fill the cores from a different socket. All codes were compiled using gcc version 4.7. The research platform runs a Linux distribution with kernel version 2.6.32.

The Cilk-like constructs are translated into API calls for three runtimes, identified as follows:

`MTH` : MassiveThreads [17, 18] is a lightweight task-parallel library that features a work-first scheduler, per-core LIFO task queues, and a random work stealer similar to the MIT-Cilk design.

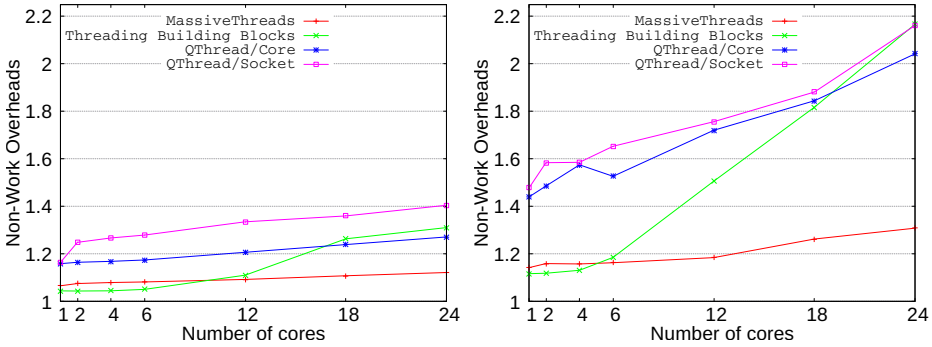
`TBB` : Threading Building Blocks [2, 19] is a C++ template library for task parallelism with a help-first approach, per-core LIFO task queues and random work stealer. Although TBB supports thread affinities [20], we do not use this feature in order to compare the same code. We use version `tbb41_20130116oss`.

`QTH` : Qthread [21–23] is a lightweight threading package that implements a help-first scheduler. Qthread adds a new level to the task queues' hierarchy called *shepherds*. Shepherds can be assigned per socket to create a shared LIFO task queue among the workers (i.e. cores) of the socket. The goal is to improve the use of the shared cache. We refer to this configuration as `QThread/Socket`. We also test a configuration with one shepherd per core, we which identify as



(a) *MATMUL* (b) *FMM*

Fig. 1. Speed-ups



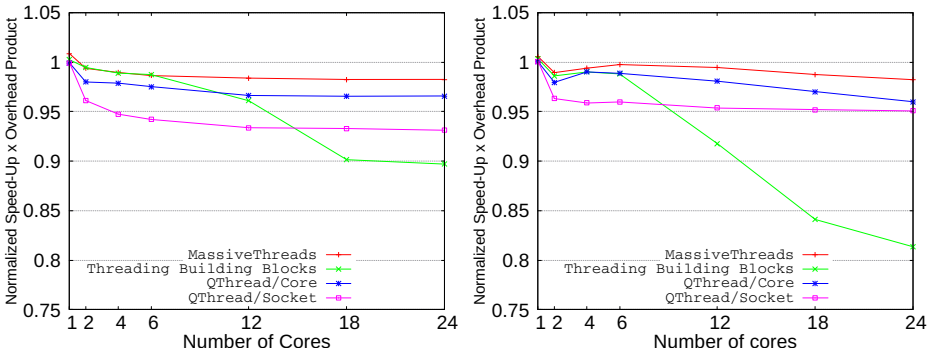
(a) *MATMUL* (b) *FMM*

Fig. 2. Non-Work Overheads

`QThread/Core`. Qthread also has a bulk work-stealer. By default it attempts to steal 50% of the victim's workload. The qthread version we use is 1.9.

2.3 Scalability Analysis

The applications were manually instrumented with our own profiling library, which we describe later. This library measures execution times, work time inflation and non-work overheads. Figures 1, 2 and 3 show the speed-ups and non-work overheads (OVR_N) for the two applications and four schedulers when scaling from 1 to 24 cores. We also show the product of the speed-up and overhead normalized by the number of cores. Using the earlier equation we derive $(\text{Speed-Up}_N \times \mathrm{OVR}_N/N) = 1/\mathrm{WTI}_N$. The product is thus a measure of the speed-down caused by work time inflation.



(a) *MATMUL* (b) *FMM*

Fig. 3. Speed-up Overhead Product

The figures show that scalability of these applications is highly dependent on the runtime. Using MassiveThreads, speed-ups of up to 21× and 18× are achieved for *MATMUL* and *FMM* at 24 cores, respectively. TBB displays very good scaling until the first socket is filled, but performance degrades at higher core counts. Qthreads performance is already degraded in the single socket scenario. However, it scales better than TBB for multiple sockets.

These results are highly correlated with the non-work overheads. MassiveThreads is the only runtime that does not suffer from a large increase, with 30% overhead in the worst case. The other runtimes suffer about 2-4× higher overheads at high core counts. Both Matmul and *FMM* have small task sizes, which MassiveThreads is designed to handle efficiently. Qthread's single core overheads demonstrate that it is more heavy and suffers under fine-grained parallelism. However, scaling to higher cores reveals just a smooth degradation. TBB's overheads are lower than MassiveThreads for a single socket but increase fast for multiple sockets. The `QTH/Socket` overheads are consistently larger than those of `QTH/Core`. `QTH/Socket` features a per-socket shared LIFO task queue which is accessed by all workers in a shepherd. The frequency of accesses to the queue is proportional to the number of workers sharing it and inversely proportional to the average task size. For small task sizes and large number of workers this method is likely to suffer from contention.

The third plot shows the *Speed-Up* × *Non-work Overhead* product normalized by the number of cores. In the ideal case this metric should yield 1.0. A value below 1.0 indicates work time inflation. The plots show that work time inflation is an important issue, contributing a further performance reduction of up to 20% in the worst case (*FMM* with TBB). Since kernels never block, WTI can only be attributed to destructive resource sharing. This effect is mainly observed as an increased number of cache misses and/or increased memory access latencies. Two factors can cause this: 1) When the memory subsystem or system interconnect is overloaded, average memory access latency increases. In addition, runtime

bottlenecks -such as excessive contention on a global lock- can steal bus cycles from the memory subsystem which further contribute to increase latencies. 2) A change in the work time can also be caused by data locality variations. Different kernel schedules, for example, impact temporal locality and cache misses.

Measuring how much of work time inflation is caused by the runtime and how much is due to locality is difficult because of the small kernel sizes and because of the high overheads of accessing hardware performance monitors using the Linux `perf` subsystem [24]. To identify work time inflation due to temporal locality issues we look for a scenario with minimal non-work overheads. For the case of *MATMUL*, MassiveThreads and TBB have overheads around or below 1.1× until 12 cores (2 sockets). Figure 3 (a) shows a work time difference of about 2% between `MTH` and `TBB` at 12 cores that must be related to different task orders. At 24 cores this difference is around 8%. The KRD metric defined in the next section can provide additional insight regarding the origin of additional cache misses.

3 Kernel Reuse Distance

To characterize the effects of task ordering on temporal locality we start with the reuse distance metric [14]. The reuse distance has traditionally been used as a measure of cache performance [25]. It processes traces of memory accesses and counts the number of unique addresses between two accesses to the same element. This count is also called the *stack distance*.

When analyzing task-parallel applications it is important to minimize perturbation to the runtime task schedule. Heavyweight instrumentation to generate address traces may impact the execution and result in a parallel schedule that is not representative. To reduce overheads we extend the method to collect data accesses only in bulk at kernel execution times. For each data structure that is an input or output to a kernel, an identifier (usually its base address), a timestamp, and its size in bytes are recorded. We rely on manual instrumentation to perform these actions.

A trace of data accesses is recorded separately for each core. To analyze the reuse on a per-node[4] basis we process a *merged trace* containing all the kernel inputs and outputs accessed by the cores sharing the same last level cache. The trace is synchronized using the timestamps. Using this *merged trace*, the stack distances are computed and the histogram is generated. When a system contains multiple nodes, we summarize the contribution of each by generating per-node histograms and then reporting their summation.

Altogether, this set of modifications on top of the reuse distance is called the Kernel Reuse Distance metric (KRD). KRD is is a low-overhead and architecture independent method that provides an intuitive measure of data reuse. Its correlation to hardware metrics such as cache misses and performance is analyzed later. Figure 4 shows a diagram explaining the methodology in a single socket

[4] in this work we use *node* as shorthand for *NUMA node*

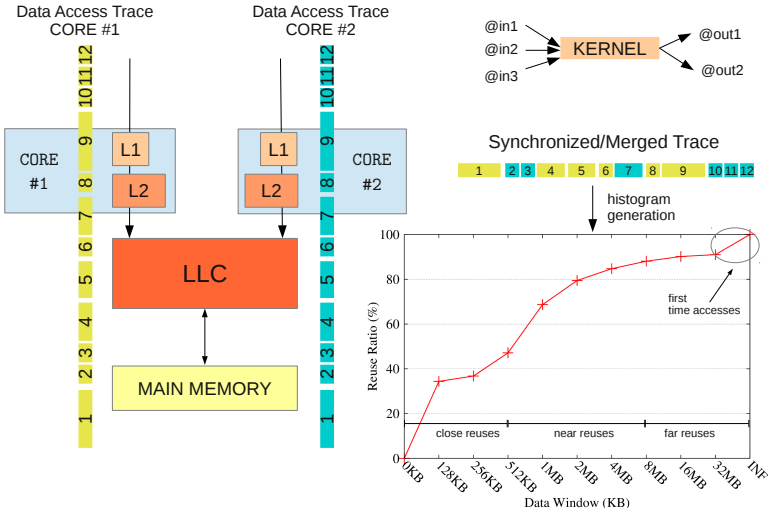


Fig. 4. Generation of the KRD metric for a single socket with two cores

environment with two cores. Two workers are running, one on each core, and generating a series of kernel data accesses. To analyze the last level cache and memory access, the traces are merged and the reuse histogram is generated. The histogram shows the ratio of data reuses that occur within a certain data window, shown on the x-axis. All elements have a first access. This event is included in the last data point labeled as `INF` (infinity). In the multiple nodes scenario, work steal activity across nodes introduces additional *cold* accesses. By looking at the number of accesses that contribute to the INF category, one can observe the effects of inter-node work steals.

For visualization purposes, we subdivide the histogram into *close*, *near*, and *far* reuses. This choice is arbitrary but will help later in describing the plots. As a rule of thumb, we use close reuses for those that fall within L2 cache size, near reuses for those within last level cache (LLC) size, and far reuses for those beyond the size of the LLC.

3.1 Implementation Details

We implemented KRD as a set of tools that can compute the histograms from traces generated by our own low overhead profiling and tracing facility called `LoI` (low-overhead instrumentation). LoI is designed to analyze task-parallel applications with fine grained kernels. LoI attempts to be as lightweight as possible in order to not influence the task parallel schedule. The library associates timestamps to events, and either aggregates execution times for individual kernels or generates timestamped traces. Timestamps are obtained by using the x86 TSC

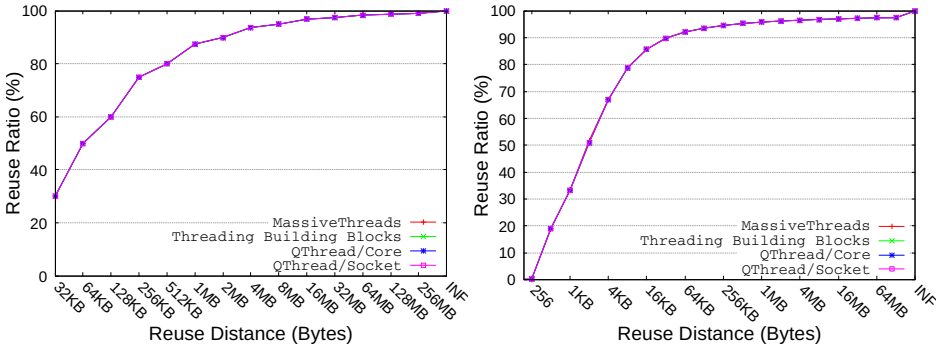
facility [26]. For both applications the tracing facility increases execution time less than 5% in the worst case.

4 Experimental Evaluation

This section describes two experiments. We begin by generating KRD profiles for *MATMUL* and *FMM* to display how reuse changes with the runtime scheduler. Next we analyze the correlation between the KRD metric and hardware performance counters.

4.1 KRD correlation with runtime schedulers

Figures 5–7 show the KRD plots for the two benchmarks using the four tested runtime schedulers on three hardware configurations: single core, one fully-populated socket and four sockets.

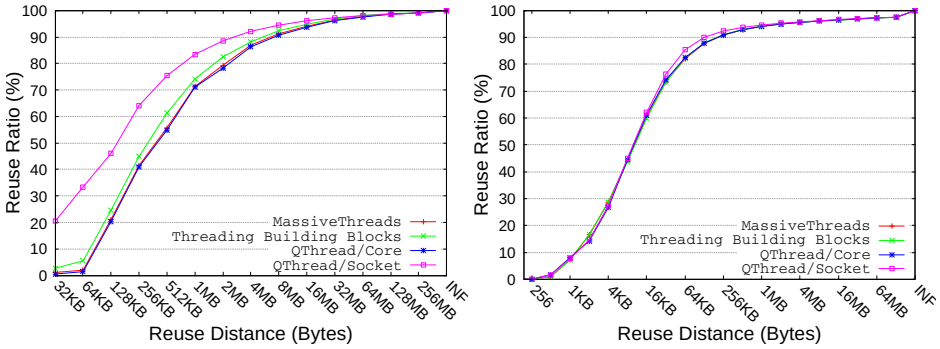


(a) *MATMUL* (b) *FMM*

Fig. 5. Kernel Reuse Distance plots for a single core

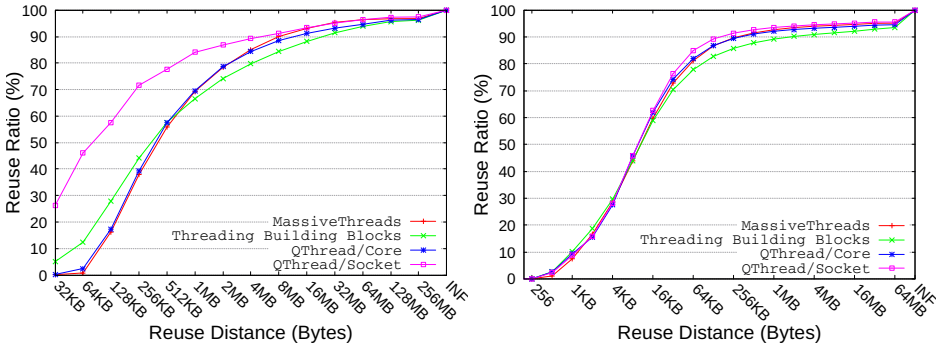
The *single core* histograms show that in the absence of work steals, different schedulers have little impact on the temporal reuse of recursive divide and conquer task-parallel codes. In fact, for *MATMUL*, the KRDs of both work-first and help-first policies are identical. This is not surprising as the recursive bisecting of the matrices and corresponding task generation are symmetric. Work-first and help-first execute the leaf kernels in reverse order, but this has no effect on the reuse distance. For *FMM* the decomposition is not completely symmetric because of a property of the algorithm which allows to discard one of the branches based on a condition (mutual interactions). However, differences between schedulers are still barely noticeable.

Differences start to emerge when one socket is fully populated (6 threads), particularly on *MATMUL*. `QThread/Socket` stands out, having the highest reuse



(a) *MATMUL* (b) *FMM*

Fig. 6. Kernel Reuse Distance plots for one socket (6 cores)



(a) *MATMUL* (b) *FMM*

Fig. 7. Kernel Reuse Distance plots for four sockets (24 cores)

ratio at almost all distances. This good performance results from `QThread/Socket`'s usage of a global LIFO queue shared by all the workers. In this design, workers tend to execute tasks that have been recently generated by other workers. Since programs are commonly optimized for data reuse on the serial path, this policy improves cache sharing [23]. `TBB` also shows improved reuse compared to `MTH` and `QTH/Core` when executing *MATMUL*. This is probably a side-effect of the `TBB` scheduling restrictions [19]. `MTH` and `QTH/Core`, on the other hand, implement just a fully distributed random work stealer. It has the worst reuse performance, but offers the advantage of simplicity. The differences between schedulers are considerably smaller in the case of *FMM*. `QTH/Socket` is still better for close reuses, but the difference is only 3% at most. Other schedulers show almost no differences. The similarity between histograms is likely a result of *FMM*'s tree traversal algorithm, which conditionally executes two kernels that operate

on independent data. Furthermore, the non-homogeneous input (*plummer* distribution) generates an irregular kernel pattern that is harder for schedulers to optimize.

The histograms for the 4-socket scenario are similar to the 1-socket case. `QTH/Socket` again shows the best reuse performance, but this time it is closely followed by `MTH` for far reuses. Surprisingly, in this multi-socket scenario, `TBB` has the worst reuse performance for far reuses, trailing the other schedulers at a noticeable distance. Compared to the single-socket plots, one important fact revealed by the four-socket KRD histograms is the larger amount of cold accesses. This is expected, as separate sockets have disjoint caches which need to be warmed up separately. KRD can be used to understand how many first time accesses occurred, which indirectly correlates to the size of the working set observed at each socket. `QThread/Socket` shows the lowest ratio of cold accesses while `TBB` shows the highest amount. A larger number of cold accesses means that the scheduler is distributing tasks that share the same working set across different nodes. TBB implements several restrictions in its scheduling algorithm that limit which tasks can be stolen and disallows the migration of tasks that have already started [19]. These limitations might be forcing TBB into a suboptimal work partitioning.

The fact that the KRD histograms can be correlated with different schedulers is an encouraging result. Next we address the question whether these plots can be correlated with actual performance.

4.2 KRD correlation with Hardware Metrics

In the second experiment, we attempt to correlate the results of the KRD metric with last level cache misses and work time inflation. To do so we select a scenario with low runtime overheads to minimize possible perturbation. For *MATMUL* using 2 sockets (12 cores), `MTH` and `TBB` present non-work inflation of about 1.1×, while the `QTH/Core` overhead is about 1.2×. The KRD plot of far reuses (i.e. beyond 18MB) for this configuration is reported in Figure 8. Table 1 reports hardware performance counters and time measurements collected as averages of five runs. The kernel times are average over all kernel executions ($\sim 1 \times 10^6$) and have been collected by reading the x86 timestamp counter at each kernel call (RDTSC). The LLC misses column reports the per-core `LAST_LEVEL_CACHE_MISSES` metric from Intel's Architectural PerfMon [26], as reported by PAPI [27].

We first compare `MTH` and `TBB`, which have similar overheads. The KRD plot in Figure 8 shows that for all distances beyond 16 MB, `MTH` has a higher percentage of reuses than `TBB`. The LLC size of the Westmere-EX chip is 18 MB, which makes it worth to analyze of the data point at 32 MB. For `MTH`, 3.57% of the kernel references access data with a reuse distance beyond 32 MB, while for `TBB` the number of far reuses is 4.5%. This 25% difference correlates to a 53% increase in LLC misses and to a work time inflation of 2.7% compared to MassiveThreads.

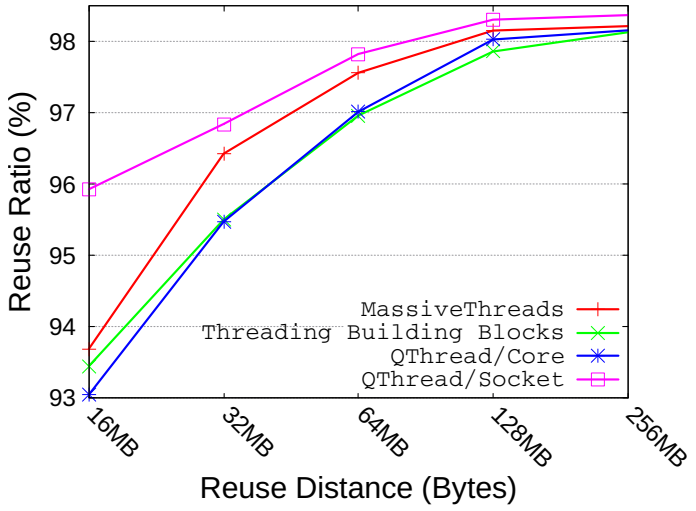


Fig. 8. Far reuses for *MATMUL* in the 2-socket, 12 core scenario

Table 1. Hardware Metrics and WTI for 2-socket scenario

Runtime	Exec. Time	LLC Misses	Kernel Time & Inflation
MTH	1.642 sec	1.829×10^6	17441ns (1.0250×)
TBB	1.742 sec	2.807×10^6	17898ns (1.0519×)
QTH/Core	1.859 sec	2.339×10^6	17767ns (1.0441×)
QTH/Socket	2.111 sec	1.987×10^6	18401ns (1.0814×)

For `QTH/Core`, the KRD plots show that it has higher number of far reuses than `MTH` but less than `TBB` for distances of more than 32MB. The number of LLC misses and the work time inflation are between those of `TBB` and `MTH`. This relation is also clearly observed at high distances (e.g. 128MB) and also for cold misses. It suggests that these data points might be good indicators for cache misses and WTI.

The KRD plot also shows that `QTH/Socket` has overall the smallest amount of far reuses (3.15% at 32MB). However, its number of cache misses is higher than MassiveThreads, and its work time inflation is the highest of all four schedulers. `QTH/Socket` has comparatively high overheads (1.33×). A closer analysis using `perf record` revealed that the *MATMUL* benchmark spends about 25% in two Qthread functions (`qt_scheduler_get_thread` and `qt_hash_lock`), both of which include memory bus locking activity. Bus locking increases memory access latencies, and is a probable explanation for the observed work time inflation.

The 2.7% difference between `MTH` and `TBB` may seem very small, but is also expected since the studied algorithm (*MATMUL*) is not particularly memory intensive. At 4 sockets TBB has about 10% higher work time inflation compared to MTH. In the case of *FMM*, the relative inflation reaches 45% for the memory bound M2L kernel on 4 sockets. Depending on the algorithm work time inflation can become an important issue.

5 Discussion

Although KRD shows correlation with work inflation and cache misses, it should be used mainly as an intuitive model. The KRD metric contains many simplifications that are the result of the constraints set by our original goal: to qualitatively measure temporal reuse in task-parallel programs. The requirement of minimal overhead is an important consideration which enables only a coarse-grained, manually-instrumented tracking of data accesses. The model does not consider other accesses such as stack accesses, based on the assumption that kernel (heap) data accesses dominate cache performance.

KRD does also not attempt to measure spatial locality among individual accesses. Our original goal was to analyze the effects of different schedulers on data reuse. Different schedules might, however, benefit more or less from prefetchers. If such an effect is large, then extending KRD with a metric to quantify spatial locality [28] might be a worthy addition.

One limitation of the current model is that it does not provide enough information to model the effects of cache coherence protocols [29,30]. When one core writes a data structure allocated in the last level cache of a different socket, this will conceptually result in a cache-to-cache *transfer*. The KRD metric currently uses only the notion of intra-socket data accesses. It can report increases in cold misses due to work stealing operations, but it cannot model misses due to cache line invalidations. As part of our future work we plan to extend KRD by classifying accesses into reads and writes. This will allow a simple modeling of the effects of cache coherence.

Finally we would like to note that, while the KRD model has been developed with task-parallel runtimes in mind, it is actually quite generic as it does not instrument tasks, but the kernels inside tasks. This allow it to be applied to study any kind of shared memory parallel framework.

6 Conclusions

In this work we have attempted to provide some insight on the impact of task-parallel schedulers on temporal locality and its effect on performance. We developed a coarse-grained version of the reuse distance metric to study reuse in task parallel executions. Based on our analysis of two benchmarks and four runtime schedulers we observed that schedulers can have considerable impact on the reuse distance, and that the reuse quality depends considerably on the system configuration. Furthermore we observed correlation between the KRD metric

and hardware metrics such as last level cache misses and average kernel execution time. However, we also observed that runtime contention can be dominant in high core count scenarios, thus minimizing overheads should take precedence over locality optimizations.

Acknowledgment

This work has been supported by a JSPS postdoctoral fellowship (P-12044). We would like to thank the anonymous reviewers for their valuable feedback.

References

1. OpenMP ARB: Openmp specification. http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf (July 2013)
2. Intel Corporation: Threading building blocks. https://www.threadingbuildingblocks.org/
3. MIT CSAIL Supertech Research Group: The cilk project. http://supertech.csail.mit.edu/cilk/
4. Frigo, M., Leiserson, C.E., Randall, K.H.: The Implementation of the Cilk-5 Multithreaded Language. In: Proceedings of SIGPLAN 1998. (June 1998)
5. Mohr, E., Kranz, D.A., Halstead, R.H.: Lazy Task Creation: A technique for Increasing the Granularity of Parallel Programs. IEEE Transactions on Parallel and Distributed Systems **2**(3) (July 1991)
6. Olivier, S.L., de Supinski, B.R., Schulz, M., Prins, J.F.: Characterizing and Mitigating Work Time Inflation in Task Parallel Programs. In: Proceedings of SC12. (November 2012)
7. Tallent, N.R., Mellor-Crummey, J.M.: Effective Performance Measurement and Analysis of Multithreaded Applications. In: Proceedings of PPoPP'09. (February 2009)
8. Knüpfer, A., Brunst, H., Doleschal, J., Jurenz, M., Lieber, M., Mickler, H., Müller, M.S., Nagel, W.E. In: The Vampir Performance Analysis Tool-Set. Springer Berlin Heidelberg (2008) 139–155
9. Barcelona Supercomputing Center: Extrae User Guide Manual. (May 2013)
10. Virtual Institute - High Productivity Supercomputing: SCORE-P User Manual. (2013)
11. McCurdy, C., Vetter, J.: Memphis : Finding and Fixing NUMA-related Performance Problems on Multi-core Platforms. In: Proceedings of ISPASS 2010. (March 2010)
12. Liu, X., Mellor-Crummey, J.: Pinpointing Data Locality Problems Using Data-centric Analysis. In: Proceedings of CGO'11. (April 2011)
13. Intel Corporation: Intel VTune Amplifier XE 2013. http://software.intel.com/en-us/intel-vtune-amplifier-xe
14. Mattson, R., Gecsei, J., Slutz, D., Traiger, I.: Evaluation techniques for storage hierarchies. IBM Systems Journal **9**(2) (1970) 78–117
15. Rio Yokota: exafmm-dev. https://bitbucket.org/rioyokota/exafmm-dev
16. Taura, K., Yokota, R., Maruyama, N.: A Task Parallelism Meets Fast Multipole Methods. In: Proceedings of the SCALA'12 workshop. (November 2012)

17. The MassiveThreads Team: Massivethreads: A lightweight thread library for high productivity languages. http://code.google.com/p/massivethreads/
18. Nakashima, J., Nakatani, S., Taura, K.: Design and implementation of a customizable work stealing scheduler. In: Proceedings of the 3rd International Workshop on Runtime and Operating Systems for Supercomputers. ROSS '13 (2013) 9:1–9:8
19. Intel Corporation: TBB: Scheduling algorithm. http://www.threadingbuildingblocks.org/docs/help/reference/task_scheduler/scheduling_algorithm.htm
20. Acar, U.A., Blelloch, G.E., Blumofe, R.D.: The Data Locality of Work Stealing. In: Proceedings of SPAA'00. (2000)
21. The Qthread Team: The qthread library. http://www.cs.sandia.gov/qthreads/
22. Wheeler, K., Murphy, R., Thain, D.: Qthreads: An API for programming with millions of lightweight threads. In: IEEE International Symposium on Parallel and Distributed Processing. (2008) 1–8
23. Olivier, S.L., Porterfield, A.K., Wheeler, K.B., Prins, J.F.: Scheduling Task Parallelism on Multi-Socket Multicore Systems. In: Proceedings of ROSS'11. (2011) 49–56
24. M.Weaver, V.: Linux perf_event Features and Overhead. In: Proceedings of the 2013 FastPath Workshop. (2013)
25. Beyls, K., D'Hollander, E.H.: Reuse distance as a metric for cache behavior. In: in Proceedings of the IASTED conference on parallel and distributed computing and systems. (2001) 617–662
26. Intel Corporation: Intel 64 and ia-32 architectures software developer's manual volume 3b:. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html
27. PAPI Team: Performance application programming interface. http://icl.cs.utk.edu/papi/
28. Weinberg, J., McCracken, M.O., Strohmaier, E., Snavely, A.: Quantifying Locality In The Memory Access Patterns of HPC Applications. In: Proceedings of the 2005 ACM/IEEE conference on Supercomputing. (November 2005)
29. Corporation, I.: An Introduction to the Intel QuickPath Interconnect. (2009)
30. Hackenberg, D., Molka, D., Nagel, W.E.: Comparing Cache Architectures and Coherency Protocols on x86-64 Multicore SMP Systems. In: Proceedings of MICRO09. (December 2009)