Available Online at www.ijcsmc.com

International Journal of Computer Science and Mobile Computing

A Monthly Journal of Computer Science and Information Technology

ISSN 2320–088X
IMPACT FACTOR: 6.017

IJCSMC, Vol. 6, Issue. 6, June 2017, pg.36 – 40

Characterization of Locality Aware Task Scheduling Mechanism

Suman[1], Pinki Rani[2]

[1,2] Department of Computer Science, Kurukshetra University, Kurukshetra, India

[1] sumansaini235@gmail.com, [2] sainipinki680@gmail.com

Abstract- The architectural features of modern computers highlight the need of parallel programming for sustained performance. This paper deals with task based programming to program modern computers. Due to lack of data locality, communication optimization and lack of task characterization support in an existing task scheduling, we intends to overview the characterization of locality aware task scheduling technique which exploits home cache locality on many-core processors. Scheduling task oblivious to the locality of home caches introduces a performance bottlenecks. This paper presents a scheduling technique where runtime system controls the assignment of home caches to memory blocks and schedules tasks to minimize home cache access penalties. Various programming models supports constructs for task based parallelism like openMP, Cilk Plus [1] and Wool etc.. Explicit parallelization is required to obtain high performance on modern computer architectures.

Keywords- Locality-Aware Scheduling, Multi-core Processor, Work stealing, NUMA, Load balancing.

I. INTRODUCTION

Modern computer architectures have extreme excess of parallel features for performance. For example multiprocessor systems, multi-core processors, multithreaded cores, cores with wide vector units and specific purpose designs. The extensive set of parallel features are supported by increasingly complex cache, memory controller and communication network structures. Programmers have coped with explicit parallelization demands by directly using parallel programming mechanisms such as threads and a mix of general-purpose, vector and special ISA instructions to obtain performance. Asynchronous programming models, which allow the programmer to write

programs where a thread or process can be oblivious to what actions the other threads/processes are doing. Currently used task scheduling techniques perform poorly since they focus solely on balancing computation load across parallel features and remain oblivious to locality properties of support structures. We contribute with locality-aware task scheduling mechanisms which improve execution time performance.

Cache Hierarchy: Many-core processors are designed to exploit parallelism by implementing multiple cores that can execute in parallel. However, such a shift in design increases processor system demands. As a result the cache hierarchies on many-core processors are becoming larger and increasingly complex. Such cache hierarchies suffer from high latency and energy consumption, and non-uniform memory access effects become prevalent.

Locality: Locality can be exploited at various hardware and software layers. By implementing private and shared caches in a multi-level fashion, recent hardware designs are already optimized for locality [2]. However, this would all be useless if the software scheduling does not cast the execution in a manner that promotes locality available in the programs themselves. Exploiting locality was an option to reduce execution time and energy consumption.

II. TASK SCHEDULING

Tasks are dynamically created during runtime. The runtime system typically maintains created tasks in a task pool and schedules them on idle threads for execution when ordering constraints (data and control dependences) are satisfied as shown in Fig. 1. New child tasks created during task execution [3] are added to the pool for subsequent scheduling. Task pool data structures used by runtime system have to be chosen carefully based on how tasks are created, queued for execution and stolen by idle threads. Task-stealing is a popular scheduling technique used in many runtime systems including those supporting the Intel Cilk Plus, TBB and Wool programming models. Idle workers or *thieves* steal work from loaded workers called *victims* in Task-stealing. The choice of workers to steal from is typically random. Adjusting the granularity of tasks is an important performance technique in addition to load-balancing. Inlining is a technique to improve fine-grained tasking performance by pruning task creation to coarsen the grain size of tasks. Inlining is also called lazy task creation [4] and cut(ting)-off. Scheduling oblivious to NUMA node locality can result in unnecessary remote node accesses.

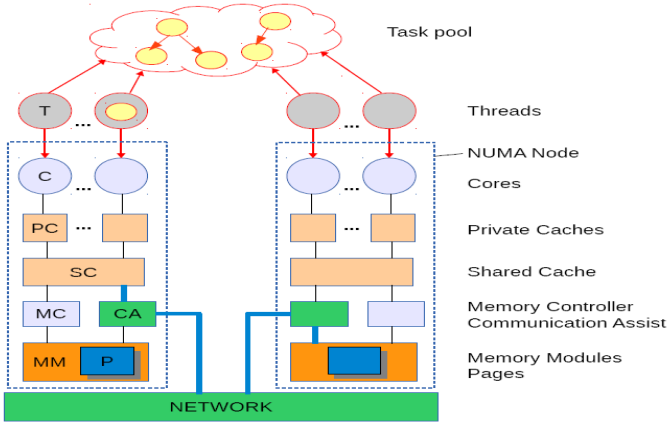

Fig. 1: Execution of tasks by the runtime system on a NUMA shared-memory system.

III. LOCALITY AWARE SCHEDULING

Recent proliferation of runtime-based programming systems stresses the need to exploit locality through runtimes. A runtime is a thin layer of efficient software that is located between the hardware (or the operating system) and the language interface, that manages resources and performs scheduling. Specifically, the scheduling algorithm of a runtime determines when and where a computation, or a task, executes. Therefore, by making the scheduling algorithm locality-aware, locality can be exploited. Unlike the operating system, runtimes are lightweight enough to manage fine-grained computation and perform dynamic load balancing. Their closer tie to the language interface also allows the runtimes to exploit high-level information to generate better schedules. Although many efforts have been made to exploit locality on a runtime, they fail to take the underlying cache hierarchy into consideration, are limited to specific programming models, and suffer high management costs.

A. The Challenges of Locality-Aware Scheduling

Exploiting locality on a many-core processor poses unique challenges. As the number of cores scales to hundreds or thousands, it will become more difficult. Challenges to develop practical locality-aware scheduler are:

1) Complexity and Generality

The complex cache hierarchies of many-core processors imply that the entire hierarchy needs to be considered when generating a schedule. However, the state of the art fails to consider complex hierarchies: Many schemes schedule tasks assuming a flat cache [5] hierarchy and those proposals that do consider cache hierarchy tend to be ad-hoc, due to the NP-hardness of scheduling. Moreover, to efficiently utilize a large-scale many-core chip, the runtime should be able to exploit locality regardless of the programming model or algorithm.

2) Management Costs

To maximize the performance gains of a locality-aware schedule, schedules should be generated and enforced at low costs. The use of fine-grained tasks makes low-cost locality-aware scheduling especially challenging. Given a fixed size input, as the core count increases, computation will have to be broken down into finer granularity. Such fine-grained tasks are more sensitive to cache misses, and many render any task management overhead visible. However, some locality-aware proposals that are based on heavyweight, managed runtimes [6] are not suitable for fine-grained task management.

3) Dynamic Load Balancing

On a large-scale many-core processor, load imbalance can take place frequently. Such imbalance occurred due to the imbalance in the software itself or due to the uneven execution of the underlying hardware. However, as core counts increase, load imbalance due to multi programming will increase. A runtime can handle dynamic load imbalance through stealing by dynamically redistributing the allocated computation. However there exists a fundamental tradeoff between locality and load balance, so many task management schemes fixate on one to sacrifice the other.

IV. METHODS TO PERFORM LOCALITY-AWARE TASK MANAGEMENT

The above challenges can be addressed by taking a systematic, framework based approach. Methods to perform locality-aware task management are :

A) **Manage Complexity with a Generic Graph-Based Framework** : Generic, graph-based locality analysis framework is both intuitive and robust. It allows to analyze the key scheduling attributes independent of hardware specifics and scale, and the resulting scheduling scheme can be applied to various cache hierarchies.

B) **Pattern-Based Task Managers to Reduce Management Costs:** An efficient locality- aware task manager can be constructed by exploiting high-level task structure information. When the explicit data dependency information is available (e.g., MapReduce [7] [8]), a runtime can exploit locality on the internal data structures used to buffer intermediate results. When such an information is not available (e.g., OpenMP [9]), the workload sharing pattern information can be conveyed to the underlying runtime to exploit locality through pattern-specific task management policies. Resulting task managers exhibit comparable performance to that from the graph analysis, while maintaining the simple task-based programming interface intact.

C) **Perform Locality-Aware Task Stealing:** Dynamic task management comprises two components:

(1) Task scheduling, or the initial assignment of tasks to threads, and

(2) Task stealing, or balancing the load by transferring tasks from one thread to another that is idling. Task stealing can be made locality-aware by honoring the specified locality-aware schedule as tasks are transferred.

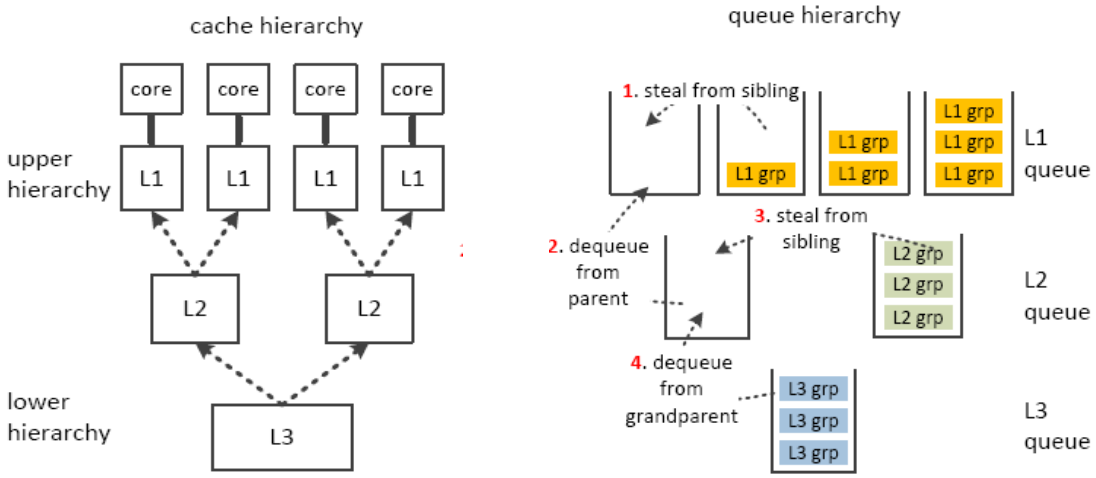

Fig. 2: Recursive stealing

In fig. 2, Top-level queues (that hold tasks) are not shown. Colored numbers indicate the order that the leftmost core visits queues. Once a task queue is empty, a thread attempts to dequeue from its L1 queue. If the L1 queue is empty, it attempts to steal from the sibling L1 queue. We interleave steals with dequeues because the L2 caches are shared. If a thread steals from a sibling L1 queue, it grabs an L1 group that shares data in the L2 cache with the task it just executed and the the tasks the sibling core(s) are currently executing. Thus we exploit the shared. If all the steal attempts fail, the thread tries a regular dequeue from the L2 queue. If the L2 queue is empty as well, it climbs down

the hierarchy and repeats the process: It attempts to steal from the sibling L2 queue, and then visits the L3 queue. When stealing, a thread grabs the task group at the tail of the victim queue, in the same way randomized stealing operates [10].

V. CONCLUSION

Many-core processors contain multiple cores supported by modern architectural features such as a distributed cache hierarchy, a high bandwidth on-chip network and multiple memory controllers. Runtime systems should consider memory behavior and on-chip communication of applications while mapping task parallelism on many core processors. The locality of home caches introduces non-uniform cache access latencies. In this paper, we quantify the performance penalty occurred by scheduling oblivious [11] to the locality of home caches. When tasks are scheduled in a locality-aware fashion, task stealing should be locality-aware as well. By preserving the task grouping and ordering information specified in the original schedule, locality can be exploited while load balancing.

REFERENCES

[1] A. D. Robison, "Composable parallel patterns with intel cilk plus," *Computing in Science & Engineering*, vol. 15, no. 2, p. 0066–71, 2013.

[2] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. In Proceedings of the 12th ACM Symposium on Parallel Algorithms and Architectures, pages 1-12, 2000.

[3] Podobas, A., Brorsson, M.: A comparison of some recent task-based parallel programming models. In: Programmability Issues for Multi-Core Computers (MULTIPROG 2010), Pisa (January 2010)

[4] E. Mohr, D. A. Kranz, and R. H. Halstead Jr, "Lazy task creation: A technique for increasing the granularity of parallel programs," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 2, no. 3, p. 264–280, 1991.

[5] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In Proceedings of the 35th Annual Symposium on Foundations of Computer Science, pages 356-368, 1994.

[6] P. Charles, C. Grotho_, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, pages 519-538, 2005.

[7] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation, pages 137-149, 2004.

[8] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating Map Reduce for multi-core and multiprocessor systems. In Proceedings of the 13th IEEE International Symposium on High Performance Computer Architecture, pages 13-24, 2007.

[9] J. Balart, A. Duran, M. Gonzàlez, X. Martorell, E. Ayguadé, and J. Labarta, "Nanos mercurium: a research compiler for openmp," in *Proceedings of the European Workshop on OpenMP*, vol. 8, 2004.

[10] K.-F. Faxén, "Wool-a work stealing library," *ACM SIGARCH Computer Architecture News*, vol. 36, no. 5, pp. 93–100, 2009

[11] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In Proceedings of the 40th Annual Symposium on Foundations of Computer Science, pages 285-297, 1999.

