KTH Information and
Communication Technology

Locality-aware Scheduling and Characterization of Task-based Programs

ANANYA MUDDUKRISHNA

Licentiate Thesis in Information and Communication Technology
Stockholm, Sweden 2014

TRITA-ICT/ECS AVH 14:01
ISSN 1653-6363
ISRN KTH/ICT/ECS/AVH-14/01-SE
ISBN: 978-91-7501-994-9

KTH School of Information
and Communication Technology
Electrum 229
SE-164 40 Kista
SWEDEN

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av teknologie licentiatexamen i informations- och kommunikationsteknik onsdagen den 5 mars 2014 klockan 10.00 i sal E.

Tryck: Universitetsservice US AB

Abstract

Modern computer architectures expose an increasing number of parallel features supported by complex memory access and communication structures. 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 on average by 44% and 11% respectively on two locality-sensitive architectures - the Tilera TILEPro64 manycore processor and an AMD Opteron 6172 processor based four socket SMP machine.

Programmers need task performance metrics such as amount of task parallelism and task memory hierarchy utilization to analyze performance of task-based programs. However, existing tools indicate performance mainly using thread-centric metrics. Programmers therefore resort to using low-level and tedious thread-centric analysis methods to infer task performance. We contribute with tools and methods to characterize task-based OpenMP programs at the level of tasks using which programmers can quickly understand important properties of the task graph such as critical path and parallelism as well as properties of individual tasks such as instruction count and memory behavior.

Acknowledgments

I thank Professor Mats Brorsson, my main supervisor, for sharp advice, helping me choose promising research topics and providing a worriless environment to build research skills. I am grateful to Peter A. Jonsson for his thorough mentoring and close collaboration. Conversations with Sverker Janson have demystified my view of research and life. I thank Vladimir Vlassov, Artur Podobas, Georgios Varisteas, Konstantin Popov, Karl-Filip Faxen, Professor Christian Schulte, Roberto Castañeda Lozano and Gabriel Hjort Blindell for advice and keeping my spirits up. Love and encouragement from Bianca, family and friends have driven every step of my research.

Contents

List of Figures

Part I

Thesis Overview

Chapter 1

Introduction

The flow of the chapter is as follows. We first provide an overview of architectural features of modern computers and highlight the need for explicit parallel programming for sustanined performance. We follow with a discussion on the suitability of task-based programming to program modern computers. We then introduce problems tackled in the thesis work. The first problem is the loss of performance due to lack of data locality and communication optimizations in existing task scheduling. The second problem we tackle is the lack of task characterization support in existing performance analysis tools. We propose solutions to both problems and list solution requirements. We summarize contributions of the thesis work briefly in the end.

Modern computer architectures expose a plethora of parallel features for performance. Examples include multiprocessor systems, multicore 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. Computer architects indicate that increasing the number of parallel features, and thereby the complexity and diversity of future architectures is the only way towards sustained performance [1].

Explicit parallelization is required to obtain high performance on modern computer architectures. Software should be written to use exposed parallel features efficiently and in a scalable manner. Auto-parallelization and auto-vectorization technology cannot replace explicit parallelization due to their meager success in parallelizing sequential code mainly due to predominance of irregular computations, use of serial algorithms and imprecise and unnecessarily serial language artifacts [2, 3]. Sequential legacy software should also be explicitly parallelized to overcome decreasing single-threaded performance [1].

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 until recently. Explicit threading is not portable - the number of threads and scheduling used depends on

available hardware parallelism. In addition, explicit threading is not a composable technique due to rigid use of threads which leads to over-subscription of system resources during nested execution. Explicit vectorization and special ISA coding are also not portable due to architectural diversity. Different vector register widths and lack of backward compatibility among accelerator generations are typical portability concerns while coding.

Task-based programming models offer an elegant solution to the portability and composability problems experienced in thread-centric programming. Programmers use portable abstractions called *tasks* to express potential parallelism in task-based programming. Potential parallelism is mapped to available hardware parallelism by a software called the *runtime system* which is implemented once per architecture. Tasks are composable since they do not impose rigid parallelism like threads. Task are also versatile since they can be used to expose both functional and data-level parallelism. Programmers can use task-based programming productively since they can focus on ways to expose abundant parallelism using tasks and delegate parallelization management to the runtime system.

Scheduling decisions of the runtime system are key to task-based program performance. Scheduling decisions are made according to scheduling policies which until now have focused mainly on load-balancing i.e., distributing computation evenly across threads. Load-balancing has the advantage of being a relatively quick decision requiring little information from task abstractions used by the programmer. Scheduling policies focused on load-balancing such as the work-stealing policy used in Cilk Plus [4] and Wool [5] and the simple central task queue based policy used in the OpenMP implementation in GCC [6] have provided excellent to acceptable performance on several generations of multicore processors and systems.

However, scheduling policies need to minimize memory access and communication costs in addition to load-balancing for sustained performance on modern architectures [3, 7]. Scheduling policies focused solely on load-balancing lose performance since they neglect data locality and communication patterns that arise during task execution. Such neglect goes against the design principles of the increasingly complex memory access and communication support structures which require scheduling to exploit data locality and communication patterns at every possible opportunity so that parallel features can run uninterrupted and provide peak performance. Performance is also lost because scheduling policies focused on load-balancing are oblivious to architectural features beyond the level of threads. Memory access cost increases and becomes more non-uniform with every architecture generation and can significantly penalize memory oblivious scheduling decisions. Performance can also take a similar hit when scheduling fails to use special communication support added to communication networks.

Minimization of memory access and communication costs requires locality-aware scheduling to be performed by the runtime system. The evolutionary step towards locality-aware scheduling requires overcoming multiple challenges. First, the overheads of locality-aware scheduling should be as small as possible. Scheduling decisions to minimize memory access and communication costs typically require deci-

sion and action times longer than the size of fine-grained tasks [8,9]. Fast heuristics and careful scheduling policy design are therefore required to keep overheads low. Second, scheduling might need additional information from the programmer requiring careful and intuitive extensions to existing interfaces. Third, the runtime system might have to shoulder additional responsibility apart from task scheduling to increase its awareness. For example, shouldering memory layout responsibility provides higher control over data locality to the runtime system but requires good interface design and low-overhead implementation. Fourth, locality-aware scheduling should be intelligently matched with load-balancing decisions which often have conflicting goals. Locality-aware scheduling concentrates the load in the vicinity of data thereby keeping data motion minimal whereas load-balancing spreads the load uniformly across the system thereby increasing data motion. Last, decisions to minimize memory access cost by latency-hiding techniques might require additional task parallelism to be exposed by the programmer at the cost of rewriting the program.

While task-based programming encourages programmers to move away from the world of threads, debugging performance problems pulls them right back in. Current state-of-the-art tools indicate performance mainly using thread states and events [10–12]. Programmers however aim to solve performance problems using task-based performance metrics. For example, programmers need to ensure that exposed task parallelism is abundant and task memory footprints fit private caches for scalable performance. Tools simply do not compute such task-based performance metrics. Programmers overcome the lack of tool support to understand task-based performance by tediously and repetitively connecting thread-specific performance information back to task instances using low-level implementation information from the compiler and runtime system.

Programmers need tool support for understanding task-based performance while solving performance problems. Good tool support can not only reduce the tedium of performance tuning but also allow programmers to understand task-based performance in a direct and fundamental manner which is not possible currently. Developing tool support to characterize task-based performance has the following technically challenging requirements. First, tools should indicate true task-based performance. Profiling often disturbs timing and cache states leading to skewed performance. Second, task-based performance information should be presented to programmer in an intuitive manner compatible with the interactive thread time-line based view used by a majority of performance visualization tools. Task-based performance data can be overwhelming since task count is typically several orders of magnitude larger than thread count. Third, tools should take care to avoid attributing performance to incorrect task instances which can happen easily in recursive and migration-enabled task execution. Such misattribution of performance leads to incorrect analysis and estimates. Last, tools should obtain task-based performance with feasible overheads independent of the type of information being profiled.

1.1 Contributions

We contribute with performance-enhancing locality-aware scheduling techniques and tool support for task-centric characterization of task-based programs. Our contributions are listed below.

- We present a locality-aware task scheduling technique which exploits *home cache* locality on the Tilera TILEPro64 manycore processor in Paper 1. The home cache is software-configurable coherence waypoint in the cache hierarchy of the TILEPro64. Scheduling tasks oblivious to the locality of home caches introduces a performance bottleneck. We have designed a scheduling technique where the runtime system controls the assignment of home caches to memory blocks and schedules tasks to minimize home cache access penalties. Our locality-aware scheduling technique shows an average execution time performance improvement of 46% in comparison to a central task queue based scheduling policy.

- We present a data distribution and locality-aware scheduling technique for task-based OpenMP programs executing on NUMA systems in Paper 2. We relieve the programmer from thinking of NUMA system details by delegating data distribution to the runtime system. We use task data dependence information to guide the scheduling of OpenMP tasks to further reduce data stall times. We demonstrate our techniques on a four socket AMD Opteron machine with eight NUMA nodes. We identify that data distribution and locality-aware task scheduling improve performance up to 11% compared to default policies and yet provide an architecture-oblivious approach for programmers.

- We provide methods to extract and visualize detailed task parallelism information from OpenMP programs in Paper 3. We demonstrate our methods by characterizing task graph properties such as exposed parallelism and length of the critical path as well as instruction count and memory behavior of individual tasks of benchmarks in the widely-used Barcelona OpenMP Tasks Suite (BOTS) [13]. We indicate the input sensitivity of and similarity between BOTS benchmarks using extracted task parallelism information. Programmers can speed up the performance tuning process by using our methods to quickly understand task parallelism.

1.2 Outline

The remainder of thesis is organized as follows.

- We present background information and related work in Chapter 2.

- We provide a summary of publications part of the thesis in Chapter 3.

- We present conclusions and future directions of our work in Chapter 4.
- Papers 1, 2 and 3 are attached as supplements in Part II.

Chapter 2

Background and Related Work

We present background information and related work pertaining to the scope of the thesis in the chapter. We first review task-based programming and present an example using OpenMP. Next, we present an overview of task scheduling techniques with focus on locality-aware scheduling. We describe performance analysis techniques and tools available for task-based programs in the end.

2.1 Task-based Programming

Task-based programming is a performance portable and productive technique in which programmers use task abstractions to express potential parallelism in applications. An architecture-specific software called the runtime system maps potential parallelism to actual hardware parallelism using parallelization mechanisms such as threads.

Task-based programming as a concept can be found in many models that offer portable abstractions to express parallelism. Popular examples of task-based models include Cilk Plus, TBB, OpenMP, OmpSs and OpenCL. An early demonstration of task-based programming can be found in the MIT Cilk runtime system project [14]. Varbanescu et al. [15] provide a comparative survey of several task-based programming models.

We show an example task-based program that computes Fibonacci numbers recursively using OpenMP in Figure 2.1a. The program also exposes a depth-based task creation cutoff for performance. Figure 2.1b reveals how the task pragma directives are transformed into runtime system calls and data structure instantiations.

We have used MIR - a task-based runtime system library developed in our group to drive all scheduling experiments discussed in the thesis. MIR supports the OpenMP tied tasks model where tasks run to completion on the same thread. We use a modified version of the Mercurium compiler [16] to translate OpenMP pragma directives to MIR library calls. MIR allows us to experiment with different

execution configurations within the same system by providing flexible scheduling and memory allocation policies. MIR allows detailed performance analysis by providing task execution traces with timing, hardware performance counter values, binary instrumentation information and task graph structure.

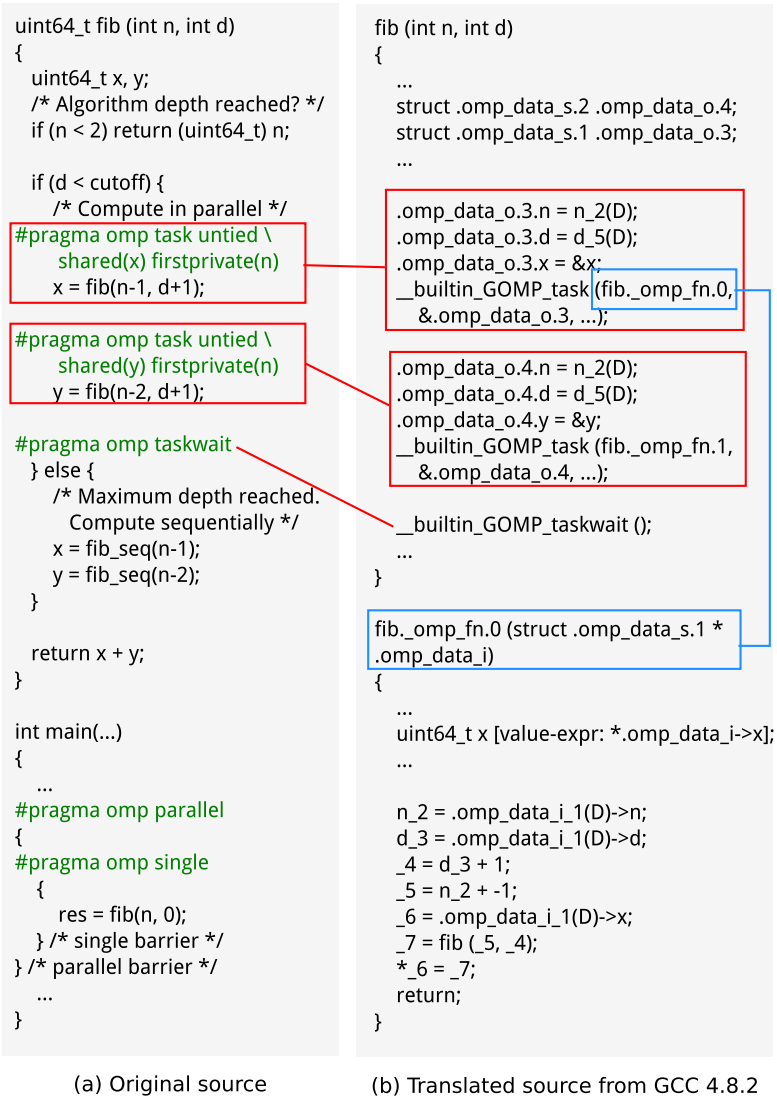


Figure 2.1: Example task-based implementation of Fibonacci number computation using OpenMP.

We show the dynamic task graph produced during execution of the Fibonacci program in Figure 2.2. The execution time and memory footprint size of tasks are indicated respectively by the size and color intensity of task nodes. Many fine-grained tasks are created when the depth-based cutoff is not used. Fine-grained tasks limit performance when their creation time becomes comparable with their execution time. Few but larger leaf-level tasks are created upon cutting-off as shown in the inset.

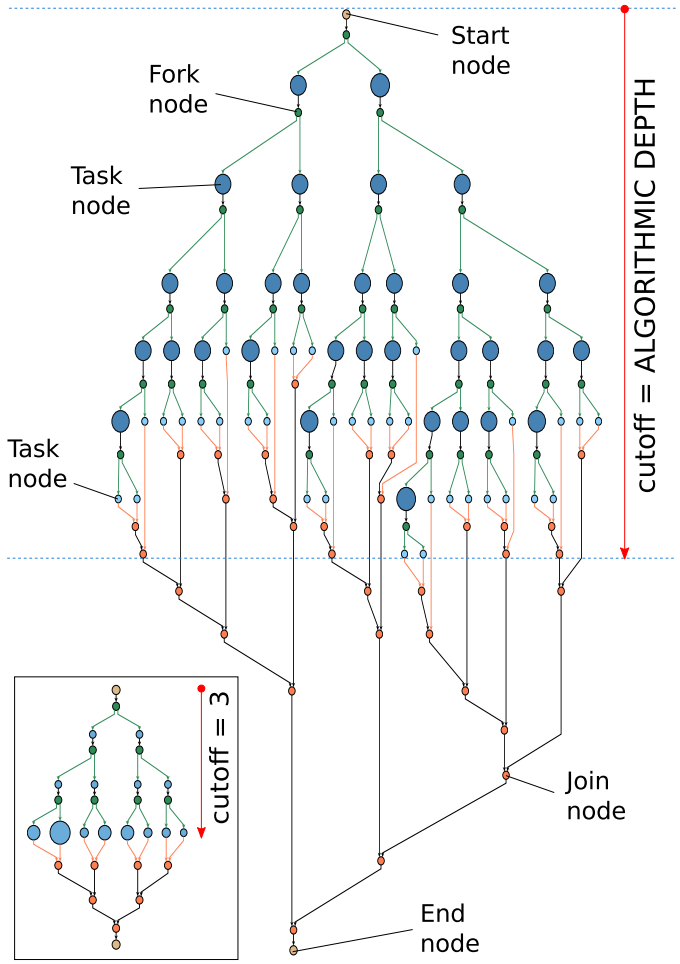


Figure 2.2: Dynamic task graph of the example Fibonacci program with input n=8. Inset shows task grain size coarsening.

2.2 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 Figure 2.3. New child tasks created during task execution are added to the pool for subsequent scheduling.

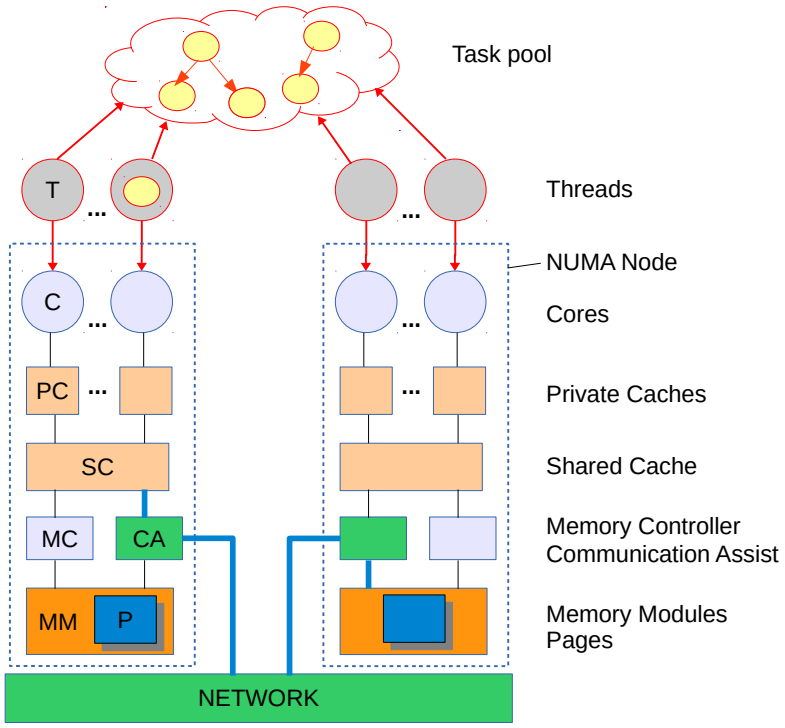


Figure 2.3: Execution of tasks by the runtime system on a NUMA shared-memory system. Scheduling oblivious to NUMA node locality can result in unnecessary remote node accesses.

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. Korch et al. [17] explore task pool implementation techniques for load-balancing. They evaluate several task pool organizations including central, distributed and hybrid (combined central and distributed) task pools with optional dynamic task stealing. Their work has inspired many features in our MIR runtime system.

Duran et al. [18] explore OpenMP task scheduling techniques based on work-first and breadth-first principles. Olivier et al. [19, 20] make an in-depth analysis of

task scheduler implementations on large multicore systems.

Work-stealing is a popular scheduling technique used in many runtime systems including those supporting the Intel Cilk Plus, TBB and SICS Wool programming models. Idle workers or *thieves* steal work from loaded workers called *victims* in work-stealing. The choice of workers to steal from is typically random. Theoretical and practical efficiency of random work-stealing combined with the work-first principle is described by the MIT Cilk runtime system project [14]. The Wool library implements *transitive leap-frogging* which improves over random victim selection [5]. Topology-aware work-stealing is employed commonly to avoid locality disruption due to random victim selection [21–23]. Work-stealing strategies parameterized by task types has been proposed to overcome performance problems introduced by global scheduling decisions [24].

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 [25] and cut(ting)-off. Both static [18] and adaptive [26] cut-off strategies for OpenMP tasks have been explored. It is rumored that the OpenMP runtime system used by Intel compiler (ICC) implements a depth-based cut-off. The `guided` scheduling option exposed by parallel looping in OpenMP is another example of inlining [27].

Locality-aware Task Scheduling

Locality-aware scheduling is crucial for performance on architectures with large, complex memory hierarchies and communication networks. For example, scheduling oblivious to locality of data in NUMA nodes can degrade performance due to unnecessary remote node accesses on NUMA machines as shown in Figure 2.3. We show a characterization of NUMA node memory access latencies performed using the BenchIT [28] tool for the AMD Opteron 6172 processor based four socket machine with 8 NUMA nodes used in our experiments in Figure 2.4. Remote node accesses for the machine are significantly more expensive than local node accesses and increase non-uniformly with the number of nodes.

Cilk-like work-stealing schedulers exploit producer-consumer locality among tasks by executing the child task immediately and queuing the parent continuation. Tasks that are close in the tasks graph are also scheduled on the same core or within the same processor thereby preserving locality in core- or chip-private caches. Acar et al. [29] provide data locality bounds for the work-stealing. They also propose a heuristic extension to the work-stealing algorithm which preserves locality between tasks that are distant in the task graph. Chen et al. [30] compare PDF - a task scheduler designed to promote constructive last-level cache sharing - against work-stealing.

Richard M. Yoo in his Ph.D. thesis describes a graph-based analysis framework for producing reference locality-aware task schedules using approximated data dependency and data sharing information from the runtime system on manycore pro-

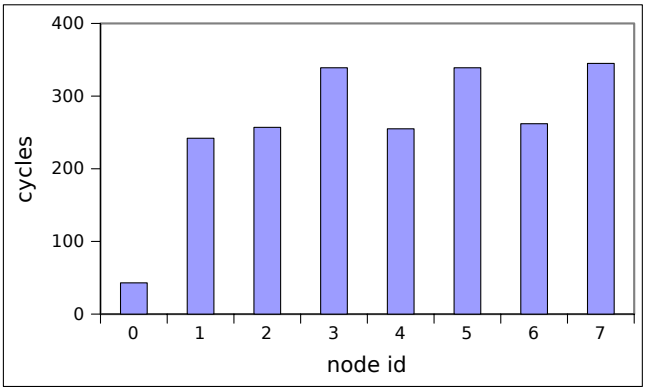


Figure 2.4: NUMA node memory access latency (4 MB) characterization of the four socket AMD Opteron machine used in our experiments. Remote node accesses are at least 5x costly.

cessors [9]. Yoo et al. [31] perform an in-depth performance analysis of task-based data-level parallelism on manycore processors. They conclude that simple heuristic based scheduling meant for recursive fork-join parallelism cannot capture the locality present in data-level parallelism applications.

Locality-aware scheduling for OpenMP has been studied extensively. Locality domains where programmers manually place tasks in abstract bins have been proposed [32, 33]. The tasks are scheduled within their domain to reduce remote node memory accesses. MTS [20] is a scheduling policy structured on the socket hierarchy of the machine. MTS uses one task queue per socket which is similar to our task queue per NUMA node. Only one idle core per socket is allowed to steal bulk work from other sockets.

Locality-aware scheduling of tasks is first-class in PGAS languages such as X10 and Chapel where programmers explicitly specify the locality of both data and tasks. SLAW is an adaptive work-stealing scheduler for Habanero Java - an X10 derivative that dynamically switches between work-first (continuation passing) and help-first (child passing) scheduling strategies based on stealing rate and stack pressure [21]. SLAW additionally improves locality by restricting stealing across programmer-defined boundaries.

A common locality-aware optimization on accelerators with private memory spaces is to balance memory transfer time back to host with scheduling dependent tasks directly on the accelerator [34–36].

2.3 Performance Analysis of Task-based Programs

Performance of task-based programs is typically analyzed using thread-based techniques. Execution time reduction and speedup of the entire program with increase

in cores are commonly used to indicate scalability [6, 37]. Idle time of threads or core utilization are typically used as a measure of load balance. Thread migration activity is typically suspected for limiting performance. A well-written guide from Intel illustrates important techniques for performance analysis of multi-threaded programs [38].

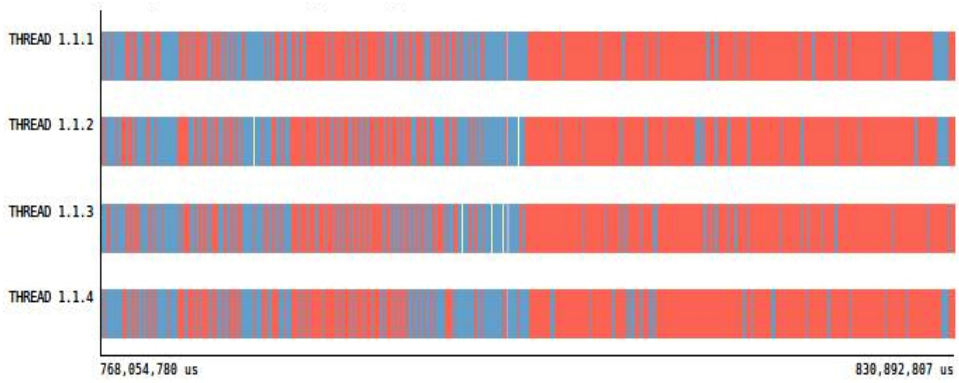
Observing performance pertaining to specific thread states and events can reveal problems in fine detail. For example, a large amount of time spent in thread states corresponding to runtime system activity such as task creation indicates the need for inlining. Reading hardware performance counter values during task execution states is typically needed to explain exact application performance [32] which is also demonstrated in Papers 1 and 2.

Paraver [39] and Vampir [40] tools support visual performance analysis using thread time-lines which show thread state and event performance. Time-lines are interactive and usually provide support for filtering and aggregation of states and events. We show a time-line visualization using Paraver of the example Fibonacci program in Figure 2.5.

The Cilkview tool [41] analyses the scalability of Cilk++ programs using established theory on work-stealing [14] and produces a speedup graph indicating scalability bounds.

The Wool runtime system library provides critical path profiling information using which task parallelism can be calculated [42]. Wool additionally provides detailed thread state times.

Call-graph techniques are also used to analyze task performance. The OmpP tool [43] and the Score-P profiling system [44] provide a textual, tree-like call-graph which connects task execution with program scope. Similarly, Intel VTune Amplifier and AMD Code Analyst tools use call-graphs to attribute thread execution to program scope which can be indirectly used to infer task performance.



(a) Zoomed-in thread timeline showing different states of execution. Note: us = cycles.

	TCREATE	TSCHED	TEXEC	TSYNC
THREAD 1.1.1	207,008,928 us	12,802,596 us	2,432,881,411 us	617,860,785 us
THREAD 1.1.2	252,741,740 us	10,692,674 us	2,444,190,131 us	562,581,693 us
THREAD 1.1.3	242,637,632 us	10,325,260 us	2,456,449,234 us	560,871,924 us
THREAD 1.1.4	241,536,922 us	10,109,086 us	2,448,710,323 us	569,509,653 us
Total	943,925,222 us	43,929,616 us	9,782,231,099 us	2,310,824,055 us
Average	235,981,305.50 us	10,982,404 us	2,445,557,774.75 us	577,706,013.75 us
Maximum	252,741,740 us	12,802,596 us	2,456,449,234 us	617,860,785 us
Minimum	207,008,928 us	10,109,086 us	2,432,881,411 us	560,871,924 us
StDev	17,287,874.22 us	1,071,396.86 us	8,531,164.82 us	23,407,892.04 us
Avg/Max	0.93	0.86	1.00	0.94

(b) Total time spent in thread states. Note: us = cycles.

Figure 2.5: Visual trace-based performance analysis of Fibonacci program execution (input: n=42, cutoff=12) using Paraver. The program has good performance since all threads spend most of their time executing tasks (TEXEC).

Chapter 3

Summary of Publications

We present a summary of publications part of the thesis in the chapter.

3.1 List of Publications

Paper 1 A. Muddukrishna, A. Podobas, M. Brorsson, and V. Vlassov, "Task scheduling on manycore processors with home caches," in *Euro-Par 2012: Parallel Processing Workshops*, pp. 357–367, Springer, 2013

Paper 2 A. Muddukrishna, P. A. Jonsson, V. Vlassov, and M. Brorsson, "Locality-aware task scheduling and data distribution on NUMA systems," in *OpenMP in the Era of Low Power Devices and Accelerators*, pp. 156–170, Springer, 2013

Paper 3 A. Muddukrishna, P. A. Jonsson, V. Vlassov, and M. Brorsson, "Characterizing task-based OpenMP programs." Submitted to ACM SIGMETRICS 2014

3.2 Summary of Paper 1

We focus on improving the performance of task-based runtime systems for manycore processors in the paper.

Manycore processors contain a large number of cores supported by highly scalable architectural features such as a distributed cache hierarchy, a high bandwidth on-chip network and multiple memory controllers. In addition, manycore processors expose a variety of software-configurable architecture features for fine tuning of performance. Runtime systems should consider memory behavior and on-chip communication of applications while mapping task parallelism on manycore processors. On top of that, runtime systems should configure architecture features properly to achieve the best application performance.

We focus on architectural locality introduced by the *home cache* feature of the Tilera TILEPro64 manycore processor. The home cache is software-configurable coherence waypoint in the distributed cache hierarchy of the TILEPro64. The locality of the home cache introduces non-uniform cache access latencies. Accesses to remote home caches take 4-6 times longer than the 12 cycle latency to the local home cache. We quantify the performance penalty incurred by scheduling oblivious to the locality of home caches.

We present a locality-aware task scheduling technique which exploits home cache locality to improve performance. Our scheduling technique controls the assignment of home caches to memory blocks according to a programmer-chosen *memory allocation policy* and schedules tasks to minimize home cache access penalties. We propose two simple memory allocation policies called *round* and *hashed*. The round policy assigns a single home cache to the allocated memory each time the application makes a successful memory allocation call. The home cache for the next allocation call is chosen in a round-robin manner from a set of pre-configured home caches. The hashed policy assigns a different home cache to each cache-line in the allocated memory each time the application makes a successful memory allocation call.

Our locality-aware scheduling policy deals tasks to specific cores and uses work-stealing to balance the load. Each task is dealt to the core that incurs the least access latency to the home caches containing input data needed by the task. We ignore output data of the task since they are buffered and incur an access latency independent of home cache locality. The scheduling policy determines home cache affinity of input data using information from the memory allocation policy. Access latencies to input data are estimated using core-to-home-cache communication latencies computed during runtime system initialization. The scheduling policy groups cores into fixed size vicinities and allows idle cores to steal tasks from cores in the same vicinity.

We have implemented memory allocation and scheduling policies in the MIR runtime system. We use three synthetic and a real-world task-based benchmark to compare the performance of the locality-aware scheduling technique against a central queue based, home cache locality oblivious scheduling policy. The three synthetic benchmarks are based on commonly occurring task composition patterns [2]. We use execution time and cache stall cycles as performance metrics.

The locality-aware scheduling policy demonstrates a significant performance improvement in execution time in comparison to the central queue locality-oblivious scheduling policy during our tests. The real-world benchmark improves performance by 45.77% while the synthetic benchmarks improve by 44.29% on average. Reduced data cache stall cycles allow us to confirm that the improvements seen are indeed due to locality-aware scheduling.

3.3 Summary of Paper 2

We focus on improving OpenMP task-based programming experience and runtime system performance for NUMA systems in the paper.

Careful data distribution and locality-aware scheduling is crucial for performance on NUMA systems. However, OpenMP neither specifies data distribution mechanisms for programmers nor provides scheduling guidelines for NUMA systems. Programmers currently distribute data using third party tools and low-level techniques and rely on the runtime system to optimize scheduling performance on NUMA systems.

We present a runtime system assisted data distribution scheme that allows programmers to control data distribution in a portable fashion without being aware of low-level system details such as NUMA node identifiers and system topology. The distribution scheme relies on the programmer to provide high-level hints in calls to `malloc` which are used by the runtime system to distribute data to different NUMA nodes on the system. We have implemented the high-level hint capability using a policy-based mechanism that has features to suit both ordinary and expert programmers.

We present a locality-aware task scheduling algorithm that leverages data distribution information to improve execution time performance on NUMA systems. The algorithm reduces memory access times by using page locality information from the data distribution scheme and task data footprint information from the programmer. The algorithm uses data footprint based heuristics and estimates of NUMA node access latencies while scheduling tasks. The algorithm additionally also allows cores to steal work preferably from nearby NUMA nodes.

The data distribution scheme and the locality-aware scheduling algorithm are implemented in the MIR runtime system. We use a mixture of pattern-based synthetic and real-world task-based benchmarks to test the performance of the combination of data distribution scheme and locality-aware scheduling algorithm. We use a third-party data distribution scheme combined with work-stealing scheduling as the comparison baseline. We use execution time and dispatch stall cycles as performance metrics.

Locality-aware scheduling for real-world applications demonstrates an execution time reduction of up to 11% compared to the work-stealing scheduler when NUMA effects degrade application performance and remains competitive in other cases. A similar comparison for pattern-based benchmarks indicates a performance improvement up to 54%. Reduced dispatch stall cycles allow us to confirm that the improvement in execution time is indeed due to locality-aware scheduling. Our measurements also demonstrate that the locality-aware scheduler can safely be used as the default scheduler for all workloads without performance degradation.

3.4 Summary of Paper 3

We focus on improving programmer understanding of task-based programs in the paper.

Performance tuning of task-based OpenMP programs is an iterative process where programmers push task parallelism to the limit at which memory memory hierarchy utilization is maximized and at the same time parallelization overheads are minimal. The iterative process typically takes a long time due to lack of tool support for understanding exposed task based parallelism. Current tools provide thread-centric information using which memory hierarchy utilization and parallelization overheads can be inferred quickly but understanding task parallelism requires tedious and repetitive attribution of thread-level information to task instances.

We describe a simple automated method to extract task parallelism exposed in task-based OpenMP programs. The method works by extracting the program task graph as well as individual task properties such as instruction mix and memory footprint using readily available compiler and binary instrumentation technology with minimal extensions to the runtime system. We processes the extracted information to derive important performance indicators such as memory sharing, computational intensity of individual tasks as well as critical path and parallelism of the task graph.

We show how to visualize task parallelism through a structure called the *Annotated Task Graph* (ATG). The ATG visually represents fork and join relationships between tasks, uses annotations to capture individual task properties and supports aggregation and filtering to help view large task graphs. Programmers can use the ATG to easily identify the shape of the task graph and quickly isolate and study interesting tasks such as those on the critical path. The ATG's visual, thread-independent representation of task parallelism complements thread specific information provided by existing performance analysis tools.

We use our tools to provide an extensive, architecture independent characterization of task parallelism in BOTS. We describe the relationship between input parameters and task parallelism to help researchers properly evaluate BOTS benchmarks. We also present an input sensitivity study of BOTS benchmarks where we quantify changes in task graph properties and architecture independent task properties with increasing data sizes. Researchers can use our results to choose BOTS inputs wisely and solve performance problems across architectures. We additionally identify BOTS benchmarks with similar task graph properties and task properties using a combination of *Principal Component Analysis* and *Hierarchical Clustering* techniques. Researchers can use our similarity analysis to find new directions to extend BOTS and avoid evaluation of redundant benchmarks-input pairs.

Chapter 4

Conclusions

We have demonstrated performance-enhancing locality-aware task scheduling mechanisms on two locality-sensitive architectures - the TILEPro64 manycore processor and a four socket Opteron multicore SMP machine. The scheduling mechanisms are simple to implement and primarily focus on minimizing access latencies to data held in remote memory locations in the distributed memory hierarchy. Since data distribution information is important for the scheduling mechanisms to work, we delegate memory allocation to the runtime system and thereby relieve the programmer from thinking about low-level memory allocation details.

We have additionally demonstrated tools and methods which enable programmers to characterize task-based OpenMP programs at the level of tasks to reveal problems in task composition and graph structure. Programmers can easily visualize task-based execution using our Annotated Task Graph structure in a thread independent manner. We have provided an extensive task-based characterization of the popular Barcelona OpenMP Tasks Suite using our tools. Our characterization will help researchers take informed decisions regarding input behavior, input sensitivity and similarity of BOTS benchmarks.

We have sketched an extension of our locality-aware scheduling mechanism on the TILEPro64 manycore processor which considers memory controller locality to minimize memory access latencies and uses special core-to-core messaging support for fine-grained task scheduling. We have additionally planned a study on adding architectural support for locality-aware task scheduling to allow better awareness and control of data locality through exposed cache coherence protocols. Extension of our methods and tools to characterize task-based programs with support for untied and implicit OpenMP tasks and a GUI to visualize and interact with the ATG are also planned.

Bibliography

[1] S. Borkar and A. A. Chien, "The future of microprocessors," *Commun. ACM*, vol. 54, p. 67–77, May 2011.

[2] M. McCool, J. Reinders, and A. Robison, *Structured Parallel Programming: Patterns for Efficient Computation.* Access Online via Elsevier, 2012.

[3] M.Duranton, D. Black-Shaffer, S. Yehia, and K. D. Bosschere, *Computing Systems: Research Challenges Ahead. The HiPEAC Vision 2011/2012.* HiPEAC Project, 2011.

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

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

[6] S. L. Olivier and J. F. Prins, "Evaluating OpenMP 3.0 run time systems on unbalanced task graphs," in *Evolving OpenMP in an Age of Extreme Parallelism*, p. 63–78, Springer, 2009.

[7] M. Duranton, S. Yehia, B. D. Sutter, K. D. Bosschere, A. Cohen, B. Falsafi, G. Gaydadjiev, M. Katevenis, J. Maebe, H. Munk, N. Navarro, A. Ramirez, O. Temam, and M. Valero, *The HiPEAC Vision.* HiPEAC Project, 2010.

[8] Z. Majo and T. R. Gross, "Matching memory access patterns and data placement for NUMA systems," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, p. 230–241, 2012.

[9] R. M. Yoo, *Locality-aware Task Management on Many-core Processors.* PhD thesis, Stanford University, 2012.

[10] M. Geimer, F. Wolf, B. J. Wylie, E. Ábrahám, D. Becker, and B. Mohr, "The scalasca performance toolset architecture," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 702–719, 2010.

[11] Intel Corporation, "Intel VTune Amplifier 2013 (document number: 326734-012)," 2013.

[12] X. Liu, J. M. Mellor-Crummey, and M. W. Fagan, "A new approach for performance analysis of openmp programs.," in *ICS*, pp. 69–80, 2013.

[13] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade, "Barcelona OpenMP tasks suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP," in *Parallel Processing, 2009. ICPP'09. International Conference on*, p. 124–131, 2009.

[14] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, *Cilk: An efficient multithreaded runtime system*, vol. 30. ACM, 1995.

[15] A. L. Varbanescu, P. Hijma, R. Van Nieuwpoort, and H. Bal, "Towards an effective unified programming model for many-cores," in *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, p. 681–692, 2011.

[16] 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.

[17] M. Korch and T. Rauber, "A comparison of task pools for dynamic load balancing of irregular algorithms," *Concurrency and Computation: Practice and Experience*, vol. 16, no. 1, p. 1–47, 2004.

[18] A. Duran, J. Corbalán, and E. Ayguadé, "Evaluation of OpenMP task scheduling strategies," in *OpenMP in a New Era of Parallelism*, pp. 100–110, Springer, 2008.

[19] S. L. Olivier, A. K. Porterfield, K. B. Wheeler, and J. F. Prins, "Scheduling task parallelism on multi-socket multicore systems," in *Proceedings of the 1st International Workshop on Runtime and Operating Systems for Supercomputers*, p. 49–56, 2011.

[20] S. L. Olivier, A. K. Porterfield, K. B. Wheeler, M. Spiegel, and J. F. Prins, "OpenMP task scheduling strategies for multicore NUMA systems," *International Journal of High Performance Computing Applications*, vol. 26, no. 2, p. 110–124, 2012.

[21] Y. Guo, J. Zhao, V. Cave, and V. Sarkar, "SLAW: a scalable locality-aware adaptive work-stealing scheduler," in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, p. 1–12, 2010.

[22] S. L. Olivier, A. K. Porterfield, K. B. Wheeler, M. Spiegel, and J. F. Prins, "OpenMP task scheduling strategies for multicore NUMA systems," *International Journal of High Performance Computing Applications*, vol. 26, no. 2, pp. 110–124, 2012.

[23] B. Vikranth, R. Wankar, and C. R. Rao, "Topology aware task stealing for on-chip numa multi-core processors," *Procedia Computer Science*, vol. 18, pp. 379–388, 2013.

[24] M. Wimmer, D. Cederman, J. L. Träff, and P. Tsigas, "Work-stealing with configurable scheduling strategies," in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '13, (New York, NY, USA), p. 315–316, ACM, 2013.

[25] 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.

[26] A. Duran, J. Corbalán, and E. Ayguadé, "An adaptive cut-off for task parallelism," in *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, p. 1–11, 2008.

[27] OpenMP Architecture Review Board, "OpenMP application program interface version 4.0," 2013.

[28] G. Juckeland, S. Börner, M. Kluge, S. Kölling, W. E. Nagel, S. Pflüger, H. Röding, S. Seidl, T. William, and R. Wloch, "BenchIT—Performance measurement and comparison for scientific applications," *Advances in Parallel Computing*, vol. 13, p. 501–508, 2004.

[29] U. A. Acar, G. E. Blelloch, and R. D. Blumofe, "The data locality of work stealing," in *Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, p. 1–12, 2000.

[30] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry, and C. Wilkerson, "Scheduling threads for constructive cache sharing on CMPs," tech. rep., 2007.

[31] R. M. Yoo, C. J. Hughes, C. Kim, Y.-K. Chen, and C. Kozyrakis, "Locality-aware task management for unstructured parallelism: a quantitative limit study," in *Proceedings of the 25th ACM symposium on Parallelism in algorithms and architectures*, p. 315–325, 2013.

[32] S. L. Olivier, B. R. de Supinski, M. Schulz, and J. F. Prins, "Characterizing and mitigating work time inflation in task parallel programs," in *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, p. 1–12, 2012.

[33] M. Wittmann and G. Hager, "Optimizing ccNUMA locality for task-parallel execution under OpenMP and TBB on multicore-based systems," *arXiv preprint arXiv:1101.0093*, 2010.

[34] C. Augonnet, S. Thibault, and R. Namyst, "StarPU: a runtime system for scheduling tasks over accelerator-based multicore machines," 2010.

[35] J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguade, and J. Labarta, "Productive programming of GPU clusters with OmpSs," in *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, p. 557–568, 2012.

[36] A. Podobas, M. Brorsson, and V. Vlassov, "Exploring heterogeneous scheduling using the task-centric programming model," in *Euro-Par 2012: Parallel Processing Workshops*, p. 133–144, 2013.

[37] A. Podobas and M. Brorsson, "A comparison of some recent task-based parallel programming models," in *Proceedings of the 3rd Workshop on Programmability Issues for Multi-Core Computers,(MULTIPROG'2010), Jan 2010, Pisa*, 2010.

[38] Intel Corporation, "Intel guide for developing multithreaded applications," 2011.

[39] V. Pillet, J. Labarta, T. Cortes, and S. Girona, "Paraver: A tool to visualize and analyze parallel code," in *Proceedings of WoTUG-18: Transputer and occam Developments*, vol. 44, pp. 17–31, 1995.

[40] H. Brunst and B. Mohr, "Performance analysis of large-scale OpenMP and hybrid MPI/OpenMP applications with Vampir NG," in *OpenMP Shared Memory Parallel Programming*, pp. 5–14, Springer, 2008.

[41] Y. He, C. E. Leiserson, and W. M. Leiserson, "The Cilkview scalability analyzer," in *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, pp. 145–156, ACM, 2010.

[42] K.-F. Faxén, "Wool 0.1 user's guide," 2013.

[43] K. Fürlinger and D. Skinner, "Performance profiling for OpenMP tasks," in *Evolving OpenMP in an Age of Extreme Parallelism*, p. 132–139, Springer, 2009.

[44] D. Lorenz, P. Philippen, D. Schmidl, and F. Wolf, "Profiling of OpenMP tasks with score-p," in *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*, p. 444–453, 2012.

[45] A. Muddukrishna, A. Podobas, M. Brorsson, and V. Vlassov, "Task scheduling on manycore processors with home caches," in *Euro-Par 2012: Parallel Processing Workshops*, pp. 357–367, Springer, 2013.

[46] A. Muddukrishna, P. A. Jonsson, V. Vlassov, and M. Brorsson, "Locality-aware task scheduling and data distribution on NUMA systems," in *OpenMP in the Era of Low Power Devices and Accelerators*, pp. 156–170, Springer, 2013.

[47] A. Muddukrishna, P. A. Jonsson, V. Vlassov, and M. Brorsson, "Characterizing task-based OpenMP programs." Submitted to ACM SIGMETRICS 2014.

Part II

Publications