A Locality Approach to Architecture-aware Task-scheduling in OpenMP

Ananya Muddukrishna
KTH Royal Institute of Technology
Stockholm, Sweden
ananya@kth.se

Mats Brorsson
KTH Royal Institute of Technology
Stockholm, Sweden
matsbror@kth.se

Vladimir Vlassov
KTH Royal Institute of Technology
Stockholm, Sweden
vladv@kth.se

ABSTRACT

Multicore and other parallel computer systems increasingly expose architectural aspects such as different memory access latencies depending on the physical memory address/location. In order to achieve high performance, programmers need to take these non-uniformities into consideration but this not only complicates the programming process but also leads to code that is not performance portable between different architectures.

Task-centric programming models, such as OpenMP tasks, relieve the programmer from explicitly mapping computation on threads while still enabling effective resource management. We propose a task scheduling approach which uses programmer annotations and architecture awareness to identify the location of data regions that are operated upon by an OpenMP task. We have made an initial implementation of such a locality-aware OpenMP task scheduler for the Tilera TilerPro64 architecture and provide some initial results showing its effectiveness in fulfilling the need to minimize non-uniform access latencies to data and resources.

1. INTRODUCTION

High-end servers built on AMD Hypertransport or Intel Quickpath interconnects exhibit NUMA (non-uniform memory access) characteristics. As an example, a four-socket AMD server typically connects each processor with two others using Hypertransport. Each processor has one DRAM memory controller so there are at least three different latencies to DRAM memory. Accesses to the local node is the fastest. Accesses to memory belonging to neighboring nodes adds some 40 ns and yet another 40 ns is added for accesses to the nodes which are two Hypertransport links away.

Another example where memory locality matters is in the TilePro64 architecture from Tilera. On this processor, sixty four tiles share an on-chip distributed shared cache and use four memory controllers to access memory pages. Therefore, non-uniform cache miss latencies are observed depending on which slice of the shared cache and which memory controller was used to bring in the missing cache line. This characteristic is not always easy to deal with and programmers must often resort to interleaving application memory across different caches and memory controllers.

OpenMP tasks is a popular and industry-accepted programming model. For certain OpenMP applications on the TilePro64, we can take knowledge of which data region a task uses and whether these regions are homed at a particular tile or not and schedule the task on a tile which homes most of the data used by a task. We have made a prototype implementation of such a scheduling policy in the experimental OpenMP runtime system Nanos++ [1] and present here an initial study on its effectiveness on the TilePro64 architecture. As far as we know, no other OpenMP task schedulers have been presented before that take locality aspects into account. We also propose an extension to the OpenMP directives to allow specification of which data regions are used by a task in order to relieve the runtime system from inferring this indirectly. We find that there indeed is performance to be saved by placing tasks on the right core if the memory access pattern is such that it allows it. For the evaluation architecture, however, it is surprisingly hard to predict whether accesses to locally homed data will have substantially lower latency as compared to remotely homed data.

2. ARCHITECTURAL LOCALITY

The spreading of processing units and caches across the chip area of existing and future multicore processors leads to an architecture with non-uniform communication latencies which depend on the physical location of on-chip resources. The TilePro64 processor is an example where strong notions of architectural locality exist. The TilePro64 conforms to a distributed shared cache architecture where a processing unit (core) and a slice of the shared last-level L2 cache are bundled into a structure known as a *tile* and tiles are distributed across the chip in a regular manner. With this architecture, a core can access its local shared L2 slice faster than other off-tile L2 slices. In effect, an off-tile shared cache slice becomes an additional L3 cache for that tile.

On the TilePro64, a block of main memory is mapped to a specific last-level cache slice called the *home*. Loads issued by cores are met by first bringing in the block of main memory into the home and then sending the home allocated block to the local L2 slice. Cache misses to blocks of main memory homed locally can therefore be loaded faster than blocks that are homed remotely. Also, since the TilePro64 employs a write-through cache policy, writes are forwarded to the home tile and invalidations are sent to local copies in any other L2 cache. Therefore, cache re-use of written data is much faster on the tile that homes the written block. The TilePro64 architecture has in total 64 tiles with 8 kB private L1 instruction and data caches and a 64 kB large slice of the L2 cache which acts as a normal L2 cache and as

an L3 cache for all tiles accessing data homed at this tile.

The mapping of main memory blocks to homes is greatly configurable on the TilePro64 by means of hashing functions. For example, *hash-for-home* is a vendor hashing function provided by Tilera which spreads all main memory blocks contained within in a OS virtual memory page to a configurable set of homes interleaved on a cache block basis. The default behavior of hash-for-home, which is the one used in this paper, is that cache blocks are interleaved with homes across all tiles on the chip.

In order to exploit task memory access patterns effectively on distributed shared cache architectures, allocation of data structures has to be done carefully with access latency in mind. We illustrate this idea by quantifying the impact of homing decisions while allocating data on the TilePro64. We consider a simple OpenMP application called *home-test*. The home-test application creates tasks that make references to blocks of memory called *regions*. These regions are either allocated such that they are all homed on a single L2 slice, or spread across all available L2 slices using TilePro64 hash-for-home, or every region is homed on a specific L2 slice. We then schedule the tasks using these different allocation schemes and collect statistics using hardware counters to illustrate the performance effect of homing decisions. Task execution time is measured using cycle counters. We also count the number of L1 misses that go to data homed either locally or remotely.

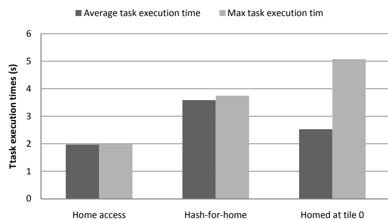

Figure 1: Average and maximum task execution times.

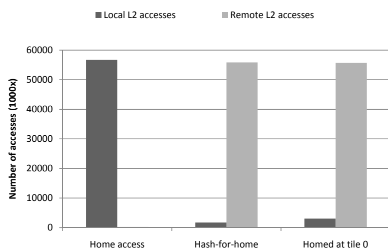

Figure 2: Average number of accesses to L2 where the home is local or remote.

Figure 1 shows the resulting average task and maximum execution time. The absolute values are not interesting but rather the relative difference between the three allocations. To home data where it is accessed here shows a 43% improvement over the hash-for-home policy which distributes the data across all tiles. Hash-for-home represents a uniformly "bad" policy but it has the great advantage that the aggregate L2 caches of all tiles effectively work as a large L3 cache. The single-tile allocation is on average a little better than hash-for-home, since at least one core has local access, but the effect on the maximum task execution time is devastating.

Figure 2 shows the causes behind the differences in execution times. It shows the average number of data accesses done by each task to data homed either locally or remotely. The home access scheme has virtually no remote data references which explains the low average and maximum task execution time.

This simple experiment illustrates the importance of taking architectural locality aspects into account when mapping data and/or computations onto a tiled architecture with non-uniform communication and memory access costs. To use this inference effectively, application data structures need to be homed with task access patterns in mind and the corresponding homing information must be conveyed to the runtime. This is easier said than done as it involves substantial compiler, interface design and memory management work. As our research is in its initial proof-of-concept phase, we assume that the programmer can explicitly expose task memory access patterns to the runtime system and allocate application data on specific homes tiles.

3. TAKING LOCALITY INTO CONSIDERATION IN OPENMP

In order to take architectural locality into account in an OpenMP program we need to have: (i) a mechanism to communicate to the runtime system which data regions a task will operate on (ii) a scheduling algorithm that discerns the home of the data regions of a task and uses this information while scheduling the task. Furthermore, there is a choice on who makes the mapping of memory regions to different homing policies. For now, we just assume that the latter is done by the programmer in some ad-hoc manner allocating memory regions to be accessed by tasks homed on tiles in a round-robin fashion.

For this initial study we are only concerned with specifying data regions of memory used by a task. We have identified the need to specify either a single region of memory used or a sequence of regions. To allow the expression regularity in the access pattern of tasks, we begin with a simple extension called `range` which indicates specific sections of the application data structure which will be read or written by a task. The C syntax of the extension is shown below:

```
#pragma omp task range(pointer, size, "r|w|rw")
```

The `range` extension is a clause to the `task` construct which specifies the start address and size of shared memory that will read (`r`) or written (`w`) or both (`rw`) by the task. The compiler interprets the specified ranges and passes this information to the runtime system. For the moment, we assume that the range is homed at some tile and that this information is known to the runtime system.

In order to specify the common situation when a task touches more than one memory region, we propose a syntax where each region is is named and registered with the runtime system and then a task can specify a range of region names it

uses. The task range syntax can also be extended to use pointer names and registered names for regions interchangeably.

```
#pragma omp region(name, ptr, size, "r|w|rw")
#pragma omp task ranges(name1, name2, ...)
```

3.1 Home-based scheduling policy

We use the experimental Nanos++ runtime system [1] to implement a simple task scheduling policy called the Home Scheduling (HS) policy which schedules a task based on where its data regions are homed. The HS policy uses distributed task queues, one for each thread. A newly spawned task is queued on the thread whose cache homes one of the regions that the task will operate on. This thread is known as the home thread which in Nanos++, is bound to a core. The cache associated with a home thread is known as the home cache. The work sharing algorithm of the HS policy can be configured to disregard certain home caches based on whether they are read, written or read-written. To balance the load, the HS policy always picks the least loaded home thread. To balance the load further, the HS policy permits unrestricted work-stealing within a vicinity of cores. The HS policy uses the latency information gathered by the architecture graph and constructs fixed sized vicinities by grouping home caches which have a low latency of communication among each other. Different vicinity configurations are shown in Figure 3.

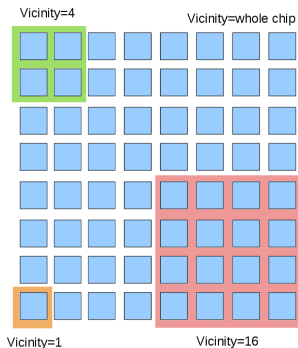

Figure 3: Vicinity configurations of the HS policy.

There are two obvious improvements to the HS scheme, one with respect to work-sharing and one with respect to work-stealing. We could make an analysis on previous work-sharing decisions and avoid placing tasks on cores which already has been assigned a task and instead pick a core based on some other region specification. Currently we only look at the overall load on the possible cores (based on the region specification). The second improvement is to steal only tasks that already has a memory region specification to the local core who needs to steal work. Currently we steal randomly within the vicinity.

4. EXPERIMENTS

We consider the SparseLU linear algebra benchmark from the BOTS [3] suite for testing the HS policy. The SparseLU benchmark performs the LU factorization of a sparsely allocated matrix which consists of sub-matrices. The tasks within the benchmark exhibit regular memory access patterns and work on a maximum of three different sub-matrices.

We use the Cilk-like (Nanos_Cilk) and Breadth-first (Nanos_BF) task scheduling policies of Nanos++ to compare the execution performance of HS. The Nanos_Cilk scheduling policy is characterized by eager child task execution, distributed task queues and random work-stealing. The Nanos_BF policy is characterized by global task queue from which all threads pick tasks for execution. Both the Nanos_BF and Nanos_Cilk scheduling policies have very good load balancing capabilities.

4.1 Methodology

We run the SparseLU benchmark on a 30X30 matrix with 125X125 floating-point sub-matrices using HS, Nanos_BF and Nanos_Cilk scheduling policies for 50 threads on the TilePro64. We allocate the sparse matrix using the hash-for-home scheme for tests with Nanos_Cilk and Nanos_BF. For testing the the HS policy, we allocate the sparse matrix by homing each sub-matrix on a different home cache. Task definitions are then annotated with this specific homing information. We also perform tests with vicinities of different sizes of 1, 4, 16 and 50. Note that a HS vicinity of 1 implies no stealing and a vicinity of 50 implies full stealing. We use performance counters on the TilePro64 to collect local and remotely homed cache access statistics.

4.2 Results

Figure 4 shows the execution times of SparseLU factorization for the different scheduling policies. For the HS policy, we have used different vicinity settings for stealing between 1 (no-stealing), 4, 16 or all 50 cores respectively called HS_1, HS_4, HS_16 and HS_50. We see here that home based scheduling with the largest steal vicinity, HS_50, performs the best with BF scheduling approximately as good. Home based scheduling without work-stealing, HS_1, obviously leads to load imbalance also illustrated in figure 5.

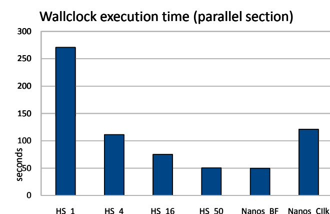

Figure 4: Execution time of SparseLU for different scheduling policies.

In order to study the effectiveness of the home scheduling policy we have measured the number of L1 cache misses for local and remotely homed data. This is shown in figure 6. Clearly home-based scheduling without work-stealing has the highest fraction of local accesses but poor performance without using work-stealing. More interesting is the fact that the locality oblivious Nanos_BF scheduler performs just as well as the HS_50 scheduler. We believe that this is due to the on-chip cache load bandwidth improvement offered by the hash-for-home distribution of task data for the Nanos_BF scheduler. The Nanos_BF scheduler incurs a very

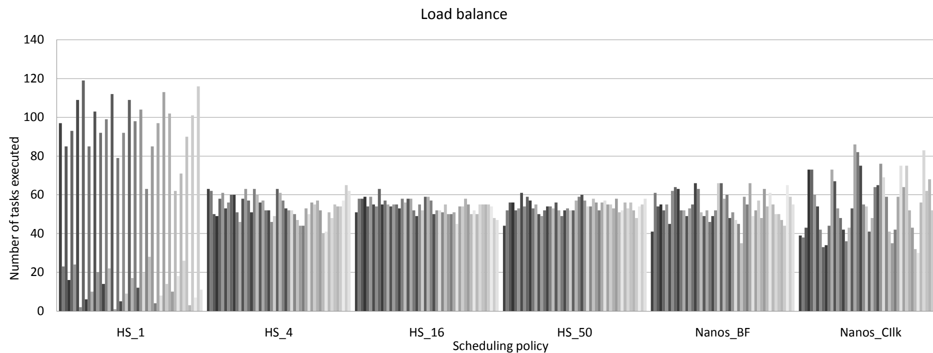

Figure 5: Number of tasks executed per thread for different scheduler types.

small scheduling decision overhead since it randomly picks tasks out of a global queue. In comparison, the HS scheduler incurs a larger scheduling overhead since it has to decide where the largest task data region is homed. The picking of the largest region is particularly detrimental while scheduling tasks of the SparseLU benchmark which operate on upto three regions (sub-matrices) of identical sizes. Also the unrestricted vicinity stealing heuristic leads to locality oblivious task execution on threads which steal successfully. The HS scheduling policy therefore needs to be tuned, as part of future work, with a latency cost model that minimizes the the latency of loading all regions and not just the largest region.

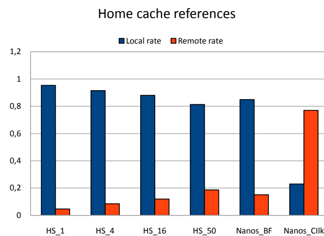

Figure 6: The ratio between local vs remote memory accesses for the different schedulers.

5. RELATED WORK

Distributed shared cache architectures are relatively new and efforts to exploit architectural locality on them are few and recent. Our work draws motivation from one of the early efforts to analyze the impact of data distribution across distributed on-chip caches by [4]. Realizing the need to match program memory access behavior with hashed memory block mapping schemes imposed by hardware, they perform compiler-based data-layout transformations such that accesses to remote caches are minimized. Our work aims to perform the similar data-layout transformations dynamically at the runtime level supported by programmer hints and architecture awareness. [2] in their recent express concerns over deepening memory hierarchies on modern processors and implement runtime OpenMP thread and data placement strategies guided by programmer and compiler hints and by using architecture awareness. Our work aims to progress in a direction similar to theirs. The idea of keeping tasks and associated data close together on a given architecture is key in high-performance computing languages such X10 and Chapel. A recent work on these languages by [5] builds a tree like notion of the memory hierarchy and first allocates user defined data structures on this tree followed by an affinity-based placement of tasks. Our work is similar in principle, but currently relies on the programmer to perform allocation of data on the cache hierarchy.

6. CONCLUSION

We have presented an initial approach on how to deal with architectural locality for OpenMP tasks and exemplified it as a prototype implementation in a runtime system and measured some key aspects on the Tilera TilePro64 architecture. While there are some obvious improvements to be made to our scheduling policy, we still have some encouraging results showing how we can utilize the idea of home mapping cache blocks onto tiles to improve performance. Our immediate future work include, besides implementing the already outlined improvements, studying applications in more detail to understand when locality can and should be exploited.

7. ACKNOWLEDGMENTS

The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under the ENCORE Project (`www.encore-project.eu`), grant agreement nr. 248647.

The authors are members of the HiPEAC European network of Excellence (`http://www.hipeac.net`).

8. REFERENCES

[1] 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*, volume 2004, 2004.

[2] F. Broquedis, O. Aumage, B. Goglin, S. Thibault, P.-A. Wacrenier, and R. Namyst. Structuring the execution of OpenMP applications for multicore architectures. In *Proceedings of 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS'10)*, Atlanta, GA, Apr. 2010. IEEE Computer Society Press.

[3] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguadé. Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP. In *38th International Conference on Parallel Processing (ICPP '09)*, page

124–131, Vienna, Austria, September 2009. IEEE Computer Society, IEEE Computer Society.

[4] Q. Lu, C. Alias, U. Bondhugula, T. Henretty, S. Krishnamoorthy, J. Ramanujam, A. Rountev, P. Sadayappan, Y. Chen, H. Lin, and T.-f. Ngai. Data layout transformation for enhancing data locality on nuca chip multiprocessors. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 348–357, Washington, DC, USA, 2009. IEEE Computer Society.

[5] Y. Yan, J. Zhao, Y. Guo, and V. Sarkar. Hierarchical place trees: A portable abstraction for task parallelism and data movement. In *LCPC'09*, pages 172–187, 2009.