Lprof: A tool for Profiling Locality awareness in a Task-based Programming Model

Evangelos Kafentarakis

Thesis submitted in partial fulfillment of the requirements for the Masters’ of Science degree in Computer Science

University of Crete
School of Sciences and Engineering
Computer Science Department
Knossou Av., P.O. Box 2208, Heraklion, GR-71409, Greece

Thesis Advisors: Prof. Angelos Bilas

This work has been performed at the Foundation for Research and Technology-Hellas
Lprof: A tool for Profiling Locality awareness in a Task-based Programming Model

Thesis submitted by

Evangelos Kafentarakis

in partial fulfillment of the requirements for the Masters’ of Science degree in Computer Science

THESIS APPROVAL

Author: Evangelos Kafentarakis

Committee approvals:

Angelos Bilas, Professor, Thesis Supervisor
Computer Science Department
University of Crete

Manolis GH Katevenis, Professor, Committee Member
Computer Science Department
University of Crete

Dimitrios S. Nikolopoulos, Professor, Committee Member
School of Electronics, Electrical Engineering and Computer Science
Queen’s University of Belfast

Departmental approval:

Angelos Bilas
Professor, Director of Graduate Studies

Heraklion, July 2013
Abstract

In this work we present lprof, a profiling tool for measuring locality of task-parallel executions. lprof traces the execution of task-parallel programs and computes locality metrics like reuse distance and NUMA distance. We use these metrics to classify the reuse distances of each execution into classes of cost according to the hardware model and NUMA characteristics of the system. We propose a methodology for computing and visualizing the reuse distance according to each class and compare a set of scheduling techniques in terms of locality.
Περίληψη

Στην εργασία αυτή παρουσιάζουμε το lprof, ένα εργαλείο μέτρησης της τοπικότητας task-parallel εκτελέσεων. Το lprof ανιχνεύει την εκτέλεση task-parallel προγραμμάτων και υπολογίζει μετρήσεις τοπικότητας, όπως απόσταση επαναχρησιμοποίησης δεδομένων και απόσταση NUMA. Χρησιμοποιούμε αυτές τις μετρήσεις για να ταξινομήσουμε τις αποστάσεις επαναχρησιμοποίησης δεδομένων για κάθε εκτέλεση σε κατηγορίες κόστους σύμφωνα με το hardware και χαρακτηριστικά NUMA του συστήματος. Προτείνουμε μια μεθοδολογία για την υπολογισμό και την απεικόνιση της απόστασης επαναχρησιμοποίησης δεδομένων ανάλογα με την κάθε κατηγορία και συγκρίνουμε μια σειρά από τεχνικές χρονοπρογραμματισμού που έχουν σχέση με την τοπικότητα.
Acknowledgements

I would like to thank my thesis supervisors, Pr. Angelos Bilas and Pr. Dimitris Nikolopoulos for their guidance in the years of my studies. I also like to thank my friends and colleagues from CARV at FORTH for their support and valuable insights about my work. I especially like to thank Dr. Polyvios Pratikakis for his guidance and help during the last year of my work.
Chapter 1

Introduction

During the last few years, the high usage of multicore architectures has led most of the programmers to create parallel applications using well-established methods of programming, such as thread-based and event-based programming. Nevertheless, these methods can lead to complex and difficultly managed source code especially when used by less experienced programmers.

Task-based programming models provide an alternative to the above methods and have as goal to reduce the complexity of programming multicore architectures. They work by allowing the programmer to specify parts of code as tasks which are then scheduled by the runtime system to run concurrently on multicore architectures, thus achieving parallel execution in parts of an application without the increased complexity of thread-based and event-based programming.

An important issue when programming multicore architectures is the locality factor. Locality is how physically close a processor core finds the data it requests from the memory. It is a very important performance factor for an application because improving locality leads to lower data access times [1]. Figure 1.1 shows an example of memory levels on a NUMA architecture system with two CPUs and two cores for each CPU. A core tries to fetch the data from the following places:

- Its L1 cache.
- Its L2 cache.
- The L2 cache of the other core in the same chip.
- The L3 cache, which is shared by both cores in the chips. This means that both cores can write it.
- The L2 caches of the cores in the remote chip.
- The L3 cache of the remote chip.
- The main memory in its local NUMA partition.
- The main memory in the remote NUMA partition.

The more of the above steps a core takes to find the data it needs the more delay it adds to the access time. Especially the last four steps add the larger amount of delay because fetching data from a remote chip or a main memory is significantly
CHAPTER 1. INTRODUCTION

Figure 1.1: Memory levels on a NUMA architecture system with two CPUs with two cores each.

slower than accessing local caches. The goal when improving locality is to allow a core to take as fewer steps as possible to find the data it needs.

Given the fact that systems become more and more parallel, the placement of the data becomes a more crucial performance factor. Therefore, the need for improving locality is increased and, as a result, so is the need for finding an accurate way of profiling a parallel application in terms of the locality factor. A way of profiling locality is measuring the reuse distance. Reuse distance is the distance between two accesses of the same data. Generally, it is difficult to accurately profile reuse distance in parallel executions due to the lack of determinism in the placement of the threads and the data. Trying to determine the reuse distance of every byte of data can be almost impossible due to the large amount of information that need to be processed and, even then, the lack of determinism will affect the variations of the results and the reliability of the profiler.

Task-based programming models have some characteristics that can help in improving the reliability of the profiling process and its performance. The first characteristic is that the scheduling policies of task-based programming models are stricter, therefore, the variations in the placement of the data are very small compared to the more traditional methods and that helps to produce more accurate results. The second characteristic is that, for each task, we know exactly which data it touches and therefore, it is not necessary to measure the reuse distances of every byte of data, but we can use more coarse-grain metrics that will help to improve the performance without losing significant precision in the results.

This work’s purpose is to create a tool for profiling applications implemented using a task-based programming model. We are focusing on NUMA architectures and we are interested in the reuse distance of an application’s data for the levels below L2 cache that include the last level cache (LLCache) and main memory ac-
cesses. We are focusing at these parts because they make up the most significant percentage of the total time spent on data accesses. The task-based programming model in which we base our tool is BDDT. BDDT uses block-based dependence analysis to resolve possible dependencies between tasks [2]. It allows a programmer to declare specific functions as tasks and it handles the job of resolving their dependencies and distributing them to be executed on the cores of the system. Another purpose of the tool is to profile different scheduling policies of BDDT regarding the amount of consideration they give to the locality factor. The contributions of this work are:

- We created a numa-aware methodology for profiling reuse distance metrics in parallel executions.

- The profiler is insensitive to non-deterministic executions and the results it produces have very small variations.
Chapter 2

Design - Metrics

In order to be able to profile and make comparisons between different scheduling policies we introduce the metric \textit{block distance}, and specify the terms \textit{producer} and \textit{consumer}. Given a block \( b \) that is touched by tasks \( (T_1 \ldots T_n) \), we define as consumer a \( T_i \) \( (1 \leq i \leq n) \) which is a reader of \( b \). We also define as producer a \( T_k \) \( (k < i) \) which can be either reader or writer of \( b \) with no other writers of \( b \) between \( T_k \) and \( T_i \). Although a reader does not \textit{produce} a block, it may have a valid copy of the block close to it and the consumer may have multiple possible locations from which it can fetch the block and that is why it can be considered a producer. For example, on a system with two processor chips, if a task runs on chip 0 and writes a block, the block will be located on the LLCache of chip 0. If then another task runs on chip 1 and reads that same block, it will create a copy on the LLCache of chip 1. If a third task (the consumer) tries to read the same block and meanwhile the blocks are not removed from the caches, it can fetch the block from two possible locations. If the consumer runs on chip 0 it will fetch the block from the LLCache of chip 0 and, respectively, the same will happen if the consumer runs on chip 1. Therefore, on the former case, the profiler will consider the writer as producer and on the latter case it will consider the first reader as producer.

We define block distance as the number of data blocks that the cores of a chip accessed after one of the chip’s cores executed a producer and before the execution of a consumer by any core of the system. If this distance is small then there is a bigger chance for the data of the producer to have stayed in the LLCache of the chip that executed the producer.

The most precise trace analysis would be achieved by gathering and processing information about every byte of the data used by the application. This would mean that we would be using block size equal to 1 byte. Such choice would produce a very large amount of metadata which would increase the profiling time and the demand for memory to a degree that would make the profiling unfeasible. Therefore, the block size has the same effects to the profiling as has to the dependence analysis, meaning that smaller sizes lead to increased memory usage and computation time,
but larger sizes lead to less precise results. The block can be of any size but we choose the same size that is used for the dependence analysis.

Figure 2.1 shows an example of an execution of nine tasks in three cores, with Cores 0 and 1 belonging to processor Chip 0 and Core 2 belonging to Chip 1. We model time in the Y axis of the figure, increasing downwards. This means Task 1 will be the first to be executed on Core 0 of Chip 0 and it will be followed by Tasks 4 and 7. Respectively, Task 3 will be the first to be executed on Core 1 of Chip 0 and it will be followed by Tasks 5 and 9, and Task 2 will be the first to be executed on Core 2 of Chip 1 and it will be followed by Tasks 6 and 8. The tasks are numbered according to the global start order. This means that Task 1 will be globally the first to start running, Task 2 will be the second etc. The application uses five regions of memory. Tasks 1 and 8 use region A, Tasks 2 and 5 use two overlapping parts of region D, Tasks 3 and 9 use the same part of region B, Task 6 uses another part of region B, Task 4 uses region C and Task 7 uses region E. Data block b belongs to memory region A. Task 1 writes block b and Task 8 reads it. The Block distance of b is equal to the sum of the number of blocks that belong to the footprints of the tasks that started running on the cores of Chip 0 after the end...
of Task 1 and before the start of Task 8. These tasks are 3, 4, 5 and 7. There was no perfect choice between including to the block distance the blocks that belong to the footprints of all the tasks that started before the consumer or only of the ones that started and finished before the start of the consumer. In the example shown in Figure 2.1, Task 7 may have or may have not accessed block b before the start of Task 8. In the former case, the right choice would be to include Task 7, because excluding it would result in incomplete block distance, but in the latter case the right choice would be to exclude it, because including it would result in a larger block distance than the right one. Since no choice is better than the other we chose to include the tasks that have only started and not have finished before the start of the consumer.

Figure 2.2 shows an example of a block’s access history. We model time in the X axis of the figure, increasing to the right. This means that T1 was the first to access the block and it was followed by T4, T8, T22 and T46. T1 was a writer and the task that initialized the block. The second was T4 which was both reader writer and modified the block. After that, T8, T22 and T46 read the block in that order. Given this access history we can find four producer-consumer pairs for that particular block as shown in 2.2a, 2.2b, 2.2c and 2.2d. In 2.2a, T4 is the first
reader, therefore it is the first consumer. Since there are no readers between T_1 and T_4, the only choice for the producer is the closest writer, therefore the first pair is (T_1, T_4). In 2.2b, T_8 is the consumer and the producer is again the closest writer, which in this case is T_4. In the cases of 2.2c and 2.2d we can find multiple possible producers. This is because T_4 was the last task that updated the block and all the readers that followed kept copies of the updated data close to the core they ran on. Therefore there can be multiple possible locations for T_46 to fetch the block from. In 2.2c, T_22 is the consumer and since the closest writer is T_4 and there is a reader between them, there are two possible producers, which are T_4 and T_8. Similarly, in 2.2d, T_46 is the consumer and the closest writer is T_4. Again there are readers between them, therefore the three possible producers are T_4, T_8 and T_22. In order to choose the best possible producer we must know two things: The first is the block distance between each one of the possible producers and the consumer and the second is the chip that they ran on. The second most important factor is the difference between a block distance and the capacity of the LLCache. If a block distance exceeds that capacity, we assume that the consumer fetches the block from the main memory, therefore the tasks that fall into that category are the least likely candidates for producers. Between the tasks which the block distances do not exceed the capacity of the LLCache, the tasks that have the highest possibility of being chosen as producers are the ones that ran on the same chip as the consumer, since the consumer tries to fetch the data from the nearest location. If none or more than one tasks have run on the same chip as the consumer, the best candidate is the one with the smallest block distance from the consumer. The same happens if all tasks have block distances that exceed the capacity of the LLCache. If, for example, in 2.2d, T_4, T_22 and T_46 ran on chip 0 and T_8 ran on chip 1 and the block distance of T_4 to T_46 exceeded the capacity of the LLCache, then T_22 and T_8 have higher possibility of being producers than T_4 and between them, T_22 has higher possibility since it ran on the same core as T_46.
Chapter 3

Implementation

Our goal is to create a way for profiling task-parallel applications implemented using BDDT and also for profiling scheduling policies of BDDT in terms of locality and reuse distance. In order to achieve that, we define the metric block distance. Block distance refers to a specific producer-consumer pair and a specific block. It is the amount of blocks that passed from LLCache of the chip that executed the producer of a block before the consumer, which could be executed in any chip, touched the block the LLCache of a chip. For example, let’s assume that tasks A, B, C were executed on chip 0 in that order and task D was executed on chip 1 chronologically after C. If task A was the producer of block b and task D was the consumer of block b, then the block distance of the producer-consumer pair A-D for block b is equal to the footprints of tasks B and C in blocks.

We chose to use this metric because it directly represents the reuse distance of a block. It refers to every one of the blocks that are part of the whole application’s footprint and, since it is possible for every block to be accessed multiple times during the execution, for every block we may calculate multiple block distances, one for every producer-consumer pair that accessed that block. Since our work is based on BDDT we need to describe the process of the dependence analysis and the scheduling policy of BDDT.

BDDT’s dependence analysis works by dividing a task’s arguments into blocks of the user-defined block size and, by comparing blocks and locating reader-writer dependencies, it creates the dependency graph of the application. A block can be of any size but the choice depends on the application and the programmer’s experience. Typical block sizes range from 512B to 2048KB. Smaller block sizes can lead to more precise dependence analysis, but add more overhead to both calculations and memory usage, because they produce a larger amount of metadata and comparisons. Larger block sizes produce less metadata, therefore they add less overhead but, since the comparisons are done between larger blocks, the dependence analysis may find some false positives among the dependencies and therefore lead to a less precise analysis. A master thread is responsible for initializing the runtime system, for creating tasks and inserting them into the dependency graph. Every
task that has its dependencies resolved can be scheduled to be executed by a worker. Each worker has a ready queue which contains tasks, and during its lifetime, the worker executes those tasks, releases other tasks from the dependency graph by resolving the dependencies that existed because of the executed tasks and enqueues every one of the released tasks into its queue or another worker’s queue, according to the scheduling policy that is applied during the execution. The master thread has its own ready queue, in which it enters every task that it creates and from which all the workers start getting their tasks. The master may postpone the creation of tasks and behave as a worker if a maximum window of tasks are pending to be executed and the rest of the workers do not execute them as fast as they are created. BDDT preallocates the whole memory that an application will need during its execution. The preallocated memory region is called grid [2]. BDDT also keeps a metadata element called Task_element [2] for every task instance that runs during the execution of an application. Every Task_element contains information about the state of a task during its lifetime, that is mostly needed for the dependence analysis part of BDDT. Part of this state is information about every argument of the task. This includes the address and size of the argument and a flag that specifies if the argument is an input argument, which means that the task behaves as a reader for it, an output argument, which means that the task behaves as a writer for it, or an inout argument, which includes both of the above cases.

In order to calculate the block distances for every block of the application footprint, we need to keep an online trace of the above information and then perform an offline analysis of that trace. We do that using the following method: Every time that a BDDT worker is about to execute a task, we keep a copy of its Task_element and store it in an array of Task_elements. At the end of the application’s execution, this array contains the task descriptors of all the tasks that ran during the execution. We use a lock every time a worker enters a task into the array and, since the worker starts executing it right after entering it to the array, we assume that the array holds the global start order of every task. The reason we keep copies of the Task_elements and not just pointers to them is because BDDT recycles the memory of the data structures that the Task_elements use, therefore at the end of the execution the Task_elements may not contain information about all of the tasks. The memory copies add a small overhead on the total execution time but not on the total task time. At the end of the execution we store the array of task descriptors to a binary file.

We created a profiling tool called lprof in order to make the offline analysis of the online tracing. lprof reads the binary file that contains the trace collected during the execution and proceeds to the following steps:

3.1 Create Array of task descriptors

By knowing the starting address of the grid, its size and the block size, we can divide the grid into blocks and express every memory region as a pair of indexes
that point to the first and the last of the set of blocks inside the grid that the memory region affects. We scan the array of task descriptors and, for every task, we create a structure that contains the index pairs for all the memory regions that the task touches during its execution. We store all the structures in an array.

3.2 Calculate all block distances

The second step is to find all the block distances between the tasks. We scan the array of task descriptors and, for every pair of tasks \((T_i, T_{i+k})\), we calculate the number of blocks that were touched by the tasks that ran at the same chip as the first task of the pair and are between \((T_{i+1}, \ldots, T_{i+k-1})\). This procedure may calculate some redundant block distances but it is multi-threaded and saves extra calculations from the next steps.

3.3 Create block usage history

The third step is to create a history of usage for every block of the grid. We create a list for every block of the grid and, by scanning the array of task descriptors, we find which tasks have touched the block and add them to the list in chronological order, by specifying also if the task was reader, writer or both.

3.4 Find all producer-consumer pairs

Next, for every block, we scan its history created in 3.3 and create a set containing the block distances of all the producer-consumer pairs that we find in the block’s history. We use the following method to create that set: We start from the last element in the block’s history which represents the last task that touched the block. If this task is a reader we mark it as a consumer and start scanning all the preceding tasks until we find the chronologically closest writer of the block. If there is no other task between the writer and the reader, we mark the writer as the producer and we add their distance (which was calculated in 3.2) to the set of block distances. In case that there are other tasks between the writer and the reader, it means that they are also readers that have valid copies of the block and it is possible for the consumer to have fetched the block from them instead of the writer. In order to determine which one of them was the most probable choice of the consumer, we have the following criteria:

1. If candidate A has block distance less than the LLCache capacity in blocks and candidate B does not, then candidate B is the most probable choice.

2. In case both block distances are less than the LLCache capacity, if candidate A ran on the same chip as the consumer and candidate B on the remote, then candidate A is the most probable choice.
3. In all other cases the most probable choice is the candidate with the smallest block distance from the consumer. Each block distance in the set of block distances contains also a remote flag that specifies if the producer-consumer pair was on the same or different chip. This flag is determined by two different cases. The first is if the block distance is less than the LLCache capacity, which means that remote is true if the producer and the consumer ran on different chips. In the case of the block distance being greater or equal than the LLCache capacity, we assume that the block is no longer in any cache, therefore remote is true if the producer had to fetch the data from a remote NUMA partition. In order to know in which NUMA partition each block was allocated, we have to know which task was the first to touch the memory page in which the block belongs. By dividing the memory grid into pages we can find the relationship between a block and a page, therefore we can find the pages that a block affects. By scanning the array of task descriptors during 3.1 we also keep extra metadata which contain information about the first task that touched each memory page that is part of the grid.

Using the above criteria we can construct a graph that looks like figure 3.1, which represents the rate that all the possible block distances were found in the results. On x-axis we have all the block distances from 0 to the maximum distance that was recorded in a run. On y-axis we have number of producer-consumer pairs. For every distance we measure the number of producer-consumer pairs that recorded that particular distance and we create a Cumulative Distribution Function (CDF) for every scheduler. The positive values of y-axis represent the number of producer-consumer pairs that ran on the same chip and the negative the ones that run on different chips. A vertical line is drawn at the spot that represents the LLCache capacity in blocks. By studying this graph, we can observe that there are four areas in the graph which affect the performance of the application differently:

1. **local_on_chip**: This category is represented in the area of positive y-axis
and before the LLCache capacity borderline. If a producer-consumer pair falls into this category, it means that they both ran on the same chip and their block distance was less than the LLCache capacity, which, according to our assumption, means that the consumer found the block on the LLCache. This leads to the fastest access times of all the other categories and it makes it the most desirable category.

2. remote_on_chip: This category is represented in the area of negative y-axis and before the LLCache capacity borderline. A producer-consumer pair that falls into this category ran on different chips and their block distance was less than the LLCache capacity. This means that the consumer found the block in the LLCache of the remote chip. This leads to worse results than the first category since the delay of the inter-chip interconnection is added to the total access time.

3. local_off_chip: This category is represented in the area of positive y-axis and after the LLCache capacity borderline. The producer-consumer pair ran on the same chip and their block distance was greater than the LLCache capacity, which, according to our assumption, means that the consumer did not find the block on the LLCache but on the main memory. This leads to slower access times than the first category due to the delay added by accessing the main memory.

4. remote_off_chip: This category is represented in the area of negative y-axis and after the LLCache capacity borderline. The producer-consumer pair ran on different chips and their block distance was greater than the LLCache capacity. This means that the access times are the slowest of all the categories because the consumer will fetch the block from the remote NUMA partition, therefore both inter-chip interconnection delay and main memory access delay will be added to the access time.

An example of a producer-consumer distribution is shown in 3.2 which is a figure of the percentage of the producer-consumer pairs that fall into the above four categories.

Regarding categories remote_on_chip and local_off_chip, there is the question of which one of the two adds more delay to the access times, namely, whether accessing a remote cache is faster than accessing the local main memory or not. The answer depends on the architecture of the system and therefore, by providing the division in categories, a programmer can judge the difference in importance of these two categories according to the system used in their experiments.

Each of the steps 3.1, 3.2, 3.3 and 3.4 is multi-threaded, and there are divided by barriers. Also, we use some architecture-dependent constants in our calculations. These constants are the LLCache capacity and the number of chips, and hence the number of NUMA partitions and the number of cores for every
chip. These constants must be modified if someone wants to use lprof on different architectures but they affect only the offline analysis. The online tracing is not architecture-dependent.

We believe that this combination of online tracing and offline analysis that calculate block distances is superior to counting cache misses, because it is more portable and it provides more information. For example, when counting cache misses it is difficult to distinguish which of the misses are the unavoidable ones or which are effect of the cache associativity. Furthermore, the block distance metric can help us to easily find an optimal cache size for a combination of an application and a scheduling policy, whereas it is more difficult to extract that information by counting cache misses.

Figure 3.2: Example of an application’s producer-consumer distribution.
Chapter 4

Evaluation

As explained in chapter 2, every BDDT worker has a ready queue which contains tasks that are ready for execution. Every time a worker executes a task, it also resolves its dependencies, releases the dependent from its tasks and inserts them into a ready queue. The worker chooses in which ready queue will it enter the task according to the scheduling policy that is followed during the execution. The placement of the tasks differs between the policies, therefore we expect it to affect performance due to the locality factor. Furthermore, we expect it to affect the block distances between producer-consumer pairs and the placement of the pairs to the processor chips. We evaluate six different scheduling policies using lprof in order to compare them in terms of locality-awareness. The schedulers we evaluate are the following:

1. Default: The master thread creates tasks and inserts them into its queue. Then, the worker threads remove tasks from the master’s queue and insert them into their queues in order to start executing them. If a worker finds its queue and the master’s queue empty it can steal work from other workers. When a worker finishes executing a task and releases a dependent task, it inserts it into its own queue [2].

2. Round-Robin: A global scheduling order is kept and every worker that releases a dependent task inserts it into the queue of the worker pointed by the global scheduling order.

3. Random: A worker that releases a dependent task inserts it into the queue of a random worker.

4. Default without work stealing: The default scheduling policy of BDDT without the work stealing part. This means that a worker will try to execute tasks only if they are in its own queue or the master’s queue.

5. Round-Robin without work stealing: Same as Round-Robin but without allowing a worker to steal work from other workers.
6. Random without work stealing: Same as Random but without allowing a worker to steal work from other workers.

In order to evaluate the schedulers, we ran a number of benchmarks using various input sizes on a machine with two 4-core processors of NUMA architecture, with each processor consisting of 256KB of L2 cache and 8MB of L3 cache. The system’s total main memory was 12GB divided in two NUMA partitions. We ran each variation ten times and we use five graphs to represent the results:

1. The first graph represents the rate that all the possible block distances were found in the results and it is explained in detail in 3.4. Every point in the graph depends on multiple values from the results, therefore we use the results from just one run, unlike the rest of them, where the results are calculated from the average of ten runs.

2. The second graph is a zoomed-in version of the first graph that shows the part before the vertical line which represents the LLCache capacity. We include this graph because it is clearer to see the start of every CDF which is the most important factor of our results since it represents the producer-consumer pairs that have block distance equal to zero and, for every benchmark, they represent the highest number of the pairs.

3. The third graph shows the percentage of the producer-consumer pairs that fall into the categories local_on_chip, remote_on_chip, local_off_chip, remote_off_chip, which are explained in 3.4. The values needed for this graph are taken from the average of the results from ten runs.

4. The fourth graph shows the number of processor ticks that were spent on executing tasks by the BDDT workers. This is calculated by adding the ticks that every worker spent on executing tasks. Since this value is affected only by the calculations and the time of all the memory accesses inside the tasks and since the calculations and the amount of memory access remain the same in every run, the only factor that makes a difference between the schedulers is the amount of cache misses which affects the time spent in memory accesses. Therefore we can assume that a more locality-aware scheduler will show better results in this graph. The values needed for this graph are taken from the average of the results from ten runs.

5. The fifth graph shows the total time of the run for each scheduler. This includes task run time, the dependence analysis part and all of the rest functions performed by the master and the workers. The values needed for this graph are taken from the average of the results from ten runs.

The benchmarks we ran are the following:

**Jacobi:** A parallel implementation of the Jacobian method for solving systems of linear equations. It uses a 2-dimensional array with tiled layout. Each parallel
task in Jacobi processes a tile of the array with a kernel implementing a 5-point stencil computation. We used two different inputs. The first was a 4096×4096 matrix in tiles of 128×128 elements for 4 iterations. The BDDT block size was 1KB. The second input differed on the number of iterations, which were 16. From the results shown in Figures 4.1 and 4.2 we can make the following observations: In both the case of 4 iterations and the case of 16 iterations we see similar results. If we focus on the case of 16 iterations, we can see in Figure 4.2(c) that, for schedulers default without work stealing and random without work stealing, the 80% of the data have remained in the local LLCache. For the rest of the schedulers the number falls to 70-72%. We expect that, for the former schedulers, the task time spent on memory accesses will be less than in the case of the latter schedulers and, since the total number of calculations remains the same, we expect that the total task time will be affected only by the time spent on memory accesses. In Figure 4.2(d), we see that this is verified, because the total task ticks in the case of the former schedulers are less than the ones of the latter schedulers.

**Cholesky:** An implementation of Cholesky decomposition, which is a decomposition of a Hermitian, positive-definite matrix into the product of a lower triangular matrix and its conjugate transpose. It consists of four parallel phases that perform tiled operations, corresponding to a task per tile per phase. We used two different inputs. The first was a 4096×4096 double precision matrix in tiles of 128×128 elements with a BDDT block size of 1KB. The second was a 8192×8192 double precision matrix in tiles of 256×256 elements with a BDDT block size of 1KB. From the results shown in Figures 4.3 and 4.4 we can make the following observations: In Figure 4.3(c) we see that in the case of the smaller input, for the default scheduler without work stealing, about 82% of the data remain on the LLCache of the local chip. For the rest of the schedulers, the number falls to about 70%. The effect on the total time can be seen in Figure 4.3(d), where the default scheduler without work stealing has the smallest number of ticks and, generally, we can see that as the percentage of the local cache data is reduced throughout the schedulers, the number of total task ticks is getting larger. However, we also see that the differences between the schedulers in terms of the total task ticks are small. This shows that task time is mostly spent on computations, therefore improving the memory access time has a smaller effect on the total task time than in the case of Jacobi. This can be seen more clearly in Figures 4.4(c) and 4.4(d), where we can see that despite of the smaller number of data that have remained on the local cache (which is expected because of the larger input), the effect of the off cache data is even smaller, which shows that increasing the input size increases the amount of computations by a larger factor than the memory accesses.

In Figures 4.3(e) and 4.4(e) we see the total time of the run, including dependence analysis overheads and effects of parallelization. We see that the default scheduler without work stealing has the worst performance and the larger variation and is followed by the random scheduler without work stealing, whereas the other schedulers are more balanced and have better performance. This is expected because work stealing provides better load balancing and so does the round-robin scheduler.
without work stealing due to the way of how the round-robin fashion works.

**SPLASH-FFT**: An implementation of the fast Fourier transformation taken from the SPLASH-2 benchmark suite. It consists of five parallel loops that alternate in transposing the input array and performing 1-dimensional FFT on each row. Each task created in the FFT calculation loop operates on an entire row of the array, while transposition phases break the array into tiles and create a task to transpose a group of tiles. We used two different inputs. The first used 1M complex doubles, split into blocks of 32 rows at the transformation phase and $32 \times 32$ tiles at the transposition phase. The BDDT block size was 512B. The second input used 4M complex doubles, split into blocks of 32 rows at the transformation phase and $32 \times 32$ tiles at the transposition phase. The BDDT block size was also 512B. From the results shown in Figures 4.5 and 4.6 we can make the following observations: We see similar results in the input case of 1M complex doubles and the case of 4M complex doubles. If we focus on the case of the larger input, we can see in Figure 4.6(c) that, for scheduler random without work stealing, around 87% of the data have remained in the local LLCache. For scheduler default without work stealing the number falls to around 83% and they are followed by scheduler round-robin without work-stealing with 78%, random with work stealing with 75%, round-robin with work-stealing with 72% and default with work stealing with 70%. Like the first two benchmarks, we see that the total task time, shown in Figure 4.6(d), falls as the percentage of the on-local-cache data gets higher. The effect of the data distribution to the total task time is not as high as in the case of Jacobi but is higher than in the case of Cholesky.

**HPL (High Performance Linpack)**: A software package that solves a (random) dense linear system in double precision arithmetic. It is part of the High-Performance Linpack Benchmark. We used an input matrix of size $16 \times 16$ blocks, with block size equal to $32 \times 32$ elements. The BDDT block size was 128B. From the results shown in Figure 4.7 we can make the following observations: In Figure 4.7(c) we see that all of the data remained in the local and the remote LLCaches. Therefore, the effect we see in Figure 4.7(d) is that the total task times are almost the same for every scheduler and the differences are determined by the ratio of local-to-remote accesses.
Figure 4.1: Results for Jacobi with 4 repetitions
CHAPTER 4. EVALUATION

(a) Distribution of all distances
(b) Distribution of distances through cache
(c) Breakdown into distance categories
(d) Total processor ticks
(e) Total execution time

Figure 4.2: Results for Jacobi with 16 repetitions
Figure 4.3: Results for Cholesky-128
CHAPTER 4. EVALUATION

(a) Distribution of all distances

(b) Distribution of distances through cache

(c) Breakdown into distance categories

(d) Total processor ticks

(e) Total execution time

Figure 4.4: Results for Cholesky-256
(a) Distribution of all distances

(b) Distribution of distances through cache

(c) Breakdown into distance categories

(d) Total processor ticks

(e) Total execution time for

Figure 4.5: Results for FFT-Splash-20
CHAPTER 4. EVALUATION

(a) Distribution of all distances
(b) Distribution of distances through cache
(c) Breakdown into distance categories
(d) Total processor ticks
(e) Total execution time

Figure 4.6: Results for FFT-Splash-22
Figure 4.7: Results for HPL
Chapter 5

Related Work

C. Fang et al. [3] have presented a method for reuse distance profiling, based on path-based analysis. They define as path, the program execution trace between two subsequent accesses of the same memory location. Using this method, they can create execution-path histories and, by relating them to specific locality patterns, they can determine the appearance of a particular locality pattern in a memory instruction.

Q. Niu et al. [4] have presented PARDA, which is a parallel reuse distance analysis algorithm, which performs online reuse distance analysis of reference traces. Their goal is to reduce the overhead of the online analysis by parallelizing the process. Their results show that they improve performance of the analysis by 13 to 50 times for a range of applications.

X. Xiang et al. [5] have presented a Higher Order Theory of Locality (HOTL), which allows someone to convert between five locality metrics. Specifically, they considered five locality metrics, which are: Footprint, Inter-miss time, Volume fill time, Miss ratio and Reuse distance. They divided these five metrics according to a hierarchy into three categories: 1st order, which includes the reuse distance metric, 2nd order, which includes the miss ratio and the inter-miss time metrics and 3rd order, which includes the footprint and the volume fill time metrics. One can derive a metric that belongs in a higher order by taking the sum of the metrics when moving up the hierarchy and a metric that belongs in a lower order by taking the difference of the metrics when moving down.

C. Cascaval et al. [6] have presented a machine independent method of estimating the number of cache misses at compile time, by using an algorithm which computes stack histograms. They use the stack histograms that the algorithm produces for specific applications to evaluate locality optimizations at compile time.
Chapter 6

Conclusions and Future Work

In this work we presented lprof, a tool which uses an architecture-independent method to perform offline analysis on task-parallel applications. We used lprof to profile applications developed using task-based programming model BDDT and also to compare different task scheduling methods applied to these applications. We profiled 4 applications using 6 different scheduling methods and we saw a relation between the distribution of the application’s data and the total time that was spent on tasks. We saw that this tool provides valuable insight about locality-awareness in task-based programming models and can be easily configured for different NUMA architectures due to the architecture-independence of the basic method that the tool uses. Future work may include a more extensive profiling of various applications and scheduling policies, including more locality-aware scheduling methods.
CHAPTER 6. CONCLUSIONS AND FUTURE WORK
Bibliography


