2011-09-13

The Pulled-Macro-Dataflow Model: An Execution Model for Multicore Shared-Memory Computers

Daniel Joseph Richins
Brigham Young University - Provo

The Pulled-Macro-Dataflow Model: An Execution Model for Multicore Shared-Memory Computers

Daniel J. Richins

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

David A. Penry, Chair
James K. Archibald
Doran K. Wilde

Department of Electrical and Computer Engineering

Brigham Young University

December 2011

ABSTRACT

The Pulled-Macro-Dataflow Model: An Execution Model
for Multicore Shared-Memory Computers

Daniel J. Richins
Department of Electrical and Computer Engineering, BYU
Master of Science

The macro-dataflow model of execution has been used in scheduling heuristics for directed acyclic graphs. Since this model was developed for the scheduling of parallel applications on distributed computing systems, it is inadequate when applied to the multicore shared-memory computers prevalent in the market today.

The pulled-macro-dataflow model is put forth as an alternative to the macro-dataflow model, having been designed specifically to accurately describe the memory bandwidth limitations and request-driven nature of communications characteristic of today's machines. The performance of the common scheduling heuristics DSC and CASS-II are evaluated under the pulled-macro-dataflow model and it is shown that their poor performance motivates the development of a new scheduling heuristic. The Concurrent Tournament Reducer (ConTouR) is developed as a scheduling heuristic which operates well with the pulled-macro-dataflow model.

ConTouR is compared to the existing heuristics Load Balancing and Communication Minimization in scheduling two programs. For both programs, the other reducers are shown to outperform ConTouR.

Keywords: pulled-macro-dataflow model, macro-dataflow model, scheduling, reduction heuristic, ConTouR

ACKNOWLEDGMENTS

The long, arduous process of writing this thesis has had its ups and its down, its joys and its disappointments, its celebrations and its tears. Not infrequent were the times that I thought completing it would be impossible. Nevertheless, I was encouraged to continue working by a few individuals. First of all I recognize Dr. David A. Penry, my advisor, who provided me with direction and a goal and helped me to work through problems. I wish to thank Dr. David K. Anthony, who always believed in me and would not have stood for failure. I express my deepest gratitude to my parents, Russell and Diane, who were always supportive and never doubted that I would be able to finish. Finally, I thank Josephine, Phillip, and Lillian for their tremendous emotional support throughout the entire process of doing research and writing this thesis. Never did they speak a word of criticism or doubt and frequent were their expressions of caring and concern. Without the support of all these generous and caring individuals I would never have had the drive to push through the hard times and reach the reward waiting at the end.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1. INTRODUCTION

Through decades of experience, consumers and professionals alike have come to expect faster and more powerful computers on a regular basis. In the past, this additional speed was provided by designing ever-more complex single-core processors that strove to maximally utilize instruction-level parallelism. The exponential growth in speed was also fueled by exponential increases in clock frequency. Ultimately, however, this approach met its demise as the power wall presented an apparently insurmountable barrier. Instead, it was sidestepped: microarchitects turned to other sources to provide better processors that would appeal to consumers.

The solution that microarchitects found is multicore processors. Individual cores no longer have to eke out tiny amounts of additional instruction-level parallelism, and clock frequency no longer has to increase exponentially. Instead, by placing two or more cores in a single processor, the computational speed of that processor has effectively been multiplied by some integer factor.

Unfortunately, this solution is not as simple as it looks. None of the programs that were written for single-core processors can automatically take advantage of the additional cores. Instead, if multicore processors consist of simpler individual cores than their single-core counterparts, the performance of old programs actually suffers. To keep up with consumer demand, programs must be rewritten to be made aware of and utilize all available cores. Ultimately, consumers are not provided with the automatic increase in performance they expect; they must rely on software developers to parallelize programs and realize the potential of their new hardware.

To make matters worse, only a small subset of programs is easily parallelized. For the remaining programs, parallelization is a painstakingly involved and error-prone process. It relies on fine-grained parallelism with intricate synchronization and interaction among threads. Only the most skilled programmers can be relied upon to produce high quality multi-threaded programs. The answer to this problem is not to expect programmers to improve but to provide programmers with better tools to ease the process of multi-threaded programming. Any such tool must effec-

tively find, expose, and exploit a program's fine-grained parallelism [1]. Based on cues from the programmer, the tool would manage the details of parallelization, leaving the programmer to focus on designing the program rather than fixing bugs.

Finding parallelism will consist of dividing programs into small tasks, only hundreds or even tens of instructions long. The intricate dependences that prevented the program from being divided into coarse-grained tasks become palatable at this level. The simple tasks produce small pieces of data to be communicated to other tasks. However, with such small tasks and so many communications, the latency of communication comes to constitute an enormous percentage of the overall program cost. Modern multicore machines communicate between cores using shared memory. It is likely that future multicore machines will continue to do so. Communication via shared memory necessarily involves a cache miss, which leads to substantial latency for every communication between cores. Hence, in scheduling tasks to execute on physical cores, it is critical to account for these long latencies.

One proposed method for representing and thereby exposing the parallelism in a program is by encoding the program as a directed acyclic graph (DAG). In a DAG, also called a task graph, the nodes represent tasks or groups of instructions to be executed. Tasks also carry a weight indicating the cost of executing the task. A higher cost indicates that a task takes longer to execute. Connecting tasks are edges, which represent communications between tasks. Like tasks, edges are assigned weights. Higher weights indicate that there is more data to be communicated from one task to another, requiring greater time. Edges also enforce an ordering between tasks. A task that consumes data cannot begin execution until after the task producing its data has completed. Ultimately, edges impose a partial ordering on the graph: no task may execute until all of its ancestors have finished executing. Thus, the program's potential for parallel execution is encoded within the structure of the task graph: any two tasks without some path connecting them may be executed simultaneously.

For the parallelism of a task graph to be effectively exploited, the runtime system will need a high-quality scheduler to map tasks to processing cores. Complicating the matter, however, is the behavior of communications between tasks. It is expensive to communicate data from one core to another; mapping two tasks to the same core, however, eliminates that communication cost. This comes with a price though, as tasks on a single core cannot execute in parallel. The scheduler,

therefore, must weigh the advantage of executing more tasks in parallel against the prospect of zeroing communication costs.

1.1 Motivation

Many schedulers have already been published for scheduling task graphs. These will be explored in more detail in Chapter 2. Nearly all existing schedulers are based on the macro-dataflow model of execution [2]. In this model, a task waits for its incoming communications to arrive, executes its instructions, and then communicates all outgoing data. Communications from a task are begun as soon as the task finishes execution and arbitrarily many communications may proceed simultaneously. A communication is assumed to have finished after waiting an amount of time equal to the weight of the edge. As with outgoing edges, incoming edges are also allowed to proceed in parallel. Scheduling heuristics work to schedule graphs such that the final task finishes as soon as possible.

The macro-dataflow model was developed for distributed systems. When applied to shared-memory multicore systems, however, the model's assumptions fall apart:

1. Communications are not actually initiated by the task producing the data. Instead, a communication occurs as a result of the receiving task requesting the data. This happens through a shared memory location. The receiving task requests the memory contents at that address. The data will not exist in the cache of the core executing the task, since the applicable cache line (if it was present at all) will have been invalidated by the core executing the sending task. The requesting core must instead proceed to some higher level of the memory hierarchy until it finds a valid version of the requested data. Hence communications are initiated by the receiving task and occur as a cache miss.

2. Communication cannot, in reality, proceed with infinite parallelism. It is limited by the size of the processor's load buffer. Thus, communications are, to some degree, sequentially ordered.

Because the macro-dataflow model so poorly represents the behavior of modern desktop computers, the scheduling heuristics developed under the model cannot be expected to perform well for programs that run on modern computers. Instead, a new model must replace this model

and either new heuristics must be developed under the new model or existing heuristics must be adapted for the new model.

1.2 Contributions

1.2.1 Pulled-Macro-Dataflow Model

This thesis presents the pulled-macro-dataflow model of execution. This model was designed to address the shortcomings of the macro-dataflow model and accurately represents shared-memory multicore systems. The significant features of the model are

1. Communications are initiated by the receiving task. Since all inter-processor communications must occur by means of a cache miss, the receiving task effectively "pulls" the data from the sending task. As a result, communication does not begin when sending tasks finish communication; rather, communications begin only once the receiving task begins execution, i.e. once all preceding tasks have finished.

2. Infinite memory parallelism is not assumed. Instead, communications may only proceed with as much parallelism as is actually present in the system. In the worst case, all communication leading into a task may have to be serialized.

The accuracy with which the pulled-macro-dataflow model predicts program performance will be examined in Chapter 4.

1.2.2 ConTouR

Because nearly all scheduling heuristics developed thus far use the macro-dataflow model, they cannot be expected to do well when evaluated using the pulled-macro-dataflow model. This thesis therefore presents the Concurrent Tournament Reducer (ConTouR), which was designed specifically for the new model. This heuristic conforms to the two-pass approach to scheduling [3]. In the first pass of this approach, a clustering heuristic combines tasks into an unbounded number of clusters, or task groups, in an attempt to minimize schedule length. In the second pass, a reducer assigns clusters to execute on physical processing cores. ConTouR, as the name implies, operates

on the second pass of the two-pass approach. It is unique among reducers in being able to utilize multiple threads for scheduling. Designers of other reducers have been content with using a single thread for scheduling.

ConTouR will be shown to operate in $O(v(v+e))$ time, where v is the number of tasks and e is the number of edges in the graph. This complexity is reducible to $O(v+e)$ given the right choice of clustering heuristic.

1.2.3 Comparison of Reducers

To understand how well ConTouR performs, this thesis will also present a comparison of ConTouR against two other reducers. This comparison examines both the theoretical performance of each reducer (comparing the calculated schedule lengths) and the measured program runtimes for programs scheduled by each reducer. No previous work has attempted to measure reducer performance by measuring actual runtimes.

1.3 Outline

The remainder of this thesis is organized as follows. Chapter 2 presents work related to this thesis. It introduces two approaches to scheduling and describes the roles of clustering and reduction. Existing heuristics are presented and briefly explained.

In Chapter 3, the macro-dataflow model of execution is described. Its applicability to shared-memory multicore processor systems is explored. In response to the limitations that the model has, the pulled-macro-dataflow model of execution is introduced. The quality of existing heuristics is explored in the context of the new model. Finally, a new reduction heuristic, the Concurrent Tournament Reducer (ConTouR) is introduced, described, and analyzed.

In Chapter 4, the quality of each of three reduction heuristics is evaluated. First, a theoretical analysis of each is given and second, their performance on actual programs is measured. ConTouR is shown to perform the best in the theoretical analysis but is seen to perform poorly when scheduling actual programs. The validity of the pulled-macro-dataflow model is also tested by comparing calculated schedule lengths to program running times. The model is found to be a

poor predictor of running times in inter-reducer comparisons; it does manage, though, to roughly predict variations in running times for each heuristic individually.

Chapter 5 concludes this thesis with a summary of what was learned as well as suggestions for future work to improve and enhance the pulled-macro-dataflow model.

CHAPTER 2. RELATED WORK

This chapter summarizes existing approaches to scheduling. It discusses one-pass and two-pass scheduling and outlines existing heuristics used for scheduling.

2.1 Preliminaries

In task graphs used to represent applications, nodes represent tasks (small pieces of computation) and edges represent dependences between tasks. The weights of nodes denote the computational costs of tasks. Edges indicate that some data must be communicated from one task to another; the weight of an edge denotes the cost (the latency) of communication.

The goal of scheduling is to assign tasks to processor cores in such a way as to minimize the makespan. Makespan refers to latest finishing time of any task in a given task graph. Calculation of makespan depends on the model of execution, which is discussed in Sections 3.1 and 3.2. When two tasks are scheduled to execute on the same core, the communication edge between them is set to zero. This reflects the fact that no data need be communicated from one core to another.

Traditionally there have been two approaches to scheduling: static and dynamic [4]. Static scheduling takes place during compilation. At first glance, this seems an attractive approach, as it avoids having to schedule at runtime, which delays the actual execution of a program. However, static scheduling requires that the entire task graph be known during compilation—a requirement that often does not hold. It further requires that the target system be known a priori, so that a scheduler will know how many threads to use. This prevents a statically scheduled program from being run on a variety of systems.

Dynamic scheduling delays scheduling as long as possible. A dynamic scheduler does not create the task graph in its entirety before running a program. Rather, tasks are created only as necessary, once their inputs are all known. This provides maximum flexibility and can even sched-

ule a graph whose structure relies on the specifics of program execution. However, the overhead of creating and scheduling each task during program execution can be significant.

In practice, many approaches to scheduling reside in some middle ground between static and dynamic.

2.1.1 Inspector-Executor Model

One compromise between the static and dynamic approaches to scheduling is referred to as *Inspector-Executor* [5]. The Inspector-Executor model delays scheduling of task graphs until runtime but before execution. A scheduler operating under this model does this by splitting scheduling into two distinct phases: inspection and execution.

Inspection is the first thing to occur at runtime. It takes input data to the program and uses those data to construct the task graph in its entirety. As soon as the entire graph has been created, scheduling can proceed. In the execution phase, the program (represented by the task graph) is executed according to the schedule and core assignments determined during inspection.

Some applications are especially well-suited for use in Inspector-Executor scheduling, such as sparse linear algebra and molecular dynamics. In general, any application whose structure cannot be known statically but can be determined from just its input data (without having to rely on the results of execution) is a good candidate.

Naturally, delaying program scheduling until runtime and execution until after inspection will incur a penalty. This penalty can be mitigated, however, if the optimized schedule determined during inspection can be reused many times before program termination.

Much work has gone into developing and expanding the Inspector-Executor approach, and the work in this thesis builds on one implementation of the approach as presented in [6] and [7]. This implementation, called *ADOPAR* (Adaptive Online Parallelization), is a runtime strategy available to *CIRA* (Compiler-Inserted Runtime Adaption) which uses a custom internal representation to store and represent programs. It builds upon the Inspector-Executor approach to instantiate a task graph, schedule the graph and map it to processing cores for execution, and then execute the program according to that schedule.

2.2 One-Pass and Two-Pass Scheduling

Scheduling task graphs for execution on multiple processor cores is widely known to be NP-hard (see, for example, [2]) except for certain very restricted types of graphs (as in [8]). The majority of work in scheduling has, therefore, focused on the development of heuristics. Within those heuristics, there are two general approaches: one-pass and two-pass scheduling. As their names imply, the two approaches manage to schedule the task graph onto cores in either one or two passes.

There are some common features of scheduling heuristics which are worth mentioning here. Most scheduling heuristics are modified list schedulers. They operate by maintaining a priority-ordered list of tasks. The list is initialized to only contain entry (root) tasks. As tasks are scheduled, the list is actively updated to contain those tasks all of whose predecessors have been scheduled. Tasks within this *ready list* are termed *ready tasks*. Tasks are scheduled one at a time, from highest priority to lowest priority. Depending on what priority function is used, each heuristic tends to create slightly different styles of schedules. Some approaches used for determining priority are *t-level*, *b-level* [9], and *ALAP* [10].

T-Level The t-level of a task is the earliest possible start time of a task in a graph. Put another way, the t-level is the longest single path from an entry task of the task graph to the task in question. When t-level is used to determine priority, tasks are typically sorted in order of increasing t-level. The t-level of a task and its descendants may change as the task is scheduled. As the t-level defines the earliest time a task can possibly start, it is equivalently referred to as the as-soon-as-possible (ASAP) level.

B-Level Complementary to the t-level, a task's b-level is the longest path from that task to an exit task in a graph. If a graph is scheduled in some topological order, the b-levels of tasks being actively scheduled never change; the b-level of a task can only change after one or more of its descendants has been scheduled. Those heuristics which use b-level as the priority measure often schedule tasks in order of decreasing b-level.

ALAP An interesting variation to t-level priority (which assigns tasks to execute as soon as they possibly can) is as-late-as-possible (ALAP) priority. Tasks that do not lie along the critical path have some flexibility in when they execute. Instead of scheduling those tasks to execute immediately upon their predecessors finishing, ALAP schedulers place those tasks as late in the schedule as they can without increasing the overall makespan.

2.2.1 One-Pass Scheduling

Some one-pass scheduling heuristics are outlined below. These heuristics assign tasks directly to the cores on which they will ultimately execute. In describing computational complexity, the value v refers to the number of tasks in a graph, e refers to the number of edges, and p refers to the number of cores.

HLFET The Highest Levels First with Estimated Times (HLFET) [11] heuristic is a very simple list scheduler. It calculates the b-level of all tasks and initially populates the ready list with entry tasks. The b-level serves as the priority measure. Tasks are scheduled to the core which allows the earliest start time. As tasks become ready, they are added to the ready list. HLFET operates in $O(v + e)$ time.

ETF The Earliest Time First (ETF) [12] scheduler also uses a ready list of tasks. The highest priority task is that task which has the earliest start time on any core. Ties are broken arbitrarily. As each task is pulled from the front of the ready list, it is scheduled to the processor allowing its earliest start time, respecting communication costs and precedence constraints. ETF has a complexity of $O(pv^2)$.

DLS Dynamic Level Scheduling (DLS) [13] introduces a quantity known as the *dynamic level* for each task. It is defined as the difference between the static b-level (the b-level computed before any scheduling and not updated as scheduling proceeds) and the earliest start time. The earliest start time is slightly distinct from the t-level in that in addition to communication costs, it must account for tasks that have already been scheduled on each core. The dynamic level directly corresponds to

priority: those tasks with higher dynamic levels get higher scheduling priority. Tasks are scheduled to start as early as possible. DLS runs in $O(pv^3)$ time.

MCP The Modified Critical Path (MCP) [10] scheduler relies on the ALAP start times of tasks. The ALAP time of each task is computed as the first step of MCP. A list is then compiled for each task, consisting of the ALAP time of the task to which the list belongs as well as the ALAP times of all its descendants. Tasks are then scheduled in decreasing order of ALAP times. In the case where a tie between tasks exists, the descendant ALAP lists belonging to each task are used as tie breakers, with higher ALAP values getting higher priority. Tasks are scheduled to the core that allows for the earliest start time. MCP has $O(v^2 \log(v))$ complexity.

2.2.2 Two-Pass Scheduling

In two-pass scheduling (proposed by Sarker and Hennesy in [3]), the two passes of scheduling are called clustering and reduction.[1] A cluster is a group of tasks which have all been assigned to execute on the same core. Clusters are distinct from cores, though, in that they are not associated with any physical cores and there may be far more clusters than there are cores. It is only during the second pass, reduction, that the clusters are assigned to execute on physical cores. It is also the job of the reduction pass to merge clusters until the number of clusters is no greater than the number of cores.

Clustering

The goal of clustering is to combine tasks into clusters in such a way as to minimize the schedule makespan, regardless of how many clusters are created. There are two competing sources of speedup for a task graph, and a clusterer must balance the two. The first is to zero communication edges by scheduling multiple tasks to execute in the same cluster. The other source of speedup is to schedule tasks to separate clusters to maximize parallel execution. The former potentially eliminates parallelism, as all tasks within a cluster must execute sequentially. Conversely, leaving tasks in separate clusters leads to high communication costs. Clusterers ultimately create

[1] These names have been modified from those given in the original paper.

as many clusters as they deem necessary to minimize makespan. Many clustering heuristics have been developed. Outlined here are some of the more prominent.

DSC Dominant Sequence Clustering (DSC) [14] works to minimize the *dominant sequence*, which is the critical path of the (partially) scheduled task graph. The priority of each task is calculated as the t-level plus the b-level. Tasks are inserted into the ready list when all their predecessors have been scheduled. However, a task may also be considered for scheduling if it is believed to be in the dominant sequence. Hence, two lists are maintained: the ready list and the partial ready list, which contains tasks with only some of their predecessors having been scheduled. The task with the highest priority from either list is chosen for scheduling; it is assigned to the cluster which minimizes its t-level. If the t-level increases no matter which predecessor's cluster it is assigned to, then the task is assigned to a new cluster. This continues until all tasks have been scheduled. DSC also contains a special case to optimally handle certain subsets of graphs: in scheduling a task u, all the predecessors of u which only have u as a child are considered for inclusion in the same cluster as u. The predecessors meeting this constraint are ordered in decreasing order of t-level plus execution cost plus communication cost with u. All predecessors up to the turning point in the ordered list are then included in the same cluster as v_x to minimize the t-level of v_x. In this way, DSC is able to optimally cluster *join* task graphs. Finally, DSC is re-run on the graph with all edge directions reversed, allowing for optimal scheduling of *fork* graphs. DSC has a complexity of $O((v + e) \log(v))$.

CASS-II The Clustering And Scheduling System II (CASS-II) presented in [15] attempts to improve on DSC by maintaining high quality schedules while reducing the runtime complexity. CASS-II works from the exit tasks of a graph backward toward the entry tasks in order to avoid re-evaluating t-levels. The priority of its ready list is computed as the t-level plus the b-level. The ready task with the highest priority is considered for inclusion in the cluster of its dominant successor (the successor through which passes the longest path to an exit task, i.e. the successor which determines the b-level). The task is included in that cluster if doing so does not increase both the b-level of the task and the b-level of the cluster. The b-level of the cluster is defined as the b-level of the first task within the cluster. If this condition is not met, the task is placed into a

new cluster of its own. Like DSC, CASS-II also contains a special case to handle join task graphs optimally. It does this by allowing tasks whose children are all exit tasks to be re-considered for clustering. So long as subsuming those tasks into the cluster containing the exit task does not increase the b-level of the cluster or the task under consideration, tasks continue to be subsumed. The complexity of CASS-II is $O(e + v \log(v))$.

DCP Dynamic Critical Path (DCP) [16] is similar to DSC in that it recognizes that the critical path of the graph changes throughout the scheduling process. It calls this changing critical path the *dynamic critical path*. DCP relies on the ASAP and ALAP start times of each task. Any task whose ASAP and ALAP times are equal is on the dynamic critical path. Tasks may be considered for clustering before their predecessor tasks if they are on the dynamic critical path; however, dynamic critical path tasks are scheduled in order. DCP works to insert tasks into idle time slots within clusters; it does so ensuring that the slot is big enough (it may delay later tasks under some circumstances) and that no ordering constraints are violated. DCP uses a lookahead strategy for selecting which cluster is best for each task: it selects the cluster where the sum of a task's ASAP time and that task's most critical child's ASAP time is minimized. During this process it may choose to create a new cluster. DCP runs in $O(v^3)$ time.

Internalization Internalization [3] starts by placing each task in its own cluster. It maintains a table *DeltaCPL* which stores the decrease in critical path length that would be yielded by merging any two clusters. It proceeds to repeatedly merge the two best clusters and update DeltaCPL until no further merging would yield any benefit. Internalization has a complexity of $O(v^2(v + e))$.

Reduction

When clustering has finished, the total number of clusters may (and almost certainly will) exceed the number of physical cores. Should a program be allowed to execute in this state, each cluster would be assigned to a new thread and the program would rely upon operating system scheduling to ensure that each thread gets executed. This approach is seriously handicapped as the operating system knows nothing about the length of clusters or their intercommunications; it

cannot, therefore, intelligently schedule clusters to minimize makespan. Additionally, the overhead of repeatedly switching threads utterly destroys any speedup that was found through clustering.

Instead of relying on the operating system for scheduling clusters, a second pass of scheduling is used, called reduction. It is the job of a reduction heuristic (called a reducer) to intelligently assign clusters to actual cores.[2] To do so, it must assign multiple clusters to a single core. This will reduce the parallelism produced by clustering but retains the advantage present in clustering: clusters executing on the same core have their communication costs zeroed.

In [17], this second pass of two-pass scheduling is referred to as "cluster merging." I have chosen to use the term "reduction" instead as a more encompassing term. In Section 3.4 the ConTouR reduction heuristic will be introduced. Rather than simply merging clusters together, ConTouR first splits clusters apart as necessary and goes on to refine the results obtained through clustering. Hence, the ultimate goal of a reducer is not just to merge clusters; it is to build upon what was already done by the clusterer and finally guarantee that only one thread will be run on each processor core.

There are two reduction heuristics presented in [17]. They are outlined here. In general, however, there has been very little work dedicated to producing high quality reducers [18].

LB Load balancing (LB) is described in [17] and [19]. The total execution cost of each cluster is found, and all clusters are sorted in increasing order of execution cost. The cluster c_m with minimum execution cost is selected and merged with the minimum cost cluster among those clusters with which c_m shares a communication edge. The merged cluster's execution cost is reflected in the sorted list of clusters and the process is repeated until each cluster can be uniquely mapped to a core. LB works to give each core approximately the same load in execution cost but also recognizes that zeroing communication edges is a huge win for scheduling. It operates in $O(v \log(v)+e)$ time.

CM Communication Minimization (CM) [17] recognizes that there is much speedup to be gained by zeroing communication edges. It operates by calculating the total communication cost between each pair of clusters. This information is stored in a priority list, wherein cluster pairs with higher

[2] A reducer assigns to precisely as many threads as there are cores under the assumption that one thread will run on each core. Consequently, I will use the terms "thread" and "core" interchangeably.

communication costs are given higher priority. CM merges the two clusters with the highest priority and updates the list to reflect this change. It repeats this process until the number of clusters no longer exceeds the number of cores. The complexity of CM is $O(v + e\log(e))$. Neither LB nor CM maintains a ready list, as it may not be possible to order clusters in such a way that any cluster is guaranteed to be scheduled before any of its successors is scheduled.

Final Scheduling

Following clustering and reduction, the final ordering of tasks on cores must be decided [17]. Additionally, if clustering and reduction failed to complete their jobs, tasks must be definitively assigned to execute on specific cores. In CIRA, this is implemented using a simple list scheduler. It can be customized to use a variety of priority functions but defaults to using a queue and assigning all tasks a uniform priority. The list scheduler fills in any holes left by the clusterer and reducer by topologically sorting the tasks and assigning each task to the core where it can start execution first. If clustering ran but not reduction, all subsequent tasks in the same cluster as a task that has been scheduled to a core are automatically scheduled to that same core. Additionally, the list scheduler in CIRA actively looks for holes in the schedule where it can schedule tasks to execute, while still respecting dependences.

2.2.3 Previous Studies

With the multitude of available scheduling heuristics, researchers have published studies comparing the heuristics to determine how they perform when compared to each other directly. In [18], Kwok and Ahmad compare 6 one-pass heuristics and 5 clustering heuristics. No reduction heuristics are examined. Among one-pass heuristics, MCP is found to perform the best. It is also reported that of the clustering algorithms, DCP tends to produce the highest quality schedules; however, it also takes inordinately long to complete scheduling. DSC, which also monitors the dynamically changing critical path, is also found to perform well, but with much shorter scheduling times. It should be noted that the work in that paper was completed before the CASS-II clusterer was published. The authors do not compare the one-pass approach to the two-pass approach since they do not examine any reducers.

In [17], Liou and Palis compare one-pass and two-pass scheduling. They conclude that the two-pass method is superior to the one-pass. To evaluate reducers, the authors use the CASS-II clusterer and compare three reduction schemes: CM, LB, and random reduction. Unsurprisingly, random reduction performs very poorly. They demonstrate, however, that LB tends to produce significantly shorter schedules than CM.

In no case for either of these studies were the heuristics evaluated using actual program running times.

CHAPTER 3. EXECUTION MODELS AND CONTOUR

This chapter presents the existing macro-dataflow model of execution and discusses why it does not accurately represent shared-memory multicore systems. It then introduces the pulled-macro-dataflow model of execution and explains how it is ideally suited for these systems. Finally, the ConTouR reduction heuristic is presented, which has been designed to operate well under the new model.

3.1 Macro-Dataflow Model

The *macro-dataflow* model of execution (see [2]) attempts to model the behavior of executing tasks on cores and communicating data between cores. This model uses the acyclic task graphs described in Chapters 1 and 2. When a program begins execution, the only tasks which may proceed are the entry tasks—those with no predecessors. As soon as a task has finished execution, it immediately initiates communication of any data that are needed by subsequent tasks. These communications are all assumed to proceed simultaneously and each concludes individually, depending on the latency encoded by the weight of the edge representing the communication. New tasks are allowed to begin execution as soon as all incoming communication edges conclude. This pattern propagates through the graph until the entire program has executed.

Let the set of tasks in the graph be represented as V and the set of edges (u, v) from task u to task v be represented as E. Let the earliest time at which a task v can start execution be denoted by S_v and the earliest time at which task v may finish execution be denoted by F_v. Let the cost of executing task v be represented by τ_v and the cost of a communication edge $(u, v) \in E$ by $c_{(u,v)}$. The model may then be described mathematically as

$$
\begin{aligned}
S_v &= \begin{cases} \max\limits_{(u,v)\in E} \{F_u + c_{(u,v)}\} & v \text{ is not an entry task} \\ 0 & v \text{ is an entry task} \end{cases} \\
F_v &= S_v + \tau_v.
\end{aligned}
$$

The macro-dataflow model was designed for scheduling applications on distributed message-passing systems. In this context, it performs fairly well: communications may indeed be begun as soon as a task completes, and many communications may take place simultaneously. The model is not accurate, however, in that communications cannot actually proceed with infinite parallelism.

When applied to more common modern systems—shared-memory multicore processors—the assumptions of the model erode further:

1. Communication is not actually initiated by the sending task. While in a message-passing system this is true, in a shared-memory system communication occurs by writing a value to a particular location in first-level cache. For the receiving task to obtain that data (i.e. for the communication to complete), it must request the data from higher-level memory.

2. No real system has infinite communication bandwidth. In particular, for a shared-memory system, the memory bandwidth is limited by the number of outstanding loads the CPU can simultaneously handle. Hence, communications in the model should not be allowed to proceed without regard to the limitations of the system.

3.2 Pulled-Macro-Dataflow Model

The macro-dataflow model is clearly unfit for modeling the behavior of shared-memory multicore machines. In response, I introduce the *pulled-macro-dataflow* model of execution. This model specifically addresses the shortcomings of the previous model:

1. Communications are initiated by the receiving task. Consequently, data needed by a task are not present when the task begins execution. Instead, the task must immediately request the data it needs before it can proceed.

2. Parallelism in communication is reasonably limited. The maximum parallelism is represented by a positive integer M. When $M = 1$, all communication edges into a task must

be done sequentially. When $M > 1$, the total cost of communication may be reduced by a factor of up to M.

As with the previous macro-dataflow model, when a program begins execution, only the entry tasks may proceed to execute. Once they complete, however, there is no more work for them to do: they do not have to communicate any data. Instead, newly-ready tasks are allowed to execute, and it is their responsibility to initiate communications. Hence, the tasks receiving communication edges can begin execution immediately upon their predecessors finishing, but their execution cost is increased by the cost of communication. Furthermore, if the memory does not have enough bandwidth to support all data being communicated in parallel, some communications will be delayed until others have finished. More precisely, communication has a latency as long as either 1) the sum of all incoming communication edges divided by M or 2) the single highest latency communication edge, whichever is greater. This can be understood intuitively by considering that no matter how much memory parallelism exists, a single communication cannot finish any faster than its built-in latency; many communications may, however, be assisted by parallel memory accesses.

Only once all communication edges have finished does the task begin its actual execution. This restriction (rather than letting execution begin as soon as the first edge has finished) reflects the simple nature of these small tasks. They consist of relatively few instructions and there is no guarantee that execution can proceed without first gathering all needed data. Furthermore, as multicore processor evolution progresses, it is entirely conceivable that core complexity will be sacrificed in favor of greater core count. With less complex cores, out-of-order processing is not guaranteed to be available. Consequently, it would be nearly impossible for execution to overlap communication.

Finally, the pulled-macro-dataflow model can be described mathematically as

$$S_v = \begin{cases} \max\limits_{u,v \in V} \{F_u\} & v \text{ is not an entry task} \\ 0 & v \text{ is an entry task} \end{cases}$$

$$F_v = S_v + \tau_v + \max \left\{ \max_{(u,v) \in E} \{c_{(u,v)}\}, \sum_{(u,v) \in E} \frac{c_{(u,v)}}{M} \right\}.$$

3.3 Comparison of the Models

I have thus far assumed that because the pulled-macro-dataflow model is a better match for modern multicore processors, heuristics based on it will necessary yield better schedules than are achievable using the macro-dataflow model. I here examine the validity of this assumption and show that, indeed, there is much room for improvement in schedule makespans, particularly when using the new model.

3.3.1 Optimal Makespans

Comparing the quality of schedules produced under each model required a program to perform optimal scheduling. This program, OPTIMIZESCHEDULE, generates random graphs and finds the schedule which produced the minimal makespans in both models. It performs only the clustering pass of two-pass scheduling but, in so doing, returns the absolute minimum makespan. Optimally scheduling arbitrary graphs is known to be an NP-hard problem [2], so finding optimal schedules for large or even medium size graphs proved to be infeasible. Ultimately, the program was able only to produce optimal schedules for graphs with 13 or fewer vertices. The basic operation of OPTIMIZESCHEDULE is outlined in Figure 3.1.

OPTIMIZESCHEDULE is effectively an optimal clusterer. Line 1 finds all the different clusterings that are possible for the graph G. Line 2 puts the clusters together such that all tasks of the graph are included once and only once, as would be the case in any actual execution schedule. For a makespan to be computed, each cluster must be ordered, as shown in line 3. In a cluster containing n tasks, there are precisely $n!$ different orderings of the tasks (a well-known property

OPTIMIZESCHEDULE(task-graph G)

1 Find all subsets s of the tasks V in G
  Store these subsets in a set of subsets S
2 **for** each combination c of subsets s in S
  if c includes each task v in V once and only once
    Store c in the set of good subset combinations C
    $\triangleright$ Each subset s in c represents a cluster within the graph G
  for each good combination c in C
    for each subset s in c
3 Find all orderings o of the tasks v in s
      Store each o in a set of orderings s_o associated with s
4 **for** each unique combination of orderings o from each s_o s.t. all subsets s contribute one ordering
    Compute the macro-dataflow and pulled-macro-dataflow schedule makespans
    Keep track of the shortest macro-dataflow and pulled-macro-dataflow makespans for G
  Return the schedules yielding the shortest macro-dataflow and pulled-macro-dataflow schedules

Figure 3.1: Algorithmic outline of program to find optimal schedules.

of permutations). Since each ordering must be considered a candidate for producing the optimal makespan, this fact alone gives OPTIMIZESCHEDULE a complexity of $O(n!)$. Finally, in line 4, a single ordering from each cluster is combined to create a total schedule for the graph. This is repeated until all combinations of all orderings have been considered. The best overall schedule is recorded for both the macro-dataflow and pulled-macro-dataflow models.

Figure 3.2 compares optimal makespan lengths of random graphs using the two models. In this plot, the optimal schedules for both models are obtained for each of 1024 random DAGs. The schedule for the macro-dataflow model is then evaluated (without changing the schedule) under the pulled-macro-dataflow model. Since the schedule for the pulled-macro-dataflow model was already shown to be optimal, the makespan for the macro-dataflow model when evaluated using the pulled-macro-dataflow model will have either the same or greater makespan. Consequently, the y-axis in the figure shows the speedup of the optimal pulled-macro-dataflow model schedule as compared to the optimal macro-dataflow model schedule, when both are re-evaluated using the pulled-macro-dataflow model.

In this figure as well as in future figures, the x-axis of the graph is a measurement of graph granularity. Granularity is a measure of the total communication costs in a graph compared to

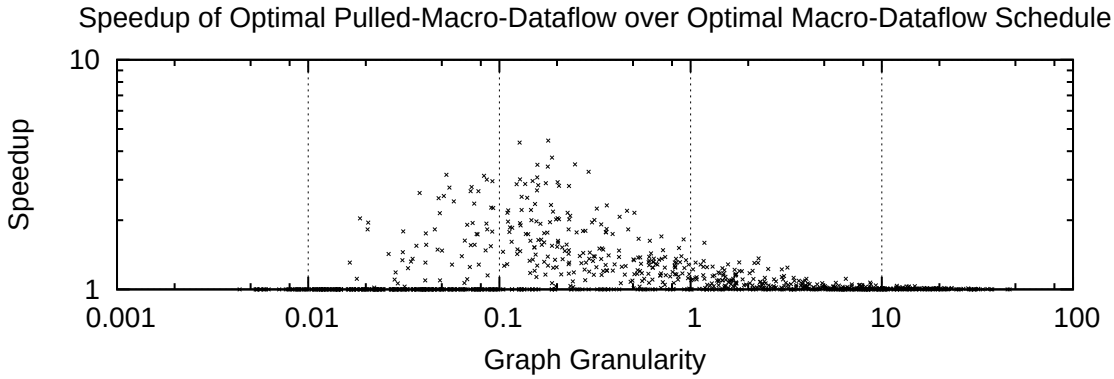


Figure 3.2: Potentially attainable speedup by using the pulled-macro-dataflow model.

the total execution costs. It is computed by finding the geometric mean of the computation cost divided by the total incoming communication cost for each non-entry task.

It can be clearly seen that even if existing clusterers, designed using the macro-dataflow model, could produce perfect schedules (which they cannot), they would still fall short of optimal schedules when the pulled-macro-dataflow model is adopted. Hence, there is validity to the assumption that heuristics can produce better schedules when designed specifically for the pulled-macro-dataflow model.

3.3.2 Quality of Existing Heuristics

Two of the highest-quality clustering heuristics which also exhibit very low computational complexity are DSC and CASS-II. Figure 3.3 shows how the schedules produced by DSC compare to the optimal schedules for the pulled-macro-dataflow model. Figure 3.4 does likewise for CASS-II.

With a slightly lower curve, DSC performs marginally better than CASS-II. Neither heuristic, however, performs admirably at granularities lower than 1; even above unity granularities, it is there is room for improvement.

With this further justification, it is clear that new heuristics are needed which will perform well under the pulled-macro-dataflow model.

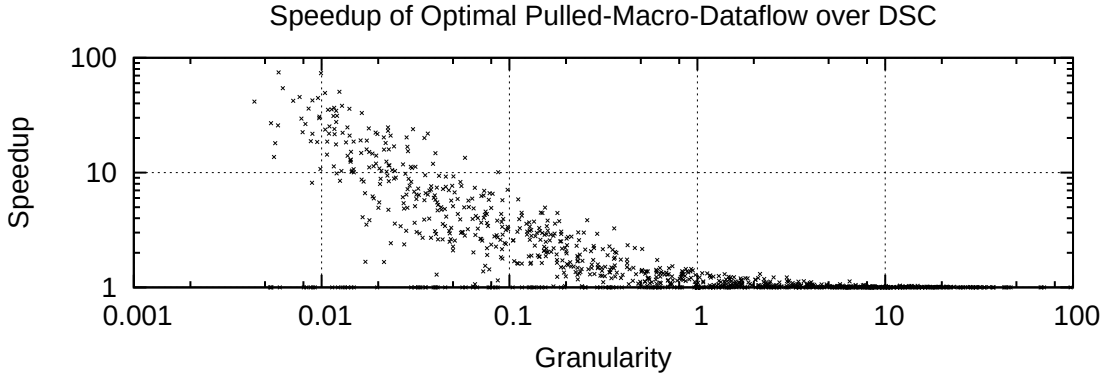


Figure 3.3: DSC clusterer compared to optimal schedules.

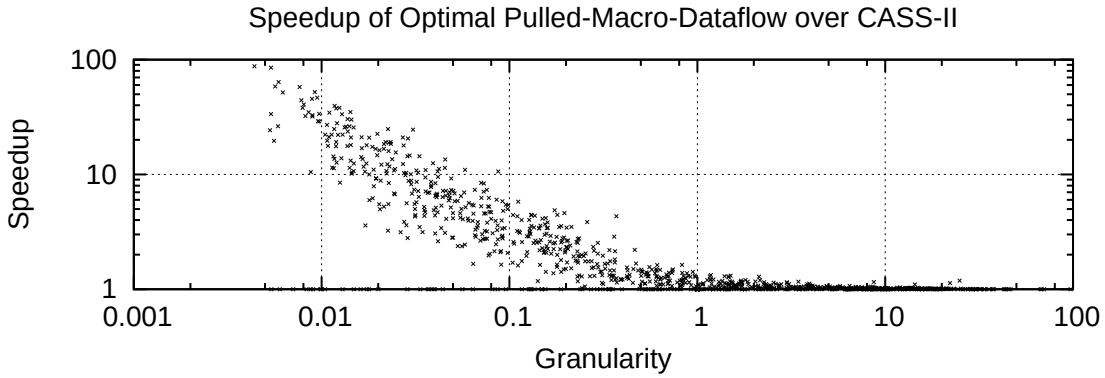
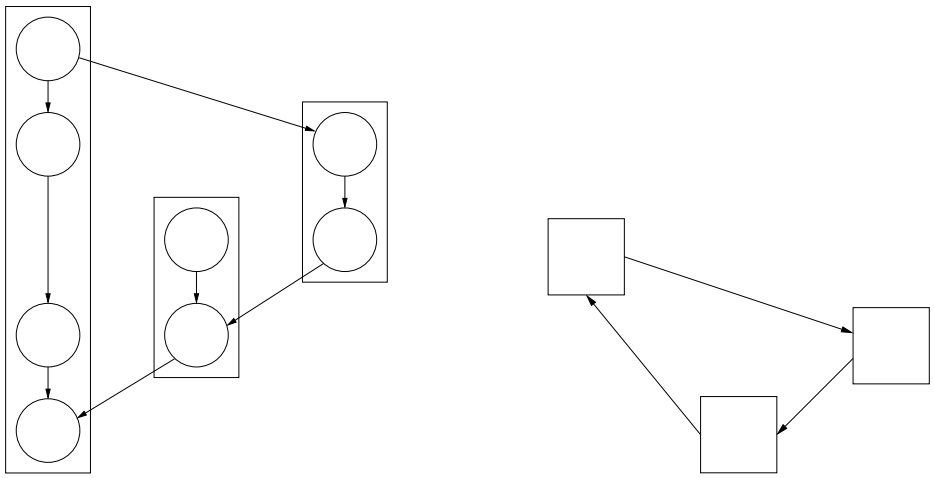


Figure 3.4: CASS-II clusterer compared to optimal schedules.

3.4 ConTouR

ConTouR (Concurrent Tournament Reducer) is a reduction heuristic. Although it was designed specifically to operate under the pulled-macro-dataflow model, it does not require that the clusterer also run under the pulled-macro-dataflow model. It works to produce as short a schedule as possible, no matter the clusterer. Consequently, it was possible to run ConTouR using DSC, as will be seen in Section 3.4.3 and Chapter 4.

Like many clusterers, ConTouR is based on a list scheduler. In contrast to those clusterers, ConTouR does not schedule tasks directly. Instead, fulfilling its role as a reducer, it schedules clusters. The clusters are treated as super tasks, with communications occurring with adjacent clusters. Where multiple communication arcs in the same direction exist between two clusters (a



(a) Clustered task graph. (b) Cyclic cluster graph.

Figure 3.5: When a task graph is collapsed into a cluster graph, the clusters may form cycles. These cycles must be broken before ConTouR can proceed. 3.5(a) shows an acyclic task graph after clustering, where clusters are indicated by boxes. 3.5(b) shows the same graph when the clusters are treated as vertices. In this format, the cycle is clearly visible.

result of multiple tasks communicating within those two clusters), the arcs are collapsed into a single communication edge.

3.4.1 Heuristic Description

A major problem that arises in designing a reducer as a list scheduler is that the clustering phase may have produced clusters which, when treated as super tasks, create a cyclic graph. Consider, for example, the graphs depicted in Figure 3.5. While the task graph shown in Figure 3.5(a) is an acyclic graph, when the clusters (shown by boxes) are collapsed and treated as individual vertices, a cyclic cluster graph is formed, as shown in Figure 3.5(b). All cycles in the cluster graph must be broken before reduction can be proceed. This is done by splitting clusters apart appropriately.

Breaking Cycles

For cycles to be broken, they must first be identified. Once identified, a cycle may be broken by splitting a single cluster into two separate clusters. ConTouR internally implements a cycle breaking function. It uses a stack-based depth-first search to identify a cycle in the cluster graph,

```
DISCOVERCYCLE(cluster-graph G)
    Perform a DFS of the cluster-graph
      for each cluster c in G
            c.visited = 0
            if c is a root task
              DFSStack.push(c)
      while DFSStack is non-empty
            c ← DFSStack.pop()
            if c.visited = 0
              c.visited ← 1
              DFSStack.push(c)
              for d in c.descendants
                    if d.visited = 0
                      DFSStack.push(d)
                      d.parent = c
                    elsif d.visited = 1
                      d.parent = c
                      ▷ This assignment completes the cycle
                      BREAKCYCLE(d)
                      Return true
            elsif c.visited = 1
              c.visited = 2
      Return false
```

Figure 3.6: DiscoverCycle. Routine to discover a single cycle in a cluster graph.

as depicted in Figure 3.6. Once DISCOVERCYCLE finds a single cycle in the cluster graph, it passes the last cluster in that cycle to BREAKCYCLE, shown in Figure 3.7. The key to BREAKCYCLE is that within each cluster of a cycle, it looks for two types of tasks: entry tasks and exit tasks. In this context, entry tasks are defined to be those tasks that have an incoming edge from the previous cluster in the cycle; exit tasks are tasks with an outgoing edge to the subsequent cluster in the cycle. As soon as BREAKCYCLE finds a cluster where the exit task executes before the entry task (depicted in Figure 3.7 as $exit < entry$), it breaks that cluster into two clusters by putting the entry task and all later tasks into a new cluster.

Once a cycle is broken, BREAKCYCLE returns control to DISCOVERCYCLE, which then returns $true$, indicating that a cycle was broken. Only when no cycles remain will execution reach

```
BREAKCYCLE(cluster d)
    succ = d
    curr = d.parent
    pred = NULL
    while (1)
      pred = curr.parent
      for each task v in curr
            if v has an incoming edge that forms part of the cycle
              entry = v
            if v has an outgoing edge that forms part of the cycle
              exit = v
      if exit < entry
            Break curr at entry
            Return
      succ = curr
      curr = pred
```

Figure 3.7: BreakCycle. Routine to break a single cycle in a cluster graph.

the end of DISCOVERCYCLE where *false* is returned, indicating to ConTouR that all cycles have been broken.

ConTouR as a Tournament Reducer

Once an acyclic graph cluster graph is formed, ConTouR can begin the true work of reduction. Other reducers schedule their clusters using a single best-guess approach: for any given cluster, they choose one other cluster that meets some limited criterion and merge the two clusters. For example, LB chooses the cluster with the smallest execution cost, while CM chooses the cluster that has the largest communication edge with the cluster being scheduled. Neither reducer considers the total impact that scheduling will have.

To remedy this, ConTouR considers the overall impact of scheduling a cluster on every thread. Thus, if a graph of c clusters is to execute on p total threads, ConTouR will speculatively schedule all c clusters on each of the p threads. It then examines the result of each speculative schedule and chooses the one schedule that produces the earliest finish time for the cluster at hand. The idea of using earliest finish time is also novel. While many clusterers schedule tasks to achieve

earliest start times, in the case of a reducer, the items being scheduled—clusters—can actually change length depending on which thread they are scheduled to (this is true for both models of execution). Hence, finish time is more relevant in reduction than start time.

When evaluating the impact of scheduling a cluster on a given thread, two things must be considered. First is the pulled-macro-dataflow model itself, as explained in Section 3.2. Each task within the cluster cannot execute before all of its predecessors have finished; once a task does start, it has to pull in all its needed data. Second, the occupancy of the thread has to be considered. Occupancy refers to the time spans which are already filled in a thread due to previously scheduled tasks. Thus, even if a cluster is allowed to execute during a certain time span because all its predecessors have finished, it may still be forced to execute later because that time span is already occupied by another cluster. Reduction very likely removes parallelism from a schedule; even if two clusters (or even two tasks within clusters) have the potential to execute in parallel, once they are scheduled to the same thread, they must be executed sequentially. ConTouR will schedule each task to begin execution as soon as all its predecessors have finished execution and as soon as there is space on the thread, whichever is later. While holes may exist in a thread, i.e. spaces where no task is scheduled due to dependency restrictions, ConTouR does not look for these holes; it simply schedules new tasks at the end of the thread.

While the approach to reduction taken by ConTouR allows each cluster to be scheduled such that it will finish as soon as possible given the scheduling decisions that have already been made, ConTouR cannot guarantee that the final schedule produced will be optimal. Indeed, it is often not optimal. As such, an exceptional case is accounted for within ConTouR: if, after producing a schedule for a given graph, a superior schedule (with a smaller makespan) would have been produced by assigning all clusters to a single thread, ConTouR discards the multi-threaded schedule and uses the single-threaded schedule instead. This adds very little overhead to ConTouR but recognizes the fact that at very low granularities task graphs may be better served by removing all parallelism. It will be seen in Chapter 4 that this simple optimization yields enormous speedup.

ConTouR as a Concurrent Reducer

Speculatively scheduling on each of p threads comes at a cost: the scheduling complexity of ConTouR is multiplied by p. This cost can be mitigated, though, by executing ConTouR itself concurrently on all p threads. This optimization divides out the factor of p from the complexity.

When ConTouR is executed as part of an Inspector-Executor framework, it runs immediately before the execution of the program it is scheduling. Hence, it can be safely assumed that as many cores as will be available for execution of the target program will also be available for execution of ConTouR. As a result of this (very safe) assumption, ConTouR requests p threads for its own execution. Each thread within ConTouR corresponds to one thread in the final program. Hence, each thread in ConTouR keeps track of a single thread to which clusters are being assigned; each thread also conducts its own speculative scheduling.

To coordinate the efforts of the many threads in scheduling, a single master thread also exists. In addition to the regular scheduling that exists for all threads, this master thread ensures that all threads are synchronized, chooses the thread that produced the best schedule for each cluster, and writes the corresponding speculative schedule to the master schedule so that all threads can see the decisions of scheduling. The overhead assigned to the master thread is small compared to the task of scheduling itself.

ConTouR Description

ConTouR is shown in its entirety in Figure 3.8. The first task that ConTouR must undertake is to internally form the clusters generated through clustering (line 1 in the figure). It does this by creating as many clusters as were reported by the clusterer and then assigning each task to the appropriate cluster. Once all assignments are made, it proceeds to break cycles until an acyclic cluster graph is formed.

It then condenses all communication and dependence information, as shown in line 2. This allows each cluster to have total communication costs (separating incoming and outgoing) with each other cluster stored locally, instead of relying on that information from its component tasks.

At line 3, ConTouR speculatively schedules all clusters onto a single thread to determine the single-thread makespan. Doing so also initializes global data structures to show that all clusters

```
CONTOUR(task-graph G)
1   Create a cluster graph C from task graph G
    cycles = DISCOVERCYCLE(C)
    while (BREAKCYCLE(C, cycles))
    for each cluster c in C
2       Sum the communications between c and its immediate predecessors and successors
        if c has no predecessors
            Put c in the ready list RL
3   Schedule all clusters into one thread and store makespan as singleThreadMakespan
    while RL is non-empty
        Retrieve the first cluster c in RL and delete it from RL
4       Send c to each thread p
        ▷ Each p simultaneously represents a processor/thread available to do scheduling
            and a processor/thread anticipated to be available for executing the schedule
        On each thread p
            for each topologically sorted task v in c
5               Calculate the earliest finish time v_f that v can finish on p
                ▷ v_f must respect both the occupancy of the thread and edge dependences
            Set the cluster finish time c_f as the finish time of the latest v_f
        On the thread p for which c_f is minimum
6           Assign c to p and update global data structures
        Mark c as scheduled
7       for any cluster d, child of c
            if d has no un-scheduled predecessors
                Put d at the back of RL
    Store latest finish time of a thread as multiThreadMakespan
8   if singleThreadMakespan < multiThreadMakespan
        Schedule all tasks onto one thread
    else
        Schedule tasks according to multi-thread schedule
```

Figure 3.8: ConTouR heuristic.

have been assigned to that thread. As the multi-thread makespan is determined, any cluster that has yet to be scheduled (among all the threads) is still shown in the first thread. In this way, a cluster is only removed from the first thread if doing so allows it to start earlier than it had previously. ConTouR is thus forced to be more conservative in splitting out clusters from the first thread and actually produces shorter makespans than if the clusters had not been scheduled to any thread from the beginning.

To determine a multi-thread schedule and makespan, the ready list is initialized with all clusters that have no predecessors. The ready list is built using a stack, so that the clusters placed last in the ready list are the first to be scheduled. As long as the ready list is non-empty, the first cluster is removed and speculatively scheduled on each thread (line 4). This involves placing the tasks of the cluster at the end of the thread, in order, and zeroing communication edges to other tasks that are 1) in the same cluster or 2) in the same thread (line 5). Once all threads have finished scheduling, the master thread chooses the thread that produced the earliest finish time for the cluster and writes its results into the global data structures seen by all threads (line 6).

Once a cluster has been scheduled, all its child clusters are considered for placement in the ready list. A child cluster is placed in the ready list once all its predecessor clusters have been scheduled. In this way, the schedule is built, one cluster at a time, from beginning to end, without ever having to go back through the graph and determine the makespan as its own step.

Finally, after the entire graph has been scheduled onto all available threads, the single-thread makespan is compared to the multi-thread makespan and the graph is scheduled according to the smaller of the two (line 8).

3.4.2 Complexity Analysis

Complexity of Breaking Cycles

It is well-known that a basic depth-first search (Figure3.6) runs in $O(v + e)$ time. In the function BREAKCYCLE (Figure 3.7), ConTouR must determine the entry and exit tasks of each cluster within a given cycle. This requires visiting every task and examining all edges of each task. Finding the right place to break a cycle therefore requires $O(v + e)$ time. Once ConTouR has determined where a cycle needs to be broken (which cluster and which task in that cluster), actually splitting the cluster requires creating a new cluster and moving over some fraction of the tasks. In a very large cluster, this could require $O(v)$ time. Hence the total cost of breaking a single cycle is $O(v+e+v)$, which simplifies to $O(v+e)$. Since this must be done once for each cycle, the complexity must be multiplied by the number of cycles to be broken. Because the cluster graph is built from an acyclic task graph, the maximum number of times that BREAKCYCLE can be called is limited. In the worst case, ConTouR would continue breaking cycles and splitting clusters until

all that remains is the original task graph, which is acyclic, with each task in its own cluster. As each cluster splitting creates one new cluster, the maximum number of times that clusters have to be split is limited to $O(v)$. Consequently, the process of breaking all cycles in a graph is $O(v(v+e))$.

Complexity of ConTouR

Creating a cluster graph internal to ConTouR (line 1) requires traversing all tasks once. This is $O(v)$. DISCOVERCYCLE and BREAKCYCLE have already been seen to be $O(v(v+e))$. Summing communications between clusters (line 2) can be done by traversing each task and each edge once, giving $O(v+e)$.

At line 3, all clusters are scheduled to one thread by traversing each task and edge once in $O(v+e)$ time. Maintaining and synchronizing multiple threads requires some constant overhead (line 4).

At line 5, each cluster must be scheduled on each thread. Since the clusters collectively cover the graph exactly once, this only requires examination of each task and each edge once, for $O(v+e)$. Performing this scheduling for p threads would increase this complexity to $O(p(v+e))$, but since p threads are used, the factor of p divides out.

Updating global data structures relies on the results obtained by the shortest thread, so no new computation is necessary at line 6. It can therefore be completed in $O(1)$ time.

At line 7, all children of the newly-scheduled cluster must be examined. This requires traversing the summed communication edges produced by line 2. There will never be more than $O(v)$ child clusters.

Finally, starting from line 8, the shorter makespan (either the single-thread makespan or the multi-thread makespan) is used to determine which of the two schedules will be chosen. Reporting the final schedule requires traversing each task once ($O(v)$).

Thus, ConTouR has a complexity of $O(v(v+e))$, which results from having to break cycles. If a clusterer can guarantee that no cycles will be formed by the clusters it creates, the complexity drops to $O(v+e)$.

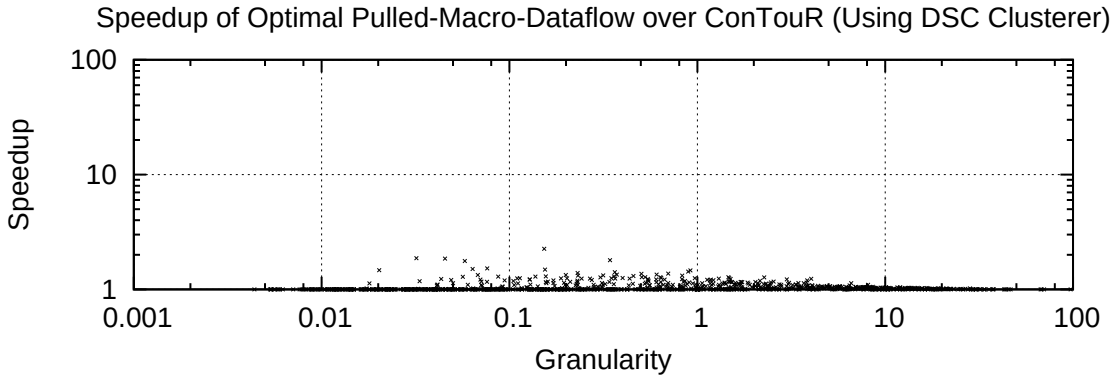


Figure 3.9: ConTouR reducer compared to optimal schedules. For this plot, ConTouR used the DSC clusterer and was allowed as many threads as there are clusters.

Table 3.1: Average speedup of optimal schedules over heuristic schedules. A smaller speedup indicates that a given heuristic was closer to optimal.

Heuristic	Speedup
DSC	3.93
CASS-II	4.38
ConTouR	1.04

3.4.3 ConTouR Compared to Optimal

ConTouR is unique among reducers in that it seeks to improve a schedule even when the number of clusters does not exceed the number of processors. Thus, even without requiring ConTouR to reduce the cluster count, it can produce higher quality schedules than were obtained just from the clusterer. LB and CM would make no changes to clustering in this situation and so cannot be meaningfully compared to optimal schedules.

Applying ConTouR, however, shows that schedule makespans are actually reduced. Figure 3.9 shows the speedup of optimal makespans over the makespans of the schedules produced by ConTouR when $M = 1$. For this experiment, DSC was used for clustering. It is clear that ConTouR vastly outperforms both DSC and CASS-II, producing near-optimal schedules for all graphs. Table 3.1 shows the average speedup over all 1024 graphs of the optimal schedules versus the schedules produced by DSC, CASS-II, and ConTouR.

CHAPTER 4. EVALUATION

With the introduction of the pulled-macro-dataflow model, it is important to evaluate existing heuristics along with ConTouR to understand how they perform. This evaluation consists of two distinct parts:

1. Theoretical performance. Because the heuristics designed for the macro-dataflow model treat communications very differently than they are ultimately evaluated under the pulled-macro-dataflow model, their relative performance may be drastically different than has been determined in past papers.

2. Performance on actual programs. Mathematical models aside, it is important to understand how well the heuristics manage to schedule real programs for execution on actual machines. Comparing actual runtimes to calculated makespans can also provide validation for the model.

4.1 Theoretical Performance

To test the various heuristics for their theoretical performance, a program was written to deterministically generate random acyclic graphs of arbitrary size. Each graph was scheduled using some combination of clustering and reduction heuristics and the final schedule lengths evaluated using the pulled-macro-dataflow model. At finer granularities, the relative importance of communication costs increases. It is useful, therefore, to distinguish among varying granularities in comparing heuristic quality.

For the following results, 1000 random graphs were generated, each consisting of 500 tasks. As DSC is generally considered the best clusterer, for this comparison of reducers, DSC is always used for clustering. It is unfortunately not possible to compare each heuristic's schedule against the optimal schedule; consequently, the makespans yielded by each heuristic are instead compared

against the makespans produced when no reducer is used. This forces CIRA to rely on the list scheduler to reduce the number of clusters to match the number of processors. It is important to note that the list scheduler has been tuned for the pulled-macro-dataflow model, allowing it to make better scheduling decisions than it would normally.

4.1.1 Load Balancing

The description of Load Balancing (LB) given in Chapter 2 comes from [17]. Unfortunately, this description is incomplete: it neglects the case where the cluster with minimum workload has no communication edges. Allowed to proceed as-is, an LB reducer would repeatedly select the minimum workload cluster, choose not to merge it with any other cluster, and leave it as the minimum workload cluster for the next iteration. To avoid this infinite non-progressing loop, I have added an additional step to LB: if a cluster has no communication edges, it is merged with the cluster which has the next-smallest workload.

Figure 4.1 shows four related plots. Each shows the speedup of LB over the list scheduler, but the number of threads to which the graphs are scheduled changes for each plot. From top to bottom, the number of threads used is 4, 8, 16, and 32. It is interesting to note the varying performance of LB across thread counts and granularities. At low thread counts and higher granularities, LB tends to perform worse than list scheduling, while at lower granularities, LB does slightly better. These results are counterintuitive: at high granularities, the relative importance of execution cost (as opposed to communication cost) should increase. Since LB focuses on execution costs, it would seem that it should perform best at high granularities. However, LB does rely on communication edges to choose how to balance loads. At low granularity graphs, it appears that this emphasis on communication edges has a significant impact on minimizing final makespan. The performance of LB also manages to improve as the number of threads increases.

4.1.2 Communication Minimization

The description of Communication Minimization (CM) given in Chapter 2 also comes from [17]. Like LB, it omits an important class of cluster graphs. It is possible, particularly in large graphs, for all communication edges to be zeroed (through cluster merging) without sufficiently

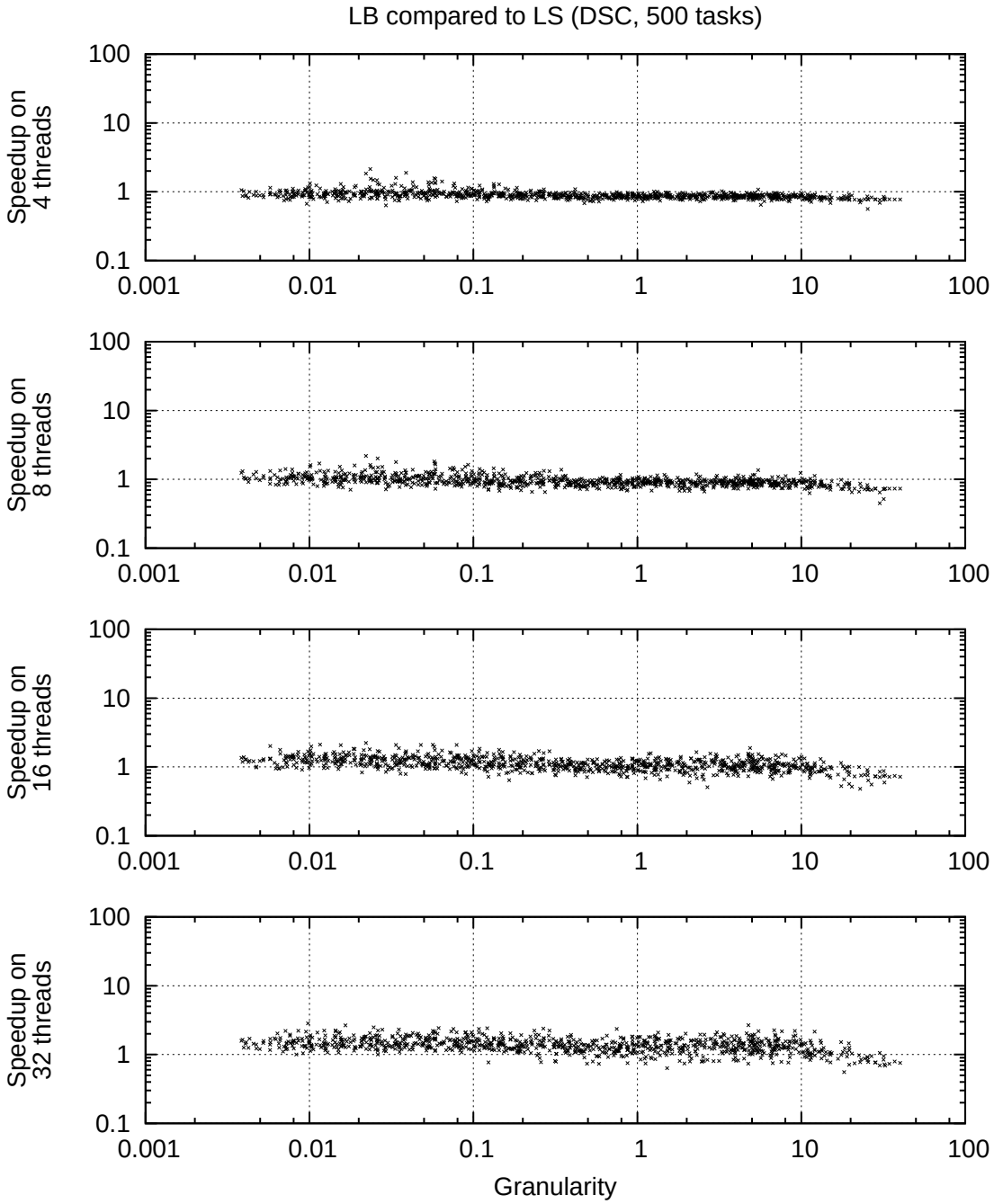


Figure 4.1: Load Balancing speedup over list scheduling (LS) using differing numbers of threads. From top to bottom, the plots show the speedup using 4, 8, 16, and 32 threads.

reducing the number of clusters. I modified CM to recognize this case and abort its attempts to further merge clusters once there are no more non-zero communication edges. It then relies on CIRA's list scheduler to manage the final reduction and assignment of clusters to threads.

Figure 4.2 shows the performance of CM compared to the list scheduler for varying thread counts. The especially poor performance of CM at high granularities is unsurprising: on high granularity graphs, communication costs are relatively unimportant, yet this is all that CM considers. It repeatedly merges clusters that have communication between them at the expense of potentially creating huge imbalance between thread loads. Despite the cost of communication being minimized, the threads are so imbalanced that the available processors are being very poorly utilized and makespan suffers. At low granularities, however, CM does much better, since communication costs are the dominant factor in determining schedule length. CM naturally eliminates the most costly parts of the graphs. Interestingly, opposite of the pattern seen in LB, CM's performance degrades with increasing thread counts.

4.1.3 ConTouR

ConTouR was the final reducer evaluated. Its performance is shown in Figure 4.3. For these tests, M was set to 1. At low thread counts and higher granularities, ConTouR performs comparably to list scheduling. This is not terribly surprising, though, as the list scheduler was optimized for the pulled-macro-dataflow model. As the thread count increases, ConTouR shows an improvement over list scheduling.

A rather unique feature of ConTouR's plots is the very distinct change in slope at around 0.1 granularity graphs. This phenomenon is easily understood, though, by re-examining ConTouR's approach to scheduling. Recall that ConTouR includes special handling for an exceptional case: some graphs may have such exorbitant communication costs that scheduling all tasks to a single thread may prove to have a shorter makespan than ConTouR's normal approach. Below some granularity, virtually all graphs exhibit this feature. In Figure 4.3, below 0.1 granularity, nearly all the graphs are being scheduled to one thread. At higher granularities, ConTouR's more complicated approach wins out. In Figure 4.4, ConTouR has been run with its two different approaches neatly split out to illustrate their different results. In both plots, the graphs were scheduled to 32 threads.

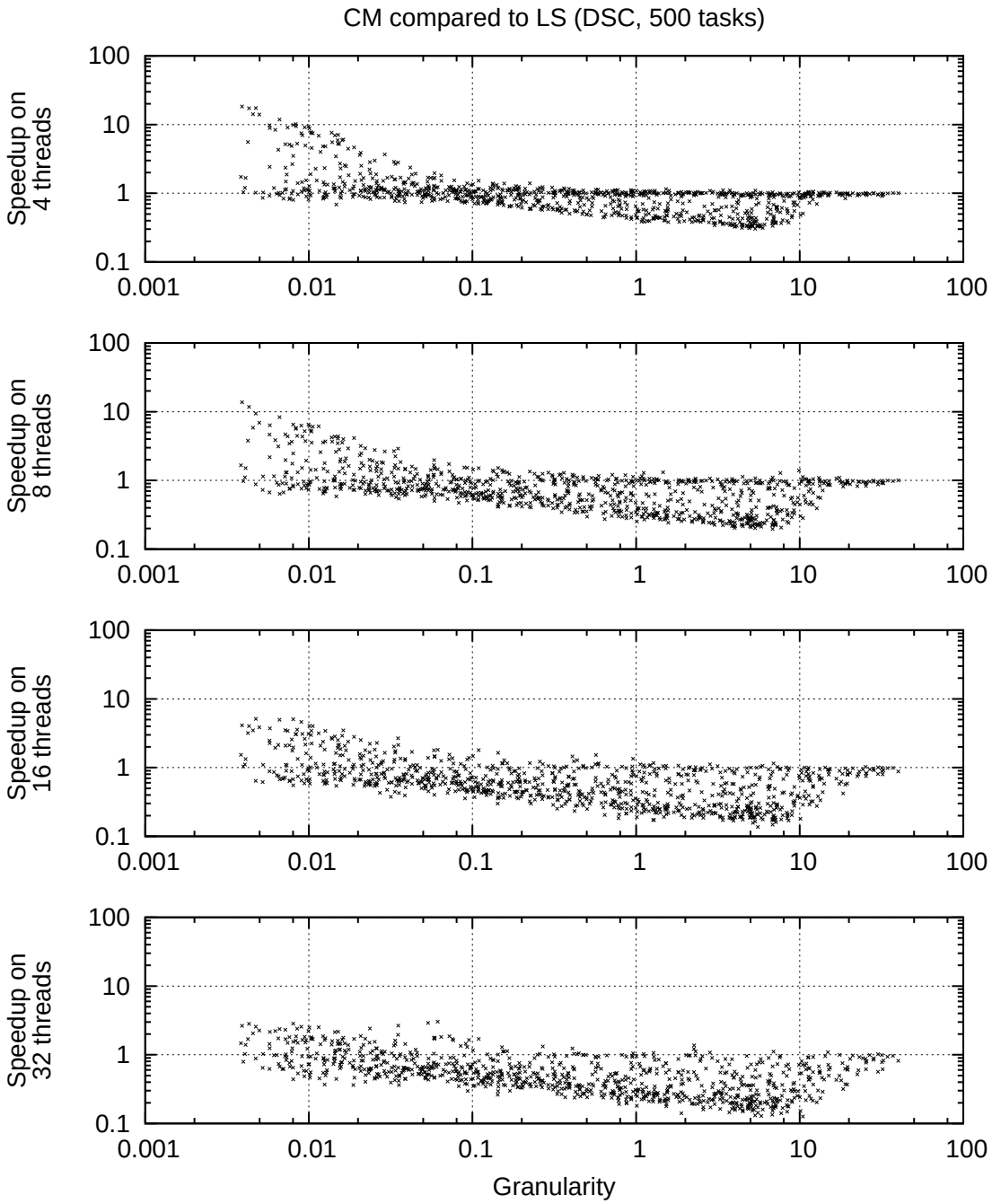


Figure 4.2: Communication Minimization speedup over list scheduling using differing numbers of threads. From top to bottom, the plots show the speedup using 4, 8, 16, and 32 threads.

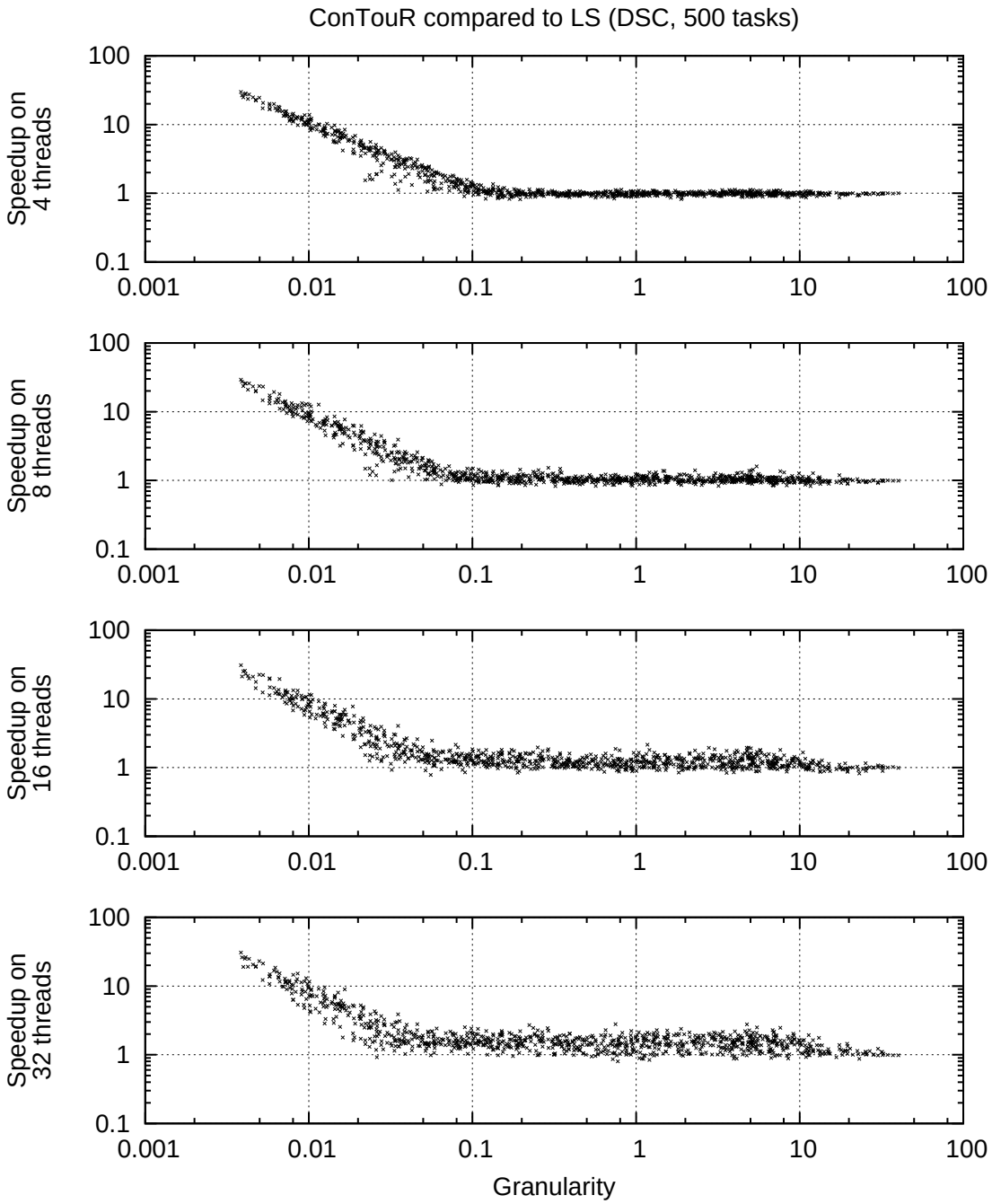
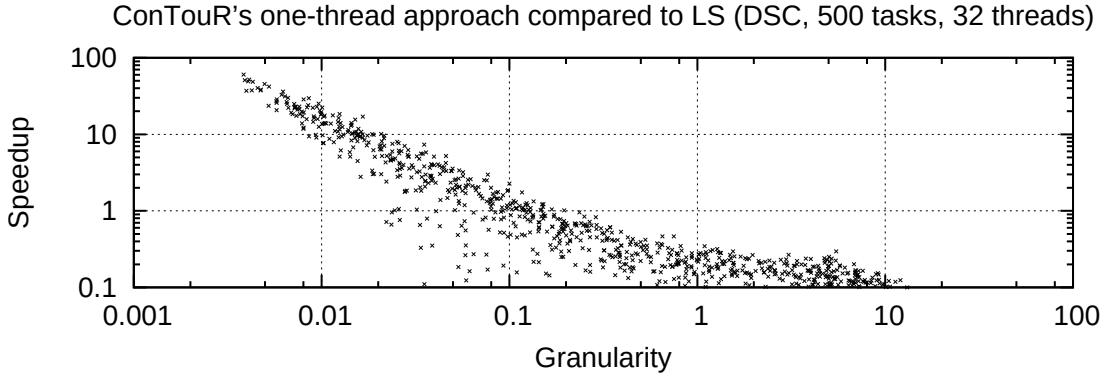
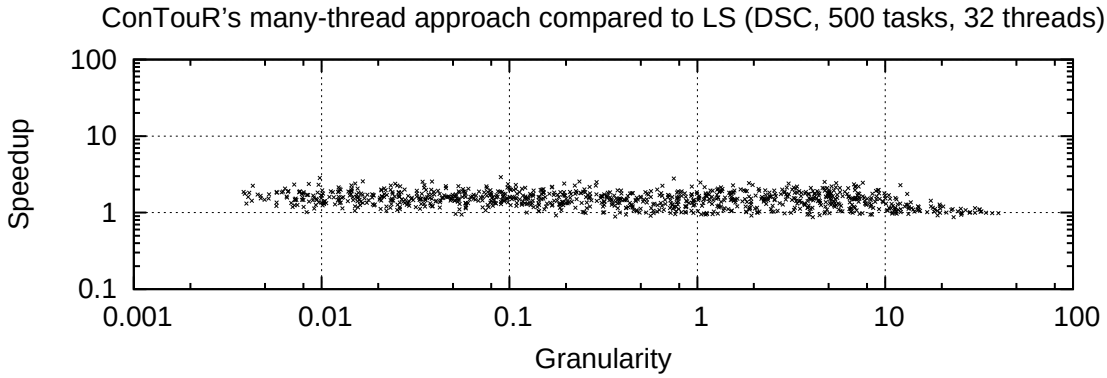


Figure 4.3: ConTouR speedup over list scheduling using differing numbers of threads. From top to bottom, the plots show the speedup using 4, 8, 16, and 32 threads.



(a) All tasks are assigned to a single thread.



(b) Tasks are allowed to execute on multiple threads.

Figure 4.4: ConTouR's two different approaches to scheduling. In 4.4(a), all tasks are always scheduled onto a single thread, clearly leading to very poor schedules at higher granularities. In 4.4(b), ConTouR's multi-threaded approach is shown for all granularities. It does not perform as well at very low granularities as the single-thread approach.

The granularity point where ConTouR's many-threaded approach begins to outperform the single-threaded approach also changes as the thread count is increased. It moves toward lower granularities, making the speedup in the many-threaded region relatively more important. This is fortunate, as the quality of schedules produced by the many-threaded approach increases with greater thread counts.

The average speedup of each reducer over list scheduling on the 1000 graphs tested is shown in Table 4.1. From a theoretical perspective, it is clear that ConTouR outperforms all other reducers.

Table 4.1: Average speedup of each reduction heuristic over list scheduling.

		Threads			
		4	8	16	32
Reducer	LB	0.92	1.04	1.25	1.48
	CM	1.63	1.41	1.24	1.12
	ConTouR	6.11	6.27	6.69	7.37

4.2 Actual Programs

In testing the performance of the reducers on real programs I have selected two programs, both of which are well-suited to the Inspector-Executor framework. In running both programs, ConTouR was executed using $M = 1$. Tests using different values for M did not show a significant difference from what is reported here.

4.2.1 Jacobi Iterative Solver

The first program is a Jacobi iterative solver for sparse matrices (the CIRA implementation was put forth in [6]). Because it executes multiple iterations of the program using the same schedule, this program is a perfect match for CIRA. The program is composed of two types of tasks, each of which is repeated n times, where n is the number of rows in the sparse matrix. The first type of task is termed a *row-task*. Given a matrix A of dimensions $n \times n$, to solve the equation $Ax = b$ the work performed by each row-task is

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)} \right), \quad i = 1, 2, \ldots, n$$

where i indicates the row and k indicates the iteration. When solving a 40,000-row matrix, 40,000 row-tasks would be instantiated, each of which is able to proceed independently of the others. The row-tasks also check whether the difference $x_i^{(k+1)} - x_i^{(k)}$ is less than some tolerance t; they return a Boolean value indicating convergence, or a difference less than t. The second type of task is the *or-task*, whose job it is to find the logical *or* of all the returned Boolean values from the row-tasks. The or-tasks are structured to form a tree, each *or*-ing together two Boolean values from the level above until only a single Boolean value remains, which indicates whether the solution has

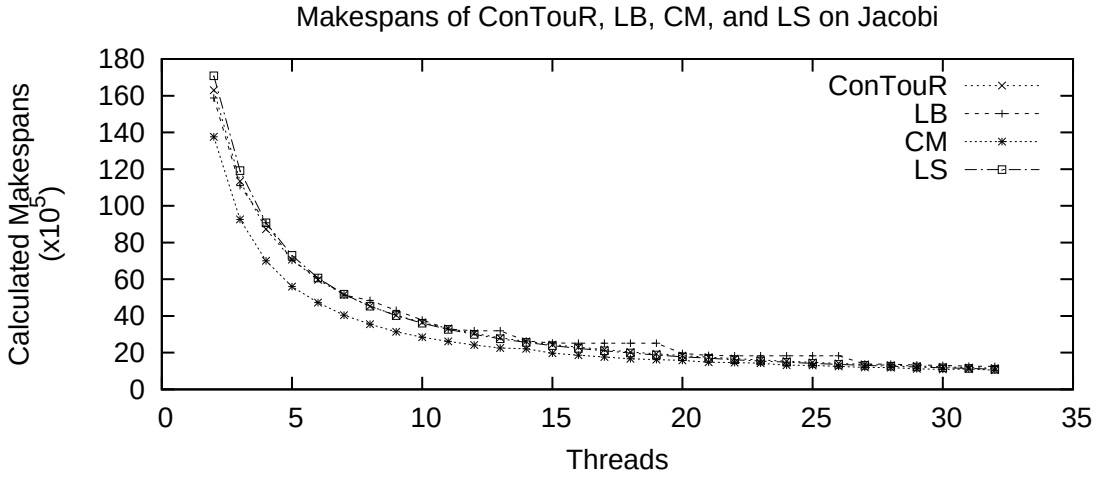
converged. With each task performing only a single *or*, the or-task tree portion of the task graph is very low granularity. The order of execution of the or-tasks is critical to producing a valid result. A given or-task can only proceed once the two tasks preceding it have completed. Only once the result of the or-task tree is *true* does the program terminate. If the result is false, the results from the last iteration are used for another iteration and the scheduled task graph is re-run.

Figure 4.5 shows the predicted makespans and the actual runtimes of the program using the DSC clusterer in combination with each of the reducers, including the list scheduler. For this experiment, six random $40,000 \times 40,000$ matrices were generated with density 0.1 and $t = 10^{-9}$. Random b vectors were also created. The schedule produced by each reducer was run until convergence was reached. The runtimes indicate the average runtime of solving the six problems.

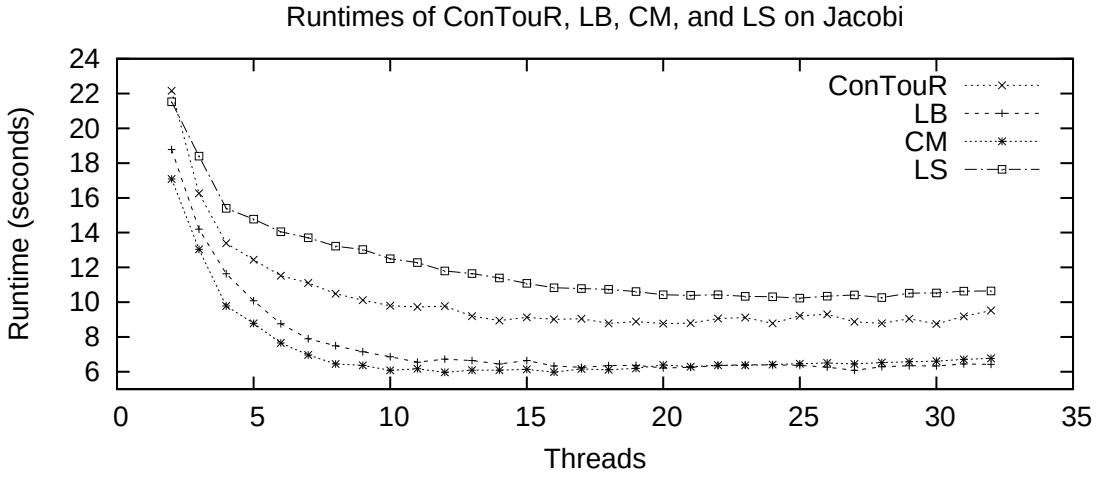
Unsurprisingly, the list scheduler performs the worst. More surprising is that LB and CM perform so well. This can be understood by considering the operation of each reducer. LB operates by merging the smallest cluster (lowest execution cost) with an adjacent cluster that has the lowest execution cost. In this program, the or-tasks all have identical execution costs, and they are all much smaller than row-tasks. Consequently, clusters chosen to merge tend to be those consisting of or-tasks (or consisting of only one or-task). More importantly, the merged clusters tend to be neighbors, as a consequence of the ordering of tasks within this program. This introduces an advantage not accounted for in ConTouR: spatial locality. Adjacent clusters and tasks operate on related data, allowing the LB schedule to benefit hugely from locality.

In the case of CM, a similar effect is seen. The communication costs between clusters tend to be about the same, since the task graph is so uniform. Any tasks consisting of equal numbers of *or-tasks* will have identical communication costs between them. CM breaks ties by using the order of graph creation, which, in this case, places tasks operating on adjacent data next to each other in creation order. As a result, CM unintentionally takes advantage of spatial locality as well. Support for this explanation can be seen even in Figure 4.5: as the number of threads increases, CM actually performs worse, as the data is spread out among threads.

ConTouR performs better than list scheduling, but not so well as LB and CM. This is likely because it is unable to take advantage of the spatial locality effects exploited by the other reducers. Nevertheless, it manages a speedup as predicted by the decreasing makespans.



(a) Calculated makespans.



(b) Measured execution times.

Figure 4.5: Makespans and runtimes on Jacobi iterative solver. The number of threads is varied from 2 to 32.

Closer examination of thread assignments confirmed that LB and CM are indeed taking advantage of spatial locality effects, while ConTouR and the list scheduler are not. Interestingly, some of these useful cache effects can also be generated in ConTouR and list scheduling by randomizing scheduling priority in the list scheduler. The list scheduler normally assigns uniform priority to all tasks; by instead giving all tasks a random priority, tasks relying on data adjacent in memory will occasionally be scheduled very close together, thereby taking advantage of spatial locality. Figure 4.6 shows the results of scheduling with random priority. The figure shows that both ConTouR and the list scheduler produce much shorter schedules when using random priori-

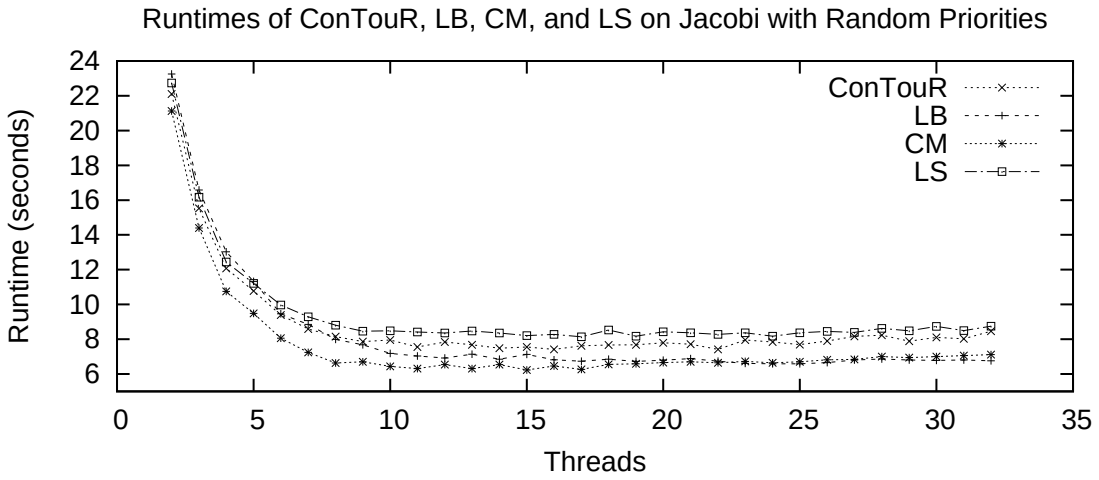


Figure 4.6: Runtimes on Jacobi iterative solver with random priorities. The other parameters are identical to those used in producing Figure 4.5.

ties. The schedules produced by LB and CM, in contrast, became slightly longer, indicating that their unintentional utilization of spatial locality was disrupted by the randomized priorities.

4.2.2 Liberty Simulation Environment

The Liberty Simulation Environment (LSE) [20] is an architectural simulator with support for multiple simulated cores and multiple simulator cores. It relies on the two-pass approach to scheduling to produce high-quality schedules. Furthermore, the schedule produced is reused millions of times[1] through the course of execution, which is perfect for mitigating the overhead of scheduling.

I used LSE to further test the reducers and the pulled-macro-dataflow model. The following experiments were run simulating a 32-core PowerPC system executing the SPLASH-2 FFT benchmark [21]. It was discovered in the course of running these experiments that the list scheduler in LSE has a bug which sometimes leads to a segmentation fault. Unfortunately, the schedules produced by LB nearly always produce a segmentation fault. Consequently, this experiment had to be run without using LB. For the remaining reducers, I ran LSE on two different systems. The first system was the same as used for the Jacobi iterative solver. It is a 32-core machine built from 8

[1]The schedule is reused once per simulated clock cycle. Any simulated program of reasonable length should run for millions of cycles.

Table 4.2: Summary of systems used to run LSE.

System	Processors	Cores	Frequency	L2 Cache per core	L3 Cache per processor
AMD	8	32	2.2 GHz	512 KB	2 MB
Intel	2	12	2.66 GHz	256 KB	12 MB

4-core AMD Opteron processors. The second system is a 12-core system built from 2 6-core Intel Xeon processors. The details of these two systems are provided in Table 4.2.

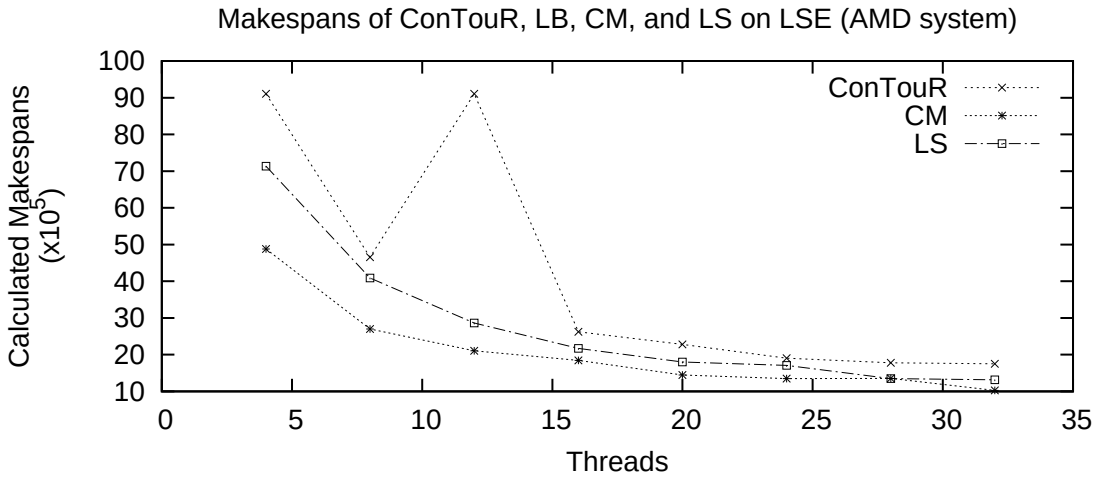
LSE is unique in that it is able to run on a single core for some time in order to estimate execution costs from measurements it takes. Once these costs are decided, scheduling takes place as normal. Consequently, the makespans predicted from each reducer will vary between runs and particularly when running on different systems.

Running on the AMD system, the number of cores used to run the simulator was varied from 4 to 32 in increments of 4. Figure 4.7 shows the calculated makespans and the measured runtimes. In this figure, the relative running times of the reducers usually agrees with the makespans, i.e. CM runs faster than list scheduling which runs faster than ConTouR, precisely as predicted by the makespans. Unfortunately, however, the shapes of the runtime curves do not match well the makespan curves: while the makespans decrease consistently as the thread count increases, the actul runtimes tend to get worse.

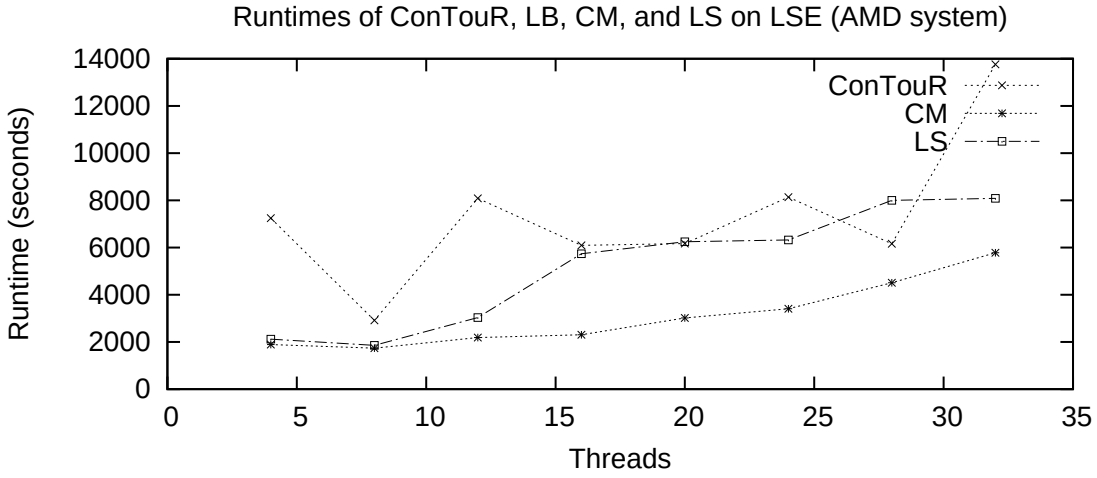
In running on the Intel system, the core count was varied from 2 to 12 in increments of 2. The results of these runs are shown in Figure 4.8. Comparing the makespans and the runtimes, there is greater inconsistency than there was on the AMD system: ConTouR was predicted to produce the shortest schedules but was measured to take the longest. Nevertheless, some of the features in the shapes of the makespan curves can be seen in the runtime curves. This indicates that even though the pulled-macro-dataflow model failed to predict the runtimes accurately, it nevertheless managed to predict some of the larger shapes of the runtime curves.

4.3 Discussion of Results

In the purely mathematical results obtained in Section 4.1 it is seen that the relative performance of LB and CM actually depend on the thread count: CM does better at low thread counts while LB does better at high counts. This is in contrast to what was reported in [17], which found



(a) Calculated makespans.

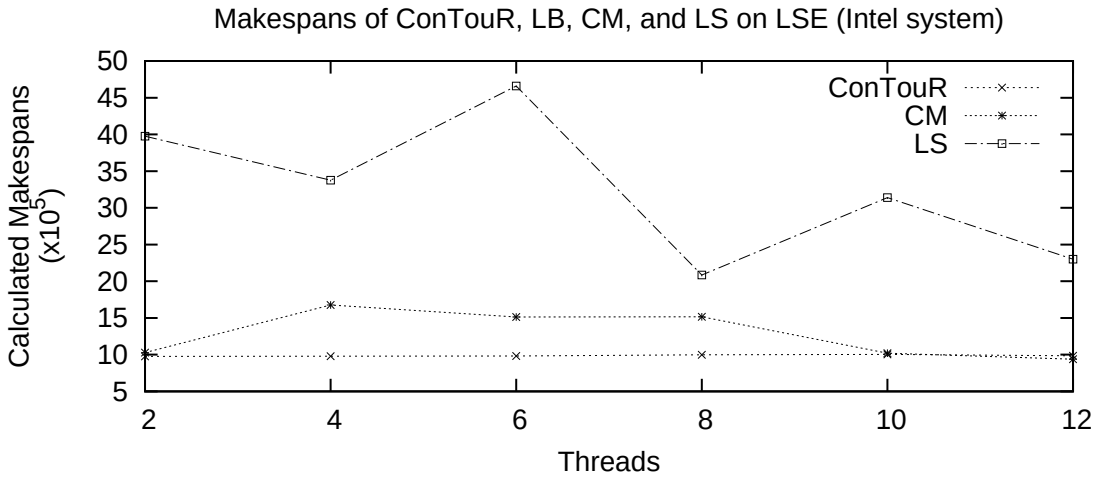


(b) Measured execution times.

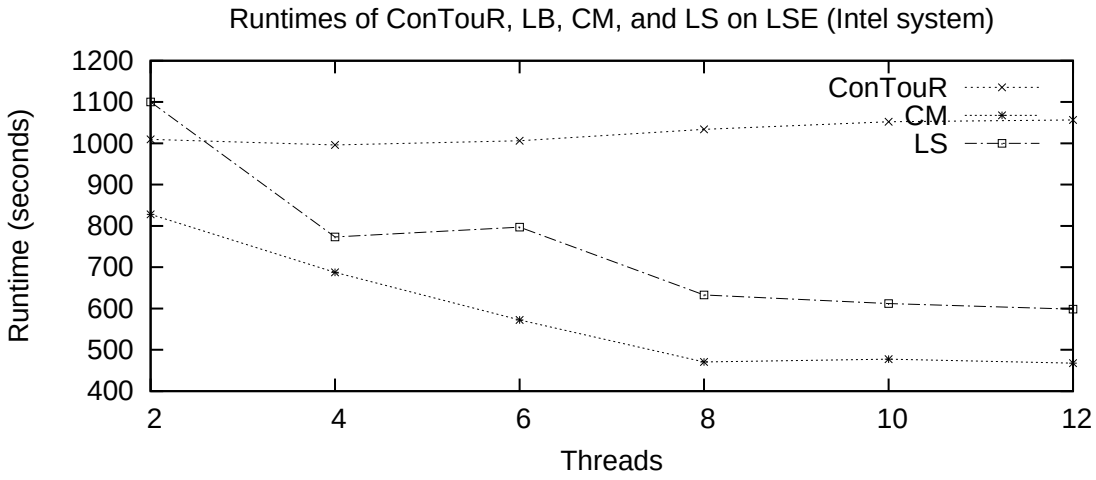
Figure 4.7: Makespans and runtimes of LSE on AMD system. The makespans of each reducer only correspond roughly to the measured runtimes.

that LB is the superior reducer. The fact that LB's performance improves with increasing thread count while CM's degrades does, however, imply that LB is the better choice of reducer for future many-core processors.

Outperforming both LB and CM, ConTouR consistently did better than list scheduling at all granularities and thread counts. This is because it incorporates both execution and communication costs in making its scheduling decisions and schedules each cluster to finish as early as possible. The majority of its speedup, though, comes from ConTouR's recognition that at very low granularities, communication costs may so far outweigh execution costs that the best scheduling decision



(a) Calculated makespans.



(b) Measured execution times.

Figure 4.8: Makespans and runtimes of LSE on Intel system. The shapes of the curves roughly to correlate between the two plots, but the relative performance of ConTouR with respect to the other reducers is entirely different than predicted.

is to place all tasks on a single thread. In these very low granularity graphs, CM also performs very well by eliminating the most expensive communication edges. CM, however, still uses all available threads, allowing for parallelism where ConTouR entirely eliminated it. But because ConTouR far outperforms CM at higher granularities, it remains the better choice for general graphs when only considering theoretical performance.

The test of the reducers on useful programs was also a test of the pulled-macro-dataflow model. The experiments did not provide the hoped-for evidence to support the validity of the model. There are a number of plausible explanations for this which fit into two general categories.

4.3.1 Experimental Shortcomings

The first category is experimental shortcomings. This includes any problems that may have arisen in the experimental process itself.

Incorrect communication costs It is entirely possible that the communication costs provided for the programs are completely inaccurate. In LSE, for example, all edges are weighted equally; unlike task costs, they are not measured through the course of program execution.

Memory parallelism These experiments used $M = 1$ for ConTouR. This forced all communication edges to be serialized and provided the attractive feature of forcing ConTouR to work especially hard to avoid communications wherever possible. While some small tests using other values for M did not show significant differences in runtimes, it is possible that adjusting M realistically in combination with accurate communication costs would have a noticeable impact on running times.

4.3.2 Model Shortcomings

The disappointing results seen may also be explained by some of the omissions from the pulled-macro-dataflow model.

Spatial locality As was demonstrated for the Jacobi iterative solver experiment, accidental exploitation of spatial locality can provide immense speedups. It is possible that LB and CM will have this advantage for nearly any realistic program that is written, simply due to the natural structure that programs take. The model would be benefited by the addition of locality awareness.

Other cache effects Cache is an essential element in program runtime. Insufficient cache can easily destroy a program's performance. In Figure 4.7(b) runtimes increased with increasing thread

counts. It is entirely possible that with fewer threads, the threads were being spread out among the processors, giving each thread access to more cache than would be available when all cores were being actively used.

Additional communications The pulled-macro-dataflow model only provides support for communications from one task to another. It does not consider initial communications, such as those needed by entry tasks to obtain the data necessary for the program. There may also be inter-iteration communications.

4.3.3 Final Remarks

Despite the poor predictions made in these experiments, I believe that the pulled-macro-dataflow model is a more accurate model than its predecessor, the macro-dataflow model. There is important evidence in support of the validity of the model: very distinct features present in the makespan plots are reflected in the runtime plots. For example, the enormous spike in Figure 4.7(a) on the ConTouR line at 12 threads is clearly present in Figure 4.7(b)). The general shape of the list scheduler line in Figure 4.8(a) is also reflected in Figure 4.8(b). These observations both point toward the pulled-macro-dataflow model, even in its present state, having some validity in modeling multicore systems.

CHAPTER 5. CONCLUSION

This thesis has presented a new model of execution tailored to shared-memory multicore processors. It has also introduced a reduction heuristic designed for the new model which was shown to outperform existing heuristics on the model. A comparison of existing reduction heuristics has demonstrated that Load Balancing tends to perform the best in application to Inspector-Executor-style programs at low granularities. While the pulled-macro-dataflow model of execution was not conclusively shown to be an accurate model of multicore machines, it was seen to have at least some validity. This chapter summarizes the contributions of this thesis and presents some possible future work that could further develop the model.

5.1 Summary

Modern multicore machines have become nearly ubiquitous. They are not, however, providing the same rapid increases in performance that consumers have come to expect and rely upon from decades of experience. In order for the increasing power of these processors to be realized, software must be rewritten to explicitly utilize all available resources. In the past, a hardware upgrade automatically yielded enormous gains in software performance. This fundamental change has introduced a major roadblock: most programs cannot be easily parallelized and the task of doing so is both arduous and error-prone.

As developers work to provide more powerful tools for the development of parallel software, it is important for program parallelism to be simply encoded and easily accessed. The task graph is a representation that meets this need very well. As a directed acyclic graph, understanding the flow of the program is straightforward and the potential for parallel execution is inherent in the graph's structure. With this representation, it becomes the job of a scheduler to assign tasks to processing cores and maintain proper dependences throughout the program.

Nearly all scheduling heuristics have assumed the macro-dataflow model of execution. This model, however, fails to accurately represent communications in a shared-memory multicore machine. It has two primary shortcomings: it assumes that communications are initiated by sending tasks and it allows infinite communication parallelism. Neither assumption is realistic in a shared-memory system. The pulled-macro-dataflow model addresses these concerns by forcing communications to be initiated by receiving tasks and by realistically limiting the number of communications that can take place simultaneously.

In accordance with the proposition that the pulled-macro-dataflow model better represents actual hardware, a new heuristic was developed that performs the second pass of two-pass scheduling. The Concurrent Tournament Reducer (ConTouR) schedules clusters using multiple threads and by assigning clusters to the thread which allows for earliest finish time. It does this in $O(v(v+e))$ time.

To validate the new model and to evaluate the performance of ConTouR, ConTouR was compared to two existing reducers: Load Balancing (LB) and Communication Minimization (CM). In evaluating makespans yielded by each reducer on random graphs, it was clear that ConTouR produced shorter schedules than either other reducer. In applying the reducers to actual programs, however, it was found that ConTouR performed far worse than either reducer despite often being predicted to do better. It is likely that much of the discrepancy between the pulled-macro-dataflow model and actual programs deals with spatial locality and other cache effects. For the low-granularity graphs expected to dominate parallel programs in the future, CM and LB seem to provide the greatest ability to effectively schedule graphs and maintain good caching behavior. A larger set of programs will need to be examined to determine which of the two is superior.

While the pulled-macro-dataflow model cannot capture cache effects, it was nevertheless shown to predict some of the more localized behavior of the reducers. The model should not, therefore, be discarded entirely.

5.2 Future Work

The pulled-macro-dataflow model is a good starting point for better representing shared-memory multicore processors. Several possible additions could make it a viable representation.

Cache Behavior

The lack of understanding of cache behavior has been seen to be a major limiting point in the efficacy of the model. By adding support for cache behavior, the model could be hugely benefited.

Spatial Locality The model has no ability to represent spatial locality. It zeroes only those communications that connect tasks on the same processor (representing temporal locality) but assumes that all other communication edges remain at their full cost. It has no notion of adjacent data and cannot, therefore, account for multiple data being collected in one communication. To gain this functionality, an understanding of the layout of data in memory would need to be added.

This addition would require the creation of a new type of edge in the task graph: the locality edge. This edge represents the fact that the data consumed by two tasks are adjacent or nearby in memory. With a sufficient knowledge of the precise data layout and cache properties of the target machine, the locality edge would even come to represent the fact that separate data would be contained in the same cache line.

Temporal Locality Temporal locality is currently accounted for in the pulled-macro-dataflow model (as it was in the macro-dataflow model) by zeroing communication edges between tasks that execute on the same core. This ignores the fact that older data in a core's cache may be evicted before they can be used by the receiving task.

A simple addition to the temporal locality support of the model would be to require that tasks on the same processor consume the data communicated between them within some window in order for their edge to be zeroed. A more complicated addition would be dependent on the specifics of the CPU architecture and could support cache replacement policies to precisely predict when eviction occurs.

Subdividing the Task Graph

It is often the case that a single task graph can be composed of distinctive parts which have very different characteristics and different granularities. For example, the or-task tree of the Jacobi iterative solver has a much lower granularity than the row-tasks. The best decision for the tree

might be to schedule it entirely on one core, leaving the row-tasks spread among many cores. Of the reducers examined in this thesis, none can explicitly recognize and deal properly with this case.

Other graphs may have sections which represent the iterations of a loop. These sections could also be separated from the rest of the task graph to be scheduled more effectively. Future schedulers should be able to take an already subdivided task graph and choose the best scheduling approach for each section.

Updating Heuristics

An unexpected result of this thesis was that CM proved to perform very well. It may be that huge benefits can be reaped from updating existing heuristics to calculate their costs according to the pulled-macro-dataflow instead of the macro-dataflow model. The effects might be most visible in clustering heuristics, which rely heavily on minimizing communication costs.

Undoubtedly, though, as such things as locality edges and support for graph subdivision are added, new heuristics will be indispensable. Some of the features of ConTouR are unique and could be considered for inclusion in future heuristics. In particular, considering finish time of clusters and operating on many threads concurrently seem significant.

5.3 Final Thoughts

The reality of modern computers is that core counts will continue to increase, while there is no guarantee that individual cores will continue to improve. Programmers need tools to help them produce multi-threaded programs. The pulled-macro-dataflow model of execution is one step toward that goal; while it did not model multicore computers as accurately as was hoped, it nevertheless captures some of the important elements.

ConTouR has demonstrated that multi-threaded schedulers can be written. It has also explored a unique approach to reduction in that it builds the schedule incrementally in much the same way that clusterers do.

Finally, a comparison of reducers has been provided which, for the first time, evaluates the reducers on actual programs, rather than relying solely on graph makespans. It was seen that LB

and CM perform the best in general, allowing for their continued use in CIRA and other scheduling tools.

REFERENCES

[1] Penry, D. A., 2009. "Multicore diversity: A software developer's nightmare." *Operating Systems Review,* **43**(2), Apr, pp. 100–101. 2

[2] Sarkar, V., 1987. "Partitioning and scheduling parallel programs for execution on multiprocessors." PhD thesis, Stanford, Stanford, CA, April. 3, 9, 17, 20

[3] Sarkar, V., and Hennessy, J., 1986. "Compile-time partitioning and scheduling of parallel programs." In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*, Vol. 21(7), pp. 17–26. 4, 11, 13

[4] Sriram, S., and Bhattacharyya, S. S., 2000. *Embedded Multiprocessors: Scheduling and Synchronization*. Marcel Dekker, Inc. 7

[5] Saltz, J. H., Mirchandaney, R., and Crowley, K., 1991. "Run-time parallelization and scheduling of loops." *IEEE Transactions on Computers,* **40**(5), May, pp. 603–612. 8

[6] Rehme, K. D., 2009. "An internal representation for adaptive online parallelization." Master's thesis, Brigham Young University, Provo, UT, August. 8, 40

[7] Penry, D. A., Richins, D. J., Harris, T. S., Greenland, D., and Rehme, K. D., 2010. "Exposing parallelism and locality in a runtime parallel optimization framework." In *Proceedings of the 7th ACM International Conference on Computing Frontiers*, pp. 117–118. 8

[8] Hu, T. C., 1961. "Parallel sequencing and assembly line problems." *Operations Research,* **9**(1), January–February, pp. 22–35. 9

[9] Kwok, Y.-K., 1997. "High-performance algorithms for compile-time scheduling of parallel processing." PhD thesis, Hong Kong University of Science and Technology. 9

[10] Wu, M.-Y., and Gajski, D. D., 1990. "Hypertool: A programming aid for message-passing systems." *IEEE Transactions on Parallel and Distributed Systems,* **1**(3), July, pp. 330–343. 9, 11

[11] Adam, T. L., Chandy, K. M., and Dickson, J. R., 1974. "A comparison of list schedules for parallel processing systems." *Communications of the ACM,* **17**(12), December, pp. 685–690. 10

[12] Hwang, J.-J., Chow, Y.-C., Anger, F. D., and Lee, C.-Y., 1989. "Scheduling precedence graphs in systems with interprocessor communication times." *SIAM Journal on Computing,* **18**(2), April, pp. 244–257. 10

[13] Sih, G. C., and Lee, E. A., 1993. "A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures." *IEEE Transactions on Parallel and Distributed Systems,* **4**(2), February, pp. 75–87. 10

[14] Yang, T., and Gerasoulis, A., 1994. "DSC: Scheduling parallel tasks on an unbounded number of processors." *IEEE Transactions on Parallel and Distributed Systems,* **5**(9), September, pp. 951–967. 12

[15] Liou, J.-C., and Palis, M. A., 1988. "A new heuristic for scheduling parallel programs on multiprocessor." In *Proceedings of the 7th International Conference on Parallel Architectures and Compilation Techniques*, pp. 358–365. 12

[16] Kwok, Y.-K., and Ahmad, I., 1996. "Dynamic critical path scheduling: An effective technique for allocating task graphs to multiprocessors." *IEEE Transactions on Parallel and Distributed Systems,* **7**(5), May, pp. 506–521. 13

[17] Liou, J.-C., and Palis, M. A., 1997. "A comparison of general approaches to microprocessor scheduling." In *Proceedings of the 11th International Parallel Processing Symposium*, pp. 152–156. 14, 15, 16, 34, 44

[18] Kwok, Y.-K., and Ahmad, I., 1999. "Benchmarking and comparison of the task graph scheduling algorithms." *Journal of Parallel and Distributed Computing*, March, pp. 1–35. 14, 15

[19] Yang, T., and Gerasoulis, A., 1992. "PYRROS: static task scheduling and code generation for message passing multiprocessors." In *Proceedings of the International Conference on Supercomputing*, pp. 428–437. 14

[20] Penry, D. A., 2006. "The acceleration of structural microarchitectural simulation via scheduling." PhD thesis, Princeton University, Princeton, NJ, November. 43

[21] Woo, S. C., Ohara, M., Torrie, E., Singh, J. P., and Gupta, A., 1995. "The SPLASH-2 programs: Characterization and methodological considerations." In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA)*, pp. 24–36. 43