Optimizing Task Distribution and Shared-Variable Accesses in an Asynchronous-Partitioned-Global-Address-Space Programming Model

by

Jeeva S. Paudel

A thesis submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Computing Science

University of Alberta

Abstract

High-performance programming systems employ a wide range of techniques to improve the performance of parallel and distributed applications on large-scale machines. Such techniques include load balancing to reduce machine idle times, colocating tasks and related data to leverage data locality, and optimizing shared-variable accesses to reduce communication costs. These goals result in complex performance trade-offs that typically require programming systems to prioritize one over the others, thereby limiting opportunities for optimization.

This dissertation identifies a novel opportunity for load balancing, proposes a new approach for workload distribution, and presents a profiling-based framework to automatically select coherence protocols aimed at specific patterns of shared-variable accesses. These approaches strike a balance between the tight budgets for run-time optimization and the exposition of new opportunities to improve the running time of applications. A prototype designed to evaluate these ideas is integrated into the X10 programming system. An empirical evaluation of these ideas, using large applications with diverse patterns of parallelism and communication, indicates that they can be applied widely and that they have significant performance merits.

Preface

Parts of Chapter 3, 4, and 5 have previously appeared in workshop, conference and journal publications. These are collaborative work with authorship shared among myself, José Nelson Amaral, Olivier Tardieu, and Levi H. S. Lelis. I did most of the artifact implementation, experimental evaluation, data analysis, and writing for the papers. The other co-authors provided significant inputs for designing the experimentations, for developing the algorithms, and for improving the presentation of the papers.

This research was conducted over a period of five years, during which the X10 programming system went through several changes. The quality of the code generated by the X10 compiler and the ability of the X10 runtime to harness the computing power also evolved significantly. Therefore, the execution time for the baseline execution of several benchmarks and applications used in this thesis vary across different implementations used in different chapters. However, the baseline for the experiments within each chapter is consistent.

To my parents.

Acknowledgements

This dissertation is an outcome of continued support from many people. I express my sincere gratitude to you all.

1. My wife and my family, for your warm love and invaluable inspiration.
2. CDOL members Paul, Matthew, You, Ricardo, Carolina, Iain, Arnamoy, and Xunhao, for your lively paper discussions.
3. Steve Sutphen, for your substantial effort in setting up the computing platform for my experimental evaluations.
4. Christopher Dutchyn, for instilling in me the love of compilers and programming systems.
5. Levi Lelis, for your collaboration on the idea of workload distribution. Thank you for helping me design the experiments, validate the results, and refine the writing.
6. Olivier Tardieu, for your excellent insights that form the mainstay of the research ideas investigated herein.
7. Keshav Pingali, Duane Szafron, Paul Lu, and Bruce Cockburn, for agreeing to be on my examining committee and for your support in completing and delivering this dissertation.
8. Last and most important, my advisor, José Nelson Amaral, for your excellent mentorship. Thank you for teaching me the rigorous process of scientific research. Your questions and insights have been instrumental in producing this dissertation.

Funding for parts of this work have been provided by the Natural Sciences and Engineering Research Council of Canada, Alberta Innovates: Technology Futures (formerly iCORE and Alberta Ingenuity), and by the IBM Center for Advanced Studies at the IBM Toronto Software Laboratory.

Contents

List of Figures

List of Tables

List of Acronyms

PGAS	Partitioned Global Address Space
APGAS	Asynchronous Partitioned Global Address Space
UPC	Unified Parallel C
MPI	Message Passing Interface
IDA*	Iterative Deepening A*
UTS	Unbalanced Tree Search
AMD	Advanced Micro Devices
DMG	Delaunay Mesh Generation
DMR	Delaunay Mesh Refinement
MCP	Matrix Chain Multiplication
RA	Random Access
SLAW	Scalable Locality-Aware Work-Stealing Scheduler
DirCoPX	Directory Based Coherence Protocol for X10
CoMX	Coherence-Policy Manager for X10
GR	GlobalRef
X10RT	X10 Runtime
API	Application Program Interface
HPC	High Performance Computing
STM	Software Transactional Memory
AMR	Adaptive Mesh Refinement
WPS	Workload Partitioning and Scheduling
WIT	Work-item Tree
StraSa	Stratified Sampling
BLDM	Balanced Largest-First Differencing Method
AIDA*	Asynchronous Iterative Deepening A*
TDS	Transposition-table-driven Scheduling

Chapter 1

Overview

Efficient execution of data and compute-intensive applications on large-scale parallel and distributed machines is challenging. Tasks and data referenced by the tasks must be co-located in a node to localize data accesses. Tasks spawned by an application must be evenly partitioned to ensure even utilization of available nodes. Efficient data-access policies are essential to minimize communication across the nodes. Manually orchestrating data layout, task execution and communication, to address such challenges, hampers programming productivity. To this end, high-performance programming systems, such as Partitioned Global Address Space (PGAS) and Asynchronous PGAS (APGAS), provide a uniform programming model for local, shared and distributed-memory computing machines. They provide mechanisms to distribute data across address partitions, to create parallel tasks, to map the tasks to the address partitions, and to manage the consistency of shared data. The address partitions may belong either to the shared-memory nodes or to the distributed-memory nodes.

Existing PGAS and APGAS models bind parallel tasks, or threads, of computation to programmer-specified address partitions in order to exploit affinity between tasks and data allocated at the address partitions. However, not all tasks benefit from such a binding because they may be agnostic to the location of data. A good example is a task that encapsulates the data necessary for its execution. The tasks that are sensitive to the location of data are called *locality-sensitive* tasks and tasks that are either agnostic or that do not bear strong affinity to the location of data are called *locality-flexible* tasks. The lack of a distinction between locality-sensitive and locality-flexible tasks enforces a one-to-one mapping between tasks and address partitions. Such a mapping over-constrains the execution of tasks and precludes opportunities for balanced distribution of work across address partitions. This inherent limitation of PGAS and APGAS models is evident in languages, such as UPC, X10, Chapel and Habanero-Java. The lack of load-balancing mechanisms across address partitions in these languages further compounds the problem of even utilization of computing resources.

Many applications recursively process different data elements to generate additional data of substantially different sizes. A good example is state-space search. A mechanism for evenly partitioning data in such applications must account for the size of dynamically generated data. Both modern and conventional high-performance programming models, such as the Message Passing Interface (MPI) programming models, PGAS and APGAS, lack such a mechanism, and cause unbalanced distribution of data. Techniques to dynamically alleviate load imbalances do exist in PGAS and APGAS systems. However, they initiate actions only after the occurrence of a load imbalance and operate only within address partitions. The motivation for this restriction is in part to encourage tasks to exploit their affinity to data allocated within the address partitions.

Shared variables are fundamental abstractions in high-performance programming systems. They typically encapsulate a significant amount of frequently accessed data. Efficient coherence protocols are essential to ensure consistency of shared-variable accesses. Implementations of popular PGAS and APGAS languages, such as UPC, X10, and Chapel, typically maintain a single global copy of each shared variable at the site of allocation. The tasks co-located with the shared variables in the same address partition can access the variables directly. However, such a protocol incurs the overhead of message transfers for all accesses to shared variables in remote address partitions. Alternative coherence protocols that employ replication of shared variables across address partitions are known to address this limitation. Although such protocols have been extensively studied in traditional distributed-memory systems, they have not been investigated in PGAS and APGAS systems.

1.1 Research Goals

A preliminary investigation offered us insights into these limitations inherent in the PGAS and APGAS models. These insights led us to pursue the research questions listed below in the context of interesting real-world applications. This study investigates the problems in the context of a specific programming system, *i.e.*, X10 — a popular realization of the APGAS model.

- Would a flexible mapping between locality-flexible tasks and address partitions yield performance gains by enabling task migration across address partitions? Task migration for load balancing entails identifying tasks to migrate, the timing of migration, and the remote node that is starving for work. Migrating locality-sensitive tasks hinders locality of data access and only adds to these overheads, potentially leading to serious performance degradations. Therefore, an efficient task-migration strategy must selectively migrate only locality-flexible tasks, and must amortize these overheads through improved utilization of computing resources.

- By accounting for the variances in the size of dynamically generated data, can a domain decomposition yield an even distribution of data throughout program execution? A static partitioning scheme specified at the start of program execution may soon result in largely imbalanced data across address partitions. Dynamically re-distributing data after each phase of data generation incurs large execution-time overheads. Thus, an efficient distribution mechanism must be able to predict the amount of data generated upon processing initial data elements with reasonable accuracy without actually running an application. If the prediction is feasible, can such a strategy be encoded into the programming system for use in a diverse range of applications?

- Can replication of data across address partitions yield better performance than maintaining a unique copy of the data at the site of allocation? Replication enables concurrent reads and local accesses to data, but requires complex mechanisms to ensure consistency. Maintaining a unique copy of data does not require coherence mechanisms, but incurs expensive message transfers for all remote data accesses. Therefore, an efficient data-management technique must adaptively employ one or more of these strategies depending upon how the data is accessed.

1.2 Contributions

The primary contributions of this dissertation include:

- A careful study of the interplay between parallel tasks and the scheduling framework in the X10 programming system leading to the key observation that locality-flexible tasks are key to efficient inter-node load balancing in distributed-memory systems. We investigate two orthogonal strategies for load balancing. The first is a *work-stealing* strategy where a node starving for work attempts to steal work from a node with surplus work. The second is a *work-dealing* strategy where a node with surplus work spreads work to nodes searching for work. We devised an effective heuristic to identify a node that is starving for work and also to determine the timing of task migration. This heuristic approximates the work-load status of a node by analyzing its history of searching for work among its peers. Experimental evaluation indicates that permitting only locality-flexible tasks for work stealing and work dealing significantly alleviates load imbalances while minimizing the performance penalty of migration itself.

- A novel adaptation of an existing approach to sampling for partitioning data. This technique systematically accounts for data recursively generated upon processing initial data. This approach processes a small amount of data to predict, with reasonable accuracy, the amount of data that will be generated by an application at runtime. Programmers do not need to manually code this solution for individual applications. A prototype implementation in X10 demonstrates that the approach can be integrated into the runtime of a programming system to support diverse applications. The even data distribution resulting from this approach yields significant performance gains.

- A practical framework for the automatic selection of a high-performing protocol among available ones to manage accesses to shared data. The framework underlying this study monitors and analyzes reads and updates of each shared data in an application to make the selection. We incorporate a directory-based protocol into the runtime system of X10. The protocol replicates shared data at the accessing nodes and maintains consistency of data copies. This protocol complements the existing protocol in X10 that keeps a unique copy of shared data and relies on message transfers for all remote accesses. Empirical evidence suggests that coordinating strategies that replicate data with the one that maintains a unique copy of the data is essential to improve the performance of shared data exhibiting diverse access behaviours. Experimental evaluation also suggests that this approach yields performance comparable to that of carefully hand-tuned applications.

1.3 Outline

The rest of this dissertation is organized as follows: Chapter 2 presents a primer on the Asynchronous-Partitioned-Global-Address-Space model, the X10 programming system, and the coherence of shared variables in multiprocessor systems. Chapter 3 proposes selective locality-aware task migration and investigates its feasibility and performance merits. Chapter 4 describes a framework for the selection of augmented coherence protocols for optimizing different patterns of shared-variable accesses. Chapter 5 presents a novel adaptation of an existing statistical technique of stratified sampling for workload partitioning. Chapter 6 summarizes the contributions and presents avenues for future work.

Chapter 2

Preliminaries

This chapter discusses important concepts underlying the Asynchronous-Partitioned-Global-Address-Space model, the X10 programming language, and the coherence of shared variables in multiprocessor systems.

2.1 The APGAS Model

The Partitioned-Global-Address-Space (PGAS) model offers a unified concurrency mechanism for both intra- and inter-node parallelism. It allows distribution of data and mapping of tasks to different address partitions. It also offers mechanisms to encode affinity between tasks and location of data. However, the PGAS model suffers from two major drawbacks. First, all existing languages in the PGAS family — Titanium [39], UPC [30], and Co-Array Fortran [57] — follow the Single Program Multiple Data model of execution, where all tasks execute the same program across machines. They implicitly assume that all processes run on similar hardware. Second, they lack support for dynamically spawning multiple parallel threads of computation. Consequently, handling non-data-parallel applications, which requires dynamic load balancing, is difficult in the PGAS model.

The Asynchronous PGAS (APGAS) model addresses such limitations by extending the PGAS model with the notions of asynchronous parallel tasks and abstractions for location of data. The work underlying this dissertation is in the context of X10 [22] — a popular realization of the APGAS model. Therefore, the next section discusses the core APGAS concepts of the X10 programming system.

2.2 X10

X10 is a high-performance, high-productivity programming system developed at IBM as part of the "Productive, Easy-to-use, Reliable Computing System" project. The X10 programming model is organized around the notions of *places* and *activities*.

2.2.1 Places

A place is an abstraction of shared, mutable data and threads operating on the data. It encodes the affinity between tasks and memory partitions. The number n of places available to an X10 program (0 to $n - 1$) and the mapping from places to nodes is specified by the user at launch time. The program starts by executing its main method at `Place(0)`. Other places are initially idle.

2.2.2 Activities

Every computation in X10 is an asynchronous activity, *akin* to a light-weight task, and runs in a place. The activities running in a place may access data located at that place with the efficiency of local access. An access to a remote place may take orders of magnitude longer and is performed using the `at (p) S` statement. An `at` statement shifts the control of execution of the current activity from the current place to place `p`, copies any data that is required by the statements `S` to `p`, and, at the end, returns the control of execution to the original place. The necessary data copying is done through runtime system calls inserted by the compiler.

X10's `finish` statement identifies the bounds of concurrency. All activities enclosed by a `finish` statement, including all nested activities, must complete before any statement subsequent to the `finish` can execute. X10's atomic blocks coordinate the mutation of shared data. The statement `atomic S` executes `S` in a single uninterrupted step. The conditional form of atomicity, `when(c) S`, executes `S` atomically when the condition `c` evaluates to true. X10 also supports dynamic barriers, called *clocks*, for synchronizing different phases of computation.

Figure 2.1 shows a snippet of X10 code. The statement `at (p) async` creates a new activity at place `p` to execute statements 4 and 5. The `for` loop ensures the creation of activities in all available places, as shown in Figure 2.2. The `finish` construct ensures termination of all activities before execution can proceed to statement 6.

```
S1:  val g = GlobalRef[Cell[Double]](new Cell[Double](0));
S2:  finish for (p in Place.places())
S3:  async at (p) {
S4:       val x = computeLocal();
S5:       at (g) atomic g()() += x;
S2:  }
S6:  val y = g()();
```

Figure 2.1: An example using `GlobalRef`.

2.2.3 Global References

An X10 variable resides in a single location: the place where it was allocated. A place cannot use or even directly refer to a variable in a different place. A special type, `GlobalRef[T]`, allows explicit cross-place references. The data encapsulated in `GlobalRef` is intended to be shared by all threads of computation across different

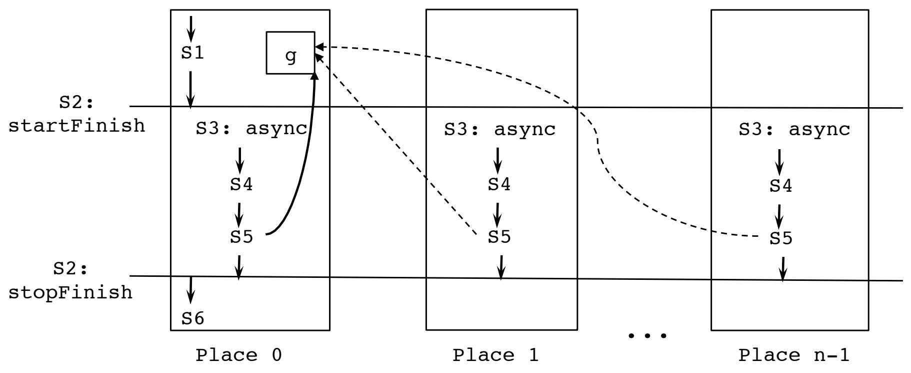


Figure 2.2: Activities, places and global references.

places. In Figure 2.1, `g` (line 1) is a `GlobalRef` containing a reference to a value `a` of type `Cell[Double]`. `GlobalRef[T]` is the only way to produce or manipulate cross-place references in X10. The `atomic` construct guarantees isolation from concurrent activities through a mutually exclusive access to `g`.

2.2.4 Distributed Arrays

X10 provides `DistArray`s to partition arrays across places. An underlying `Dist` object specifies which elements, or a range of elements should be allocated at different places. `Dist` uses subsidiary `Region` objects to abstract over the shape and dimensionality of arrays.

2.2.5 Load Balancing

X10 employs a work-stealing scheduler to address any runtime load-imbalances within places. The work-stealing scheduler uses a pool of threads, called *workers*, to run programs. Each worker maintains a private double-ended queue – a deque – of pending tasks. A worker primarily operates on its own deque, pushing one activity to the bottom of the deque for each `async` construct it encounters. When a worker completes one activity, it pops the next activity to run from its deque. If the deque is empty, the worker attempts to steal a pending activity from the deque of a randomly selected worker. Since each worker primarily interacts with its own deque, contention is minimal and only arises with load imbalance. Moreover, a thief tries to grab an activity from the top of the deque whereas the victim always pushes and pops from the bottom, further reducing contention.

Available X10 releases do not yet support work stealing across places.

2.3 Shared-Variable Coherence

Shared memory is integral to many high-performance programming systems. In a shared-memory system, processors may read and write to a single shared variable.

Unless care is taken, a processor may read a stale value of a variable, if multiple processors have access to multiple copies of the shared variable and at least one access is a write. Programming systems provide a set of rules, called *coherence protocols*, to ensure that multiple cached copies of shared variables are kept up-to-date. A read (or a write) access to a shared variable by a processor is referred to as a read hit (or a write hit) if the variable is available in the processor's local memory. If a processor cannot find the variable in its local memory, then the accesses is referred to as a read miss or a write miss.

Coherence protocols come in many variants but follow one of the following two policies when a processor writes to a shared variable.

Write Invalidate In the write-invalidate policy, a processor attempting to write to a shared variable initiates a coherence transaction to invalidate the copies in all other processors. Once the copies are invalidated, the requestor writes to the shared variable without the possibility of another processor reading the variable's old value. If another processor wishes to read the variable after its copy has been invalidated, it must initiate a new coherence transaction to obtain the variable, and it will obtain a copy from the processor that wrote it, thus preserving coherence.

Write Update In the write-update policy, a processor attempting to write to a shared variable initiates a coherence transaction to update the copies in all other processors to reflect the new value it wrote to the variable.

2.3.1 Coherence Protocol for X10's Global References

The X10 coherence protocol that manages accesses to shared data in `GlobalRef`s is herein called the *X10Protocol*. Values protected in `GlobalRef`s are retrieved by the application operation `g()`. The X10Protocol maintains a unique copy of a `GlobalRef` and dictates guarded access to `g()`; it can only be called when `g.home == here`, where `here` is the *id* of the referencing place. The `GlobalRef g`, shown in Figure 2.2, is directly accessible to activities in `Place 0` as shown by the solid black arrow.

Any operation, other than passing around a global reference or comparing two of them for equality, requires a place shift back to the home place of `g`, often with `at(g.home)`, or simply `at(g)` (line 5). For instance, all accesses to `g` in Figure 2.2 from places 1 to $n-1$ requires shifting the activity to shift to `Place 0`, as indicated by the dashed arrows. For all remote accesses that occur as a result of place-shifting `at`-clause, the X10Protocol exchanges messages between places.

2.4 Related APGAS Languages

Habanero-Java [17] and Chapel [19] belong to the same family of APGAS languages as X10. These languages differ in terms of their syntax and their support for different programming constructs. For instance, the syntax of Habanero-Java and X10 explicitly distinguish accesses to remote data from accesses to local data in order to keep the programmer aware of the costs involved. However, Chapel tends to make this distinction less explicit. If a Chapel task running on locale 0 (akin to a *place* in X10) wants to access a variable stored on locale 1, it can do so simply by naming the variable, relying on the compiler and runtime to provide the communication necessary to implement the remote reference. In contrast, X10 and Habanero-Java require programmers to explicitly specify the place of the remote data while accessing them.

Despite such differences, the fundamental mechanisms for mapping tasks, for managing the consistency of shared variables, and for workload partitioning are similar in these languages. They all suffer from the three fundamental limitations identified in this dissertation. Namely, the lack of a distinction between locality-flexible and locality-sensitive tasks limiting opportunities for load balancing, the support for naïve workload distribution resulting in a dynamic load imbalance, and the failure to employ augmented coherence-protocols to manage shared variables with different access patterns. As such, the conclusions drawn in this dissertation should be equally applicable to Chapel and Habanero-Java in addition to X10.

Chapter 3

Locality-Aware Task Migration

Balancing the computational workload of an application amongst available processors is critical for performance. A popular technique used to balance workload is task migration. Task migration assigns work to threads that are in idle or underloaded states for lack of work. An even distribution of workload reduces the number of processor cycles that are wasted on idle threads and improves performance. The resulting performance improvements are especially pronounced for tasks that perform significant amounts of computations and recursively generate more work for peer workers in a processor.

Task migration may have an adverse impact on performance. Migrating tasks may incur significant overhead. This overhead is due to bookkeeping and computation needed to determine the timing of task migration, the most favourable tasks for migration, and the best node to which a task should migrate [12, 29, 46]. The state explorations needed to make these decisions often require expensive data movement across the network and synchronization among multiple workers. Furthermore, task migration may also disrupt the affinity between tasks and data by launching tasks at idle or underloaded nodes rather than at nodes to which the tasks bear affinity. Therefore, many programming systems, including PGAS and APGAS, restrict task migration within shared-memory nodes to prioritize locality over load balance, and to minimize these overheads.

This chapter describes a novel insight that is key for the implementation of scalable task-migration strategies in distributed-memory systems: selective migration of tasks that are agnostic to the location of data strikes a balance between locality and load balance. We demonstrate this idea by implementing distributed work-stealing and work-dealing approaches to task migration in the X10 programming system.

3.1 Selection of Tasks for Migration

Completely avoiding the overheads of task migration in distributed-memory systems is impractical. However, there are measures that can help reduce and mitigate these overheads and thereby reduce the amount of time that processors are idle. The processors may remain idle because of memory or input-output operations, or for lack of work. Migration of some tasks contributes more toward minimizing these idle times than others. This section describes a task model to identify such tasks. A task is favourable for migration to a remote processor if the task meets any of the following conditions:

i) The processor's cache is warm for that task because the related data is already in its local memory. Therefore, there is no additional overhead in reloading a cold cache, or in copying the data from the remote processor.
ii) The task is local to the remote processor because it is a subtask of some task migrated to that processor. Thus, no extra cost needs to be paid.
iii) The task is coarse enough to overcome the cost of migration by keeping the target node busy for sufficient time to prevent frequent migrations.
iv) The task encapsulates the data necessary for its computation.

A task that qualifies for migration is a *locality-flexible* task. A task that bears strong affinity to a processor is a *locality-sensitive* task. This distinction frees a runtime scheduler from making load-balancing decisions for expensive locality-sensitive tasks and focus only on locality-flexible tasks. The designation of tasks as sensitive or flexible can be done in multiple ways: by an optimizing compiler, or by a runtime based on execution histories of similar tasks. It can also be done by a programmer using the algorithmic details and the semantic knowledge of the application, which may not be readily available through static or dynamic program analyses.

The proposition that locality information is beneficial is agnostic as to where such annotations come from. The prototype for the evaluation of this idea, presented in this chapter, relies on programmer-specified locality hints to identify locality-flexible tasks. This approach requires minimal changes to the X10 programming system.

3.1.1 Locality-Flexible Tasks in Applications

Following is a description of two parallel applications, Delaunay Mesh Generation and Turing Ring. These applications illustrate the existence of locality-flexible tasks in applications and the need for distinguishing such tasks from locality-sensitive tasks. Then a discussion of Iterative Deepening A* (IDA*) and Unbalanced Tree Search (UTS) provides examples of applications where all tasks are locality-flexible.

A task's designated node, as specified in its source code, is herein called the *source* node. The node where the task is launched after migration is called the *target* node.

Delaunay Mesh Generation A Delaunay mesh generator creates a mesh of Delaunay triangles from a given set of points. The algorithm starts by initializing a work-list with the points to be processed, and by initializing the mesh with a single large triangle encompassing all the points. A triangulation procedure then picks a new point from the work-list, determines the triangle containing the point, creates a *cavity* by splitting the triangle into three new triangles that share the point, and performs re-triangulation until all the new triangles are valid Delaunay triangles. Initially, there is only one large triangle. However, as the triangulation process unfolds, the mesh gets populated with several new triangles. The points in these newly formed triangles can be processed in parallel because the final mesh is the same regardless of the order in which the points are processed. Thus, the intermediate triangles can be distributed among different nodes for parallel and distributed processing. The triangles allocated in each node create other Delaunay triangles using the points they enclose. Therefore, allocating triangles and their encapsulated points at the same node is crucial to minimize remote memory references.

An even distribution of the initially generated triangles among nodes does not guarantee a balanced workload throughout the mesh-generation process. The number of points encapsulated by different triangles, and the number of re-triangulations needed for processing different points could be different. Some nodes might have to perform more triangulations than others. Therefore, a natural question arises: If a node is idle after processing all the points in its work-list, then would it be beneficial, performance-wise, to migrate a triangle with unprocessed points from other nodes?

The triangulation procedure is a task with the following properties: i) it encapsulates all the data necessary for its computation, eliminating the need for repeated remote memory references; ii) copying of the triangle and its points from the source node to the target node is necessary only once; iii) all the new triangles created by the target node have local access to other points in the triangle because they are already copied to the target node's local memory; and iv) it is coarse enough to keep the target node busy for sufficient time because it spawns several new tasks to process its points in parallel. Therefore, such a task is locality flexible and is favourable for migration to remote nodes.

Turing Ring The Turing ring problem simulates the interactions between predators and preys in a ring of cells. The algorithm (shown in Figure 3.1) initializes each `cell` with a number of predators and preys, and evenly distributes the ring across nodes. The algorithm iteratively updates the predator and prey populations accounting for their death, birth and migration. The population of predators and preys in each cell can be processed in parallel using `async` activities as shown in lines 5 and 7.

```
S1:  /* wl is a distributed ring of cells */
S2:  Worklist wl ← DistArray<Cell>
S3:  wl.initialize();
S4:  finish {
S5:      for each Cell c in wl async at (c.place) {
S6:            finish {
S7:                 async c.updatePreyPop();
S8:                 c.updatePredatorPop();
S9:            }
S10:           BodiesToMigrate mBodies = c.updateCellIDs();
S11:           wl.update(mBodies);
S12:      }
S13:  }
```

Figure 3.1: Pseudocode for the Turing Ring problem.

The populations of the cells can change by as much as two orders of magnitude in a single iteration resulting in a load imbalance. There are two types of tasks that can be migrated to alleviate the load imbalance: i) the task that updates predator or prey populations (lines 7 or 8). Migration of such tasks requires copying all information about predators/preys in the cell — including their birth, death and migration rates — to the target node to compute the new population. The new population must then be copied back to the source node to permit re-distribution of bodies; ii) the outer task that performs all operations in a cell, including population update and migration of predators and preys. Migration of such tasks requires copying of entire cells to the target node. However, once the cell is copied, there is no need to copy the results back to the source node because all other operations (lines 5 – 8) on the cell are now local to the target node. Hence, no more remote memory references will be needed. Further, the migrated task makes work available for other co-located workers in the target node. Thus, the outer `async` that processes an entire cell is a locality-flexible task that is suitable for migration during load balancing.

The costs of executing `async` (line 7), and `async at(p)` (line 5) are different. The `async` construct works on data that is already in the place where the task is created and does not need to create data copies. However, the execution of `async at (p)` creates copies of all the values used in S to the place `p`. The locality-flexible approach applies to `async at (p)`, *i.e.,* the scheduler is meant to migrate tasks operating on copies of data rather than place-resident data. Such tasks are precisely the ones that have poor locality, and hence, are annotated as *locality flexible.*

IDA* and UTS Not all applications possess easily distinguishable *locality-flexible* and *locality-sensitive* tasks. Consider IDA* and UTS algorithms for example. IDA* performs a cost-bound depth-first search of a state-space from a given start state using a heuristic function and finds the optimal solution path to a goal state. The disjoint parts of the state space in IDA* can be searched in parallel. The expansion of a state in IDA* requires only the cost-bound and can occur independent of expansions of other states. Thus, all parallel tasks in IDA* are *locality flexible.* Similarly, all tasks processing the nodes in UTS are also locality flexible because they do not bear affinity to data allocated at any single processor. However, the size of the subtrees generated upon expansion of different states in IDA* and different nodes in UTS can vary significantly. Thus, it is difficult to statically identify such tasks and decide *a priori* whether migration of the task in hand can offset the associated overheads to yield performance benefits.

In such applications, the runtime system may migrate any randomly selected tasks. Experimental evaluations show that the proposed scheduling algorithms do not degrade performance on such applications as well, which are not particularly suited to locality-aware scheduling.

3.2 Migration of Locality-Flexible Tasks

This work investigates the merits of locality-aware task migration by implementing distributed work-dealing and work-stealing schedulers. Under work dealing, the runtime scheduler dynamically adjusts the mapping of tasks to nodes to alleviate any load imbalance. Under work stealing, an idle worker in a node steals tasks from other co-located or remote workers. Both of these schedulers employ the same policy for task migration. A locality-flexible task may be migrated to an idle or an underloaded node only if the locality-flexible task's designated place is not idle or underloaded. These schedulers complement the existing work-stealing scheduler in X10 that operates only within shared-memory nodes.

3.2.1 Work Dealing

The work-dealing scheduler maps all *locality-sensitive* tasks to their designated places. The scheduler must decide if task migration is both necessary and beneficial only for *locality-flexible* tasks.

A common approach to make these decisions is to compare the number of tasks assigned, the number of tasks pending, or the expected completion times of the tasks in different places. Such an approach entails several runtime explorations and computations across all places. This work relies on a simple heuristic that a place must be *underloaded* if any worker in the place fails to successfully steal work after n consecutive attempts. In the prototype implementation, n is the number of workers per node. The idea is to exploit the principle that if an idle worker fails to retrieve work from its co-located workers in n attempts, then it is most likely that most of the workers in the node are either idle, and thus also searching for surplus work, or do not have any extra work to be stolen. Thus, it would be beneficial, in terms of load distribution, to map a dynamically spawned *locality-flexible* task to this node. By uniquely learning from the stealing operations to initiate task migrations to underloaded nodes, this work prevents expensive book-keeping operations.

Migrating a task to a remote node by starving workers in the task's designated place is counter-productive – it incurs the overhead of task migration and that of unnecessary work stealing within a place. Therefore, the scheduler maps a *locality-flexible* task `async(p)` to the task's designated place `p` if `p` is underloaded. Only if `p` is not underloaded, the scheduler proceeds to identify an underloaded node in the cluster. Using this strategy, the work-dealing algorithm prioritizes the utilization of cores over the utilization of nodes.

Control-Logic for Work-Dealing

Algorithm 1 describes the control logic for work dealing. The work-dealing scheduler checks the boolean variable `initiateWD` – initially set to false – to determine if there is any load imbalance in the system. An idle worker in a place sets `initiateWD` to true if it fails to steal successfully in n consecutive attempts. Multiple workers in multiple nodes may be attempting to steal in their home places. The scheduler tracks all such underloaded or idle places using `targetNode` – a boolean array.

The scheduler scans the `targetNode` to find the first underloaded place, say `q`, in the cluster, and deals out work to that place. In preparation for sending a task to the chosen underloaded place `q`, the scheduler first sets `targetNode(q.id)` to false to indicate that the place will no longer be underloaded because it is about to

Algorithm 1: Control logic for work dealing.

```
Require: A sequence of place-sensitive and place-flexible tasks
    $\langle$async$_1$(p), async$_2$(p), ..., async$_n$(p), $\rangle$.
Ensure: Locality-aware mapping of tasks async$_i$(p), and work dealing.
 1: for each place p$_i$ in the Program do
 2:   if local steal fails $n$ times consecutively then
 3:     atomic initiateWD $\leftarrow$ true
 4:     atomic targetNode(i) $\leftarrow$ true
 5: for each async$_i$(p) in the Program do
 6:   if isPlaceFlexible(async$_i$(p)) && !isTargetNode(p.id) && initiateWD
      then
 7:     for id in targetNode $\setminus$ targetNode(p.id) do
 8:       if (atomic targetNode(id)) then
 9:         atomic targetNode(id) $\leftarrow$ false
10:         initiateWD $\leftarrow$ targetNode.reduce(Or)
11:         create closure using body of async$_i$(p) activity
12:         prepare closure for remote execution
13:         create an async to run the closure at place(id); break
14:   else
15:     perform locality-guided mapping of async$_i$(p)
```

receive a task for execution. To decide whether work dealing needs to be employed on subsequently arriving `async`$_i$`(p)` tasks, the `targetNode` array is scanned to see if there are other places searching for surplus work and the `initiateWD` variable is set accordingly. Then, using the `async`$_i$`(p)`'s body, a closure is created. The closure is marked for remote execution and finally an `async` activity is created at the chosen place `q` to run the closure. A worker in the idle or the underloaded place performs the book keepings to minimize interruptions on the operation of busy places.

3.2.2 Work Stealing

Under distributed work-stealing, local and remote workers may attempt to retrieve tasks from deques at the same time. Frequent steal requests from remote nodes may interfere with the operation of local workers in a node, thereby, significantly degrading the performance of the local workers. Thus, the key to scaling distributed work-stealing implementations is to use data structures that minimize the contention among local and remote workers. This work maintains separate deques for locality-flexible and locality-sensitive tasks, and permits remote workers to steal only from the deques of locality-flexible tasks. Using multiple deques or specialized deques to reduce contention is a standard practice in scheduling [2, 3, 54]. The proposed distributed work-stealing algorithm maintains a *shared* deque and per-worker *private* deques in each place, as shown in Figure 3.2.

Tasks that bear affinity to a place are mapped to its *private* deque, while locality-flexible tasks are mapped to the *shared* deque. Under this scheme, each worker can operate on its private deque without interruptions from remote workers. The

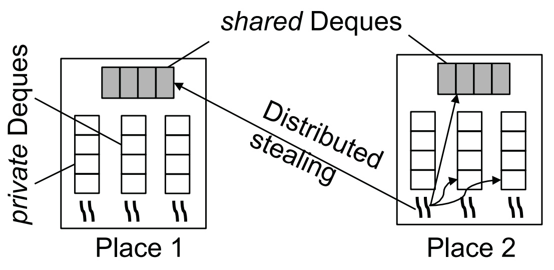


Figure 3.2: Private deques offer a place-local view of locality-sensitive tasks and a shared deque offers a global view of locality-flexible tasks.

last-in-first-out access policy of a worker's private deque leads the local worker to execute the most recently created task and thus, offers a higher chance of exploiting cache locality. The shared deque, in contrast, is manipulated in a first-in-first-out manner to ensure that any steal operation, whether local or remote, receives the oldest task in the deque. Older tasks potentially contain the largest amount of work in the task graph leading thieves to remain busy for a longer period of time and reducing the overhead of frequent steals.

Control-Logic for Distributed Work-Stealing

Algorithm 2 shows the control-logic for distributed work-stealing. Load imbalances within a place are managed by X10's intra-place work-stealing scheduler. The algorithm first attempts to steal from other co-located workers, then from the local shared deque, and finally from the remote shared deque. A worker attempts distributed work-stealing only when: i) all the other co-located workers are either busy executing their own activity and do not posses surplus work to be stolen, or are themselves attempting to steal work; and, ii) the local shared deque is empty. The algorithm checks the boolean variable *active* determine whether there are any running activities in a place. The variable is set to true when a worker starts processing work from its deque. An idle node has the variable set to false. The algorithm checks the number of *spare* workers and the total number of workers (*i.e., size*) in a node to determine if the node is underloaded.

Under distributed work-stealing, the thief — worker attempting a steal operation — holds a lock on the remote shared deque, retrieves an activity, creates a closure using the stolen activity, marks this closure for remote execution, and creates a new `async` activity at its home place to execute the closure. If the thief's attempt to steal from a remote worker fails, the thief first probes the network to see if any remote task has spawned tasks at its home place before continuing to explore other places. This approach eliminates unnecessary steal operations and also prevents polluting other place's caches. The thief continues this process until it finds work or until it has explored all available places.

Algorithm 2: Async mapping & distributed work stealing

```
Require: A sequence of locality-sensitive and locality-flexible tasks
    ⟨async_1(p), async_2(p), ..., async_n(p), ⟩.
Ensure: A mapping of tasks async_i(p) to private or shared deques, task
    execution strategy and distributed work stealing.
 1: for each async_i(p) in the Program do
 2:   if !isPlaceFlexible(async_i(p)) then
 3:     map async_i(p) to a private deque of place p
 4:   else if !isActive(p) || spares>0 || size(p)<max_threads then
 5:     map async_i(p) to a private deque of place p
 6:   else
 7:     map async_i(p) to the shared deque of place p
    // Control logic for local and distributed work-stealing policies
 8: activity ← poll the private deque
 9: while !activity and !terminated do
10:   activity ← probe the network for incoming tasks
11:   if !activity then
12:     activity ← steal from co-located workers
13:     if !activity then
14:       activity ← steal from the shared deque in its place
15:       if !activity then
16:         remoteVisits++
17:         activity ← steal from remote place's shared deque
18:         $p_{visited}$.add(remotePlaceVisited)
    // Distributed Work-stealing
19: for $p_j$ in places $\setminus p_{visited} \wedge p_i$ do
20:   activity ← atomically poll from the shared deque of $p_j$
21:   if !activity then
22:     create closure using the body of activity
23:     annotate activity to prepare it for remote execution
24:     async at ($p_j$) closure
25:   else
26:     break
```

3.3 Experimental Evaluation

This section describes the experimental setup and discusses the main findings from the performance evaluation.

3.3.1 Experimental Setup

Platform: Our performance measurements use a blade server with 16 nodes, each featuring two 2 GHz Quad-Core AMD Opteron processors, with 8 GB RAM, 64KB L1 cache, 512KB L2 cache, 2MB L3 cache, and 20 GB of swap space, running CentOS GNU/Linux version 6.0.

Compiler and Runtime: The `x10c++` compiler version 2.2 is used for measurements and the command-line arguments `-O -NO_CHECKS` are passed to the compiler to enable optimizations and disable array bounds, null pointer, and place checking. The nodes in the cluster are connected by an InfiniBand network with a bandwidth of 10 Gbit/s and use the MVAPICH2 library for communication. The experimental runs set X10_NTHREADS=8 to create eight worker threads per place and vary the number of places from 1 to 16 so that the number of threads is the same as the total number of cores available.

Applications: The experimental evaluation uses applications with unstructured data and irregular parallelism because they present significant opportunities for load balancing. These include the following applications from the Cowichan suite [59, 76]: i) *Quicksort* sorts an array of 100M elements using quick-sort; ii) *Turing ring* solves a set of coupled differential equations modelling system dynamics using 1M bodies; iii) *k-Means* implements a *k*-means clustering algorithm resulting in four clusters and using 1000 iterations; and iv) *n-Body* simulates the forces acting on a system of 220K bodies using the Barnes-Hut algorithm. The evaluation also uses the following applications from the Lonestar suite [44], that we ported from the Galois framework to the X10 language for distributed-memory systems: v) *Agglomerative clustering* performs clustering of 2M points by building a hierarchical tree in a bottom-up manner; vi) *Delaunay mesh generation* (DMG) deals with 2D Delaunay triangular mesh generation using 80,000 points; and vii) *Delaunay mesh refiner* (DMR) refines a Delaunay mesh of 550K triangles such that no angle in the mesh is less than 30°.

Methodology: Each application is run 10 times to account for variances, such as work-stealing in the runtime, and scheduling policies in the operating system. The performance charts do not show the 95% confidence intervals because they are small.

3.3.2 Results and Discussion

The evaluation compares the performance of eight different schedulers:

i) `X10WS` is X10's existing intra-node work-stealing scheduler.
ii) `WD` combines `X10WS` with the distributed work-dealing scheduler.
iii) `Eager-WD` proactively maps each *locality-flexible* task in a load-balance-aware manner rather than waiting until the occurrence of a load-imbalance.
iv) `DistWS` combines `X10WS` with the distributed work-stealing scheduler.

The relative merit of `DistWS` is justified only if it consistently outperforms distributed work-stealing that unconditionally permits any task to be stolen, and also does not incur the overhead of maintaining a shared deque per place. Thus, we compare performance of `DistWS` against:

v) `DistWS-NS-ND` allows a worker to steal any task directly from a remote worker's private deque and avoids the need for a shared deque per place.
vi) `DistWS-NS` also permits any task to be stolen, but maintains separate deques for locality-flexible and locality-sensitive tasks similar to `DistWS`. Making all tasks locality-flexible would mean mapping all tasks in an application to the shared deque. Such an approach incurs high overhead because local workers

must retrieve tasks only from the shared deque by competing with local and remote workers, instead of retrieving tasks from their private deques. Thus, for a fair comparison, `DistWS-NS` maps tasks to their designated place's per-worker deque until all workers are busy. A task is mapped to the place's shared deque if and only if all workers in the place are busy and its shared deque is empty.

vii) `DistWS-RS` is similar to `DistWS`, but permits stealing of randomly chosen tasks without any concern for the tasks' locality flexibility or sensitivity.

viii) `DistWS-PS` is similar to `DistWS`, but permits stealing of only a subset of locality-flexible tasks because not all locality-flexible tasks are annotated as such in the program source code.

An evaluation of these algorithms on applications with parallel tasks of varying granularities and communication patterns, using a varying number of worker threads, helps us understand:

The impact of thread and node count on speedup Figure 3.3 shows the sequential execution times of the applications, and Figure 3.4 shows the application speedups with different scheduling approaches relative to the sequential execution time. Execution over a single node present no opportunities for inter-node load balancing, but incurs extra costs to maintain separate deques, to schedule locality-flexible and locality-sensitive tasks separately, and to explore the runtime load-status to decide on a deque for the locality-flexible tasks. Thus, all algorithms except `DistWS-NS-ND` exhibit slowdown compared to `X10WS` at 1 – 8 worker counts. `DistWS-NS-ND` is an exception because it does not maintain a shared deque per place. When using multiple nodes, `WD` and `DistWS` outperform `X10WS`. `DistWS` outperforms `WD` for all cases except n-Body at 16 threads where they exhibit similar performance.

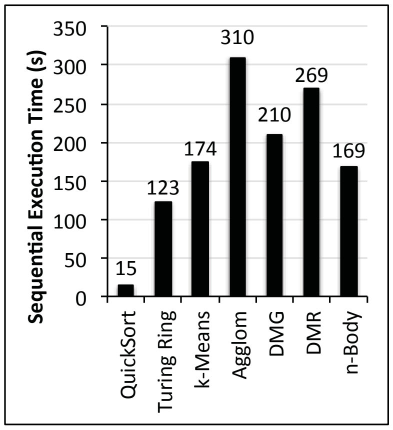


Figure 3.3: Sequential execution time using `X10WS`.

`WD` and `DistWS` exhibit larger impact at higher number of workers because the applications process, create or destroy work items — such as triangles and points — at a faster and a varied rate resulting in an imbalanced workload across different nodes. The best speedups achieved with `DistWS` over `X10WS` on Quicksort, Turing ring, k-Means, Agglom, DMG, DMR and n-Body are 14%, 12%, 18%, 28%, 32%, 28%, and

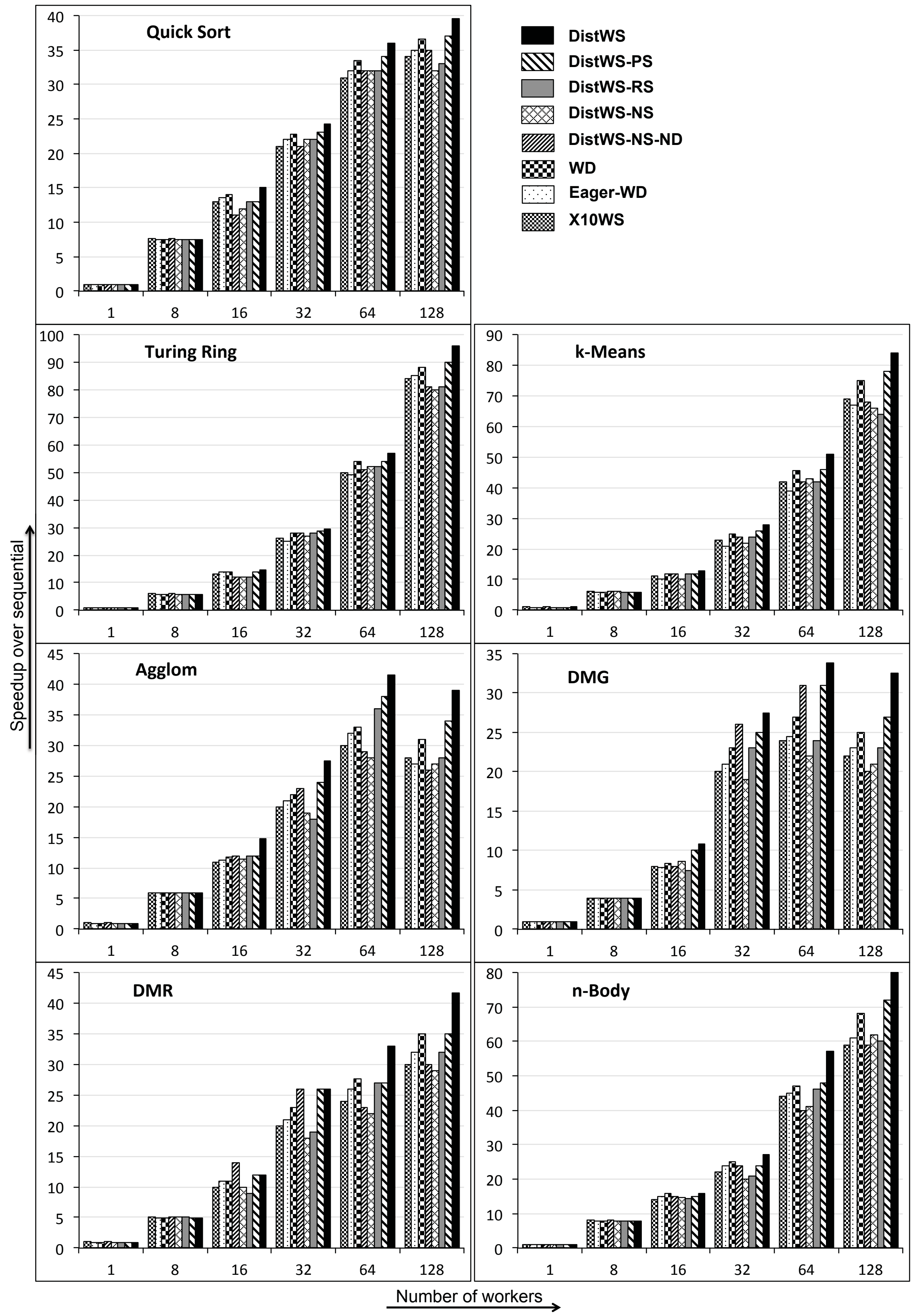


Figure 3.4: Speedup over sequential execution time.

26% respectively. Similarly, the best speedups achieved with `WD` over `X10WS` on these applications are 8%, 7%, 8.5%, 10%, 13%, 14%, and 13% respectively.

i) *Proactive and Reactive Work Dealing:* Instead of triggering distributed work-dealing after the occurrence of a load imbalance, the evaluation also investigates whether proactive work-dealing (`Eager-WD`) improves performance. While `WD` uses the knowledge of stealing to identify an underloaded or an idle node, `Eager-WD` must perform extra work to identify idle or underloaded nodes. `Eager-WD` must first scan all available nodes to identify the idle nodes so that it can send work from *locality-flexible* tasks to such nodes. If all nodes are busy, `Eager-WD` must compare the amount of surplus activities in each node to identify an underloaded node. These status checks and runtime explorations critically affect the performance of `Eager-WD`. Thus, it performs poorly compared to `WD`, and exhibits similar or marginal speedups ($< 4\%$) compared to `X10WS`.

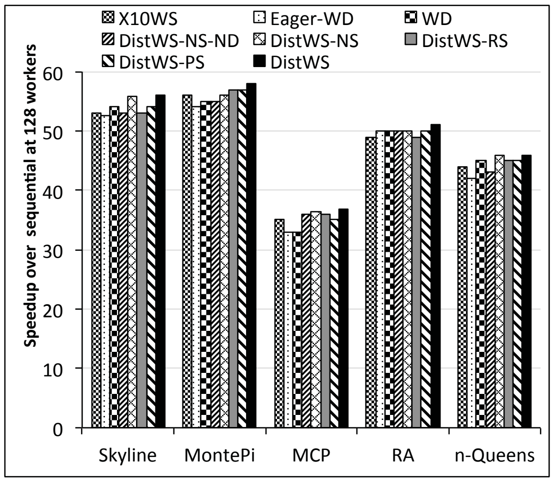
The impact of the amount of work migrated on performance The amount of work transferred during migration is determined by the granularity of the tasks migrated, measured as the time that it takes for a single thread to execute that task, and by the number and granularity of tasks spawned by the migrated tasks. Task granularity affects the performance of `DistWS` by influencing how long a thief node will perform useful work before requiring another steal.

A separate experimental evaluation investigated the performance impact of task granularities using smaller benchmarks: i) *skyline* solves a system of linear equations with 5000×5000 skyline matrix coefficients, ii) *montepi* computes Monte-Carlo estimation of π using 1M points, iii) *mcp* finds an optimal order for multiplying a chain of 5000 matrices, iv) *random access* (RA) updates integers at random memory locations in a distributed table; and v) *n-Queens* finds all different ways of placing n queens on a n x n board such that none of the queens hit any other queen in one move. The tasks in these applications are substantially finer-grained than the tasks in larger applications used in the experimental evaluation as shown in Table 3.1. The largest speedup relative to `X10WS` achieved on these smaller applications using different scheduling approaches is less than 6%, as shown in Figure 3.5. This supports the claim that only tasks that perform significant computation are suitable candidates for distributed work-stealing.

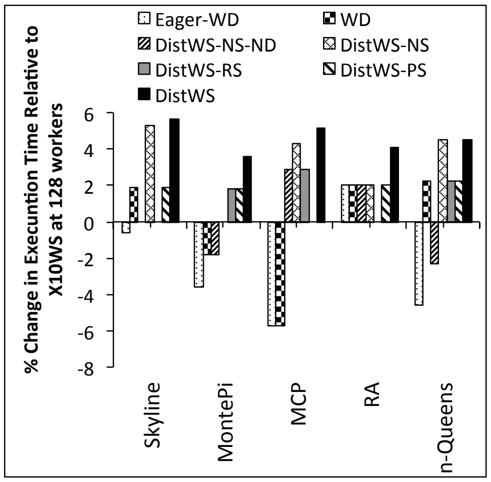
Table 3.1: Task granularities (in ms).

Quicksort	Turing Ring	k-Means	Agglom.	DMG	DMR	n-Body	Skyline	MontePi	MCP	RA	n-Queens
1.1	1.86	383	529	732	899	623	0.93	0.005	0.09	0.006	0.97

`WD` did not exhibit significant sensitivity to the granularity of tasks, hence, it is not discussed in this evaluation section.



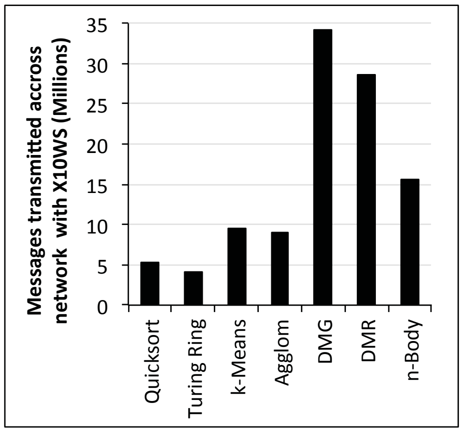
(a) Speedup over sequential at 128 workers.



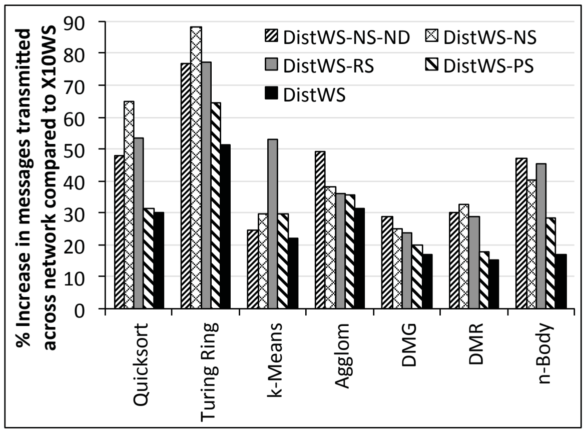
(b) Speedup relative to X10WS.

Figure 3.5: Speedup on applications with small task granularities.

The merit of migrating only locality-flexible tasks `DistWS` performs considerably better than `DistWS-NS-ND`, `DistWS-NS` and `DistWS-RS` as shown in Figure 3.4. `DistWS` strives to migrate only tasks that require minimum data transfer between the victim and thief nodes or the tasks that do not require frequent access to the remote data. `DistWS-NS-ND`, `DistWS-NS` and `DistWS-RS` in contrast, choose tasks for migration without such considerations. As a result, `DistWS-NS` transmits a significantly larger amount of data across the nodes than `DistWS`, as shown in Figure 3.6.



(a) Messages Transmitted with X10WS

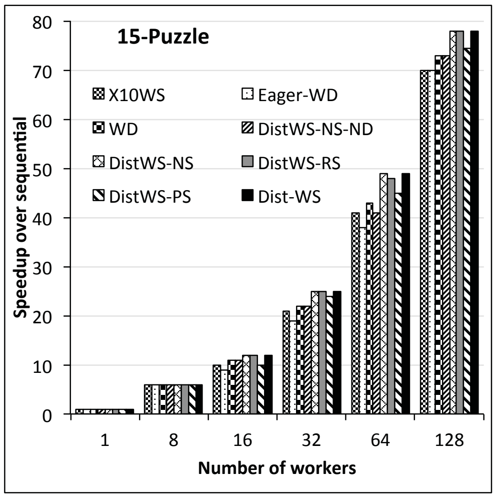


(b) Messages Transmitted Relative to X10WS

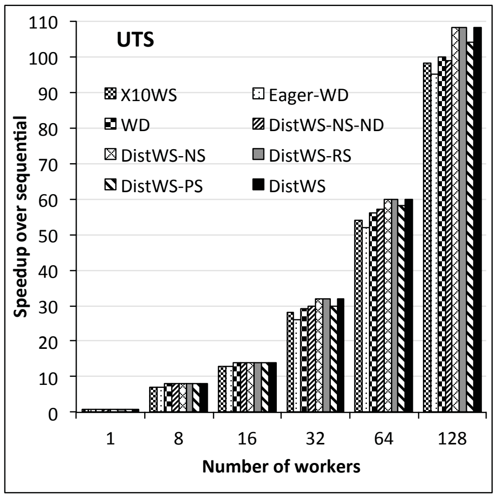
Figure 3.6: Messages transmitted across network at 128 workers.

The performance of proposed algorithms when only a sub-set of available locality-flexible tasks are exposed to the scheduler `DistWS-PS` represents the case where a programmer may annotate only some of the available locality-flexible tasks as such. The speedup graphs in Figure 3.4 show that `DistWS-PS` achieves at least the same performance as `X10WS` on all applications at 8 and 16 workers. At higher worker counts, `DistWS-PS` outperforms `X10WS` by a considerable margin. However, the performance gains are lower than that achieved with `DistWS`. This occurs because `DistWS` offers more scheduling freedom than `DistWS-PS` by exposing all available locality-flexible tasks to the load-balancing scheduler.

The performance impact of proposed algorithms on applications that are not suited to selective locality-aware scheduling All tasks in IDA* and UTS applications are locality-flexible but encapsulate significantly varying amounts of work. To evaluate the performance impact of locality-aware scheduling, IDA* was applied to solve the 15-puzzle problem, and UTS was used to expand nodes in a binomial tree of size 157 billion nodes generated on the fly using a split-table random number generator. In these applications, `DistWS` and `WD` permit migration of all tasks, and `DistWS-NS` permits migration of randomly chosen tasks. Both approaches yield similar speedups of up-to 10x over `X10WS` as seen from Figure 3.7. These results indicate the merit of the proposed approach even on applications that are not well suited for selective locality-aware scheduling.



(a) 15-Puzzle



(b) UTS

Figure 3.7: Speedup over sequential execution time at different worker counts.

3.4 Limitations

The proposed task-migration strategies must ensure that: i) all migrated tasks directly access only co-located data, and ii) results from the computation of migrated tasks are properly returned to the original place of the migrated tasks.

Ensuring Place-local Data Accesses Accesses to non-final variables in X10 are permitted only for objects residing at the same place as the `asyncs`. For instance, the code in Figure 3.8 creates a distributed array of one hundred elements distributed over all places via a block distribution with each element initialized to 0. The accesses to `distArray` in lines 8 and 11 of Figure 3.8 are valid for place `p1` because the array elements and their enclosing `async` (line 6) are co-located at place $p1$. If the async activity is launched at a remote place, say $p2$, after migration, then those accesses will no longer be valid. Fortunately, the X10's type system checks and identifies such non-local data accesses that may occur as a result of migration. An easy way to ensure that data accesses are local even after migration is to explicitly type-cast the array accesses using the `at` construct as shown in lines 9 and 12.

```
S1: val arrayDist:Dist = Dist.makeBlock(1..100 as Region);
S2: val distArray:DistArray[Int] = DistArray.make[Int](arrayDist,0);
S3: finish {
S4:    for (p in arrayReg) {
S5:      val p1 = arrayDist(p); val p2 = p1.next();
S6:      async (p1) {
S7:         // async (p2) {
S8:         val newVal = distArray(p) + x;
S9:         // val newVal = at(p1) distArray(p) + x;
S10:        Console.OUT.println("Array val at " + p.toString() + " "
S11:                                              + distArray(p));
S12:                                   // + at(p1) distArray(p)); }}}
```

Figure 3.8: An example illustrating the use of the `at` statement.

Returning Results from Migrated Tasks A side-effect of the copying semantics of the `at` statement is that all computations by tasks shifted to a new place occur on copies of the data within the scope of the `at` statement. Therefore, the expressions evaluated at remote nodes by migrated tasks must be returned to the original place by using the `at` statement.

3.5 Related Work

Our approach to task migration differs from prior work in the selection of tasks and nodes for migration, and the locality information used to guide migration.

Selection of Nodes Prior approaches use system state information [29], the processing speeds and relative workloads of the sender and receiver nodes [46], and the number of processes allocated to nodes [12] to select destination nodes for launching migrated tasks. Collecting such information may require communication among several nodes, and expensive synchronization operations. To the best of our knowledge, the idea of relying on work-stealing events to trigger task migration is original.

Selection of Tasks Several prior approaches support locality-aware scheduling. The COOL programming model allows programmers to specify locality hints for scheduling tasks and distributing objects [21]. The Scioto framework categorizes tasks based on their affinity to processors and encourages distributed steals of low-affinity tasks [27]. The Legion programming model permits user-guided placement of tasks [10]. Unlike `DistWS`, these approaches either forsake statically specified or derived locality-information during task migration or require the scheduler to migrate all tasks in a region if any task in the region is stolen.

Majo and Gross use execution histories to identify favourable tasks for migration [50]. In contrast, `DistWS` analyzes task granularities and the number of remote memory references to identify favourable tasks for migration.

Locality Information SLAW [36] and HotSLAW [54] use hardware topology to guide migration of tasks. Stealing from the nearest processor in the memory hierarchy is the key to their performance. Cong *et al.* and Guo *et al.* rely on the rate of work-stealing to coordinate multiple task-migration strategies [25, 35]. Unlike these approaches, this work uses application-level locality to guide task migration.

Task-migration strategy optimized specifically for an application also exists. The lifeline-graph-based load-balancing approach outperforms our approaches on the Unbalanced Tree Search (UTS) benchmark [70]. This is expected because they use a dedicated load balancer for UTS that first performs random stealing, followed by lifeline-based load balancing. When a node fails to steal, it quiesces and informs the outgoing edges in the lifeline graph. A failed attempt to distributed stealing in our approach does not help future steals. However, by remaining in a quiescent state after a failed steal and also by informing other nodes in its outgoing edges, lifeline graphs help reduce the overheads of failed steals. However, the authors do not consider how lifeline graphs could be made locality-aware.

Distributed Work-Stealing and Task Suspension Tardieu *et al.* demonstrate the applicability of work-stealing scheduling principles to X10's task parallelism with suspension [73]. Although their work-stealing engine supports multi-place programs, it balances the computations only inside places. They do not consider load balancing across places. This work indicates the potential for extending their recipes to distributed work-stealing as well.

3.6 Summary

This chapter argued that the selection of locality-flexible tasks is important for efficient load balancing. An experimental evaluation revealed that the coordination between X10's intra-node work-stealing and the proposed distributed task-migration approaches yields upto 32% speedup. The evaluation also revealed that the performance gains are more pronounced on applications with larger task granularities. Even applications that are not best suited for selective task-migration strategies do not lose performance with the proposed approaches. Another important contribution in this chapter is a heuristic based on intra-node work-stealing that cheaply detects dynamic load-imbalances.

Chapter 4

Optimizing Shared-Variable Accesses

Shared variables are fundamental abstractions for communication in high-performance programming systems. They encapsulate a significant amount of frequently accessed data. Efficient coherence protocols are essential to ensure consistency of shared-variable accesses. Prior studies employ caching [71], communication coalescing [24], and task-localization [7] techniques to improve the access performance of shared variables. However, they hinder productivity by only supporting programs written in specific ways, or by requiring programmers to annotate the access patterns of shared variables. Such approaches also fail to coordinate multiple protocols aimed at optimizing different patterns of accesses. As a result, they degrade the performance of applications that are not suited to the employed optimizations.

The key principle guiding the work in this chapter is that a coordination among multiple low-overhead coherence protocols is crucial for delivering a consistent and a scalable performance of shared-variable accesses. To this end, this chapter describes a novel framework for profile-directed selection of a high-performing coherence protocol, among available ones, for optimizing accesses to each shared variable in an application. This framework, incorporated atop X10, improves the performance of several applications by coordinating: i) the *X10Protocol* that maintains a single copy of shared variables, and ii) the Directory-based Coherence Protocol for X10 (*DirCoPX*) that replicates shared variables.

4.1 Shared-Variable Access Patterns

Some patterns of shared-variable accesses are prevalent in several applications. This section characterizes such commonly occurring patterns, and describes the techniques used in this work for detecting and optimizing them.

- **Migratory Pattern** Under the migratory access pattern, a variable is read and modified, in turn, by different nodes. Accesses to a shared variable fit the migratory pattern if: i) during a write-hit, the variable appears in a shared state with exactly two copies and the current writer is not the same as the last writer, or ii) during a write-miss on the shared variable, there is a single cached copy. In a write-invalidate coherence protocol, an access to a migratory variable causes a read-miss followed by an invalidation request to the node that last updated the variable. To reduce network contention, a run-time system can combine the read miss with the invalidation request (see section 4.2).

- **Producer-Consumer Pattern** Under the producer-consumer access pattern, a variable is updated by one node (producer) and subsequently read by an arbitrary number of other nodes (consumers). Accesses to a shared variable fit the producer-consumer pattern iff: i) the variable is written multiple times, ii) there is only one write sharer, and iii) there are multiple read sharers. The idea is that if a node repeatedly writes to a shared variable that is being read by multiple nodes, then the data accesses follow a producer-consumer pattern. The run-time system can *eagerly copy* such variables to the consumers. In the best case, the consumers never wait to receive the current values.

- **Stencil Pattern** Under the stencil pattern, each variable in a multidimensional grid is updated with weighted contributions from a subset of its neighbouring variables. Accesses to a shared variable fit the stencil pattern iff: i) the variable is organized as an array or rail [69], ii) there are accesses to multiple elements of the variable, iii) there are writes to each element of the variable, and iv) the variable's home node is in the read-sharers vector of adjacent nodes. In a stencil pattern the elements of the array are called points in the stencil. The idea is that if the stencil points are updated using neighbouring points, then that access to the array exhibits the stencil pattern. Replication of shared portions of remote arrays at the referencing nodes localizes the accesses and also avoids fine-grained data transfers across nodes. Such a strategy should update the local copy before a local read in the event of a write to the remote copy and vice-versa.

- **Read-Mostly Pattern** Under the read-mostly pattern, a variable is initialized once and subsequently only read, either sequentially or concurrently, by many threads. Accesses to a shared variable fit the read-mostly pattern iff: i) its sharing pattern does not fit the migratory, producer-consumer or stencil patterns, and ii) its read count is equal to or more than 1.5x its write count. A replication-based strategy that creates copies of the variables in each local node can leverage spatial and temporal locality to efficiently manage such variables.

- **Accumulator Pattern** Under the accumulator pattern, a variable is updated in a commutative and an associative manner using local values generated at each node. Unoptimized accumulator variables often entail repeated updates using values from other nodes. Such variables can be optimized using a collective reduction operation. Identifying the commutativity and associativity of operations on variables entails expensive data-flow analysis and cannot be guaranteed through profiling alone. Therefore, based on an empirical evaluation, this work employs the X10Protocol to manage such variables.

- **General read-write variables** Under the general read-write pattern, variables are read from and written to by multiple threads. Accesses to a shared variable that do not fall into any of these patterns are categorized as the general-write pattern. Replicating such variables reduces the read latency, but increases the write latency due to the added expense of updating or invalidating all their remote copies. Empirical evidence suggests that they benefit most from the X10Protocol.

4.2 Directory-based Coherence Protocol for X10

Directory-based Coherence Protocol for X10 (DirCoPX) is a new implementation of a traditional distributed directory-based coherence protocol for X10. To the best of our knowledge, this is the first implementation of a distributed protocol for X10. DirCoPX relies on replicating shared variables, allowing concurrent reads at many nodes, and reducing the number of coherence messages transmitted over the network to improve performance. A key ensuing challenge is to maintain consistency of the copies and to ensure that a read returns the most-recent value stored by a write. DirCoPX achieves consistency of variables using a write-invalidate policy.

4.2.1 Directory and Shared-Variable States

DirCoPX maintains an auxiliary data structure, called *directory entry*, for each shared variable to record one of the following states: i) *clean*: a coherent copy of the shared variable at its home; ii) *modified*: one and only one coherent copy of the shared variable that has been modified; and, iii) *shared*: a coherent copy of the shared variable that may be shared with one or more other nodes. Unlike an invalidation-based cache-coherence protocol, a shared variable cannot be invalid at the home node.

Directory entries in DirCoPX are distributed, with each node maintaining the entries for the shared variables allocated to its local memory. This distribution avoids a single hot spot in the machine. A directory entry comprises of:

i) *a bit vector* of n bits, where n is the number of nodes in the system. Each bit in this vector indicates whether or not the corresponding node has a copy of the shared variable in its local memory.

ii) *state bits* to indicate the state of the variable. A variable in the *modified* state, should have only one bit in the bit vector set because this is a write-invalidate protocol.

iii) *a lock bit*, which indicates, when set, that a coherency operation is pending and prevents other coherency operations. This synchronization is critical to avoid races.

A directory entry indicates the state of a shared variable globally. However, each node that has a copy of the shared variable must also record the state of the copy. The DirCoPX protocol requires that each shared variable copy be in one of the following possible states: i) *invalid*: the copy allocated in this node has been invalidated; ii) *shared*: there is one, or more, copies of the shared variable in the system and they all have the same value; and, iii) *modified*: the copy is the single valid copy in the system, *a.k.a* dirty.

This work is in the context of X10, thus the terms *shared variable* and `GlobalRef` (GR) are used interchangeably in further discussions.

Figure 4.1 illustrates directory entries for GRs allocated across nodes 0 to $n-1$, along with the states for GRs and directories. The directory entry for GR1 signifies that GR1 is in the clean state in node 0; thus, the bit n=0 in the bit-vector is set and the state of GR1 in its home node is clean. The entry for GR2 indicates that it is shared between node 0 and node $n-1$ and is in the shared state. A shared variable is created in the *clean* state with only the bit corresponding to its home node set.

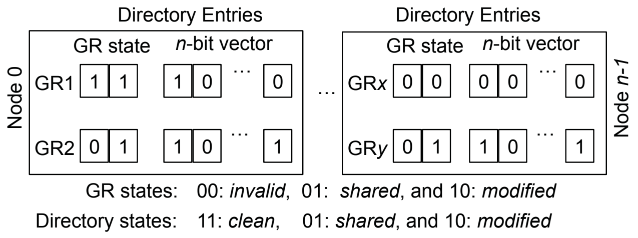


Figure 4.1: States maintained in distributed directories.

4.2.2 DirCoPX Optimization Strategies

DirCoPX employs two distinct strategies to optimize shared-variable accesses.

- When a shared variable is accessed by a single remote node N_i, DirCoPX first establishes that N_i has the variable in the modified state. Then, DirCoPX allows N_i to execute an indefinite number of reads and writes to the variable without any communication with other nodes. When multiple nodes make read-only accesses to a shared variable, DirCoPX requires each node to communicate with the variable's home node only once — to obtain a copy — and then, permits the nodes to complete an indefinite number of reads to the shared variable. This strategy achieves the desired optimizations (section 4.1) for read-mostly, producer-consumer, and stencil variables.
- When a shared variable is accessed by different nodes, and at each node the variable is used exclusively, each node experiences a read miss followed by an invalidation request. DirCoPX merges the invalidation request with the preceding read-miss request to reduce the number of coherence messages. This strategy achieves the desired optimizations for migratory variables.

	Directory State		
	Clean	Modified	Shared
Read Hit	Do Nothing	Do Nothing	Do Nothing
Write Hit	$\underline{invariant: A = H}$	Do Nothing	$A \xrightarrow{inv\ req} H$ $H \xrightarrow{inv\ req} C^*$ $C^* \xrightarrow{ack} H$ $\underline{H \xrightarrow{ack} A}$ $C^*: shd \mapsto inval$ $A: shd \mapsto mod$ $H: V[A] \leftarrow 1$ $H: V[C^*] \leftarrow 0$ $H: shd \mapsto mod$
Read Miss	$A \xrightarrow{req} H$ $\underline{H \xrightarrow{copy} A}$ $A: inval \mapsto shd$ $H: clean \mapsto shd$ $H: V[A] \leftarrow 1$	$A \xrightarrow{req} H$ $H \xrightarrow{wb\ req} C$ $C \xrightarrow{wb} H$ $\underline{H \xrightarrow{copy} A}$ $A: inval \mapsto shd$ $C: mod \mapsto shd$ $H: mod \mapsto shd$ $H: V[A] \leftarrow 1$	$A \xrightarrow{req} H$ $\underline{H \xrightarrow{copy} A}$ $A: inval \mapsto shd$ $H: V[A] \leftarrow 1$
Write Miss	$A \xrightarrow{req} H$ $\underline{H \xrightarrow{ack} A}$ $A: inval \mapsto mod$ $H: V[A] \leftarrow 1$ $H: clean \mapsto mod$	$A \xrightarrow{req} H$ $H \xrightarrow{wb/inv\ req} C$ $C \xrightarrow{wb} H$ $\underline{H \xrightarrow{copy} A}$ $C: mod \mapsto inval$ $A: inval \mapsto mod$ $H: V[A] \leftarrow 1$ $H: V[C] \leftarrow 0$	$A \xrightarrow{req} H$ $H \xrightarrow{inv\ req} C^*$ $C^* \xrightarrow{ack} H$ $\underline{H \xrightarrow{copy} A}$ $C^*: shd \mapsto inval$ $A: inval \mapsto mod$ $H: shd \mapsto mod$ $H: V[C^*] \leftarrow 0$ $H: V[A] \leftarrow 1$

Figure 4.2: Actions taken by each local operation on a node that is not the home place of the GR. *inval, shd, mod* and *wb* are short forms for *invalid, shared, modified,* and *writeback*, respectively. An *req* is a request.

4.2.3 DirCoPX Protocol Operation

DirCoPX is a fairly standard distributed coherence-protocol implemented in software. For completeness, this section describes the actions and state changes in the directory and the shared variables in the event of a write hit. Figure 4.2 then summarizes the actions and necessary state changes for other events on shared variables. The node where the shared variable is allocated by the X10 program will be henceforth referred to as the *home* node, and any other node that stores its copy as the *remote* node for that variable.

Protocol Table Figure 4.2 shows the actions and necessary state changes when a given event takes place. The table uses the following naming conventions: A is a remote node where an event occurs, H is the home node, C is a remote node that contains a copy of the GR, C^* indicates multiple remote nodes that have a copy of the GR. Above the horizontal line are the actions that occur when a given event takes place according to the state of the local copy in node A and the state of the GR in the directory. For instance, when a read miss occurs in node A and the state of the GR is clean, then A sends a request to H, and H responds by sending a copy of the GR to A. Below the horizontal line are the changes to the states. $A : inval \mapsto shd$ indicates that the state in A changes from *invalid* to *shared*. $H : V[A] \leftarrow 1$ indicates that the bit corresponding to node A is set in the bit vector of the GR in the home node H.

As an example, we describe the protocol actions for a *Write Hit*. A write hit to a variable in the *clean* state can only occur in the home node H, and there is no need for any state change because a variable in the *clean* state remains in the same state when modified by the home node itself. A write hit to a variable in the *modified* state does not require any protocol action because the remote node A is already the sole owner of the shared variable. A write hit to a *shared* variable requires that the remote node A send an invalidation request to the home node H. If the Lock bit of the GR is set, then the request must wait for that bit to be reset. The home node H will then i) set the lock bit of the GR to prevent other protocol operations from occurring until this one is completed; ii) send an invalidation message to all nodes C^* that have a shared copy; iii) wait for acknowledgment from all nodes C^*; iv) send an acknowledgement of the invalidation to the requesting node A; v) update the GR state to *modified*; vi) set the bit corresponding to the node that has the variable in *modified* state; and, vii) release the GR lock. Upon receiving the acknowledgement of the invalidation, the requesting node A transits this shared variable to the *modified* state.

4.2.4 Augmented DirCoPX for Migratory Data

For a GR that follows a migratory access-pattern, distinct nodes read and write the data in turn. When a new remote node accesses the GR, the last node that accessed the same GR has a copy in the modified state. Thus, under the base DirCoPX protocol, the read-modify-write access to the migratory data results in a read-miss followed by a read-exclusive request resulting in several request and acknowledgement messages over the network. The augmented DirCoPX protocol converts these two transactions into a single read-invalidate request initially sent to the home node. The following description of the extended protocol uses the same notations as in DirCoPX description.

When it is known that a GR follows a migratory access pattern, upon a read miss in the local cache of node A, the runtime of A can issue a *read-invalidate* request, which is sent to the home node H. H forwards the read-invalidate request to C, which holds a modified copy of the data, indicating that A is the new owner. To give up ownership of the GR, C sends the GR current value to A, sends an acknowledgement to H, and changes the status of the GR to invalid. H sends an acknowledgement to A indicating that the directory state has been updated. A changes its status to modified and proceeds with the read and write operations.

This optimization improves performance through: i) reduced write-stall time because the data is available at the requesting node before writing to it, and ii) reduced network contention because fewer invalidations are sent over the network.

4.3 Framework for Pattern Detection and Optimization

This work uses a profiling framework to detect the patterns of shared-variable accesses. The profiling framework operates in two phases. In the first phase, the framework monitors the access patterns of shared variables; and in the second phase, it analyzes the access pattern and annotates the variables with semantic hints for the runtime to select an available augmented protocol. The framework allows independent selection of a protocol for each shared variable.

An important related question is: how long should an application be profiled? Repeated profiling runs of the applications used in this study under different inputs indicated that their underlying behaviour of shared-variable accesses is independent of the input workload. Such applications can be profiled for the entire duration of their execution in the first phase and later, executed in the optimized mode using the annotations obtained from the profiles. The framework supports offline-profiling for such applications. The shared variables in such applications can be annotated using a single offline profiling run because the same coherence protocol yields the best performance for all inputs.

Some applications, however, exhibit different patterns of shared-variable accesses under different inputs. For example, a convex-hull computation exhibits different communication patterns for different input shapes, such as rectangles and triangles. Therefore, the framework also support online profiling. Online profiling for the entire duration of an application's execution may incur significant overheads. An empirical evaluation using the applications used in this study suggest that collecting profiling information until 5% of reads are processed ensures that: i) the execution of an application is past its initialization phase, ii) most of the shared variables have been accessed, and iii) widely spread apart reads and writes to a shared variable are also captured.

For the twelve applications used in the evaluation, the total number of reads corresponds directly to the number of particles, bodies, elements, triangles and iterations that is specified as input to the applications. The profiling framework, therefore, uses the values of such inputs to control the duration of online profiling.

4.3.1 Profiling Variable Accesses

The profiler gathers the following metrics: number of read accesses, number of write accesses, last writer, set of readers, and the set of writers to identify the usage pattern of a shared variable and then to annotate it with an appropriate protocol for coherence management. For each variable, the profiler maintains: i) a counter for the number of read and a counter for the number of write accesses, ii) bit vectors for read and write sharers – each bit indicating whether the n^{th} node reads or writes the variable, and iii) the last writer – an integer that stores, for every access to the variable, the identity of the node issuing a read-invalidate request.

4.3.2 Coherence-Policy Manager

With three different coherence policies – X10Protocol, DirCoPX, and augmented DirCoPX – available, the runtime system must decide which policy to use. A different policy may be selected to manage each individual GR. We designed and implemented a new coherence-policy manager, herein called *CoMX*, into the runtime system of X10 to switch between these policies to manage the coherence of shared variables. If new protocols or policies are introduced in the future, it will be trivial to extend CoMX to include them in its selection.

The X10 Runtime, called X10RT, provides various functions to declare, allocate, and update GRs. It offers `apply()` operators to return the pointer to a GR and also to return the value encapsulated in the GlobalRef type. Each GR has a *home* property indicating the node where it is allocated. CoMX extends the GlobalRef X10RT API to support additional properties for monitoring a GR's access pattern. CoMX updates the associated properties for each event corresponding to an allocation, a read, a write or an apply. CoMX also maintains a *protocol* field for each variable to indicate which protocol to use. At startup, each GR's protocol field is initialized to the X10Protocol.

The profiler analyzes the statistics gathered about each GR (in both offline and online mode) to identify its underlying sharing pattern and sets the protocol field in each shared variable with the suitable choice of the protocol.

We define functors — function-object classes — for X10Protocol, DirCoPX, and augmented DirCoPX in the C++ implementation of the `GlobalRef` struct. Guided by the value in the protocol field, the runtime creates a temporary instance of an appropriate functor object and passes it to the GR access functions. Depending upon the pattern denoted by a variable's property, the runtime manages the GR accesses and performs coherence actions using the suitable functor instance. Function-objects are as efficient as in-place code as they are usually expanded inline. An accumulator variable can be optimized with a reducible implementation only if it performs operations that are both commutative and associative. Partial runtime profiling alone is insufficient to correctly identify such operations. Therefore, CoMX currently employs the X10Protocol on accumulator variables. Overall, CoMX employes DirCoPX protocol for only four kinds of variables. Thus, the implementation needs to identify only these variables and fall back to the X10Protocol for all other cases.

4.3.3 Switching Between Coherence Protocols

Switching the coherence protocol from X10Protocol to DirCoPX or its augmented version is straightforward because there are no directory entries, GR states or additional properties to maintain in the X10Protocol. X10Protocol only manages a single copy of the GR allocated at its home.

In DirCoPX, and its augmented version, the home property is irrelevant because a GR may be accessible at non-home (i.e., remote) nodes as well. However, while switching from DirCoPX to the X10Protocol, it is necessary to ensure that the home node has the most recently updated value and that other copies maintained by remote nodes are no longer accessible to any other nodes. This condition can be reached within DirCoPX by the actions required for the home node to be the exclusive owner of the GR. This design ensures X10Protocol's principle of location-constrained access.

For all the applications used in the evaluation, there was no need to change back the state of a GR from DirCoPX to X10Protocol, only from X10Protocol to DirCoPX.

4.3.4 Relieving Programmers from the Coherence Burden

Other partitioned-global-address-space languages, such as Chapel, also require explicit locality-constrained access to shared variables. While accessing shared-variables, programmers must bear the onus of explicitly specifying the address partition where the shared variable is allocated and also of ensuring its node-local access. The CoMX framework shifts such a burden from programmers to the compiler and the runtime, which collaborate to offer location-agnostic accesses to the shared variables. Under the location-agnostic scheme, a node may also directly access remote data independent of any location constraints — the compiler will consider such accesses properly type-checked and the runtime will perform the actions required to provide coherence guarantees for the remote accesses.

This approach works out-of-the-box to optimize accesses to shared variables in X10 and precludes the need for programmers to understand and use specialized data structures or annotations. It is a more natural approach to optimizing accesses to shared variables than retrofitting `GlobalRefs` using Pragmas [34] or optimization approaches targeted at manually identified data-access patterns.

4.4 Experimental Evaluation

This section describes the experimental setup and discusses the main findings from the performance evaluation.

4.4.1 Experimental Setup

Platform: This evaluation uses a blade server with 16 nodes, each featuring two 2 GHz Quad-Core AMD Opteron processors, with 8 GB of RAM and 20 GB of swap space, running CentOS GNU/Linux version 6.2.

Runtime and Compiler: The nodes in the cluster are connected by an InfiniBand network with a bandwidth of 10 Gbit/s and use MVAPICH2 library for communication. The experimental runs create eight threads per place and vary the number of places from 1 to 16 so that the number of threads is the same as the number of cores. The `x10c++` compiler version 2.3.1 is used for all measurements and the command-line arguments `-O -NO_CHECKS` are passed to the compiler to enable optimizations and disable array bounds, null pointer, and place checking.

Table 4.1: Applications and their input data set.

Programs	Description	Input
Stream-EP	Embarrassingly parallel HPC Stream	5MB, 10K iters
k-Means	Distributed Lloyd's clustering (1000 iters)	8K pts, 8 centroid
MontePi	Monte-Carlo approximation of π	10 billion points
LinearReg	Best-fitting straight line through 2D points	10M points
Jacobi	Solution to finite diff. Helmholtz equation	10K*10K grid
Stream	HPC Stream Benchmark	2048*2048 array
Moldyn	Lennard-Jones potential on bodies	8788 bodies
UTS	Unbalanced Tree Search	156 billion nodes
N-Body	Force calculation using Barnes-Hut algo.	1M bodies
Agglom	Clustering 2D points (bottom-up manner)	10 M points
DMG	a Delaunay Mesh Generator	100K points
DMR	a Delaunay Mesh Refiner (30° minimum)	680K triangles

Applications: Table 4.1 shows the applications and input data used in the experimental evaluation. The applications use shared variables of different types and exhibit diverse communication patterns as shown in Figure 4.3.

Methodology: Each application is run twenty times to account for variances, such as work-stealing in the X10 runtime, and scheduling policies in the operating system. The variations are very small leading to 95% confidence intervals also very small, hence not shown in the performance charts for execution times and speedups.

The versions of the Operating System and the X10 Compiler used in this evaluation are different from the ones used in the evaluation in Chapter 3. Therefore, the baseline performance of the benchmarks differ across these two evaluations.

4.4.2 Results and Discussion

The premise of this work is that automatically identifying the shared-variable access patterns in applications will allow a compiler and its runtime to dynamically switch between the coherence protocols to improve performance. Thus, this experimental evaluation examines:

Performance gains from DirCoPX and CoMX The sequential execution times of the applications used in this study are shown in Figure 4.4. Sequential execution of UTS for very large trees requires excessive machine time, therefore, its sequential performance was measured using smaller trees. The command-line parameters used to generate the trees for sequential benchmarking of UTS are: binomial tree, t=0, r=559, b=2000, m=2, and tree size=57354859 resulting in a node expansion rate of 2.104 MNodes/sec. Olivier *et al.* describe the UTS input parameters in detail [58].

Table 4.2 shows the code restructurings used for hand-tuning, and Figure 4.3 shows the techniques used in each application. The hand-tuned applications leave the

						Hand Optimizations					
Programs	**GR**	**Reads**	**Writes**	**Pattern**	**S/ Ag**	**A**	**B**	**C**	**D**	**E**	**F**
Stream-EP	a	iter	2	Read	S	✓					
K-Means	a	iter*clusters*2	Same	General	Ag						✓
	b	iter*cluster*2	½ Reads	Read	Ag	✓					
MontePi	a	points	Same	Accum	S			✓			
LinearReg	a	nodes * 4	½ Reads	Read	Ag	✓					✓
	b	nodes	Same	Accum	Ag			✓			✓
Jacobi	a	nodes*gridSize*2 = iter_space*2	Same	Producer Consumer	Ag		✓				✓
	b	nodes*gridSize	1/5 Reads	Stencil	Ag				✓		
Stream	a	nodes*iters	2	Read	S	✓					
Moldyn	a	nodes*points* mdsize	0	Read	Ag	✓					✓
UTS	a,b,c	½ nodes	nodes	Write	S					✓	
	d	nodes* tasksPerPlace	Same	General	Ag						✓
N-Body	a	treeLevels	Same	General	Ag					✓	
	b	bodies	bodies*3	Write	Ag					✓	
	c	treeLeaves	Same	Write	Ag					✓	
	d	tree(Level+Leaves)	treeLeaves	Read	Ag	✓					✓
	e	treeLevels	treeLevel	Stencil	Ag	✓			✓		
Agglom	a,b	iters*clusters*2	Same	General	Ag						✓
	c	iters*clusters*2	None	Read	Ag	✓					✓
DMG	a	conflictingPoints << points	Same	Accum	Ag				✓		
	b,c	affectedTriangles	None	Read	Ag	✓					
DMR	a	affectedTriangles	Same	Producer Consumer			✓				✓
	b,c	overlapTriangles	nodes	Read	Ag	✓					

Figure 4.3: Shared variables in different applications. *iter, clusters, points, nodes, bodies, gridSize* refer respectively to number of iterations, clusters, points, nodes, bodies, and the size of grid allocated to each node. *S* and *Ag* refer to scalar and aggregate types. In N-Body, *treeLevels*, and *treeLeaves* are the number of levels and leaves in the oct tree. In DMG and DMR, *affectedTriangles* are the triangles that need to be re-triangulated and *overlapTriangles* are the triangles that overlap and cannot be processed in parallel.

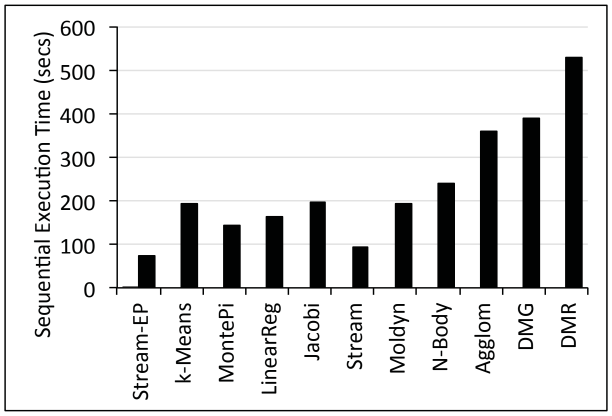


Figure 4.4: Sequential execution time using the X10Protocol.

migratory variables intact because DirCoPX's runtime policy of combining invalidation signals into read-miss requests could not be applied statically. Note that one code-transformation on a variable can lead to additional optimization opportunities. For example, an array of elements can be split into individual objects to reduce false sharing. The individual objects can be then replicated to localize accesses.

Table 4.2: Optimizations for hand-tuned applications.

	Access Patterns	**Restructurings**
A	Read mostly	Replicate node-local copies to reduce remote accesses
B	Producer Consumer	Eager copying to reduce synchronization time
C	Accumulator	Collecting Sum Reducer to reduce remote writes
D	Stencil	Replicate stencil arrays to reduce remote reads
E	General Read-Write	Keep variables intact to localize write accesses
F	Aggregate data	Split into smaller aggregate/scalar data to reduce false sharing

In Figure 4.5, the five bars corresponding to each application show normalized speedups relative to the X10Protocol at 128 workers obtained when using DirCoPX, random selection of protocols, CoMX, offline profiling, and the best hand-optimized implementation of the application.

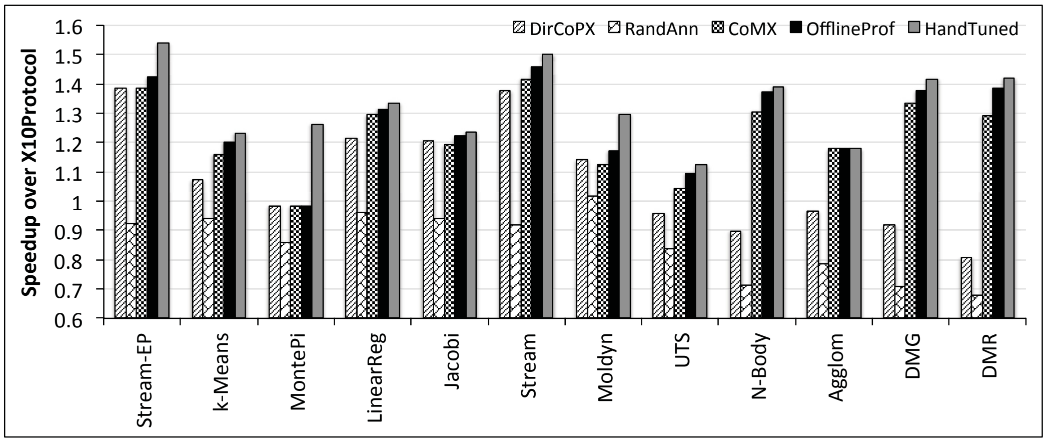


Figure 4.5: Application speedup over X10Protocol at 128 workers.

DirCoPX performs far superior than the X10Protocol for some applications (Stream-EP, k-Means, LinearReg, Jacobi, Stream, Moldyn). For others (UTS, N-Body, Agglom, DMG, DMR), it is necessary to combine both DirCoPX and the X10Protocol through CoMX to achieve a higher performance than the X10Protocol alone. For some (Jacobi, Moldyn), the overhead of the online CoMX leads to a marginally inferior performance compared to using DirCoPX exclusively. However, switching to Offline Profiling recovers the performance. For MontePi, exclusive use of the X10Protocol is the most performant solution. The hand-tuned versions have superior performance because this version applies optimizations that are not performed by CoMX: i) splitting aggregate variables to reduce false sharing; and ii) using sum reducers to reduce frequent remote accesses.

Source of Performance Gains Figure 4.6 shows the number of messages transmitted across the network when using the X10Protocol, the DirCoPX, CoMX, and the hand-tuned versions of the applications. For applications with write-dominant and general read-write variables, DirCoPX transmits the largest number of messages across the network in comparison to other versions. This increase in message traffic is expected because applications with a large write-to-read ratio suffer from the write-invalidate scheme of DirCoPX – DirCoPX sends several invalidation messages to read-only copies before a write can occur. However, when operating DirCoPX in tandem with X10Protocol, the applications send significantly fewer messages and achieve overall performance improvement.

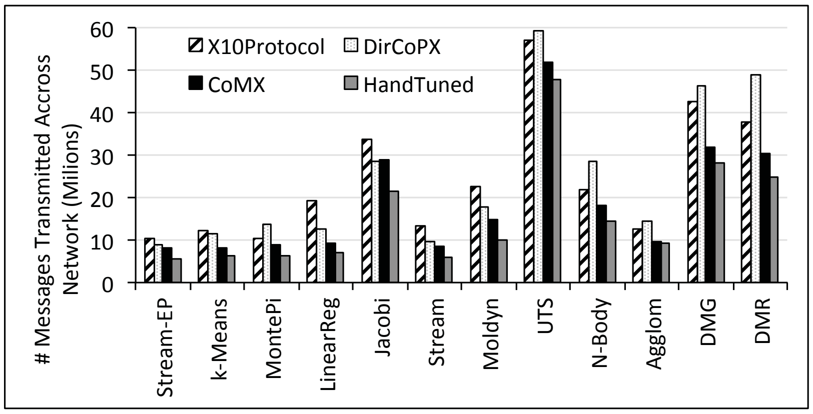


Figure 4.6: Messages transmitted across network at 128 workers.

Need for an automated pattern identifier Random selection of coherence protocols for managing accesses to shared variables significantly worsened performance of the applications studied, as indicated by the third bars in Figure 4.5, labelled *RandAnn*. The differences in speedups achieved using CoMX, which correctly employs the desired coherence protocol, and the random selection of protocols are stark: they range from 15% to 80%. These results justify the merit of the automated coherence-policy manager – CoMX.

CoMX's ability to identify access patterns Figure 4.3 shows the number of shared variables in each application, their access types and the number of reads and writes. With offline profiling, CoMX correctly identifies the access patterns of all variables in all applications. For the sake of comparison, we also employ online profiling. With online profiling, CoMX makes an incorrect decision about one shared variable in N-Body. The initial stage of N-Body computation is dominated by reads to the variable storing information about the bodies, but its later stages periodically rebuild the octree of bodies. Limiting the profiling duration to 5% of total reads precludes the profiler from observing writes to the variable in the later stage, thereby, incorrectly identifying the variable's access pattern as read-mostly instead of general read-write. CoMX employs DirCoPX to manage that variable instead of the more suitable X10Protocol. Even with this inaccuracy, CoMX outperforms X10Protocol by 30% by correctly identifying the patterns of other variables in N-Body (see Figure 4.5).

Need for coordinating DirCoPX and X10Protocol Employing DirCoPX instead of the X10Protocol improved performance of six of the twelve applications evaluated. Unfortunately, this approach also worsened performance of other six applications: MontePi, UTS, N-Body, Agglom, DMG, and DMR. These applications use variables with general read-write or write-dominant patterns (see Figure 4.3), which suffer from the replication-based strategy of DirCoPX.

Coordination between DirCoPX and the X10Protocol using CoMX improves performance of eleven out of twelve applications over the X10Protocol alone. MontePi's performance remains the same with both protocols because CoMX employs the X10Protocol to manage the only shared variable in MontePi. Although the variable exhibits the accumulator access pattern, CoMX chooses the X10Protocol for lack of enough information to prove the commutativity and associativity of the operations on the variable. Overall, the consistent performance gain over a wide range of applications, with no degradation on any of them, points to the merit of coordinating the two coherence protocols.

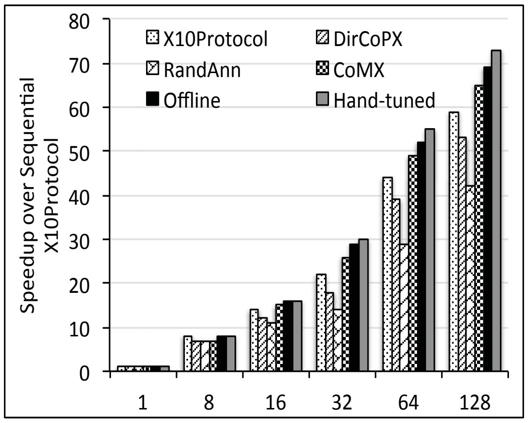
Online Profiling Overhead in CoMX Table 4.3 shows the memory overhead incurred by CoMX. The overhead is expressed in terms of the additional memory required per shared variable for storing: i) shared-variable access history. This includes the number of read/write accesses and patterns of accesses that are identified by the source and destination for each shared variable access; and ii) directory entry for each shared variable — that is, the bit vector. This overhead is acceptable (*i.e.*, < 40%) for most applications when collecting profiling statistics until 5% of total read accesses to shared variables have occurred. The overhead is as high as 63% for N-Body because it periodically creates new shared variables in different address partitions, which leads to greater number of copies and directory entries.

Table 4.3: Memory overhead of the CoMX coherence-policy manager (in %).

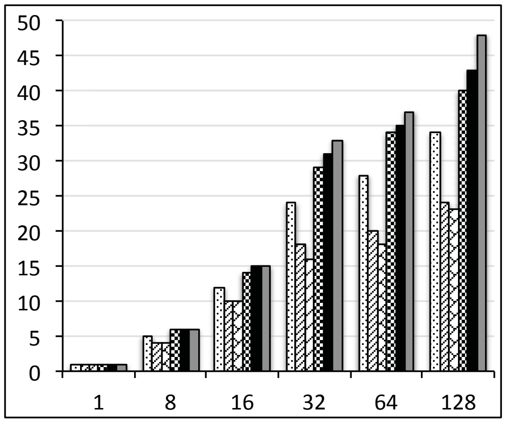
Stream-EP	k-Means	MontePi	LinerReg.	Jacobi	Stream	Moldyn	UTS	n-Body	Agglom	DMG	DMR
19	22	13	21	26	19	14.9	39	63	27	33	39

For applications that benefit from DirCoPX, the overheads incurred by CoMX lead to marginally shorter speedup bars for CoMX compared to that for DirCoPX, as seen from Figure 4.5. The overheads of online profiling in maintaining data structures to monitor usage of variables and in instrumenting their accesses can be avoided with offline profiling. The applications perform marginally better with offline profiling (see *OfflineProf* in Figure 4.5) compared to CoMX that employs online profiling.

Figure 4.7 shows that the overheads incurred by DirCoPX and CoMX do not inhibit the speedups achieved on N-Body and DMR applications at higher worker counts as well. Other applications too exhibit similar trend but are not shown for lack of space. This speedup trend bodes well for similar performance of our approach on larger clusters as well.



(a) N-Body



(b) DMR

Figure 4.7: Performance of different versions of N-Body and DMR at different worker counts (shown along x-axis).

4.5 Related Work

Prior work on optimizing access to shared data can be categorized as follows.

Distributed Shared-Memory Systems (DSMs) -IVY [49], TreadMarks [5], Munin [11], and DASH [48] – employed weakened memory model [41], distributed directory, and object-variable directory with multiple strategies – such as software release consistency, write-shared protocol and update-with-timeout mechanism – to reduce consistency-related communication [16]. They either supported a single coherence policy or required programmers to understand complex consistency models or to annotate shared variables with their access behaviour, and worsened performance for incorrect annotations.

Software Coherence Protocols PGAS languages, such as Chapel, maintain a unique copy of a shared variable and rely on one-sided communication for remote accesses. The XL UPC compiler uses a distributed symbol table, called *Shared Variable Directory* [9], and caches the remote addresses of shared objects to overlap communication and computation [31]. These approaches deliver good performance for some access patterns, but do not ensure low-latency for diverse patterns.

Transactional Memories Chapel and Fortress improve the performance of shared-data accesses through Software Transactional Memories (STMs) that (i) move data to a local node to localize write accesses [14, 38], and (ii) expose data accesses of parallel tasks to the scheduler to better manage conflicts [13]. This work is different in that it aims to reduce communication and latency through coherence protocols rather than using STMs to efficiently manage the conflicts.

Specialized Data Structures such as Multiphase Shared Arrays [26] and specialized variables [40] in Charm and Global Arrays [55] encode data-management policies into the data structures [74] and allow variable accesses in specific modes, such as read only, write once and accumulator. The restricted set of operations on variables simplifies consistency management and enables targeted optimizations. However, designing specific policies for data structures with diverse access patterns hinders productivity and requires programmers to use alternative mechanisms, such as message passing, for unsupported modes of accesses.

Optimization Techniques such as communication coalescing [4, 80], prefetching [15, 32, 72], and ghost zones [52] help reduce the overhead of shared variable accesses. However, efficient coalescing and prefetching techniques require addresses of the memory locations to be read *a priori*. Improper ghost zone sizes or prefetching from inaccurate addresses negatively impact overall performance. Our technique does not degrade the performance of any of the applications studied.

Application-Centric Approaches Wen *et al.* implement ghost regions in Titanium for Adaptive Mesh Refinement (AMR) [75]. They suggest that their approach may transfer to X10, but no implementation work has been undertaken. Claridge *et al.* implement exchange of irregular ghost regions for AMR in Chapel [20].

Hardware Cache Coherence Protocols are widely used to manage shared data accesses [1, 6, 18, 79]. For brevity, we consider two closely related works: Zebchuk *et al.* [79] propose tagless directory-based protocol that uses bloom filters to reduce space overhead. Unfortunately, false positives from the bloom filter lead to erroneous behaviours. For example, a write request for a shared variable can return a stale value if the owner does not yet hold a valid copy of the data because of false positives in the bloom filter. Acacio *et al.* [1] explore a directory protocol that uses prediction to find the current owner of the data during read and write misses. Locating an owner is easy in DirCoPX because the `home` property of `GlobalRef` identifies its home node.

4.6 Summary

Employing optimized coherence protocols for targeted patterns of shared-variable accesses improves application performance. The key underlying challenge is to automatically identify the patterns of communication that shared-variable accesses follow, without any explicit semantic hints from programmers. This work used offline profiling to automatically annotate variables with one of the two desired coherence-protocols — an existing static protocol and the directory-based protocol. Coordinating these protocols led to speedups on the applications studied in the range of 15% to 40% over the static protocol. These performance improvements, along with the productivity benefits resulting from this approach, should encourage other programming systems, such as Chapel, to invest effort in designing similar coherence protocols and protocol managers.

Chapter 5

Stratified Sampling for Even Workload Partitioning

State-space exploration is a fundamental algorithm in the domains of Artificial Intelligence and Combinatorial Optimization. A state-space exploration algorithm — or simply a state-space algorithm, such as A* [37] and IDA* [43], typically consists of a start state and a transition function that processes each state to generate a set of child states. The set of states, including the start state, generated by an application represents its state space. A state-space algorithm processes an application's state space to find a goal state, or a specific state that minimizes an objective function.

One way to achieve high performance in state-space algorithms is to process disjoint portions of states in different processing nodes of a cluster. However, it is difficult to foresee how many states will be processed to solve a given problem because state-space algorithms usually operate on an implicitly-defined state space. When an implicit definition of the state space is used, the current state produces a list of child states and each state in this list, in turn, recursively produces other child states as the algorithm progresses. Such algorithms also use techniques to reduce the effort of state-space exploration by avoiding processing of states deemed as unfruitful — *e.g.*, the algorithm uses heuristic functions to guide the search. The difficulty in predicting the number of states processed by such algorithms in each region of the state space leads to potential load imbalances in parallel-processing solutions.

This chapter presents a technique, called Workload Partitioning and Scheduling (`WPS`), for evenly partitioning the computational workload in an application. `WPS` samples a small portion of the application's state-space using the statistical technique of stratified sampling [23] to estimate the total number of states that will be processed by the application. `WPS` then uses such an estimate for partitioning the state-space of the application into sub-partitions with the goal of producing partitions whose sizes are as similar as possible.

Chapter 3 discussed generic work-stealing and work-dealing techniques for alleviating dynamic load imbalances. This chapter focusses on a class of applications — namely, recursive data parallel — and leverages their underlying algorithmic properties to minimize the dynamic load imbalances. As such, the proposed technique applies to a specific class of applications. This proposed technique yields better performance for such applications when compared to the task-migration strategies discussed in Chapter 3.

5.1 Preliminaries

This chapter refers to a state in an application's state-space as a *work item*, or simply an *item*. Let $S(n^*) = (N, E)$ be a *Work-Item Tree* (*WIT*) rooted at item n^*, representing the set of items processed by an exploration algorithm while solving n^*. N is a set of items and E is the set of edges in the tree. A *WIT* is formed by the items reachable from n^*. For each $n \in N$, $child(n)$ is the set of items generated when n is processed: $child(n) = \{n_i | (n, n_i) \in E\}$. We call $child(n)$ the children of n, and we call the edges $(n, n_i) \in E$ the actions available from n. In contrast with the Artificial Intelligence literature, a *node* refers to a processing node in a cluster, and not to a vertex in the *WIT*. Also, S is used to refer to $S(n^*)$ whenever n^* is clear from context. An item n is expanded when a computer node processes n. This work deals with implicitly-defined *WITs*, as described next.

Figure 5.1 shows the (3×3)-Sliding Tile Puzzle (8-puzzle), an example of a state-space exploration problem. For each state of this puzzle, there is a set of available actions. For instance, the state shown in Figure 5.1 (left) has three available actions: move tile 5, 6, or 7 onto the blank space. Moving a tile onto the empty space generates another state. The objective in this puzzle is to find the shortest sequence of actions that transform the given state to the goal state shown in Figure 5.1 (right).

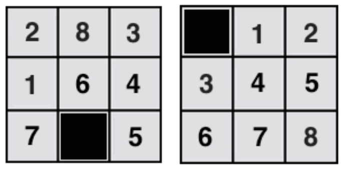


Figure 5.1: A random state (left), and the goal state (right) of the (3×3)-Sliding-Tile Puzzle, also known as the 8-puzzle.

The *WIT* of the 8-sliding-tile puzzle (as the other *WITs* we deal with in this chapter) is implicitly defined. That is, the *WIT* is not available a priori, but items in the *WIT* can be generated by applying actions from the initial item. *WIT*s of the 8-sliding-tile puzzle have at most 181,440 different states, which could be stored explicitly in memory. However, we are interested in problems that are too large to be stored explicitly in memory. For example, one of the application domains we use is the (4×4)-Sliding-Tile Puzzle, which has $\frac{16!}{2}$ different states.

5.1.1 Problem Formulation

Given M processing nodes in a computer cluster and an implicitly defined *WIT*, the *Work-Load Distribution Problem* consists in partitioning the items in the *WIT* into

M parts $W_1, W_2, \cdots, W_M$ of similar size. The goal is to minimize $\sum_{i,j \in \{1,\cdots,M\}} |W_i| - |W_j|$, where $|W_i|$ is the size of W_i. All items in the *WIT*s of the applications used in this study take approximately the same amount of time to process. However, the proposed algorithm could be easily adapted to deal with items that have different processing times.

In addition to being implicitly defined, the tree representing the *WIT* is often unbalanced, which poses a significant challenge for an even workload partitioning. The *WIT* is usually unbalanced because exploration algorithms use enhancements, such as a heuristic function [64], to guide the exploration to more promising parts of the state space (details in Section 5.4). As a result, the tree grows more quickly toward the directions deemed as promising by the algorithm.

Consider, for example, that an initial item produces two items i_1 and i_2, and that the computing cluster has two processing nodes. Given that the *WIT* is unbalanced, the size of the subtree rooted at i_1 might be very different from the size of the subtree rooted at i_2. A trivial solution of assigning i_1's subtree to one node and i_2's subtree to another node will lead to workload imbalance. The lack of *a priori* information about the sizes of the subtrees rooted at i_1 and i_2 further complicates the Work-Load Distribution Problem.

The `WPS` algorithm presented in this chapter uses the statistical technique of stratified sampling introduced by Chen [23] for quickly partitioning the *WIT* into parts of similar size.

5.1.2 Chen's Stratified Sampling

Knuth [42] presents a technique to estimate the size of the tree expanded by a search algorithm such as chronological backtracking. His technique repeatedly performs a random walk from the root of the tree. When all branches have the same structure, a random walk down one branch is enough to estimate the size of the entire tree. Knuth observed that his technique was not effective when the tree is imbalanced. Chen [23] addressed this problem by stratifying the search tree to reduce the variance of the sampling process. Chen's technique is herein referred to as *Stratified Sampling* (`StraSa`). `WPS` uses `StraSa` to estimate the *WIT* size and, based on such an estimation, it finds a partition of the items in the *WIT*.

Definition 1 (Stratification) *Let $S = (N, E)$ be a* WIT. *$T = \{t_1, \ldots, t_n\}$ is a stratification for S if it is a disjoint partitioning of N. If $n \in N$, $t_i \in T$ and $n \in t_i$, then $T(n) = t_i$ states that the stratum of n is t_i.*

`StraSa` is a general approach for approximating any function of the form $\varphi(n^*) = \sum_{n \in S(n^*)} z(n)$, where $S(n^*)$ is a *WIT* rooted at n^* and z is any function assigning a numerical value to an item. $\varphi(n^*)$ represents a numerical property of the search tree rooted at n^*. For instance, if $z(n) = 1$ for all $n \in S(n^*)$, then $\varphi(n^*)$ is the size of the *WIT*. Instead of traversing the entire *WIT* and summing all z-values, `StraSa` assumes that subtrees rooted at items of the same stratum have equal values of φ and thus only one item of each stratum, chosen randomly, is expanded. This selective expansion is the key to `StraSa`'s efficiency because trees of practical interest are too large to be examined exhaustively.

Given an item n^* and a stratification T, `StraSa` estimates $\varphi(n^*)$ as follows. First, it samples the *WIT* rooted at n^* and returns a set A of *representative-weight* pairs, with one such pair for every unique stratum seen during sampling. Given a pair $\langle n, w\rangle \in A$ for stratum $t \in T$, n is the unique item of stratum t that was expanded during sampling and w is an estimate of the number of items of stratum t in the *WIT* rooted at n^*. $\varphi(n^*)$ is then approximated by $\hat{\varphi}(n^*)$, defined as

$$\hat{\varphi}(n^*) = \sum_{\langle n,w\rangle \in A} w \cdot z(n)\,. \tag{5.1}$$

Algorithm 3 shows our adaptation of `StraSa` in detail.

Algorithm 3: `StraSa`, a single probe

```
Require: root $n^*$ of a tree and a stratification $T$
Ensure: a sampled tree $ST$ represented by an array of sets $A$, where $A[i]$ is
    the set of pairs $\langle n, w\rangle$ for the items $n$ expanded at level $i$, and an array of
    sets $C$, where $C[i]$ is the set of items generated at level $i$ but not expanded.
 1: $A[0] \leftarrow \{\langle n^*, 1\rangle\}$
 2: $i \leftarrow 0$
 3: while stopping condition is false do
 4:   for each element $\langle n, w\rangle$ in $A[i]$ do
 5:     for each child $\hat{n}$ of $n$ do
 6:       if $A[i+1]$ contains an element $\langle n', w'\rangle$ with $T(n') = T(\hat{n})$ then
 7:         $w' \leftarrow w' + w$
 8:         with probability $w/w'$, replace $\langle n', w'\rangle$ in $A[i+1]$ by $\langle \hat{n}, w'\rangle$ and
            insert $n'$ in $C[i+1]$; insert $\hat{n}$ in $C[i+1]$ otherwise
 9:       else
10:         insert new element $\langle \hat{n}, w\rangle$ in $A[i+1]$
11:   $i \leftarrow i + 1$
```

The set A is divided into subsets, one for every layer in the search tree; $A[i]$ is the set of representative-weight pairs for the strata encountered at level i. In `StraSa`, the strata must be partially ordered such that an item's stratum is strictly greater than that of its parent in the *WIT*. Chen suggests that this constraint can always be guaranteed by adding the depth of an item in the *WIT* to the stratification and then sorting the strata lexicographically. In this implementation of `StraSa` the depth of exploration is implicitly added to the stratification: strata at each tree level are treated separately by the division of A into the $A[i]$. If the same stratum occurs on different levels, the occurrences are treated as though they were of different stratum.

$A[0]$ is initialized to contain only the root of the *WIT* to be probed, with weight 1 (line 1). In each iteration (lines 4 – 10), all the items from $A[i]$ are expanded to get representative items for $A[i+1]$ as follows. Every item in $A[i]$ is expanded and its children are considered for inclusion in $A[i+1]$. If a child $\hat{n}$ has a stratum t that is already represented in $A[i+1]$ by another item n', then a *merge* action on

$\hat{n}$ and n' is performed. A merge action increases the weight in the corresponding representative-weight pair of stratum t by the weight $w(n)$ of $\hat{n}$'s parent n (from level i) since there were $w(n)$ items at level i that are assumed to have children of stratum t at level $i+1$. $\hat{n}$ will replace the n' according to the probability shown in line 8. Chen [23] proved that this probability reduces the variance of the estimation. Once all the states in $A[i]$ are expanded, `StraSa` expands the items in $A[i+1]$. This process continues until it reaches a level i^* where $A[i^*]$ is empty.

One run of the `StraSa` algorithm is called a *probe*. $\hat{\varphi}^{(p)}(n^*)$ is the p-th probing result of `StraSa`. `StraSa` is unbiased, i.e., the average of the $\hat{\varphi}(n^*)$-values converges to $\varphi(n^*)$ in the limit as the number of probes goes to infinity.

Chen [23] states the following theorem:

Theorem 1 *Given a stratification T and a set of p independent probes $\hat{\varphi}^{(1)}(n^*)$, $\cdots, \hat{\varphi}^{(p)}(n^*)$ from a* WIT *$S(n^*)$, $\frac{1}{p}\sum_{j=1}^{p}\hat{\varphi}^{(j)}(n^*)$ converges to $\varphi(S)$ as p grows large.*

Each `StraSa` probe outputs a subtree of the *WIT* called *sampled tree* (*ST*). In contrast with Chen's version of `StraSa`, our version of the algorithm also outputs an array of sets C containing the items encountered during sampling which were not expanded. C is organized by levels, e.g., $C[i]$ is the set of items `StraSa` encountered but did not expand at level i of the *WIT*. The `WPS` algorithm uses C to evenly divide the workload among different processing nodes, as described next.

5.2 `WPS`: Workload Partitioning & Scheduling

Algorithm 4 shows a high-level description of `WPS`. `WPS` operates in four phases: sampling, estimating, partitioning, and distributing.

Algorithm 4: Workload Partitioning and Scheduling

```
Require: starting item $n^*$ of the $WIT$ and a stratification $T$
Ensure: solution for the problem represented by $n^*$
1: $[A, C] \leftarrow StraSa(n^*, T)$ // see Algorithm 3
2: $\chi \leftarrow ComputeSubtreeSizes(A, T)$ // see Algorithm 5
3: $\{W_1, W_2, \cdots, W_M\} \leftarrow BLDM(\chi, C)$ // see [53]
4: for $i \in \{1, \cdots, M\}$ do
5:     asynchronously copy $W_i$ to node $i$
```

5.2.1 Sampling

In the Sampling phase, `WPS` employs `StraSa` on the *WIT* to selectively process only one among several items of the same stratum at each level of the *WIT*. Following this technique, this phase produces a sampled tree *ST* and a set C of items that were encountered but not expanded. The subtree *ST* is used to estimate the size of subtree rooted at items of different strata (see Section 5.2.2 below), while the items in C are partitioned amongst the available processing nodes according to the size of the subtrees provided by ST (see Section 5.2.3 below).

`WPS` offers programmers a customizable labelling system to define properties that constitute two items to be similar. For instance, in an Iterative Deepening A* (IDA*) search tree, two items may be considered to belong to the same stratum if their h-values, *i.e.,* their estimated cost to the goal node are equal.

Typically, increasing the number of `StraSa` probes will improve the accuracy of the tree size prediction. In `WPS` we use multiple probes to improve the prediction accuracy of the size of the subtrees rooted at items of different strata. In the partitioning phase, for stratum t encountered at level i, instead of using the Y_t^i value produced in a single probe, we use the average of the Y_t^i-values computed across multiple probes.

An important question is: how many probes are sufficient to yield best performance? An empirical evaluation indicates that a larger number of probes improves the accuracy of estimation, but incurs large sampling overhead. Likewise, a smaller number of probes with lower overhead may lead to poor estimates of the workload metric. This study used manual tuning to determine the number of probes that yields the best performance (details in Section 5.5.2).

The findings from this study establish the significant performance merit of the sampling-based workload distribution technique. Future studies may use automatic techniques to identify the optimal number of probes for high performance. For instance, they could use profile-guided tuning, where `WPS` could continue the probing process until the execution time stops improving.

5.2.2 Estimating

In this phase, `WPS` computes the estimated size of the subtrees rooted at each item $n \in ST$. To compute this estimate, `WPS` traverses the ST bottom up and uses dynamic programming, as shown in Algorithm 5. In Algorithm 5 the values of Y_u^i represent the estimated size of the subtree rooted at the node of stratum u at level i of the WIT. The traversal of the ST, represented by the structure A, starts at the deepest level and moves toward the root (line 2). The values of Y_u^{i+1} are used to compute the values of Y_u^i (line 6). In this phase `WPS` produces a collection χ of Y_u^i values for every u and i encountered in the ST.

5.2.3 Partitioning

In the Sampling phase, `WPS` processes a small subset of the items in the WIT through `StraSa`. In this phase `WPS` partitions the remaining items in the WIT — the items not processed by `StraSa` — into M groups, where M is the number of processing nodes available. The items not processed by `StraSa` are the items in C as well as the items reachable from the items in C.

`StraSa` ensures that for each item n' in $C[i]$ there is a unique item n in $A[i]$ with $T(n) = T(n')$. Moreover, given Chen's assumption that items of the same stratum

```
Algorithm 5: ComputeSubtreeSizes
  Require: sampled tree $A$ and stratification $T$
  Ensure: a collection $\chi$ of the estimated subtree sizes $Y_t^i$
      for each level $i$ and stratum $t$ in $A$.
  1: $\chi \leftarrow \{\}$
  2: for $i \leftarrow$ tree depth to 1 do
  3:   for each item $n$ in $A[i]$ do
  4:     $Y^i_{T(n)} \leftarrow 1$
  5:     for each child $n''$ of $n$ in the $WIT$ do
  6:       $Y^i_{T(n)} \leftarrow Y^i_{T(n)} + Y^{i+1}_{T(n'')}$
  7:     insert $Y^i_{T(n)}$ in $\chi$
  8:   $i \leftarrow i - 1$
```

root subtrees of the same size, $Y^i_{T(n)}$ in χ is an estimate of the number of items in the subtree rooted at n' (number of items reachable from n'). Thus, at this point, the problem of evenly partitioning the workload reduces to the NP-Hard multi-way number partitioning problem [33]: the algorithm must partition the items n' in C into M parts $W_1, W_2, \cdots, W_M$ such that the sum of the $Y^i_{T(n')}$ values in each part W_j and W_k with $j, k \in \{1, 2, \cdots, M\}$ are as similar as possible to each other. WPS employs the Balanced Largest-First Differencing Method (BLDM) [53] to compute an approximated solution to the number partitioning problem. BLDM is a widely used, and effective, algorithm that performs k-way partitioning for $k \geq 2$ in $\mathcal{O}(n \log n)$ time.

This work uses the number of items rooted at each given item as a workload metric for even distribution. It assumes that the time required to process an item is constant throughout the WIT — an assumption that holds in all the applications studied in this chapter. WPS could be easily adapted to use other workload metrics as well. For example, in applications where work items have different processing times, StraSa could estimate the total processing time of subtrees as opposed to estimating the size of the subtrees. Then, BLDM would be used to partition the items not processed by StraSa into parts of similar processing time.

5.2.4 Distributing

In this phase, WPS stores one subset W_1 in local memory for processing in the current processing node and distributes the remaining $W_{j|j=2..M}$ subsets of items to the $M-1$ remaining processing nodes. The items are copied to the nodes asynchronously to ensure that a processing node does not need to wait for completion of data transfer to any other nodes. Multiple independent threads can be used to parallelize the copying operations across the processing nodes. In the applications used in this study, the work-items in different W_j subsets can be processed in any order as the applications generate valid results for all orders of processing of the items.

In applications where the work items must be expanded in an orderly fashion, processing of parent items first may be necessary to ensure that their children do not wait for a prolonged time. Work items in such applications can be processed in a monotonically increasing order of $A[i]$ because the parent items are stored at higher levels of the ST than their children. Optimizing the scheduling and distribution strategy for applications that exhibit irregular dependencies among the work items is beyond the scope of this work.

5.3 WPS Accuracy

`WPS` is intended to evenly distribute the workload among different processing nodes in a cluster, thus minimizing the need for load-balancing operations. An empirical evaluation (Section 5.5.2) indicates that `WPS` performs a good partitioning of the work list and consequently requires infrequent load-balancing operations. However, the approximations of the stratified sampling technique and of the multi-way number partitioning algorithm may leave room for some load imbalance both inside and across multiple nodes in a cluster. Therefore, `WPS` is intended to operate in coordination with existing load-balancing schedulers, such as work stealing, that manage intra- and inter-node load imbalances, and not to replace them completely. For the applications studied in this chapter, there was limited scope for inter-node load balancing. Nonetheless, coordinating `WPS` with existing load-balancing schemes in runtimes of programming systems is beneficial because: i) there is no load-balancing overhead if there is no load imbalance in the system; and ii) the infrequent load-balancing operations that may be necessary will be handled by the existing load-balancing techniques.

5.4 Application Problem: An Example

We apply `WPS` to parallelize three algorithms: IDA*, Delaunay Mesh Generation, and Delaunay Mesh Refinement. As a demonstration, this section describes how `WPS` can be applied to the IDA* algorithm.

IDA* is a fundamental algorithm in Artificial Intelligence for solving state-space search problems. Given a start state s^*, IDA* expands a tree while performing a Depth-First Search from s^* with cost bound d in the state space. IDA* uses a cost function defined as $f(n) = g(n) + h(n)$, where $g(n)$ is the cost to reach state n from s^*, and $h(n)$ is the estimated cost-to-go from n. The value of the cost bound d is initially set to the heuristic value of s^*.

In each iteration, IDA* expands all states n that it encounters such that $f(n) \leq d$. If a goal is not found, then d is increased by setting it to the lowest f-value larger than d observed in the previous iteration. If IDA* uses an admissible heuristic — a heuristic that never overestimates the optimal solution cost for any state n — then IDA* is guaranteed to return a path from s^* to the goal state, if one exists, with the optimal solution cost.

Parallelizing IDA*

The tree IDA* expands during search with a given cost bound is `WPS`'s *WIT*, and the states in the tree represent the items. In each IDA* iteration, `WPS` partitions the *WIT* into M parts of similar size. The M processors detect termination of each iteration and compute an estimate of the cost for the next iteration. `WPS` is used once again to partition the new *WIT* defined by the new cost bound. If a processor finishes its part of the search, it tries to steal items from other processors. All processors stop once the solution is found. Processor idling leads to substantial performance degradation because an iteration of IDA* does not start until the previous one is completed. Therefore, a balanced workload partitioning scheme is crucial for agent-search domains such as IDA*.

5.5 Experimental Evaluation

This section describes the experimental setup and discusses the main findings from the performance evaluation. Preliminary results from this evaluation were published in the Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (2014) [61].

5.5.1 Experimental Setup

Platform: Performance measurements use a blade server with 16 nodes, each featuring two 2 GHz Quad-Core AMD Opteron processors, with 8 GB of RAM and 20 GB of swap space, running CentOS GNU/Linux version 6.2. All binaries were compiled with GCC version 4.4.3 using the -O3 flag.

Compiler and Runtime: The `x10c++` compiler version 2.3.1 is used for all measurements. The nodes in the cluster are connected by an InfiniBand network with a bandwidth of 10 Gbit/s and use MVAPICH2 library for communication. The experimental runs create eight worker threads per node and vary the number of nodes from 1 to 16 so that the number of threads is the same as the total number of cores.

Start States for WPS: This evaluation uses Korf's [43] one hundred instances of 15-puzzle and 3 random instances of 24-puzzle (shown in Table 5.1). The start states for the puzzles can be generated through random valid permutations of the tiles. There is only one valid start state for DMG — the triangulation that encapsulates the given points. For DMR, triangulation of any given bad triangle is a valid start state. DMG uses 10M points and DMR uses 68M triangles as inputs.

Table 5.1: Instances of 24-Puzzle solved optimally.

13 14 17 22 9 21 8 10 6 7 5 16 0 24 1 15 2 23 4 3 18 19 12 11 20
2 0 10 19 1 4 16 3 15 20 22 9 6 18 5 13 12 21 8 17 23 11 24 7 14
9 6 15 10 0 20 17 16 5 24 2 3 21 14 7 18 13 19 4 12 11 22 8 23 1

Stratification: This evaluation uses a stratification that labels items n_1 and n_2 with the same stratum if three conditions are met: (1) $h(n_1) = h(n_2)$; (2) n_1 and n_2

generate the same number of children and grandchildren; and (3) the children and grandchildren have the same heuristic values. Lelis *et. al* [47] first introduced this stratification and showed that StraSa produces good estimates of the IDA* *WIT* size using such a stratification. For DMG and DMR, two items are said to have the same stratum if they generate the same number of children and grandchildren. Unless stated otherwise, WPS employs five probes of StraSa on 15-Puzzle, DMG, and DMR, and 25 probes on 24-Puzzle. Section 5.5.2 discusses the performance impacts of different stratifications and number of probes.

Methodology: Applications are run twenty times to account for variances, such as work-stealing in the X10 runtime, and scheduling policies in the operating system. The performance charts include 95% confidence intervals for execution times, speedups, and steals-to-tasks ratios.

The versions of the Operating System and the X10 Compiler used in this evaluation are different from the ones used in the evaluation in Chapter 3. Also, the input workload for the benchmarks used in this evaluation are different from the input workload used in the evaluations in Chapters 3 and 4. Therefore, the baseline performance of the benchmarks reported in this Chapter differ from those reported in previous chapters.

Workload Distribution Algorithms: The idea of performing state-space exploration to generate enough parallel tasks for distribution amongst processing nodes is not new. WPS provides a systematic way to do this without requiring programmers to code this solution. Unlike other existing techniques, WPS estimates the number of tasks that will be generated after processing the initial tasks to evenly divide the total workload in an application. This evaluation compares the performance of WPS against the following workload-distribution algorithms:

i) X10WS: X10's default intra-node work-stealing scheduler
ii) WD scheduler complements X10WS with task migration across nodes. WD migrates tasks to a node whose worker repeatedly fails to steal task from its co-located peers [60, 62].
iii) Eager-WD scheduler is similar to WD but proactively maps tasks in a load-balance-aware manner rather than waiting until the occurrence of a load-imbalance.
iv) DistWS scheduler complements X10's intra-node work-stealing with inter-node work-stealing [60, 63].

The evaluation also investigates the implications of operating WPS in isolation and in tandem with these schedulers:

v) WPS*: WPS operating without coordination with any other scheduler;
vi) WPS: WPS operating in coordination with X10WS; and
vii) WPS+DistWS: WPS operating in coordination with both X10WS and DistWS.

A closely-related algorithm intended specifically for IDA* also exists, it is called AIDA*. Section 5.5.3 compares the performance of WPS against AIDA*.

Initial Partitioning of Work-List: WPS partitions the *WIT* based on the predicted size of the children reachable from each item in the *WIT*. X10WS and DistWS employ the following approach to partitioning. The sliding-tile and DMG applications start with a single state and generate several states during the course of their execution. Therefore, in 15- and 24-Puzzles, the algorithms perform an iterative-deepening search for a few levels until there are at least $10 \times M$ items in the search frontier, where M is the number of available processing nodes. This initial exploration produces a sufficient number of subtree roots — about 1,300 nodes — while not overflowing the memory resources of the evaluation platform. Similarly, in DMG, the algorithms perform triangulation until $10 \times M$ triangles are formed in the mesh. These triangles and their encapsulated points are then distributed among M processing nodes. In DMR, the implicitly-defined worklist of triangles to be refined is distributed amongst M processing nodes. Unlike `WPS`, `X10WS` and `DistWS` are unable to account for the dynamically generated items during workload distribution.

5.5.2 Results and Discussion

The sequential execution times of 15-puzzle, DMG and DMR are shown in Figure 5.2. The sequential execution times of the three instances of 24-puzzle are long and they significantly differ from one another. Therefore, the parallel execution time for each instance at 128 workers is reported separately in Figure 5.3. The speedup results for other applications are shown in Figure 5.4.

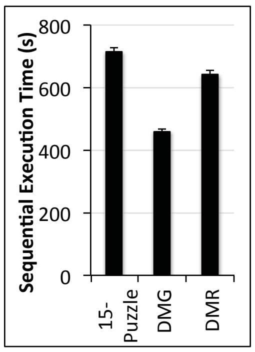


Figure 5.2: Sequential execution time.

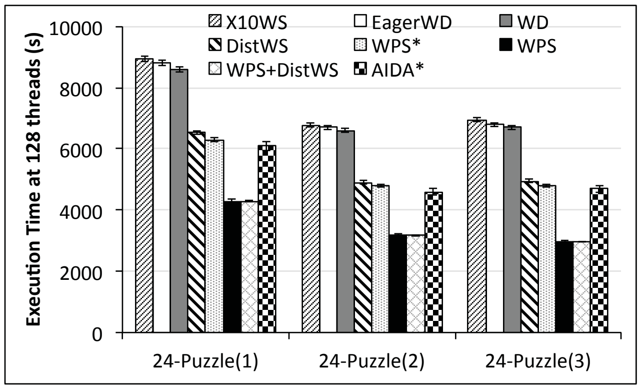
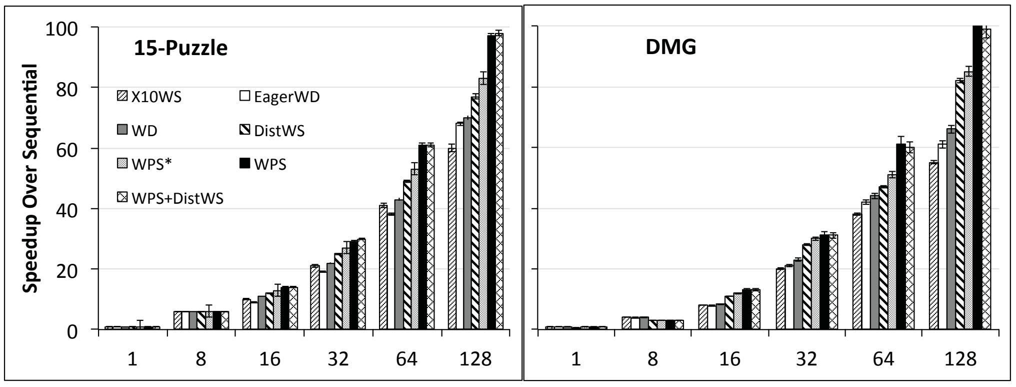


Figure 5.3: Execution times of 24-Puzzle at 128 workers.

The results from the experimental evaluation help us understand:

The performance benefits of `WPS*` For each application, Figure 5.4 shows the speedups achieved using different workload-distribution strategies at different worker counts. The speedups are relative to the sequential execution time using `X10WS`.



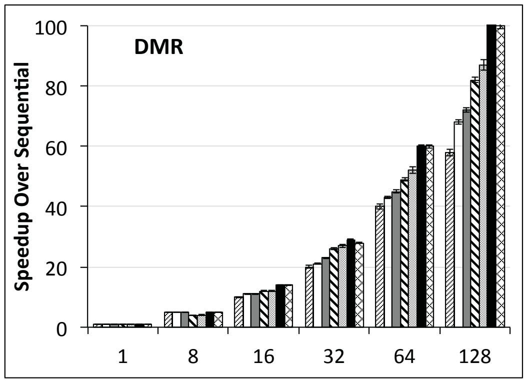


Figure 5.4: Application speedup over sequential execution time using different schedulers at different worker counts.

`WPS`* *consistently outperforms* `Eager-WD`*,* `WD`*, and* `DistWS`. For instance, the 86x speedup on DMR at 128 workers achieved using `WPS*` represents 23%, 20%, and 18% improvements over `Eager-WD`, `WD`, and `DistWS`, respectively. `Eager-WD` and `WD` yield relatively small speedups over `X10WS`. `Eager-WD` yields 11%, 10%, and 14% speedups on 15-puzzle, DMG and DMR respectively. `WD` exhibits better speedups — 14%, 16%, and 19% on the same applications. `DistWS` performs much better. It yields speedups of 22%, 32% and 29%. The algorithms exhibit similar trend on 24-Puzzles too. `DistWS` outperforms `X10WS` by 26%, 27%, and 28% on three instances of 24-Puzzle. `WPS*` outperforms `X10WS` by 29%, 29%, and 31% on the same instances.

The performance impact of coordinating WPS* with other schedulers `WPS*` aims to evenly distribute the workload among processing nodes, but not among multiple threads within a processing node. Therefore, `WPS` scheduler coordinates `WPS*` with `X10WS`. `WPS` yields significant speedups over `X10WS` on 15-Puzzle, DMG, and DMR — 38%, 45%, and 43% respectively. `WPS` *outperforms* `DistWS`*— the best performing scheduler among* `X10WS`*,* `Eager-WD`*,* `WD`*, and* `DistWS`*— by 21%, 18%, and 20% on 15-Puzzle, DMG, and DMR respectively.* `WPS` outperforms `DistWS` by 34%, 35%, and 40% on three instances of 24-Puzzle.

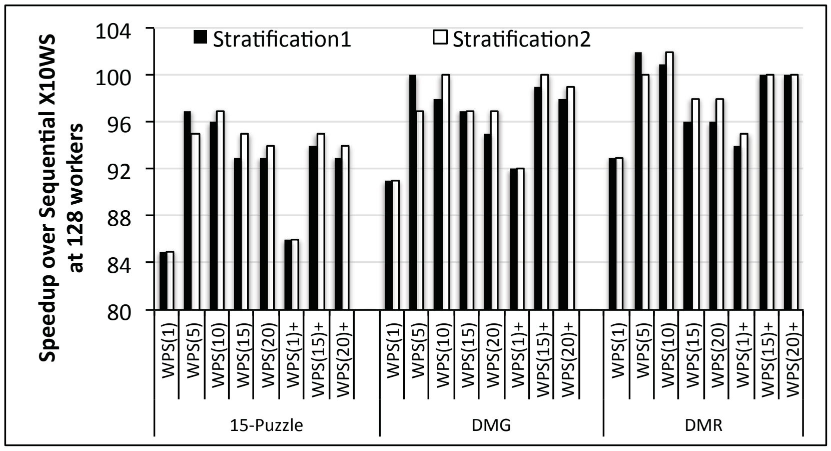


Figure 5.5: WPS performance using different stratifications. The number of probes used are indicated in the parentheses, and the '+' sign in the x-axis labels indicate WPS+DistWS.

WPS+DistWS does not yield significant speedup over WPS. This is a result of the even workload distribution generated by WPS, which leaves little opportunities for DistWS to migrate items between nodes for load balancing.

Further evaluations do not discuss WPS* and WPS+DistWS because they are not interesting in terms of overall speedups.

The impact of the precision of stratification on WPS performance The quality of the stratification guiding the sampling process may substantially impact the accuracy of the tree-size predictions. Ideally, a stratification would determine that two items are of the same stratum iff they root subtrees of the same size, which would allow StraSa to produce a perfect estimate of the tree size in a single probe. A trivial example of such a stratification is one in which every item belongs to its own stratum. In this case, there would be far too many strata to sample from and the approach would not be effective. In practice one should use a compact stratification (i.e., a stratification with a small number of strata) that has a low variance on the subtree sizes rooted at items of the same stratum. Multiple StraSa probes can be used to improve the accuracy of predictions when using low-quality stratifications.

Figure 5.5 shows the performance impact of different stratifications and of different numbers of StraSa probes on WPS. Stratification1 (S1) is the stratification described in Section 5.5.1, while Stratification2 (S2) is a version of S1 with fewer strata. Namely, S2 considers two items to be of different stratum if the number of children and grandchildren they produce differ by more than two. The numbers in parentheses in Figure 5.5 represent the number of probes used. WPS performs worse in all domains when using one probe (see WPS(1)). With a single probe, the performance does not improve even when coordinating WPS with DistWS (see WPS(1)+).

WPS yields best performance under S1 and S2 at five and ten probes of StraSa, respectively. This difference in the number of probes in which each stratification yields the best performance is because S1 has more strata than S2. Thus, a StraSa

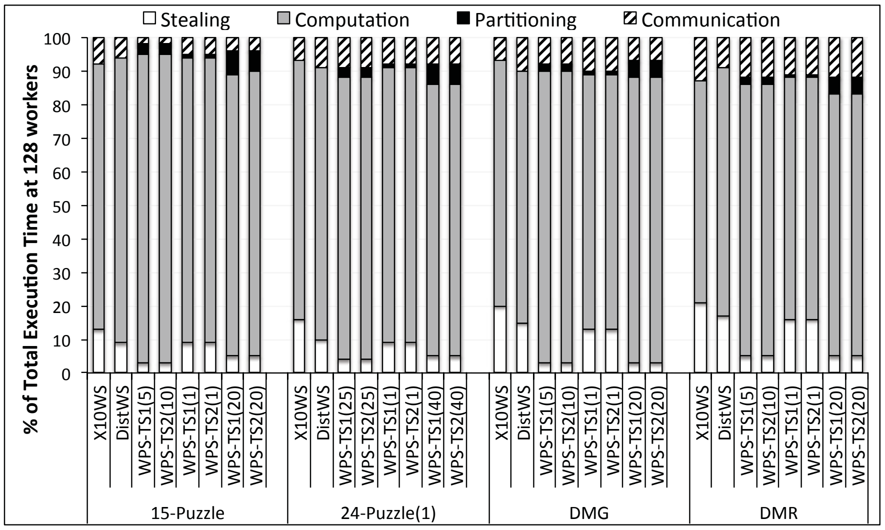


Figure 5.6: Breakdown of total execution time.

probe using S2 will tend to be faster than a StraSa probe using S1. Increasing the number of probes beyond five for S1 and ten for S2 is not beneficial. In fact, the performance worsened at larger number of probes because of the overhead of sampling. The loss in performance is regained by coordinating WPS with DistWS. The 24-Puzzles exhibit best performance under both stratifications at 25 probes, and exhibit similar trends overall.

The sources of performance gains Fig. 5.6 shows the breakdown of execution times of applications using X10WS, DistWS, and WPS. The performance gains arise from three major sources:

i) *Reduced Work-stealing Operations:* WPS reduces the execution time spent on work stealing in the range of 10% to 16% over X10WS and in the range of 7% to 12% over DistWS. The work-stealing time also accounts for the machine idle times — when workers are unsuccessfully searching for surplus work to steal — because an idle worker may continuously try to steal work from other workers. The reduced need for work-stealing operations means that the nodes will be mostly performing useful computations. Thus, actual computations in applications contribute more to the total execution time with WPS – in the range of 7% to 13% over DistWS and in the range of 14% to 16% over X10WS. These performance gains are achieved at the cost of 1% to 2% of the total execution time spent on workload partitioning using WPS.

ii) *Reduced Communication over the Network:* An even distribution of workload using WPS necessitates fewer message transmissions across the network. This reduced communication stems from fewer steal operations, fewer synchronized access to the shared deques of remote workers, and fewer accesses to data required to process stolen tasks. Table 5.2 shows the average number of messages transmitted across the network obtained from twenty runs of each application using X10WS, DistWS, and WPS at 128 threads. As expected, WPS requires fewer message transmissions across the network compared to X10WS and DistWS.

Table 5.2: Messages transmitted across network at 128 workers (in millions).

Applications	# of Messages Transmitted		
	X10WS	DistWS	WPS
15-Puzzle	12.43	10.49	3.40
24-Puzzle(1)	86.39	81.78	69.58
DMG	42.68	36.84	26.03
DMR	37.92	32.29	22.89

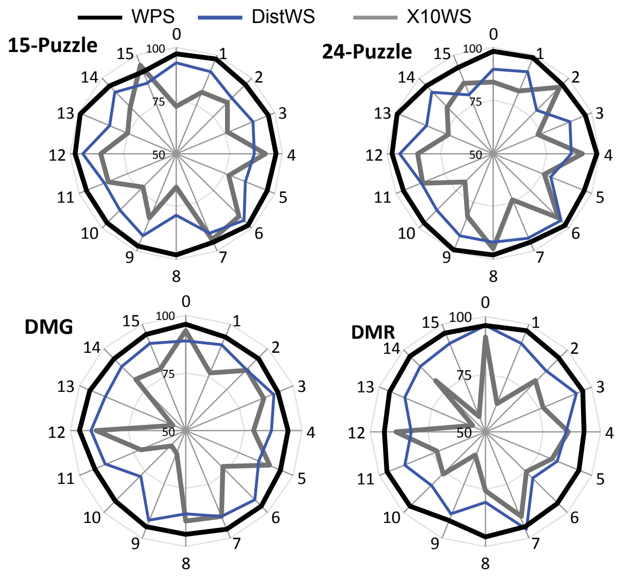
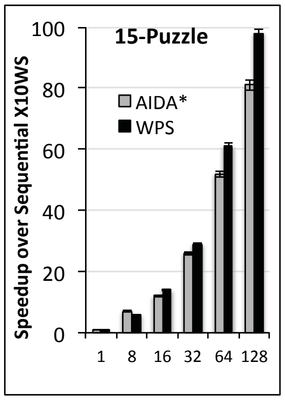


Figure 5.7: Average node utilization (128 threads). The figure shows the node utilizations starting from 50% for better clarity.

iii) *Improved Node Utilization:* Through an improved workload distribution, reduced work-stealing operations, and reduced machine idle times, WPS achieves an increased and uniform CPU utilization compared to X10WS and DistWS. The radial axes and the points in the circumference of the circles in Fig. 5.7 respectively indicate the average CPU utilization of eight cores in a node and a node in the cluster.

With X10WS, the standard deviations of average node utilization for 15-Puzzle, DMG and DMR are 12.45, 18.37, and 17.5, respectively. With DistWS, the standard deviations are 9.71, 11.49, and 10.2, respectively. With WPS, the standard deviations are 2.32, 3.12, 3.09 respectively. The lower standard deviations of average node-utilizations with WPS point to an improved load-distribution achieved with WPS.

DistWS permits workers in a processing node to steal tasks from remote processing nodes if all the co-located workers lack surplus work. Thus, DistWS performs more work compared to X10WS, which operates only within a node, resulting in an increased CPU utilization over X10WS. However, DistWS still exhibits a non-uniform CPU utilization because a large number of tasks stolen for load-balancing incur overheads of remote data accesses that are necessary for processing the stolen tasks.



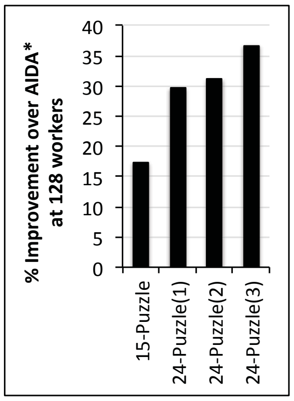


Figure 5.8: WPS and AIDA* performance on 15- and 24-Puzzles.

Hence, some nodes show heavy utilization while the others show lighter utilization.

WPS reduces the need for work stealing across processing nodes, thereby, enabling workers to perform useful computations. Thus, the node utilization circles for WPS are larger and more uniform compared to those for X10WS and DistWS.

5.5.3 Asynchronous IDA* (AIDA*)

Similar to WPS, AIDA* combines a data-partitioning scheme with work stealing for parallel and distributed implementation of IDA* [67]. AIDA* operates in three phases: i) in the *data-partitioning phase*, each processor redundantly expands the tree to generate enough frontier nodes; ii) in the *distributed node-expansion phase*, each processor expands its portion of the nodes generated in the preceding phase to generate additional finer-grained nodes; and, iii) in the *asynchronous search phase*, processors perform IDA* on their subtrees until a solution is found. All processors search to the same threshold. After completion of an iteration, the processors begin a new search pass through the same set of subtrees using a larger threshold.

This implementation of AIDA* uses the following constraints to closely match the original algorithm: i) selects neighbouring processors as the victims for work stealing; ii) partially re-orders the nodes in the local work list and only allows nodes of average size to be stolen; iii) steals in chunk sizes of five nodes; and iv) permits stealing of at most half of a victim's items. The cluster used in this evaluation is fully connected, where stealing from a randomly selected victim has the same effect as trying to steal from a neighbour in terms of communication latency. Nevertheless, we modify our load-balancer to mimic the original AIDA* algorithm.

AIDA* differs from WPS as follows: i) unlike AIDA*, WPS expands the search tree on a single processor to generate enough frontier items for distribution among processors; ii) while AIDA* relies on a robust distributed load balancer to mitigate the imbalance in subtrees distributed to the processors, WPS rarely needs load balancing across processing nodes. WPS relies only on intra-node load balancer to mitigate load imbalance among workers co-located in a processing node.

For the three instances of 24-Puzzle, Figure 5.3 shows the execution time performance using AIDA* and `WPS`. For the 15-Puzzle, Figure 5.8 shows the speedups achieved with AIDA* and `WPS` at different worker counts. Figure 5.8 also shows the performance gains with `WPS` relative to AIDA* at 128 workers. AIDA* does not account for the variability in the subtree sizes during initial distribution of the frontier nodes. Consequently, processing nodes often suffer from frequent load imbalances and require several expensive load-balancing operations. In addition, AIDA* also needs to perform expensive item sorting and coalescing to support efficient migration of items. As a result, `WPS` outperforms AIDA* by 17%, 29%, 31% and 36% on 15-Puzzle, and the three instances of 24-Puzzle.

5.6 Related Work

A popular task-distribution strategy in agent-search domains is to expand the search tree until there are enough states in the search frontier for distribution among processors [45, 66]. Such a strategy does not account for the size of dynamically generated states, thereby, causing a load imbalance. Parallel Window Search is another approach for load balancing where multiple worker threads search the same tree in parallel using different cost bounds [65]. The demerit of this approach is that some processors may search the tree with a search bound that is too high and result in wasted work if the optimal solution is found at lower depths by some other workers.

Transposition-table-driven scheduling (TDS) uses hash-function-indexed transposition table for uniform workload distribution [68]. The demerit of TDS is that it relies on robust dynamic optimizations, such as message coalescing, and requires: (i) dynamically identifying states with identical source and destination processors; (ii) coalescing them into a single message of appropriate granularity; and (iii) migrating the coalesced message packets at precise control-flow points. Efficient implementation of such operations often requires expensive data and control flow analyses.

Niewiadomski *et al.* [56] present a sampling-based approach for workload distribution in implicit graphs. Unlike their approach, `WPS` selectively samples only items of unique stratum and therefore does not require application programmers to manually identify the best sampling size for each application.

Frameworks, such as PREMA [8], Scioto [28], and Turbine [77], also aim to support automatic load balancing in irregular parallel applications. These framework require programmers to expose parallelism, and migratable data and objects in applications using special runtime libraries. Unlike these approaches, `WPS` does not require rewriting existing X10 applications, and only requires users to specify stratifications during program execution.

Mendéz-Lojo *et al.* [51] exploit high-level algorithmic structure and amorphous data-parallelism to optimize irregular applications. In particular, they focus on three optimizations: exploiting cautious operator implementations, one-shot implementations, and iteration coalescing. Similar to their work, `WPS` also systematically exploits the algorithmic structure, *i.e.,* recursive data-parallelism, of applications to improve their runtime performance. However, here the focus is on workload partitioning. As a future work, it would be interesting to investigate techniques for extending `WPS` for optimizing a range of applications with amorphous data-parallelism.

5.7 Summary

This chapter presents an algorithm for evenly partitioning workload in distributed shared-memory machines. The distribution relies on a sampling-based prediction of the size of the sub-tree rooted at a given item. An experimental evaluation of this approach on IDA*, Delaunay Triangulation and Delaunay Mesh Refinement algorithms yields substantial speedups compared to state-of-the-art approaches, including ones that are specifically targeted at individual applications.

The main strength of the algorithm is that its workload distribution strategy applies to a range of iterative work-list-based data-parallel irregular applications and does not require manual tuning of sampling strides. Another strength of the algorithm is its ability to integrate well with mainstream load-balancers, such as work-stealing schedulers. Although not necessary for applications studied in this evaluation, the algorithm may benefit from coordination with work-stealing schedulers, but it does not require a complete overhaul or removal of the existing schedulers.

Chapter 6

Conclusions and Future Work

The work underlying this dissertation investigates techniques for task distribution and for shared-variable management to improve the performance of parallel and distributed applications. The investigation leads to three observations about APGAS programming systems:

- First, not all parallel tasks are locality sensitive. All tasks are nonetheless bound to one address partition, thereby, restricting opportunities for load balancing. This work provided evidence that the migration of locality-flexible tasks alleviates dynamic load-imbalance to improve application performance. An important observation from the empirical evaluation is that coarse-grain tasks benefit more from this technique than fine-grain tasks.
- Second, a single coherence protocol does not yield consistent performance gains for diverse patterns of shared-variable accesses. We offer a framework that automatically identifies the access pattern of each shared variable in an application and employs a high-performing coherence protocol among three available protocols. The resulting performance gains are comparable to the best hand-tuned applications.
- Third, in recursive data-parallel applications, a naïve data partitioning scheme does not yield an even workload distribution. Using the statistical technique of stratified sampling to predict the amount of work that will be generated by an application dynamically improves workload partitioning. The approach is quasi-dynamic because it does not require executing the entire application to make the prediction. Through the integration of this approach into the X10 programming system, this research demonstrates that programmers do not need to manually code this solution for individual applications.

Future Work

This research establishes the significant promise of selective locality-aware task-migration strategy. The prototype implemented in this research to evaluate this idea uses manual annotations to identify locality-flexible tasks. The manual approach may not be practical in applications that are written in languages that do not require programmers to specify the affinity preferences for all parallel tasks. Therefore, a key future investigation is to use program analyses — static or dynamic — to automatically identify locality-flexible tasks.

The selection of coherence protocol uses end-to-end profiling in the offline mode. It also relies on the correlation between the input parameters and the access patterns of shared-variables to control the duration of online profiling. Identifying such a relationship between input parameters and shared-variable access patterns requires a keen understanding of the applications. In the worst case, the input parameters may have an irregular or non-linear impact on the access patterns. An interesting line of future research is to use sampling-based profiling techniques to determine the access patterns of shared variables. Sampling unavoidably introduces inaccuracies in the collected profiles, which may in turn lead to incorrect decisions about access patterns, but it also reduces the cost of profiling and may make it viable to profile for a larger fraction of the execution time. Prior research has shown that applying statistical correction techniques can reduce the bias incurred by the sampling, and thereby, yield reasonably accurate profiles [78].

The prototype implementation of the workload-partitioning algorithm uses manual tuning to determine the number of probes for different stratifications that yield the best performance. Manual tuning is tedious because it requires exploring a large space of possible combinations of these two parameters. Consequently, it is difficult to ensure the best confluence of stratifications and number of probes that yield accurate predictions while minimizing the overheads of sampling. Machine-learning and profiling techniques could be used to better explore the large space of stratifications, the number of probes, and their combinations.

Bibliography

[1] M. Acacio, J. Gonzalez, J. Garcia, and J. Duato. Owner Prediction for Accelerating Cache-to-Cache Transfer Misses in a CC-NUMA Architecture. In *Conference on Supercomputing*, pages 49–49, 2002.

[2] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The Data Locality of Work Stealing. In *Symposium on Parallel Algorithms and Architectures*, pages 1–12, Bar Harbor, Maine, United States, 2000.

[3] U. A. Acar, A. Chargueraud, and M. Rainey. Scheduling Parallel Programs by Work Stealing with Private Deques. In *Symposium on Principles and Practice of Parallel Programming*, pages 219–228, Shenzhen, China, 2013.

[4] M. Alvanos, M. Farreras, E. Tiotto, J. N. Amaral, and X. Martorell. Improving Communication in PGAS Environments: Static and Dynamic Coalescing in UPC. In *International Conference on Supercomputing*, pages 129–138, Eugene, Oregon, USA, 2013. ACM.

[5] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *Computer*, 29(2):18–28, Feb. 1996.

[6] J. Archibald and J.-L. Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems*, 4(4):273–298, Sept. 1986.

[7] R. Barik, J. Zhao, D. Grove, I. Peshansky, Z. Budimlic, and V. Sarkar. Communication Optimizations for Distributed-Memory X10 Programs. In *Parallel Distributed Processing Symposium*, pages 1101–1113, May 2011.

[8] K. Barker, A. N. Chernikov, N. Chrisochoides, and K. Pingali. A Load Balancing Framework for Adaptive and Asynchronous Applications. *IEEE Trans. Parallel Distrib. Syst.*, 15:2:183–192, Feb. 2004.

[9] C. Barton, C. Casçaval, G. Almási, Y. Zheng, M. Farreras, S. Chatterje, and J. N. Amaral. Shared Memory Programming for Large Scale Machines. In *Conference on Programming Language Design and Implementation*, pages 108–117, Ottawa, Ontario, Canada, 2006.

[10] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: Expressing Locality and Independence with Logical Regions. In *Conference on High Performance Computing, Networking, Storage and Analysis*, pages 66:1–66:11, Salt Lake City, Utah, 2012.

[11] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. In *Principles and Practice of Parallel Programming*, pages 168–176, Seattle, Washington, USA, 1990.

[12] S. Bertozzi, A. Acquaviva, D. Bertozzi, and A. Poggiali. Supporting Task Migration in Multi-processor Systems-on-Chip: A Feasibility Study. In *Proceed-*

ings of the conference on Design, automation and test in Europe: Proceedings, pages 15–20, 3001 Leuven, Belgium, Belgium, 2006.

[13] M. J. Best, S. Mottishaw, C. Mustard, M. Roth, A. Fedorova, and A. Brownsword. Synchronization via Scheduling: Techniques for Efficiently Managing Shared State. In *Conference on Programming Language Design and Implementation*, pages 640–652, San Jose, California, USA, 2011.

[14] R. L. Bocchino, V. S. Adve, and B. L. Chamberlain. Software Transactional Memory for Large Scale Clusters. In *Principles and Practice of Parallel Programming*, pages 247–258, Salt Lake City, UT, USA, 2008.

[15] B. Cahoon and K. S. McKinley. Data Flow Analysis for Software Prefetching Linked Data Structures in Java. In *Parallel Architectures and Compilation Techniques*, pages 280–291, 2001.

[16] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Techniques for Reducing Consistency-related Communication in Distributed Shared-Memory Systems. *ACM Transactions On Computing Systems*, 13(3):205–243, Aug. 1995.

[17] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. Habanero-Java: The New Adventures of Old X10. In *International Conference on Principles and Practice of Programming in Java*, pages 51–61, New York, NY, USA, 2011. ACM.

[18] L. M. Censier and P. Feautrier. A New Solution to Coherence Problems in Multicache Systems. *IEEE Transactions on Computers*, 27(12):1112–1118, Dec. 1978.

[19] B. Chamberlain, D. Callahan, and H. Zima. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[20] B. Chamberlain, S.-E. Choi, T. Hildebrandt, V. Litvinov, G. Titus, J. Lewis, K. Maschhoff, and J. Claridge. Chapel HPC Challenge Entry, Nov. 2011.

[21] R. Chandra, A. Gupta, and J. L. Hennessy. Data Locality and Load Balancing in COOL. In *Symposium on Principles and Practice of Parallel Programming*, pages 249–259, San Diego, California, United States, 1993.

[22] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An Object-oriented Approach to Non-uniform Cluster Computing. In *Conference on Object-Oriented Programming, Systems, Languages & Applications*, pages 519–538, San Diego, CA, USA, 2005. ACM.

[23] P.-C. Chen. Heuristic Sampling: A Method for Predicting the Performance of Tree Searching Programs. *SIAM Journal on Computing*, 21:295–315, 1992.

[24] W.-Y. Chen, C. Iancu, and K. Yelick. Communication Optimizations for Fine-Grained UPC Applications. In *Parallel Architectures and Compilation Techniques*, pages 267–278, Washington, DC, USA, 2005.

[25] G. Cong, S. Kodali, S. Krishnamoorthy, D. Lea, V. Saraswat, and T. Wen. Solving Large, Irregular Graph Problems Using Adaptive Work-Stealing. In *International Conference on Parallel Processing*, pages 536 –545, Portland, Oregan, USA, 2008.

[26] J. DeSouza and L. V. Kale. MSA: Multiphase Specifically Shared Arrays. In R. Eigenmann, Z. Li, and S. P. Midkiff, editors, *Languages and Compilers for High Performance Computing*, volume 3602, pages 268–282. 2005.

[27] J. Dinan, S. Krishnamoorthy, D. B. Larkins, J. Nieplocha, and P. Sadayappan. Scioto: A Framework for Global-View Task Parallelism. In *International Conference on Parallel Processing*, pages 586–593, Washington, DC, USA, 2008.

[28] J. Dinan, S. Krishnamoorthy, D. B. Larkins, J. Nieplocha, and P. Sadayappan. Scioto: A Framework for Global-View Task Parallelism. In *Intern. Conf. on Parallel Processing*, pages 586–593, Washington, DC, USA, 2008.

[29] D. Eager, E. Lazowska, and J. Zahorjan. Adaptive load sharing in homogeneous distributed systems. *Software Engineering, IEEE Transactions on*, SE-12(5):662–675, May 1986.

[30] T. El-Ghazawi and L. Smith. UPC: Unified Parallel C. In *Conference on Supercomputing*, Tampa, Florida, 2006. ACM.

[31] M. Farreras, G. Almasi, C. Cascaval, and T. Cortes. Scalable RDMA performance in PGAS languages. In *Symposium on Parallel & Distributed Processing*, pages 1–12, 2009.

[32] M. Ferdman and B. Falsafi. Last-Touch Correlated Data Streaming. In *Intern. Symp. on Performance Analysis of Systems and Software*, pages 105–115, 2007.

[33] M. R. Garey and D. S. Johnson. *Computers and Intractability.* Freeman, San Francisco, 1979.

[34] D. Grove, O. Tardieu, D. Cunningham, B. Herta, I. Peshansky, and V. Saraswat. A Performance Model for X10 Applications: What's Going on Under the Hood? In *ACM SIGPLAN X10 Workshop*, pages 1:1–1:8, San Jose, California, 2011.

[35] Y. Guo, R. Barik, R. Raman, and V. Sarkar. Work-first and Help-first Scheduling Policies For Async-finish Task Parallelism. In *International Parallel and Distributed Processing Symposium*, pages 1–12, Washington, DC, USA, 2009.

[36] Y. Guo, J. Zhao, V. Cave, and V. Sarkar. SLAW: A Scalable Locality-aware Adaptive Work-stealing Scheduler for Multi-core Systems. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 341–342, Bangalore, India, 2010. ACM.

[37] P. E. Hart, N. J. Nilsson, and B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost paths. *IEEE Transactions on Systems Science and Cybernetics*, SSC-4(2):100–107, 1968.

[38] M. Herlihy and Y. Sun. Distributed Transactional Memory for Metric-Space Networks. In *Conference on Distributed Computing*, pages 324–338, Cracow, Poland, 2005.

[39] P. N. Hilfinger, D. Bonachea, D. Gay, S. Graham, B. Liblit, G. Pike, and K. Yelick. Titanium Language Reference Manual. Technical report, Berkeley, CA, USA, 2006.

[40] L. V. Kale and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System based on C++. In *Conference on Object-oriented programming systems, languages, and applications*, pages 91–108, Washington, D.C., USA, 1993.

[41] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *International Symposium on Computer Architecture*, pages 13–21, Queensland, Australia, 1992.

[42] D. E. Knuth. Estimating the Efficiency of Backtrack Programs. *Math. Comp.*, 29:121–136, 1975.

[43] R. E. Korf. Depth-First Iterative-Deepening: An Optimal Admissible Tree Search. *Artificial Intelligence*, 27(1):97–109, 1985.

[44] M. Kulkarni, M. Burtscher, C. Casçaval, and K. Pingali. Lonestar: A Suite of Parallel Irregular Programs. In *International Symposium on Performance Analysis of Systems and Software*, Boston, MA, USA, 2009.

[45] V. Kumar and V. N. Rao. Parallel Algorithms for Machine Intelligence and Vision. chapter Scalable Parallel Formulations of Depth-first Search, pages 1–41. Springer-Verlag New York, Inc., New York, NY, USA, 1990.

[46] S.-M. Lau, Q. Lu, and K.-S. Leung. Adaptive Load Distribution Algorithms for Heterogeneous Distributed Systems with Multiple Task Classes. *Journal of Parallel and Distributed Computing*, 66(2):163 – 180, 2006.

[47] L. H. S. Lelis, S. Zilles, and R. C. Holte. Predicting the Size of IDA*'s Search Tree. *Artificial Intelligence*, pages 53–76, 2013.

[48] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-based Cache Coherence Protocol for the DASH Multiprocessor. In *International Symposium on Computer Architecture*, pages 148–159, Washington, USA, 1990.

[49] K. Li. IVY: A Shared Virtual Memory System for Parallel Computing. In *Internationl Conference on Parallel Processing*, pages 94–101, 1988.

[50] Z. Majo and T. R. Gross. Memory Management in NUMA Multicore Systems: Trapped Between Cache Contention and Interconnect Overhead. In *International Symposium on Memory Management*, pages 11–20, San Jose, California, USA, 2011.

[51] M. Méndez-Lojo, D. Nguyen, D. Prountzos, X. Sui, M. A. Hassaan, M. Kulkarni, M. Burtscher, and K. Pingali. Structure-driven Optimizations for Amorphous Data-parallel Programs. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 3–14, Bangalore, India, 2010. ACM.

[52] J. Meng and K. Skadron. Performance Modeling and Automatic Ghost Zone Optimization for Iterative Stencil Loops on GPUs. In *Conference on Supercomputing*, pages 256–265, NY, USA, 2009.

[53] W. Michiels, J. Korst, E. Aarts, and J. Leeuwen. Performance Ratios for the Differencing Method Applied to the Balanced Number Partitioning Problem. In *STACS 2003*, volume 2607 of *Lecture Notes in Computer Science*, pages 583–595. Springer Berlin Heidelberg, 2003.

[54] S.-J. Min, C. Iancu, and K. Yelick. Hierarchical Work Stealing on Manycore Clusters. In *Conference on Partitioned Global Address Space Programming Models*, 2011.

[55] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Aprà. Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit. *Int. J. of High Performance Computing Applications*, 20(2):203–231, May 2006.

[56] R. Niewiadomski, J. Amaral, and R. Holte. A Parallel External-Memory Frontier Breadth-First Traversal Algorithm for Clusters of Workstations. In *International Conference on Parallel Processing*, pages 531–538, Aug 2006.

[57] R. W. Numrich and J. Reid. Co-Array Fortran for Parallel Programming. *SIGPLAN Fortran Forum*, 17(2):1–31, 1998.

[58] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C.-W. Tseng. UTS: An Unbalanced Tree Search Benchmark. In *Conference on Languages and Compilers for Parallel Computing*, pages 235–250, LA, USA, 2007.

[59] J. Paudel and J. N. Amaral. Using Cowichan Problems to Investigate the Programmability of X10 Programming System. In *ACM SIGPLAN X10 Workshop*, San Jose, CA, USA, 2011.

[60] J. Paudel and J. N. Amaral. Hybrid Parallel Task Placement in Irregular Applications. *Journal of Parallel and Distributed Computing*, 2014. doi:10.1016/j.jpdc.2014.09.014.

[61] J. Paudel and J. N. Amaral. Stratified Sampling for Even Workload Partitioning. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 503–504, Edmonton, AB, Canada, 2014. ACM.

[62] J. Paudel, O. Tardieu, and J. N. Amaral. Hybrid Parallel Task Placement in X10. In *X10 Workshop*, pages 31–38, Seattle, Washington, USA, 2013.

[63] J. Paudel, O. Tardieu, and J. N. Amaral. On the Merits of Distributed Work-Stealing on Selective Locality-Aware Tasks. In *International Conference on Parallel Processing*, pages 100–109, Lyon, France, 2013.

[64] J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison & Wesley, 1984.

[65] C. Powley and R. Korf. Single-Agent Parallel Window Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(5):466–477, May 1991.

[66] V. N. Rao, V. Kumar, and K. Ramesh. A Parallel Implementation of Iterative-Deepening-A*. In *National Conference on Artificial Intelligence - Volume 1*, AAAI'87, pages 178–182. AAAI Press, 1987.

[67] A. Reinefeld and V. Schnecke. AIDA* – Asynchronous Parallel IDA*. In *Canadian Conference on Artificial Intelligence*, pages 295–302, 1994.

[68] J. Romein, H. Bal, J. Schaeffer, and A. Plaat. A Performance Analysis of Transposition-Table-Driven Work Scheduling in Distributed Search. *IEEE Transactions on Parallel and Distributed Systems*, 13(5):447–459, May 2002.

[69] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove. X10 Language Specification. http://x10.codehaus.org/x10/documentation.

[70] V. Saraswat, P. Kambadur, S. Kodali, D. Grove, and S. Krishnamoorthy. Lifeline-based Global Load Balancing. In *Symposium on Principles and Practice of Parallel Programming*, pages 201–212, San Antonio, TX, USA, 2011.

[71] J. Savant and S. Seidel. MuPC: A Runtime System for Unified Parallel C. Technical Report CS-TR-02-03, Michigan Tech University, MI, USA, Sep 2002.

[72] E. Speight and M. Burtscher. Delphi: Prediction-Based Page Prefetching to Improve the Performance of Shared Virtual Memory Systems. In *Conference on Parallel and Distributed Processing Techniques and Applications*, pages 49–55, 2002.

[73] O. Tardieu, H. Wang, and H. Lin. A Work-stealing Scheduler for X10's Task Parallelism with Suspension. In *Symposium on Principles and Practice of Parallel Programming*, pages 267–276, New Orleans, Louisiana, USA, 2012. ACM.

[74] B. K. Totty and D. A. Reed. Dynamic Object Management for Distributed Data Structures. In *Conference on Supercomputing*, pages 692–701, Minneapolis, Minnesota, USA, 1992.

[75] T. Wen, J. Su, P. Colella, K. Yelick, and N. Keen. An Adaptive Mesh Refinement Benchmark for Modern Parallel Programming Languages. In *Conference on Supercomputing*, pages 40:1–40:12, Reno, Nevada, 2007.

[76] G. Wilson. Assesing the Usability of Parallel Programming Systems: The Cowichan Problems. In *IFIP Working Conference on Programming Environments for Massively Parallel Distributed Systems*, pages 183–193, Basel, Switzerland, 1993.

[77] J. M. Wozniak, T. G. Armstrong, K. Maheshwari, E. L. Lusk, D. S. Katz, M. Wilde, and I. T. Foster. Turbine: A Distributed-memory Dataflow Engine for Extreme-scale Many-task Applications. In *Workshop on Scalable Workflow Execution Engines and Technologies*, pages 5:1–5:12, Scottsdale, Arizona, 2012.

[78] B. Wu, M. Zhou, X. Shen, Y. Gao, R. Silvera, and G. Yiu. Simple Profile Rectifications Go a Long Way - Statistically Exploring and Alleviating the Effects of Sampling Errors for Program Optimizations. In *European Conference on Object Oriented Programming*, pages 654–678, 2013.

[79] J. Zebchuk, V. Srinivasan, M. K. Qureshi, and A. Moshovos. A Tagless Coherence Directory. In *Symposium on Microarchitecture*, pages 423–434, New York, NY, USA, 2009.

[80] Y. Zhu and L. J. Hendren. Communication Optimizations for Parallel C Programs. In *Conference on Programming Language Design and Implementation*, pages 199–211, Montreal, Quebec, Canada, 1998.