ALLEVIATING THE VARIABILITY AND COMMUNICATION OVERHEADS OF IRREGULAR PARALLELISM FOR MANY-CORE CHIPS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Camilo A. Moreno
August 2016

This dissertation is online at: http://purl.stanford.edu/rs281ww7523

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Christos Kozyrakis, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Mark Horowitz

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Oyekunle Olukotun

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost for Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.

Abstract

Technology trends and architectural developments continue to enable an increase in the number of cores available on multicore chips. Certain types of applications easily take advantage of increased concurrency; but there are important application domains that struggle to do so. Such irregular applications, often related to sparse datasets or graph processing, have time-variable, hard-to-predict parallelism. They tend to resist blocking, partitioning, and other static parallelization methods; requiring instead the use of dynamic load-balancing layers. These layers, often taking the form of task-management abstractions, by necessity rely more strongly on communication and synchronization than regular applications. As core counts reach into the hundreds, more careful handling of this synchronization becomes necessary to avoid bottlenecks and load imbalances, in ways that are not seen at lower core counts. To the extent that on-chip communication delays become more important due to scaling trends, the challenges presented by irregular parallelism are also accentuated.

We explore solutions to these issues in two dimensions: On the software front, we design a task-stealing runtime layer appropriate to the needs of irregular parallel applications on hundred-core chips. By setting up fast and efficient ways to share information, the runtime is able to embrace the varying relationship between available parallelism and core count, adapting dynamically to both abundance and scarcity of work. When tested with representative sparse-data and graph-processing applications on 128 cores, Runtime overhead is reduced by 60%, simultaneously achieving 15% faster execution and 29% lower system utilization, and does so without hardware assistance, unlike prior solutions.

On the hardware front, we address the "latency multiplying" effects of relying on

on-chip coherent caches for communication and synchronization between cores. We propose a set of techniques that enable efficient implementation of both synchronization-oriented and data-sharing access patterns by reducing the number and complexity of cache-coherence transactions resulting from these activities. Using ISA hints to express preferred placement of both data access and atomic operations, faster and more efficient communication patterns can result. This can be done in ways that are compatible with existing architectures, require only minimal code changes, and present few practical implementation barriers. We show experimentally on 128 cores that these tools yield many-fold speedups to synchronization benchmarks, and cut the optimized task-stealing runtime's remaining overhead, for a 70% speedup to stealing benchmarks and 18% to actual applications, along with increased energy efficiency.

The result of this twofold approach is to significantly improve the viability of irregular parallelism on many-core chips. We minimize waste and enable faster communication within the chip, both by implementing a smarter, more context-aware low-level software layer, and by finding user-friendly ways to implement more efficient communication-oriented access patterns at the cache coherence level.

Acknowledgements

I would like to thank the following people:

My wife Kirstin and our little ones Lucia, Sebastian, and Ines, all four of whom have supported me and been neglected many, many times in the pursuit of this work, even if the kids may be too young to remember much of it later.

My advisor Christos Kozyrakis, for clarifying guidance, great ideas, and on-target critiques; but most of all, for extreme patience in the face of stubborn hard-headedness.

Other faculty who at different points provided guidance on this work, either during meetings, PPL retreats, or by serving on the reading committee and/or being present at my defense. This includes profs. Mark Horowitz, Kunle Olukotun, Mendel Rosenblum, Pat Hanrahan, Roger Howe, and possibly others I have forgotten. I'd also like to thank Prof. Bill Dally, under whom I first dabbled in research; I learned a lot even if it didn't work out.

I'd like to thank my co-workers, collaborators and mentors from Intel Labs, including Richard Yoo, Daehyun Kim, Jongsoo Park, and especially Chris Hughes, who has always been patient, supportive, and glad to share experience and expertise with the clueless.

Finally I'd like to thank my friends and colleagues here at Stanford, all those with whom I've shared weekly lunches, occasional PPL retreats, fortuitous run-ins, class projects, or IVGrad events with. I've never been among so many brilliant people before, it's been both humbling and uplifting, and I count myself lucky to have lived it.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

For a long time, the performance of single-core computers grew in ways that more than satisfactorily addressed the growth in computing demands from applications, thanks in great part to Moore's law [70]. However, with the end of Dennard scaling's [18] predictable performance and power improvements, single-core performance famously levelled off a few years ago. This increased the focus on multi-core architectures as a way to continue to scale general-purpose processor performance with transistor count. Now, even low-cost smartphones have multiple cores, and datacenters and HPC systems can employ chips containing several dozen cores [10]. Multi-socket systems can already surpass the hundred-core threshold, and very soon single-socket systems will as well [90].

With larger numbers of cores, communication latencies within the chip grow in importance, driven by the difference in scaling between core logic and interconnect delays. Process scaling can enable faster logic and increased transistor count, but it worsens interconnect delays due to transmission line geometry [19]. This results in architects having to give greater consideration to propagation delays in all kinds of architectural decisions with each process generation, resulting, for example, in increased complexity of on-chip caches and networks.

Large numbers of cores also create performance challenges related simply to the number of simultaneous active threads. These may include increased synchronization needs [4], and more significant contention bottlenecks and load imbalances. The

increased relevance of interconnect latency will only compound these problems. This combination has the potential to become a significant barrier to scaleability.

Applications with regular parallelism — meaning intelligible, structured, and/or predictable — can work around these issues in multiple ways. Some workloads can be partitioned into discrete pieces having very low communication needs, as is classically the case with dense matrix multiply [40]. Necessary communication may be made less frequent by temporal blocking [98], for example. Regular access patterns may be exploited through multi-buffering and prefetching [7]. Generally, strategies that maximize locality at each cache hierarchy level [20], aside from reducing single-core memory delays, help reduce vulnerability to communication problems.

There are, however, emerging application domains where irregularity is common. Two of these are sparse-data solvers, such as conjugate gradient methods [32] and others (MUMPS [11], SuperLU [61]); and graph-based applications such as max-flow/min-cut [12] and strongly-connected-component detection [46]. In both of these domains, performance has significant dependence on the input data (e.g., sparse matrices or irregular graphs), and large variations in parallelism are common both between executions and during a single execution. Variability decreases the effectiveness of existing parallelism optimizations such as partitioning and prefetching, and many new algorithms have been formulated specifically to handle irregular datasets. To maintain load balance and maximize parallelism, such sparse- or irregular-centric applications often rely on dynamic scheduling with task stealing [25]. However, dynamic scheduling necessarily carries a larger need for synchronization among threads. On many-core chips, with larger core counts and communication challenges, greater reliance on synchronization creates overheads that lead to poor performance scaling.

1.1 This Work

This thesis addresses the scaling of irregular parallel applications on many-core chips. At a software level, we study the synchronization needs of state-of-the-art dynamic scheduling approaches when translated to many-core chip platforms.

At a hardware level, we explore factors exacerbating communication penalties in

many-core chips, and propose practical ways to reduce their impact. By these two efforts, we reduce both the applications' need for synchronization, and the accompanying performance and energy penalties.

On the software front, we observe that the combination of variable parallelism and increased core counts can result in significant periods of both relative abundance (*thread constraint*) and scarcity (*task constraint*) of available work, during a single execution, to a degree not observed on smaller systems. These two regimes place different demands on the load-balancing portion of the application code (often implemented as an abstracted task-stealing runtime layer). A good solution in this space must be able to handle both phases occurring unpredictably during the same execution. Existing runtimes on multi-core fail to do this, instead tending to assume thread-constrained operation, since it has historically dominated on chips with small core counts [17, 35].

We show that in task-constrained phases, it is important to quickly and efficiently match new work to ready cores, which current approaches struggle to do. For energy efficiency, the number of cores utilized should also respond in proportion to the availability of work, enabling the exploration of power management or multi-processing on the underused cores. Conversely, when returning to thread-constrained phases, we must minimize system-wide overhead to maximize overall productive throughput.

We develop a work-stealing runtime around these insights. We combine a hierarchical state-tracking scheme, a hybrid task-stealing and work-sharing approach, and dynamic adaptation to changing conditions. We implement this system fully in software, unlike prior work [57, 86], with careful consideration of contention chokepoints, data movement, and cache-coherence traffic patterns in hundred-core chips. This design leads to faster, smarter stealing and efficient core utilization during task-constrained phases; it can dynamically emphasize either low overheads or low stealing latency depending on available parallelism; and it manages idle threads by opportunistically relying on them for runtime work and by allowing for power savings.

As a result, our runtime accelerates the discovery and stealing of work by up to 13x in a 128-core system. Through an overall 60% lowering of runtime overhead, it speeds up completion of real-world irregular workloads by 15% on average, while

simultaneously reducing total core utilization by 29%. This recovery of core idle time, in turn, presents significant opportunities for core power management as well as prioritized multi-programming.

We then examine the hardware. Through a preliminary study, we verify that on-chip-cache communication delays represent more than 50% of the runtime's remaining performance overhead. Furthermore, though some of these delays are unavoidable given the desired information flow, a large portion is due to how the on-chip caches respond to access patterns resulting from synchronization through atomic updates, and from data sharing.

We propose a small set of architectural solutions to address the observed issues at the cache coherence level in a way that retains coherence and consistency benefits and does not change how most data is handled. We first leverage hints for explicitly placing data at a convenient cache tier, either private [78] or shared depending on programmer intentions. We also make limited use of remote execution concepts (similar in spirit to some cluster solutions [71]); we enable a small subset of atomic operations to be performed at shared cache controllers to reduce latencies when atomically operating on shared values. We accomplish this without compromising compatibility with existing programming or memory models, sacrificing generality of application, or imposing new architectures or major code rewrites. Our solution also has smaller practical barriers to implementation than prior work.

We prove the effectiveness of these methods through simulation, easily cutting communication delays in a task-stealing runtime to show 70% speedup on task-stealing benchmarks, and 18% speedup to the runtime's client applications. We show many-fold improvements in weak-scaling performance for synchronization benchmarks. We also show significant reductions in cache energy use.

1.2 Contributions

This work makes the following contributions:

- Through analysis of representative workloads, we show how the combination of

irregular parallel applications and hundred-core chips highlights on-chip communication latencies, as well as the breakdown of abundance assumptions in task-stealing as increasingly urgent scaling problems. In response we develop a task-stealing dynamic runtime designed for such situations. This runtime relies on communicating quickly and efficiently about system state; maintaining efficiency across changing conditions through dynamic adaptation; and requiring no hardware support, unlike prior solutions.

- When tested with irregular applications running on a simulated 128-core system, this new runtime's efficient handling of both task-constrained and thread-constrained situations cuts overhead by 60% to speed up overall execution by 15% over the state-of-the-art. At the same time, it reduces core utilization by 29%, opportunistically making cores available for dynamic power management.

- To further reduce communication penalties in irregular code, including that of our runtime, we introduce cache-coherence level support for direct access to both private and shared caches in remote tiles, enabling common communication patterns to be simply expressed. Because atomic operations can be critical for synchronization primitives, we also enable remote execution of certain atomic operations on data in shared caches.

- We implement and test this set of tools on our simulated 128-core chip. Benchmark performance is improved many-fold; and the task-stealing overhead of our runtime is significantly reduced, yielding a further 18% speedup to user applications. Along with this we observe power savings due to improvements in cache use. Our experiments also prove that this scheme can be integrated into a general-purpose, cache-coherent architecture without large code rewrites, loss of coherence, forced adoption of new programming models, or significant practical implementation issues.

Chapter 2

Motivation and Background

Our work is motivated mainly by the intersection of two ongoing trends in computing: the increasing core counts in many-core chips, and the increasing relevance of applications exhibiting irregular parallelism. In this section we will explore these two trends in greater depth.

2.1 Many-Core Systems

Since the end of Dennard scaling [18], and the resulting slowdown in single-processor ILP improvement, processor architects have sought new ways to leverage the transistor counts made possible by process advances. One of the main directions has been to increase the number of cores per chip [45]. This has resulted, as Figure 2.1 illustrates, in an ongoing trend towards *many-core* chips [21].

One can already obtain from vendors such as *Oracle, Cavium*, and *Intel*, among others, chips with 32, 48, and 61 general purpose cores, respectively; these chips have caches with coherence capability, offering a shared-memory abstraction.

In the near future, even larger chips will come to market. Phytium's MARS and Applied Micro's X-Gene 3 have been announced with 64 cores; both Tilera (now EZChip) and Intel have imminent 72 core chips; and EZChip has also announced a 100-core chip for sampling in 2016.

Research questions abound regarding the architectural direction that the future

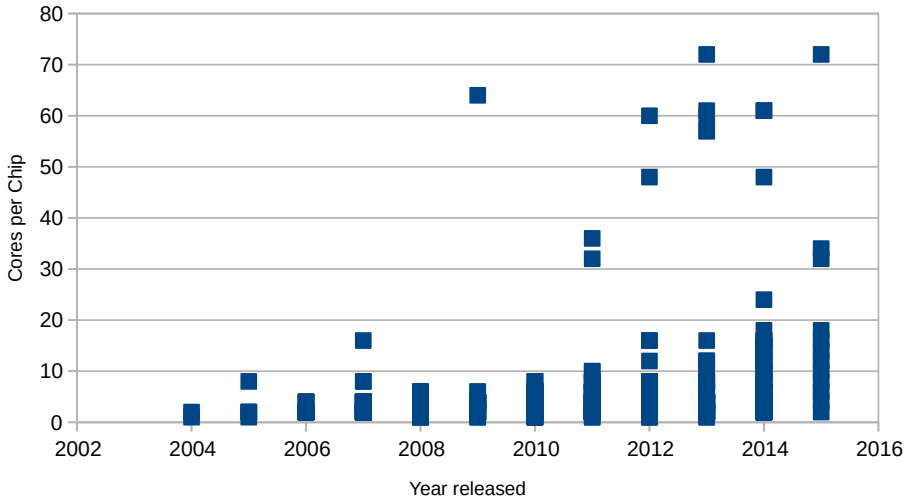


Figure 2.1: Evolution of chip core-count over time, for commercially-available chips. Steadily evolving mainstream offerings make up the underlying trend. The top right scattering comprises chips from Intel's MIC series, Cavium, Tilera, and Oracle's SPARC. Some of this data is from CPU DB [29].

of many-core chips should take, and published research exists exploring many of them well into the thousand core scale [21]. One example domain is the scaling of cache coherence [53, 85] and on-chip networks [38]: Current chips from both *Oracle* and *Intel* employ ring networks; likely due to their logical simplicity and efficient use of space. However, to reach the thousand-core scale, networks and caches must embrace more complex structures to achieve performance while avoiding severe space limitations [63].

The type of core that should be used is another interesting question. Maximizing parallelism through core and thread counts favors simple cores. On the other hand, decades of single- and few- core system dominance, plus the difficulty of parallel programming, favor the single-thread performance of complex cores. Current many-core offerings vary on this point. Some offer complex cores, attempting to lure users along gently from mainstream multi-core chips; others attempt to disrupt by offering large numbers of simple cores.

Though it is clear the evolution will require careful design choices, there is great

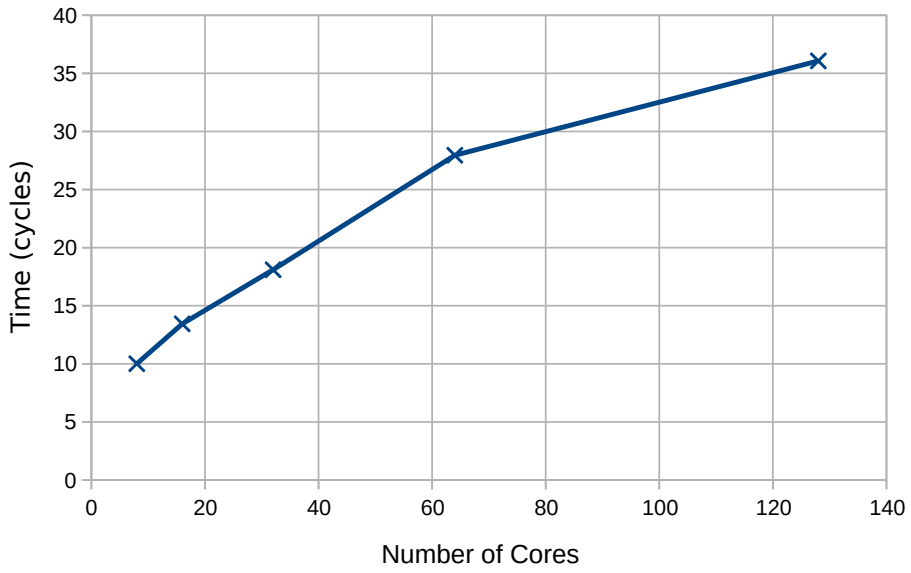


Figure 2.2: Average packet latency for coherence messages on a simulated 2D-mesh many-core chip, for different core counts. The number of network hops per packet determines a square-root relationship to core count.

promise in many-core chips for high-throughput, efficient general purpose computing.

2.2 The Challenges of Many-Core

Communication delays are one of the principal challenges to scaling performance of fine-grain parallelism on many-core systems. In contrast to CMOS logic, propagation delays over on-chip traces are not easily improved by process scaling, due to RC propagation physics [19]. This creates increasing imbalance in computation vs. communication capability. As we increase the number of cores per chip, we also increase the relative distance cross-chip messages must cover, as well as the diameter of the on-chip network, to a degree dependent on the chosen topology.

Figure 2.2 shows the chip size vs. delay relationship for a single network traversal on a simulated 2D-mesh architecture. These communication latencies can become a significant performance problem, especially for fine-grain parallelism.

More clever hierarchical network topologies could flatten this curve somewhat by

constraining the growth in *number of hops*. However, they do not reverse the trend; nor do they eliminate physical distances as a component of latency. Some network topologies are also more vulnerable to hotspots in sub-optimal traffic patterns.

For our experiments we will model a system with a flat 2D-mesh interconnect, and a relatively shallow cache hierarchy, with no regional shared caches (only single-core and globally-shared elements). These are practical choices for simplicity of the study, and mimic the structure of several current and upcoming many-core chip offerings; however, our results should be easily applicable to deeper hierarchies. This is because some of the problems we explore depend simply on the overall number of cores; because shared tiers in deeper cache hierarchies will see similar scaling problems; and because deeper hierarchies still have the potential, in the face of poor locality, to make coherence penalties worse. Another reason to target a flat system is because it helps to isolate the basic communication & synchronization questions we wish to study, from separate questions of regional cache locality optimization. In spite of this, we intend to create solutions amenable to future incorporation of hierarchical locality issues.

Some kinds of applications can leverage many-core systems with very little inter-thread communication, thus reducing harm from the physical factors just described. This is possible when work can be statically scheduled, prefetched, or otherwise performed with higher locality, and planned in advance. Examples of such workloads might include blocked (dense) matrix multiplication [40], as well as temporal blocking in stencil-based calculations [98]. However, not all applications can take advantage of these methods.

This work considers applications where irregular parallelism leads to more significant synchronization needs. Many workloads in machine learning and graph processing, two fields of recently increased relevance, belong to this class. Two other factors increase the relevance of irregular parallelism to many-core performance: As other accelerators, such as GPUs, excel at handling very predictable parallelism, general-purpose cores are more often left to do the irregular work [13]. Finally, even within a single many-core processor, as we succeed over time in exploiting easily parallelized portions of code, the remaining code regions are more irregular and will increasingly

become an obstacle to overall performance [45]. We will explore irregular parallelism in more detail in Section 2.3.

On current shared-memory, general purpose multiprocessors, the most common way to handle synchronization is through primitives such as locks, semaphores, and barriers. There is an implicit communication requirement to these primitives, regardless of how they are implemented, which makes them fundamentally sensitive to chip scaling. However, as they are usually implemented on current systems through sharing of memory locations, this vulnerability is multiplied by the cache-coherence protocols that mediate shared access on most chips. Cache coherence requires multiple network delays to resolve most cache sharing events; and this penalty is even steeper when any of the operations must guarantee atomicity.

The above discussion of propagation latency and cache coherence is one part of why it is especially challenging to scale synchronization-dependent code on many-core platforms. Another set of difficulties is caused simply by the larger number of cores. Synchronization and contention delays grow with the number of threads involved, independent of the underlying communication mechanism, both in the average and in the worst-case. Figure 2.3 is one example of how the *number of steps* required to accomplish a specific task increases with the system's core count. In this example, a thread in a task-stealing benchmark is checking queues in search of work to steal. Though the benchmark in question maintains a constant density of work relative to queues as the system's size increases, the integer number of queues that represents a given *percentage* of the system necessarily increases as well. This situation applies as much to resource contention, such as fighting over network links, as to synchronization through primitives such as barriers.

Some might argue that future breakthroughs in device physics (e.g., optical interconnect [59]), may solve the scaling problems we have outlined. This is not necessarily true. Even with hypothetical low-latency communication, the sheer number of threads can cause latency through contention and synchronization delays. Appropriate solutions will be needed for efficient coordination among hundreds of threads, regardless of the underlying connection technology.

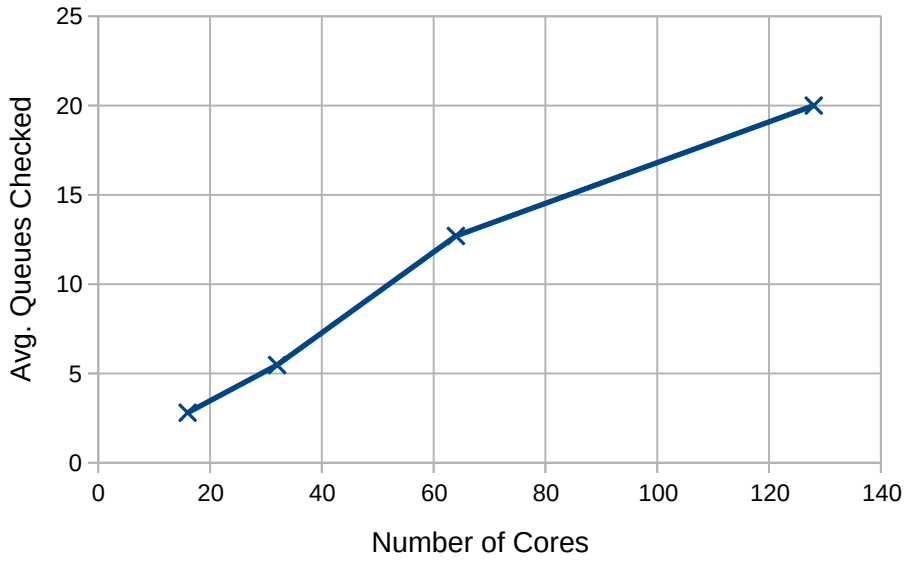


Figure 2.3: Number of task queues consulted during a single search, in a task-stealing benchmark that maintains task density constant relative to the number of queues in the system. As the system's size increases, searching the same "percentage" of the queues requires more consultations.

2.3 Irregular Applications

We say that an application has irregular parallelism when its performance characteristics, such as critical path, available parallelism, memory access pattern, duration of tasks, etc., are unknown prior to execution or difficult to predict. Such lack of knowledge in turn reduces the ability and the benefit of static scheduling, prefetching, partitioning, and other ways of planning ahead. The unpredictability of these applications is caused by dependence on input data [56]. In many of them, the extent of required work, along with dependencies or implicit ordering among necessary tasks, is only dynamically discovered as execution progresses. Irregularity also means that performance characteristics may vary during the course of a single execution [55].

Opposite on the predictability spectrum we find applications such as, for example, dense matrix multiply. The structure of the task graph of a dense matrix multiplication depends simply on the data's row & column dimensions. The compute delay per block, and the memory access pattern, are both predictable. All of this information

can be used to efficiently execute the algorithm with very little communication [40]. This maintains scaling penalties low, even on a large number of cores.

Some applications are only partially irregular. Hashjoin [54], for example, has a data-dependent memory access pattern, which causes some level of unpredictability. But the tasks are statically known, abundant, homogeneous, and free of explicit dependencies. Though these kinds of applications may sometimes be treated as irregular, their needs are different in ways we will later explore.

For our focus on irregular parallelism, we will consider applications from the sparse- and graph- dataset domains. These two areas have seen significant research and optimization focus in the last few years, partly because they are important for contemporary products and services, and partly because their irregular parallelism presents a challenge to fast and efficient execution.

Sparse data is often encountered, for example, in machine learning. When working with sparse matrices (those where most of the entries are zeros), sparsity-specific data storage formats and processing algorithms are used. By letting the data's contents guide the course of the computation [11, 32, 48, 84], they can be more efficient than using dense-data algorithms on sparse datasets.

Many graph applications also have irregular parallelism. Some examples include Strongly Connected Component detection [46], max-flow/min-cut [36], and dynamic mesh refinement [47]. Irregularity may be driven in these cases by the structure of the graph, or by algorithm specifics (e.g., results-dependent processing order, instead of full-graph iterations). In either case, the applications' performance will depend significantly on the dataset's contents.

One popular strategy for the execution of irregular parallelism on multi-core systems is to use a dynamic runtime layer to provide load balance, usually through task-stealing. In such a system, threads expose available work by organizing it into *tasks*, and manage these tasks through shared data structures such as queues. There is a variety of systems implementing task-stealing solutions [17,35], and later sections will discuss performance and scalability questions arising when these are used in a many-core context.

Next we will discuss some of the representative irregular applications used in our

experiments.

2.3.1 Applications Examined

We focus our experiments mainly on three representative graph- and sparse-data applications: Maximum flow or `mf`, Forward solver or `fs`, and GT fold or `gt`. These applications represent real-world problems with interesting data-dependent variability. They differ from each other in the character of their load imbalance and their dynamic task generation. They are introduced in greater detail below.

Maximum-Flow / Minimum-Cut

Maximum Flow / Minimum Cut describes a graph problem encountered in different guises across different application domains. Given a graph with weighted edges, the problem can be described as seeking the best line along which to partition the graph into two subgraphs (one containing a source node and the other a sink node) such as to minimize the sum value of the edges being cut. Alternatively, the weighted edges may be understood as links providing a certain carrying capacity between nodes. The problem then consists of determining the maximum total flow that such a network could accommodate between the source and the sink, and finding out which links are instrumental to that limit. Two examples of domains where this problem is important are distributed processing and image analysis.

Goldberg's [36] push-relabel algorithm (parallelized later by Anderson [12]) is a popular method for solving this problem. Among graph algorithms, it is more complex than others such as shortest path and breadth first search, due to its reliance on transactions between pairs of nodes, and the dynamic evolution of the graph. The algorithm operates on an input graph whose nodes are connected by weighted edges. The goal is to saturate the network's capacity by "pushing" as much flow as possible from a source node to a destination node. This is accomplished by repeatedly applying two main operations. A *push* operation moves excess flow from one node to its neighbor, according to the capacity of their shared edge. A *relabel* operation maintains the desired direction of flow by updating a per-node metric which approximates nodes'

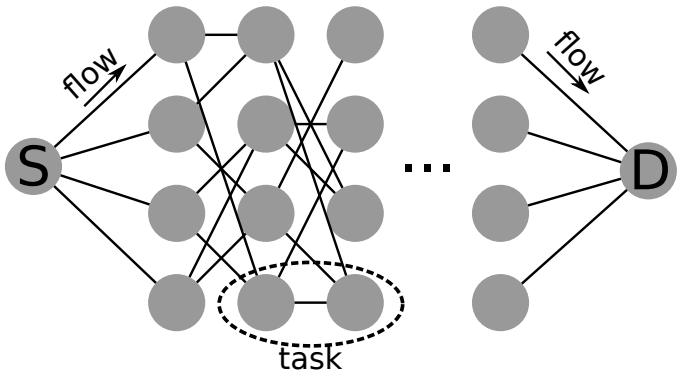


Figure 2.4: *Maximum-Flow* finds the edges in a graph which limit the flow that can be carried between a source and a drain. In Goldberg's algorithm, each dynamic task represents either a node-pair interaction or a relabeling of a node's neighbors.

distance from the sink. Though real-world implementations often layer on extra heuristics for faster convergence, our experiment will focus only on the base algorithm.

Individual tasks in this algorithm are of varying duration, since each task decides whether to push or relabel, and may also contend for multiple locks. In practice tasks last between one and two thousand cycles. They have a variable fan-out factor, as 0, 1, or 2 new tasks may succeed each task, depending on its result. The application's task-graph may even depend on the actual order of execution; such that the total number of tasks is not constant for a given dataset.

Sparse Forward Solver

Sparse linear system solvers are one of the most important application domains in high performance computing, and forward triangular substitution is a key kernel in such algorithms. One popular way of solving sparse linear systems, for example, is a conjugate gradient algorithm with incomplete LU preconditioner [84] that uses both forward and backward triangular substitutions in each iteration until it converges. Due to its limited, fine-grain parallelism, triangular substitution is often a bottleneck when scaling to non-trivial numbers of threads. For example, in the *high-performance conjugate gradient* benchmark [32], a popular new tool for ranking supercomputers,

the majority of programmers' optimization effort is spent on parallelization of triangular substitution. Forward and backward substitution are symmetric, so it is sufficient for us to consider only one of them.

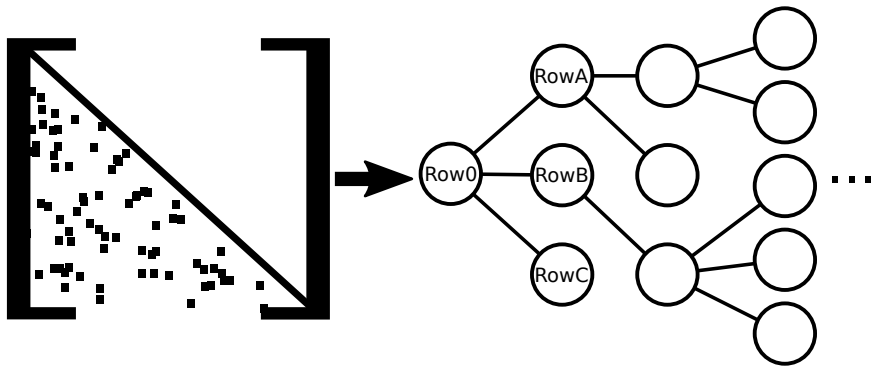


Figure 2.5: *Sparse Forward Solver* solves a sparse lower diagonal matrix row by row, following a dependency graph defined by the non-zero elements. Each row becomes a dynamic task with one dependency per non-diagonal non-zero element.

In forward triangular substitution, the given input matrix statically determines the task dependency graph, and hence the level of parallelism. A non-zero element at location (i, j) in the input matrix creates a dependency where the computation of the ith row depends on the computation of the jth row. In other words, if we interpret the input matrix as an adjacency matrix, the resulting graph is itself the task dependency graph, with each task representing the computation of one row. In this way, the non-zero pattern of the matrix determines the amount of parallelism, its evolution over time, and the critical path. Tasks vary in duration by the number of non-zeros and dependents, but on average last about two thousand cycles. Our main datasets for this workload were computed using code from the HPCG [32] benchmark, and correspond to discretizations of Laplacian operators.

RNA Folding

GTFold [66] is an algorithm that calculates the folded shape of an RNA segment. It does so in incremental fashion, first calculating the preferred shape of all shortest possible subsequences of nucleotides in the given segment, and then combining these,

step-by-step, into longer sequences. The execution time of each such task is dependent on the specific nucleotides present, and can vary significantly with the resulting geometry. Because of the iterative, pair-wise combination pattern, each task may trigger 0, 1, or 2 successors. Maximum parallelism occurs at the start, and narrows linearly.

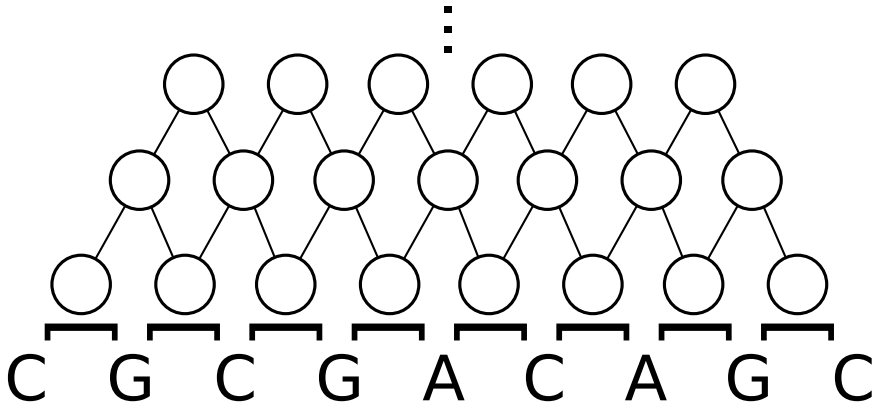


Figure 2.6: *GTFold* computes the preferred shape of an RNA sequence by starting with sub-sequences of length 2 and combining them incrementally into longer ones. The resulting task graph has a triangular shape, where all but the initial set of tasks have two dependents, and each task's duration is determined by its specific subsequence.

On the surface, GTFold has more structured parallelism than `mf` and `fs`, because of the straightforward shape of its task graph. But its tasks are variable-length and input dependent; its available parallelism varies over time; and its tasks depend on one another. Tasks are also longer on average than those of our other applications: the average task duration is around fifteen thousand cycles. This difference relative to our other workloads affects trade-offs regarding load balance.

Other Applications

We also include in some of our experiments two applications with statically generated workloads of independent tasks: Hashbuild or `hb` and Hashjoin or `hj`. These represent the more "embarrassingly parallel" kinds of applications whose only source of imbalance is a more or less irregular memory access pattern. They have thus been

the target of some load-balancing efforts in the past.

`hb` and `hj` are extracted from two phases of a common hash-based algorithm for database table join operations. Our implementation resembles that described in [54]. Hashbuild uses multiple threads to construct a hash table using keys from the original inner table. Hashjoin streams over the rows in the outer table and uses the constructed hash table to seek matches to inner table rows. Both applications have statically generated tasks, as the tables are known at the start, and no dependencies exist to require dynamic generation during execution. There is also consequently wide parallelism.

Finally, for a few more specific comparisons, we have created a small number of microbenchmarks.

`rtb` simulates a dynamically-generated workload with very short tasks, and a persistent need for load-balancing. It is designed to stress the overhead of dynamic task-stealing as a load-balance method. It works by executing 100 generations of tasks, with each generation having as many tasks as there are cores, but generating such tasks in an unbalanced manner. As only a fraction of each generation of tasks creates the whole next set, frequent load-balancing help is required to maintain efficient execution. Its use of dummy (nop loop) tasks avoids introducing unrelated processing or traffic into the system, and focuses the stress on task management and its basic overheads.

`hist`, as its name implies, is a microbenchmark that processes an input dataset into a shared histogram. In our case the histogram has 256 bins, and the input dataset is evenly divided among worker threads and handled in a straightforward manner. This is meant to stress fine-grain, unpredictable sharing of atomically-incremented data.

`tick` is a microbenchmark where N threads each execute a number of lock/unlock cycles, choosing randomly from among 0.5N ticket-style locks. This is meant to stress both contention and overhead in lock acquisition and release, modeling the situation that may arise, for example, in graph applications where nodes are protected by locks.

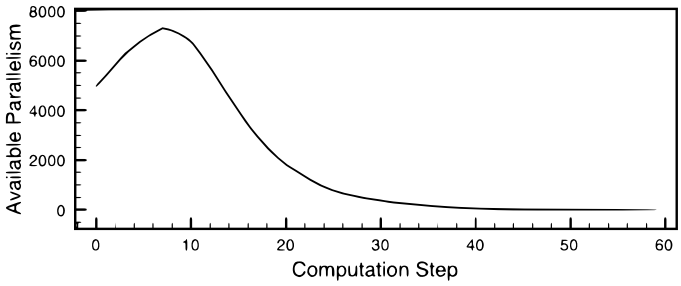


Figure 2.7: From Kulkarni [55], parallelism over time for one execution of a Mesh Refinement algorithm, when modeling an infinite number of cores. Computation steps in this context refers to algorithm iterations, independent of core count.

2.3.2 Available Parallelism

Dynamic generation of tasks is a frequent characteristic of irregular applications. Even when the complete set of required tasks is known at compile time (which is unlikely in practice), dependencies among tasks can mean that only a subset of them can be executed concurrently at any given time [26]. We refer to this "ready" subset of tasks as *available parallelism*.

When we say, then, that certain applications show "variability in parallelism during execution", we refer to changes in this value over time, occurring naturally as in-progress tasks either enable or create a greater or smaller number of dependent follow-ups. We can illustrate an application's available parallelism over time using a parallelism profile [26].

Figure 2.7 shows one example of an irregular application's parallelism profile, from a published study [55] which simulates a computer with an infinite number of cores in order to reveal an algorithm's maximum parallelism potential. At each timestep in the algorithm, it shows the number of tasks available, corresponding to the maximum number of threads it could possibly occupy concurrently given a capable system. The mesh refinement workload in this case peaks at more than 7000 available tasks, but then tapers down and spends many timesteps well below 100. What is important about many irregular applications, including those used in this thesis and introduced in Section 2.3.1, is that the profile's overall shape and scale of variation are difficult or impossible to predict. Furthermore, once we choose a "real" (non-infinite) system

for execution, the effective profile might be further shaped by the finite core-count, altering the relationship between abundant and scarce periods.

Available parallelism can be influenced by changes in task granularity [26]. For example, to better utilize a many-core system, we might partition a workload into a larger number of smaller tasks that can be executed concurrently. This increase in available parallelism, however, may carry a penalty in task management overhead. Our chosen applications have generally fine-grained tasks, potentially supplying sufficient parallelism for a many-core system, but task management must be carefully accomplished.

As other works [26, 55] have shown, there is no shortage of irregular applications where, for some periods, available parallelism is more than enough to occupy a hundred-core system. However, periods of relatively limited parallelism are also found throughout. The performance impact of these *narrow* periods has in the past been downplayed, but it grows along with the number of cores in the system, in ways that can be understood through Amdahl's law [45]. Appropriate task management solutions for irregular applications and many-core chips must be able to thrive across these different operating regimes presenting different fundamental performance challenges.

When available parallelism is far larger than the number of hardware threads, we will say that the system is *thread-constrained*. Under this regime, the most useful steps a task management system can take for performance purposes is to maximize overall productivity, by minimizing its own overhead. Any actual load-balancing intervention must be infrequent, and use as few core cycles (globally) as possible. This is the regime generally assumed by current offerings in dynamic task management for multi-core chips, such as Chase-Lev and Cilk [24, 35]. Relative to the core counts of current mainstream processors, this is probably the right approach.

When available parallelism is smaller, equal, or only somewhat larger than the number of hardware threads (to an extent dependent on the degree of load imbalance), we will say that the system is *task-constrained*. Overall management overhead may not be the most important concern in this case. We can take a cue from research on networked clusters, which, despite being quite different architecturally, do often

contain hundreds of cores. First, load-balance interventions may become both more frequent, and more laborious [31]. Second, *overall* task management overhead may not be as important as specifically avoiding overheads on the threads that are at any time making progress [82]. When properly handled, task-constrained situations may still result in full system utilization; these two conditions are not mutually exclusive. Finally, it is worth pointing out that task-constrained operation may, by its nature, increase the application's vulnerability to the kinds of communication delays we have previously discussed as a growing challenge of many-core systems.

Whereas under regular parallelism one may see extended periods of task constraint as evidence of a poor match between system size and problem size, we will argue that (a) in irregular parallelism, it may be unavoidable (and possibly more optimal) to choose a much larger system than the *narrow* portions of the workload would suggest, and run partially idle during those times; and (b) in real-world situations, it might be inefficient or impossible to sufficiently understand a dataset in advance, in order to make a careful choice regarding, e.g., mapping to an appropriately-sized system. It is therefore important for the real-world usability of both software and hardware to be able to handle thread-constrained and task-constrained conditions dynamically, in the best way possible.

2.4 Existing Runtime Systems for Irregular Applications

2.4.1 Task Stealing Models

At the most basic level, we assume a fork-join parallel programming model [27]. The application starts in serial execution with a single "master" thread, which spawns a set of worker threads. These workers persist until program completion. During parallel portions, we follow a dynamic, task-stealing runtime model [17] in which both master and worker threads repeatedly retrieve and execute tasks from task queues. Any thread can create and enqueue new tasks. A task is generally represented by a function pointer and a set of arguments. We assume that dependencies are resolved

before a task is inserted in a queue, such that all queue contents are ready for possibly concurrent execution; at this fundamental level we make no assumptions about the ordering of items in these queues.

In a shared memory system, like in most multi-core chips, maintaining a shared state for task storage is quite easy. However, manipulation of the shared state (i.e., task enqueues and dequeues) requires synchronization. If we put all tasks in a single queue, contention in updating that state, due to the required synchronization, grows rapidly with the number of workers. On most systems, even at modest numbers of cores, this contention limits parallel performance unless tasks are very coarse-grained.

Thus, actual implementations of dynamic runtimes typically use one queue per worker, so that each worker can enqueue and dequeue to its "own" queue efficiently most of the time [24]. The downside is that when a worker finds its queue empty, it must either sit idle until the end of the current parallel phase, or invest time in finding a task in another queue. This process of *task or work stealing* [17, 25, 35] has become widely used, with a variety of runtime layer implementations in existence, and a significant body of research exploring best practices. The next section will discuss some of this work.

2.4.2 Task Stealing Runtimes

The basic groundwork for task stealing was laid in works such as the original Cilk [17]. A number of the parallelism toolkits currently in use implement task stealing in a similar way, including, among others, Intel's TBB (with the latest version of what was originally Cilk), and IBM's XWS [25, 35].

As described in Section 2.4.1 above, task-stealing implementations focused on multicore chip performance tend to employ one queue per worker thread. This is done to avoid centralized contention and to reduce the overhead of each thread's enqueues and dequeues. Even on systems needing "frequent" load balance, enqueues and dequeues constitute the more common case compared to the frequency of stealing events. To further manage contention for queues, especially the impact of stealing attempts on a queue's owner thread, non-blocking protocols [24, 35] have been formulated. We will

use one such protocol, *Chase-Lev*, as a starting point for our design. Its advantage lies in further speeding up a thread's frequent accesses to its own queue.

We will now explore a few of the research directions seeking to maximize the performance and efficiency of this general approach to task-stealing.

Many of the basic "best practices" for global task-stealing efficiency were developed a while ago through topics such as *steal half* [16, 31, 42] and *target randomization* [35]. Both are well-accepted and effective in thread-constrained conditions (when tasks are numerous relative to threads). They are also attractive because they require no sharing of information among threads. These kinds of best practices will be used in our baseline implementation, and we will re-evaluate their appropriateness at the hundred-core scale.

The debate between "work-first" and "help-first" [41, 94] policies is an example of the tension in task stealing systems between per-thread efficiency and critical path progress. The specific question is whether newly created tasks should be immediately executed, with parent scopes paused and enqueued, as may be the case in single-thread sequential execution; or whether the new tasks should be enqueued until the parent scope is done. These two choices accrue overhead in different ways. Similarly, one can ask whether, from an energy allocation perspective, it makes sense to favor thieves or victims [82]. The overarching question in these discussions is the balance between "work overhead" (which accrues at the start of every task) and "critical path overhead" (which accrues only on steal events). Our need to obtain good results in conditions of both abundance and scarcity may require reexamining policies on this front.

Work-sharing vs. stealing [6, 17, 34] arguments tread related ground, though often the intent is to eliminate the need for queue synchronization. This can be done by having active threads take the time to push new tasks to others, instead of having the others steal them. This idea is generally counterproductive because of the overheads it introduces. Only in some distributed systems can it result in sufficiently reduced implementation complexity and faster work discovery.

Queue splitting is another attempt to manage contention by reducing the need to synchronize [6, 31]. In these systems, the thread that owns each queue has (mostly)

synchronization-free access to it; a guarantee achieved by restricting others' stealing to only a portion of said queue. This can indeed be worthwhile in times of relative abundance of work. It becomes less helpful when queues are short, or when they may be empty, both frequent under irregular parallelism.

Task splitting, batching, and coalescing [25,94] are further techniques attempting to reduce overhead, in this case by combining the benefits of both coarse- and fine-grain task management. This kind of flexibility is not practical in all situations; it tends to be at its best in "divide and conquer" or recursive algorithms, which are more amenable to runtime decisions about task granularity.

Finally, it is worth mentioning that the performance challenges of fine-grain task stealing have motivated some proposals to move beyond software optimization and implement dynamic task management in hardware [57,60,86]. Carbon [57], for example, used hardware for both task storage and high performance stealing. ADM [86] achieved similar performance while storing tasks in conventional memory; it further regained flexibility through software scheduling policies. However, it relied on active-messaging mechanisms, which involve special hardware of a different sort. Both of these solutions have significant drawbacks that we seek to avoid in our work; we will discuss this hardware angle in greater depth in Section 2.5.2. Hardware concerns aside, there are aspects of the implementation of these systems, such as smarter communication patterns, that are worth exploring in a fully-software solution, something the cited works declined to do.

2.4.3 Shortcomings for Many-Core Systems

We study task-management on chips with hundreds of general-purpose cores. Now we will discuss why existing research in task-stealing has poorly explored this space, and how existing solutions are inadequate.

Most research into task stealing in shared-memory multi-cores is necessarily carried out on available hardware. Consequently, its scope is generally limited to chips with at most tens of general purpose cores. We have discussed how on a hundred-core architecture, both increasing physical delays, and the need to interact with a greater

number of threads, creates more significant communication overheads and increased contention over specific resources. But with current hardware's small core counts and relatively short network distances, the effect of these scaling factors is limited. Moving into the hundred-core era will have an effect on the efficiency and performance of existing task-stealing runtime designs through excess contention among worker threads and harder to ignore communication delays.

Contention can be addressed through changes to the runtime. For example, we can impose a hierarchy on the threads' communication pattern, to bound the number of participants at each level. This carries a risk of worsening latency, but may later enable optimization when executing on hierarchical cache hardware. Such a structure would also enable us to re-evaluate reliance on completely random stealing; though attractive in its simplicity, randomization can be inefficient and redundant in ways that become a liability at large core counts.

Fundamental communication delays are not addressable from a software perspective; however, communication patterns resulting from fine-grain sharing and cache coherence protocols multiply the impact of physical delays on current systems. There is no lack of proposals for accelerating task management through extra hardware, as we will see in Section 2.5. For practical reasons we would like to avoid such narrowly-targeted hardware; instead, in Chapter 4 we seek solutions for more generally reducing the coherence message traffic resulting from concurrent accesses to shared memory.

Figure 2.8 is an example of how task-stealing can increase in cost with increasing system size, even when available parallelism remains constant relative to core count. It shows the duration in number of cycles of an average steal attempt, at various system sizes, in a state-of-the-art runtime during a run of our `rtb` benchmark. This dynamic workload requires continuous fine-grain load balancing. Though the benchmark actively maintains the average available parallelism per core as the system scales, the cost of even a successful steal is increasing with system size. This is a consequence of two factors: First, an increase in the discrete number of queue checks per steal attempt (e.g., a constant percentage of a larger number); second, the increased latency of each such event, resulting from contention as well as physical travel delays.

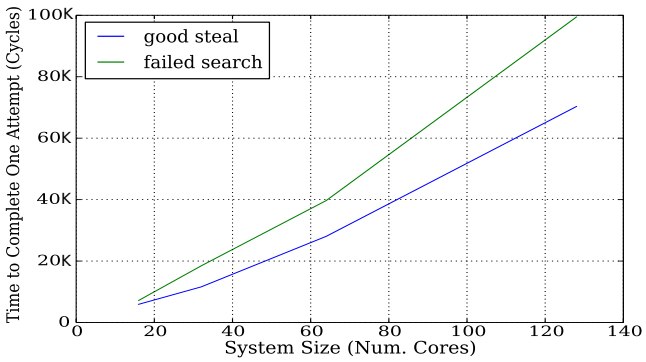


Figure 2.8: Duration in cycles of a steal attempt in a state-of-the-art runtime, compared to the number of cores. The `rtb` workload used here maintains constant available parallelism relative to system size at each point, yet the time required to execute (or decide to fail) a steal increases.

Another factor affecting the performance of task stealing, is the end of the long-time assumption that available parallelism is plentiful compared to the hardware's core count. This abundance assumption is less often true when we are executing irregular applications on hundred-core chips. Though many irregular applications are often capable of thousands of concurrent tasks, lengthy periods of narrower parallelism are also frequent, and what parallelism exists must be dynamically discovered. The time required to steal tasks increases in relevance, both because it is more likely to be on the critical path during narrow parallelism sections, and because task-constrained operation makes the stealing searches more time-consuming to begin with.

Because of all of the above reasons, existing task-stealing solutions for multicore chips are not appropriate for irregular parallelism on hundreds of cores. Current best practices scale poorly into the hundreds of cores, suffering from long stealing times and high contention. There is significant waste of both compute and network resources when we rely on stealing and management policies appropriate for few cores and abundant parallelism.

In certain architectural domains, systems already exist which do tally up to hundreds or even thousands of cores, and solutions exist for scaling problems analogous to what single chip many-cores will face. We will consider some ideas from these areas, while being mindful of architectural differences.

In cluster systems, multiple chips communicate over a board, and multiple such nodes may communicate over a wired network. Compute clusters suffer communication latencies at chip and board boundaries which are orders of magnitude longer than we observe across one chip. David et al. [30] indicate, for example, a latency increase of between 2x and 7.5x when cache requests must cross AMD Opteron sockets vs. being issued within the same socket, even under zero contention. Latencies between different boards are even longer. These latency differences, along with NUMA and distributed memory models, change the relationship between computation and communication, and emphasize different task management needs compared to single chips.

In this context, task stealing has been explored frequently [14,31,69,80]. One general difference of interest to us is that alternatives to flat randomized stealing have been more frequently explored, due to steeper penalties or hard barriers when traffic does not match the hardware's node structure. However, other differences make these results hard to generalize to our case. Cluster runtimes must often manage the application's data, explicitly moving it across NUMA boundaries to where tasks are executed; something our situation does not require. Also, because of long network latencies, cluster systems tend to favor much coarser-grain parallelism. Finally, runtimes must often implement synchronization layers on top of the chosen interconnect and NUMA systems.

Another field where one may say that dynamic task-management exists for managing hundreds of cores is GPUs [92, 95]. But once again, this situation only superficially resembles our scenario; there are architectural differences that drive solutions in different directions. For example, thread group/warp scale communication and coordination is of crucial importance in GPUs; explicit cache management and data movement may be necessary for correctness and safety, as full cache coherence is uncommon; and, until very recently, even generating a new task on a GPU was not actually possible [97], requiring hybrid CPU/GPU solutions. Once again, architectural differences mean that GPUs are at their best on very different applications and algorithms compared to CPUs.

There is, then, a lack of appropriate task-stealing solutions for our requirements:

Applying the lessons from existing multi-core task-stealing work, while efficiently leveraging hundred-core chips, with understanding of on-chip communication penalties; allowing irregular applications with fine-grain, time-variable parallelism to maximize progress, while being smart and efficient about periods of lower parallelism; and avoiding purpose-specific hardware in order to maximize both flexibility and the prospect of practical adoption.

2.5 Communication and Synchronization Support in Many-Cores

We have mentioned that we wish to avoid proposals with task-stealing specific hardware. However, we do not wish to avoid hardware proposals altogether. After we achieve a software runtime with the necessary qualities, we do target the on-chip communication patterns resulting from the runtime's activity. We do so at the most fundamental level, hoping for a broadly-useful, versatile solution.

Hardware acceleration of synchronization activities on multicore systems has been a topic of research ever since it was clear that parallel programming in general could be served by a relatively small number of primitives and sharing patterns. However, work in this area has often been too single-purpose to adopt into a general purpose architecture, has not targeted shared-memory collaboration on coherent systems, or has forced significant trade-offs in functionality, compatibility, or practicality for implementation and usage.

2.5.1 Current Hardware Support

We will start with a review of hardware features relevant to communication that might be considered commonplace on currently popular systems.

We assume that a many-core chip provides the ability for cores to communicate implicitly through a shared memory model, by reading and writing to the same locations. We also assume automatic coherence among caches serving different cores. We assume that the hardware and instruction set support basic, general-purpose

atomic operations, sufficient to implement synchronization primitives [43] such as spin locks and barriers for example. It is also not unusual for contemporary systems to support streaming-oriented memory access (such as with cache bypass instructions), intended for bringing large amounts of low-locality data from/to DRAM while protecting higher-locality data in cache. While not strictly a core-to-core communication feature, it is important because bringing data to and from a core while bypassing local cache is something that can be repurposed for reducing cost and latency of on-chip communication.

Certain parallelism-oriented features which are slowly coming to market but were not used in either the applications or runtimes in this work include Intel's transactional extensions (TSX) [99], available in recent chips, and memory versioning, as in Oracle's upcoming M7 [10].

We have already mentioned that communication and synchronization delays have been a more urgent problem in cluster, supercomputer, and GPU architectures. Domain-appropriate hardware acceleration features exist to address some of these cases. There are some interesting ideas to keep in mind.

In distributed-memory clusters, latencies on inter-node data sharing have accentuated communication bottlenecks sooner. Some networking solutions have, accordingly, included hardware to compensate. Infiniband [62, 71] is one popular example: interface hardware can include features to facilitate updates to shared data in remote nodes with separate memory spaces. The capabilities and usage model of these features are shaped by typical task durations, network delays, and memory models of cluster systems. For example, some of the related work has focused on achieving *one-sided operation*; that is, eliminating the need for the processor that owns a given memory space to be interrupted in order to service remote updates to data in that space.

Systems falling into the supercomputer category tend to have greater focus on scale-up performance of specific applications, and have often sought low-overhead communication more eagerly than more generic clusters. Some systems of this type have included support for certain atomic operations in the memory hierarchy [33, 39] as a way to achieve high processing throughput without saturating I/O or network

links. The basic concept of moving atomic ops towards the data is important to us, as atomics are crucial to the use of synchronization primitives, and atomicity is a significant factor in cache system overhead. In supercomputing architectures, the specific choice of atomic operation, as well as its implementation, have usually favored data throughput rather than control-flow, and have tended to be tightly coupled to specific programming models (e.g., stream processing, combining networks).

Finally, some recently released GPUs offer ISA extensions implementing a measure of cache coherence [50], through on-demand flushing and refreshing of local cache. This is a step towards ease of use for such systems, which previously have offered little to no coherence; and is necessary for close integration into heterogeneous designs. However, the necessary programming complexity and safety risks of such a usage model are among problems we have sought to avoid in our solution.

2.5.2 Relevant Research Trends

Beyond the technology currently available, there are many research directions relevant to hardware acceleration of on-die communication and synchronization activities.

Perhaps the first obvious approach to accelerating a problematic algorithm is to implement a full solution to the specific high-level problem, in hardware. One clear example of this, for the specific case of fine-grain task-stealing, are some of the runtimes mentioned in Section 2.4.2, such as CARBON [57] and HAQu [60]. However, these are only representative of this category; similar kinds of solutions exist in research for other high-level problems.

In general, though each such solution may well achieve performance and/or efficiency in its target use, that is far from the only criterion that makes a solution desirable. Narrow scope of application can be a barrier to adoption in anything but a specialized system, given the necessary hardware investment. There are other problems with hardware solutions at this level. Some of these include a loss of flexibility due to hard-coded choices; greater system complexity resulting in challenges for both programmers and architects; and complications when integrating with, for example, a multiprocessing OS.

Somewhat removed from high-level, problem-specific hardware, are features such as active messages [96]. Active messages seek to provide an alternative system for communication and short operations on remotely cached data. With proper use they allow for speedup through avoidance of protocol overheads in communication-dependent code. But this solution also suffers from key drawbacks: First, its use requires adopting a significantly different programming model alongside coherent shared memory, which is a barrier to adoption by other than very specialized programmers. Second, real-world implementations must solve significant challenges regarding deadlock avoidance and fairness, especially in multiprocessing operating systems. Finally, similar practical implementation and cost barriers apply as with the earlier high-level hardware proposals. Combined, these factors make it difficult to justify at implementation time on a general-purpose processor.

Taking one more step away from higher-level solutions, there is also a wealth of research on hardware support of more basic and popular synchronization primitives, such as locks and barriers [5, 87, 91, 93]. Targeting primitives rather than higher level constructs preserves some flexibility and breadth of application, while still improving performance. There are at least two main drawbacks, however. First, hardware-based primitives are generally subject to resource limitations leading to fall-back complexities and OS-level sharing questions. Second, to some extent they still reduce flexibility by hard-coding details. For example, as research by T. David has shown [30], different locking mechanisms may have different strengths; but a hardware implementation must usually choose one. In general, the case for inclusion of these systems in general-purpose chips is still somewhat difficult and limiting.

A different research direction seeking to reduce penalties related to propagation delays is *near-data computing* or *processing-in-memory*. The recent efforts by J. Ahn [8,9] are a good example. These works attempt to improve efficiency and throughput during bulk data updates by placing extra execution elements closer to memory. With the target uses being quite different from our on-chip synchronization latency concerns, the design of the hardware and the risk-reward calculus of off-loading are different from what they would be in our case. But this line of thinking provides one plausible idea for addressing shared-memory use with atomic operations, if it could

be implemented efficiently enough at an on-chip cache level. Even then we would have to be careful when thinking about the resulting programming model.

At the level of basic cache dynamics, many architects have sought to improve efficiency in the face of variations in data locality, most often through run-time monitoring. Kurian et al. [58] is a good example of this, combining streaming-type uncached accesses with per-core locality monitoring to assign different policies to different lines. Similarly, Nilsson et al. [72] enhance a DRAM controller with a cache whose contents are dynamically determined through monitoring of the memory channel. For our purposes, these ideas would be more compelling if on-chip communication among cores were more often the target, rather than single-core locality. A second concern is our need to ensure that atomic operations are part of the scheme.

Park et al. [78] also target cache-coherence overheads, but without monitoring (avoiding the time and space overhead), and specifically targeting communication (producer-consumer). It uses programmer knowledge transmitted through ISA support to trigger direct access to remote cores' local caches. These priorities are a step closer to what we require, and the solution also carries low added complexity for both programmers and hardware designers. It still lacks, however, mention of atomic operations; it also fails to incorporate use of shared caches, leaving many communication patterns unserved. We will use its ideas as one of our starting points for hardware proposals in Chapter 4.

2.5.3 Hardware Needs of Irregular Applications

There are, as we have seen, interesting proposals for hardware acceleration of on-chip communication and synchronization at multiple levels. We have briefly mentioned why some of these are not suitable for our purposes. In this section we will focus on the hardware needs of irregular parallelism specifically. This will highlight how the solutions we have seen so far are inappropriate for the purpose.

Irregular parallelism requires fast and efficient communication (e.g., for propagating knowledge of the changing state of the system) and synchronization (e.g., for efficient fine-grain task steals). Existing cache-coherence mechanisms on multi-core

chips will not be appropriate for these needs when scaled to many-cores.

Irregular applications can exhibit a variety of available parallelism profiles, which may affect a runtime's choice of "best policy" at different times and on different topics during execution. Therefore we would like to maintain flexibility. The applications themselves may have communication patterns beyond those related to task management that programmers would like to accelerate; and, with variable parallelism, the main program may at times occupy only a portion of the system, suggesting that both power management and multi-processing should be able to be incorporated.

None of the existing or prior proposed hardware fulfills all of our above requirements. Some, such as profiling-based cache insertion optimizations, are easy to use, but lack flexibility or performance. Some, such as hardware task-stealing constructs, offer performance but are too specific, lacking flexibility. Others, such as active messages, are flexible and performant, but would require substantial amounts of new code and different programming models. Finally, we want, if possible, to avoid having to include multiple single-purpose hardware features in an architecture to accomplish our goals.

2.6 Summary of Motivation and Goals

We expect that general-purpose, many-core chips will be an important part of the near-future computing landscape; they show promise as flexible and capable platforms for general-purpose code with varying levels of predictability and parallelism. However, this promise is tempered by the increasing expense of communication among large numbers of cores. To the extent that more regular parallelism is efficiently exploited by existing technologies, irregular code will become an increasingly important [13] stumbling block for overall performance, and penalties for fine-grain communication will be a large part of the problem.

At the same time, irregular parallel applications are becoming more important due to the popularity of related domains such as machine learning and graph analytics; and they are becoming a focus of research due to the challenges they present for fast, efficient execution. Though parallelism in these applications can peak at hundreds or

thousands of concurrent tasks, significant variation is observed across applications, datasets, and over time. This makes it difficult to efficiently plan or predict execution, leading to reliance on dynamic scheduling, which creates greater dependence on communication.

In view of these trends, we believe it is important to enable irregular parallel applications to execute more quickly and efficiently on many-core chips in the hundred-core scale.

From the application's point of view, the first necessary step is to both improve the effectiveness of the dynamic runtime, and reduce its task management overhead, by designing it with many-core chip platforms in mind. This may require significant changes to how stealing is done at large core counts, improving performance and system utilization, especially during task-constrained operation. It may require freeing unnecessary cores during times when the workload allows. And it may require ideas for increasing the system's flexibility in the face of changing conditions.

From the hardware point of view, given dynamic scheduling's greater reliance on fine grain communication and synchronization, and the fact that current platforms rely on shared memory and coherence for these activities, we will seek ways to reduce on-chip communication penalties without radically changing the programming model or significantly overhauling the architecture. We intend to do this by targeting specific communication patterns that defy the cache's locality assumptions, and introducing ways to more efficiently match the movement of data to the intended communication pattern. We seek to reduce these inefficiencies without requiring substantial user code rewrites or losing compatibility with existing systems and programming models. Finally, we intend for our solutions to be practical to implement and as generic as possible, for better prospects of inclusion into general purpose hardware.

Chapter 3

Task-Stealing Runtime for Irregular Parallelism

3.1 Introduction

As discussed in Chapter 2, certain types of applications, including many of those operating on graphs and sparse datasets, exhibit large variations in available parallelism during execution and across datasets. We refer to this as *irregular parallelism*, and it makes static scheduling difficult, inefficient, and unreliable. Dynamic scheduling runtimes based on task stealing have been proposed for these kinds of situations.

In the multi-core chip realm, a rich body of work has explored the design of such runtimes. However, the combination of irregular, unpredictable parallelism with upcoming hundred-core chips presents new challenges for this dynamic approach. For example, though the mesh refinement workload earlier presented in Figure 2.7 is capable of generating, at its peak, over 7000 concurrent tasks, greater than 50% of the algorithm's timesteps can muster less than 128 concurrent tasks. More importantly, the height of the peak and the shape of the long tail for a given execution are determined by the input dataset rather than the algorithm.

For these applications, we observe that parallelism exhibits different phases relative to core count. The unpredictable mix of these phases of either relative abundance

or scarcity of parallel work leads to inefficient utilization of hardware resources. Optimizing the performance of applications exhibiting this kind of parallelism requires detecting and addressing both types of phases, each with different performance characteristics, placing conflicting demands on the runtime. In task-constrained phases, it is important to reduce stealing overhead, optimizing the critical path by quickly locating and distributing new work to idling cores. While doing so, we must occupy the smallest number of cores necessary, to create opportunities for sharing or managing power use on the remaining cores. During thread-constrained or "task abundant" phases, overall runtime overheads must be as small as possible, and all threads must be kept busy. This requires maximizing the efficiency of the runtime, to allow the system's resources to focus on the application's work.

In this chapter we design a task stealing runtime capable of scaling irregular workloads to chips with hundreds of cores, by handling both thread-constrained and task-constrained phases in ways that lead to high performance and efficient use of the system.

We first characterize in detail the parallelism in representative irregular workloads. We then set up a baseline runtime representing best practices in many-core task-stealing. To improve throughput and utilization when stealing during task-constrained phases, we analyze variations on work-pushing, which some have found appropriate for high-latency situations in distributed systems [34]. We discover that one fundamental problem is the time required for even a *single round* of searching through the system. This search time makes up the majority of the time spent in runtime code, and is made worse by threads failing to gather and share information about queue occupancy.

We propose a set of compact data structures for efficient sharing of information about both queue state and thread availability. This design enables low-latency lookups by both task producers and consumers, and its hierarchical logical structure reduces contention among threads and enables decision-making in response to changing conditions. It requires no hardware support beyond cache-line level coherence and existing atomic operations.

When tested on a 128 core system, this design reduces stealing delays by up to

13x through faster searches and lower queue contention. Unlike earlier hierarchical task management systems in single-chip work [57, 86], our design is implemented in software and preserves a flat, single level of queues.

Having reduced search times, we then improve efficiency in task-constrained phases. Existing runtimes designed for thread-constrained (task-abundant) operation waste energy and core cycles whenever tasks become less plentiful. We analyze ways to right-size the system, releasing idle cores when possible for other work or power savings, while maintaining the prompt discovery of new work which is essential for critical-path performance. We find that an approach incorporating both task-stealing when thread-constrained, and work-pushing when task-constrained, works best.

We also use the information in our tree structure for dynamically adaptive runtime behavior. We find that, when work is abundant, it is beneficial to be selective about victims; however, when work is scarce, the speed of single-task stealing should be prioritized instead. The runtime also responds to the availability of idle threads by increasing the frequency of updates to its own metadata, since idle workers can update this without interrupting useful work. Adaptive approaches reduce the need for manual tuning of runtime details, and are appropriate to the irregular nature of our workloads.

Our 13x speedup in locating work reduces applications' overall core-cycle usage by 29% on average, opening the way for power optimizations and multi-processing. By making smarter stealing choices, greater core availability leads to speedups of up to 30% over current state-of-the-art task stealing, with an average of 15% on target workloads executing on the same simulated 128-core system.

3.2 Background

3.2.1 Application Parallelism Model

We assume applications are written using a fork-join model [27], where execution starts with a single "master" thread, and then spawns worker threads to initiate a parallel phase. Upon the end of the parallel phase, the workers join and the program

```
void* worker_thread_func(void* argPtr) {
    while (my_task = dequeue_task()) {
        // ... process my_task
        // create and enqueue new tasks as necessary
        enqueue_task(new_task);
    }
    return NULL;
}

int main() {
    // ... initialization ...

    // seed initial task(s)
    enqueue_task(initial_task);

    // begin parallel section
    for (int i = 0; i < num_threads; ++i) {
        pthread_create(args[i],&worker_thread_func);
    }
    for (int i = 0; i < num_threads; ++i) {
        pthread_join();
    }

    return 0;
}
```

Figure 3.1: Basic sketch of our parallelism model.

returns to serial execution. A program may undergo multiple such parallel phases over time. During a parallel phase, we assume work is distributed among the worker threads using a dynamic scheduling runtime. All threads repeatedly retrieve tasks from task queues, and any of them is able to create and enqueue new tasks during execution. A task in this context can be thought of as a function pointer and a set of arguments. All tasks in the queue are ready to be executed in any order.[1] Figure 3.1 is a brief example of the general structure of an application following our model.

[1]Some task queue models allow for dependences between tasks. We assume that dependences are resolved before inserting tasks in the queue.

The task queue is simply storage for ready tasks, and can be implemented in many ways; in practice, there is often one queue per worker, for performance reasons. If the queue is empty, the worker thread seeks or waits for new work, or waits for all workers to be finished. When all workers are finished, control passes back to the master thread to continue serial execution.

3.2.2 Irregular Parallelism Examples

In recent years, there has been a surge in research on applications operating on sparse datasets. This includes graph analytics (e.g., [1, 1–3, 22, 64, 79, 89]) and sparse linear algebra-based applications (e.g., high-performance conjugate gradient [32]). While operations on dense datasets often have statically known data dependencies, a sparse operation's dependency graph is a function of the input. This leads to parallelism that is both reduced and harder to extract compared to many dense operations. Furthermore, the amount of parallelism available can vary over time in hard-to-predict ways.

Dependencies among tasks in a sparse-oriented algorithm are determined by the structure of the input data; but the amount of work per task also varies due to the data's content. For example, a task's complexity may depend on the degree of a graph vertex, or on the number of non-zero elements in a matrix row. Both of these metrics may be non-uniform [15] in sparse datasets. As was discussed in Chapter 2, one consequence of variable and unpredictable amounts of work per task is the need for dynamic load balance to keep as many threads busy as possible — we cannot rely on a static assignment of work to threads, as many threads may be idle for a large portion of the program's execution while plenty of work waits to be done by busy threads.

Skipping unnecessary work allows sparse algorithms to break dependencies that would exist in dense datasets, providing new opportunities for parallelism. For example, when we apply gaussian elimination to a dense matrix, e.g., to solve a system of equations, we must complete work on one row before beginning work on another. In contrast, when solving a system represented by a sparse matrix, many rows may

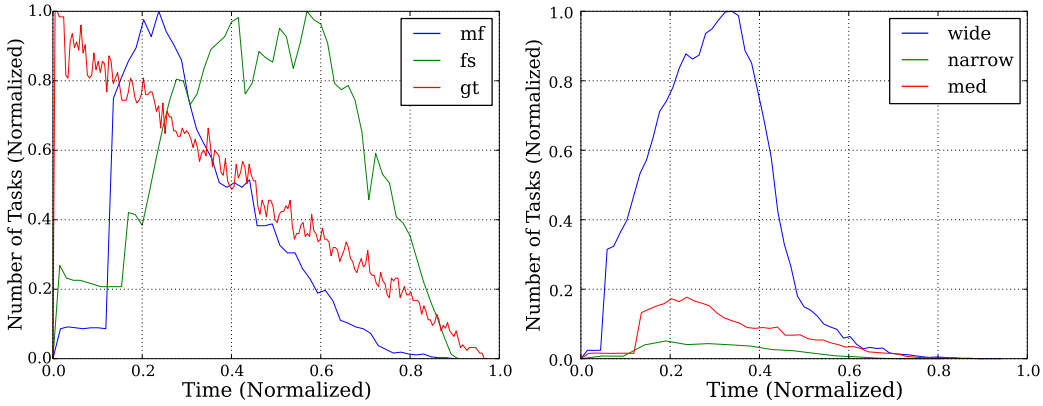


Figure 3.2: Variable parallelism over time from three different applications (left) and from three different inputs for the same application (right).

actually be independent because the elements that would create a dependence are zero. A key effect of this is that we can't know how many parallel tasks comprise a computation and/or what the dependency graph will look like until we analyze the input. In some cases, such an analysis would be as expensive as the computation itself. Thus, operations on sparse datasets often result in irregular parallelism: they have an unpredictable task dependency graph, where both relationships and task durations are input-determined.

We now demonstrate this via a characterization of our workloads. We measure the number of tasks available chip-wide at regular time intervals during execution, including both in-progress tasks and pending tasks in queues, using a simulated 256 core processor. For the datasets we have chosen, the number of available tasks calculated in this manner is a good estimate of the amount of available parallelism. The actual length of each such "task" varies across workloads (see Table 3.3), but in our case it's between a few thousand and twenty thousand cycles.

The left graph in Figure 3.2 shows the number of tasks available at each point in time, for a sample workload from each of three applications (reviewed in Table 3.2). The horizontal and vertical axes are normalized to the maximum value for each workload. All three workloads exhibit large variations in the number of tasks available over time. We can generally describe their behavior as covering three stages of execution:

(1) A ramp-up, where parallel work is discovered; (2) A steady-state, where either the available parallelism has been maximized, or the machine is thread-constrained; And (3) a tail, where parallelism generally decreases over time, though not usually monotonically. `mf` and `fs` show these stages quite clearly, whereas `gt` exhibits a very short ramp-up and steady-state, and is dominated by a long tail.

The right graph in Figure 3.2 shows the parallelism profile of `mf` for three different inputs. It can be seen that the amount of parallelism varies significantly among these three inputs, as well as over time for each input. The relationship between the three stages is also different for each input.

It is a challenge, then, to size a system targeting these kinds of workloads, or to predict available parallelism before execution to provision the "right" number of cores. We expect system designers will have to provide a system with as many cores as is reasonable given a power/area budget, and find reliable methods to utilize as many cores as available parallelism allows, in a dynamic manner. This means distributing work efficiently to grow utilization during ramp-up phases; maintaining high throughput during thread-constrained phases; and releasing resources to right-size the system during tail phases.

3.2.3 Task Stealing at the Hundred-Core Scale

Irregular parallelism strongly motivates a dynamic runtime for managing and delivering work to threads. That is, threads should be able to request tasks, and receive them promptly and with minimal disturbance to other threads. This is usually accomplished with a simple task queue abstraction. In practice it involves significant challenges.

For fine-grain task management on a shared memory multicore system, there is a rich body of work exploring the dominant paradigm of one queue per worker, with stealing [17,25] for load-balance. As an example, the state of the art suggests threads should select targets randomly, steal 1/2 of a queue's contents each time, and focus on low overhead for busy threads (by, for example, favoring non-blocking protocols and letting stealing threads do most of the work).

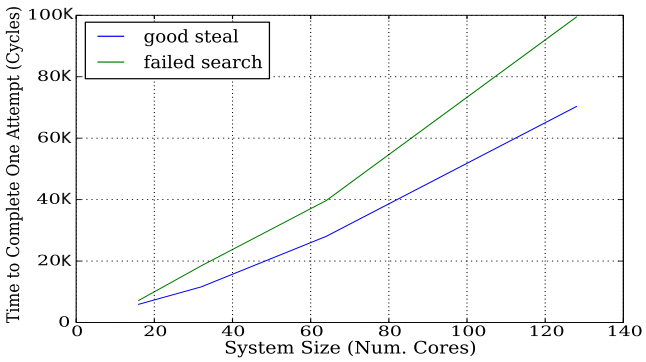


Figure 3.3: Time in cycles required for a full steal attempt on a task-stealing runtime, vs. number of cores. A steal attempt includes searching for and copying from a target work queue. A "failed" search is one that concludes that all queues are empty. This benchmark keeps available parallelism per core constant as the number of cores increases.

There are two main shortcomings when we apply these techniques to irregular parallelism and hundreds of cores. The first is that single-chip task-stealing has mostly assumed that available parallelism in user applications is much larger than the system's number of cores [17]. Even with relatively few cores, this assumption can prove optimistic when facing the kind of irregularity we have just discussed. The second problem is that scaling the number of cores increases stealing costs. This increase is proportional to the number of queues being searched[2], but is also affected by increased contention, and by increasing relative importance of propagation delay.

Existing, state-of-the-art software solutions [35] are not designed for systems of this scale. While overhead may remain relatively low during abundance of work, especially for statically-generated workloads, stealing in particular scales poorly beyond a modest number of cores. Stealing is crucial when task abundance fails to dominate, as well as for dynamically generated workloads in general, depending on the application's inherent load balance characteristics.

Figure 3.3 illustrates the problem. It shows the number of cycles an average steal attempt takes when our `rtb` load-balance benchmark executes on a state of the

[2]If the workload is always thread-constrained, searches may be short even with many workers. But as we have emphasized, this is no longer a safe assumption.

art task-stealing runtime across various system sizes. Successful and unsuccessful searches are separated. The microbenchmark in question scales available parallelism with system size to maintain the odds of successfully stealing as core count increases. This ensures that increasing core counts does not cause increasing scarcity of work.

While unsuccessful searches will by nature take longer on architectures with hundreds of cores, the cost of a successful steal also increases dramatically with system size. This can further threaten parallel scalability in a secondary way: with more cores, it would be beneficial to generate *smaller* tasks, to expose as much parallelism as possible; but the higher overhead of stealing may force us to instead *increase* task size to achieve better efficiency in terms of stealing effort vs. reward.

We could employ hardware to accelerate task stealing. However, state-of-the-art proposals in this direction [57, 86] have called for relatively expensive and specialized hardware, which both sacrifices flexibility, and also makes a difficult case for inclusion in actual systems. Instead we will analyze software-only solutions, which have not been appropriately explored in prior work. Only after exhausting software improvements will we consider hardware options, and we will do so with specific goals (see Chapter 4).

Hundreds or thousands of cores exist in today's large multi-socket and multi-node systems, forcing researchers to face scale-related costs of task stealing [14, 31, 69, 80]; yet these solutions are not suitable to our use. These systems are commonly interconnected in a hierarchical manner, with latencies across sockets and over network links several orders of magnitude larger than on-chip. These systems often do not have globally shared memory for task storage; so task descriptions and application data may need to be explicitly partitioned and moved around the system. Therefore, though task stealing solutions developed in these systems may address numerous cores, the situation demands a framework optimized for moving application data between NUMA tiers and across networks, co-locating tasks and data as needed, and minimizing traffic across links. Because of their physical characteristics, these systems are best suited for coarse-grain tasks.

As we can see, there is a lack of existing solutions to handle the confluence of fine-grain, irregular parallelism with thread-constrained and task-constrained phases,

on the one hand, and hundred-core chips with increasing communication penalties, on the other. We need our solution to handle variability in a way that both maximizes performance of the application, and controls wasteful compute overhead as conditions allow it.

3.3 Experimental Methodology

3.3.1 Modeled System

Table 3.1: Experiment Settings

Core	64-256 cores, dual-issue in-order x86 Core-private 32 KB 4-way L1-I$, 1 cycle Core-private 32 KB 8-way L1-D$, 1 cycle
Tile	1 core per tile Per-tile 512 KB 8-way L2$, 16 cycles, inclusive 2MB shared L3$ & directory slice
Inter-connect	2-D mesh, 1 physical network for coherence messages 2 cycles per hop (1 at router and 1 at link), 64-bit links
Directory	MOSI protocol, 20 cycles
Memory	100 cycles
Compiler	GCC 4.8.2 -O3 w/ libgomp

We evaluate our task stealing scheme on a modified version of the Graphite PIN-based simulator [68]. Table 3.1 shows the configuration of the simulated system. We use simulation to evaluate behavior on future processors with a structure not unlike that shown in Figure 3.4: A 2D arrangement of hundreds of cores communicating over a scalable mesh-based on-chip interconnect. We rely on simulation rather than existing silicon because challenges in efficient dynamic scheduling become more pronounced in designs with hundreds of cores. Other than the number of cores and interconnect design, the processor configuration is similar to that of a modern many-core processor, such as Intel's Knights Corner.

Simulation also allows us to do more detailed profiling, critical for close inspection

Table 3.2: Applications Tested

App	Description	Challenges	Parallelism
fwdsolver (`fs`)	Solve a sparse lower triangular system	Dependency tracking	Dynamic, data dependent, mostly narrow-wide-narrow
maxflow (`mf`)	Find maximum flow through a graph	Mutable graph; variable # and length of tasks	Dynamic, data dependent, can be wide, then narrow
gtfold (`gt`)	Compute shape of RNA	Variable task duration, decreasing parallelism	Dynamic, widest at start linear narrowing
hashbuild (`hb`)	Hash table creation	load balance given locality-optimized tasks	Static, mostly wide
hashjoin (`hj`)	Join tables with hashing	load balance of many short tasks	Static, mostly wide

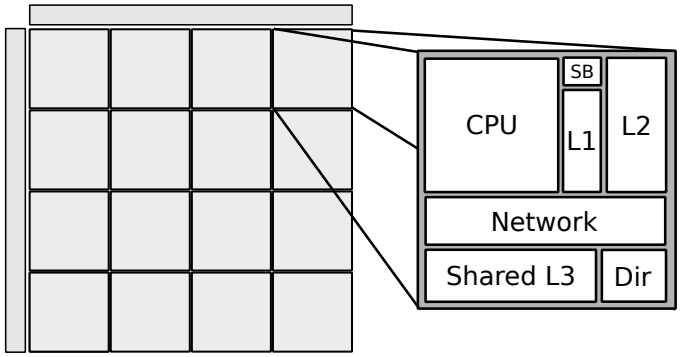


Figure 3.4: The general structure of our simulated many-core architecture is a 2D mesh of identical tiles, each containing one core, private caches, and a shared slice of LLC with corresponding directory.

of the transient behavior of fine-grain dynamic scheduling; this is difficult to accomplish in real hardware without significant interference. To this end, we augment the simulator with instrumentation capabilities including (a) per-thread, fine-grain tracking of cycles spent in various parts of the task management code; and (b) periodic logging of runtime status, including queue sizes and thread states.

3.3.2 Applications

We evaluate using three graph- and sparse-data applications with varying load imbalance and dynamic task generation profiles (`mf`, `fs`, and `gt`), two "embarrassingly

Table 3.3: Datasets Used

App	Name	Descr.	# tasks	Peak Par.	Cycles / Task
`fs`	thin	5-pt 100^2 Lap.	10k	30	1k
	med	27-pt 40^3 Lap.	64k	125	2.8k
	wide	27-pt 48^3 Lap.	110k	156	2.8k
`mf`	thin	32x16 graph	18k	98	1.6k
	med	64x32 graph	130k	400	1.4k
	wide	64x48 graph	203k	2.1k	1.2k
`gt`	thin	133 bases	8.4k	62	12k
	med	250 bases	30k	130	17k
	wide	300 bases	33k	152	16k
`hb`		20k rows	20k	20k	8k
`hj`		40k rows	40k	40k	4k

parallel" applications with statically generated tasks (`hb` and `hj`), and a dynamic load imbalance microbenchmark. Our irregular applications represent real-world problems, and show interesting data-dependent variability. For a detailed discussion of what these applications do, please refer back to Section 2.3.1. As a brief reminder, they are listed in Table 3.2.

Table 3.3 summarizes the inputs chosen for each application. For each dynamic application, we use three datasets that result in very different task graphs. Though primarily interested in the "med" and "wide" inputs, which explore more interesting and real-world likely relationships between available parallelism and system core count, we choose some "thin" datasets as well.

3.4 Design Exploration

3.4.1 Baseline Runtime

Our baseline runtime makes use of current best practices for software task-stealing on multicore chips, as detailed in Chapter 2. It uses circular-buffer-based deques, one per thread, where owner enqueues and dequeues, as well as remote steals, are implemented using a Chase-Lev [24] style non-blocking protocol. In this protocol,

threads synchronize through atomic updates to the deques' head and tail pointers, in a way that reduces owner thread overheads, and minimizes synchronization with thief threads except when necessary. Workers finding their queue empty pick a random queue and attempt to steal half of its contents [35]; if this queue is also empty, the worker picks the next queue and repeats, continuing until work is found or all threads are finished.

Figure 3.3 showed that the time required for a single steal attempt increases along with the size of the system, even when we resize the workload along with the system, to maintain available parallelism per core. One effect contributing to this poor scaling is the increasing average latency between tiles; this slows down coherence transactions, which are frequent during steal searches. Increased numbers of both threads and queues are another factor, causing more steps on average per steal search, and increased contention on each transaction. Further observations using the microbenchmark in question show that even when tasks are generated in a well-balanced manner, ideal scaling is elusive; each thread requires at least one "failed" steal search to conclude execution, plus even a well-balanced workload will result in a certain number of legitimate steals, by the nature of dynamic scheduling.

Even if we could maintain high levels of parallelism in our workloads as we scale up our systems (and, per Section 2.3.2, this is unlikely), it is clear that the performance of task stealing suffers.

Figure 3.5 shows the execution time of our dynamic applications, with the mid-size dataset, on systems with 64, 128, and 256 cores, using the baseline runtime. We break down execution time into components based on profiling from our simulator: ACTIVE represents time spent executing tasks; ENQ, enqueuing; DEQ, dequeueing; STEAL, stealing (both searching and copying); MGMT, runtime management; and WAIT, idling due to a thread's perceived lack of work. Each bar shows the average breakdown across all threads in the system.

While the time spent doing useful work scales well with larger number of cores, the time spent stealing grows dramatically, making 256 core runs slower than 64 cores in all cases. From this graph it seems, for a scalable runtime, we must first address stealing time.

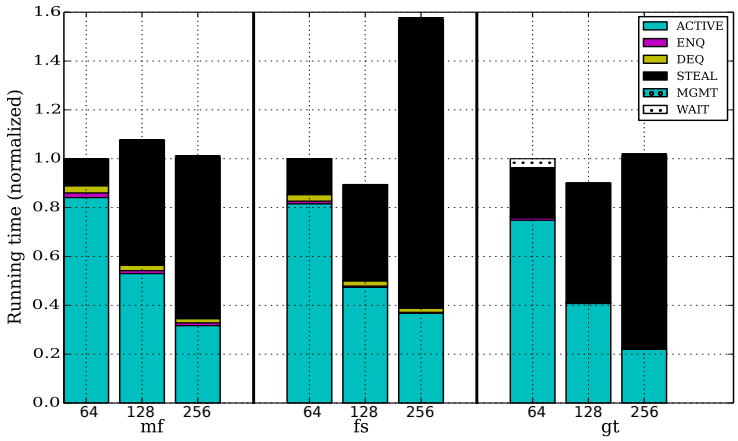


Figure 3.5: Execution time for dynamic applications with the "med" datasets on the baseline runtime. For each workload, we show 64, 128, and 256 core samples.

3.4.2 Policies to Address Stealing Time

At 128 threads, stealing, including both the search for a victim queue and the act of obtaining tasks from it, is the largest portion of runtime overhead in our dynamic applications. It comprises on average 43% of execution time with the medium inputs. On our microbenchmark, stealing accounts for fully 2/3 of the average thread's execution time in the 8x unbalanced case.

To explore cheaper stealing, we implement and compare alternative work-discovery policies:

A. *Persistent stealing*, the baseline policy, where worker threads continuously scan all queues as long as they are in need of work. This does not explicitly interfere with active workers, but potentially creates extra contention for their queues. It is also wasteful, especially if we expect to be task-constrained for any significant time.

B. *Notify one*, where after a first full round of searching, idle threads wait. Active threads must then find and wake an idle thread upon new task creation. The hope is to reduce search time and resource contention, and match the system

size to the workload more efficiently. However, it has the drawback of breaking the "work-first" principle, by explicitly distracting active workers.

C. *Notify all*, where a work-generating thread notifies all waiting threads at once, with zero search, by setting a single global flag. This provides a way for threads to idle and be summoned back to work, while potentially reducing the distraction of worker threads.

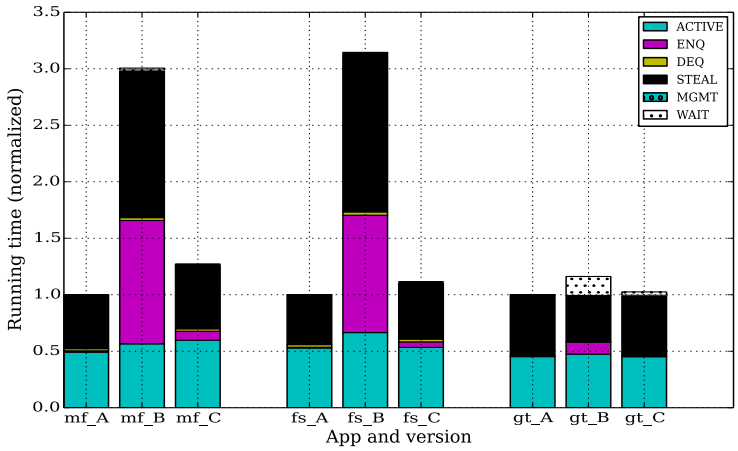


Figure 3.6: Comparison of work discovery policies in the baseline task-stealing runtime. Three variants tested on three dynamic applications at 128 cores.

Both of our alternative work discovery policies incorporate *work sharing*, the idea of having task producers "push" tasks to idle workers. However, they still include work stealing, since workers perform one full steal attempt before giving up. We did not study a pure work sharing approach because that has been shown to consistently underperform versus work stealing [17], except under specific circumstances [6, 34] which do not apply to our work.

Figure 3.6 shows that option A, the pure work stealing approach, is the best for two reasons. First, on a system of this scale, a single round of search through the task queues is very time consuming, and all of the policies include this first pass. Idle threads take a long time to arrive at the point where they must decide whether to try

again, or wait for pushed tasks. Thus, this decision has little effect on total time spent stealing. Second, having work-creating threads locate idle threads and push work to them suffers from scale problems similar to stealing search, reflected in increased ENQ times. Option B then makes the situation worse by placing this overhead on active threads. Option C increases enqueue times less dramatically, by avoiding a search for idle threads, but does not reduce the global time spent stealing.

We analyzed other ideas for locating idle threads while avoiding a search penalty. These relied on having idle threads organize themselves into a prioritized order for notification, such as a stack or queue. These versions performed no better, suffering from the same issues as the above, plus additional contention over shared data structures.

One lesson from this experiment is that any slowdown of active workers is a significant penalty, even if this interruption is relatively brief and search-free. This is especially important because a common case in dynamic task management is for a thread to enqueue a task and almost immediately dequeue it. Forcing it to do anything in the interim can carry a large performance penalty.

Another lesson is that, at 128 cores, even a single search through all queues can be too long for *any* fall-back strategies to make a difference. We need to have threads collaborate on work discovery in the first place — currently, threads duplicate each other's searches, wasting time and power, and increase contention for queues. We look at such a collaborative approach next.

3.4.3 Hierarchical Collaboration for Faster Stealing

We now group threads together, to reduce redundancy in work discovery. At regular intervals, one thread per group (preferentially an idle one), surveys the state of the group's queues, and records it in a cache-line-sized structure. This is effective in reducing the communication cost during work discovery, but only by a constant factor.

Therefore, we create a hierarchy of such information-sharing groups; the result is a tree structure (Figure 3.7). At each level, a "volunteer" thread periodically updates the status. The metadata in these *status nodes* is then used as hints to accelerate

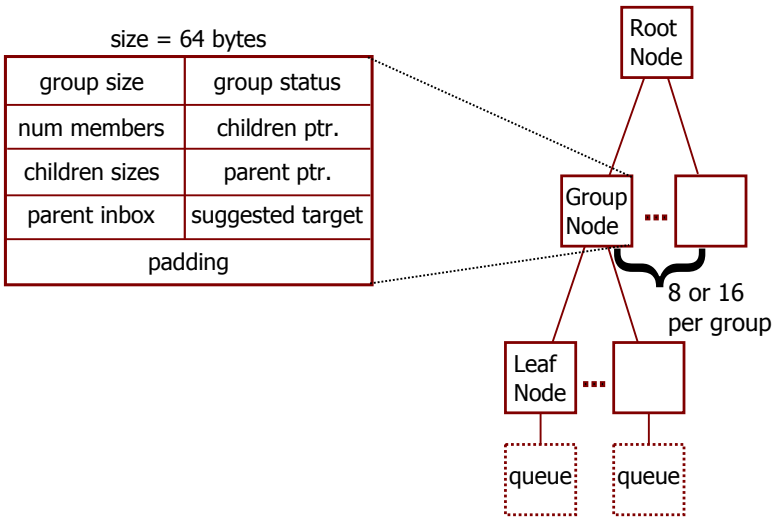


Figure 3.7: Hierarchical structure for gathering task & thread state. Each node fits in one cache line and is periodically refreshed by a member of the group.

work discovery. In particular, a worker who finds his queue empty can query the tree for work to steal. It starts at its own leaf, traverses upwards until it finds pending work, and then back down to the queue that contains it. This search resembles a DFS traversal, with a time complexity of O($\log N$) steps rather than the baseline's O(N).

Communicating about tasks through hierarchical means is an idea that has been explored in hardware-assisted proposals such as Carbon [57] and ADM [86], as well as various works targeted at cluster systems. Our difference is that this is a software-only solution, and that we use it in a single-chip system with a flat cache hierarchy, instead of responding to NUMA/NUCA boundaries or bottlenecks.

This scheme has two main drawbacks. The first is that we incur overhead for periodic maintenance of the status nodes. We mitigate this by (1) preferentially choosing idle threads to perform maintenance, and (2) reducing the frequency of updates when all threads in a group are busy, since in thread-constrained operation, status information is less frequently needed. The second drawback is that we must now manage trade-offs regarding the frequency of updates to status information. The impact of *staleness* on stealing results varies across applications, inputs, and even runs of the same workload.

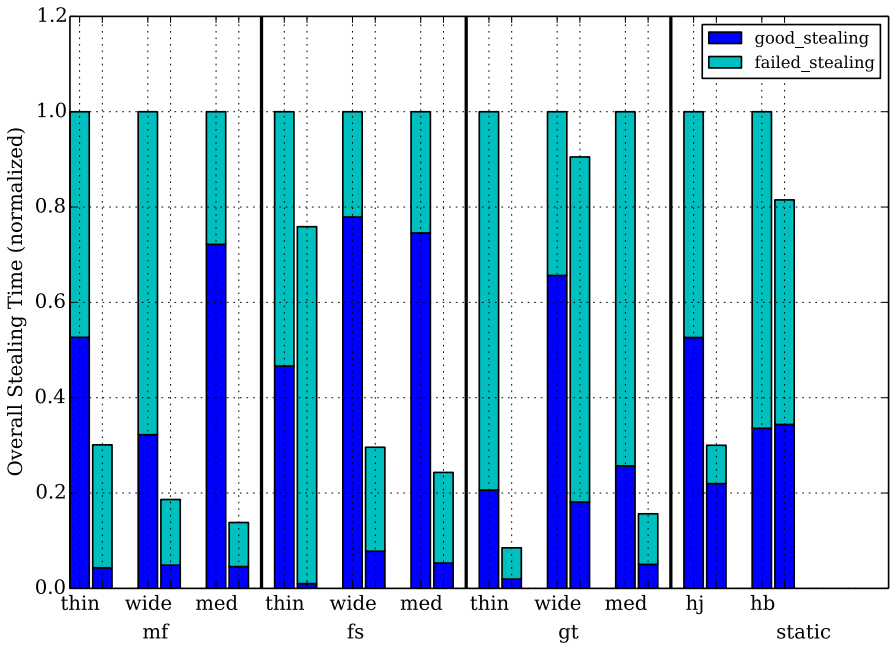


Figure 3.8: Total cycles spent in runtime stealing activities for each workload. The baseline runtime is on the left of each pair, and the pair is normalized to it; the right is the new runtime. Bars also show the portion of that time that resulted in no work being located. Data from 128 cores.

Figure 3.8 shows total time spent stealing, using the new approach, compared to the baseline. Across our dynamic applications, at 128 threads, the new design reduces the time for a successful steal by 13x. During task-constrained operation, the time it takes a thread to conclude there are no more tasks is also reduced by 12x.

With the system tuned to achieve this level of performance, status-node maintenance overhead is 2.4% of the average thread's total time. As the majority of this overhead is incurred by otherwise idle threads, actual performance loss is even smaller.

3.4.4 Dealing With Narrow Parallelism Sections

The degree to which improved stealing results in either application speedup or improved energy efficiency may vary depending on the balance between *thread-constrained*

and *task-constrained* operation, among other factors. During *thread-constrained* execution, stealing in general is both less frequent and less difficult, such that our advantage from its improvement is diminished. However, it's still possible to see improved overall throughput when work is abundant but poorly balanced.

During narrower, *task-constrained* parallelism, our potential advantage is twofold: First, reducing the time required to discover when no work remains during narrow periods can lower effective system utilization, translating into energy. More importantly, faster stealing may directly speed up the application's critical path by reducing the time that new tasks spend waiting in queues to be executed. Tasks waiting in queues while threads are available to execute them are wasted opportunities. To the extent that such a task is on the application's critical path, it is relevant to overall speedup. The runtime's challenge is to minimize this latency between task creation and discovery, without penalizing the throughput of active threads nor giving up possible improvements in energy efficiency.

Our runtime's new tree structure can help accomplish prompt discovery during *task-constrained* phases. But because it gathers the state of worker threads as well as that of tasks, it can also be used to efficiently inform waiting threads of new work. As information about thread and queue state flows up the tree, in the form of updates to groups, and groups-of-groups, information about current best steal targets flows down to child groups or idle threads during group updates. The task of suggesting steal targets to idle threads can even become a simple part of status-node maintenance.

To achieve prompt work discovery, without unduly penalizing execution in other ways, we compared the following policy variations of waiting and notification:

A. *Continuous stealing*: Similar to a classic flat runtime: idle threads repeatedly search the tree for work until they find it or execution ends. This does not allow for right-sizing the system's utilization, but it is the simplest method and might result in new work being found promptly.

B. *Semi-continuous stealing*: Given that *status nodes* are refreshed only at certain intervals, we can reduce cache traffic,at the risk of some performance, if threads that fail to find work wait a short time before searching again.

C. *Notify on node refresh*: Given that a thread from each group already takes the time to update the *status node*, and that it is preferentially an idle thread, can we make work-sharing part of this update, issuing suggested steal targets to waiting threads? This amortizes the overhead to active threads which is one of the drawbacks of work-sharing, and allows idle threads to remain dormant if no new tasks exist. However, depending on the interval between status node updates, this might not notify threads about new work as promptly as direct work-sharing.

D. *Notify on task creation*: Finally, we lean fully on the work-sharing model, allowing active threads to use the tree to quickly find idle threads to suggest new tasks to. This incurs active-thread overhead, but may be the most "prompt" way to get new work to idle threads.

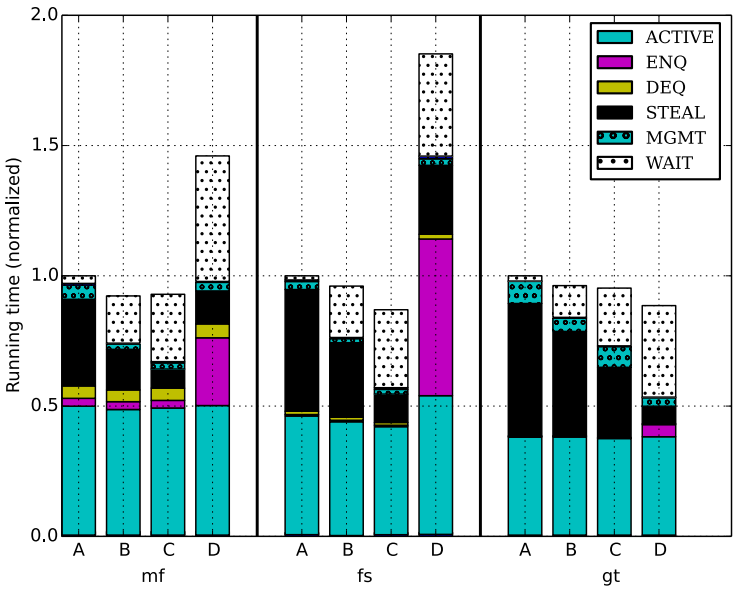


Figure 3.9: Hierarchical runtime, comparing different variations of stealing, waiting, and notification behavior. A: continuous stealing, B: semi-continuous stealing, C: notify on node refresh, and D: notify on task creation.

As shown in Figure 3.9, in the dynamic applications (`mf`, `fs`, and `gt`), *notify*

on node refresh (C) leads to the best overall performance, by actively matching new tasks to idle threads within a bounded interval from the task's creation. It also avoids burdening busy threads with this job. Digging further into the results shows that in A and B, continuous searching fails to find new tasks with consistent promptness, increases the average duration of steal searches, and fails to release unneeded cores during narrow times.

In `gt` specifically, *notify on task creation* (D) has a slight edge over *notify on node refresh* (C). In this case, the improvement to stealing efficiency (threads receiving new task info much faster, for less time wasted) is significant enough to overcome the overhead of direct work-sharing. This difference in behavior is due to the relatively long duration of tasks (about 18K cycles compared to less than 3K for the other applications), which increases the delay penalty of more indirect work-sharing methods, and decreases the harm from short runtime interruptions.

It is an inherent feature of applications and datasets in this space that many of the system's cores may not be utilized throughout the full execution. Though we have achieved much faster steal operations, and explored the *promptness*, from the task's point of view of task discovery in this section, a significant portion of our worker threads will still have long periods during which their help is not required. Having now found strategies to reveal these previously waster idle times, we will touch on ideas for leveraging the freed resources in Section 3.5.3.

3.4.5 Leveraging Hierarchical Centralized State

The Chase-Lev protocol used in our runtime relies on synchronization through atomic updates to deque head and tail pointers. As such, stealing operations present more design choices than lock-based synchronization schemes: Once a thief thread decides how many tasks to take from a victim queue, it must also decide on the number and granularity of atomic queue operations to achieve it. It is especially relevant during thread-constrained situations, where we must minimize overall runtime overhead: if threads grab more tasks at once, stealing is more efficient, and locality among tasks is better-preserved; but the risk of a specific operation failing due to competition is

also increased. Fine-granularity steals, conversely, are faster per-operation and have better success probability, which makes them desirable in task-constrained situations; but they are inefficient for stealing large numbers of tasks.

As we adjust stealing granularity, we also increase selectivity towards victim queues: threads will refuse to steal from queues that appear to have less than twice the per-transaction number of tasks. When tasks are abundant, this selectivity can increase overall efficiency.

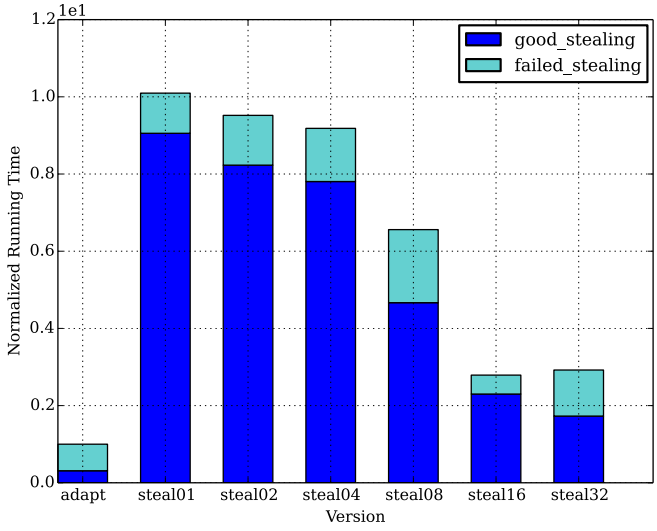


Figure 3.10: Dependence of stealing portion of running time on granularity of steal transactions, for a thread-constrained workload, comparing fixed settings to an adaptive policy that adjusts granularity in response to estimated available parallelism. Data from `hj` at 128 cores, normalized to best case.

Experimental data in Figure 3.10, obtained using a thread-constrained workload on 128 cores, shows that stealing transaction granularity is a tuning factor of consequence. It also illustrates how we can leverage the tree of *status nodes* to make global decisions based on the accumulated state of queues chip-wide. Not only does this allow us to arrive at a good setting with no hand-tuning; it also allows policies to adapt as conditions change over time, possibly yielding better results than any individual fixed policy.

Another tunable parameter in our runtime is the interval between *status node*

updates. Too short an interval increases runtime overhead, possibly slowing down execution (especially in thread-constrained situations). Too long an interval reduces overhead, but may make the gathered information less accurate and hence less valuable, reducing our speedup in task-constrained situations.

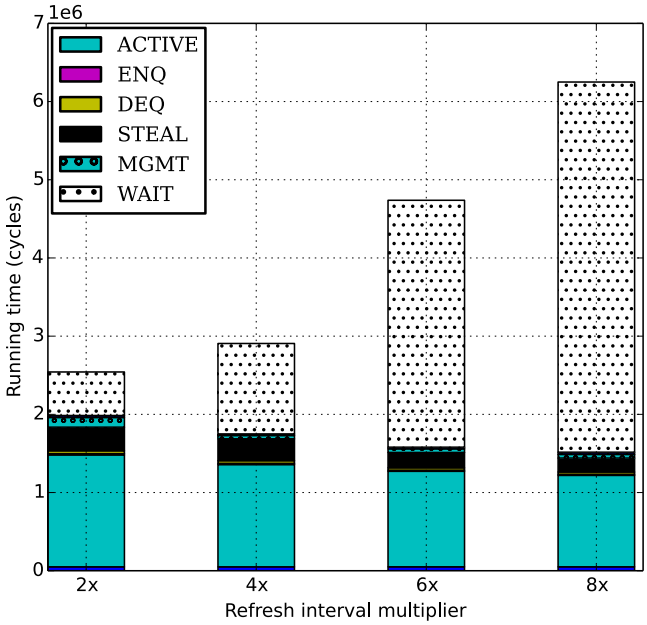


Figure 3.11: Dependence of execution time on status node refresh interval, showing trade-off between actual utilized cycles (non-idle), and total running time. Time spent stealing also decreases with larger interval. Data from 128 core execution of a task-constrained workload.

Figure 3.11 shows one example of the consequences of varying this interval. For this task constrained scenario, overall running time is strongly dependent on the refresh interval, reflecting our concerns about prompt task discovery. However, though execution takes longer, overall system utilization (e.g., non-idle core cycles) is reduced with longer intervals. Profiting from this type of efficiency would require a careful strategy regarding energy management or multi-processing.

We mentioned earlier that there is a dynamic aspect to our system's implementation of meta-data update intervals. When there is no idle thread in a group to carry out management duties, update intervals are increased. This lowers overhead

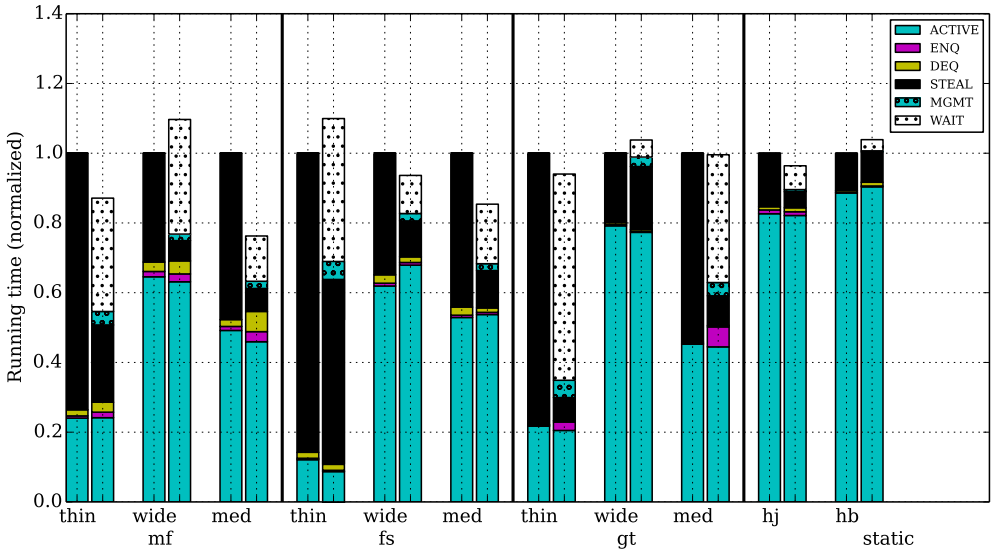


Figure 3.12: Overall running time comparison on 128 cores for all applications and datasets. In each pair, the left bar is the baseline. Bars are divided to show the portion of execution spent in each activity.

in *thread-constrained* conditions, when all threads are busy and stealing is less frequent. The availability of global-scope information in the "root" node holds promise for further smart policies. For example, a system could tune said update intervals in response to some "degree of novelty" metric based on recent update results. Exploring such possibilities is left for future work.

3.5 Overall Results

We now compare the overall performance and scalability of our new runtime to the baseline. We also briefly discuss idle core times during narrow parallelism periods.

3.5.1 Performance

Figure 3.12 compares overall running time on 128 cores. Target applications using our new design outperform those using the baseline by as much as 30%, with an

average of 15% over our target datasets, and are still ahead by 8% when including control datasets where available parallelism is less of a factor. This speedup is achieved without changes to the application's code beyond the runtime layer, and in spite of the runtime only representing a fraction of the original execution time even with the baseline design. This speedup results from an order-of-magnitude reduction in task search latencies; improved system throughput during load-imbalanced conditions; more efficient granularity in stealing interactions; and lower queue contention.

A striking result, visible in Figure 3.12 as a relative increase in "wait" time, is highlighted more clearly in Figure 3.13. Total system utilization, measured as the portion of "busy" cycles during execution, has been reduced to only 71% of the baseline, with some workloads showing much larger advantage. Reduced core utilization reflects improved runtime efficiency enabling the release of unneeded cores during task-constrained conditions, while other cores remain busy. Put another way, reductions in runtime overhead do not necessarily improve overall performance: sometimes tasks we help locate are not blocking the critical path; sometimes we help threads quickly conclude that no work is currently available.

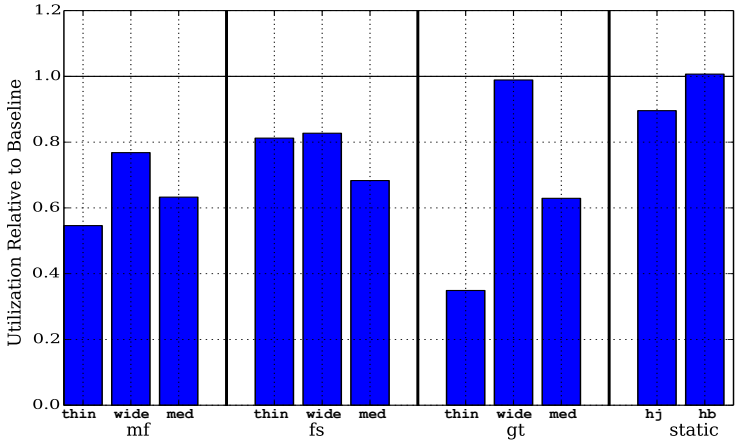


Figure 3.13: Overall portion of core cycles utilized by the hierarchical runtime on 128 cores, normalized to the baseline's utilization (in the baseline, all cores are 100% busy).

As we will later discuss, this right-sizing of active threads is a significant advantage, one the baseline runtime is incapable of. By releasing execution resources, improved utilization can be leveraged for power savings, or it can be re-purposed for opportunistic execution of other programs. Section 3.5.3 discusses these opportunities. It is important to note that improvements to speedup and utilization do not necessarily correlate for the same applications, nor are they necessarily accrued during the same stages of execution.

With our static applications, `hj` and `hb`, where all tasks are known at the start, the new runtime's performance is very similar to the baseline, whose inefficiency in this case is very slim. These applications prove that our new runtime manages its added complexity to maintain low overhead on active threads, and that it can even squeeze some advantage due to more efficient stealing during periods of abundance.

3.5.2 Scaling

Table 3.4: Microbenchmark Weak Scaling (16 to 256 Cores) Slowdown Comparison

Benchmark variant	Baseline ratio	Improved ratio
Unbalanced	4.9	1.6
Task-constrained	3.5	2.3
Balanced	2.2	2.2

Scaling the performance of irregular parallelism applications into the hundreds of cores faces two key challenges. First, the cost of synchronization and communication increases due to the physically larger system. Second, the changing relationship between available parallelism and core count makes locating work harder.

Our software runtime mainly addresses the second problem. By turning a linear relationship between search steps and system size into a logarithmic one, it improves the scaling of runtime overhead. However, it also reduces the system's vulnerability to the first problem, physical delays, by reducing the number of queue interactions and their contention delay. Table 3.4 shows the relative increase in execution time of a load-balance microbenchmark when going from 16 to 256 cores. Ideal scaling for

this workload would be flat, as tasks per core are kept constant; clearly the baseline runtime is far from ideal. Our new hierarchical runtime provides improved scalability, especially during higher-imbalance and task-constrained situations.

Table 3.5: Application Speedup for 128 vs 64 Cores

Application	Baseline ratio	Improved ratio
`mf`	0.92	1.33
`fs`	1.12	1.28
`gt`	1.10	1.16

Table 3.5 shows the speedup achieved by moving from 64 to 128 cores, for each of the main irregular applications, on both runtimes. In each case our new runtime scales better than the baseline.

3.5.3 Idle Time and Energy Opportunities

Workloads with unpredictably varying parallelism will sometimes result in significantly less available work than cores, especially on large systems. For efficiency reasons, the runtime may wish to *right-size* the system, matching the number of active cores to available parallelism, and releasing or powering down the rest. Dynamically surrendering unnecessary resources is not a new idea. Operating System schedulers do this, for example; though at a much coarser granularity.

Our runtime improves the ability of the system to be right-sized. The baseline's performance relies on idle threads continuously searching for work, and improving this through "pushing" techniques proved unsuccessful. In contrast, the hierarchical runtime can put workers into an idle state when they have nothing to do.

Figure 3.13 showed that the hierarchical runtime keeps workers busy, performing work or searching for it, only 71% as many cycles as the baseline. Some of this reduction is from faster overall execution, but much of it is from allowing some cores to idle while others continue. In practical terms, these idle times exist in between when a thread fails to find work, and when it is told about the possibility of new work. Figure 3.14 shows a sample distribution of the duration of such idle periods

for an average thread during a `mf` execution. In this case most idle periods last more than 4K cycles, and the overall average is over 11K cycles. Even on current-day architectures, many of these idle periods would provide sufficient time for short-term sleep states. Ongoing research into fine-grain power management also promises to leverage even shorter idle periods [65, 81, 88] in the future.

Many idle periods occurring on the same thread are separated merely by fast, unsuccessful steal attempts, as idle threads often compete with each other or with queue owners for new tasks. There is therefore the possibility, through further experiments, of lengthening the average idle period to allow for deeper sleep states or more efficient time sharing.

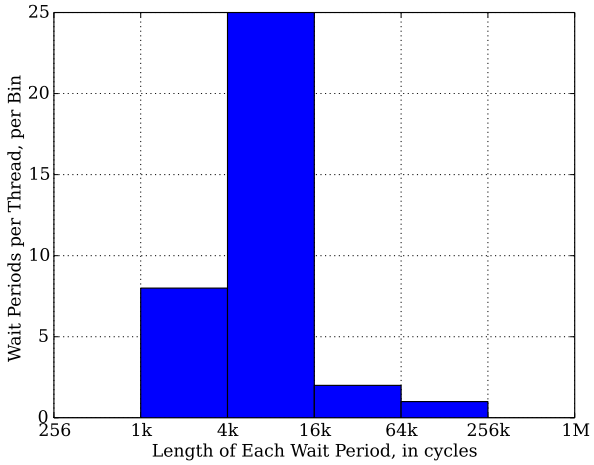


Figure 3.14: Distribution of the duration of core idle periods for threads in `mf`, using the "thin" input, on 128 cores.

3.6 Conclusion

We have explored the interaction of future hundred-core processors with the increasingly important domain of irregular applications, such as those that operate on graphs and sparse datasets. We have discussed how, though these applications do often make use of all available cores during at least part of their execution, there can be large

fractions of time where work is insufficient. For full system utilization a dynamic runtime used in these scenarios must, therefore, operate efficiently during both abundance and scarcity of tasks. This is challenging since the principles of a good runtime in each scenario are at odds: When work is abundant, we desire low overhead to active threads, and infrequent, high efficiency stealing. When work is scarce, we want a smarter runtime that quickly locates new work with low stealing latency, and one that allows for right-sizing of the thread pool to match narrow workload periods.

The hierarchical meta-data structure of our runtime allows for sophisticated stealing policies and creates a global view of the system, without resorting to multi-level queues, increasing overheads on active threads, or relying on custom hardware. The runtime is able to adapt as conditions change: it keeps overheads low when work is abundant, matching the performance of a state-of-the-art baseline on embarrassingly parallel applications. It also improves task-stealing latency as work becomes scarce, resulting in both execution speedup and reduced system utilization for irregular applications. The runtime's view of global state presents further opportunities for dynamic adaptation.

Despite our improvements, runtime overhead remains a significant portion of execution time for some of our applications. The next question is, what can we do to advance further?

Profiling shows that a majority of the overhead in both task stealing and runtime maintenance comes from on-chip cache-coherence delays on runtime meta-data. In this chapter we have mainly sought to reduce the *number* of such cache requests; the *latency* of each such request is our next target, explored in Chapter 4. Though we cannot fundamentally reduce on-chip electrical distances, observation of cache level communication patterns resulting from shared access to memory reveals opportunities to improve latency through architectural proposals. There are many constraints on what we can do; but if we succeed, we will be helping not just our task-stealing case, but rather a much wider scope of synchronization and communication patterns which result in similar behavior at the cache-coherence level.

Chapter 4

Accelerated Data Sharing With Atomic Ops for Many-Core Chips

4.1 Introduction

The relatively small core counts on currently popular chips, combined with the abundance of regular parallelism, have prevented communication latencies from being a dominant performance problem on multi-core chips. As chips scale towards hundreds of cores [21], however, the relative time and energy cost of communication is certain to become a significant factor [19].

One common example of such regular parallelism, with predictable memory access patterns, high data locality, and the ability to be partitioned in coarse grain, is dense matrix multiplication [40]. However, as Chapter 3 has illustrated, several emerging application domains exhibit different, irregular parallelism, making them more vulnerable to the aforementioned communication costs. Graph processing is one such domain [36,46,47,74]; sparse data is another [32,61,84]. In both cases, the amount of parallelism depends on the data being processed, and performance depends strongly on communication [77].

Ongoing trends in chip architecture promise to make this situation more difficult. Silicon process scaling affects transistors and delay lines differently, altering the balance between compute and communication latencies. Larger numbers of cores

per chip increase network complexity and resource contention. Increased availability of accelerators and SIMD units may also exhaust regular parallelism, increasing the importance of irregular workloads for general-purpose CPU performance [13].

On most multi-core chips, when threads need to communicate with one another, they do so through shared access to memory, made coherent by an on-chip cache system. In most popular architectures, this model prioritizes safety and convenience, using directory- or snooping-based protocols of varying complexity. However, these protocols result in significant inefficiency in the presence of frequent sharing due to latency and increased traffic.

In regular parallelism, prefetching or multi-threading can reduce these penalties. However, irregular applications have a harder time making use of such techniques. Unpredictable access patterns foil prediction hardware, and variations in available parallelism make it difficult to hide access latencies by generating enough threads to occupy hundreds of cores. Hence, irregular parallel applications remain sensitive to communication latency.

We have a specific example of the communication needs of irregular-workloads in the task-stealing runtime developed in Chapter 3. The runtime reduces inefficient activity and makes better use of gathered knowledge about the state of task queues and threads. However, as we will quantify early in this chapter, its own performance and efficiency depend on communication latency, as it manages tasks and maintains meta-data through shared memory.

Using this runtime as a driving example, along with a few benchmarks, this chapter presents communication-oriented enhancements to the cache hierarchy of a many-core chip. The goal is to enable memory-based data sharing and synchronization which result in direct and efficient traffic patterns at the cache level. A second goal is to maintain flexibility by avoiding reliance on hardware implementations of high-level features or full synchronization primitives.

We present instruction set (ISA) hints which provide an interface to the aforementioned hardware enhancements. Using published work [78] as a starting point, we are able to address a significant set of communication and synchronization patterns with only small changes to the code. At a first level, ISA hints trigger direct access to data

in both private and shared remote caches. We then present two different approaches to speedy and efficient concurrent atomic operations, as this is important, for example, in synchronization primitives. Our proposals dramatically accelerate specific sharing patterns based on interleaved atomic updates by enabling a small subset of atomic operations to be performed at the shared LLC, and they do so while remaining fully integrated with existing cache coherence.

We show that relatively small changes to the chip's cache controllers are sufficient to enable these features, by implementing hardware support on a simulated 128-core system. We then demonstrate how these new techniques can be used to accelerate several portions of the task-stealing runtime we have developed. We measure the runtime's performance with multiple irregular applications, and with a set of synchronization benchmarks.

Multi-fold speedups to our benchmarks showcase the potential benefits of decreased access latency, and increased efficiency. In our irregular applications, simply using the accelerated version of the runtime improves overall execution speed by as much as 18% relative to the end of the prior Chapter. Cache system energy use is also significantly reduced.

Contrary to prior work, our features improve performance and efficiency without demanding significant code rewrites, forcing new system architectures, or abandoning familiar memory models.

4.2 Background

Our sample many-core architecture contains 128 cores in a tiled arrangement, with a general structure as earlier illustrated in Figure 3.4. Each tile contains a general-purpose CPU core, private caches, a slice of a shared global LLC with matching directory, and an interface to the on-chip network, which in our case is a 2D mesh. This architecture resembles available chips such as those in Intel's Many Integrated Core series [90], but is sufficiently generic for solutions to be portable to different network topologies and deeper cache hierarchies.

Sparse and graph-based dataset processing are two domains where we often find

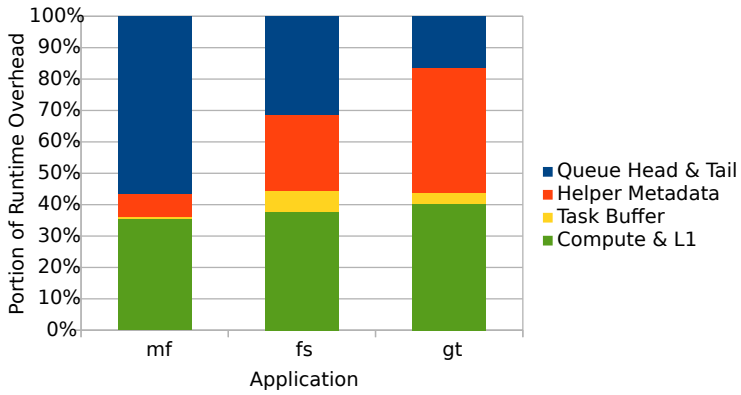


Figure 4.1: Modeling hypothetical zero-latency coherence among local caches reveals the portion of the task-stealing runtime's overhead attributable to coherence activity on each data structure. The bottom portion is that not related to coherence delays. Data from our three main irregular applications on a 128 core system.

irregular parallelism. As discussed in depth in Chapter 3, irregular parallelism cannot be efficiently handled with common static scheduling methods. Significant effort is invested into the search for more regular algorithms, with some success [12, 48, 67]. However, this is a time-consuming and uncertain process; it cannot be relied on to eliminate communication issues in general. Instead, irregular parallelism frequently requires a dynamic runtime to handle evolving and unpredictable parallel work in a load-balanced manner.[1]

Examining dynamic runtimes of this type, such as the one developed in Chapter 3, we encounter shared memory being used for communication in several situations, including synchronization and shared access to data. The resulting coherence traffic in the cache system of a chip like ours is expensive enough to significantly affect performance and efficiency. A preliminary study simulating zero tile-to-tile coherence latency, summarized in Figure 4.1, reveals that coherence delays on a few specific data structures account for more than 50% of the optimized runtime's remaining overhead.

This overhead is not due to software inefficiency; in Chapter 3 we already reduced the number of memory requests issued by the runtime, and improved the benefit

[1] At this scale, even some cases of regular parallelism may need such a runtime to achieve consistent scalability [57].

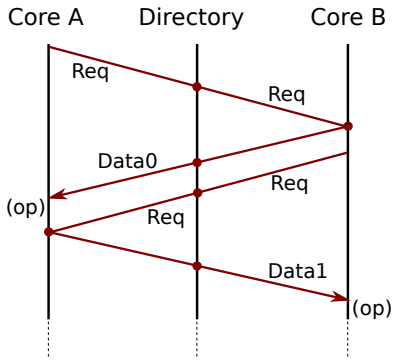


Figure 4.2: Cache coherence traffic when two cores interleave writes to (or atomic operations on) shared data, under baseline on-die coherence. Each request endures four network delays, as the line is always found in someone else's private cache.

derived from each request. What we must now do is achieve faster communication among threads, by improving the manner in which the hardware handles relevant cache requests.

The implementation of cache systems relies on assumptions about data locality which have proven very useful for single-thread applications in general, as well as for many workloads where multiple threads share a dataset. However, data used primarily to synchronize or communicate among threads often defies the basic assumption. As a result, relevant cache blocks often suffer *ping-ponging*, where a private cache will insert a block, only to have it invalidated before reuse. Threads then suffer longer-than-necessary latency penalties, as cache lines are often found to have been invalidated, and must be located, fetched and re-inserted (See Figure 4.2).

This is one inefficiency we must eliminate, as it has a significant effect on latency. Secondary effects are increased network congestion and energy consumption.

One of our stated goals is to avoid sacrificing a user-friendly coherence and consistency model. Much relevant work has explored this boundary between performance, ease of use, and hardware complexity in relation to cache coherence.

Many proposals optimize cache insertion and eviction through locality analysis. However, they often focus on single-core locality, or suffer significant profiling overheads [58, 72] in storage, compute, and reaction time.

GPUs have generally offered non-existent or manual coherence. In this domain,

lack of coherence was an early performance decision, since coherence penalties would be large in GPUs due to the volumes of data involved. Offering coherence only when requested could possibly facilitate some of the performance opportunities we seek. However, it would increase programming complexity and create correctness risks, which we wish to avoid. Recent work in the GPU space seeks a middle ground [50,75], improving programmer friendliness as part of the move towards practical heterogeneous architectures. However, the usage model is still quite different from the norm for general-purpose systems.

Some proposals implement coherence as a layer on distributed systems. They seek the improved usability of coherence with low hardware complexity and minimal network traffic [44, 83]. The result of these compromises can include coarse tracking granularity, poor flexibility, and relaxation of consistency guarantees, making programming more difficult.

Relaxing consistency guarantees, in fact, is another trend in architecture [83]. So-called *SC-for-DRF* (Sequential Consistency for data-race-free parallelism) models, for example, renounce ordering guarantees on all memory operations except for atomic requests and fences. This relaxation broadens possibilities for performance enhancement. The supporting argument is that, in "properly" written code, only a small set of synchronization-related data should rely on ordering guarantees. However, as our intention is to maintain compatibility with existing code, avoiding significant rewrites, as well as maintaining accessibility, this trade-off is not one we are willing to make.

Beyond the level of tweaking coherence protocols or consistency models, research has more directly explored solutions to the latency of on-chip synchronization and communication. Many proposals invest significant hardware in implementing high-level constructs such as hardware locks [93] or task queueing systems [49, 57]. The resulting performance is good, but it comes at the expense of narrow scope, sacrificing flexibility and limiting the return on hardware investment. A number of other proposals require adopting a very different compute model. Examples of this include stream computing [28,39] and message-passing [96]. Some of these are relatively flexible in their scope of application; however the effort to understand the model and write appropriate new code increases the barrier to entry and reduces the probability

of adoption. Many solutions reliant on hardware acceleration can also suffer from practical challenges at implementation time such as OS-level resource sharing and safe context switching. These often require fall-back methods [23], which make the final solution relatively complex.

In compute clusters with distributed memory models, network latencies have already brought analogous communication cost issues to the fore. Some measures that have been researched under these conditions include direct modification of remote nodes' memory without the owner's participation [51], and the crafting of distributed synchronization primitives [52, 71] based on that capability. Many of these solutions have relied on the existence of a secondary processor (usually the network adapter) at each node, with DMA access to the local memory.

Further enhancing such network chips or memory controllers with more complex data processing hardware has also netted performance gains on HPC systems. Two examples are combining networks and processing in memory [8, 37]. Application of these cluster results to fine-grain, on-chip synchronization is made difficult, however, by orders-of-magnitude differences in computation vs. communication ratios. Furthermore, solutions driven by parallel throughput needs are not appropriate for low-latency synchronization, even if differences in scale could be overlooked.

Neither current hardware nor ongoing research, then, addresses our needs in ways that ensure (a) general applicability to a broad set of problems, instead of specific hardware for specific primitives; (b) high practicality from both hardware and software implementation perspectives; and (c) compatibility with existing systems and codes, avoiding complete rewrites or redesigns, and maintaining a low barrier to entry.

4.3 Architectural Features for Accelerated Fine-Grain Data Sharing

We will start this section with a review of our design goals. Then we will present the proposed ISA, and afterwards describe our architectural proposals.

We start with direct reads and writes to data residing in non-local caches. First,

we draw from the *ULPS* work of Park et al. [78]. By enabling producer threads to write directly into the private caches of known consumers (known either to the producer or to the directory), coherence traffic and access latencies can be reduced through avoidance of unnecessary insertions and evictions. As we will later show, this is done without giving up full cache coherence, or introducing a separate programming model.

To target all relevant communication patterns, it is necessary to add complementary hardware support in multiple directions. First, to address producer-consumer patterns where consumers are unknown or varying, we provide direct reads and writes to *shared-cache* locations. On our sample architecture this means interacting directly with the LLC+directory slices. On other chips, shared intermediate caches could be targeted the same way. One example of its usefulness is in task-stealing runtimes: A thread may update certain meta-data with knowledge that it will be used by someone else, without knowing who that someone else will be.

Another need we must fill, if we aim to accelerate synchronization, is to improve the efficiency of interleaved atomic operations on shared data. Atomic operations are essential for many synchronization patterns and primitives; however, when used in this way, their locality is at odds with assumptions that make local caching worthwhile. Improving on this situation requires distinct mechanisms from those used for remote cache access. We present two methods, with different benefits and hardware requirements.

4.3.1 Design Goals

Our overall goal is to improve the performance of applications that have significant dependence on communication through shared memory.

To improve on prior work, especially hardware proposals specific to dynamic task-stealing [49,57,86], or to locking [23,52,93], we want our investments in extra hardware to be suitable for a relatively wide range of uses. This is both to minimize the loss of flexibility that accompanies more targeted solutions, and to broaden the solution's

appeal for inclusion in general-purpose architectures. This is why we target fundamental sharing patterns in memory: These are as relevant to basic primitives such as barriers and locks, as to our Chase-Lev [24] queue-based dynamic runtime.

Given the popularity of the cache coherent shared-memory model (in both software and hardware communities), we limit ourselves to techniques that avoid significant changes to, or abandonment of that model. We shall also minimize required software changes. Much has been written about how difficult parallel programming is in practice; we wish neither to propose a new programming model, (e.g., stream computing [39]) nor to significantly alter the existing one. We consider compatibility with existing mainstream programming models an important goal for providing added value to a given architecture's software/hardware ecosystem.

Finally, the solutions we propose should be practical to implement in real-world settings. This includes being robust during context switching & multi-processing; not creating interruption or fairness issues at the core level; and minimizing the architectural state needing to be saved and restored.

4.3.2 ISA & Usage Model

Achieving our goals of compatibility, generality, and practicality partially depends on how the new hardware capabilities are presented in the ISA, and how they are leveraged. Here we describe these aspects.

All of the mechanisms presented here are implemented as extensions to the ISA. Not as new instructions, but as optional variants of existing instructions, where a hint can express placement preference. A programmer or compiler substitutes these hint-bearing instructions in lieu of their classic variants anywhere that a suitable communication pattern is anticipated. The hints indicate when loads and stores should target a private cache, or all current sharers, or access a given line from its tile of shared LLC. On atomic operations, the hint simply indicates whether performing the operation at the shared cache tile, rather than in the conventional way, is preferred. Table 4.1 summarizes this usage model.

No specific hardware implementation is mandated by the ISA. As long as a system

Table 4.1: ISA for Location Hints on Memory Operations

Instructions	Hint Value	Meaning	Default
Loads	`tile#`	Read directly from tile #'s private cache	
	`shared`	Read directly from shared LLC	
	`local`	Read from local cache in classic manner	Yes
Stores	`tile#`	Write directly to tile #'s private cache	
	`known`	Write to known sharers	
	`shared`	Write directly to shared LLC	
	`local`	Write to local cache in classic manner	Yes
Atomic ops	`shared`	Execute at or push to shared LLC	
	`local`	Execute locally in classic manner	Yes

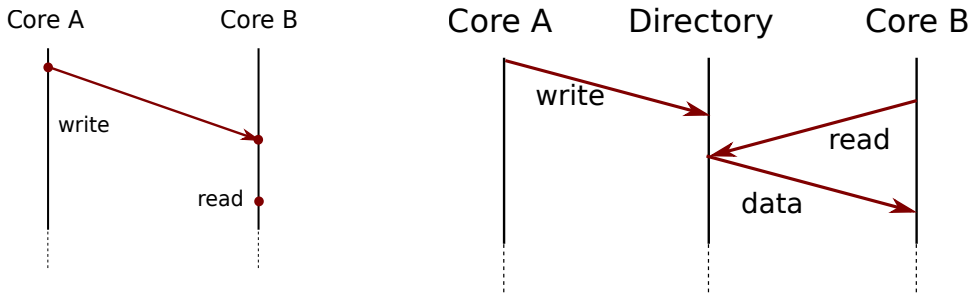
maintains the memory model, thus preserving portability, even platforms that do not choose to leverage these hints should be able to decode and execute the corresponding memory accesses correctly. A system implementing support, on the other hand, would spot the hints at the decode stage, and use them to change the way the instruction is issued.

4.3.3 Direct Access to Private and Shared Remote Caches

Previous work by Park et al. [78] has presented performance benefits when memory access instructions are allowed to target a specific tile's private cache. This enables advancing data to a future consumer for minimal latency (see the left side of Figure 4.3), and avoids pollution of the producer's cache. An indirect request can also leverage the directory's current list of sharers to push updates to multiple private caches.

With this first step, producer-consumer relationships with known consumers (known to either the requesting core, or to the directory through the current sharers list) can be made faster and more efficient. This can be useful in implementing, for example, a fast tree barrier. However, in many situations (including our task-stealing runtime) it is hard or impossible to know the future consumer of a piece of data.

We therefore propose a complementary approach: to enable memory reads and writes to target shared on-die caches. Despite promising a smaller gain vs writing to a



(a) In remote private cache access, a specific private cache becomes the data's designated home. Most useful when the consumer is known, and has critical latency requirements.

(b) In remote shared cache access, the data "lives" at its home tile of shared LLC/directory. When consumers' identities are not known, or all sharers have equal latency requirements, this still provides significant improvement over the default case.

Figure 4.3: Cache coherence traffic patterns when two cores interleave writes to shared data, optimized by use of remote private and remote shared cache access.

private cache, it enables reduced contention, access latency, and network traffic across a wider range of situations. It does this by still avoiding insertion and invalidation of low-reuse cache lines into private caches (see the right side of Figure 4.3). The main piece of information this leverages is each user thread's knowledge that the data in question has chip-scale reuse but little local cache reuse.

In summary, we give up some of the latency advantage of direct placement into remote private caches, but we in turn avoid needing to know *who* the next user of the data will be.

In our simulated architecture, the last-level cache (LLC) is distributed across the chip, with one slice at each tile; cache lines are associated with a single slice, according to an address hashing function. Each LLC slice is co-located with a slice of directory, implementing MSI-style coherence among the private caches. When we say that we can target shared cache, we refer in our specific case to these LLC / directory slices. The ideas we explore are not specific to this particular cache hierarchy, however — they can be applied to most hierarchies that include a mixture of private and shared caches.

The above-described support for direct writes and reads to the shared LLC requires

only minor changes to the coherence protocol, and to the cache controller hardware at the LLC and private caches.

- A private cache controller must be able to read and write requests directly to the shared cache, using, e.g., existing stream buffer hardware to bypass the private caches.
- When issuing write requests to the shared cache, writeable (exclusive) state for the cache line in question is neither requested nor expected; rather, write requests are sent along with a mask indicating the bytes to be written.
- Upon receipt of the request, the cache line's *home* LLC / directory controller issues any necessary write-back or invalidate requests to current owners (though, when properly used, this should be rare), proceeding with the read or write directly at the LLC once the cache line is in an acceptable state (i.e., uncached for writes, and an up-to-date copy for reads).

If all concurrent sharers of a cache line access it in this fashion, the line remains continuously at its "home" LLC slice, making interleaved reads and writes by different cores faster by completely avoiding invalidations and write-backs. If a "normal" cached access is issued, no correctness problem results; only potential inefficiency.

This scheme avoids weakening existing sequential consistency guarantees by naturally serializing requests at each cache line's own directory. It is also still possible to use the stream buffer for deferred / buffered writes, rather than sending requests immediately, with the same weakening of consistency this usually entails.

4.3.4 Adding Support for Atomic Operations

Neither of the two schemes just described (remote private and remote shared cache access) supports atomic operations (i.e., read-modify-write) in their current form. This is because atomicity requires that, while the relevant cache line remains protected from interference by other cores, a functional unit must locally perform the arithmetic component of the atomic operation. This is not possible (or at least practical) when the requesting core and the cache line in question are in distant locations.

Synchronization primitives in current code rely on atomics (for example, locks rely on *CAS* or similar). Our aforementioned task-stealing runtime relies on atomics also, for ensuring the safety of transactions on individual task queues. Applications where multiple threads update a shared dataset may also rely on atomic operations rather than mutex locks, as a faster method when datatypes and operations allow; a classic example of this is computation of a shared histogram using atomic adds. We will examine this later in benchmark form.

The simplest solution for this case assumes that we wish to continue to perform atomic operations at the requesting core, but would like the data to appear to reside in the shared cache as much as possible. We propose to simply fetch the data to the requesting core (again using stream buffer facilities to avoid local cache churn); execute the atomic operation locally; and then force an immediate write-back to the shared cache.

Though this does not completely avoid network delays due to fetching of cache lines, it does in most cases reduce the next user's access latency by pre-positioning the cache line at the shared LLC (see Figure 4.4).

One interesting quality of this method is its low hardware requirements. It would require (1) that the stream buffer be able to hold a cache line in writeable (exclusive) state just long enough to perform an atomic operation, and (2) that the local cache controller prevent interference with this cache line during this operation.

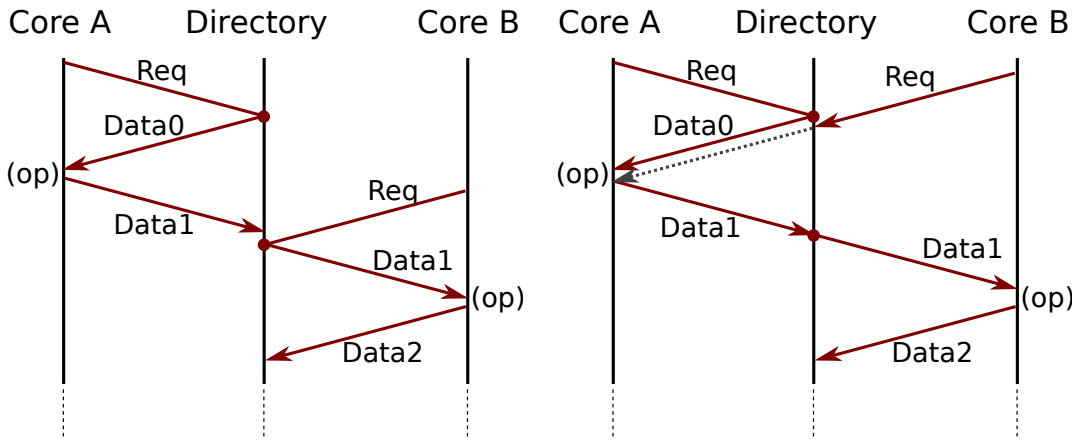
Atomic operations on shared-cache hosted data are then performed in this manner:

1. The private cache controller at the core executing the atomic requests exclusive status.

2. Upon receipt, said controller places the data in the stream buffer, and places the line into a "locked" state.

3. The core performs the desired atomic operation on the locally buffered cache line in the classic manner.

4. The modified cache line is immediately flushed out and written back to its home LLC, with the controller "unlocking" the line.



(a) When atomic updates from different cores are far enough apart, requesting cores suffer only two network latencies per update; same as with non-atomic remote access.

(b) When atomic updates from different cores are very close in time, some requesting cores may still suffer essentially four network latencies on updates, which is no improvement over the baseline.

Figure 4.4: Remote shared cache access with atomics at the requesting core. In most cases (though not always), coherence traffic and core request latency are reduced, as data is always pushed back to its home LLC tile after each update, to be ready for the next request.

This mechanism still avoids inserting the shared data into a private cache, which saves energy. At the end of the operation, the updated data is in the shared LLC. If another core then requests an operation on that line, it will have reduced latency compared to a system without this mechanism.

One interesting case, however, can occur when atomic operations from different cores issue simultaneously or in very quick succession relative to on-die travel times. In such cases the resulting access latency may be little better than the baseline case, as the second core's request must still wait for the first core to finish and push the data back. (see the right side of Figure 4.4).

4.3.5 Direct Access In-Cache Atomics

As Figure 4.4 shows, despite the improvements to atomics, back-to-back (or concurrent) atomic updates by different cores can still suffer round-trip latency similar to the baseline as the target cache line bounces between the LLC and the requesting cores. High-contention scenarios are known to be troublesome already, and avoided when possible in current code; however, they become increasingly likely with more threads, finer grain parallelism, and longer on-chip delays. The technique we describe in this section helps alleviate this pressure.

Even when competing requests are sufficiently separate in time to realize the maximum benefit of pre-emptively pushing data back to the shared cache, we have not reduced the number of cache line transfers among requesting cores. The time and energy for this can be significant relative to the computation performed, even if the on-chip latencies are not at this point on the scale of DRAM accesses [8] or cluster network links [71]. However, as many-core chips grow larger, cross-chip delays will only increase in significance.

For even better performance and efficiency, we now consider performing atomic operations at the shared cache itself, avoiding the round-trip between the cache and the requesting core. Our intention is to enable the cache line to remain at the shared cache, and have the cores send to that shared cache's controller their requests for atomic operations.

From observation of software needs, it is clear that a very limited set of integer-only atomic operations can cover a large swath of synchronization-related sharing scenarios. Examples include mutex locks, dependency counters and similar atomically-incremented data, as well as our shared queue head and tail pointers. In our experiments, we will implement support for both *Compare-and-Swap* and *Fetch-and-Add* style atomic operations, for 64-bit or smaller values.

A simple integer ALU is sufficient to perform these atomic operations. Hence, we model a system where each tile's shared LLC / directory controller includes such an ALU and minimal related logic, woven into its normal coherence processing and

shared LLC read/write logic.[2] While an integer ALU is not a trivial piece of hardware, each tile of our chip already contains a superscalar core with SIMD ALUs, private caches, a shared cache slice and directory, and a network interface; we contend that the space overhead is quite small. Unlike solutions targeting data throughput [39], we do not require the ability to concurrently operate on more than one value per cache line.

Aside from the need for new execution resources at the shared cache controllers, executing atomic operations away from the requesting core presents specific challenges. First, we must still, of course, ensure atomicity for each request. Second, we must convey operands (such as the "swap" value in compare-and-swap) to the execution location. And third, we must convey the result of the operation back to the requesting core (all our supported atomic ops return a value; though not all applications make use of it, it's crucial for some, e.g., lock acquisition).

Atomicity / lack of interference is ensured by relying, for each cache line, on placement at its home tile of shared cache. This creates a unique point of serialization for each address. If we then process arriving requests one at a time, atomicity is implicitly guaranteed. Because this is the same controller that processes other coherence requests relevant to the cache line in question, no correctness issues are introduced if traditional cache accesses mix with in-cache atomics.

Conveying arguments and responses to and from the atomic operation is also straightforward. We enhance the existing cache coherence protocol with new message types for atomic requests and replies. An atomic request may for example contain an op id, the target address, the size of the target data, and an extra argument whose use varies according to the op id. The complimentary reply type is similar, except that it trades the extra argument field for the result of the atomic operation. The size of such a request would certainly not be out of line with that of existing coherence message types.

With this technique the communication pattern for concurrent atomic updates on

[2]We have assumed a system where each slice of shared LLC is co-located with a corresponding slice of coherence directory, such that the corresponding controller has local access to both. For the purposes of this section, however, the directory is the more fundamental of the two, as it serializes updates to the cache line.

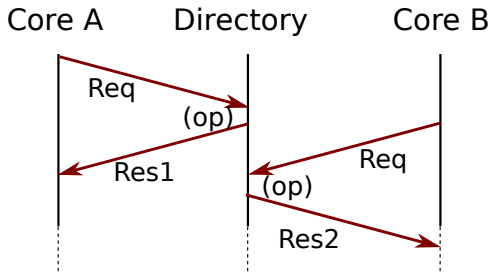


Figure 4.5: If we enable the atomic operation to be performed at the shared cache controller, we reduce even further the latency per request; the data never leaves its home tile, and requesting cores wait for only two network messages to receive their result. These responses may also overlap other requests.

shared data resembles Figure 4.5. This is a significant improvement over Figure 4.4.

Executing such an operation proceeds as follows.

1. At the core requesting the atomic operation, the memory request unit sends an opcode, as well as the non-memory operand, to the private cache controller.

2. The private cache controller ensures the line is not in its private cache, and forwards the request to the line's home directory tile.

3. The directory controller receives the request. If there are pending requests on the target line, it buffers the new request, similar to other pending coherence requests. If not, it prepares to perform the atomic operation by either reading the line from the shared LLC, or requesting write-back from the current private holder — though the latter will be rare under proper usage.

4. Once the cache line is obtained, the controller executes the atomic operation on it, using the information in the request.

5. The controller then checks if further requests for the line are queued up. If so, it performs the next one. If there are no further requests for this line pending, the controller writes the line back to the shared LLC.

6. For each request, the controller may generate and send a response containing the result, according to the specific op, to the requesting core.

Performing atomic updates in this manner presents advantages compared to doing the same operations at the cores. The lack of repeated invalidations and writebacks greatly reduces the latency of the operation (in this regard it improves on the method from Section 4.3.4 during higher contention). It significantly boosts the throughput for atomic operations on the same cache line — these can now happen at a rate of one per cycle, limited by the directory's request processing rate, rather than one every tens or hundreds of cycles, as was previously the case, limited by network delays and coherence transactions. If a line is highly contended and is the target of many atomic operations, this should significantly improve performance. Finally, offloading an atomic operation to the shared cache relieves both the requesting core and its private cache of the need to ensure non-interference. This can be advantageous to performance depending on the specific architecture's prior atomicity solution.

We believe these in-cache atomics are a practical way to bring a useful amount of computation closer to the data in a many-core, on-die cache context. This relies on the existing cache-coherence message infrastructure for requests and replies, and minimizes any loss of compatibility with existing code. Combined with the previously discussed ideas for direct (non-atomic) access to data in both private and shared caches, this completes a very capable toolset for reducing average cache system latency during frequent communication over shared memory.

We expect this significant speedup to come as a result of fewer network traversals and cache coherence events. We expect the energy consumption of the cache system and network to also be correspondingly improved.

In the following sections we proceed to experimentally test the performance of our proposals by applying them to both the dynamic runtime from Chapter 3 and a set of microbenchmarks.

Table 4.2: Experiment Settings

Core	64-128 cores, dual-issue in-order x86 Core-private 32 KB 4-way L1-I$, 1 cycle Core-private 32 KB 8-way L1-D$, 1 cycle
Tile	1 core per tile Per-tile 512 KB 8-way L2$, 16 cycles, inclusive 2MB shared L3$ & directory slice
Inter-connect	2-D mesh, 1 physical network for coherence messages 2 cycles per hop (1 at router and 1 at link), 64-bit links
Directory	MOSI protocol, 20 cycles
Memory	100 cycles
Compiler	GCC 4.8.2 -O3 w/ libgomp

4.4 Experimental Methodology

4.4.1 Modeled System

As with the previous chapter, we simulated and tested our ideas on a modified version of the Graphite PIN-based simulator [68]. Table 4.2 shows the configuration of the simulated system. We use simulation to facilitate architectural exploration, enable detailed fine-grain profiling, and to target chip architectures with hundreds of cores and a scalable mesh interconnect. Other than the number of cores and interconnect design, the simulated configuration is similar to that of a modern many-core processor, such as those in the Intel MIC line [90].

In our simulated many-core chip, as Table 4.2 shows, we have modeled a tiled structure with one core per tile, accessing data through a straightforward cache system consisting of a shared, tiled LLC with co-located coherence directory, which itself supports one private L2 and an L1D/L1I cache pair per tile.

Table 4.3: Applications Tested

App	Description	Challenges	Parallelism
fwdsolver (`fs`)	Solve a sparse lower triangular system	Dependency tracking	Dynamic, data dependent, mostly narrow-wide-narrow
maxflow (`mf`)	Find maximum flow through a graph	Mutable graph; varying # and length of tasks	Dynamic, data dependent, can be wide, then narrow
gtfold (`gt`)	Compute shape of RNA	Variable task duration, decreasing parallelism	Dynamic, widest at start linear narrowing
rt bench (`rtb`)	Microbenchmark for task stealing	Persistent load balance issues in task creation	Dynamic, constant width parallelism
histogram (`hist`)	Microbenchmark for shared counters	Shared counters atomically incremented	Uniform dataset partitioning
ticketlock (`tick`)	Microbenchmark for lock acquisition	Contended locks acquired w/atomic ops	One thread / core, random lock contention

Table 4.4: Datasets Used

App	Descr.	# tasks	Peak Par.	Cycles / Task
`fs`	27-pt 40^3 Lap.	64k	125	2.8k
`mf`	64x32 graph	130k	400	1.4k
`gt`	250 bases	30k	130	17k

4.4.2 Microbenchmarks and Client Applications

The main set of motivating workloads in our tests consists of three main irregular parallel applications, all of which rely on and stress different aspects of our task-stealing dynamic runtime described in Chapter 3. Aside from these applications, we will also use three microbenchmarks. We will describe all of these workloads here.

The runtime in question is the implementation of Chase-Lev style [24] circular-queue-based task stealing that we developed in Chapter 3. In its basic approach it resembles well-known systems such as Cilk [35]. It has one task queue per thread, and relies on atomic operations to implement a low-overhead synchronization protocol appropriate to fine-grain tasks. It is further tailored to both many-core systems and irregular workloads, by use of a hierarchical scheme for efficient gathering and sharing of state, implemented fully in software. This scheme uses cache-line sized data

structures and carefully designed access patterns to enable faster stealing and more efficient assignment of work to threads that need it. It can make use of work-sharing, which some applications find advantageous, and is able to act dynamically on gathered knowledge about global state. Overall, it is a good example of an irregular, fine-grain parallel workload presenting multiple kinds of communication and synchronization patterns.

The three main applications that will use this runtime represent real-world problems, from both the sparse- and graph-data domains. They show interesting data-dependent variability, and stress different aspects of the runtime. They were introduced in the prior chapter, but are listed again in Table 4.3, and Table 4.4 summarizes the inputs chosen for these applications in this experiment. For detailed descriptions, please refer to Section 2.3.1.

One of our microbenchmarks (`rtb`) will also use the runtime; the benchmark will focus on doing little else but continuous fine-grain load balance operations. Whereas the real applications have been optimized to minimize the portion of their execution attributable to the software runtime, this benchmark will illustrate a more extreme dependence on runtime performance.

We will also use two benchmarks that create high contention situations which scale poorly with the number of cores in the system. The first one, `hist`, simply splits a large dataset among all threads, which then collaborate to fill a single, globally shared histogram; synchronization is at the bin-level through atomic operations. The other benchmark, `tick`, creates contention for a limited set of ticket-style locks (as described, for example, in [30]),maintaining the locks to threads ratio constant as the system size scales. During execution, all threads simulate entering and exiting protected sections repeatedly, targeting the locks at random.

4.4.3 Evaluation of Proposed Features

We briefly describe how we make use of direct access to remote caches and in-cache atomic operations, in both our task-stealing runtime and our separate microbenchmarks, to optimize access to shared and synchronization-related data structures according to their corresponding communication patterns. First let us focus on the runtime.

Remember that the runtime uses internally different kinds of metadata (see Figure 3.7 in Chapter 3). These mainly comprise: (a) task queues with both a head and tail each (said head and tail occupy different cache lines and have different usage patterns); (b) cache-line sized nodes, logically organized into a tree structure, holding gathered system state information; (c) the leaves of said tree, each belonging to and maintained by one thread.

We first target the nodes of the state-gathering tree. These are our runtime's main tool for accelerating stealing operations: they are read by thieves during steal attempts, and are written to periodically by group maintainers (who, as often as possible, are idle rather than active threads). Since they are not accessed with atomic operations, and are intended for quick access by thieves, simple shared cache placement is best.

The *leaves* of our metadata tree represent each thread's self-maintained *public view* of its status, and also receive steal hints. Since this data is also not accessed atomically, we can use simple cache placement, either private or shared, according to whether each application is more sensitive to either latency / staleness of gathered state visibility (possible in *task-constrained* operation where this delay may be significant in the critical path), or to the overhead on active threads of maintaining a public state at all (possible in *thread-constrained* operation, where throughput depends on minimizing distractions on active threads). Similar to the "tail" case below, it is not 100% clear whether there is a single overall "best" choice across all cases; this fact helps illustrate the need for flexible solutions, and how even under the same framework, different applications may have different needs.

The *head* of each task queue is a pointer into the circular buffer, used (and contended) only among thieves seeking to obtain tasks, with the steal attempts in question consisting of atomic increments. Shared-cache placement for non-atomic accesses, coupled with in-cache atomic increments, reduces the latency of interleaved steal attempts by multiple threads.

The other side of each thread's task queue is the *tail*. This is where the owner thread enqueues and dequeues. During abundance of tasks, it should suffer little to no interference from other threads. However, if the queue is almost empty, thief threads must interact with the tail to ensure the queue's integrity as they steal. Therefore, the tail might be best placed in either the owner's private cache, so that interference from thieves does not cause local misses that slow down the owner thread; or in the shared cache, so that thieves may see lower latency accesses despite the owner thread's own activity.

As a further optimization specific to `fs` and `gt`, we can use in-cache atomic operations to reduce contention on dependency counters used to trigger new tasks, by accelerating the atomic *fetch and add* operations used to do so (*mf* and *rtb* do not use dependency counters in this way).

We now address the `hist` and `tick` benchmarks. Both of these present situations that demonstrate the slowdowns that can result from contention among increasing numbers of cores. In both cases, performance and efficiency can be significantly improved. In `hist` we optimize the sharing of histogram bins, which all threads concurrently update using atomic increments. In `tick`, we target the ticket locks, which threads repeatedly attempt to enter for brief periods.

In each of these cases we test two configurations for keeping the target lines at the shared cache: First, we use our simpler method, which performs atomic updates at the requesting core, but immediately pushes the data back to the shared cache. Second, we use in-cache atomic operations to avoid moving the cache line at all. We compare both of these to the baseline performance.

4.5 Experimental Results

We first present the impact of our methods on the performance and scalability of our microbenchmarks. Later, we show how using our methods translates into performance and energy efficiency improvements in our irregular applications using dynamic task stealing.

4.5.1 Microbenchmark Performance and Scalability

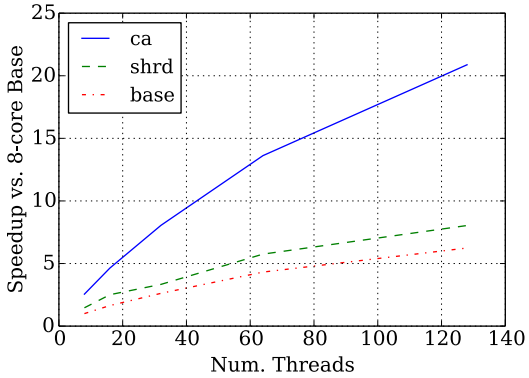


Figure 4.6: Histogram benchmark speedup for baseline (`base`), remote shared access with core-side atomics (`shrd`), and in-cache atomic (`ca`) variants. Performance is normalized to that of `base` on 8 cores.

We evaluated `hist` and `tick` in three configurations: a *baseline* with standard MESI cache coherence [76]; a *shared* variant, doing remote shared cache placement with atomics performed at the requesting core; and a *ca* variant, using in-cache atomics to keep the cache line always at the shared cache. Both benchmarks show significant scaling and performance improvements, as can be seen in Figures 4.6 and 4.7. The relative advantage of in-cache atomics versus direct-access requester-side atomics varies between the two benchmarks.

The *per-event* latencies seen in Figs. 4.8 and 4.9 help explain the significant performance difference. Keeping the cache line at the shared location reduces the latency of each requester's operation by avoiding evictions out of private caches – an increasingly consequential advantage as the system scales. It also allows atomic

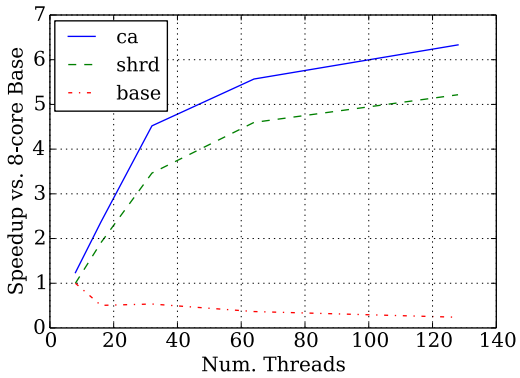


Figure 4.7: Ticket-lock benchmark speedup for baseline (`base`), remote shared access with core-side atomics (`shrd`), and in-cache atomic (`ca`) variants. Performance is normalized to that of `base` on 8 cores.

operations to be done on the cache line at a fundamentally faster rate because the line no longer has to endure transit times across the chip between updates. When atomic operations are done at the cache, only the requests and results must travel; and these can do so concurrently, whereas if the cache line must move, it must do so serially.

In `tick`, baseline performance actually worsens with more cores because, despite maintaining the lock to core ratio constant, random contention over the whole set of locks causes larger systems to have greater probability of temporary imbalances where multiple threads contend for the same lock. The average latency to acquire a lock then increases not only due to longer coherence latencies, but also due to longer queueing times (more threads contending, and each thread's transaction taking longer).

The baseline `hist` does not suffer as heavily from increasing contention, but its concurrent atomic operations on shared data are also sensitive to increasing distances, causing very shallow scaling. As the figures show, for both benchmarks we are able to significantly improve the per-event delay. It is worth noting that even with our best results, increasing system size still causes slower average times per operation, but our hardware support has significantly reduced the intensity of this penalty compared to the baseline.

The histogram benchmark derives a larger portion of its maximum speedup from

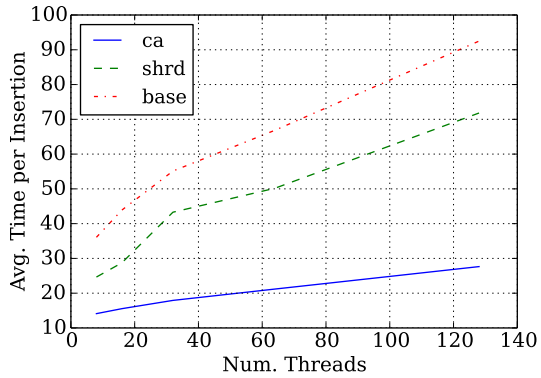


Figure 4.8: Latency in cycles for a single shared bin update in the Histogram benchmark, for the same three variants as Figure 4.6.

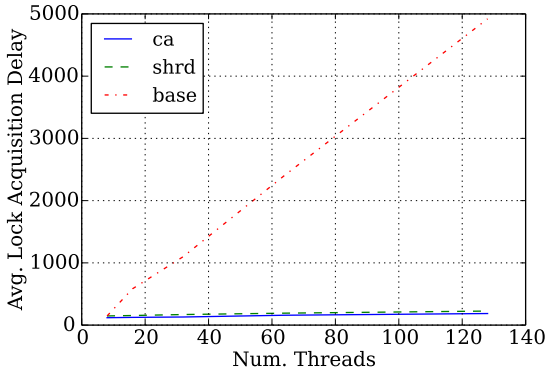


Figure 4.9: Latency in cycles for a single lock acquisition in the Ticket-lock benchmark, for the same three variants as Figure 4.7.

actually moving atomic operations to the shared cache. The ticket-lock benchmark reacts differently, seeing the bulk of its improvement just from holding the lock in shared cache, even if atomic operations are still performed by pulling the cache line to the requesting core and then pushing it back. These differences relate to different uses for the atomic operations: in the ticket-lock benchmark, threads must always wait for the result, as they need this result to decide if it succeeded. The histogram works differently; it uses atomic updates to safely increment counters, but has no immediate use for the result. Sparse direct solvers [61] and a variant of stochastic gradient descent algorithm called HOGWILD! [73] are examples of applications that

exhibit data sharing and synchronization behavior similar to `hist`.

Bringing the atomic operation itself to the shared cache is not as important for `tick` because the rate of issuing atomic accesses is limited by each core needing to receive its result, before being able to check if it is their turn, and being able to proceed to a new atomic access. Other applications where receiving the result of the atomic operation is necessary before a thread can proceed are likely to show similar behavior (including for example the use of head and tail pointers in the dynamic task stealing run-time, as will be discussed in the next section).

4.5.2 Dynamic Task Stealing Performance

We now evaluate private- and shared-cache placement as well as in-cache atomics when used to optimize our task-stealing dynamic runtime. We target different parts of the runtime differently, as proposed earlier, tailoring the application of our hardware features to the expected communication patterns. First we will discuss the combined effect, and later break down the results in greater detail relative to each communication pattern.

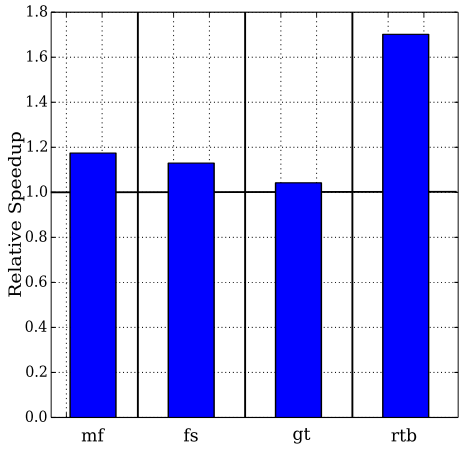


Figure 4.10: Speedup of the dynamic runtime applications on 128 cores, when using direct cache access and in-cache atomics in the runtime as appropriate, relative to Chapter 3's software-only performance.

Figure 4.10 shows the result. An overall speedup of 1.7x is observed for the runtime stealing benchmark; the applications that use the runtime show 1.18x, 1.13x, and 1.04x speedups, even though the runtime already represents only a fraction of their execution.

To get an idea about the source of these speedups, Figure 4.11 shows a breakdown of the average amount of time threads spent in various activities. *Active* time is spent executing actual tasks. *Waiting* time is spent idle (e.g., when there is insufficient work in the system such that some portion of the threads must wait). *Stealing* is time spent both searching for and taking tasks from another thread's queue. *Maintenance* is time spent updating the hierarchical metadata nodes, which make fast searches for work possible. We see here that we have reduced the overall portion of the time spent in runtime overhead activities, as well as positively impacting waiting time and active time.

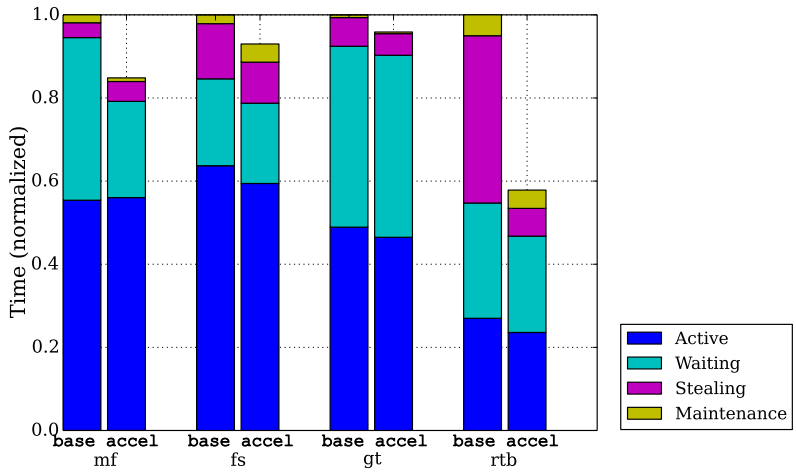


Figure 4.11: Breakdown of execution time for dynamic runtime applications, showing the portion of execution time spent in each activity, comparing an accelerated system vs a SW-only runtime.

Figure 4.12 shows, furthermore, the average time that tasks remain un-stolen during task-constrained periods. This graph demonstrates that our speedups also correlate with more up-to-date information enabling faster steals, which, during task-constrained operation, can contribute to reductions in the app's critical path.

`gt` particularly shows that its overall performance with the software-only runtime was not as strongly limited by the runtime's communication overheads as the other applications. `rtb` conversely shows the potential impact for applications very heavily constrained by such overheads.

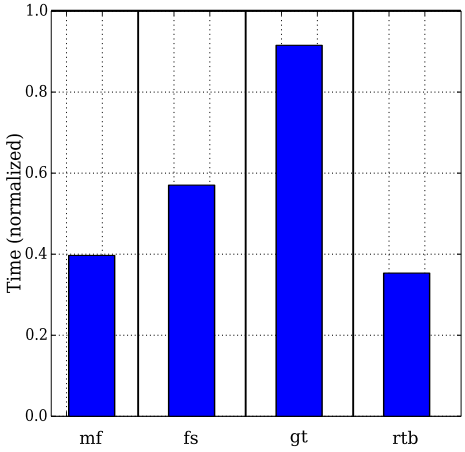


Figure 4.12: Decrease in waiting time of stolen tasks (time between placing a task on an empty queue, and the task being stolen). This impacts overall running time in task-constrained conditions by affecting the critical path. Data from 128 cores, normalized to the SW-only system.

We will briefly drill into some of the specific results from targeting individual data structures in our software runtime. Note the numbers cited are, as above, speedups to the overall application. The underlying improvements to the runtime are larger, but we report bottom-line effects on the applications.

Most important was the targeting of the *head* of the task queues with shared placement and in-cache atomic operations. This reduced the latency of interleaved updates from thieves, and yielded significant speedups for all applications, including 49% for `rtb`. When targeting the *tail* of the same queues, which are mostly used by owner threads (except during task-constrained situations), we also observed positive, though smaller, speedup, with 6% for `rtb`, and a higher 16% on `mf`.

Placement of the state-gathering nodes in shared cache also yielded a speedup, as expected, with 15% on mf, and 2% to 4% on the other applications. Finer grain

statistics reveal that this placement significantly reduces the time cost to thief threads of consulting the information in these nodes, speeding up stealing activities.

Targeting the *leaves* of this same tree, we obtained a small boost, mainly from shared-cache placement, which can reduce the staleness of the tree's contents by speeding up the state-gathering process. Interestingly, however, `rtb` slightly favored private cache placement of these leaves. We can hypothesize that the benchmark's short tasks and relatively high system utilization emphasize reducing the performance penalty on active threads rather than improving steal information.

Finally, targeting dependency counters used for triggering enqueues of new tasks, `fs` sees greater performance improvement than `gt` (about 8% vs. 1% overall, respectively), due to differences in both the numbers of dependencies per task (`fs` can potentially have many more), and the length of said tasks (`gt`'s tasks are much longer, reducing the relevance of per-task overheads).

One final piece of metadata that exists in the runtime but which we have not mentioned, are the buffers which at each queue hold the enqueued task descriptors. Our profiling revealed that this data structure is accessed comparatively seldom, and is neither used for actual synchronization, as the queue head and tail are, nor is it an object of significant contention. As such, its contribution to the runtime overhead is very small, and it was not included in our analysis.

4.5.3 Dynamic Task Stealing Energy Efficiency

As expected, along with our performance results we observe reduced energy use when we operate on data directly at the shared cache, and use in-cache atomic ops. (Figure 4.13). The principal source of these energy savings is a significant reduction in L1 cache activity, because our shared-cache memory accesses, as well as our in-cache atomic operation requests, both bypass L1 and L2. Though this is achieved at the expense of increased usage of the core's stream buffer as well as the chip's L3 caches, it is well worth the trade-off in this case. There is also a reduction in network energy, but it is more subtle; a lot (not all) of our optimizations' advantage lies simply in pre-positioning data in places where it would later be required to be; this doesn't

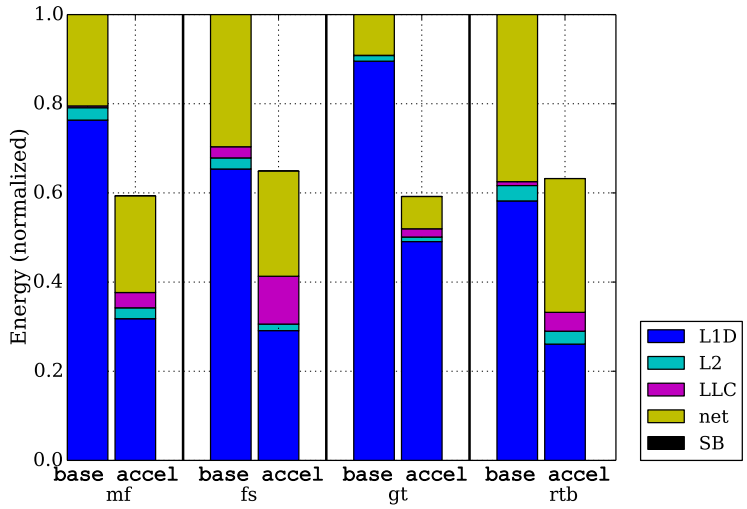


Figure 4.13: Decrease in cache-system energy due to direct cache access and in-cache atomic operations, including both private L1 & L2 caches, the shared LLC, and the on-chip network. 128 cores.

necessarily reduce as much as time-shift network activity.

Energy efficiency was not the main target of our efforts; it is possible that the savings in memory system energy might be further reduced in future experiments focused on low power. Furthermore we have not accounted for any core energy savings, such as might result from finishing execution sooner.

4.6 Conclusion

We have shown that irregular, fine-grain parallelism can create performance and efficiency challenges for many-core coherent-cache systems, when it comes to either synchronization through shared memory accesses, or unpredictable access patterns on shared datasets. We have proposed a set of ISA extensions enabling purposeful placement of data in non-local on-chip caches with minimal extra hardware requirements; This handful of extensions covers a gamut of use cases, allowing for efficient sharing of data with either known or unknown consumers, by bypassing local private caches and operating directly on remote shared or private caches. We also proposed

two different methods for support of atomic operations on such remotely placed data, both oriented towards reducing access latencies on interleaved atomic operations. If we're willing to augment our on-chip cache hierarchy in specific ways (without losing compatibility with existing functions), we can dramatically speed up synchronization schemes or fine-grain data sharing reliant on certain kinds of highly-contended structures.

We demonstrated the usefulness of these extensions by using them to optimize a runtime with fine-grain, low overhead synchronization needs, resulting in dramatic speedups on both benchmarks and real applications that make use of this runtime.

In the future we can expand this exploration to more applications (particularly implementing more classic synchronization primitives), and explore more concretely both the energy and practicality implications, as well as scalability trends on even larger, thousand-core scale systems.

Chapter 5

Conclusion

As available general-purpose processors reach the hundred-core scale, compute capability grows thanks to core counts and transistor improvements. Propagation delays across said chips, though, fail to improve concurrently due to unfavorable physical scaling effects.

Compounding this physical scaling is the protocol overhead of communicating and synchronizing over shared memory and coherent cache. This is the most common model of parallel programming, due partly to programming simplicity, and partly to the lack of widespread availability of more direct methods.

Regular applications can mitigate scaling issues by reducing communication through schemes such as prefetching, partitioning, and blocking. But irregular parallelism, such as is often found in graph and sparse applications, remains vulnerable. Task durations, dependencies, and memory access patterns all may vary both between and during executions, forcing greater reliance on dynamic scheduling. Furthermore, on hundreds of cores, available parallelism may vary relative to hardware thread count more strongly than is usually expected, creating periods of relative scarcity during which different priorities must drive the scheduler's work.

Though there is much research on task stealing as a method of dynamic scheduling, existing solutions have arisen mostly assuming one of two scenarios: few cores, fast communication, and abundant parallelism; or numerous cores, very long network latencies, and distributed memory. Neither scenario captures the same issues we

observe.

In Chapter 3 we propose a runtime with hierarchical gathering of state, reducing stealing latencies while allowing for more sophisticated policies, and creating a global view of the system's status, which enables adaptation over time. This runtime keeps overhead low when work is abundant, but also facilitates quick stealing in scarcity conditions. Increased efficiency allows portions of the system to idle during narrow phases, opening the door for power savings or multi-programming. Finally, the runtime is implemented fully in software, unlike earlier solutions.

Communication and synchronization challenges on many-core chips have also been the subject of research. However, proposed solutions tend to be too specific to one primitive or pattern, too different in programming model, or too impractical to implement widely, from either a software or hardware perspective. In general, high-performance hardware solutions for scaling-related communication issues are not lacking, but what is lacking is solutions that are general-purpose, compatible, and economically viable in hardware and software.

In Chapter 4 we bring together a set of techniques enabling direct interaction (including atomic operations) on data in remote caches, without correctness risks or large barriers to entry for users. The manner in which these are incorporated into the coherent cache system results in a practical, compatible, and general-purpose solution. First, we propose cache-placement ISA extensions which, implemented with minimal hardware support, allow for efficient sharing between producers and consumers by directly targeting remote or shared caches. Second, we present two proposals for accelerating concurrent atomic updates to data, with significant performance upside for synchronization primitives and atomically shared data. In both cases we require only minimal changes to user code and cache hardware, and maintain compatibility with existing code.

Combined, our software optimization and hardware proposals significantly improve the performance, scaling, and efficiency of irregular parallel applications, as well as other synchronization-limited code, when executed on chips with hundreds of cores.

Below we review our claimed contributions:

- Through analysis of graph and sparse applications of emerging importance, we show how the combination of irregular parallelism and hundred-core chips highlights growing communication latencies and the breakdown of abundance assumptions in task-stealing as urgent scaling challenges.

- Because multi-core-targeted dynamic runtimes become inefficient at large core counts, we develop a task-stealing runtime designed for this scenario, one which embraces a more dynamic relationship between available parallelism and system size. On a 128-core system, it cuts runtime overheads by 60% to improve overall target workload performance by 15% and reduce core utilization by 29%. It does so through proper handling of task-constrained situations; dynamic adaptation for efficiency during abundant times; and a combined stealing / pushing approach. It requires no hardware support, unlike prior solutions.

- Because communication through shared data is very common in parallel code, and creates growing performance penalties in many-core chips, we introduce cache-coherence-level support for direct access to private and shared remote caches. This reduces latencies and network traffic during shared access to data. Using our tools, common producer-consumer communication patterns can result in faster and more efficient on-chip data movement.

- Because atomic operations carry large performance penalties despite being crucial to synchronization over shared memory, we also propose limited execution of atomic operations on shared-cache data, and at remote tiles. Combined with direct access to remote caches, these tools substantially reduce excess cache traffic and latency, enabling much improved scaling and many-fold speedups on synchronization benchmarks.

- Because practicality, compatibility, and usability are important to adoption of architectural features into real-world systems, we demonstrate how our hardware features can be integrated into a general-purpose architecture, without resulting in large code rewrites, loss of coherence, new memory models, or significant practical implementation issues. Their application to our optimized

task-stealing runtime demonstrates both ease of use and performance potential. On a 128-core system, runtime stealing benchmarks see a 70% performance boost, and user applications an average of 12% speedup beyond the optimized software baseline, along with cache system power savings.

5.1 Future Work

There are many directions open to further study, at levels more or less directly related to the scope of work already covered.

One crucial angle when continuing the study of many-core systems and irregular applications will necessarily be, how to make productive use of cores (or their share of energy) not otherwise occupied during applications narrow-parallelism periods, without harming the ability for parallelism to grow again. Research into fine-grain power management could explore how to reduce overall power density, or how to apply extra power to speed up the portions of the system still occupied. Another direction could be a scheme for prioritized multi-programming; maintaining full system utilization by advancing secondary work during "lulls" in the primary workload.

In the future, when many-core platforms with sufficient numbers of cores come to market, it would be interesting to test our software ideas on real silicon, as much to validate our results as to enable head-to-head comparisons against the scaling of existing task-stealing libraries, an aspect not included in this work due mainly to practical issues. It is not clear exactly when sufficiently large systems with the right structure will exist.

Within the scope of the task-stealing runtime, further work exists in multiple directions. One of them is wider application of dynamic adaptation based on known state. The gains we have shown are only the beginning; heuristics for turning knowledge into task-stealing policies, for example, warrant further study. Another possible direction is to expand the scope of the runtime beyond task management, into other common kinds of metadata (e.g., task-dependency tracking), or even to include some awareness of the client application's own data locality. Though this is extra difficult in irregular applications, it might be relevant especially as we seek to translate our

learned lessons into systems with deeper cache hierarchies with significant grouped tiers. One final thought is the eventual possibility of combining hundred-core scale chips in multi-socket or networked setups to produce very compact and powerful cluster-scale computing. A software runtime capable of executing irregular parallelism efficiently in this context would need to operate efficiently at multiple scales, with a possibly hybrid approach to on- and off-chip task management.

On the hardware front, the seamless combination of "automatic" cache coherence with low-risk, sharing-oriented placement and atomicity bears further study. We have for example not explored remote atomic operations on data in *private* caches. Practical support for this might require dealing with trickier ownership issues or relaxing the consistency model, weakening the cost/benefit analysis. Beyond fully exploring the potential for in-cache atomic operations to enable sharing-heavy algorithms to thrive, one may also explore whether any atomic operations beyond those we propose should be supported at the cache controller. Candidates to evaluate could include both non-integer operations and operations on values larger than 64 bits.

Bibliography

[1] Combinatorial Blas v 1.3. `http://gauss.cs.ucsb.edu/~aydin/CombBLAS/html/`.

[2] Galois v 2.2.0. `http://iss.ices.utexas.edu/?p=projects/galois/download`.

[3] Graphlab v 2.2. `http://graphlab.org`.

[4] José L Abellán, Juan Fernández, and Manuel E Acacio. A g-line-based network for fast and efficient barrier synchronization in many-core cmps. In *Parallel Processing (ICPP), 2010 39th International Conference on*, pages 267–276. IEEE, 2010.

[5] José L. Abellán, Juan Fernández, and Manuel E. Acacio. Efficient hardware barrier synchronization in many-core cmps. *IEEE Trans. Parallel Distrib. Syst.*, 23:1453–1466, 2012.

[6] Umut A Acar, Arthur Charguéraud, and Mike Rainey. Scheduling parallel programs by work stealing with private deques. In *ACM SIGPLAN Notices*, volume 48, pages 219–228. ACM, 2013.

[7] Ramesh C. Agarwal, F.G. Gustavson, and M. Zubair. Improving performance of linear algebra algorithms for dense matrices, using algorithmic prefetch. *IBM Journal of Research and Development*, 38(3):265–275, May 1994.

[8] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. In

Proceedings of the 42Nd Annual International Symposium on Computer Architecture, ISCA '15, pages 105–117, New York, NY, USA, 2015. ACM.

[9] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 336–348, New York, NY, USA, 2015. ACM.

[10] Kathirgamar Aingaran, David Smentek, Thomas Wicki, Sumti Jairath, Georgios Konstadinidis, Serena Leung, Paul Loewenstein, Curtis McAllister, Stephen Phillips, Zoran Radovic, et al. M7: Oracle's next-generation sparc processor. *IEEE Micro*, (2):36–45, 2015.

[11] PatrickR. Amestoy, IainS. Duff, Jean-Yves LExcellent, and Jacko Koster. Mumps: A general purpose distributed memory sparse solver. In Tor Srevik, Fredrik Manne, AssefawHadish Gebremedhin, and Randi Moe, editors, *Applied Parallel Computing. New Paradigms for HPC in Industry and Academia*, volume 1947 of *Lecture Notes in Computer Science*, pages 121–130. Springer Berlin Heidelberg, 2001.

[12] Richard J. Anderson and João C. Setubal. On the parallel implementation of goldberg's maximum flow algorithm. In *Proceedings of the Fourth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '92, pages 168–177, New York, NY, USA, 1992. ACM.

[13] M. Arora, S. Nath, S. Mazumdar, S.B. Baden, and D.M. Tullsen. Redefining the role of the cpu in the era of cpu-gpu integration. *Micro, IEEE*, 32(6):4–16, Nov 2012.

[14] J Eric Baldeschwieler, Robert D Blumofe, and Eric A Brewer. A tlas: an infrastructure for global computing. In *7th ACM SIGOPS workshop: Systems support for worldwide applications*, pages 165–172, 1996.

[15] Albert-László Barabási and Réka Albert. Emergence of Scaling in Random Networks. *Science*, 276(5439), 1999.

[16] Petra Berenbrink et al. The natural work-stealing algorithm is stable. *SIAM Journal on Computing*, 32(5):1260–1279, 2003.

[17] RD Blumofe and CE Leiserson. Scheduling multithreaded computations by work stealing. In *Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on*, pages 356–368. IEEE, 1994.

[18] Mark Bohr. A 30 year retrospective on dennard's mosfet scaling paper. *Solid-State Circuits Society Newsletter, IEEE*, 12(1):11–13, 2007.

[19] Mark T Bohr et al. Interconnect scaling-the real limiter to high performance ulsi. In *International Electron Devices Meeting*, pages 241–244. INSTITUTE OF ELECTRICAL & ELECTRONIC ENGINEERS, INC (IEEE), 1995.

[20] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM.

[21] Shekhar Borkar. Thousand core chips: A technology perspective. In *Proceedings of the 44th Annual Design Automation Conference*, DAC '07, pages 746–749, New York, NY, USA, 2007. ACM.

[22] Aydn Buluc and John R Gilbert. The combinatorial blas: design, implementation, and applications. *HPCA*, 25(4):496–509, 2011.

[23] Irina Calciu, Tatiana Shpeisman, Gilles Pokam, and Maurice Herlihy. Improved single global lock fallback for best-effort hardware transactional memory. *9th ACM SIGPLAN Wkshp. on Transactional Computing*, 2014.

[24] David Chase and Yossi Lev. Dynamic circular work-stealing deque. In *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '05, pages 21–28, New York, NY, USA, 2005. ACM.

[25] Guojing Cong et al. Solving large, irregular graph problems using adaptive work-stealing. In *Parallel Processing, 2008. ICPP'08. 37th International Conference on*, pages 536–545. IEEE, 2008.

[26] D. E. Culler and G. K. Maa. Assessing the benefits of fine-grain parallelism in dataflow programs. In *Proceedings of the 1988 ACM/IEEE Conference on Supercomputing*, Supercomputing '88, pages 60–69, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[27] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[28] William J. Dally, Francois Labonte, Abhishek Das, Patrick Hanrahan, Jung-Ho Ahn, Jayanth Gummaraju, Mattan Erez, Nuwan Jayasena, Ian Buck, Timothy J. Knight, and Ujval J. Kapasi. Merrimac: Supercomputing with streams. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, SC '03, pages 35–, New York, NY, USA, 2003. ACM.

[29] Andrew Danowitz, Kyle Kelley, James Mao, John P. Stevenson, and Mark Horowitz. Cpu db: Recording microprocessor history. *Commun. ACM*, 55(4):55–63, April 2012.

[30] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 33–48. ACM, 2013.

[31] James Dinan et al. Scalable work stealing. In *Conference on High Performance Computing Networking, Storage and Analysis*, page 53. ACM, 2009.

[32] Jack Dongarra and Michael A. Heroux. Toward a New Metric for Ranking High Performance Computing Systems. Technical Report 4744, Sandia National Laboratories, 2013.

[33] Jr. Dunigan, T.H., J.S. Vetter, III White, J.B., and P.H. Worley. Performance evaluation of the cray x1 distributed shared-memory architecture. *Micro, IEEE*, 25(1):30–40, Jan 2005.

[34] Derek L Eager, Edward D Lazowska, and John Zahorjan. A comparison of receiver-initiated and sender-initiated adaptive load sharing. *ACM SIGMETRICS Performance Evaluation Review*, 13(2):1–3, 1985.

[35] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the cilk-5 multithreaded language. In *SIGPLAN PLDI '98*, pages 212–223, 1998.

[36] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35(4):921–940, October 1988.

[37] Allan Gottlieb. An overview of the nyu ultracomputer project. In *Experimental Parallel Computing Architectures*, pages 25–95. Elsevier Science Publishers, 1987.

[38] Boris Grot, Joel Hestness, Stephen W. Keckler, and Onur Mutlu. Kilo-noc: A heterogeneous network-on-chip architecture for scalability and service guarantees. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 401–412, New York, NY, USA, 2011. ACM.

[39] Jayanth Gummaraju, Mattan Erez, Joel Coburn, Mendel Rosenblum, and William J. Dally. Architectural support for the stream execution model on general-purpose processors. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 3–12, Washington, DC, USA, 2007. IEEE Computer Society.

[40] J. Gunnels, C. Lin, G. Morrow, and R. van de Geijn. A flexible class of parallel matrix multiplication algorithms. In *Parallel Processing Symposium, 1998. IPPS/SPDP 1998. Proceedings of the First Merged International ... and Symposium on Parallel and Distributed Processing 1998*, pages 110–116, Mar 1998.

[41] Yi Guo. *A scalable locality-aware adaptive work-stealing scheduler for multi-core task parallelism*. PhD thesis, Rice University, Houston, 2010.

[42] Danny Hendler and Nir Shavit. Non-blocking steal-half work queues. In *PODC 2002*, pages 280–289, New York, NY, USA, 2002. ACM.

[43] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.

[44] Mark D Hill, James R Larus, Steven K Reinhardt, and David A Wood. Cooperative shared memory: software and hardware for scalable multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 11(4):300–318, 1993.

[45] Mark D. Hill and Michael R. Marty. Amdahls law in the multicore era. *IEEE COMPUTER*, 2008.

[46] Sungpack Hong, Nicole C Rodia, and Kunle Olukotun. On fast parallel detection of strongly connected components (scc) in small-world graphs. In *High Performance Computing, Networking, Storage and Analysis (SC), 2013 International Conference for*, pages 1–11. IEEE, 2013.

[47] Benot Hudson, Gary L. Miller, and Todd Phillips. Sparse parallel delaunay mesh refinement, 2007.

[48] Takeshi Iwashita, Hiroshi Nakashima, and Yasuhito Takahashi. Algebraic Block Multi-Color Ordering Method for Parallel Multi-Threaded Sparse Triangular Solver in ICCG Method. In *International Symposium on Parallel and Distributed Processing (IPDPS)*, 2012.

[49] Mark C Jeffrey, Suvinay Subramanian, Cong Yan, Joel Emer, and Daniel Sanchez. A scalable architecture for ordered parallelism. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 228–241. ACM, 2015.

[50] Wenhao Jia, Kelly A. Shaw, and Margaret Martonosi. Characterizing and improving the use of demand-fetched caches in gpus. In *Proceedings of the 26th*

ACM International Conference on Supercomputing, ICS '12, pages 15–24, New York, NY, USA, 2012. ACM.

[51] Weihang Jiang, Jiuxing Liu, Hyun-Wook Jin, Dhabaleswar K Panda, Darius Buntinas, Rajeev Thakur, and William D Gropp. Efficient implementation of mpi-2 passive one-sided communication on infiniband clusters. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 68–76. Springer, 2004.

[52] Alain Kägi, Doug Burger, and James R Goodman. Efficient synchronization: let them eat qolb. In *ACM SIGARCH Computer Architecture News*, volume 25, pages 170–180. ACM, 1997.

[53] John H. Kelm, Matthew R. Johnson, Steven S. Lumettta, and Sanjay J. Patel. Waypoint: Scaling coherence to thousand-core architectures. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 99–110, New York, NY, USA, 2010. ACM.

[54] Changkyu Kim, Tim Kaldewey, Victor W. Lee, Eric Sedlar, Anthony D. Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. Sort vs. hash revisited: Fast join implementation on modern multi-core cpus. *Proc. VLDB Endow.*, 2(2):1378–1389, August 2009.

[55] Milind Kulkarni et al. How much parallelism is there in irregular applications? In *ACM sigplan notices*, volume 44, pages 3–14. ACM, 2009.

[56] Milind Kulkarni and Keshav Pingali. Optimistic parallelism requires abstractions. In *In PLDI*. ACM Press, 2007.

[57] Sanjeev Kumar, Christopher J. Hughes, and Anthony Nguyen. Carbon: Architectural support for fine-grained parallelism on chip multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 162–173, New York, NY, USA, 2007. ACM.

[58] George Kurian, Omer Khan, and Srinivas Devadas. The locality-aware adaptive cache coherence protocol. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 523–534, New York, NY, USA, 2013. ACM.

[59] Nevin Krman, Meyrem Krman, Rajeev K. Dokania, Jose F. Martnez, Alyssa B. Apsel, Matthew A. Watkins, and David H. Albonesi. Leveraging optical technology in future bus-based chip multiprocessors. In *in Proc. 39th Int. Symp. Microarchitecture*, pages 492–503, 2006.

[60] Sanghoon Lee, D. Tiwari, D. Solihin, and J. Tuck. Haqu: Hardware-accelerated queueing for fine-grained threading on a chip multiprocessor. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 99–110, Feb 2011.

[61] Xiaoye S Li. An overview of superlu: Algorithms, implementation, and user interface. *ACM Transactions on Mathematical Software (TOMS)*, 31(3):302–325, 2005.

[62] Jiuxing Liu, Jiesheng Wu, and Dhabaleswar K Panda. High performance rdma-based mpi implementation over infiniband. *International Journal of Parallel Programming*, 32(3):167–198, 2004.

[63] Gabriel H. Loh. The cost of uncore in throughput-oriented many-core processors. In *In Proc. of Workshop on Architectures and Languages for Troughput Applications (ALTA)*, 2008.

[64] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. GraphLab: A new parallel framework for machine learning. In *UAI*, Catalina Island, California, July 2010.

[65] Kaisheng Ma et al. Architecture exploration for ambient energy harvesting nonvolatile processors. In *HPCA 2015*, pages 526–537, Feb 2015.

[66] Amrita Mathuriya, David A. Bader, Christine E. Heitsch, and Stephen C. Harvey. Gtfold: A scalable multicore code for rna secondary structure prediction. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, pages 981–988, New York, NY, USA, 2009. ACM.

[67] Ulrich Meyer and Peter Sanders. Delta-stepping: A parallel single source shortest path algorithm. In *ESA*, pages 393–404, 1998.

[68] J.E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A distributed parallel simulator for multicores. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–12, Jan 2010.

[69] Seung-Jai Min et al. Hierarchical work stealing on manycore clusters. In *5th Conf. on Partitioned Global Address Space Prog. Models*, 2011.

[70] Gordon E Moore. Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp. 114 ff. *IEEE Solid-State Circuits Newsletter*, 3(20):33–35, 2006.

[71] S. Narravula, A. Mamidala, A. Vishnu, K. Vaidyanathan, and D.K. Panda. High performance distributed lock management services using network-based remote atomic operations. In *Cluster Computing and the Grid, 2007. CCGRID 2007. Seventh IEEE International Symposium on*, pages 583–590, May 2007.

[72] J. Nilsson, A. Landin, and P. Stenstrom. The coherence predictor cache: a resource-efficient and accurate coherence prediction infrastructure. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pages 10–17, April 2003.

[73] Feng Niu, Benjamin Recht, Christopher Re, and Stephen J. Wright. HOGWILD!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. In *Advances in Neural Information Processing Systems (NIPS)*, 2011.

[74] Stephen Olivier and Jan Prins. Scalable dynamic load balancing using upc. In *Parallel Processing, 2008. ICPP'08. 37th International Conference on*, pages 123–131. IEEE, 2008.

[75] Marc S Orr, Shuai Che, Ayse Yilmazer, Bradford M Beckmann, Mark D Hill, and David A Wood. Synchronization using remote-scope promotion. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 73–86. ACM, 2015.

[76] Mark S Papamarcos and Janak H Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *ACM SIGARCH Computer Architecture News*, volume 12, pages 348–354. ACM, 1984.

[77] Jongsoo Park, Mikhail Smelyanskiy, Karthikeyan Vaidyanathan, Alexander Heinecke, Dhiraj D. Kalamkar, Xing Liu, Md. Mostofa Ali Patwary, Yutong Lu, and Pradeep Dubey. Efficient Shared-Memory Implementation of High-Performance Conjugate Gradient Benchmark and Its Application to Unstructured Matrices. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2014.

[78] Jongsoo Park, Richard M. Yoo, Daya S. Khudia, Christopher J. Hughes, and Daehyun Kim. Location-aware cache management for many-core processors with deep cache hierarchy. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 20:1–20:12, New York, NY, USA, 2013. ACM.

[79] Keshav Pingali et al. The tao of parallelism in algorithms. In *PLDI*, pages 12–25, New York, NY, USA, 2011. ACM.

[80] Jean-Noël Quintin and Frédéric Wagner. Hierarchical work-stealing. In *Euro-Par 2010-Parallel Processing*, pages 217–229. Springer, 2010.

[81] Krishna K. Rangan et al. Thread motion: Fine-grained power management for multi-core systems. In *ISCA 2009*, pages 302–313. ACM, 2009.

[82] Haris Ribic and David Yu. Energy-efficient work-stealing language runtimes. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, pages 513–528. ACM, 2014.

[83] Alberto Ros and Stefanos Kaxiras. Complexity-effective multicore coherence. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 241–252. ACM, 2012.

[84] Youcef Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, 2003.

[85] D. Sanchez and C. Kozyrakis. Scd: A scalable coherence directory with flexible sharer set encoding. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pages 1–12, Feb 2012.

[86] Daniel Sanchez, Richard M. Yoo, and Christos Kozyrakis. Flexible architectural support for fine-grain scheduling. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 311–322, New York, NY, USA, 2010. ACM.

[87] Steven L. Scott. Synchronization and communication in the t3e multiprocessor. In *ASPLOS*, 1996.

[88] Naomi Seki et al. A fine-grain dynamic sleep control scheme in mips r3000. In *IEEE ICCD 2008*, pages 612–617, 2008.

[89] Jiwon Seo, Jongsoo Park, Jaeho Shin, and Monica S Lam. Distributed sociaLite: A datalog-based language for large-scale graph analysis. *Proceedings of the VLDB Endowment*, 6(14), 2013.

[90] A Sodani. Knights landing: 2nd generation intel" xeon phi" processor. In *August issue of Proceedings of Hot Chips: A Symposium on High Performance Chips*, 2015.

[91] T. Soga, H. Sasaki, T. Hirao, M. Kondo, and K. Inoue. A flexible hardware barrier mechanism for many-core processors. In *Design Automation Conference (ASP-DAC), 2015 20th Asia and South Pacific*, pages 61–68, Jan 2015.

[92] Markus Steinberger, Michael Kenzel, Pedro Boechat, Bernhard Kerbl, Mark Dokter, and Dieter Schmalstieg. Whippletree: Task-based scheduling of dynamic workloads on the gpu. *ACM Trans. Graph.*, 33(6):228:1–228:11, November 2014.

[93] Trur Biskopst Strm, Wolfgang Puffitsch, and Martin Schoeberl. Chip-multiprocessor hardware locks for safety-critical java. In *JTRES*, 2013.

[94] Alexandros Tzannes, George C Caragea, Rajeev Barua, and Uzi Vishkin. Lazy binary-splitting: a run-time adaptive work-stealing scheduler. *ACM Sigplan Notices*, 45(5):179–190, 2010.

[95] Stanley Tzeng, Anjul Patney, and John D. Owens. Task management for irregular-parallel workloads on the gpu. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, pages 29–37, Aire-la-Ville, Switzerland, Switzerland, 2010. Eurographics Association.

[96] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ISCA '92, pages 256–266, New York, NY, USA, 1992. ACM.

[97] Jin Wang, Norm Rubin, Albert Sidelnik, and Sudhakar Yalamanchili. Dynamic thread block launch: A lightweight execution mechanism to support irregular applications on gpus. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 528–540, New York, NY, USA, 2015. ACM.

[98] G. Wellein, G. Hager, T. Zeiser, M. Wittmann, and H. Fehske. Efficient temporal blocking for stencil computations by multicore-aware wavefront parallelization.

In *Computer Software and Applications Conference, 2009. COMPSAC '09. 33rd Annual IEEE International*, volume 1, pages 579–586, July 2009.

[99] R.M. Yoo, C.J. Hughes, K. Lai, and R. Rajwar. Performance evaluation of intel's transactional synchronization extensions for high-performance computing. In *High Performance Computing, Networking, Storage and Analysis (SC), 2013 International Conference for*, pages 1–11, Nov 2013.