

Scheduling Task-Parallel Programs in a Multiprogram Workload

Karl Fenech

Supervisors: *Murray Cole*

Christian Fensch



MSc Computer Science

School of Informatics

University of Edinburgh

2013

Abstract

With commodity multicore architectures already prevalent, the microprocessor industry is poised to leap into the manycore era within the next few years. To avail of such machines' multiprocessing capabilities, software developers are increasingly incentivized to parallelize their programs. This trend poses new challenges for the system scheduler, which will now need to consider the parallel characteristics of the respective programs when seeking to maximize the system-level performance of its multiprogram workloads.

In this project, we reconcile two orthogonal approaches: work-stealing task-scheduling for efficiently unravelling structured parallelism from each program, and scalability-based processor-partitioning for dynamically optimizing the programs' core allocations within the context of the current multiprogram workload on the manycore machine. We experimentally demonstrate that, for low- to moderate-scalability programs, our multiprogram scheduler can succeed in achieving modest improvements over mainstream thread and task schedulers.

Keywords: parallel programming; structured parallelism; algorithmic skeletons; divide-and-conquer; manycore multiprocessing; multiprogram workloads; processor partitioning; scheduling; scalability; dynamic optimization

Acknowledgements

My heartfelt gratitude goes to my supervisors, Dr Murray Cole and Dr Christian Fensch, who have been a steady source of insight, guidance, and support throughout the entirety of this project, from the inception of the idea, right up to their detailed feedback on the final drafts of this document. It is an enriching privilege to get the opportunity to work under the supervision of such accomplished individuals who still exhibit enthusiastic interest in your progress. This experience has been both exciting and educational, imparting me with a profound appreciation and knowledgeability of this area of research.

I would like to thank the EPCC, especially Dr Mark Bull and Dr David Henty, who kindly accepted to grant me free access to the Morar manycore server, on which all our experiments (collectively spanning hundreds of compute hours) were run.

My thanks also go to my lecturers and tutors from the [University of Edinburgh](#) for delivering this quality degree, as well as the Informatics Teaching Organisation (ITO) staff for their helpfulness. On a retrospective note, I remain grateful towards my past teachers and mentors, both from academia and from industry, who reinforced my passion for the subject over the years. In particular, I would like to thank Prof Gordon Pace for encouraging me to pursue postgraduate studies.

Special thanks go to my family and close friends for their unwavering support and encouragement throughout all the years. Finally, I salute Gavi, 'Tasha, and 3PT, for their role in reifying the trans-spectral.

“The biggest sea change in software development since the OO revolution is knocking at the door, and its name is Concurrency.” — Herb Sutter

The research work disclosed in this publication is partially funded by the Strategic Educational Pathways Scholarship Scheme (Malta).

The scholarship is part-financed by the European Union – European Social Fund.



Operational Programme II – Cohesion Policy 2007–2013
Empowering People for More Jobs and a Better Quality of Life
Scholarship part-financed by the European Union
European Social Fund (ESF)
Co-financing rate: 85% EU Funds; 15% National Funds



Investing in your future

Table of Contents

Chapter 1: Introduction.....	1
1.1 Aim	1
1.2 Hypothesis.....	2
1.3 Contributions	3
1.4 Common Symbols	4
1.5 Document Structure.....	5
1.5.1 Submission Details	6
Chapter 2: Background.....	7
2.1 The Concurrency Revolution.....	7
2.2 Parallel Abstractions	9
2.2.1 Threaded Programming	10
2.2.2 Task Parallelism	11
2.2.3 Structured Parallelism.....	12
2.2.4 Divide-and-Conquer Skeleton	13
2.3 Thread Scheduling.....	15
2.3.1 Processor Partitioning.....	15
2.3.2 Program Scalability	17
2.4 Performance Metrics	17
2.4.1 Turnaround Time	18
2.4.2 Throughput	19
2.4.3 Variability	19
2.5 Parallel Programming in .NET Framework	20
Chapter 3: Related Work.....	21
3.1 Task Schedulers.....	21
3.1.1 Skandium.....	21
3.1.2 Task Parallel Library	22
3.1.3 Work-Stealing Task Scheduler	24
3.2 Partitioning Schedulers.....	28
Chapter 4: Skeleton Designs.....	29
4.1 Divide-and-Conquer Skeleton	29
4.1.1 Functional Parameterization.....	29
4.1.2 Execution Interface	30
4.1.3 Scheduling Extensibility.....	31
4.1.4 Declarative Model.....	31
4.1.5 Object-Oriented Model.....	32
4.2 Task Generation	33
4.2.1 Fork-Join Pattern	33
4.2.2 Asynchronous Execution.....	35

4.3	Program Hierarchy	39
4.4	Sample Programs	41
4.4.1	Sum of Squares	42
4.4.2	Monte Carlo Pi	43
4.4.3	Strassen Matrix Multiplication	45
4.4.4	Quicksort	46
4.4.5	Mergesort	48
4.5	Common Design Considerations	49
4.5.1	Program Reinitialization	49
4.5.2	Granularity	50
4.5.3	Nested Parallelism	53
Chapter 5: Scheduler Designs		54
5.1	Scheduler Hierarchy	55
5.1.1	Free Task Scheduler	56
5.1.2	Affinity Task Scheduler	56
5.1.3	Contiguous vs. Dispersed Allocation	57
5.2	Multiprogramming Schemes	58
5.2.1	Oversubscribed Free Multiprogramming	59
5.2.2	Oversubscribed Pinned Multiprogramming	61
5.2.3	Shared-Scheduler Multiprogramming	62
5.2.4	Default-Scheduler Multiprogramming	63
5.3	Multiprogramming Task Schedulers	64
5.3.1	Scheduler Partitioning	64
5.3.2	Scalability-Based Partitioning	66
5.3.3	Dynamic Repartitioning	68
5.3.4	Task-Availability-Based Repartitioning	72
5.4	Multiprogramming Test Strategies	76
5.4.1	Cyclic Tests	76
Chapter 6: Experimental Setup		78
6.1	Hardware Platform	78
6.2	Software Platform	79
6.3	Statistical Methods	79
Chapter 7: Experimental Results and Analysis		81
7.1	Granularity	81
7.2	Scalability	84
7.2.1	Contiguous vs. Dispersed Allocation	86
7.3	D&C Phases	88
7.4	Symmetric Tests	90
7.5	Cyclic Tests	92
7.5.1	Full Program Suite	92
7.5.2	Low- to Moderate-Scalability Subset	96
7.5.3	Comparison with Related Work	99

Chapter 8: Conclusion	102
8.1 Achievements.....	102
8.2 Potential Improvements and Future Work.....	103
8.3 Closing Remarks	105
Bibliography	106

Tables of Figures

Figure 1.1. Concurrent execution of two divide-and-conquer programs.....	2
Figure 1.2. Common symbols	4
Figure 2.1. Intel microprocessor trends	8
Figure 2.2. Multithreading on multiprocessors	10
Figure 2.3. Stack of parallelism abstractions	12
Figure 2.4. Activity flow for the D&C skeleton	14
Figure 2.5. Processor partitioning by program.....	16
Figure 2.6. Processor partitioning by program scalability	17
Figure 3.1. Task scheduling in Skandium	22
Figure 3.2. Global queue and per-thread local queues in TPL.....	23
Figure 3.3. Class diagram showing the composition of the task schedulers	25
Figure 3.4. Structure of the work-stealing task scheduler	26
Figure 3.5. Task production in the work-stealing task scheduler	27
Figure 3.6. Task consumption in the work-stealing task scheduler	27
Figure 4.1. D&C skeleton class basics	29
Figure 4.2. Three-level divide-and-conquer execution flow	30
Figure 4.3. Simple parallel quicksort, expressed declaratively.....	32
Figure 4.4. Parallel quicksort using fork-join pattern	33
Figure 4.5. Task execution in D&C skeleton using fork-join parallelism.....	34
Figure 4.6. Blocking vs. asynchronous execution of D&C recursion.....	36
Figure 4.7. Task execution in D&C skeleton using asynchronous parallelism	37
Figure 4.8. Unravelling nested asynchronicity at each recursive step	38
Figure 4.9. Class hierarchy for the D&C skeleton and programs.....	39
Figure 4.10. Map-reduce methods for functional parameterization.....	40
Figure 4.11. Colour convention for the D&C diagrams	41
Figure 4.12. Sample D&C execution of sum-of-squares	42
Figure 4.13. Sample D&C execution of Monte Carlo Pi.....	44
Figure 4.14. Sample D&C execution of Strassen matrix multiplication.....	46
Figure 4.15. Sample D&C execution of quicksort	47
Figure 4.16. Sample D&C execution of mergesort	48
Figure 4.17. Main components of the GranularDivCon class	50
Figure 4.18. Delimiting the split phase	52
Figure 5.1. Class diagram showing the composition of our task schedulers.....	55
Figure 5.2. Structure of the affinity task scheduler	57
Figure 5.3. Contiguous vs. dispersed allocation of 8 pinned threads.....	58
Figure 5.4. Oversubscription by multithreaded applications	59
Figure 5.5. Oversubscription by task-parallel programs.....	60
Figure 5.6. Oversubscribed free multiprogramming	60
Figure 5.7. Oversubscribed pinned multiprogramming	61

Figure 5.8. Programs feeding a shared task scheduler	62
Figure 5.9. Shared-scheduler multiprogramming scheme	63
Figure 5.10. Reconciliation of task parallelism with processor partitioning	64
Figure 5.11. Processors partitioned among task schedulers	65
Figure 5.12. Task schedulers can enact processor partitioning.....	66
Figure 5.13. Statically-partitioned multiprogramming	67
Figure 5.14. Thread-based processor reallocation	69
Figure 5.15. Decoupling of thread pool from task queue superstructure.....	70
Figure 5.16. Constitution of the multiprogram scheduler.....	70
Figure 5.17. Transitioning a worker thread across task schedulers	71
Figure 5.18. D&C execution under static partitioning	73
Figure 5.19. D&C execution under task-availability-based repartitioning	73
Figure 5.20. Heuristic for reallocating processors among programs.....	74
Figure 5.21. Cyclic test harness	76
Figure 6.1. Processor setup in the manycore servers.....	78
Figure 7.1. Execution time vs. granularity	82
Figure 7.2. Execution time vs. granularity, showing region of best performances	83
Figure 7.3. Speedup vs. concurrency	85
Figure 7.4. Speedup vs. concurrency for contiguous vs. dispersed allocation	87
Figure 7.5. Cumulative execution times of tasks in each D&C phase.....	89
Figure 7.6. MNTT for symmetric tests involving 8 instances of the same program.....	90
Figure 7.7. MNTT for symmetric tests of low-scalability programs.....	92
Figure 7.8. NTTs of our 5-program workload	93
Figure 7.9. NPs of our 5-program workload	93
Figure 7.10. ANTT of our 5-program workload	95
Figure 7.11. STP of our 5-program workload	95
Figure 7.12. NTTs of our 3-program workload	97
Figure 7.13. NPs of our 3-program workload	97
Figure 7.14. ANTT of our 3-program workload	98
Figure 7.15. STP of our 3-program workload	98
Figure 7.16. Scalabilities of PARSEC benchmarks	100
Figure 8.1. Nested parallelism attained by embedding sub-skeletons	104

Chapter 1: Introduction

Parallel programming has been receiving a lot of attention since the microprocessor industry's shift to multiprocessing at the turn of the century, following the cessation of the "free lunch" of exponential performance improvements for sequential applications after hitting physical limits such as the power wall [1]–[3]. The *de facto* technique for harnessing parallelism has traditionally been threaded programming, despite its various shortcomings [4], [5]. Recent attention has been shifting towards higher-level parallel programming frameworks, principally building on the notion of tasks, with popular products including Intel TBB (2006) [6], OpenMP (2008) [7],¹ and Microsoft TPL (2010) [8].

The microprocessor industry is now on the verge of leaping from the multicore to the manycore era [3], [9], spurring initiatives such as the Intel [Many Integrated Core \(MIC\)](#) architecture [10] and [Single-Chip Cloud Computer \(SCC\)](#) project [11]. Consequently, multiprogram workload scheduling is re-emerging as another important topic of research. As the number of processors in commodity machines rises drastically, it will become increasingly commonplace to have multiple programs – each internally parallel – executing concurrently on the same machine. This is a consequence both of need and of supply: A typical user workstation needs to have many applications active concurrently, yet most of these would not individually have the algorithmic scalability to exploit the full concurrency of a manycore machine. Strategies for dynamically distributing the active workload in a manner that best exploits the machine's multiprocessing capabilities will become essential to achieve high performance [12]–[15].

1.1 Aim

The principal aim of this project is to devise and investigate scheduling strategies for optimizing the performance of task-parallel programs embodying divide-and-conquer algorithms, particularly when executing concurrently as a multiprogram workload on a manycore machine.

¹ Although OpenMP was first released in 1997, the concept of tasks was only introduced in OpenMP 3.0, released in 2008.

1.2 Hypothesis

The divide-and-conquer algorithmic pattern inherently captures information about the computational characteristics of its implemented applications. Due to its well-defined formulation as a recursion, it follows a structured execution whose constitution may be accessed and controlled by the underlying scheduler.

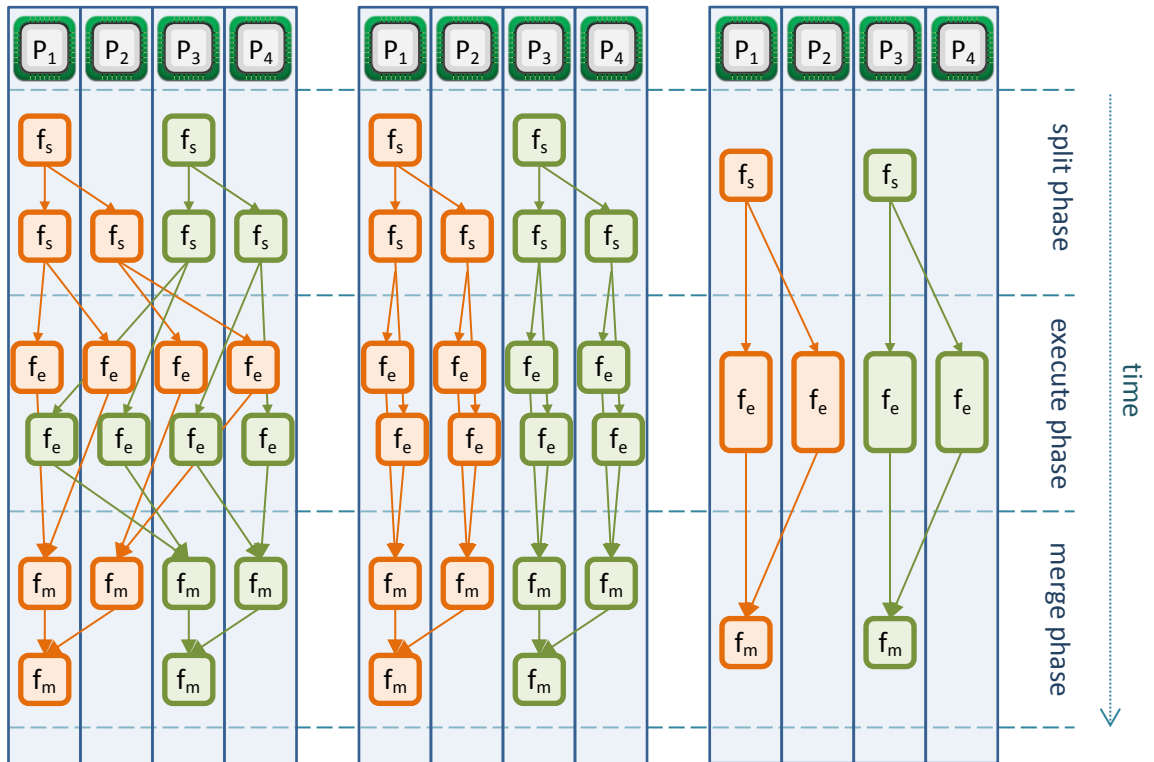


Figure 1.1. Concurrent execution of two divide-and-conquer programs, providing a preview of the nature of our scheduling strategies. Note that f_s represents a split operation; f_e , an execute (being the base case of the recursion); and f_m , a merge. The programs may be scheduled as: spread across all processors (left); allocated to dedicated subsets of the processors (centre); and having their granularities adjusted dynamically for their current allocations (right).

A pattern-aware scheduler may exploit this structural rigour to optimize applications' amenability to parallelism (as hinted in Figure 1.1). By inferring knowledge about each application's scalability and data locality, the scheduler can guide their decomposition into parallel subtasks *and* improve the subtasks' distribution across the system's multiprocessors, boosting their concurrent execution within the context of the multiprogram workload.

Our hypothesis is that such schedulers can achieve better system-level performances than traditional ones (including the default thread schedulers present in mainstream operating systems), especially for multiprogram workloads having variable scalabilities.

1.3 Contributions

The main research contributions of our project are as follows:

- We create a fresh implementation of the divide-and-conquer algorithmic skeleton for the .NET Framework, demonstrating the power of parallelism abstractions by layering our skeleton atop the task infrastructure provided by the Task Parallel Library (TPL) in .NET.
- We develop a novel extension to the work-stealing task-scheduling algorithm for handling multiprogram workloads. Our multiprogram scheduler employs a common pool of pinned worker threads that service a collection of program-specific task queue superstructures. This scheduler will serve as the mechanism through which we perform processor partitioning.
- We adapt the work of Sasaki et al. [13] and use program scalability as the base policy for driving our processor partitioning decisions. On top of this, we enact a dynamic reallocation strategy that regularly polls the number of available tasks in each program and adjusts processor allocations accordingly, with the goal of promoting full utilization of the machine's processors.
- Through our multiprogram scheduler, we investigate the potential for synergy between work-stealing task-scheduling and scalability-aware processor-partitioning, and report on the performance obtained for a sample multiprogram workload. Our experiments show that our scheduler can outperform the Linux thread scheduler by 2% for low- to moderate-scalability workloads.

1.4 Common Symbols

Figure 1.2 presents the common set of symbols that will be used in the diagrams we provide to complement our discussions throughout this dissertation.


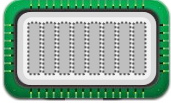





	²	processor ³
		multicore or manycore processor
		thread
	⁴	thread pinning
	⁵	program instance
	⁶	sequential code snippet
		task queue

Figure 1.2. Common symbols used throughout this document

Our diagrams also make extensive use of colour coding, whose significance can typically be inferred from context. In most cases, colours are used to differentiate either among programs (like in Figure 1.1) or among execution phases (like throughout Section 4.4, p. 41).

² Source: [CPU Icon](#) by [Chris Banks](#)

³ A “processor” should always be assumed to correspond to a single logical core, rather than an entire multicore or manycore chip, unless using the other symbol below. A logical core constitutes the smallest hardware component in a central processing unit (CPU) that can support thread-level parallelism.

⁴ Source: [Pin Icon](#) by [Anna Shlyapnikova](#)

⁵ Source: [App X Executable Icon](#) by [Untergunter](#)

⁶ Source: [Source Code Icon](#) by [Fatcow Web Hosting](#)

1.5 Document Structure

This dissertation has been intentionally structured to interleave two orthogonal streams: the parallelism abstraction offered by tasks and algorithmic skeletons; and the processor partitioning of multiprogram workloads. The reconciliation of these two aspects is one of our central contributions (and will be finally brought to fruition in Section 5.3, p. 64).

Chapter 1 introduced our project by setting out the context that motivated our investigation. It subsequently established our aim and the hypothesis that we shall be evaluating, followed by an overview of the main contributions of our system.

Chapter 2 visits the background pertinent to our research, including: the recent proliferation of multiprocessing; the various parallelism abstractions available to application developers as a front-end for harnessing it; the thread-scheduling algorithms that traditionally served as the low-level back-end; and performance metrics suitable for parallel systems. Chapter 3 explores some published work that is related to – and serves as the foundation for – our research. Given that our focus is on task parallelism, we analyse the designs of the task schedulers underlying Skandium and TPL. We then examine a recently-published processor-partitioning scheduler, giving a taste of this alternate direction of parallel optimization.

Chapter 4 presents the design of our divide-and-conquer skeleton and the parallel programs that we build upon it, offering insights into their dynamic behaviour. Chapter 5 delves into our scheduler designs, first explaining how we enact processor affinity, then defining some multiprogramming schemes for executing task-parallel programs concurrently. Subsequently, it presents our novel multiprogram scheduler, which synergizes the relationship between task scheduling and processor partitioning.

Chapter 6 briefly describes our experimental setup, outlining our hardware platform, software platform, and the statistical methods we employ. Chapter 7 is dedicated to our experimental results and analysis, covering aspects such as task granularity, hardware concurrency, program scalability, and multiprogramming schemes. We compare our results to related work and justify any differences.

Finally, Chapter 8 serves as the conclusion which closes up our discussion, summarizing our achievements and potential improvements. It is followed by the Bibliography, which lists all the publications cited throughout this dissertation.

1.5.1 Submission Details

The original proposal for this project was nominated by Murray Cole as "[Scheduling Multithreaded Java Programs in a Multiprogram Workload](#)". Some content in this dissertation has been adapted from our own Informatics Research Review (IRR) and [Informatics Research Proposal \(IRP\)](#) submissions, albeit largely re-narrated or significantly expanded.

This dissertation is complemented by a Visual Studio solution comprising all our source code, including the D&C skeleton, sample programs, schedulers, optimizers, multiprogramming schemes, and test harnesses. We also provide the raw data gathered from our final experiments, as well as a spreadsheet containing the aggregated results in tabular format, alongside the associated charts used in this document.

The submitted paper copies of this dissertation were accompanied by a signed "project copyright permission letter" and "own work declaration" form.

For updates and corrections to this document effected after submission, one may refer to our [DivCon](#) webpage,⁷ or contact us by email on karl.fenech@gmail.com. The current version of this document corresponds to revision 65.

⁷ URL: <http://dogmamix.com/DivCon/>

Chapter 2: Background

Our background spans a number of interrelated areas. We first discuss how the proliferation of multiprocessing is driving application developers to embrace parallel programming. This leads us to explore the levels of abstraction at which such parallelism may be expressed, culminating with structured parallel programming through algorithmic skeletons. This corresponds to the front end of our research, since it represents the interface exposed to the application developers. Next, we study the function of thread scheduling, which constitutes the back end of our investigation, since it provides the rudimentary mechanism through which parallelism is actualized. Finally, we present some performance metrics for evaluating parallel systems, and justify our choice of the .NET Framework as our development platform.

2.1 The Concurrency Revolution

For several decades, software developers have readily benefitted from the effect of Moore's Law [16] on microprocessor architectures. Exponential improvements in transistor densities regularly gave rise to increased clock speeds, instruction-level parallelism (ILP), and on-chip cache capacities [1], [17]. These hardware improvements translated into inherent performance gains for applications, including sequential ones, without requiring any alteration to their software implementations [2]–[4], [18].

However, the convenient trend of ever-increasing clock frequencies eventually ran into physical limits imposed by the “power wall”, encountering issues with power consumption, heat dissipation, and current leakage [1], [2], [17], [19] (see Figure 2.1 below). Similarly, ILP advances have been exhibiting diminishing returns, requiring complex architectural improvements for merely incremental performance gains [20], [21].

At the turn of the century, mainstream microprocessor manufacturers switched to thread-level parallelism (TLP) as the primary means of sustaining performance improvements [1], [3]. In the field of central processing units (CPUs), two hardware technologies flourished [12], [22]: simultaneous multithreading (SMT), which permits multiple threads to be executed simultaneously on a superscalar architecture by replicating the registers and program counters [23]; and chip multiprocessing (CMP), which integrates multiple cores,

each capable of independent execution, onto the same chip [17], [20]. Today, CMP (popularly known as “multicore”) has become ubiquitous [3], with client architectures supporting hundreds of cores [15] expected by 2020 [3].

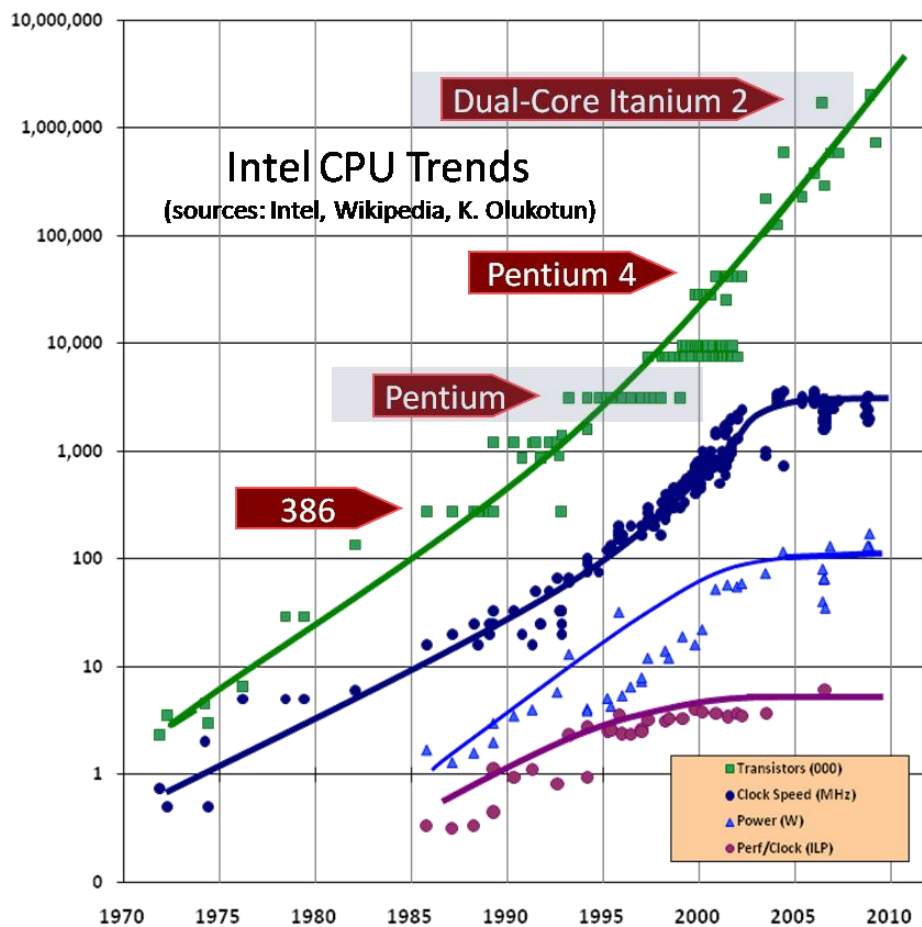


Figure 2.1. Intel microprocessor trends, showing that increasing transistor densities have ceased giving higher clock speeds since 2005. Copied from Sutter [1].

As a consequence of the proliferation of thread-level parallelism in modern microprocessors, applications seeking to fully exploit their capabilities will need to be concurrent [1], [2]. Notwithstanding hopes for a “quantum leap” in compilers’ abilities at automatic parallelization [24], such technology has not yet made its way into mainstream software development [2], [4], [5]. Thus, it is becoming increasingly important for software developers to take the initiative in parallelizing their applications [3], [4], [25].

“Concurrency is the next major revolution in how we write software.”

— Sutter [1]

Despite the hardware imperative, parallel programming has experienced disappointingly slow uptake in mainstream application development [3], [4]. The major obstacle hindering its adoption is the conceptual difficulty that most developers experience when reasoning about concurrency [1]–[3], [5], [26], often finding themselves “quickly overwhelmed” [4]. The situation is exacerbated by the inadequacy of most current languages and tools at expressing and harnessing parallelism [2]–[4]. In the next section, we shall explore how parallel abstractions influence this issue.

“[We need] higher-level abstractions that help build concurrent programs, just as object-oriented abstractions help build large componentized programs.”

— Sutter & Larus [4]

2.2 Parallel Abstractions

Cole [27] highlights two main aspects to parallelization: **problem decomposition**, being the identification of the potential parallelism in the expressed algorithm; and **distribution**, through which parallel candidates are mapped onto the available processors for concurrent execution. This distinction often manifests as an important abstraction presented by the software framework to the application developer, wherein the developer is expected to specify the parallel constructs (in some form), but then relegates the responsibility of their instantiation and concurrent execution to the runtime environment.

The degree of abstraction has strong implications on the roles played respectively by the developer and the runtime in harnessing parallelism, with lower-level frameworks affording more flexibility but less insulation from performance-critical factors, such as communication, synchronization, granularity, and load-balancing, as well as greater exposure to parallelism’s hazards, including nondeterminism, races, deadlock, livelock, and starvation [2]–[4].

“The aggressive goal of the parallel revolution is to make it as easy to write programs that are as efficient, portable, and correct (and that scale as the number of cores per microprocessor increases biennially) as it has been to write programs for sequential computers.”

— Asanovic et al. [2]

2.2.1 Threaded Programming

A thread is an execution primitive, spawned by a process, that may be independently managed by the operating system [28].⁸ Threaded programming, as exposed through implementations such as POSIX threads (pthreads) and Windows threads, is today's dominant paradigm for parallel programming [4], [5]. Multithreaded programs may harness the concurrency of multiprocessing systems by having their threads execute on distinct processors, with oversubscription handled transparently by the thread scheduler using pre-emptive time-slicing [28]. (Thread scheduling will be discussed in Section 2.3, p. 15.)

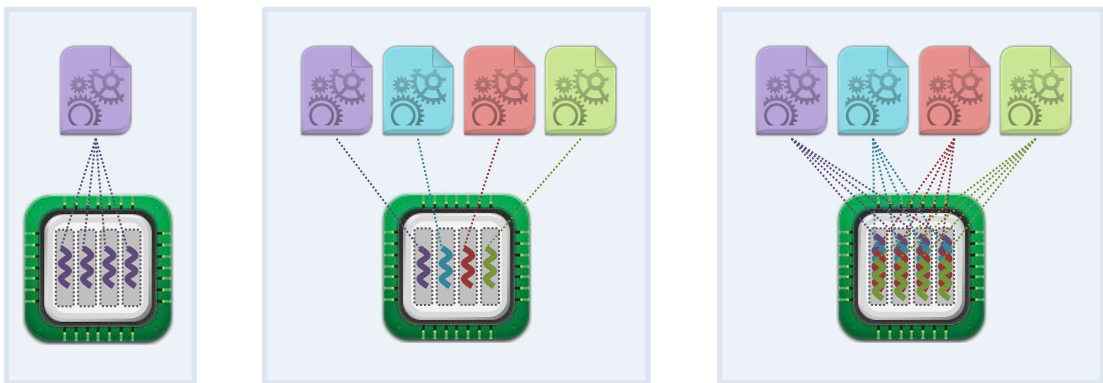


Figure 2.2. Multithreading on multiprocessors. Traditionally, most research has focussed either on individual multithreaded programs (left), or on workloads of multiple sequential programs (centre). However, multithreaded multiprogramming (right) is now receiving substantial attention.

The main appeal of the threaded programming paradigm is its shared memory abstraction, whereby all threads of the same process may be assumed to access a single logical address space, irrespective of which processor they are being executed upon. This makes threading appear like a “seemingly straightforward adaptation” of sequential programming, which could explain its present-day popularity [5]. The costs incurred from intercommunicating via main memory are mitigated through the presence of multiple levels of caches, which maintain consistency using cache coherence protocols [29].

Whilst suitable for embarrassingly parallel algorithms, threads impart too much responsibility on the application developer, who has to carefully synchronize their execution wherever they interact, including for shared-memory access [5]. The arbitrary

⁸ We shall be restricting our discussion to kernel-level threading. User threads, wherever mentioned or implied, should be assumed to follow a one-to-one mapping onto kernel threads, as is specifically the case for threads in the .NET Framework under its current implementation.

interleaving of threads across distinct code entities can cause leaky abstractions, resulting in spaghetti code that widens “the conceptual gap between the static program and the dynamic process” (analogous to the adverse effects of `goto` statements before the advent of structured programming) [30], [31].

“Threads, as a model of computation, are wildly nondeterministic, and the job of the programmer becomes one of pruning that nondeterminism. [...] Rather than pruning nondeterminism, we should build from essentially deterministic, composable components. Nondeterminism should be explicitly and judiciously introduced where needed, rather than removed where not needed.” — Lee [5]

Furthermore, threads are expensive to maintain. Their creation and destruction require costly system calls; similarly, context-switching between threads incurs overheads [32]. In order to prevent these issues from degenerating into a performance bottleneck, it becomes necessary to treat threads as long-lived entities, ideally reusable across multiple operations [33], which may possibly be fed from a higher level of abstraction.

2.2.2 Task Parallelism

Most modern programming frameworks introduce the notion of a thread pool, whereby the execution runtime assumes responsibility for maintaining a persistent set of reusable threads [33]. This way, application developers are insulated from explicit thread lifecycle management, and may instead focus on expressing potential parallelism through a series of tasks, to be picked up and executed by the thread pool [8]. Tasks, which represent a “finite CPU-bound computation”, are typically implemented as user-space constructs, making them significantly more lightweight than threads, thereby allowing for parallelism to be captured at finer granularities [8]. Runnable tasks are scheduled for execution through a task queue, which is typically serviced by a thread pool [8], [25], [34]. (See Section 3.1, p. 21, for a discussion of task scheduler implementations.)

Campbell et al. [34] describe some high-level patterns for tasks. For scenarios involving static parallelism, tasks may be programmed using the traditional fork–join pattern, where one spawns a number of tasks and waits for them to collectively complete. Algorithms based on data-flow constraints may be better-served by the “futures” pattern, wherein each task serves as a placeholder for a data value that is yet to be computed [4]. Dynamic parallelism may be achieved by spawning nested tasks from executing tasks. Finally,

asynchronous programming is possible through continuations, which are tasks that may only be started once an antecedent task (or collection thereof) completes.

Collectively, the aforementioned patterns may be visualized as giving rise to a graph-structured flow of execution, whose directed edges represent control dependencies among the tasks. Whilst the patterns' flexibility permits task graphs to be composed arbitrarily, this requires careful design from the application developer. In some cases, it may be possible to raise the level of abstraction further by providing pre-composed task graphs through structured parallelism.

2.2.3 Structured Parallelism

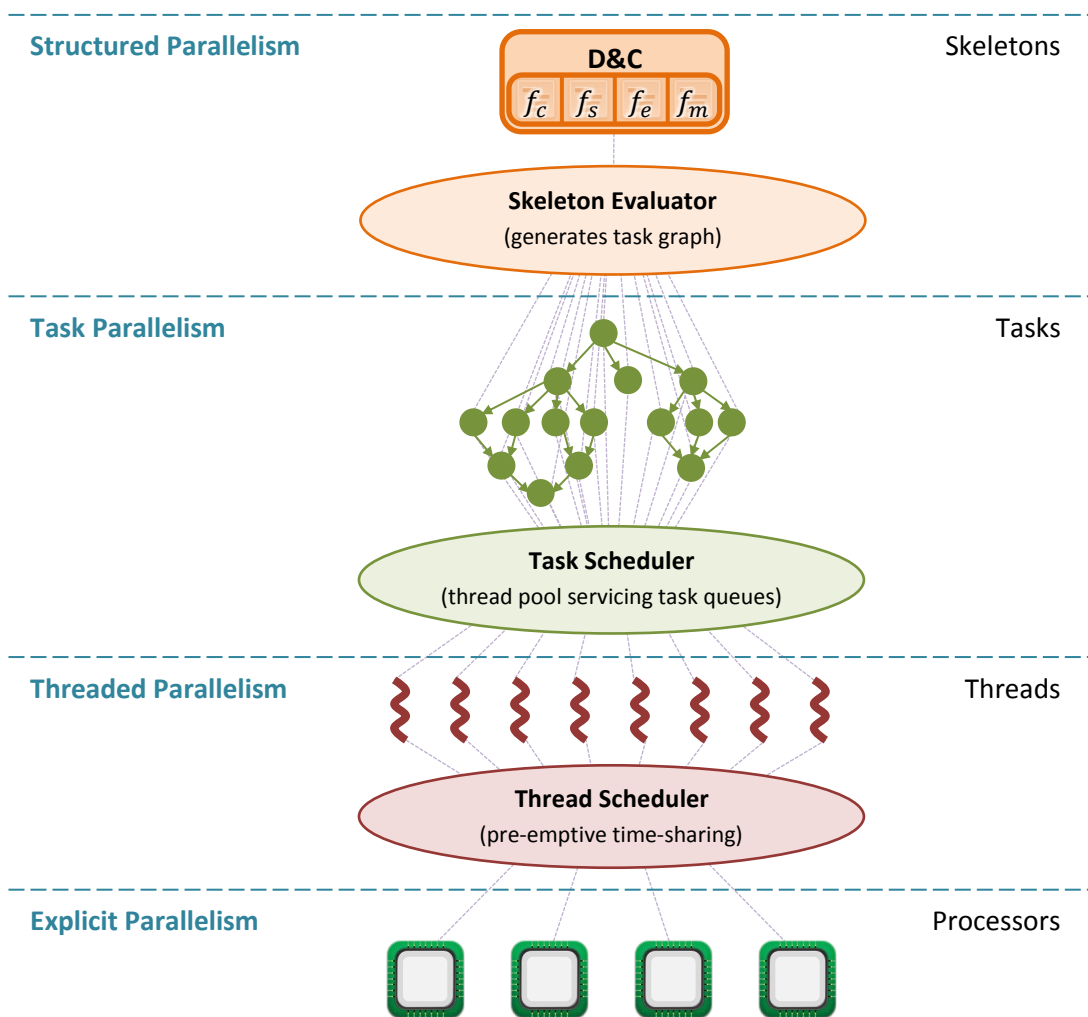


Figure 2.3. Stack of parallelism abstractions, demonstrating how each level builds on the one below it to simplify the developer's role in harnessing the machine's multiprocessing capabilities.

Structured parallel frameworks empower developers to express their algorithm as an instance of a parallel pattern from a supplied patterns library. These patterns would correspond to classes of algorithms that are popular candidates for parallelization [3], [35], such as loops of independent iterations, or decomposable problems that are solved recursively using a divide-and-conquer approach [25], [31]. By taking advantage of the natural boundaries intrinsic to the algorithm, structured parallelism captures the developer's algorithmic *intent* [4], rather than a specific tortuous implementation.

Structured parallelism insulates application developers from the responsibility of defining parallel units of execution at any degree of granularity. Rather, developers would only need to specify how such parallelism may be extracted from the problem, leaving it up to the runtime to avail of this potential. Consequently, the runtime may apply heuristics to dynamically come up with a good strategy for splitting up the work into parallel chunks, whilst accounting for data locality and load balancing (as exemplified through the various loop scheduling schemes in OpenMP [36]).

Algorithmic skeletons present a common formalization of structured parallelism [37]. By building on a consistent collection of patterns, skeletal programming aims to “transcend architectural variations”, permitting portability across disparate architectures whilst maintaining performance through architecture-specific implementations of the patterns library [35].

2.2.4 Divide-and-Conquer Skeleton

The divide-and-conquer (D&C) skeleton provides the common algorithmic pattern underlying our investigation. The skeleton is functionally parameterized through: a condition function, f_c ; a split function, f_s ; an inner skeleton, Δ ; and a merge function, f_m [25]. In general, the inner skeleton could be defined as another skeleton type, permitting nested recursion [25]. However, for the scope of our experiments, we will assume that the inner skeleton will correspond to the execution of a muscle function, f_e , which represents the sequential operation serving as the base case of the divide-and-conquer recursion.

The semantics of the D&C skeleton are as follows [25], [27]: Given an initial problem, the condition function is applied to determine whether it should be split. If affirmative, the problem is divided into a collection of subproblems using the ‘split’ function, which are

then fed into the same D&C skeleton to be processed recursively. Once all the corresponding subresults have been computed, they are combined through the ‘merge’ function to give the result for that level. On the other hand, if the condition function declined to split, then the problem is fed into the ‘execute’ function, which directly computes its corresponding result. This is demonstrated diagrammatically in Figure 2.4.

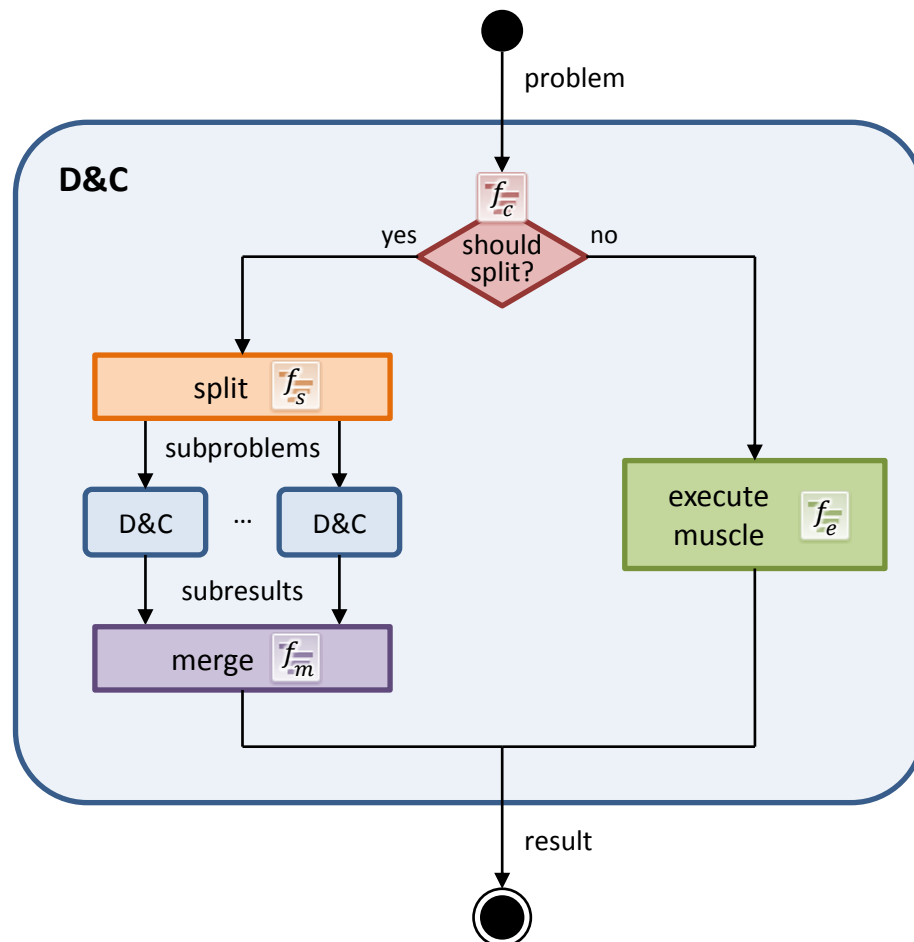


Figure 2.4. Activity flow for the D&C skeleton.

The self-referential definition of the D&C block underlies its recursive execution.

D&C’s potential parallelism arises from the concurrent evaluation of the independent subproblems at each level of the recursion [27], wherein each subproblem may cause a new task to be spawned. The number of subtasks produced by each invocation of the split function is known as the “branching factor”, and may be as small as two. However, each recursion level causes the degree of concurrency to increase exponentially – for example, after a recursion depth of ten, a binary D&C algorithm may have as many as 1024 tasks available for concurrent execution.

2.3 Thread Scheduling

General-purpose operating systems employ a thread scheduler for allocating available threads onto processors for execution. Such schedulers typically use time-slicing, whereby multiple threads may be given the illusion of concurrent execution on the same processor by being each run for short periods, known as quanta, in a round-robin manner [6], [28]. Once a thread's quantum expires, it is pre-empted, and the scheduler picks the next thread to be swapped in.

This approach promotes fairness, but also incurs overheads, since a context switch must be performed at the end of each quantum [28]. Each context switch needs to save the processor state of the current thread, and load the saved state of the next thread; this procedure takes several clock cycles. Furthermore, context switches impose indirect performance penalties due to the "perturbation of processor caches like the instruction, data, address translation, and branch-target buffers" [38]. Context switching among threads associated with different virtual address spaces requires the processor's translation lookaside buffer (TLB) to be invalidated; consequently, any caches tagged using virtual addresses would need to be flushed [38].

On symmetric multiprocessing architectures (such as SMT and CMP), the operating system can schedule a thread onto any processor. In such cases, the scheduler's goal is to achieve load-balancing by having the current workload distributed more or less evenly across all processors. At the same time, the scheduler also aims to promote affinity – a given thread should be kept running on the same processor for as long as possible, in order to improve its potential for cache reuse, and only migrated to another processor when there is a specific need to restore the load balance [12], [28]. Most systems also allow thread affinity to be controlled explicitly by the user, who may pin a specific thread to a subset of the processors.

2.3.1 Processor Partitioning

As the number of cores in commodity multiprocessing machines continues to rise, the inadequacy of mainstream schedulers for executing parallel workloads is becoming more pronounced [12]. By treating each thread as an independent unit of execution, schedulers forgo consideration of inter-thread dependencies that could have a significant impact on the throughput of the system. Specifically, collaborating threads that are tightly-

coordinated or share a lot of data would execute faster if scheduled onto the same core or chip [12].

Processor partitioning is a technique whereby specific threads are pinned to specific processors in a manner that promotes their inter-thread affinity [13]. Threads executing on the same physical core (through oversubscription or simultaneous multithreading) would share its private L1 and L2 caches; threads executing on distinct cores within the same physical chip share the L3 cache; whilst threads executing on distinct chips may only intercommunicate through main memory [39]. Thus, closely-related threads would benefit from constructive interference if allocated to proximate processors, since cache reuse is boosted by the locality of reference (both temporal and spatial) arising from their overlapping working sets [39].

Most parallel programs are configured to initialize at least as many threads as there are logical cores, so as to fully utilize the multiprocessor machine when run in isolation [12], [13], [34]. When running as part of a multiprogram workload, the programs themselves provide a natural boundary for processor partitioning [12], [40] (see Figure 2.5). By pinning each compute-intensive program's threads to run on a dedicated subset of the machine's cores, performance gains may be reaped due to the "high cache-hit ratio and low synchronization overhead" [40]. Additionally, the context-switching performance penalty is significantly reduced: Since the threads would be associated with the same virtual address space, the TLB and caches do not need to be flushed [38].

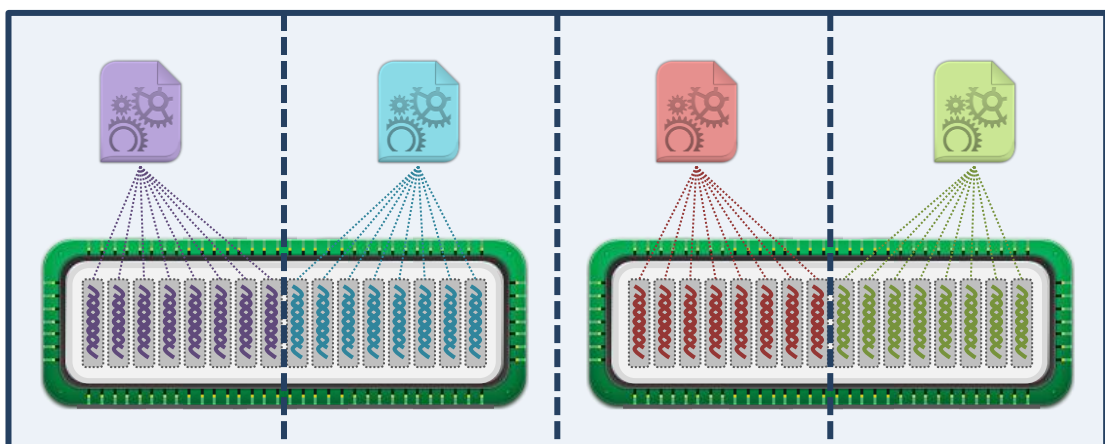


Figure 2.5. Processor partitioning by program. In a multiprogram context, threads belonging to the same program instance may execute faster when scheduled onto the same core or processor chip, rather than intermingled with the other programs' threads.

As cores become abundant, processor partitioning may also serve to provide “performance isolation and security between multiprogrammed applications” [3].

2.3.2 Program Scalability

Figure 2.5 (above) depicts an egalitarian partitioning strategy, where each program is allocated an equal number of cores. However, this strategy is suboptimal when the programs exhibit different scalabilities, since highly-scalable programs would (by definition) perform better when allocated large numbers of cores than poorly-scalable ones would – in fact, some programs cease to yield any speedup altogether beyond a certain degree of concurrency. For this reason, system throughput may be boosted by partitioning the processors among the programs based on their scalability [13], as exemplified in Figure 2.6 (below).

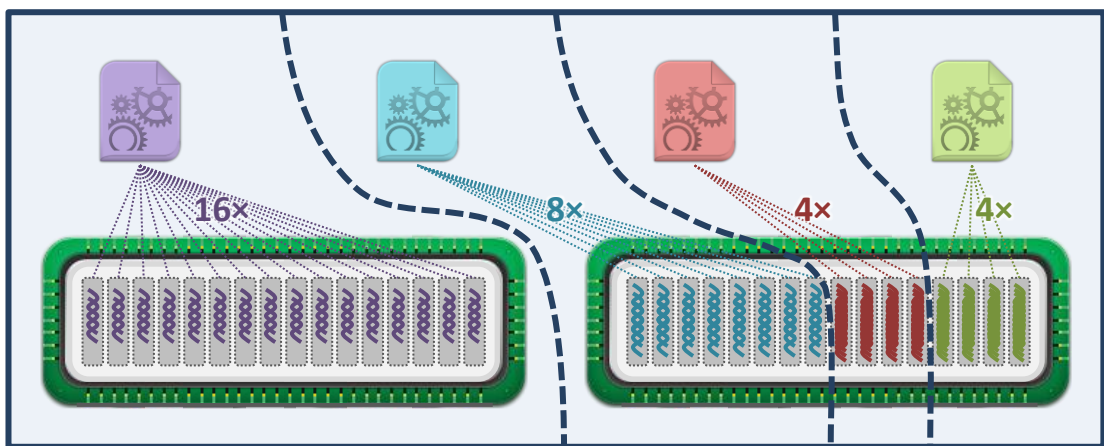


Figure 2.6. Processor partitioning by program scalability. In this example, the first program is twice as scalable as the second (for some notion of scalability), and is therefore allocated twice as many processors. Similarly for the third and fourth programs, which are only half as scalable as the second.

2.4 Performance Metrics

There is broad consensus that the preferred performance metric for an individual program, P , should be its execution time (i.e. wall-clock time), T [41]. The speedup of a program, S , is defined as the ratio of its sequential execution time on a single processor, T_{seq} , to its parallel execution time on the multiprocessing machine, T_{par} , and is the prevalent metric for evaluating parallel processing systems [42]. Efficiency, E , is the ratio of the speedup to the number of processors used [43].

Eyerman & Eeckhout [41] present various metrics for assessing system-level performance of multiprogram workloads. By executing a single program, P_i , in isolation, granting it exclusive access to all the machine's processors, one may obtain its performance in "single-program mode", C_i^{SP} . When executed concurrently with other programs, one would measure its "multiprogram mode" performance, C_i^{MP} .

The latter performance intuitively depends on the degree of multiprogramming and the nature of the other programs. However, even for a fixed multiprogram workload, the performance would vary depending on the implementation of the scheduler responsible for distributing the programs over the available cores. Thus, by testing different schedulers against a given workload, one can quantitatively measure and compare the efficacies of their scheduling strategies.

2.4.1 Turnaround Time

Eyerman & Eeckhout [41] promote normalized turnaround time, NTT_i , as the main user-oriented performance metric for each program, since it "quantifies the user-perceived turnaround time slowdown due to multiprogram execution".

$$NTT_i = \frac{C_i^{MP}}{C_i^{SP}}$$

This value may be averaged across all n executing programs to obtain a system-level measure, average normalized turnaround time ($ANTT$) [41]:

$$ANTT = \frac{1}{n} \sum_{i=1}^n \frac{C_i^{MP}}{C_i^{SP}}$$

$ANTT$, which is adopted by Sasaki et al. [13], is a lower-is-better metric. When the degree of multiprogramming is n , $ANTT$ is expected to vary between an ideal value of 1 (which indicates that there is no program interference or resource contention whatsoever) and an upper bound of n (which would be equivalent to executing the programs consecutively in isolation) [41].

A related system-level metric is maximum normalized turnaround time (*MNTT*), which is similarly defined [41]:

$$MNTT = \max_{i \in [1..n]} \left(\frac{C_i^{MP}}{C_i^{SP}} \right)$$

2.4.2 Throughput

Eyerman & Eeckhout [41] suggest that performance studies should also use a system-oriented metric such as throughput, which measures “the number of programs completed per unit of time” and, therefore, gives an indication of the rate at which each program progresses. Each program’s normalized progress, NP_i , is the reciprocal of its NTT_i :

$$NP_i = \frac{C_i^{SP}}{C_i^{MP}}$$

The system throughput, STP , may be obtained by summing the normalized progress of each program, giving a higher-is-better metric, that again ranges between 1 and n [41]:

$$STP = \sum_{i=1}^n \frac{C_i^{SP}}{C_i^{MP}}$$

2.4.3 Variability

Typical hardware architectures are nondeterministic at the clock-cycle level. Each processor chip, as well as main memory, is driven by its own local clock, which can give rise to minor timing fluctuations whenever they interact over buses [44]. Pre-emptive thread scheduling at the operating system level introduces further unpredictability, since a program may get interrupted at arbitrary points in time [45]. The effects of these factors are amplified in parallel scenarios, where differences in the outcomes of resource contention may lead to substantial performance variability [45]. Therefore, any performance measurements should be consolidated by running experiments repeatedly and applying statistical methods over the results, as discussed for our experiments in Section 6.3 (p. 79).

2.5 Parallel Programming in .NET Framework

We developed our implementation artefacts using the [.NET Framework](#), with [C#](#) as the programming language. As a high-level software development platform, .NET is similar in scope to Java, but has seen significant improvements in its support for parallel programming since .NET 4.0 (released in 2010), which introduced structured parallelism through the [Task Parallel Library \(TPL\)](#) and [Parallel LINQ \(PLINQ\)](#) [26], [34]. Furthermore, C# supports first-class [anonymous functions](#), including [lambda expressions](#), which are hailed as “necessary ingredients” for enabling the succinct expression of parallelism using a library-based approach in a strongly-typed language [8].

Chapter 3: Related Work

This chapter will be mostly dedicated to the designs underlying existing task schedulers, which serve as the foundation for our multiprogram scheduler. The last section describes the scalability-based manycore partitioning (SBMP) scheduler of Sasaki et al. [13], thereby exposing the orthogonal scheduling approach offered by processor partitioning.

3.1 Task Schedulers

Our investigation focuses on programs expressed as instantiations of the divide-and-conquer skeleton. In Section 2.2.4 (p. 13), we discussed how this skeleton could be unravelled to provide potential parallelism as a series of tasks. However, for these tasks to effectively utilize the concurrent capabilities of multiprocessing machines, they need to be serviced by a task scheduler that can map them onto the available processors. We shall now study the design of the task schedulers powering two parallel libraries: Skandium [25] and Microsoft Task Parallel Library (TPL) [8], [34]. (Other mainstream task schedulers are provided by Intel Threading Building Blocks (TBB) [6], [46] and OpenMP [7], but will not be discussed here due to space constraints.)

3.1.1 Skandium

Skandium [25] evaluates skeletons using a producer–consumer workflow, wherein each skeleton produces tasks to a shared task queue, known as the “ready queue”, whence they may be consumed by a pool of worker threads for execution. By default, the size of the thread pool is equal to the number of logical cores on the machine.⁹

Initially, each skeleton would enqueue the root task representing the top-level problem instance. As each task is being processed, it may dynamically spawn further subtasks; these are also added to the ready queue, pending execution. Subsequently, the parent task transitions into the waiting state, making it ineligible for scheduling. The procedure is repeated recursively according to the skeleton structure, until one arrives at the muscle functions, which, being sequential, are directly executed. Once all its subtasks have been

⁹ As explained in Section 2.1 (p. 7), simultaneous multithreading permits each physical core to execute multiple threads concurrently at the hardware level; these are called “logical cores”.

completed, the parent task is reinserted into the ready queue, so that it may subsequently collate the results of its subtasks and complete its execution [25].

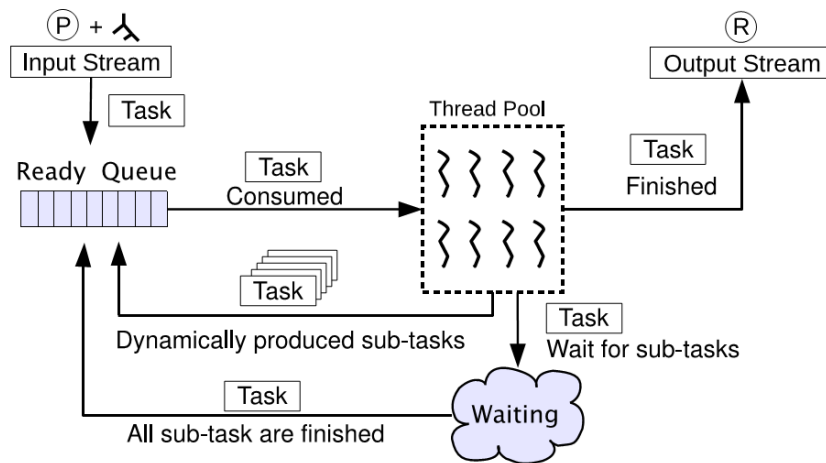


Figure 3.1. Task scheduling in Skandium, depicting how the pool of worker threads services the ready queue of pending tasks. Copied from Leyton & Piquer [25].

The use of a centralized queue in Skandium serves as a source of contention, since its access must be synchronized across all threads. This issue becomes more pronounced as the granularity of the tasks becomes smaller and/or the number of cores becomes larger, leading to a point where the synchronization overheads would offset any performance benefits from the parallel execution [34].

3.1.2 Task Parallel Library

Task Parallel Library (TPL), by default, uses a work-stealing task scheduler that is tightly integrated with the thread pool of the .NET Framework. This scheduler mitigates the bottleneck arising from Skandium's centralized queue by maintaining a dedicated task queue for each worker thread, with threads mainly fetching tasks from their own local queues. If its local queue becomes empty, a thread may fetch tasks from the global queue, which would contain the root tasks. (In our case, the root tasks would represent the top-level problem generated by each D&C program instance.) If the global queue is also empty, a thread may engage in work-stealing from another thread's queue [8], [34].

In order to promote data locality, worker threads add and remove tasks at the same end of their local queue in LIFO fashion. The rationale is that a recently-added task is more likely to share some of the same data as the last-executed task; thus, the data may be reused from

cache. On the other hand, thief threads steal tasks from the *opposite* end of the remote queue. This is particularly beneficial for divide-and-conquer algorithms, since it inherently promotes load-balancing by ensuring that the largest tasks (from higher levels in the recursion) are stolen first, thereby keeping the thief thread busy for a substantial period of time before its queue becomes empty again [8], [34].

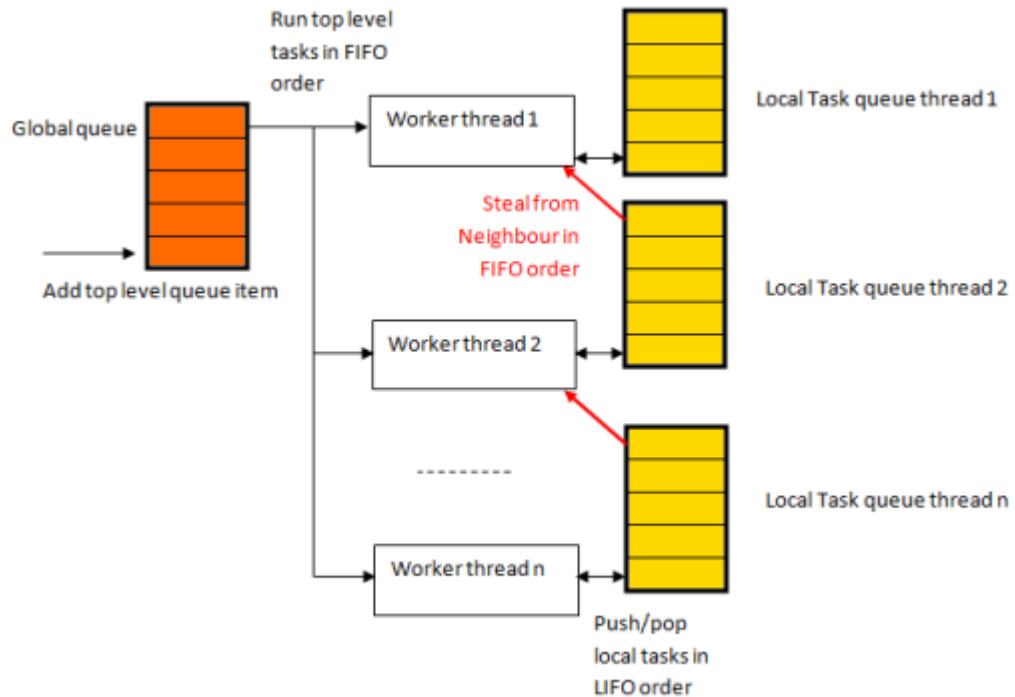


Figure 3.2. Global queue and per-thread local queues in TPL, showing LIFO pushing/popping of local tasks, and FIFO work-stealing of remote tasks. Copied from the “[Task Parallel Library](#)” article by Sacha Barber on CodeProject.

The internal implementation of the task queue uses lock-free execution to minimize the synchronization required for preventing race hazards when multiple threads contend to take tasks from the same queue. Specifically, the algorithms for pushing and popping tasks at the local queue avoid the need for acquiring an explicit lock in the majority of cases, only falling back to locking when there is a chance of being raced by a thief thread [8].

The .NET thread pool dynamically adjusts its number of worker threads using a hill-climbing heuristic that aims to maximize throughput. By design, once they start executing, tasks cannot be pre-empted from their worker thread; thus, a task that blocks (due to I/O or synchronization) would stall its processor. In such cases, thread injection is used to avoid underutilization and starvation [34]. However, since the heuristic does not distinguish

between blocked tasks and long-running compute-intensive operations, thread injection can result in counterproductive thread oversubscription.

TPL's default scheduler achieves high throughputs and improved scalabilities over large numbers of processors [8]. Leyton & Piquer [25] report a 3.4× speedup when using Skandium to run quicksort over 8 cores; Leijen et al. [8] push this up to 5.1× using TPL, registering an improvement of 50% (without accounting for experimental or implementational differences).

Internally, TPL implements this default scheduler within the `ThreadPoolTaskScheduler` class. Apart from its tight coupling with the thread pool, this class is declared as both internal and sealed, making it impossible to extend without reverse-engineering. Mono has a similar internal implementation within its `Scheduler` class.

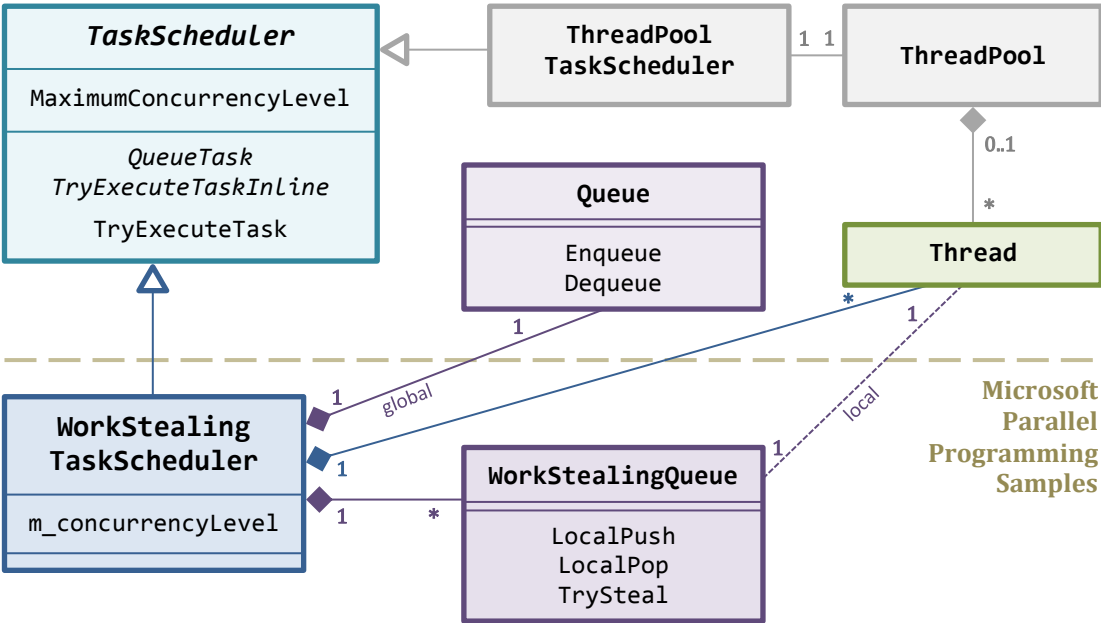
3.1.3 Work-Stealing Task Scheduler

In their “[Samples for Parallel Programming with the .NET Framework](#)” project, Microsoft provide the source code for a number of custom task schedulers. Among these, one finds the `WorkStealingTaskScheduler` class,¹⁰ which implements a work-stealing task scheduler whose function is quite similar to a simplified version of the TPL default scheduler. The main behavioural difference is that it does not employ thread injection, but abides by the fixed concurrency level specified to its constructor.

Figure 3.3 (below) shows how task schedulers derive from `TaskScheduler`, the abstract base class that serves as the extension point through which TPL allows custom task schedulers to hook into its task infrastructure [34]. The main abstract method that derived classes need to implement is `QueueTask`, for queuing a new task onto the concrete scheduler.¹¹

¹⁰ The implementation of this class is discussed in the “[Building a custom thread pool: a work stealing queue](#)” blog post by Joe Duffy.

¹¹ This method is never called directly from user code. Rather, it is called by the TPL infrastructure (under the hood) whenever the user creates a new task to be scheduled onto the specific scheduler, such as by calling a `TaskFactory.StartNew` overload that accepts a `TaskScheduler` parameter.



Microsoft
Parallel
Programming
Samples

Figure 3.3. Class diagram showing the composition of the task schedulers. The built-in `ThreadPoolTaskScheduler` and the sample `WorkStealingTaskScheduler` both derive from the `TaskScheduler` base class.

The structure of the `WorkStealingTaskScheduler` is shown in Figure 3.4 (below). It spawns a batch of worker threads to operate its scheduling logic, with the aim of efficiently executing any queued tasks. It initializes a `Queue<Task>` to serve as the global queue, as well as a `thread-local WorkStealingQueue<Task>` to be the local queue for each worker thread. Both of these data structures can expand their capacities dynamically to accommodate any number of tasks as required. The `LocalPush` and `LocalPop` methods of the `WorkStealingQueue` data structure internally employ low-lock techniques (using the `Interlocked` class for atomic operations) to reduce synchronization overheads, whilst all other accesses are protected as critical sections (using the `Monitor` class).

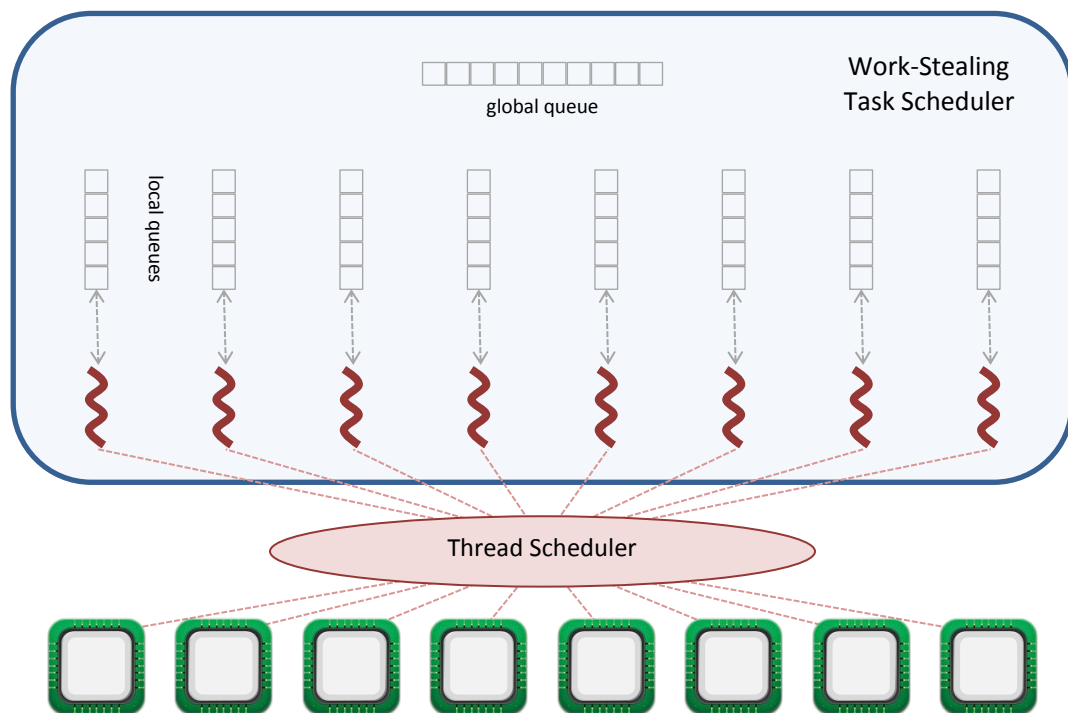


Figure 3.4. Structure of the work-stealing task scheduler, containing one global task queue and per-thread local queues. The number of worker threads typically corresponds to the number of logical cores.

`WorkStealingTaskScheduler` implements the inherited `QueueTask` method such that new tasks are pushed onto the local queue if originating from a worker thread, or enqueued to the global queue otherwise, as shown in Figure 3.5 (below).

Each worker thread primarily services its own local queue. Once this empties, it picks up new externally-introduced tasks from a global queue. If this also empties, it steals from other threads' queues, as depicted in Figure 3.6 (below). In any case, once a task is picked, it is executed through the inherited `TryExecuteTask` method of the base `TaskScheduler` class.

We shall be using the `WorkStealingTaskScheduler` class as the basis of our schedulers, with minor modifications to improve its extensibility for our requirements, as discussed from Section 5.1 (p. 55) onwards.

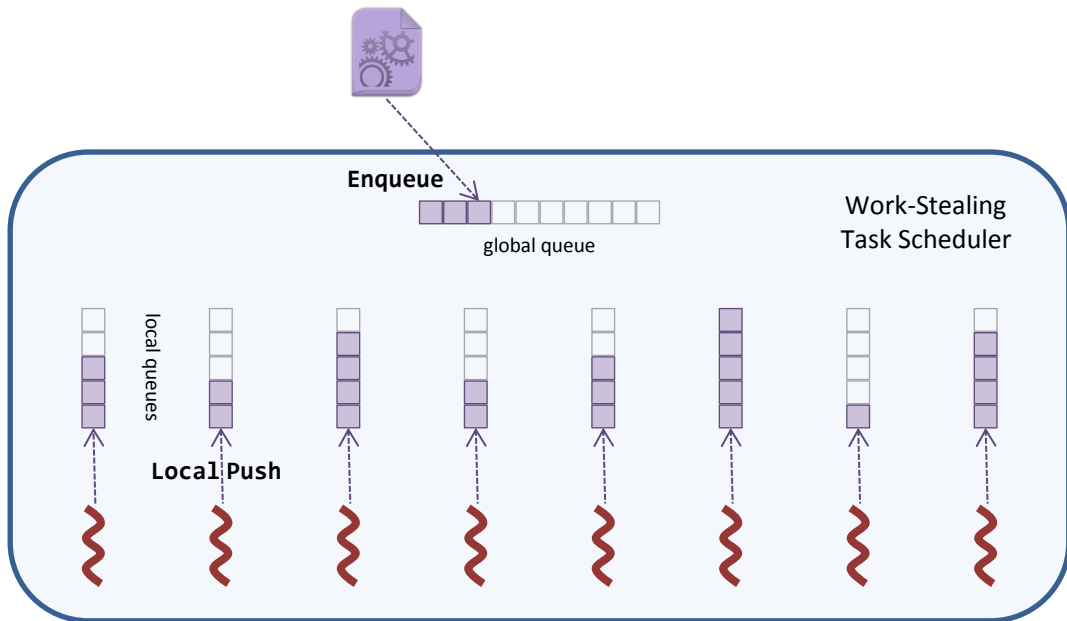


Figure 3.5. Task production in the work-stealing task scheduler. Worker threads push new tasks to their local queues, whilst external threads add to the global queue.

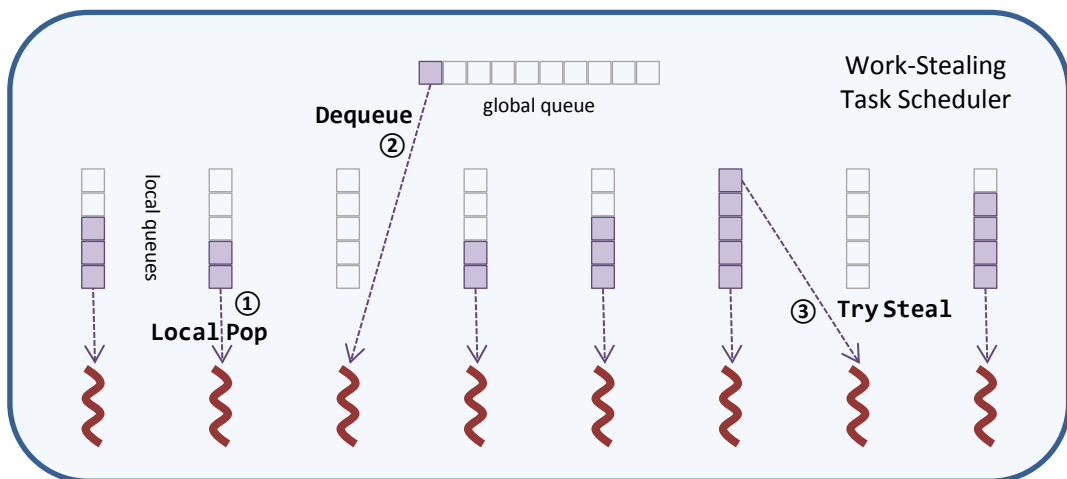


Figure 3.6. Task consumption in the work-stealing task scheduler. In order of preference, worker threads attempt to consume tasks from: ① their local queue; ② the global queue; ③ other threads' queues (work-stealing).

3.2 Partitioning Schedulers

Sasaki et al. [13] propose a scalability-based manycore partitioning (SBMP) scheduler, designed for multithreaded multiprogrammed workloads executing on manycore machines. The scheduler dynamically infers each program's scalability and thereby attempts to come up with an optimal core allocation that maximizes the system's performance according to some desired metric, such as average normalized turnaround time (ANTT). For each program, the scheduler maintains a scalability table that records its relative performances when executed on various numbers of cores. Program performance is measured by counting cumulative retired instructions per second (IPS), using performance monitoring units (PMUs) provided by the processors. This information is collated across all programs, and used to drive a hill-climbing algorithm for arriving at a global assignment decision.

The SBMP scheduler performs a repartitioning either when a program is created or terminated, or when a program's scalability is detected to have changed sufficiently to signify a different execution phase. Experimental results show that the SBMP scheduler can outperform the default Linux scheduler by 18% for single-phase programs, and by 8% for programs in general when phase prediction is enabled [13].

Chapter 4: Skeleton Designs

In this chapter, we present our design for the divide-and-conquer skeleton, first describing its interface and high-level structure, then delving into the execution pattern that underlies its implementation. Sections 4.3–4.5 will subsequently be dedicated to an exposition of the parallel programs we developed upon this skeleton, including a discussion of various design considerations that affect their performances.

4.1 Divide-and-Conquer Skeleton

4.1.1 Functional Parameterization

The divide-and-conquer (D&C) skeleton serves as the foundation for all our parallel programs in this investigation. Its algorithmic structure has already been discussed in Section 2.2.4 (p. 13); we shall now describe how we adapt it to the task infrastructure of the Microsoft Task Parallel Library (TPL).

At the root of our design, we define a base class, `DivConBase`, whose four abstract methods permit the functional parameterization of the D&C skeleton:

```
1 public abstract partial class DivConBase<TParam, TResult>
2 {
3     protected abstract bool ShouldSplit(TParam problem, int level);
4     protected abstract TParam[] Split(TParam problem, int level);
5     protected abstract TResult ExecuteMuscle(TParam problem, int level);
6     protected abstract TResult Merge(TResult[] subResults, int level);
7
8     public Task<TResult> Input(TParam problem) { /* ... */ }
9 }
```

Figure 4.1. D&C skeleton class basics, with type signatures adapted from Skandium [25]. Abstract methods are to be overridden by concrete implementations of D&C programs.

Leijen et al. [8] extol parametric polymorphism ([generics](#)) as a “necessary ingredient” for expressing structured parallelism in strongly-typed languages. We employ two type parameters: `TParam` for the parameter type that will represent our problem (and subproblem) instances, and `TResult` for the result (and subresults).

4.1.2 Execution Interface

As shown in Figure 4.1 (above), the only functionality exposed publicly for consumers is the `Input` method, which represents an asynchronous invocation of the D&C skeleton on the specified top-level problem instance. The returned `Task<TResult>` instance is a “future”, from which the final result may eventually be obtained, once the computation completes, through its `Result` property. Accessing this property before the task has completed would cause the current thread to block, similar to calling `wait` under the fork-join pattern.

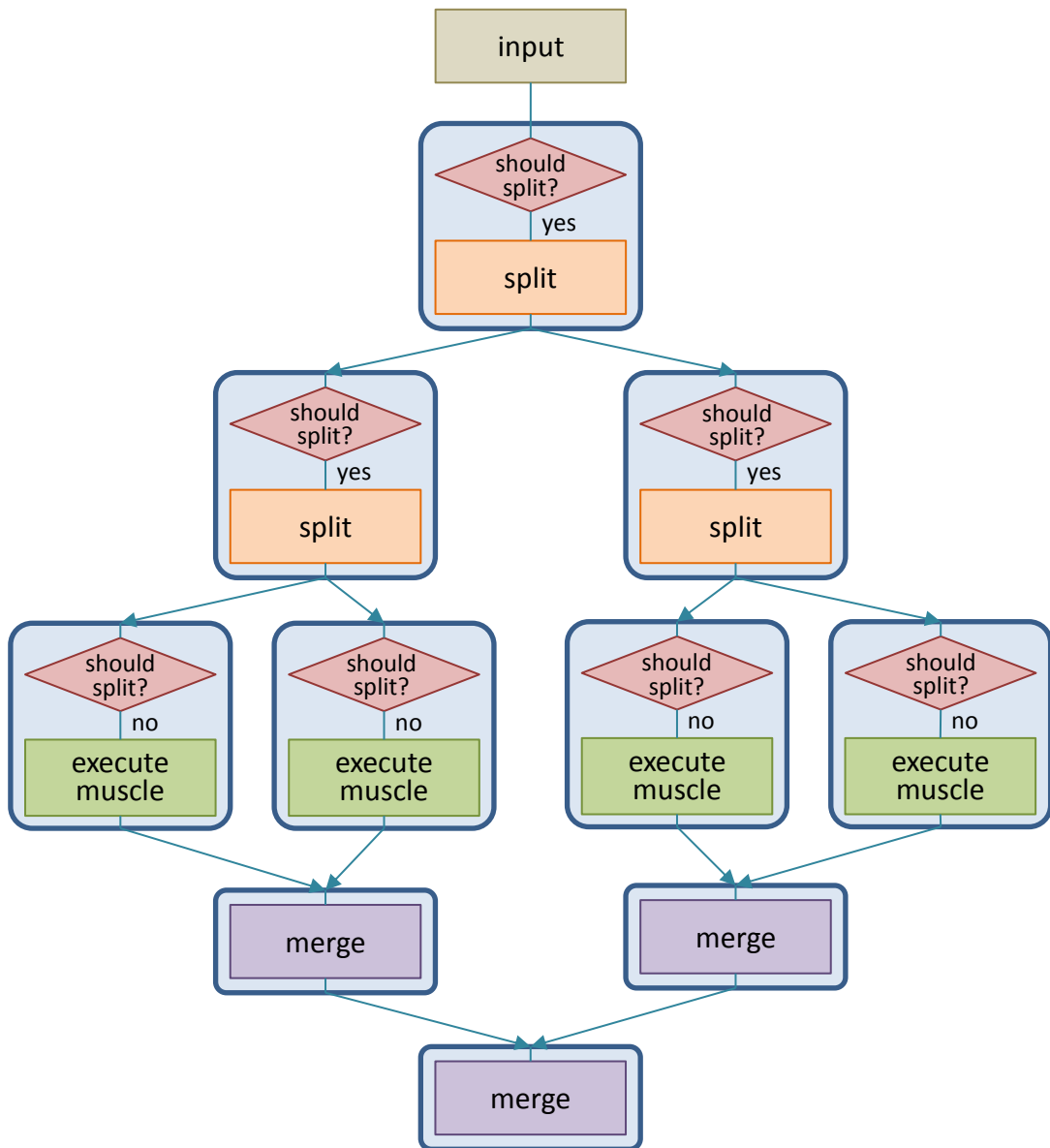


Figure 4.2. Three-level divide-and-conquer execution flow

Under the hood, `Input` dynamically creates a task graph, where each split spawns further subtasks, as exemplified in Figure 4.2 (above). Conceptually, the `Task<TResult>` future returned from the method call would be equivalent to the final merge. The details of this behaviour will be discussed in Section 4.2 (p. 33).

The performance implications of the rhombus-like shape of this task graph are, in effect, a manifestation of (an extended version of) Amdahl's Law [47]. A D&C program gradually transitions from sequential to parallel (during splitting) and back (during merging). Specifically, for a branching factor b , a program would have b^{i-1} tasks at level i of the recursion.

4.1.3 Scheduling Extensibility

One of the strongest design features of TPL is its separation of the task abstraction from the task scheduler, permitting the various parallel patterns defined in Section 2.2.2 (p. 11) to be executed using custom task schedulers as described in Section 3.1.3 (p. 24). Most task-creation facilities, such as `TaskFactory.StartNew` (for spawning root tasks or subtasks) and `Task.ContinueWith` (for registering continuations), accept a `TaskScheduler` parameter, through which custom schedulers may be specified [34].

We exploit this architectural extensibility by designing our D&C skeleton to accept a `TaskScheduler` instance as one of its parameter, which it will subsequently use for scheduling all the tasks it creates during its execution.

4.1.4 Declarative Model

C# supports first-class anonymous functions, such as [lambda expressions](#), which permit parallel constructs to be expressed succinctly [8]. This functional style of programming may be blended with structured parallelism in imperative languages to reap the “rich sources of concurrency” exposed through the higher-order functions [4], such as is done in [PLINQ](#) for exploiting data parallelism. This way, one may approach the abstractional power of declarative systems, where developers can focus on the problem-domain logic, whilst remaining agnostic to the control flow underlying the program execution and, therefore, largely unaware of any notion of sequential or parallel execution [27].

To demonstrate this concept, we provide a declarative interface to our D&C skeleton, which can be consumed as shown in the simplified parallel quicksort implementation below:¹²

```
1  var quickSort = CustomDivConFactory.Create<int[], int[]>(
2      taskScheduler: TaskScheduler.Default,
3      shouldSplit: (nums, level) => nums.Length > 1024 && level < 10,
4      split: nums => new [] {
5          nums.Where(n => n <  nums[0]),
6          nums.Where(n => n >= nums[0]) },
7      executeMuscle: nums => { Array.Sort(nums); },
8      merge: results => results[0].Concat(results[1]));
9
10 int[] source = new [] { 8, 2, 9, 1, 0, 4, 15, 3, /* ... */ };
11 int[] sorted = quickSort.Input(source).Result;
```

Figure 4.3. Simple parallel quicksort, expressed declaratively

Figure 4.3 testifies to the expressive power of structured parallelism. When combined with [LINQ](#), it permits a parallel (albeit inefficient) implementation of the quicksort algorithm to be fully expressed in just eight lines. (By contrast, a subclassing-based implementation of parallel quicksort in Skandium requires four classes spanning 26 lines of code [25].) Furthermore, it is trivially easy to also parallelize the split logic just by slapping an [AsParallel](#) decorator call onto the arrays, thereby achieving nested parallelism.

4.1.5 Object-Oriented Model

Notwithstanding its succinctness, the inefficiencies arising from the code in Figure 4.3 (above) betray the issues that may arise from such a functional expression. LINQ is designed to honour referential transparency (like in functional languages); thus, our declarative skeleton causes a new array to be created behind the scenes for each [Where](#) and [Concat](#) call (at each level of recursion), leading to huge memory demands.

Given that we are targeting shared-memory multiprocessors, we can design our D&C skeleton to assume statefulness, thereby taking advantage of the design benefits of object-oriented programming (OOP). For sorting algorithms, we can discard the notion of referential transparency and permit in-place implementations, using encapsulation to

¹² We assume that the reader is familiar with the parallel quicksort algorithm. For a quick overview of how it may be implemented using the D&C skeleton, refer to Section 4.4.4 (p. 46). Note that the implementation in Figure 4.3 (above) simplistically picks the first element, `nums[0]`, as the pivot value; in practice, this can lead to highly-unbalanced splits, and is discouraged.

maintain a single instance of the elements array internally. Similarly, we employ inheritance and polymorphism to structure our programs as a class hierarchy, with common functionalities implemented in base classes and overridden as necessary in subclasses, as discussed in Section 4.3 (p. 39).

4.2 Task Generation

We shall now explain how the D&C skeleton dynamically generates the task graph portrayed in Figure 4.2 (p. 30) along its recursive execution. We present two possible design patterns, and discuss their comparative performances.

4.2.1 Fork-Join Pattern

Both Leijen et al. [8] and Campbell et al. [34] advocate a fork-join pattern for parallel divide-and-conquer algorithms. In Figure 4.4, the recursive case of the parallel quicksort implementation uses `Parallel.Do`¹³ so that the two subranges can be sorted in parallel.

```
1  static void ParQuickSort<T>(T[] dom, int lo, int hi)
2      where T : IComparable<T>
3  {
4      if (hi - lo <= Threshold)
5          InsertionSort(dom, lo, hi);
6
7      int pivot = Partition(dom, lo, hi);
8      Parallel.Do(
9          delegate { ParQuickSort(dom, lo, pivot - 1); },
10         delegate { ParQuickSort(dom, pivot + 1, hi); }
11     );
12 }
```

Figure 4.4. Parallel quicksort using fork-join pattern. Copied from Leijen et al. [8].

We generalize this behaviour to be applicable across all D&C algorithms implemented using our skeleton; a condensed version of our implementation is given in Figure 4.5. In the recursive case, the parent task splits the problem into subproblems, spawns subtasks for processing each subproblem, collects their respective subresults, and finally merges them.

¹³ The `Parallel.Do` method has been renamed to `Parallel.Invoke` in the official release of TPL.

```

1  public Task<TResult> Input(TParam problem, int level = 1)
2  {
3      // Spawn a new task for processing the current problem.
4      return TaskFactory.StartNew(() => Process(problem, level));
5  }
6
7  protected virtual TResult Process(TParam problem, int level)
8  {
9      // Check whether to split the current problem.
10     if (ShouldSplit(problem, level))
11     {
12         // Split the current problem into subproblems.
13         TParam[] subProblems = Split(problem, level);
14
15         // Process each subproblem recursively (using new subtasks).
16         var subTasks = new Task<TResult>[subProblems.Length];
17         for (int i = 0; i < subProblems.Length; ++i)
18             subTasks[i] = Input(subProblems[i], level + 1);
19
20         // Collect the subresult from each subtask.
21         var subResults = new TResult[subProblems.Length];
22         for (int i = 0; i < subProblems.Length; ++i)
23             subResults[i] = subTasks[i].Result;
24
25         // Merge subresults to obtain the current result.
26         return Merge(subResults, level);
27     }
28     else
29     {
30         // Execute muscle to obtain result directly for current problem.
31         return ExecuteMuscle(problem, level);
32     }
33 }

```

Figure 4.5. Task execution in D&C skeleton using fork–join parallelism

Unfortunately, this approach has a fundamental design shortcoming. In both above implementations, each recursive step incurs a synchronization barrier (inherently in the `Parallel.Do` call or aggregately when reading the subtasks' `Result` property), causing the parent task to *block* (as if calling `WaitAll`) until all its subtasks complete. Since task scheduling is not pre-emptive, the blocked parent task does not relinquish control of the thread it is executing on, but holds on to it, keeping it unavailable for processing other tasks.

A possible workaround would be to process one of the subproblems directly within the parent task (thereby reducing the number of spawned subtasks by one at each recursive step). However, this introduces inconsistency into the design [34], and hinders the task scheduler's flexibility due to the parent task becoming extensively long-lived, preventing reallocation to another thread for the duration of its execution.

In the spirit of the aforementioned optimization, TPL introduces the notion of “task inlining”:¹⁴ If the current thread is instructed to wait on a task that has not yet started being executed by another thread, then the scheduler may allow it to execute the said task directly, rather than block. However, the behaviour is nondeterministic, since the parent thread could be outraced by thief threads stealing its tasks, consequently being constrained to block. Additionally, even if inlining is successful and the parent thread gets to execute a subtask, it might subsequently still need to block until all the other subtasks complete too. This imposed synchronicity is particularly problematic for unbalanced D&C algorithms, such as quicksort.

The default thread pool in the .NET Framework mitigates the issue by firing up new worker threads to compensate whenever a task blocks or takes too long to complete.¹⁵ Apart from incurring thread overheads, this approach is unsuitable for our scheduler, since most of our tests require a fixed degree of concurrency. Such bounded schedulers are susceptible to starvation, with all threads but one getting blocked (despite the availability of abundant pending tasks). In fact, it was this performance degeneration that spurred us to come up with a better execution pattern.

4.2.2 Asynchronous Execution

The solution we devised was to switch the recursion from a blocking operation to an asynchronous one. Whenever a parent task spawns subtasks, it attaches a [continuation](#) to each subtask, and then finishes, freeing up its thread for other tasks. Within the continuation, each subtask checks whether it is the last to complete from among its siblings (by performing an [atomic decrement](#) on a shared counter initialized to the number of subtasks); if it *is* the last, it spawns a new task constituting the merge operation for the parent level of the recursion.

¹⁴ Refer to the “[Task.Wait and ‘Inlining’](#)” article by Stephen Toub for an explanation.

¹⁵ Thread injection has been mentioned in Section 3.1.2 (p. 23); refer to the “[.NET CLR Thread Pool Internals](#)” article by Aviad Ezra for an elaboration.

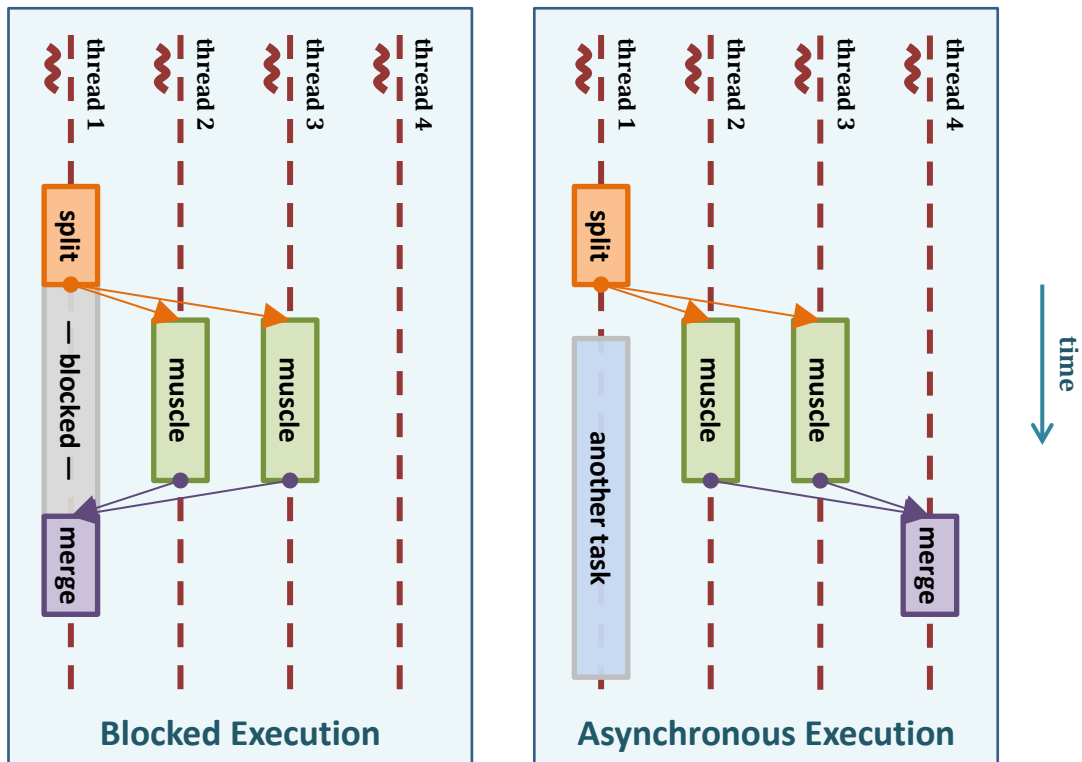


Figure 4.6. Blocking vs. asynchronous execution of D&C recursion

A condensed version of our asynchronous implementation is given in Figure 4.7 (below). Its logic is modelled after the internal implementation of the `TaskFactory.ContinueWhenAll` method in TPL.¹⁶ An essential component is the `TaskCompletionSource`, which acts as the producer side for a task whose state can be controlled explicitly (rather than through the scheduler).¹⁷ The `TaskCompletionSource` is signalled by the last subtask to complete, causing the merge task (which would be registered as its continuation) to get scheduled. Since all this behaviour is intrinsic to the TPL infrastructure, no thread-blocking synchronization is involved.

¹⁶ We could not call this method directly since its implementation is buggy (prone to deadlock) under some versions of Mono.

¹⁷ Refer to “[The Nature of TaskCompletionSource<TResult>](#)” by Stephen Toub.

```

1 // Create a task source to indicate completion
2 // when all subtasks collectively complete.
3 var completionSource = new TaskCompletionSource<bool>();
4 int subTasksLeft = subProblems.Length;
5
6 // Register a continuation for each subtask.
7 foreach (var subTask in subTasks)
8     subTask.ContinueWith(_ =>
9     {
10         // Atomically decrement the number of subtasks left,
11         // and set the completion source if it was the last.
12         if (Interlocked.Decrement(ref subTasksLeft) == 0)
13             completionSource.SetResult(true);
14     });
15
16 // Register a continuation to the completion source,
17 // so that the merge task is spawned when all subtasks complete.
18 Task<TResult> mergeTask = completionSource.Task.ContinueWith(_ =>
19 {
20     // Collect the subresult from each subtask.
21     var subResults = new TResult[subProblems.Length];
22     for (int i = 0; i < subProblems.Length; ++i)
23         subResults[i] = subTasks[i].Result;
24
25     // Merge subresults to obtain the current-level result.
26     return Merge(subResults, level);
27 });

```

Figure 4.7. Task execution in D&C skeleton using asynchronous parallelism (excerpt)

A complication present in this design is its recursive treatment of futures. Since the `Process` method is now asynchronous, it returns as soon as it spawns its subtasks, without waiting for the recursive subcomputation to complete. Thus, it cannot return the subresult of its merge operation directly (`TResult`); rather, it must return a future representing the merge task itself (`Task<TResult>`).

The recursive nature of D&C requires this notion of futures to be propagated across the recursion levels. At level i , the merge task must follow the collective completion of all its level $i + 1$ subtasks, whose merge tasks must, in turn, have followed all their respective level $i + 2$ subtasks, and so on. The merge tasks are not available immediately when the subtasks are spawned, but would be constructed dynamically as they execute.

Thus, when the `Input` method spawns a subtask, it does not acquire a `Task<TResult>`, but a `Task<Task<TResult>>` representing a future that will eventually give the merge task. This nested asynchronicity quickly becomes unwieldy when ascending back up the recursion graph, so we unravel it at each step using `TaskExtensions.Unwrap`, as shown in Figure 4.8.

This nifty method converts a `Task<Task<TResult>>` into a `Task<TResult>` by creating a proxy task that returns the result once the *inner* task completes.¹⁸

```
1 public Task<TResult> Input(TParam problem, int level = 1)
2 {
3     // Spawn a new task for processing the current problem.
4     Task<Task<TResult>> outerTask = TaskFactory.StartNew(() =>
5         this.Process(problem, level));
6
7     // Create a proxy task that unwraps the inner task.
8     return outerTask.Unwrap();
9 }
```

Figure 4.8. Unravelling nested asynchronicity at each recursive step

The asynchronous implementation yielded significant performance gains over the fork–join pattern, approaching twofold speedups for highly-scalable programs. Furthermore, it improved their performances’ consistency (reducing variance), since the nondeterministic risk of blocking is eliminated.

¹⁸ Refer to the “[How to: Unwrap a Nested Task](#)” article on MSDN for more details. The `Unwrap` method may be internally implemented using a `TaskCompletionSource`, as is done in .NET 4.0 and Mono (per its [TaskExtensionsImpl](#) class).

4.3 Program Hierarchy

Now that we have explained how our D&C skeleton runs, we shall discuss how we built our parallel programs atop it, as outlined in Figure 4.9.

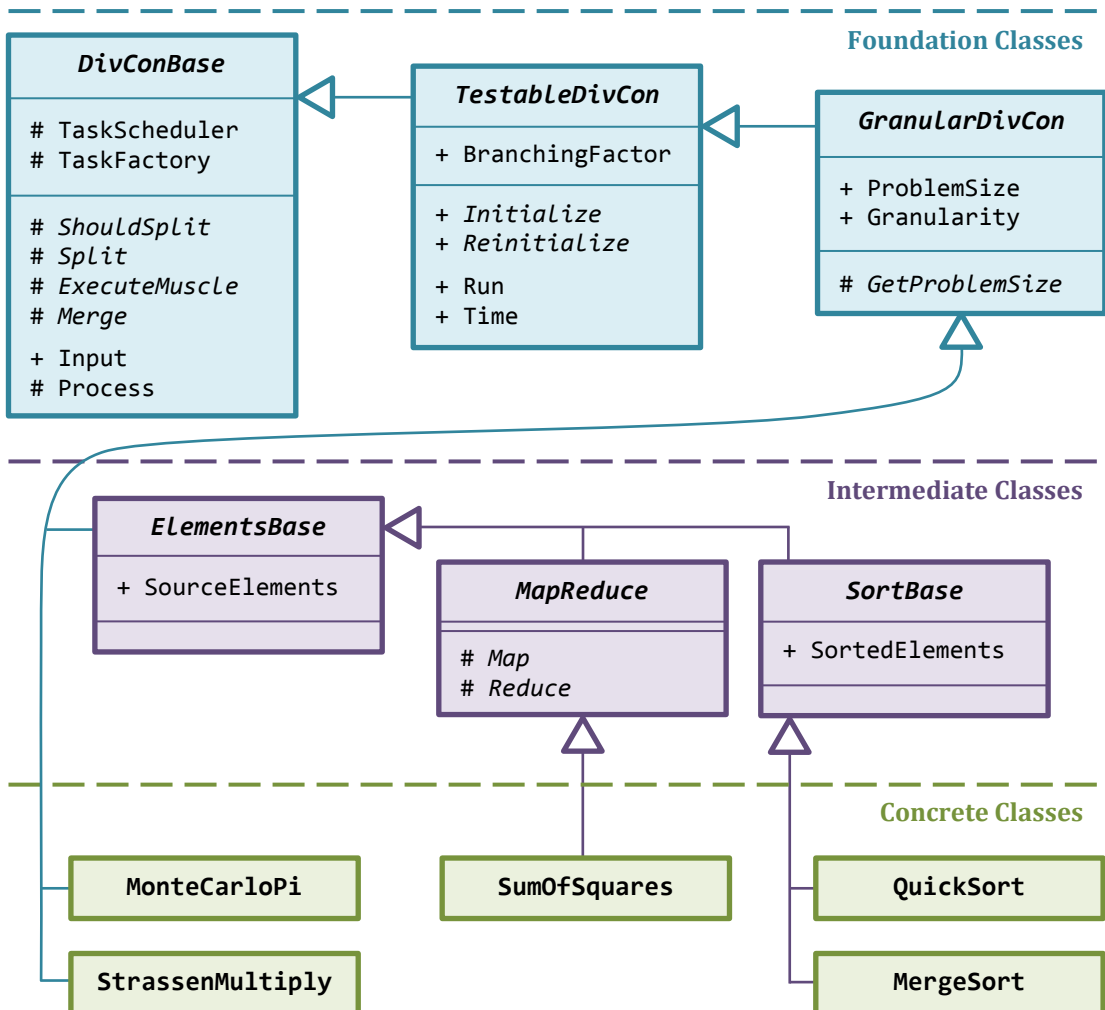


Figure 4.9. Class hierarchy for the D&C skeleton and programs

DivConBase, whose implementation we have been discussing so far, represents the core D&C skeleton, embodying the logic for executing arbitrary D&C algorithms as a task graph on a specified scheduler.

TestableDivCon introduces some convenient functionality for our tests, such as the `Initialize` abstract method, which is to be overridden by the parallel programs for initializing problem instances with random data. Its `Time` method may be called to initialize such a random problem and measure its execution time. The `BranchingFactor` property should be overridden to give the appropriate value according to the D&C algorithm.

GranularDivCon introduces the notion of granularity. Concrete programs are expected to implement the `GetProblemSize` method for indicating the size of a specified problem (or subproblem) instance, with the semantics of such a measure being left up to the program itself (as long as its values are numeric). For example, sorting algorithms can return the number of elements falling under the current subrange. Similarly, concrete programs should set the `ProblemSize` property to indicate the size of their overall problem (whose data may be generated randomly during initialization), whilst `Granularity` should give the target subproblem size at which to stop splitting (thereupon proceeding to compute the subproblem directly through the sequential execute muscle). In return for programs supplying this context, our D&C framework can support richer interaction with their execution, as discussed in Section 4.5.2 (p. 50).

`ElementsBase` serves as the basis for parallel programs operating over large collections of elements, such as map–reduce and sorting algorithms. It implements the `Initialize` method to generate an array of random numbers whose length corresponds to `ProblemSize`; this array is stored in its `SourceElements` property. `ElementsBase` fixes the `TParam` type to be `Range`, which contains `Min` and `Max` indexes representing the bounds of the current range (or subrange). It also provides a nominal implementation of the `Split` method that takes a range and divides it evenly into a sequence of subranges, whose number is equal to the `BranchingFactor` value. For example, with a branching factor of 4, the range `[0, 100)` would be split into `[0, 25)`, `[25, 50)`, `[50, 75)`, and `[75, 100)`.

`MapReduce` may be seen as a specialization of the D&C skeleton to handle map–reduce programs (and thereby qualifying as a skeleton in its own right). It defines its own set of abstract methods to be overridden by concrete programs, as presented in Figure 4.10, with the `Reduce` operation assumed to be associative. However, it fully implements all of the D&C skeleton’s methods to channel the computation to `Map` and `Reduce`. (For more details and a concrete example, refer to Section 4.4.1, p. 42.)

```
1 public abstract partial class MapReduce<TElement, TResult>
2     : ElementsBase<TElement, TResult>
3 {
4     protected abstract TResult Map(TElement element);
5     protected abstract TResult Reduce(TResult res1, TResult res2);
6     protected virtual TResult Reduce(TResult[] results) { /* ... */ }
7 }
```

Figure 4.10. Map–reduce methods for functional parameterization

SortBase, the base class for sorting algorithms, introduces the SortedElements property. For in-place sorting implementations, this would reference the very same array as SourceElements; otherwise, it is initialized to an empty array of the same length, to be populated with the sorted result. SortBase implements ExecuteMusc1e to call the .NET built-in `Array.Sort` method for handling the base case of the recursion (like Skandium uses Java's `Arrays.sort` in the base case of its parallel quicksort [25]).

4.4 Sample Programs

We shall now mention the repertoire of parallel programs that we implemented as instances of our D&C skeleton. After consulting Tsogkas [48] for an analysis of the characteristics of some D&C algorithms, we handpicked a selection that is well-suited for the scope of our experiments; namely, compute-intensive applications spanning a good range of scalabilities.

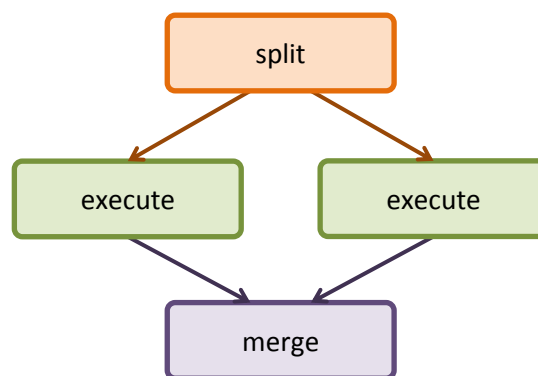


Figure 4.11. Colour convention for the D&C diagrams presented in the rest of this section

4.4.1 Sum of Squares

The sum-of-squares program is a simple map–reduce instance, where Map is a squaring operation (x^2) and Reduce is a summation (Σ). By default, it initializes an array of 134,217,728 integers in memory (equivalent to 512 MB), populated with random values ranging from 0 to 1024. (The exclusive upper bound is kept low so as to avoid integer overflows.)

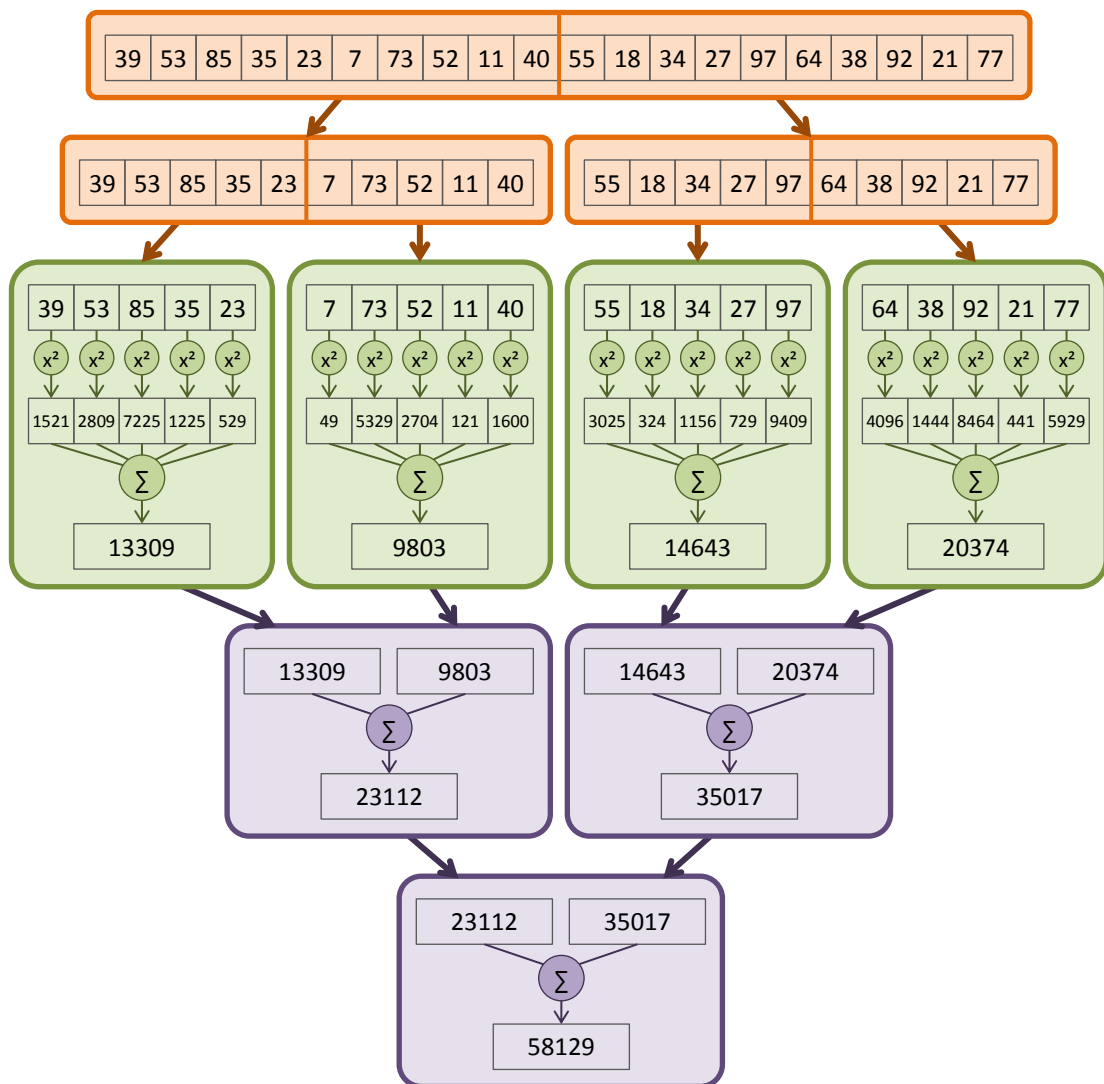


Figure 4.12. Sample D&C execution of sum-of-squares

Figure 4.12 shows a sample execution of this program. The split operation just divides the current range according to the branching factor. At the base case, the map operation (squaring) is iteratively applied over all elements belonging to the current subrange, with

their results aggregated using the reduce operation (summation). Finally, the merge operation again applies the reduce operation to aggregate the subresults from its subtasks.

Map–reduce programs allow their branching factor to be altered to an arbitrary value. For example, the branching factor may be set to correspond to the number of logical cores on the machine, so that full concurrency would be achieved after just one split (as shown in Figure 4.13 for the next program). However, since the split and merge operations are computationally trivial, the performance gain is negligible.

The data parallelism offered by map–reduce programs such as sum-of-squares is embarrassingly parallel, yielding near-linear speedups even on manycore machines, thereby belonging to the top end of our scalability spectrum.

4.4.2 Monte Carlo Pi

The `MonteCarloPi` program uses a Monte Carlo simulation to estimate the value for π (π).¹⁹ Assume a circle whose radius, r , is 0.5 units, inscribed within a square of length, l , 1. Using standard definitions, we know that:

$$\begin{aligned}Area_{circle} &= \pi r^2 \\Area_{square} &= l^2\end{aligned}$$

By combining the two equations and substituting our dimensions, we can compute the ratio of their areas, R , as:

$$R = \frac{Area_{circle}}{Area_{square}} = \frac{\pi r^2}{l^2} = \frac{\pi(0.5)^2}{(1)^2} = \frac{\pi}{4}$$

Conversely, if R is known, we could compute π as:

$$\pi = 4 \cdot R$$

We therefore employ Monte Carlo methods to estimate this ratio by sampling a large number of random points within the square, and testing how many of them fall within the

¹⁹ This explanation is adapted from the “[Estimating Pi with Monte Carlo Methods](#)” tutorial by Joe Freeman.

circle using Pythagoras' theorem. For example, if 785,113 out of 1,000,000 random points fall within the circle, then we could estimate π as:

$$\pi = 4 \cdot R \cong 4 \cdot \frac{785,113}{1,000,000} \cong 3.14$$

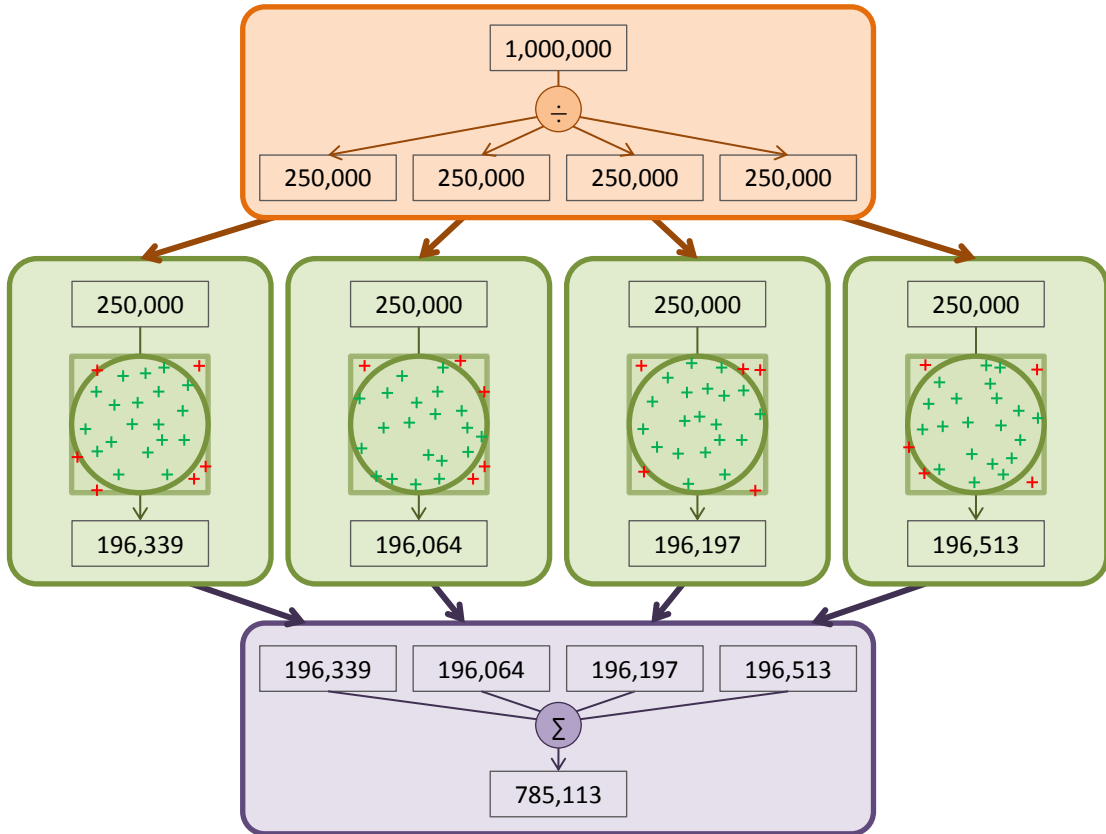


Figure 4.13. Sample D&C execution of Monte Carlo Pi

This computation is trivial to parallelize. The program does not require any initialization; the initial problem is defined to be the *number* of random points to be sampled, and set to 1,073,741,824 (being 2^{30}) for our experiments. In the split operation, this number is divided evenly among the subtasks. Like in SumOfSquares, the branching factor may be set to an arbitrary value; in the example on Figure 4.13, we set this to 4, equivalent to the number of logical cores on that machine. In the base case, the Monte Carlo simulation is performed for the given number of points, and the number of in-circle points returned as the subresult. In the merge operation, the subresults are summed to obtain the total number of in-circle points overall.

4.4.3 Strassen Matrix Multiplication

The Strassen algorithm for matrix multiplication [49] permits large square matrices (of order n) to be multiplied with an asymptotic complexity of $O(n^{\log_2 7})$, which is approximately $O(n^{2.8})$, and therefore improves upon the $O(n^3)$ complexity of the standard matrix multiplication algorithm. It partitions each of the given pair of matrices, A and B , into four submatrices of order $\frac{n}{2}$, denoted as $A_{1,1-2,2}$ and $B_{1,1-2,2}$. It subsequently performs a set of 7 subcomputations on these, $I-VII$, each involving submatrix addition and multiplication:

$$\begin{aligned}
 I &= (A_{1,1} + A_{2,2}) \times (B_{1,1} + B_{2,2}) \\
 II &= (A_{2,1} + A_{2,2}) \times B_{1,1} \\
 III &= A_{1,1} \times (B_{1,2} - B_{2,2}) \\
 IV &= A_{2,2} \times (B_{2,1} - B_{1,1}) \\
 V &= (A_{1,1} + A_{1,2}) \times B_{2,2} \\
 VI &= (A_{2,1} - A_{1,1}) \times (B_{1,1} + B_{1,2}) \\
 VII &= (A_{1,2} - A_{2,2}) \times (B_{2,1} + B_{2,2})
 \end{aligned}$$

The algorithm's suitability for the D&C skeleton arises from this submatrix multiplication, which may be performed recursively, using the Strassen algorithm again. (The branching factor of this program is therefore 7.) Once all subcomputations have completed, the result matrix, C , may be computed through its partitions:

$$\begin{aligned}
 C_{1,1} &= I + IV - V + VII \\
 C_{1,2} &= III + V \\
 C_{2,1} &= II + IV \\
 C_{2,2} &= I - II + III + VI
 \end{aligned}$$

The execution flow is shown in Figure 4.14 (below). Both the split and the merge tasks need to perform submatrix additions (since, due to the D&C skeleton design, the subtasks can only take care of the recursive multiplications). Submatrix additions have an algorithmic complexity of $O(n^2)$. Despite being substantially lighter than the multiplications' $O(n^{2.8})$ or $O(n^3)$, these still incur a computational overhead, meaning that this parallel program will not be as scalable as the previous two.

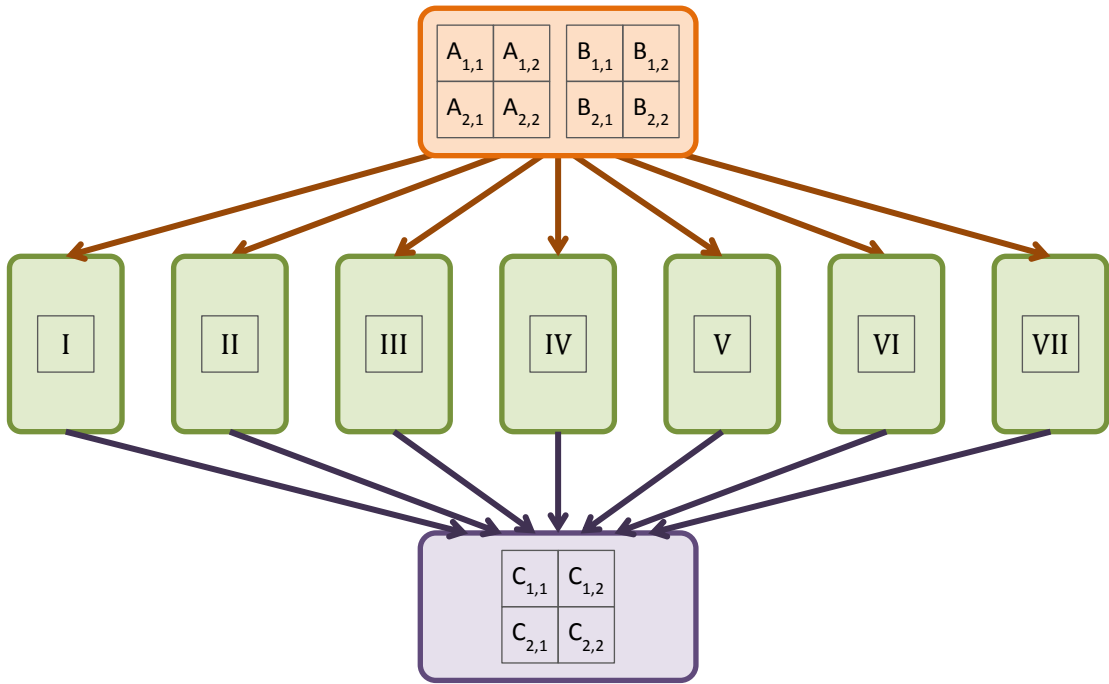


Figure 4.14. Sample D&C execution of Strassen matrix multiplication.

Only a single split level is shown here for simplicity. In practice, most runs would use 2 or 3 levels, resulting in 49 or 343 execute muscles respectively.

For our tests, we initialize our program with a pair of square matrices of order 1536, thereby each having 2,359,296 elements (9 MB), randomly populated with values between 0 and 1024 (exclusive).

4.4.4 Quicksort

Quicksort [50] is a sorting algorithm that, on average, can sort n items with $O(n \log n)$ comparisons. Its recursive definition makes it a natural D&C candidate. In the split operation, it picks a pivot value, and reorders the array such that all elements *smaller* than the pivot are placed together at its front, whilst all elements *larger* than the pivot are placed together at its rear. (We arbitrarily lump elements *equal* to the pivot with the latter group.) Subsequently, the two subarrays are sorted recursively. Once a subarray's length falls below a threshold (being the target granularity), the recursion is stopped and the base case engaged, which calls .NET's sequential `Array.Sort` method (as mentioned on p. 41). The merge operation does nothing.

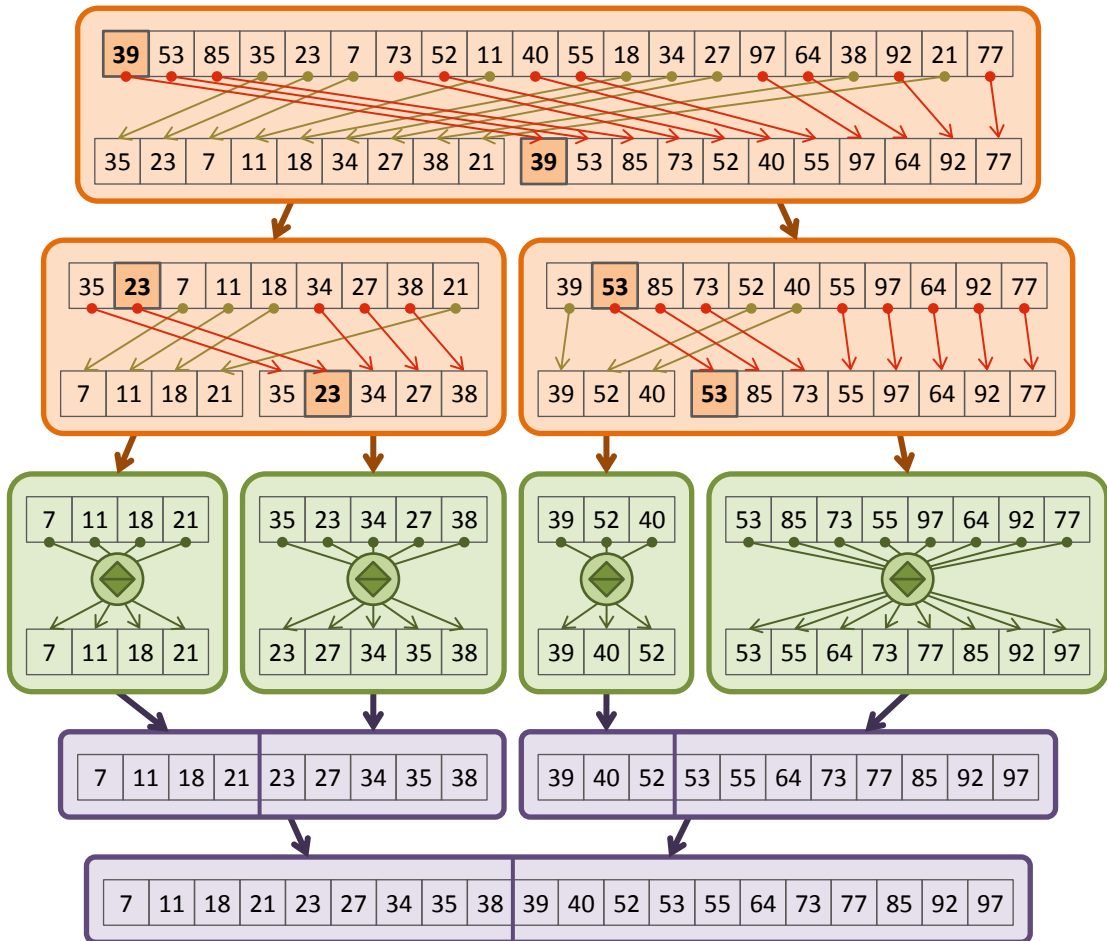


Figure 4.15. Sample D&C execution of quicksort. In this example, each pivot value (shown in bold) is chosen arbitrarily from among the first few elements of the subarray.

Quicksort’s amenability to parallelization comes from the disjoint quality of the subarrays resulting from each split, permitting them to be processed independently. However, each split operation performs a sequential sweep over the n elements in its array (or subarray), entailing $O(n)$ comparisons. Consequently, the parallelism is unravelled gradually, with the level i split operation(s) being processed over 2^{i-1} tasks (i.e. 1, 2, 4, 8, ...). This limits the scalability of the program, since several processors would remain unutilized for the early levels of the recursion.

For our tests, we initialize an array of 67,108,864 integers (256 MB), randomly populated with non-negative 32-bit integer values. Quicksort is our only parallel program that exhibits unbalanced behaviour, since the sizes of – and, therefore, work associated with – the subarrays may vary substantially, depending on the choice of the pivot value. To minimize this misbalance, at each split, we sample the first 100 elements of the subarray and pick

their median as the pivot. (Systematic sampling, which surveys elements spread throughout the subarray, might have yielded sturdier pivot choices, but incurs heavy memory overheads since, for large subarrays, each sample would incur a cache miss.)

4.4.5 Mergesort

Mergesort is another D&C sorting algorithm that is similar to quicksort, but performs its work in the merge phase instead of the split phase. In the split phase, it just divides the range of elements evenly among its subtasks. In the base case, it also uses .NET's sequential `Array.Sort` method to sort the elements along the subrange. Finally, in the merge phase, it sweeps sequentially along the n elements from the pair of independently-sorted subranges and rewrites them in-order, incurring $O(n)$ comparisons.

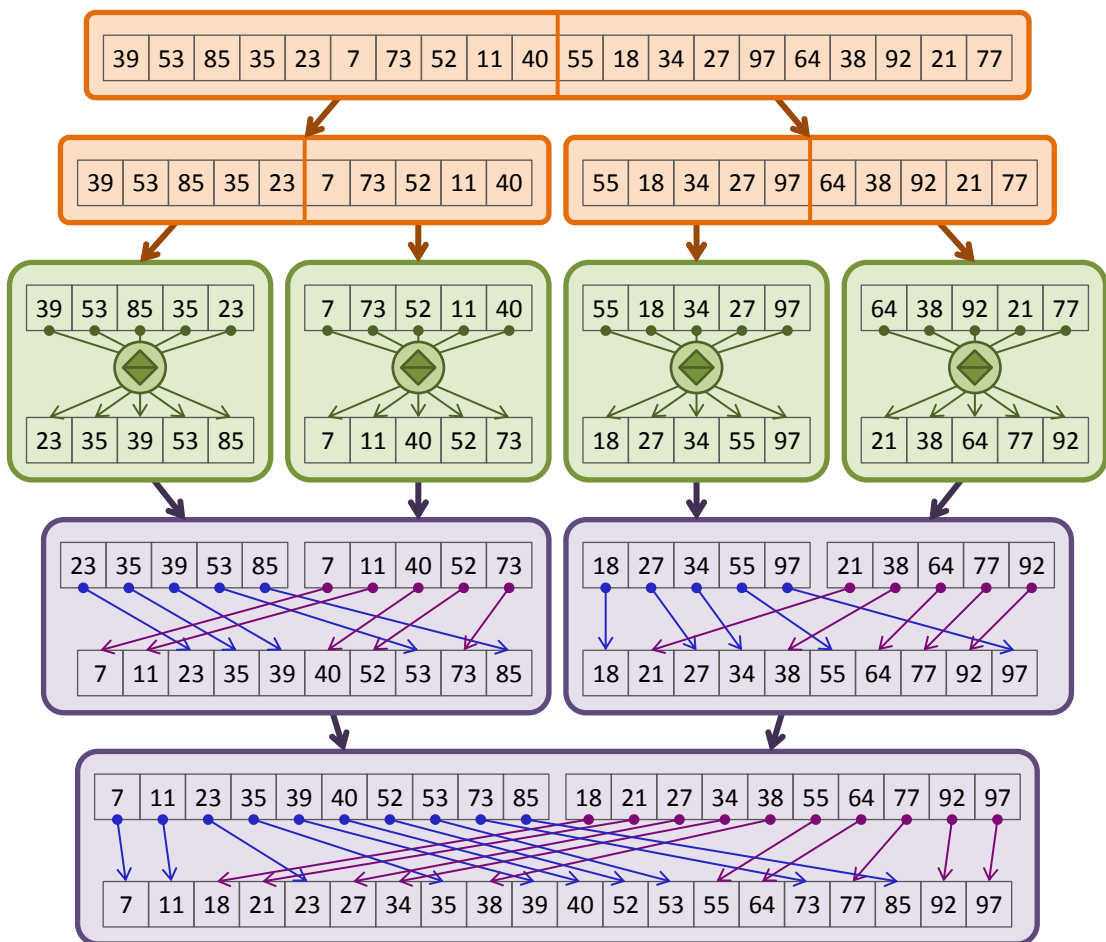


Figure 4.16. Sample D&C execution of mergesort. Unlike quicksort, this D&C sorting algorithm always produces size-balanced subtasks.

Mergesort suffers from the same processor underutilization issue as quicksort, since the merge operations at the final levels of the recursive ascent may only be performed through a limited number of tasks. Furthermore, mergesort is an out-of-place algorithm (as opposed to the in-place quicksort), meaning it is expected to perform slower. For our tests, we use the same problem size as for quicksort.

4.5 Common Design Considerations

4.5.1 Program Reinitialization

One of our major concerns for the cyclic multiprogramming performance tests (discussed in Section 5.4.1, p. 76) was the issue of reinitializing each program’s problem data between one run and the next. Under these tests, each program is reinitialized and restarted immediately upon completing (whilst the others are still executing). However, given that several of our programs work against large memory arrays (256 MB for sorting; 512 MB for map–reduce), reinitializing a program’s data whilst other programs are executing could lead to substantial negative interference, especially on the shared off-chip memory bandwidth, if the programs happen to be executing on cores belonging to the same processor chip [51].

To avoid this, we redesigned our system to eliminate the need for reinitialization by reusing the same data across consecutive runs of the same program instance. The extent of illicit performance gains arising from such data reuse (such as from temporal locality or warmed-up branch predictors) is limited, due to the large sizes and randomized nature of the data arrays.

However, two of our programs, quicksort and mergesort, were designed to overwrite their data while running. A sorting algorithm executed over pre-sorted data completes in drastically less time, even if the number of comparisons performed is identical, due to highly-successful branch prediction.²⁰ Thus, we extended our sorting programs to support non-destructive execution by writing their results to a distinct array, rather than overwriting the source. By integrating the “copying” into the algorithm itself, the overheads

²⁰ Refer to the StackOverflow question [“Why is processing a sorted array faster than an unsorted array?”](#) and the detailed explanation given in its [accepted answer](#) for a fascinating discussion of this behaviour.

are minimized: In mergesort, the copying is fully parallelized as part of the execute muscle, whilst in quicksort, it is performed organically within the first-level split (whilst the elements are being split based on the pivot). On 64 cores, the performance loss for quicksort is merely 3%, whilst for mergesort, the non-destructive version surprisingly achieves a *gain* of 3%.

4.5.2 Granularity

In Section 4.3 (p. 40), we mentioned that the `GranularDivCon` base class introduces the notion of granularity, thereby permitting our framework to interact with the D&C programs by acquiring awareness of their problem sizes. A rudimentary benefit is that it can provide a standard implementation of the `ShouldSplit` method, as shown in Figure 4.17.

```
1 public abstract partial class GranularDivCon<TParam, TResult>
2     : TestableDivCon<TParam, TResult>
3 {
4     public long ProblemSize { get; set; }
5     public long Granularity { get; set; }
6
7     protected abstract long GetProblemSize(TParam problem);
8
9     protected override bool ShouldSplit(TParam problem, int level)
10    {
11        return GetProblemSize(problem) > Granularity;
12    }
13 }
```

Figure 4.17. Main components of the `GranularDivCon` class

`ShouldSplit` may still be overridden by the concrete programs to provide more restrictive behaviour. For example, `StrassenMultiply` ceases to be efficient if the orders of the matrices drop below a certain threshold, making it preferable to stop splitting (and switch to standard matrix multiplication) before this happens.

The `GranularDivCon` class also allows our framework to trivially create sequential instances of our D&C programs, by setting their `Granularity` to be equal to their `ProblemSize` (representing the overall problem size). This way, their first (and only) invocation of `ShouldSplit` would return `false`, causing the D&C skeleton to proceed to compute the entire problem directly through a single `ExecuteMuscle` call (without any `Split` or `Merge` operations).

Another benefit of the GranularDivCon design is that our system can heuristically compute a suitable value of Granularity for parallel systems, alleviating this decision from the application developer. Under the current implementation, the system sets this value such that it results in a number of execute muscles that is approximately equal to a small multiple of the number of logical cores on the machine. Campbell et al. [34] recommend 16 tasks per core for quicksort, to compensate for the inherent load-imbalance among its subtasks. However, given our improved pivot-selection heuristic in quicksort, as well as the better load-balance in the other programs, we found that good performances could be achieved even when using a target of just 4 execution muscles per core.

For D&C programs whose split operation reduces the problem size by the same factor as the branching factor, the granularity is straightforward to compute: An approximate target of n execute muscles may be achieved by setting the granularity to $1/n$ the overall problem size. Among our programs, the only exception is the Strassen multiplication, whose split operation reduces the problem size by a factor of 4 (since each submatrix has $1/4$ the number of elements of its parent matrix), despite the branching factor being 7. Thus, a target of n execution muscles requires the granularity to be set to $1/[4^{\lceil \log_7 n \rceil}]$ the overall problem size.²¹

The above computation is an approximation only in the case of D&C algorithms whose split operation may produce unbalanced subproblems – namely, quicksort. Instead of granularity, one could alternatively have used the recursion depth as the basis of the split condition. The second parameter of the ShouldSplit method provides the current recursion level; by comparing this against a target recursion depth, one could constrain the D&C program to produce an exact number of execute muscles (provided that this number is a power of the branching factor). Specifically, for a program with a branching factor of b , a total of n execute muscles may be produced by splitting until a recursion depth of $\log_b n$.

²¹ This result is mathematically related to the algorithmic complexity of the Strassen algorithm, which is $O(n^{\log_2 7})$ [49]. Note that Strassen assumes n to be the *order* of the matrix, rather than its number of elements (which would be the order squared).

The issue with using the recursion depth as the split condition is that, unlike granularity, it does not even out load misbalance. Figure 4.18 (below) shows how the two strategies influence the division of a problem of size 1000 into a target of 4 execute muscles. A level-based approach would split until a recursion depth of 2 to produce exactly 4 muscles, whilst a granularity-based approach would split until the subproblem size is 250. In our example, the latter approach produces 5 muscles (one more than the target); however, their subproblem sizes vary by a standard deviation of merely 27, as opposed to 191 for the level-based approach.

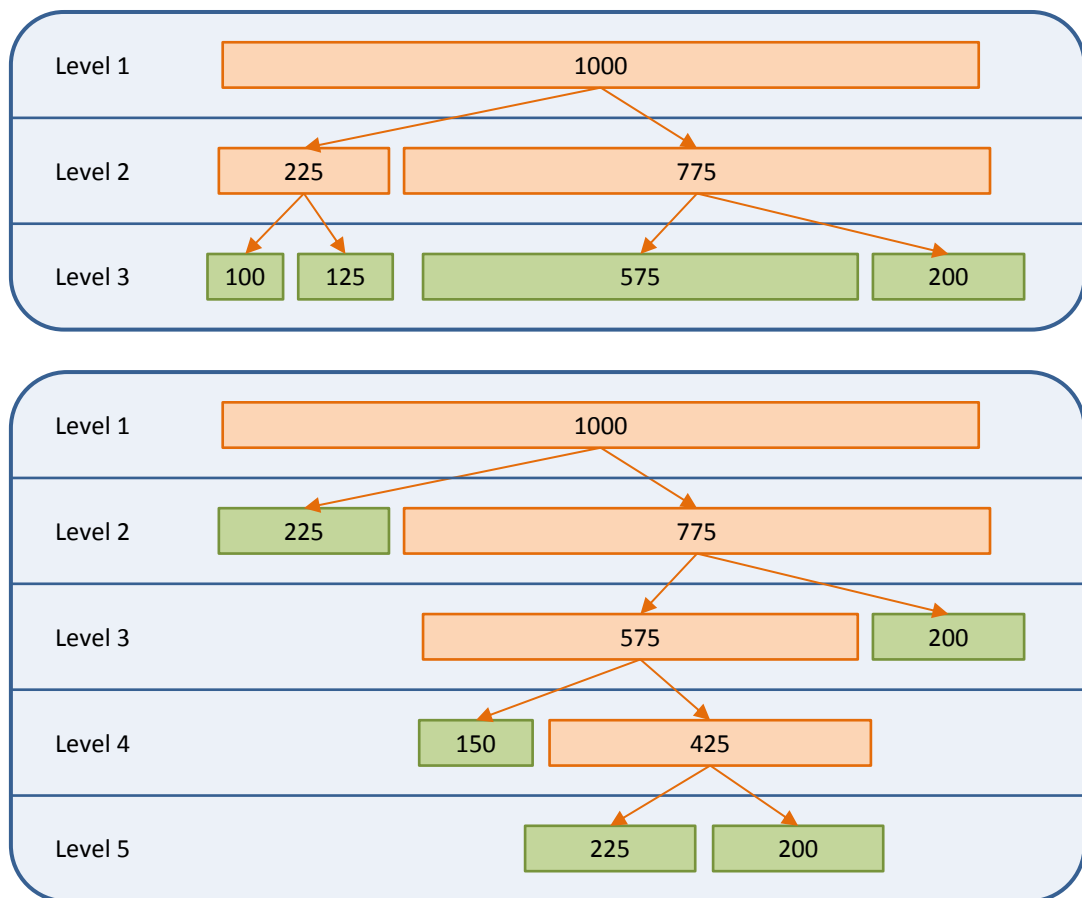


Figure 4.18. Delimiting the split phase based on recursion depth (top) versus granularity (bottom). The tasks are drawn to-scale with respect to their subproblem sizes; however, this should not be construed as representative of their computational requirements, since the split operation is typically substantially cheaper to process than the execute muscle for a given subproblem size. Note that none of our D&C programs suffer from such extreme misbalance in practice.

4.5.3 Nested Parallelism

In our programs, we have assumed that the split and merge operations of our D&C programs should be *sequential*. The reasons behind this decision were threefold: It conforms to the definition of the D&C skeleton muscles in Skandium [48]; it simplifies the design of our programs; and it makes their performances easier to analyse. However, as mentioned briefly in Section 4.1.4 (p. 32), we acknowledge that better scalabilities might be possible by parallelizing these operations as well.

Chapter 5: Scheduler Designs

The previous chapter explained how our D&C skeleton empowers programs to leverage its structured parallelism to transparently generate task graphs. This fulfils the first parallelization challenge presented in Section 2.2 (p. 9): problem decomposition. Now, we proceed to address the next challenge: the efficient distribution of tasks onto processors for concurrent execution.

We open this chapter by explaining how we extend the structure of the traditional work-stealing task scheduler to support explicit processor affinity through thread pinning. In Section 5.2, we present a number of multiprogramming schemes that permit concurrently-executing programs to feed their tasks into one or more task schedulers, showing how a shared scheduler would eliminate the need for thread oversubscription. Section 5.3 is the most important part of our system design, since it presents the novel approach through which we reconcile task parallelism with processor partitioning, discussing both mechanism and policy. Finally, Section 5.4 gives an overview of the principal tests that we ran to evaluate our system, whose results will be presented towards the end of Chapter 7.

5.1 Scheduler Hierarchy

Figure 5.1 extends the class hierarchy presented in Figure 3.3 (p. 25) to introduce the two task schedulers of our system.

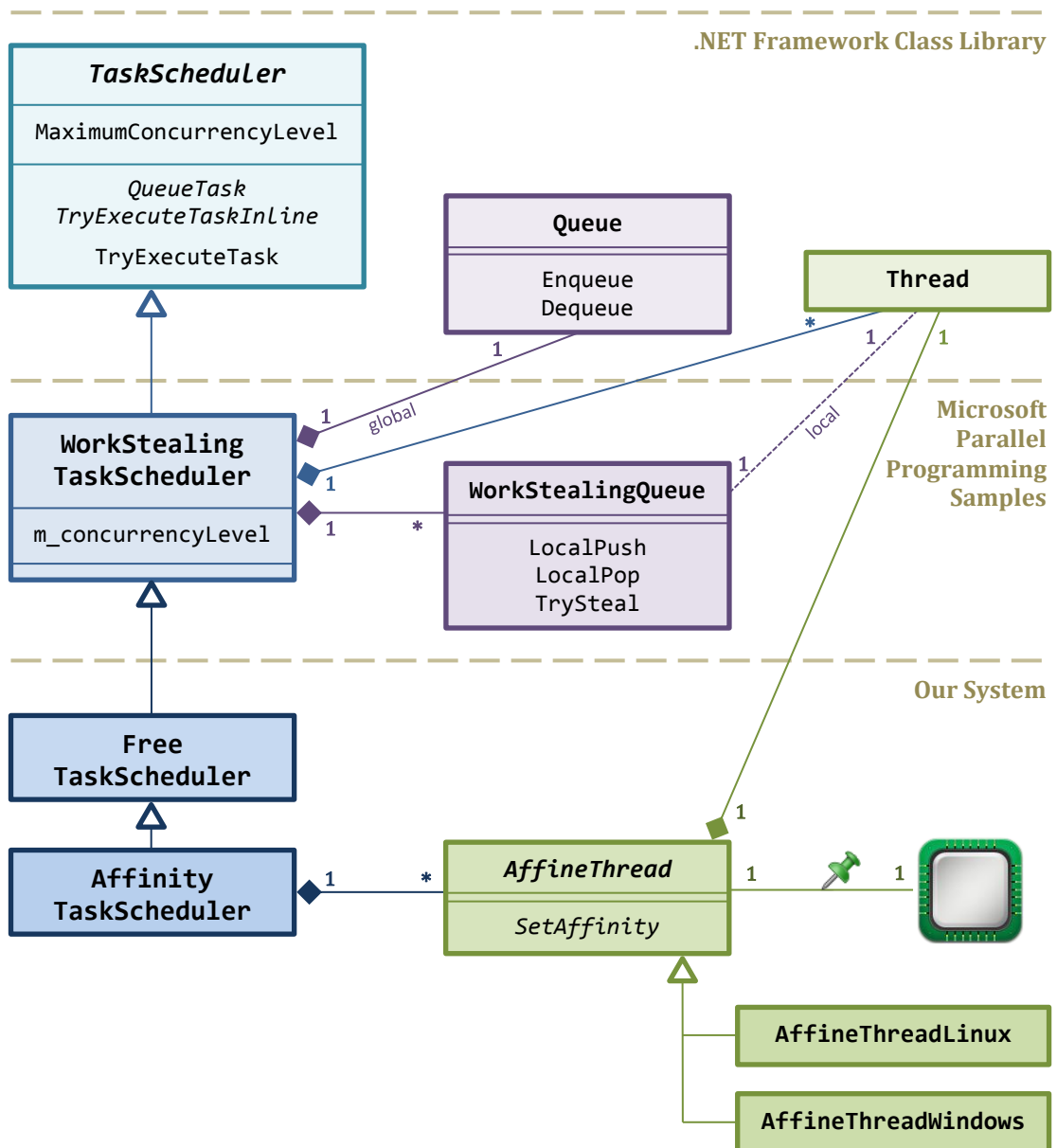


Figure 5.1. Class diagram showing the composition of our task schedulers. The one-to-one correspondence between affine threads and processors is an imposition of our system; in practice, the relationship may be many-to-many.

5.1.1 Free Task Scheduler

Our `FreeTaskScheduler` is a lightweight wrapper over the `WorkStealingTaskScheduler` from the [Microsoft Parallel Programming Samples](#) (described in Section 3.1.3, p. 24). The name was chosen to signify the liberty granted to the operating system's thread scheduler for allocating the worker threads onto the available processors (as opposed to the `AffinityTaskScheduler` discussed in the next section). Due to inheritance, its structure is the same as that of `WorkStealingTaskScheduler` (presented in Figure 3.4, p. 26).

We introduce some minor changes to improve the consistency of the scheduler's behaviour for our tests. Specifically, we alter its default number of worker threads to correspond to the number of logical cores on the machine; we suppress initialization of new dedicated threads for tasks marked as `LongRunning`; and we prohibit [task inlining](#) for external threads. The latter change was crucial for getting sensible measurements of execution times over limited concurrencies – with inlining permitted, some tasks would get executed on the main thread itself, thereby illicitly inflating the actual degree of concurrency.

5.1.2 Affinity Task Scheduler

`AffinityTaskScheduler` extends `FreeTaskScheduler` such that, rather than accepting a plain (numeric) concurrency level, its constructor can take a set of specific processor identifiers. For each specified processor, `AffinityTaskScheduler` initializes an `AffineThread` wrapper that creates a worker thread and pins it to the said processor, as shown in Figure 5.2 (below).

The .NET Framework provides built-in support for setting processor affinity for threads through its [ProcessorAffinity](#) interface;²² however, this functionality is not implemented in Mono. Instead, we use [Platform Invocation Services \(P/Invoke\)](#), which play a similar role to the Java Native Interface (JNI),²³ to perform platform-specific system calls for setting affinity. Specifically, we use `SetThreadAffinityMask` on Windows and `sched_setaffinity` on Linux, structuring our class design to dynamically choose the appropriate call for the current platform.

²² Refer to the [“Running .NET threads on selected processor cores”](#) tutorial by Lenard Gunda.

²³ Refer to the [ThreadAffinity](#) Java class, implemented by Ruslan Cheremin, for sample JNI code.

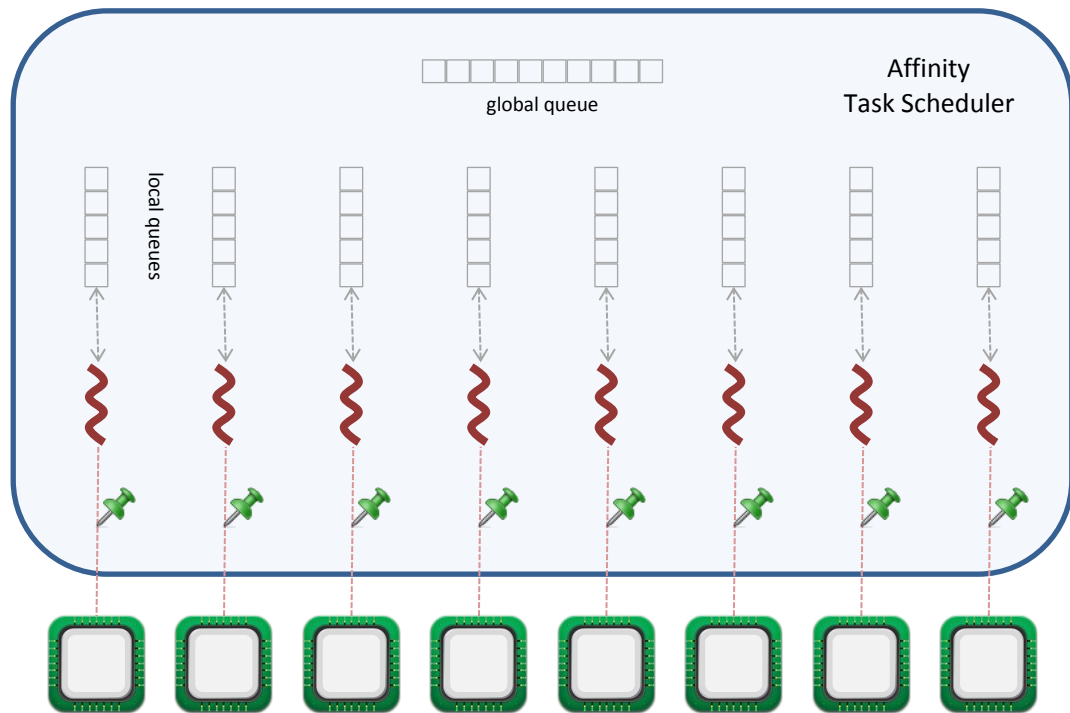


Figure 5.2. Structure of the affinity task scheduler. This scheduler builds on top of the work-stealing task scheduler (whose structure was presented in Figure 3.4, p. 26), inheriting its global queue and per-thread local queues. However, the affinity task scheduler pins each worker thread to a distinct core.

5.1.3 Contiguous vs. Dispersed Allocation

Whenever our system needs to allocate a subset of any n processors to an affinity task scheduler, it defaults to assuming a contiguous allocation (e.g. processors 1 to n). Such processor proximity can improve the performance of a parallel program due to constructive interference (as discussed in Section 2.3.1, p. 16). However, cores residing on the same chip share its off-chip bandwidth, which can become a performance bottleneck for parallel programs with high memory access demands [51]. In order to investigate the extent of this issue, we devised an alternate allocation strategy where the worker threads of the affinity task scheduler are pinned to a *dispersed* subset of the cores, as depicted in Figure 5.3 (below). (Note that this allocation strategy is only employed for the stand-alone experiment whose results are presented in Section 7.2.1, p. 86. For all other scenarios, contiguous allocations should be assumed.)

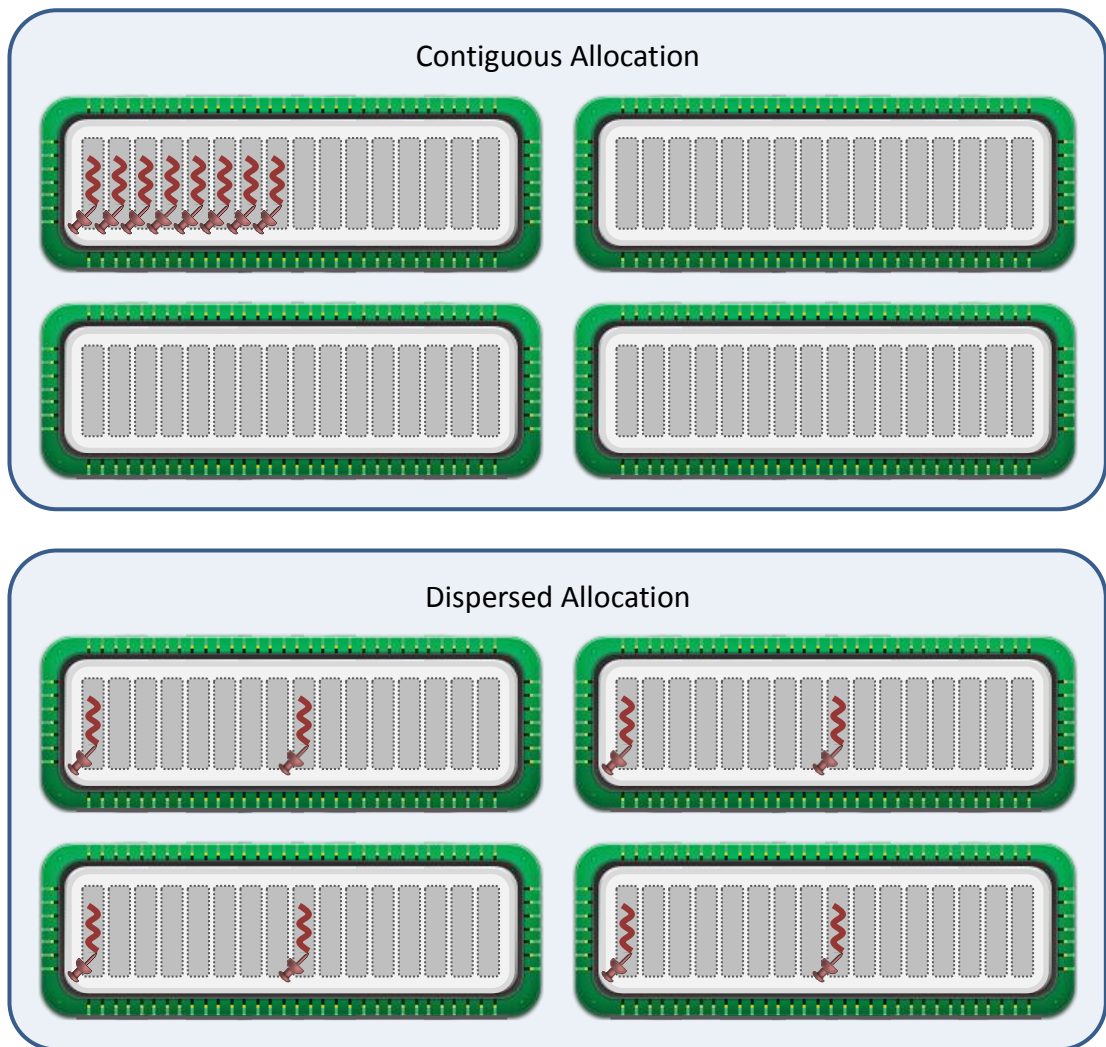


Figure 5.3. Contiguous vs. dispersed allocation of 8 pinned threads over a 64-core machine comprised of four 16-core processor chips

5.2 Multiprogramming Schemes

Since our parallel programs are implemented as instances of the D&C skeleton, they need to feed into a task scheduler before their computations may be delegated onto worker threads. So far, our scheduling discussions have been agnostic of the program workload, simply assuming that the programs would somehow have their top-level tasks inserted into the appropriate scheduler's global queue. We shall now proceed to describe a number of multiprogramming schemes we devised, each of which shows how programs may be serviced by one or multiple task schedulers.

5.2.1 Oversubscribed Free Multiprogramming

As mentioned in Section 2.3.1 (p. 16), parallel applications that have no awareness of their system's multiprogramming context would typically attempt to individually maximize their utilization of the machine's multiprocessing capabilities by initializing one thread (or more) per logical core. This is particularly the case for task-parallel libraries, such as Skandium [25] and TPL [34], which initialize such a pool of worker threads for servicing their task queues. Thus, in a multiprogramming scenario, some processors would get oversubscribed with multiple threads from the various programs (see Figure 5.4), leaving it up to the operating system to schedule these threads efficiently using round-robin time-slicing.

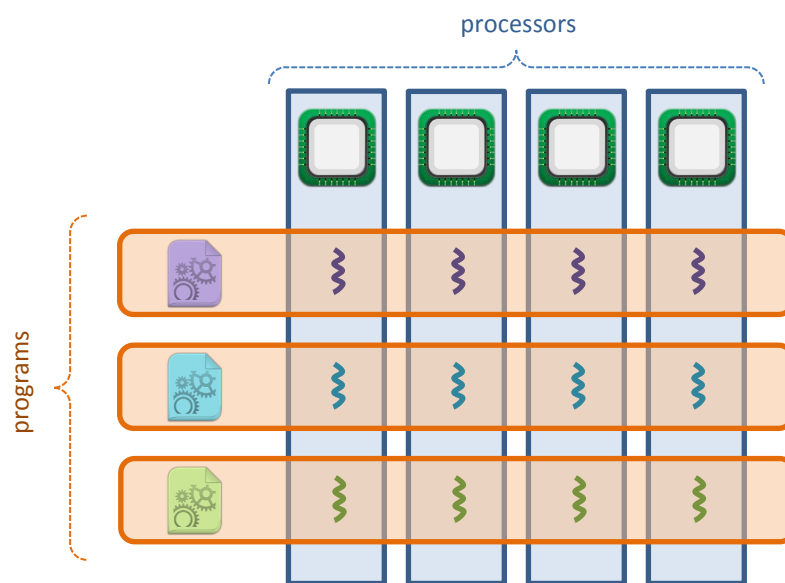


Figure 5.4. Oversubscription by multithreaded applications. For simplicity, this diagram shows each processor running exactly one thread from each application. In practice, most applications do not pin their threads, meaning that the thread scheduler is free to migrate them into arrangements different from the above.

We shall adopt this behaviour as the baseline against which to compare our optimizing scheduler (discussed later) for tests involving multiprogrammed workloads. We could simulate it either by running each program as a separate process, or by instantiating a distinct task scheduler for each program instance, as illustrated in Figure 5.5 (below). We picked the latter approach, since it allows us to run our programs within the same virtual address space, simplifying the implementation of our tests.

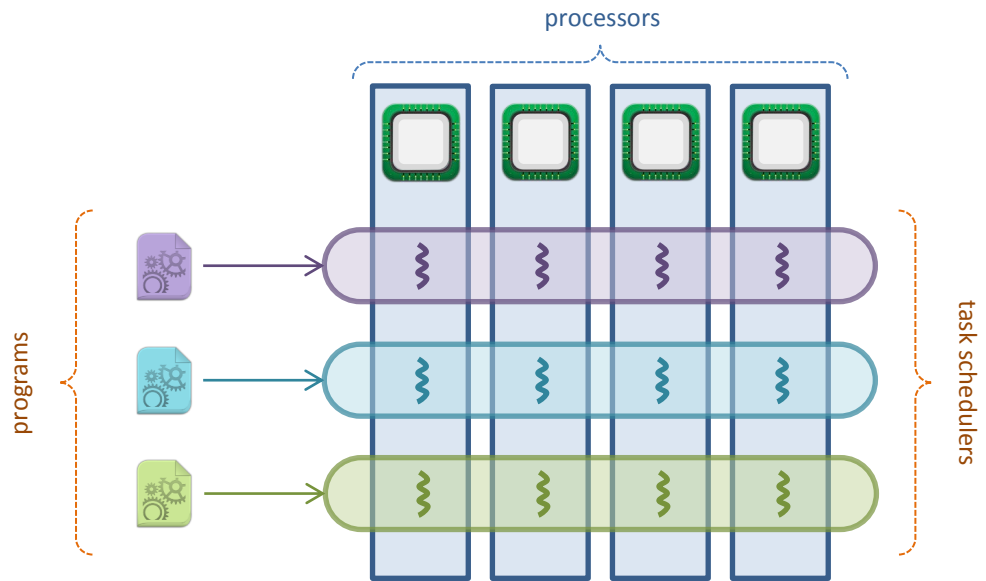


Figure 5.5. Oversubscription by task-parallel programs feeding dedicated task schedulers

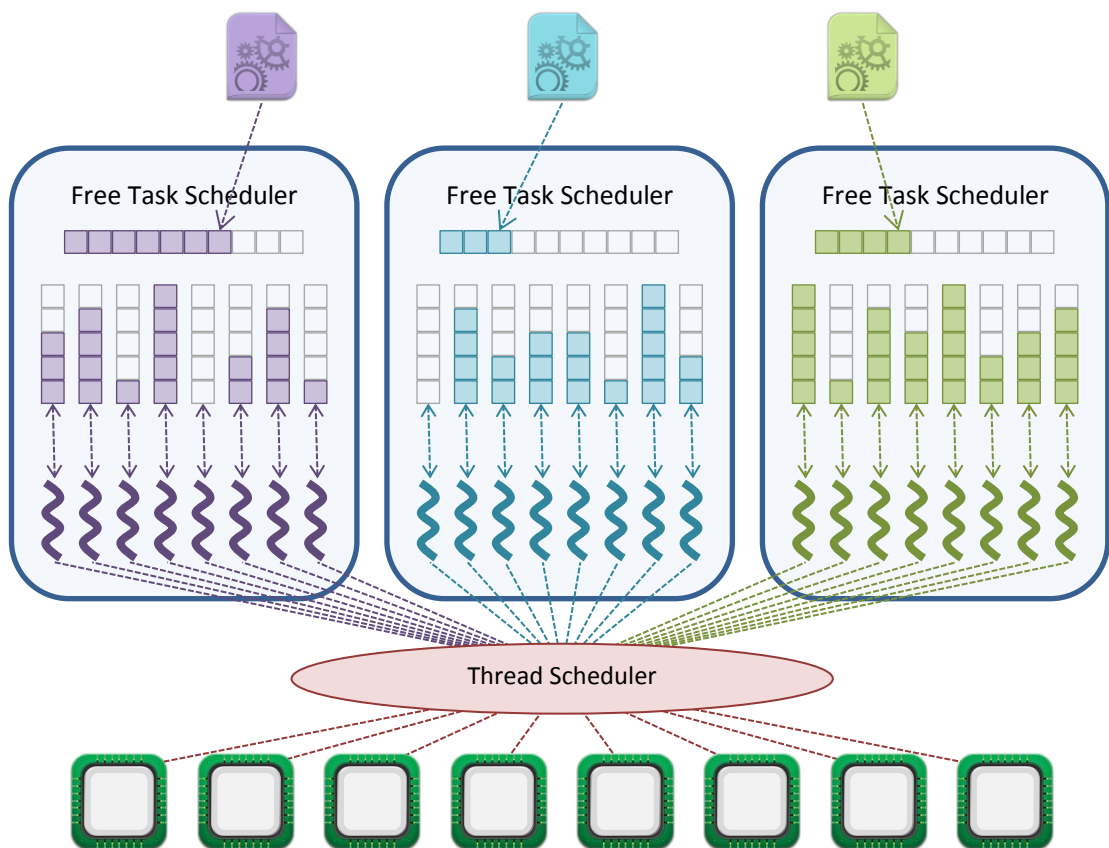


Figure 5.6. Oversubscribed free multiprogramming. The thread scheduler is responsible for distributing all the worker threads (from the various task schedulers' respective thread pools) over the available processors.

Under our “oversubscribed free multiprogramming” scheme, a distinct FreeTaskScheduler is created per program, with each task scheduler initializing a free thread per core, as shown in Figure 5.6 (above). Each program issues tasks to its own task scheduler. Since the threads are not pinned, they can be migrated across processors by the operating system’s thread scheduler, according to its own scheduling strategy.

5.2.2 Oversubscribed Pinned Multiprogramming

Under the “oversubscribed pinned multiprogramming” scheme, a distinct AffinityTaskScheduler is created per program, with each task scheduler initializing a pinned thread on each core, as shown in Figure 5.7. Thus, each core is oversubscribed exactly per the degree of multiprogramming.

This scheme is identical to free multiprogramming, except that it inhibits any thread migration by the operating system’s thread scheduler, and may therefore be used to comparatively measure the performance benefit offered by the latter.

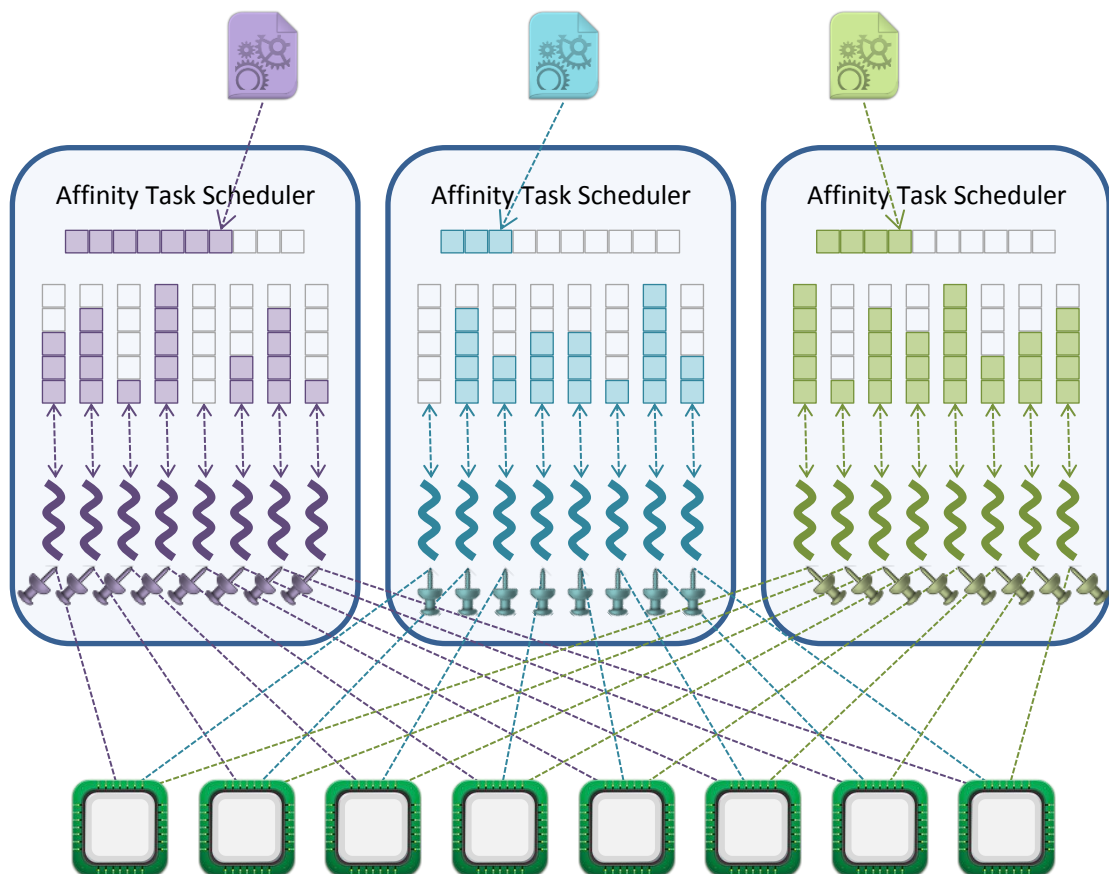


Figure 5.7. Oversubscribed pinned multiprogramming

5.2.3 Shared-Scheduler Multiprogramming

A drawback of thread oversubscription is that it incurs performance penalties whenever the thread scheduler needs to context-switch among a processor's threads, as discussed in Section 2.3 (p. 15). If the oversubscribed threads belong to different programs, they would not share any data. Rather, they would suffer from destructive interference due to their contention over the limited cache space, causing each other's data to be evicted, further exacerbating the cost of the context switch due to the subsequent cache misses.

This behaviour is undesirable, especially when one considers that load-balancing is already provided for by the work-stealing task scheduler (residing further up in the system runtime architecture stack). Thus, rather than creating a distinct task scheduler with a plethora of threads per program, we could improve performance by creating a single task scheduler that is shared among all programs in the multiprogram workload. In practice, this simply means that all the D&C skeleton instances representing our programs would insert their top-level tasks into the same global queue, wherefrom they may get picked and distributed among worker threads per the scheduling logic of the task scheduler.

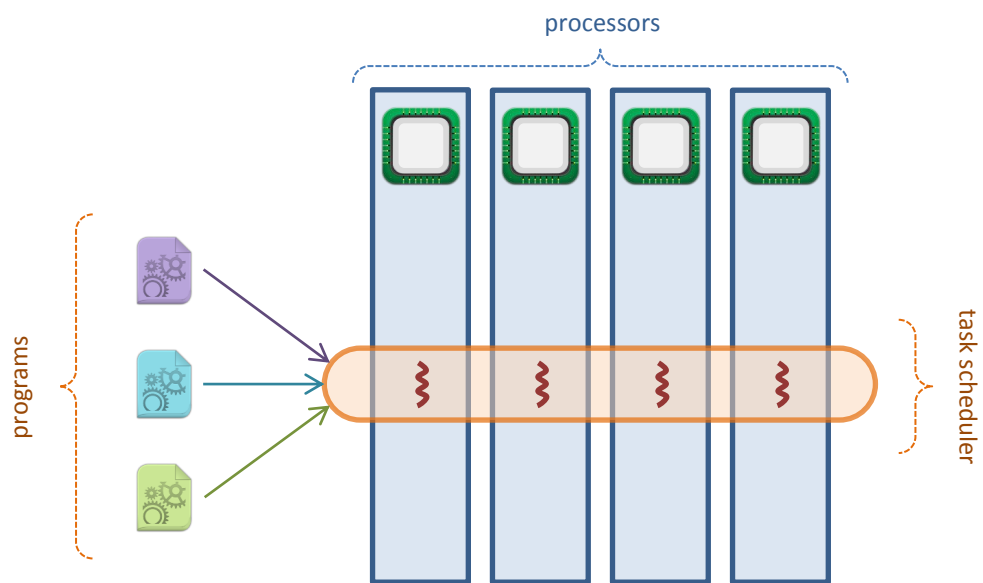


Figure 5.8. Programs feeding a shared task scheduler

Thus, in the “shared-scheduler multiprogramming” scheme, a single `AffinityTaskScheduler` is created for the entire test, initializing a single pinned thread on each core. Each program issues tasks to this same shared task scheduler, as shown in Figure 5.8 and Figure 5.9.

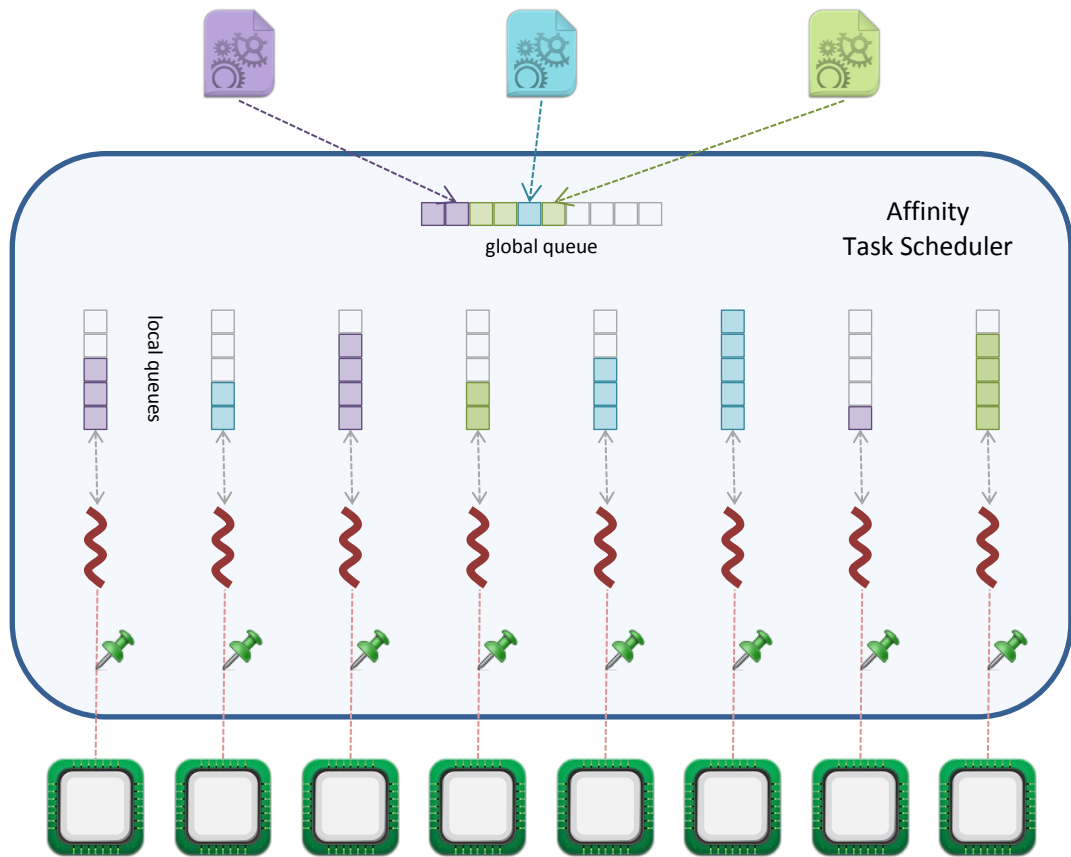


Figure 5.9. Shared-scheduler multiprogramming scheme

5.2.4 Default-Scheduler Multiprogramming

In the “default-scheduler multiprogramming” scheme, all programs issue their tasks to the default task scheduler provided for TPL by the .NET Framework or Mono (described in Section 3.1.2, p. 22). Since this default scheduler is designed to run as a singleton, all programs would be issuing their tasks to the same single scheduler. Thus, this scheme is similar to the shared-scheduler scheme discussed in the previous section, except that the default task scheduler’s implementation is likely to be better optimized, does not perform thread pinning, and has the liberty of employing thread injection to gradually spawn new worker threads (thereby oversubscribing some processors) when it deems fits.

5.3 Multiprogramming Task Schedulers

In Section 2.3.1 (p. 15), we discussed the performance benefits that may be reaped from pinning each program's threads onto a distinct subset of the machine's processors. In the case of task parallelism or structured parallelism, programs do not have direct control over the threads executing their computations; rather, these would be managed by the underlying task scheduler, which commands a pool of worker threads for servicing its task queue(s), as had been depicted in Figure 2.3 (p. 12).

5.3.1 Scheduler Partitioning

The crux of our project concerns the reconciliation of task parallelism with processor partitioning, as illustrated in Figure 5.10. In doing so, we are transcending one level of abstraction from traditional research in the latter area, which typically only considers explicit thread parallelism [13].

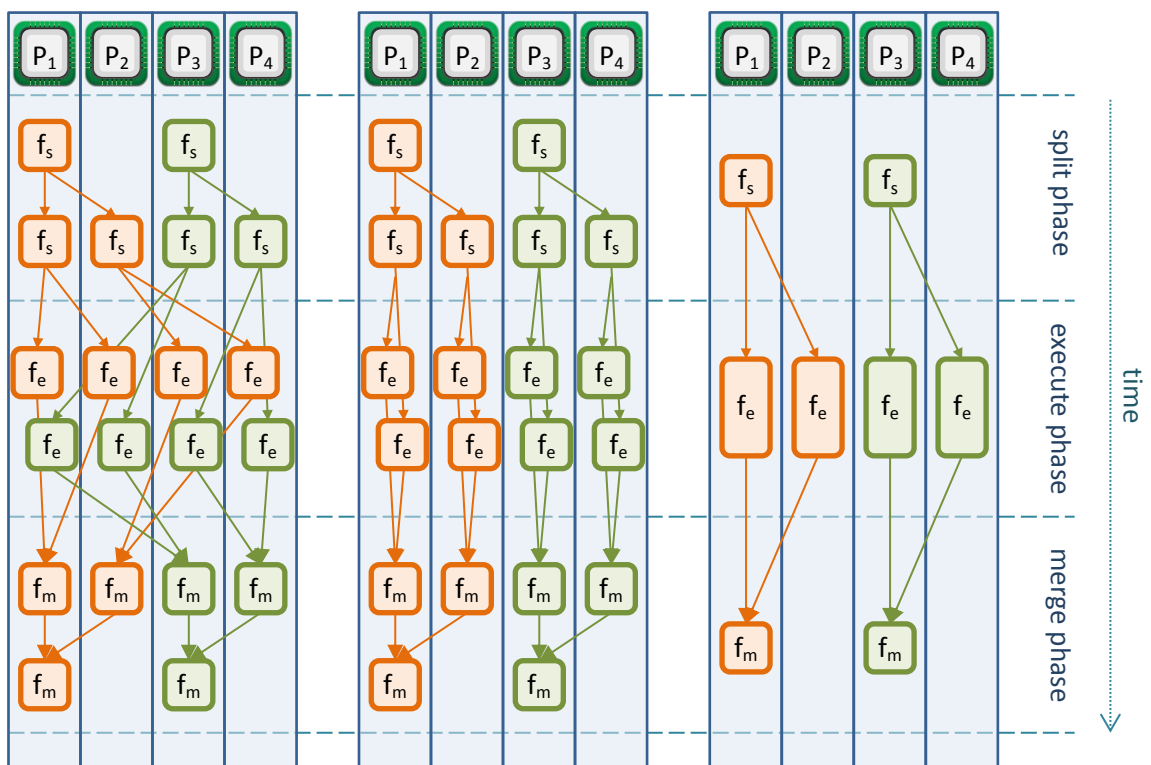


Figure 5.10. Reconciliation of task parallelism with processor partitioning. This diagram depicts a pair of D&C programs being executed as: spread across all processors (left); allocated to dedicated subsets of the processors (centre); and having their granularity adjusted (right). The latter optimization was a direction we explored early in our project, but decided not to pursue since its effects were either marginal or counterproductive, as reported in Section 7.1 (p. 81).

In its most basic form, task-parallel partitioning can be achieved by initializing a dedicated task scheduler for each program, with its worker threads pinned to a distinct subset (or “partition”) of the processors, as shown in Figure 5.11.

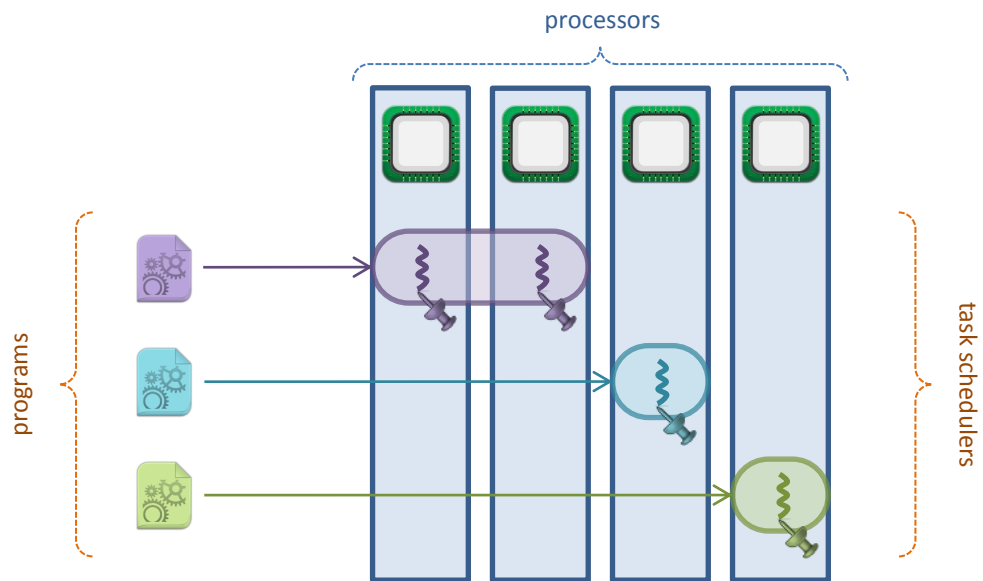


Figure 5.11. Processors partitioned among task schedulers servicing distinct programs. In this example, the topmost program is allocated two processors, whilst the others are only allocated one processor each.

Each task scheduler internally embodies work-stealing logic for enacting efficient load-balancing among its allocated processors (see Figure 5.12 below). Therefore, there is no need for thread oversubscription, permitting us to avoid the overheads of thread context-switching altogether.

Section 5.1.2 (p. 56) mentioned that our `AffinityTaskScheduler` class can take a sequence of processor identifiers as a parameter to its constructor, allowing one to specify the exact set of processors over which it should spawn pinned worker threads. In a multiprogrammed context, this permits complete control over the allocation strategy, making it possible to experimentally attempt to identify the best-performing configuration for a given workload.

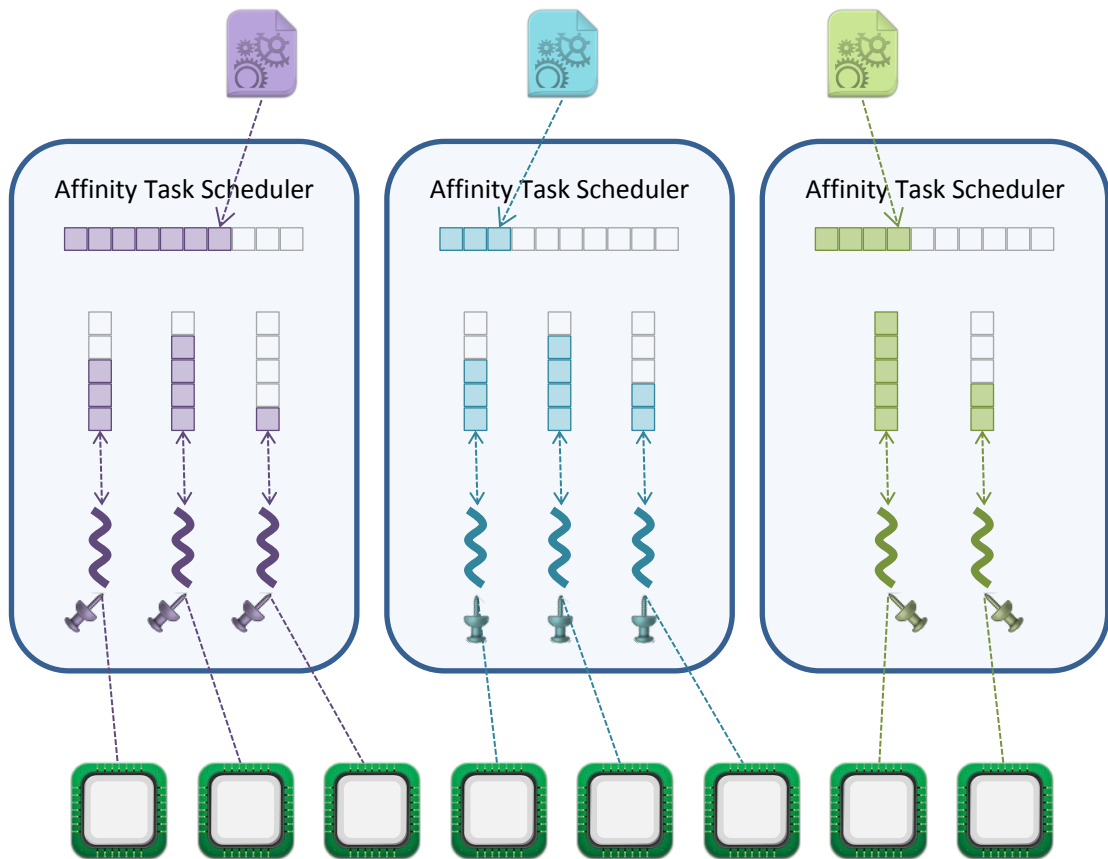


Figure 5.12. Task schedulers can enact processor partitioning by pinning their respective worker threads onto distinct subsets of the machine’s processors

5.3.2 Scalability-Based Partitioning

Whilst the flexibility offered through the explicit initialization of `AffinityTaskScheduler` instances is suitable for experimentation, we wanted to design a solution that could come up with performant allocations heuristically for any given multiprogram workload, insulating the application developers from the responsibility of this decision. We build on the direction taken by Sasaki et al. [13], as suggested in Section 2.3.2 (p. 17) and detailed in Section 3.2 (p. 28), and use program scalability as a metric for guiding the processor partitioning decisions.

For this end, we construct a `MultiProgramScheduler` class (see Figure 5.13 below). This class takes an arbitrary multiprogram workload, and infers each program’s scalability to come up with a suitable allocation for the entire workload.

Sasaki et al. [13] populate a scalability table that records each program's speedups on various numbers of cores, and subsequently apply a hill-climbing algorithm against it to arrive at the allocation that should minimize the workload's average normalized turnaround time (ANTT). Our MultiProgramScheduler uses a simplified version of this heuristic that assumes each program's scalability to be a single numeric value corresponding to its parallel speedup when executed in isolation over the entire machine. This way, it would only need to measure each program's sequential execution time and its fully-parallel execution time (both averaged across multiple runs), and compute their ratio. We feel that this simplification was justified in our scenario since our programs scale much more cleanly than the PARSEC benchmark suite used by Sasaki et al. (For a graphical representation of our actual scalabilities, refer to Section 7.2, p. 84.) In particular, our speedups always increase monotonically with higher concurrencies, unlike PARSEC, which experiences slowdowns in some cases [13].

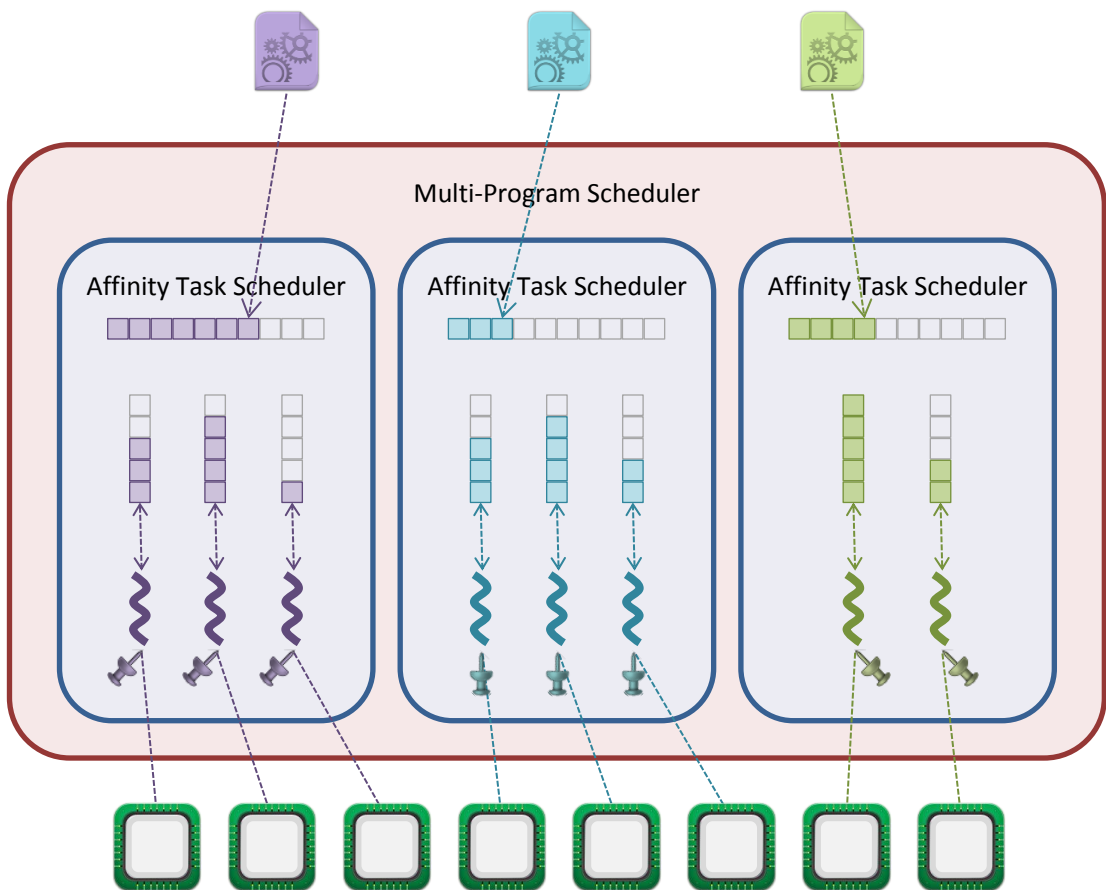


Figure 5.13. Statically-partitioned multiprogramming. The multiprogram scheduler internally initializes an affinity task scheduler for each program, with pinned worker threads spanning its allocated processor range.

Once all these measurements have been taken, the programs are respectively allocated a distinct contiguous range of processors whose size is proportionate to their scalability. For example, given a workload of programs that individually exhibit 23×, 13×, and 12× speedups on 64 cores, their multiprogrammed allocations would be 31, 17, and 16 cores respectively. The contiguous allocation boosts proximity for processors allocated to the same program, increasing the likelihood that they would share at least the L3 cache.

`MultiProgramScheduler` does not derive from `TaskScheduler`, and therefore does not handle task scheduling directly. Instead, it internally creates an `AffinityTaskScheduler` for each program, initialized with worker threads pinned to its respective processor allocation, as shown in Figure 5.13 (above). This setup constitutes our “statically-partitioned multiprogramming” scheme.

5.3.3 Dynamic Repartitioning

The multiprogram scheduler implementation described so far only performs static partitioning, with processor allocations needing to be established *before* the multiprogram workload commences being executed. In practice, this approach is inadequate, not least because it cannot support the introduction of new programs into the workload after it has commenced. Additionally, programs that complete execution cannot have their processors reallocated to other ongoing programs, severely compromising the system’s overall performance as the said processors remain unutilized. Finally, programs undergo various phases along their execution, as observed by Sasaki et al. [13], making it unlikely that the initial allocation is suitable for the entirety of their execution.

In order to address the above issues, we extend our `MultiProgramScheduler`’s design to support dynamic processor reallocation across the task schedulers dedicated to the various programs in the current workload. In our initial implementations, we worked at the thread level: A processor reallocation is performed by shutting down the pinned worker thread from the old task scheduler, and firing up a new one to replace it in the new task scheduler, as depicted in Figure 5.14 (below).

The main appeal of this approach is the simplicity with which it builds upon the `AffinityTaskScheduler` architecture. However, it incurs thread creation overheads for each processor reallocation, succumbing to the same performance degradation that task parallelism was designed to avoid.

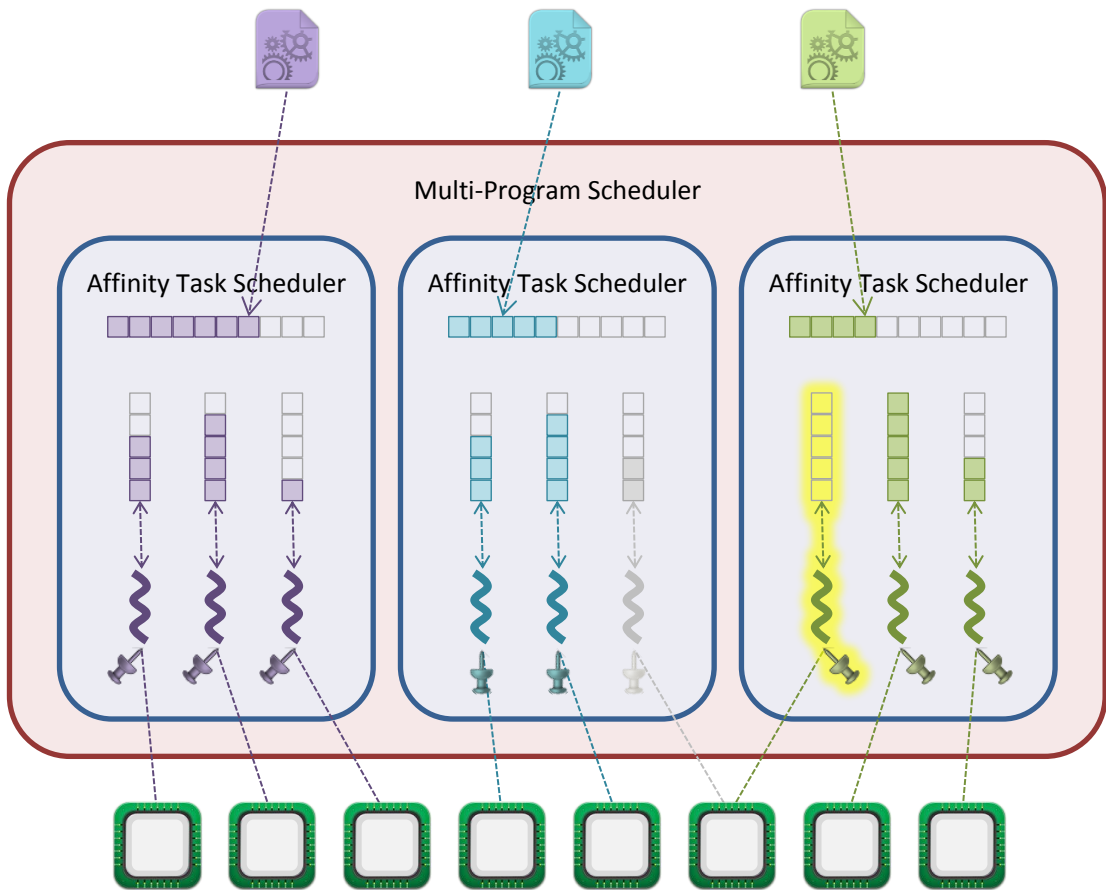


Figure 5.14. Thread-based processor reallocation, showing the termination of the associated pinned thread from the centre scheduler's pool, and the initialization of its replacement in the right scheduler's pool.

Therefore, we endeavoured to eliminate these threading overheads and come up with a mechanism that can reallocate processors efficiently, which brings us to the culmination of our system's structural design. We take the novel step of decoupling the worker thread pool from the task queue superstructure that it services. Each program is still associated with a dedicated task scheduler that contains a global queue and per-thread local queues, ensuring that work-stealing remains localized per program. However, the thread pools are extricated from their fragmented distribution across the various task schedulers, and instead combined into a unified pool managed by the workload-wide `MultiProgramScheduler`, as shown in Figure 5.15 and Figure 5.16 (below).

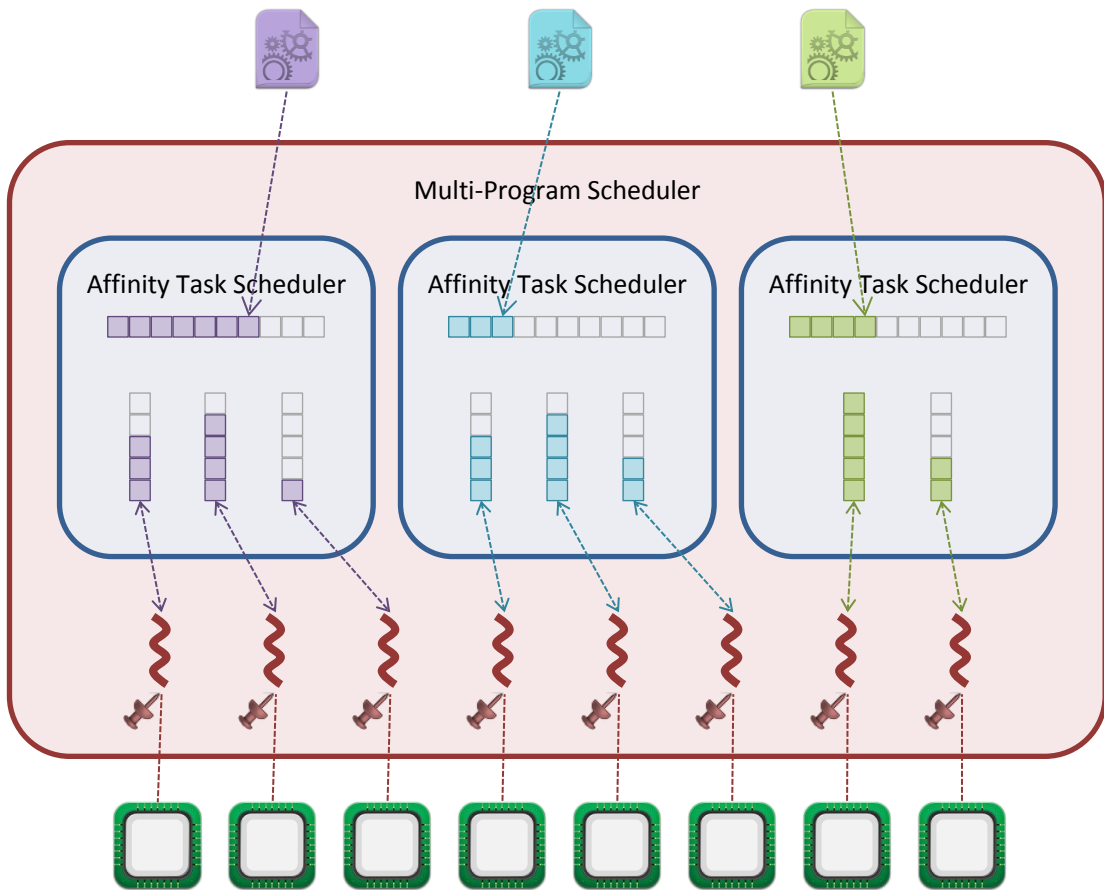


Figure 5.15. Decoupling of thread pool from task queue superstructure. This design drastically simplifies the procedure for processor reallocation.

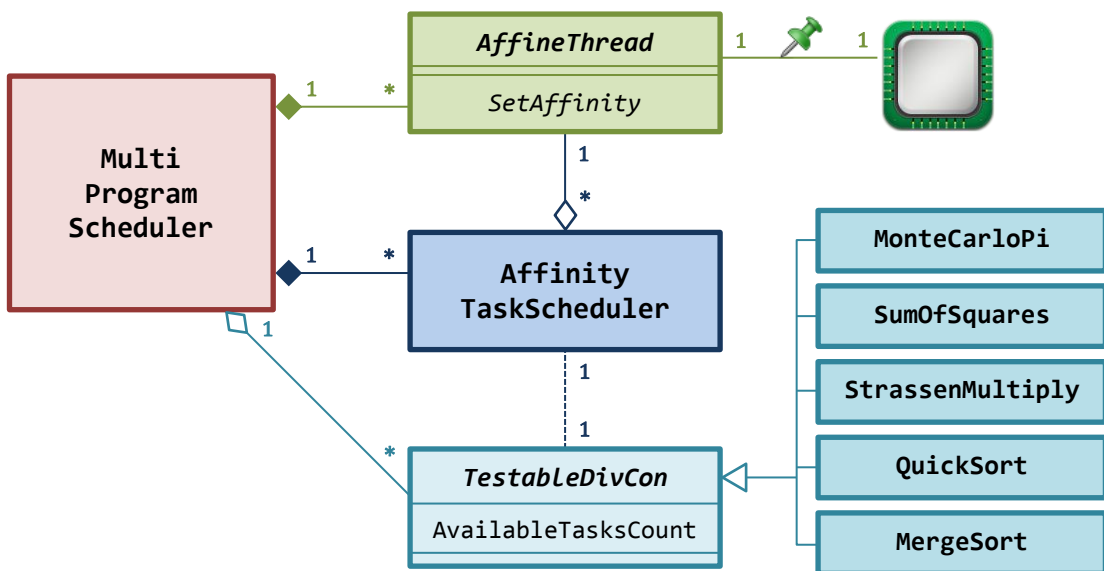


Figure 5.16. Constitution of the multiprogram scheduler. This class diagram unites the D&C program hierarchy from Figure 4.9 (p. 39) with the scheduler hierarchy from Figure 5.1 (p. 55). Note that each affinity task scheduler now only *aggregates* (rather than *composites*) its affine threads.

Figure 5.16 (above) shows that the multiprogram scheduler initializes a single thread pool, with one worker thread pinned to each logical core. For each task-parallel program, it initializes a dedicated affinity task scheduler, which will dynamically be assigned a subset of the worker threads according to the scalability and task availability of its associated program.

The assignment of worker threads to task schedulers is managed through a mapping maintained dynamically by the MultiProgramScheduler instance. Consequently, reallocating a processor from one program to another becomes a simple matter of reassigning its pinned thread across their respective task schedulers. This is a lightweight user-space operation, allowing threads to dart in and out of the various programs' queue superstructures without incurring system-level overheads.

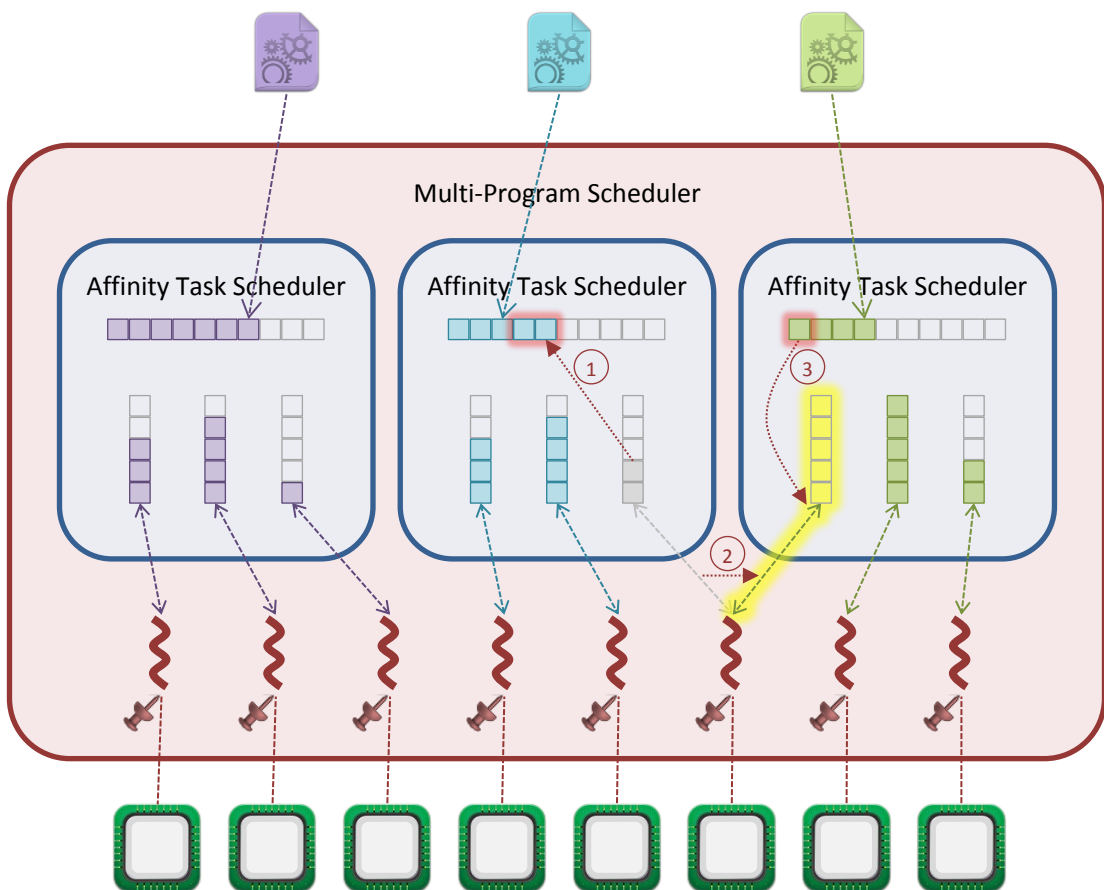


Figure 5.17. Transitioning a worker thread across task schedulers. The procedure involves:

- ① relegating pending tasks from the local queue back onto the old global queue;
- ② reassigning the thread to the new task scheduler;
- ③ picking up a task from the new global queue.

Understandably, this transition mandates a number of task-level considerations in order to be performed smoothly, without disrupting the task infrastructure's expected behaviour. The most important restriction is that an active task cannot be migrated across threads once it has started executing, since this would otherwise violate "thread-affine abstractions", such as critical sections (which are meant to be entered and exited on the same thread) [34]. Thus, we permit our worker thread to complete its current task before commencing the transition.

Once the current task completes, the thread dumps any pending tasks from its local queue back into the global queue of its former program's task scheduler, in order for them to subsequently be eventually picked up by the other threads still assigned to it. The transferred thread then assumes a new local queue associated with the target scheduler, and commences execution by fetching the first task from its global queue. This entire procedure is outlined in Figure 5.17 (above). (The new local queue would then be gradually populated with new locally-pushed tasks from the worker thread itself, per the usual logic originally depicted in Figure 3.5, p. 27.)

5.3.4 Task-Availability-Based Repartitioning

In the previous section, we have described our structural design for enabling dynamic processor repartitioning. In the spirit of the design principle promoting the separation of mechanism and policy, we have architected this functionality such that it can be invoked to accommodate the decisions made by any repartitioning strategy. We shall now proceed to describe the policy we enacted for dynamically inferring performant allocations.

Due to the nature of the D&C skeleton, a program's potential for parallelism is bounded by its current recursion depth (as was discussed in Section 4.1.2, p. 31). For example, a program executing its initial split or final merge would only have a single available task, and may therefore only utilize one processor. A program's potential parallelism increases exponentially with each level of recursion, based on the program's branching factor (which is 2 for quicksort and mergesort; 7 for Strassen multiplication; and any arbitrary number for map-reduce and Monte Carlo Pi). A simplified depiction of this effect is given in Figure 5.18 (below).

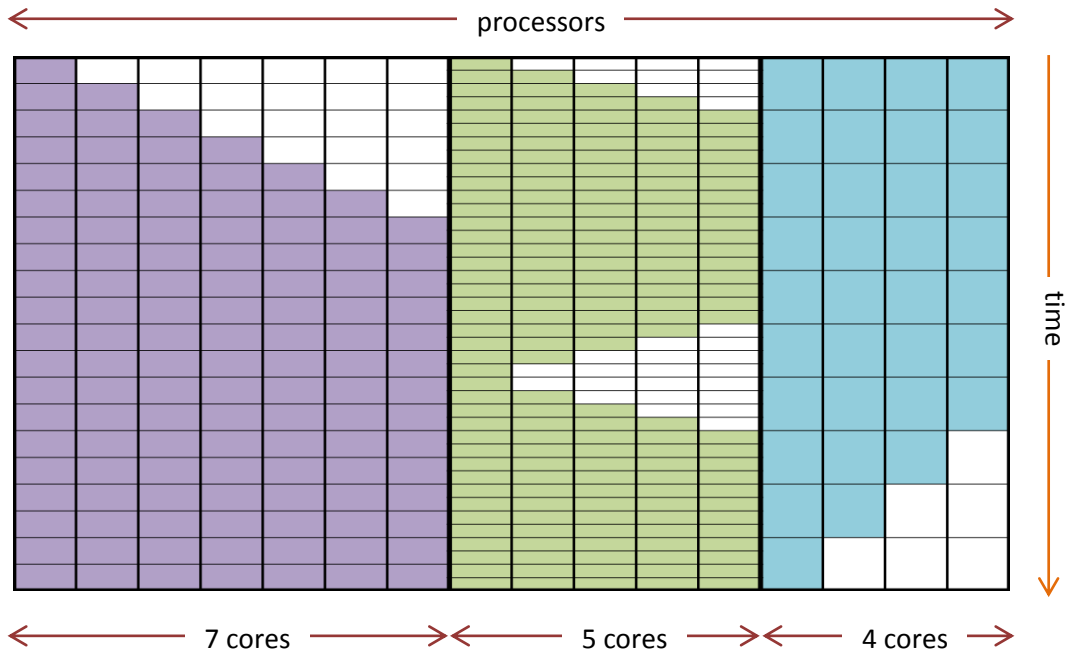


Figure 5.18. D&C execution under static partitioning. Each colour represents a D&C program, whilst each coloured box represents the execution of one of its tasks. Uncoloured boxes indicate processors that remain unutilized due to the curbed parallelism inherent in D&C algorithms.

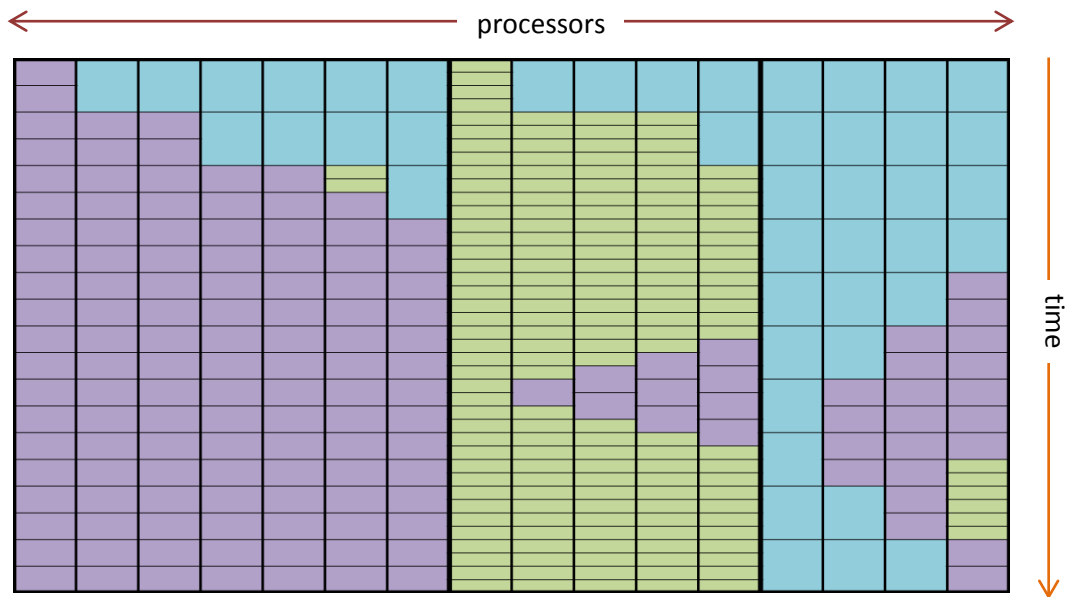


Figure 5.19. D&C execution under task-availability-based repartitioning. All processor slots are now utilized.

Notwithstanding our focus on the D&C skeleton, we strived to come up with a policy that could adapt to task parallelism in general. As the basis of our policy, we still use the scalability measure described in Section 5.3.2 (p. 67); this gives us the base allocations. Then, at regular intervals – say, every 100 ms – our system queries each parallel program in the workload for its current number of available tasks. Using this collated information, the system can identify any processor allocations that exceed their program’s current number of available tasks, and reallocate them to other programs that have excess tasks available, thereby boosting overall utilization. Figure 5.19 (above) shows the effect of this policy on a cyclically-executed set of D&C programs.



Speedup on 32 cores	15.18	10.43	9.71
Scalability ratio	0.43	0.30	0.27
Proportional share	13.75	9.45	8.80
Base allocation	14	9	9

Tasks available	18	1	12
Capped allocation	14	1	9

Active allocation (with updated score):

Remaining: 8	14 (1.018)	1 (0.106)	9 (1.023)
Remaining: 7	15 (1.091)	1 (0.106)	9 (1.023)
Remaining: 6	15 (1.091)	1 (0.106)	10 (1.137)
Remaining: 5	16 (1.163)	1 (0.106)	10 (1.137)
Remaining: 4	16 (1.163)	1 (0.106)	11 (1.250)
Remaining: 3	17 (1.236)	1 (0.106)	11 (1.250)
Remaining: 2	18 (1.309)	1 (0.106)	11 (1.250)
Remaining: 1	18 (1.309)	1 (0.106)	12 (1.364)

Remaining: 1	18 (1.309)	1 (0.106)	12 (1.364)
Remaining: 0	18 (1.309)	2 (0.212)	12 (1.364)

Figure 5.20. Heuristic for reallocating processors among programs. Green-striped entries indicate the program selected for receiving another processor at the end of the current step. Red-striped entries indicate programs not eligible to receive further processors during this stage.

Figure 5.20 gives a sample dry-run of the heuristic we use for dynamic repartitioning. The first (topmost) part of the table shows the static (scalability-based) partitioning that is

performed when the workload is initially introduced; this gives the base allocations. In the second part, we check how many tasks are available in each program, and cap their allocations accordingly – this would require $O(m)$ processing overall for m programs.

In the third part, we introduce the notion of a “score” (displayed in parentheses), which is taken to be the ratio of a program’s current allocation to (what would have been) its proportional share (according to its scalability). This score is indicative of a program’s relative eligibility for receiving more processors, with lower scores meriting higher precedence. Thus, for each available processor, we pick the program with the lowest score, and increment its current allocation by one. The influence of scalability on this procedure is seen at the steps where there are 4 and 3 processors remaining, with the allocation in both consecutive cases going to the leftmost program, due to its superior scalability.

Initially, we only allow processor allocations that do not exceed the target program’s number of available tasks. However, if this limit is reached for all programs (due to an insufficiency of tasks on a workload-wide level), we proceed to the fourth part, where we apply the same procedure, but without the task-availability bound.

This reallocation heuristic is implemented efficiently using a heap structure; thus, reallocating n processors to m programs only requires a total of $O(n \log m)$ steps, plus some $O(n)$ readjustments at the end to promote contiguity among the reallocations. Furthermore, it is only applied when there actually exist programs in the workload that have insufficient tasks; in all other cases (which should constitute the vast majority), the repartitioning scheduler would only incur the initial $O(m)$ check.

The efficacy of this strategy is equally applicable to scenarios where the workload varies dynamically due to programs terminating, or new ones being introduced, at arbitrary points in time. A program that terminates would have its processor allocation completely redistributed among the remaining programs; similarly, a new program would be granted some processors from the other programs according to its relative scalability.

This setup thereby constitutes our “dynamically-repartitioning multiprogramming” scheme: A single `MuLtiProgramScheduler` is used to initialize a pinned thread on each processor, but maintains a distinct `AffinityTaskScheduler` with work-stealing task queues for each program, dynamically reallocating threads to programs based on their scalabilities and numbers of available tasks.

5.4 Multiprogramming Test Strategies

To conclude our design chapters, we shall present the main test harness that we implemented for evaluating the efficacies of the various task schedulers and multiprogramming schemes.

In our initial experiments, we used to constrain our tests such that each program in the multiprogram workload was only run once. Once all programs were ready, the overall execution time was recorded, the programs collectively reinitialized, and then another test started. However, this setup was not appropriate for system-level performance metrics, since the last program to complete would effectively be executing in single-program mode, causing results to get skewed by the longer-running programs.

5.4.1 Cyclic Tests

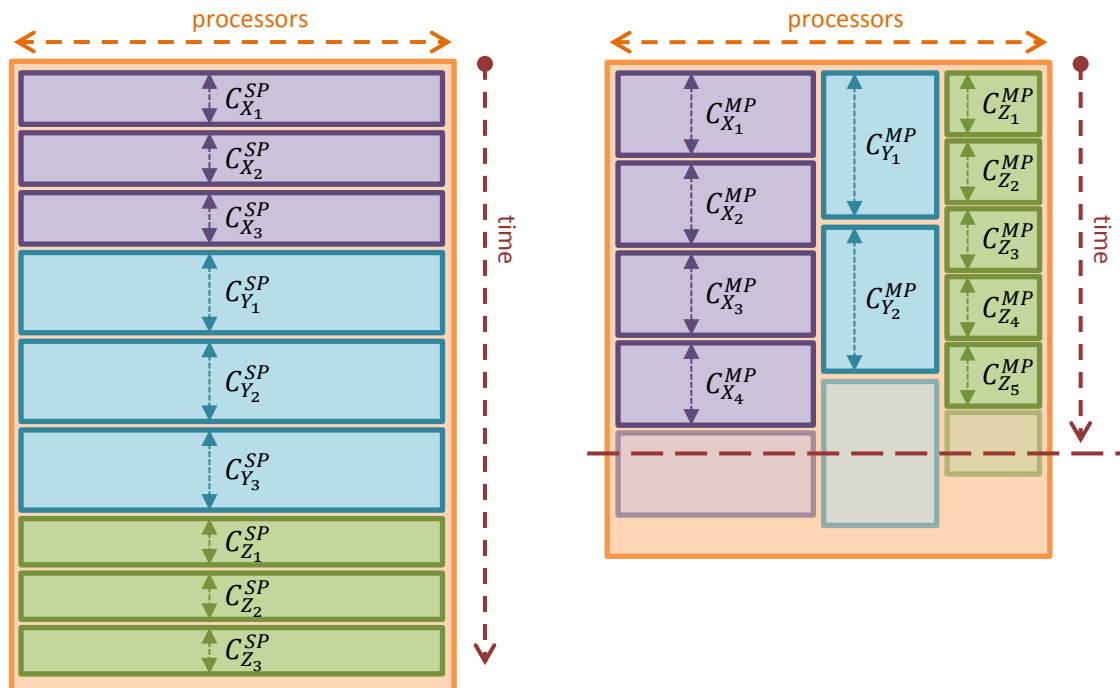


Figure 5.21. Cyclic test harness. $C_{X_1}^{SP}$ denotes the measured execution time of the 1st run of program X in single-program mode; similarly for $C_{X_1}^{MP}$ in multi-program mode.

Once we eliminated the issue of interference arising from program reinitialization (see Section 4.5.1, p. 49), we could implement a cyclic test harness to accurately measure the programs' performances, as depicted in Figure 5.21 (above). First, the single-program

execution time, C^{SP} , is measured for each program by running it in isolation over the entire machine's processors. The test is run 5 times for each program, and the average taken.

For measuring the multi-program execution times, C^{MP} , the test engine initializes the problem data for all the programs, starts the timer, and launches the programs. As soon as any program completes, the time elapsed is recorded, and the program immediately restarted (whilst the others are still executing). This process is repeated until the timer reaches a specified limit – after which, all subsequent timings are discarded (and programs no longer restarted). The number of times that each program gets to run varies; however, its multi-program execution time is taken as the average of all its measured timings.

Once we have all the above measurements in place, we can calculate each program's normalized turnaround time (NTT) and normalized progress (NP) per the definitions given in Section 2.4 (p. 17), aggregating them to get the system-level performances.

Chapter 6: Experimental Setup

6.1 Hardware Platform

We perform our experiments on a Dell [PowerEdge C6145](#) machine, consisting of two processor-based servers. Each shared-memory server has four sockets with AMD [Opteron 6276](#) “Interlagos” processors,²⁴ each containing 16 cores, thereby giving a total of 64 cores per server. Each core runs at a base frequency of 2.3 GHz and has a dedicated 1 MB L2 cache. Each processor shares a 16 MB L3 cache among its cores, and accesses main memory via four HyperTransport 3.0 links, each having a peak bandwidth of 6.4 GT/s. Each server is equipped with 128 GB of DDR3 memory.

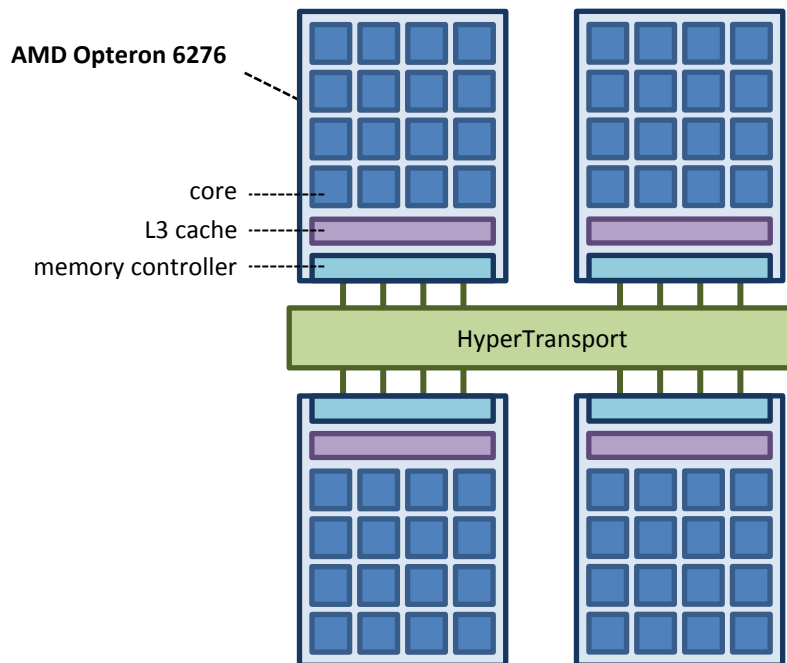


Figure 6.1. Processor setup in the manycore servers. Each server consists of 64 cores organized into four 16-core processor chips.

²⁴ Note that the term “processor” is used ambiguously across the industry. AMD considers each chip of eight cores to collectively be a single processor, whilst software companies tend to treat each core as a “processor” in its own right, as evidenced in “processor affinity” discussions. As indicated in Section 1.4 (p. 4), we have assumed the latter convention throughout this document.

When submitting jobs (using the Sun Grid Engine) onto the manycore servers, we reserve *all* their cores for the entire duration of our experiment, even in the case of tests utilizing limited concurrency, thereby ensuring exclusive machine access. This way, we eliminate the risk of interference from other users or programs, allowing us to obtain reproducible results.

6.2 Software Platform

Our PowerEdge manycore has [Scientific Linux 6.3](#) (running Linux kernel version 2.6.32 for x86-64) installed as its operating system. Microsoft only targets the .NET Framework for Windows platforms; thus, we will instead use [Mono](#), an open-source cross-platform implementation of the framework.²⁵ We initially used the latest stable release for Linux, [Mono 2.10.8](#); however, this version suffered from severe performance issues when multithreading on multiprocessor machines (including substantial *ad hoc* variance in the execution times of identical sequential codes), as documented in our investigation posted to StackOverflow under "[Mono multiprocessing performance issue](#)". We subsequently upgraded to the latest beta release, [Mono 3.0.12](#), which seems to largely resolve the issue through its new [SGen](#) garbage collector; we use this version for all our experiments.²⁶

6.3 Statistical Methods

In order to obtain sensible results that would allow us to induce sound conclusions, each test is run repeatedly (at least 10 times), and the execution times measured across all runs averaged. Some tests engender such repetition intrinsically, such as the multitude of C^{MP} readings given by each cyclic multiprogram test (as explained in Section 5.4.1, p. 76). For all other tests, we authored a small script to submit multiple instances of the job to the manycore machine (for consecutive execution), and then aggregate their results.

²⁵ The procedure for compiling Mono is described in the "[Compiling Mono From Tarball](#)" article on the official Mono website.

²⁶ A new major stable release, [Mono 3.2.0](#), was published on 24 July 2013. Whilst offering modest performance gains on commodity machines, we found that this version suffered from stability issues on Morar, frequently causing the job to hang, and therefore refrained from upgrading to it.

Apart from the averages, we also calculate the standard deviation among the execution times of the runs for each test. Standard deviation indicates the extent of the variation, thereby imparting important insight about the performance's consistency. Since our tests only constitute a "sample" of runs, we use the formula for sample standard deviation, applying Bessel's correction. Based on this, we compute the confidence intervals for our measured averages, assuming a Student's t -distribution and targeting a 99% confidence level. We represent the confidence intervals graphically, using error bars, in all our charts in Chapter 7 (below).²⁷

²⁷ For definitions of these terms, refer to published literature on statistics, such as *Statistics for Experimenters: Design, Innovation, and Discovery* by Box et al. [52].

Chapter 7: Experimental Results and Analysis

In this chapter, we present the results of the experiments through which we evaluated the various aspects pertaining to our investigation, including the effects of task granularity, hardware concurrency, program scalability, and multiprogramming schemes. Each experiment is accompanied by a critical analysis that attempts to systematically explain the causes underlying the results, as well as compare them to related work (where applicable).

7.1 Granularity

In Section 4.5.2 (p. 50), we explained the notion of granularity and its influence on the extent of potential parallelism yielded by our D&C skeleton. In our first experiment, we investigate the effect of this granularity on parallel performance. We run each respective D&C program (with a fixed problem size) in isolation over all 64 cores, starting with a granularity equal to the problem size, and halving it in each subsequent test, down to $1/65,536$ (i.e. 2^{-16}) of the problem size.

As discussed in the aforementioned section, this ratio would typically correspond to the total number of execute muscle tasks that the D&C skeleton ends up with at the base case of the recursion. Thus, it serves as an indicator of the potential parallelism presented by the D&C program to the task scheduler. (In quicksort, this number is only a close approximation due to its unbalanced split. In Strassen multiplication, the computation is more intricate: ratios of $1/2 - 1/4$ yield 7 execution muscles; ratios of $1/8 - 1/16$ yield 49 execution muscles; whilst ratios of $1/32$ or lower yield 343 execution muscles, with `ShouldSplit` explicitly overridden to stop at this limit in the case of this particular program.)

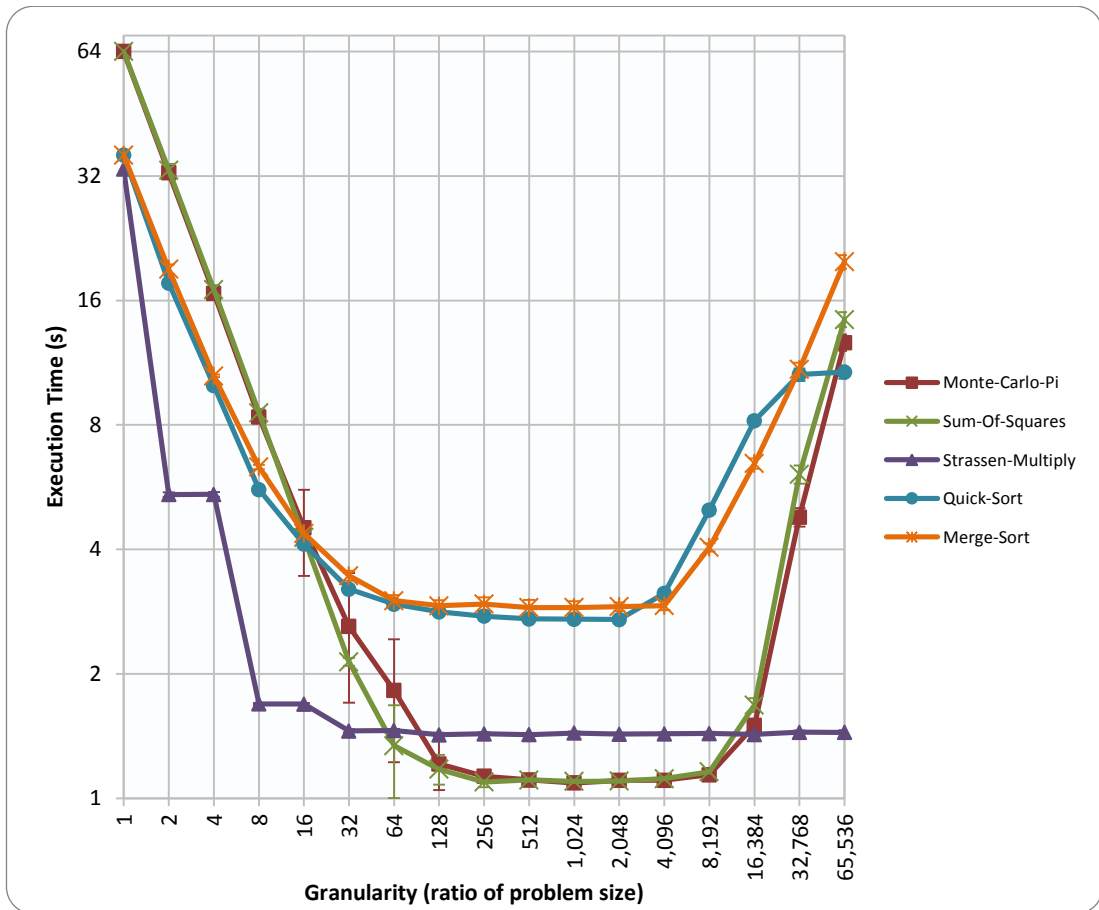


Figure 7.1. Execution time vs. granularity for our D&C programs, run independently over 64 cores. A performance optimum emerges prominently towards the centre.

Figure 7.1 presents our results. Very coarse granularities expectedly suffer from poor performances due to their curbed opportunity for parallel execution, thereby succumbing to the consequences of Amdahl’s Law [47]. As the granularity is made finer (yielding more parallel tasks), performances improve almost uniformly, with full machine utilization initially achieved at a granularity ratio of $1/64$. However, this granularity generates as many execute tasks as there are logical cores on the machine, leaving no room for compensating against variances in their respective execution times. Consequently, the overall completion time of the execute phase becomes bound to its slowest task.

This effect is most clearly demonstrated in the case of Monte Carlo Pi, where the substantial variance among its tasks’ execution times (as indicated through its large error bars) causes its performance on 64 execute tasks to be 67% slower than its best

performance on 1,024 tasks.²⁸ In the case of all programs, finer granularities mitigate this issue by creating an abundance of such tasks, allowing threads assigned faster-executing tasks to restore balance by processing more tasks overall.

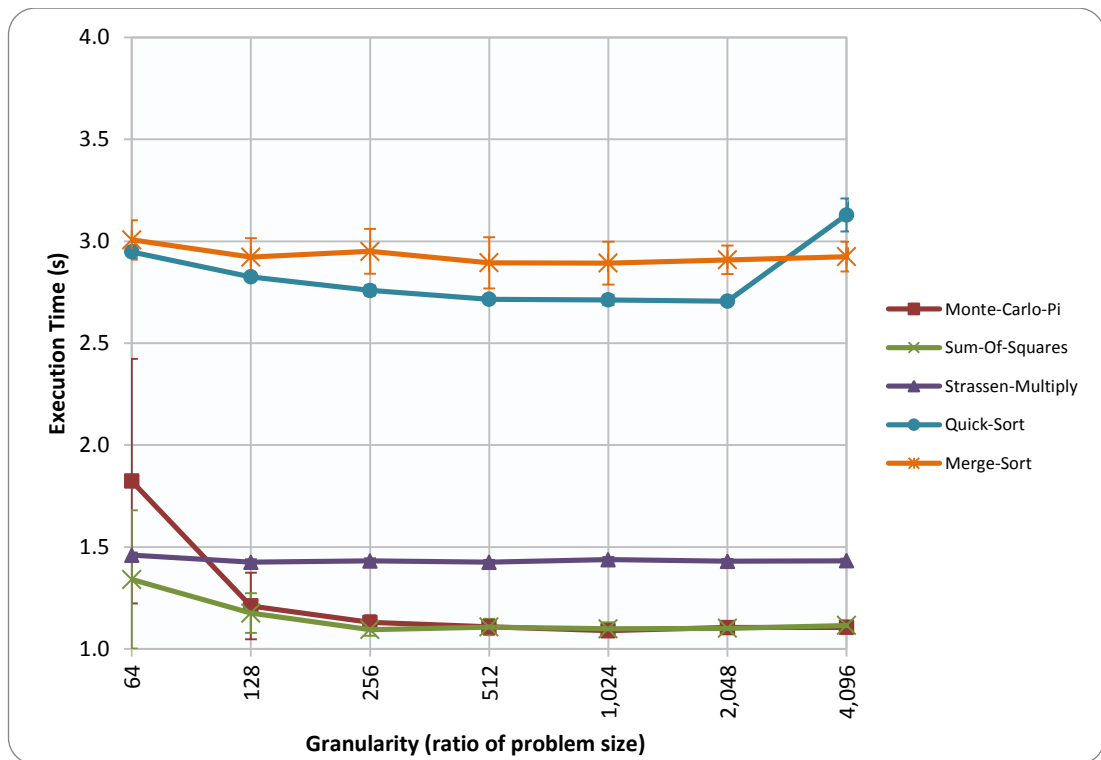


Figure 7.2. Execution time vs. granularity, showing region of best performances. Note how the performances remain largely consistent over the wide range of granularities. Unlike Figure 7.1, this chart’s vertical axis has a linear scale.

The most interesting outcome from this experiment, as amplified in Figure 7.2, is that the good performances subsequently persist over a wide range of ever-finer granularities, only starting to deteriorate at ratios of $1/4,096$ and beyond. This result testifies to the efficiency of task parallelism, demonstrating that it can accommodate surplus parallelism of up to $32\times$ the machine’s multiprocessing capabilities without experiencing any perceptible slowdowns. By comparison, informal benchmarks show that thread oversubscription by a factor of just $2.5\times$ can cause a 40% slowdown for compute-bound multithreaded applications [32].

²⁸ This large variance among the execute tasks of the Monte Carlo Pi program is unexpected. As mentioned in Section 6.2 (p. 79), the cause seems to be performance deficiencies in Mono’s implementation of its garbage collector on multiprocessing systems, despite that our Monte Carlo Pi muscle function does not instantiate any objects on the heap during its execution.

At granularities of $\frac{1}{4,096}$ and finer, the task parallelism overheads swamp the computation and cause performance to deteriorate, eventually reaching a point where the parallel programs would run slower than their sequential counterparts.

On the basis of these results, we configured our system to aim for granularities that would result in approximately 4 execute muscles per core for each program (as discussed in Section 4.5.2, p. 51). Whilst Figure 7.2 (above) suggests that the best performances are achieved when targeting 16 execute muscles per core (as similarly recommended by Campbell et al. [34]), this only applies when the programs are run in isolation, leading us to scale it down in anticipation for the multiprogramming tests.

7.2 Scalability

In our next experiment, we evaluate our parallel programs' respective scalabilities by measuring their speedups when executed (in isolation) over various degrees of physical concurrency. For each program, we first measure the sequential execution time, which involves processing the entire problem through a single `ExecuteMuscle` call (run on a single core). Then, we initialize an `AffinityTaskScheduler` whose number of pinned threads (over distinct contiguous cores) is gradually increased over consecutive tests, up to a maximum of 64 (representing full utilization of our manycore machine). We measure the parallel execution times, and obtain the speedups by calculating the factor by which they reduce the sequential execution time (as defined in Section 2.4, p. 17).

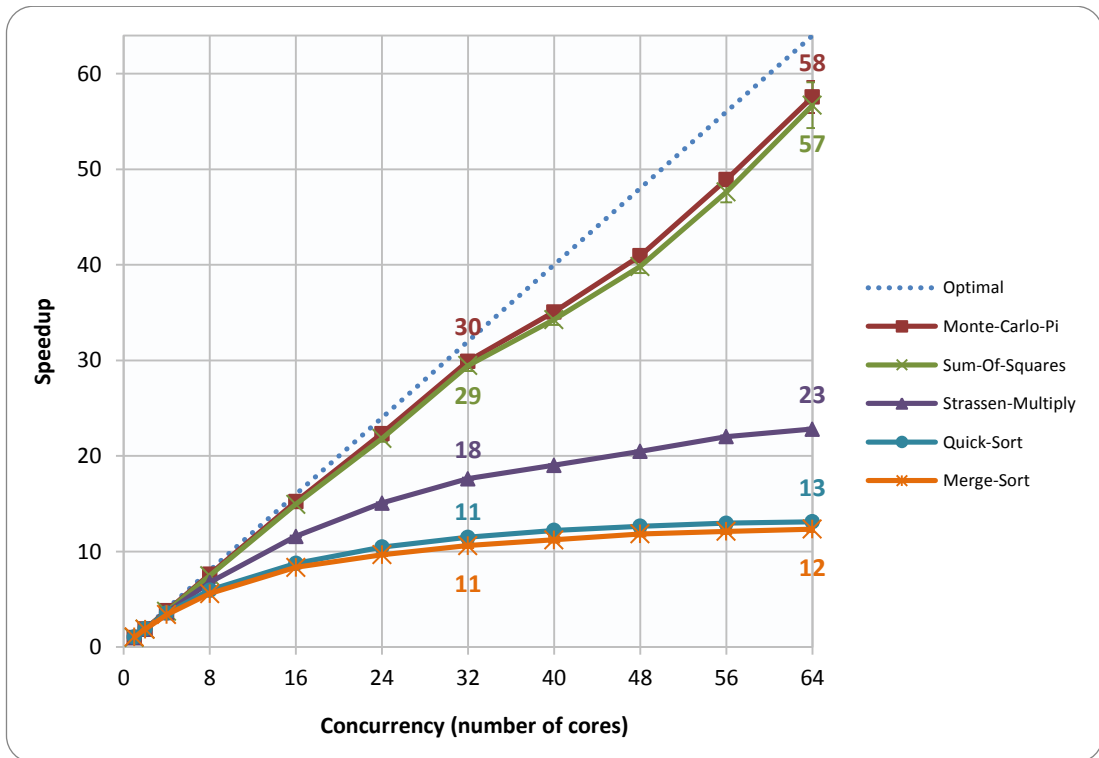


Figure 7.3. Speedup vs. concurrency for our parallel programs executed over various numbers of contiguously-allocated cores on the 64-core machine

Figure 7.3 presents the observed scalabilities, with results corroborating the predictions we made throughout Section 4.4 (p. 41). Our Monte Carlo Pi and sum-of-squares (map-reduce) programs exhibit near-linear speedups, achieving parallel efficiencies of 90% and 89% (compared to the optimal) on 64 cores. Both of these programs are embarrassingly parallel, with their split and merge operations constituted of trivial computations (such as the summation of a small quantity of integers in the latter), thereby lending themselves to high scalabilities. Strassen multiplication shows moderate scalability, due to the submatrix additions that need to be performed sequentially during its split and merge operations. Finally, quicksort and mergesort give relatively low scalabilities, due to the sequential $O(n)$ sweeps over the n -element arrays required in their split (for quicksort) or merge (for mergesort) operations.

The above results also permit us to evaluate the implementational efficiency of our D&C skeleton when compared against other published results. In their investigation of Skandium’s performance over 16 cores, Tsogkas [48] reports speedups of 6.0× for quicksort and 4.8× for mergesort. Our counterpart D&C programs achieve speedups of 8.8× and 8.3× respectively over the same number of cores, which constitute an improvement of 46% and

71% over Tsogkas’s results. Separately, Leyton & Piquer [25] report a 3.4× speedup when using Skandium to run parallel quicksort over 8 cores, whilst Leijen et al. [8] improve this to 5.1× using TPL (as already mentioned in Section 3.1.2, p. 24). Our corresponding speedup of 6.0× (which is 74% efficient) outperforms them by 75% and 17% respectively.²⁹

7.2.1 Contiguous vs. Dispersed Allocation

In Section 5.1.3 (p. 57), we mentioned that contiguous core allocations could cause memory-intensive programs to suffer performance bottlenecks due to the shared off-chip bandwidth [51]. Therefore, we repeat our scalability experiments using a dispersed allocation (as depicted in Figure 5.3, p. 58) in order to compare its performance.

Figure 7.4 (below) presents the speedups resulting from the two allocations for up to 16 threads. The dispersed allocation consistently outperforms the contiguous one, since the threads can take advantage of the memory bandwidth from across *all* the processor chips. The most interesting case is sum-of-squares (a map–reduce), our most memory-intensive program, which improves its 16-thread speedup from 14.9× to 15.9× when dispersed, overtaking Monte Carlo Pi and achieving an efficiency of 99%.

At the same time, we note that the performance improvements, albeit significant, scale only modestly with respect to the available memory bandwidth. Even in the case of sum-of-squares, a 4× increase in bandwidth only yielded a 7% improvement. A possible interpretation of this result is that our programs are much more compute-bound than memory-bound, making physical concurrency (i.e. number of cores) the dominant factor in their performance, not memory bandwidth.

An alternate explanation revolves around the non-uniform memory access (NUMA) architecture of our manycore machine. Since each program’s problem data is initialized (before the test) sequentially on the main thread, it might have been allocated entirely to the memory module associated with its processor. Consequently, any memory access requests would still need to be handled by a single memory controller, which therefore

²⁹ These comparisons are subject to undiscussed experimental variations, such as problem sizes, and therefore do not fulfil the *ceteris paribus* assumption that would be expected of a proper scientific investigation. For this reason, these comparisons should be taken as merely informal observations.

becomes a performance bottleneck cancelling the benefits of the increased overall memory bandwidth.

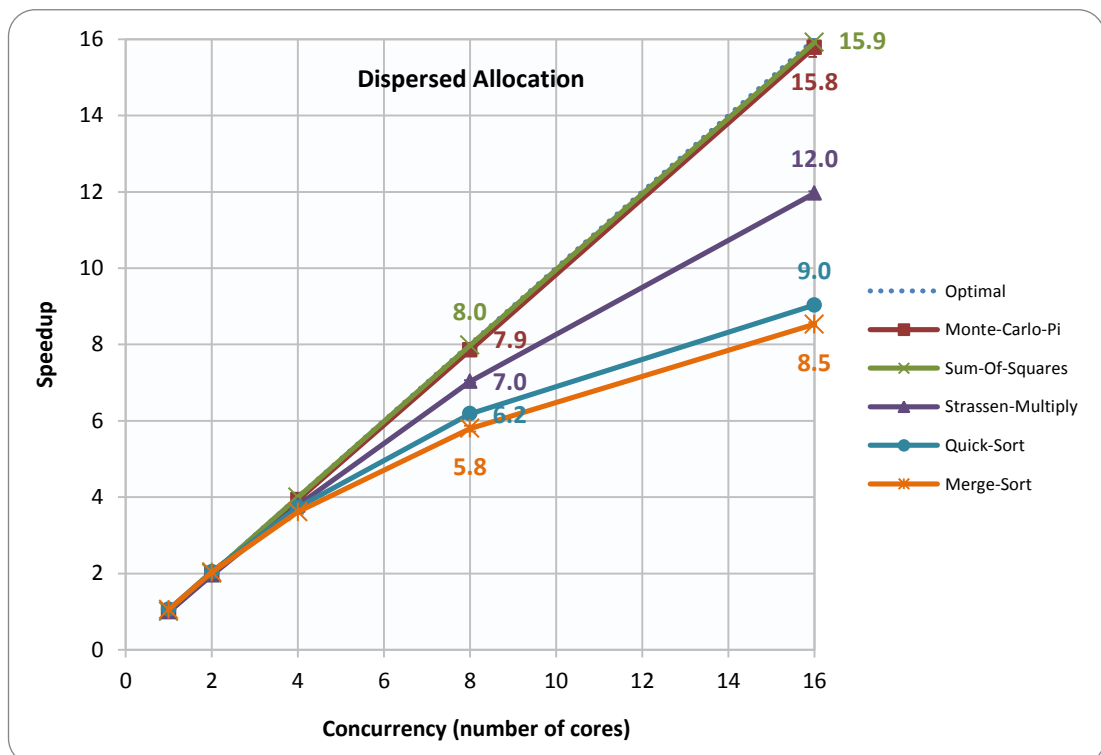
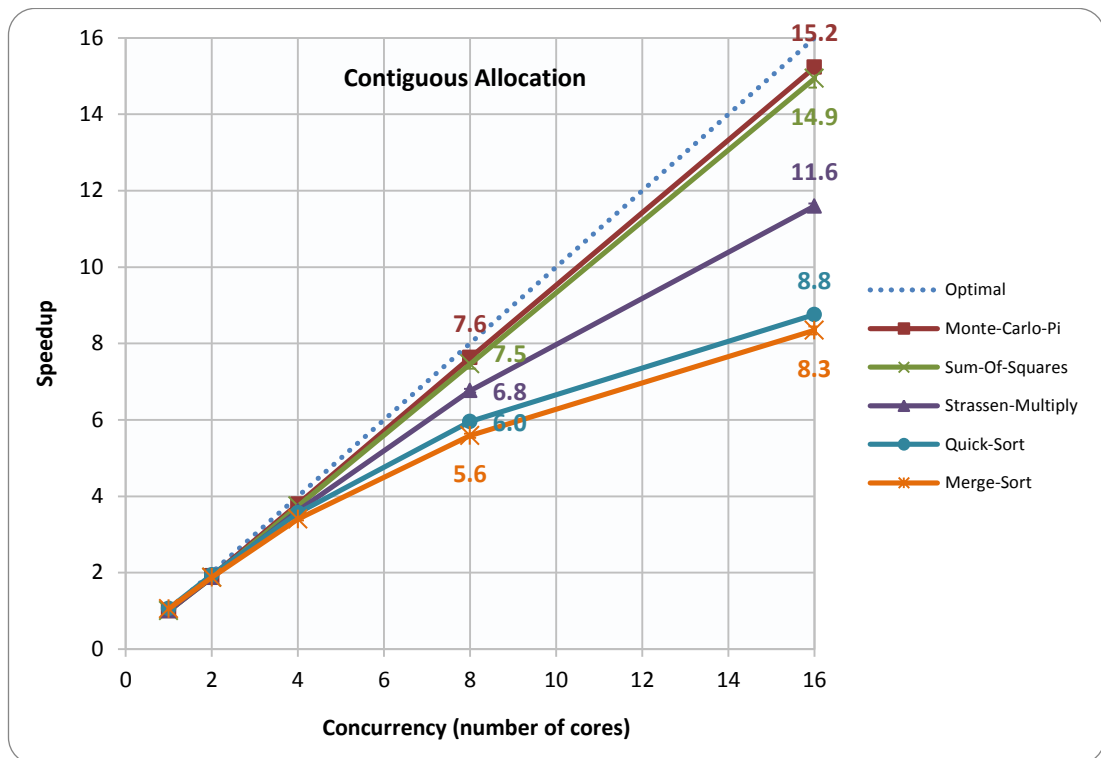


Figure 7.4. Speedup vs. concurrency for contiguous vs. dispersed allocation of pinned threads over cores of the 64-core machine

7.3 D&C Phases

Sasaki et al. [13] evaluate their parallel programs' scalabilities empirically, relying on instrumentation to measure their progress through cumulative retired instructions, but treating their underlying algorithmic structure as a black box. This limits the opportunity for interpreting results analytically. Given that we are investigating a specific class of parallel programs – namely, divide-and-conquer – we endeavoured to delve deeper and study the performance characteristics of our applications as they traverse the D&C recursion graph. To this end, we introduced execution hooks into our D&C skeleton so that our system could transparently collect statistics about the individual execution times of the split, execute, and merge functions.

Figure 7.5 (below) presents each phase's cumulative execution time (as summed across all tasks) for each program respectively. Again, the results corroborate our predictions from Section 4.4 (p. 41). Monte Carlo Pi, sum-of-squares, and mergesort keep splitting (with a branching factor of 2) until Level 8, switching to their execute phase at Level 9 (with 256 execute tasks, being 2^8). Quicksort, on the other hand, transitions from its split to its execute phase gradually along Levels 8–11. As explained in Section 4.5.2 (p. 50), its unbalanced splits cause some recursion paths to produce subproblems with sizes meeting the target granularity earlier than others. Finally, Strassen multiplication (with a branching factor of 7) switches to the execution phase at Level 4 (where it would have 343 execution tasks, being 7^3).

Monte Carlo Pi and sum-of-squares incur negligible computation in their split and merge phases, taking at most 3 ms and 29 ms for their Level 8 merge (involving the summation of 128 pairs of integers); over 99.9% of their time is spent in the execute phase. Strassen multiplication incurs considerable computation in both its split and its merge phases, constituting 14% and 3% of its overall time respectively. However, 87% of its split phase is spent at Level 3, where it would already have 49-way parallelism (through 7^2 split tasks), thereby mitigating its effect on scalability.

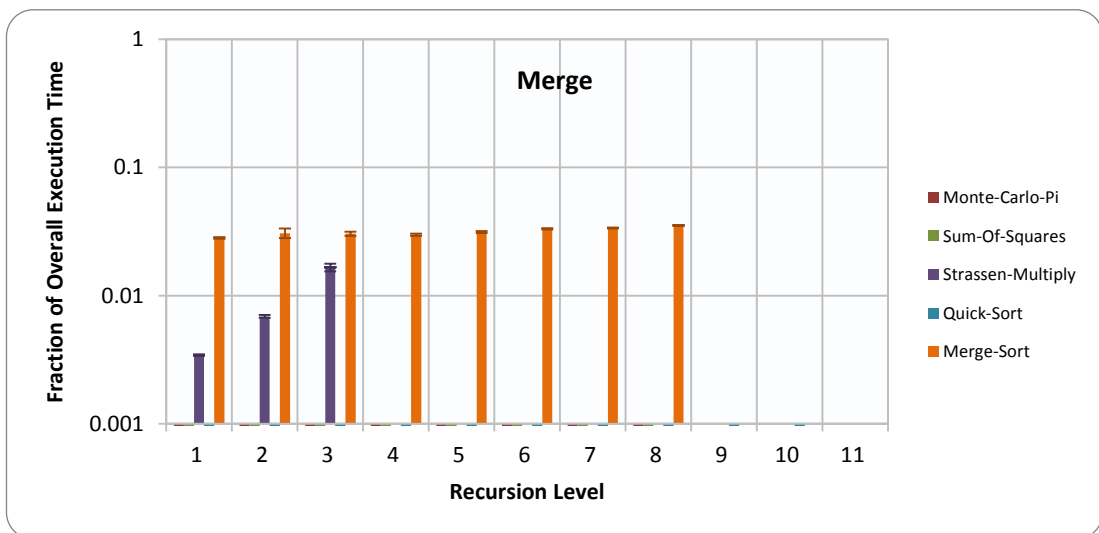
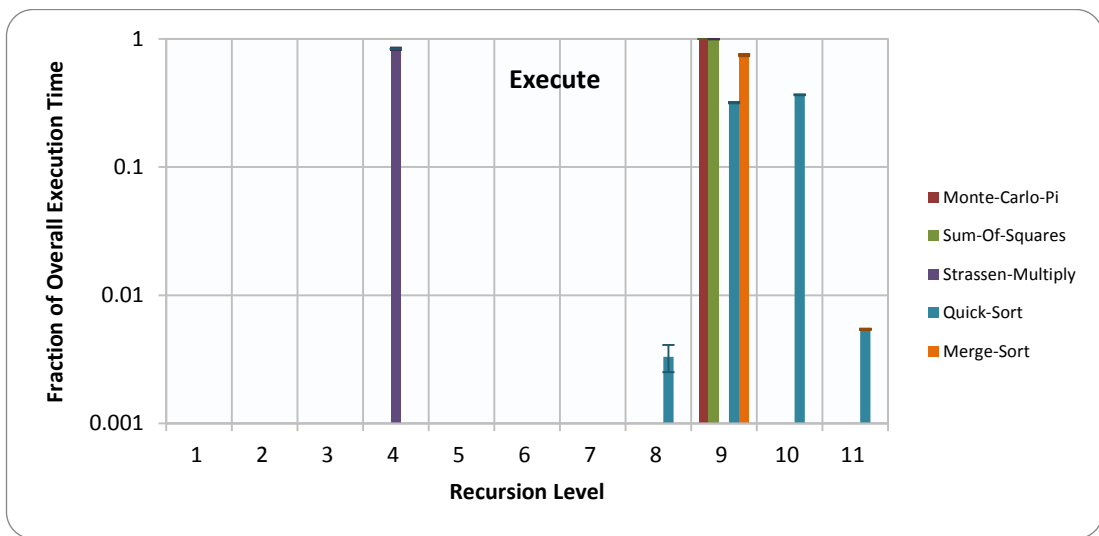
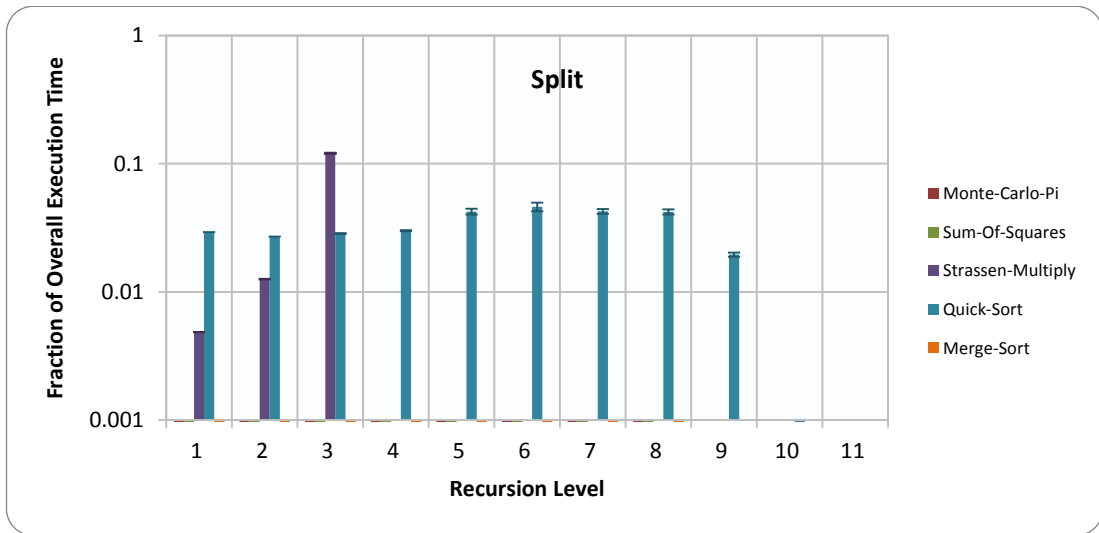


Figure 7.5. Cumulative execution times of tasks in each D&C phase. The vertical axes are logarithmic so as to allow small values to be visible.

Finally, quicksort and mergesort incur substantial computation in their split and merge phases respectively, constituting 31% and 25% of their overall times. In their case, this computation is spread out more or less evenly across all their levels, since each level entails a fresh sweep over the *entire* array of elements (albeit divided among varying numbers of parallel tasks). Quicksort’s split noticeably starts to taper off in the final levels, as some recursion paths would have transitioned to the execute phase.

7.4 Symmetric Tests

As a precursor to our multiprogramming tests, we ran some experiments to evaluate how our parallel programs would individually fare in a multiprogrammed context. In each symmetric test, we initialize a workload of 8 instances of the *same* given program (initialized with different random problems), and measure their execution times when run concurrently, over 64 cores, under the various multiprogramming schemes described in Section 5.2 (p. 58) and Section 5.3 (p. 64).

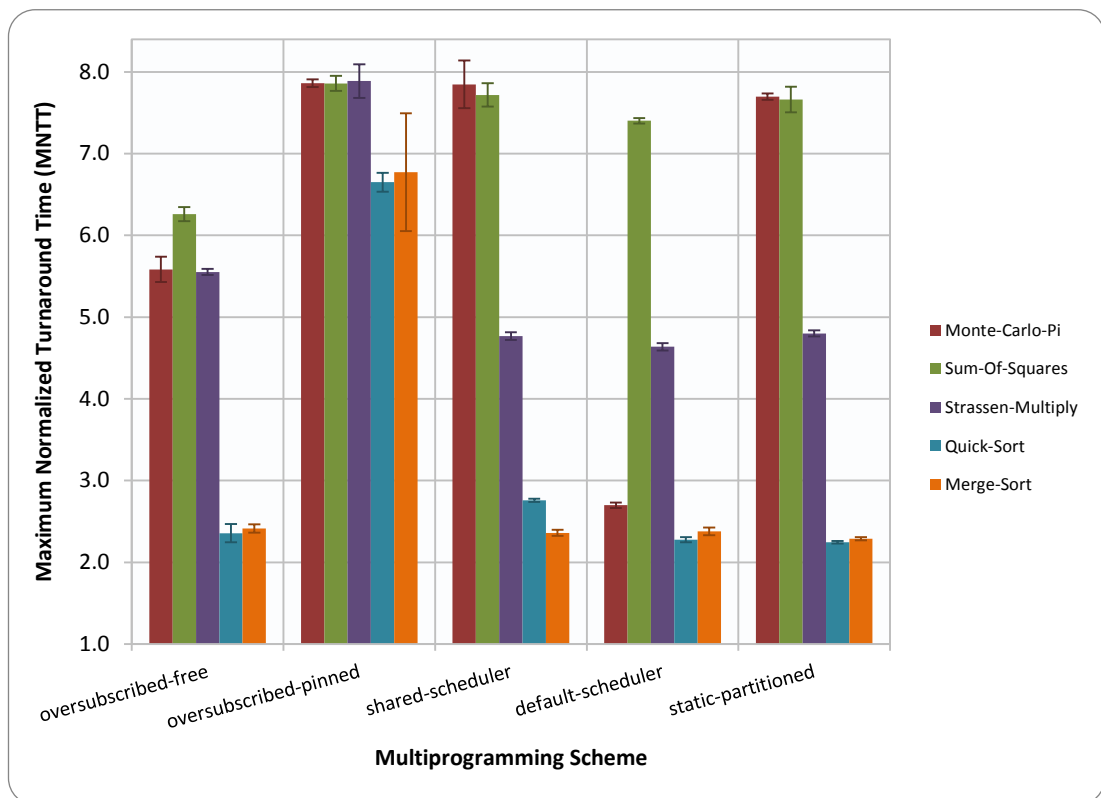


Figure 7.6. MNTT for symmetric tests involving 8 instances of the same program executed concurrently over 64 cores (lower is better). Under most multiprogramming schemes, low-scalability programs perform better than high-scalability ones.

Figure 7.6 (above) reveals some interesting trends. In general, the normalized performance of the low-scalability programs is drastically better than that of the high-scalability programs when run under an 8-instance workload. Under most schemes, quicksort and mergesort exhibit an MNTT of 2–3, whilst Monte Carlo Pi and sum-of-squares often have an MNTT of 7–8, which is only slightly better than if the 8 instances were to be executed consecutively in isolation.

This result is intuitive. High-scalability programs, by definition, can make efficient utilization of any number of cores they are assigned, leaving little opportunity for improvement through scaling down. On the other hand, low-scalability programs exhibit diminishing returns over larger numbers of cores; thus, when their effective concurrency is implicitly reduced due to contention with other program instances, their efficiencies would substantially improve.

There is a large performance difference between the two thread-oversubscription schemes. Despite both causing an average oversubscription of 8 threads per core, the free scheme – which leaves the operating system’s thread scheduler at liberty to distribute the threads among the cores – outperforms the pinned scheme by 20–30% for the three high-scalability programs, and by around 65% for the low-scalability ones. This demonstrates that the thread scheduler performs an adequate job at boosting performance through thread migration.

Note that, in the measurements reported for our statically- or dynamically-partitioning multiprogram scheduler in this and subsequent tests, we do not include the execution time it required for initially measuring each program’s scalability (per the procedure described in Section 5.3.2, p. 67). We feel that this omission is justified, since the said overhead is conceptually analogous to a compile-time (rather than run-time) cost. Once the scalability for a particular D&C program has been measured over a representative problem, it may be reused indefinitely for processing similar problems (through the same program), thereby amortizing its cost.

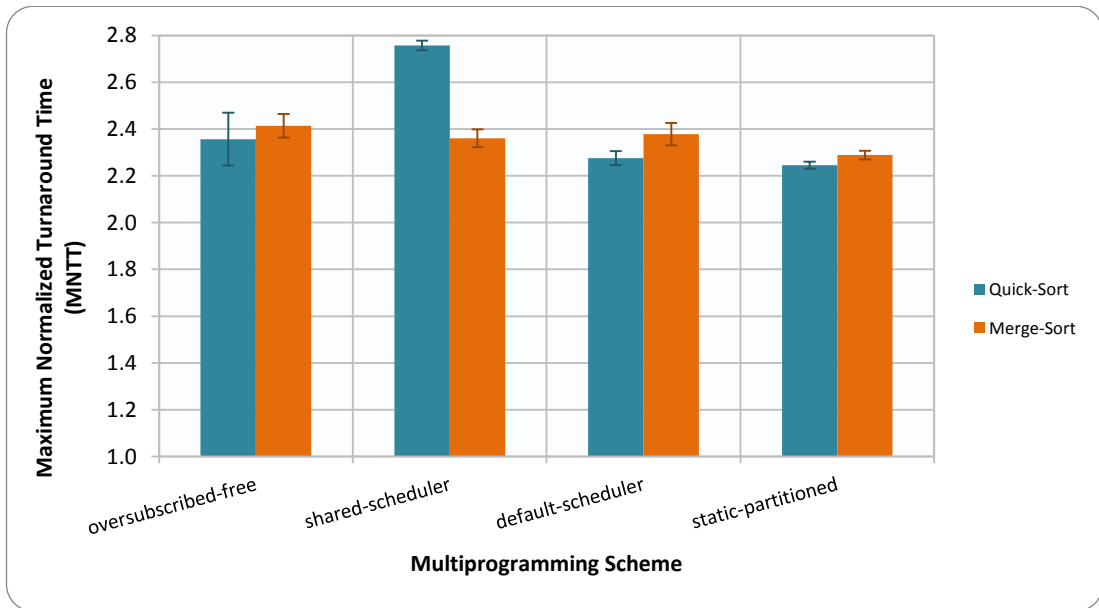


Figure 7.7. MNTT for symmetric tests of low-scalability programs. Static partitioning achieves the best performance by a small margin.

Finally, Figure 7.7 considers the performances of just the two low-scalability programs, and shows that our statically-partitioned scheduler has the potential to outperform all other schemes (albeit not by a statistically-significant margin). In this case, since each workload consists of instances of the same program, the partitioning would be egalitarian (with each instance receiving an equal number of cores).

7.5 Cyclic Tests

The main experiments of our project involve running cyclic tests, as described in Section 5.4.1 (p. 76).

7.5.1 Full Program Suite

Initially, we run our cyclic tests over multiprogram workloads comprising all our five parallel programs. The cyclic tests differ from the symmetric tests of the previous section in that they only run a single instance of each program, but mix the different programs into the same workload, and execute them repeatedly for a total time interval of 8 minutes (chosen empirically such that each program would get to run around 100 times in each test).

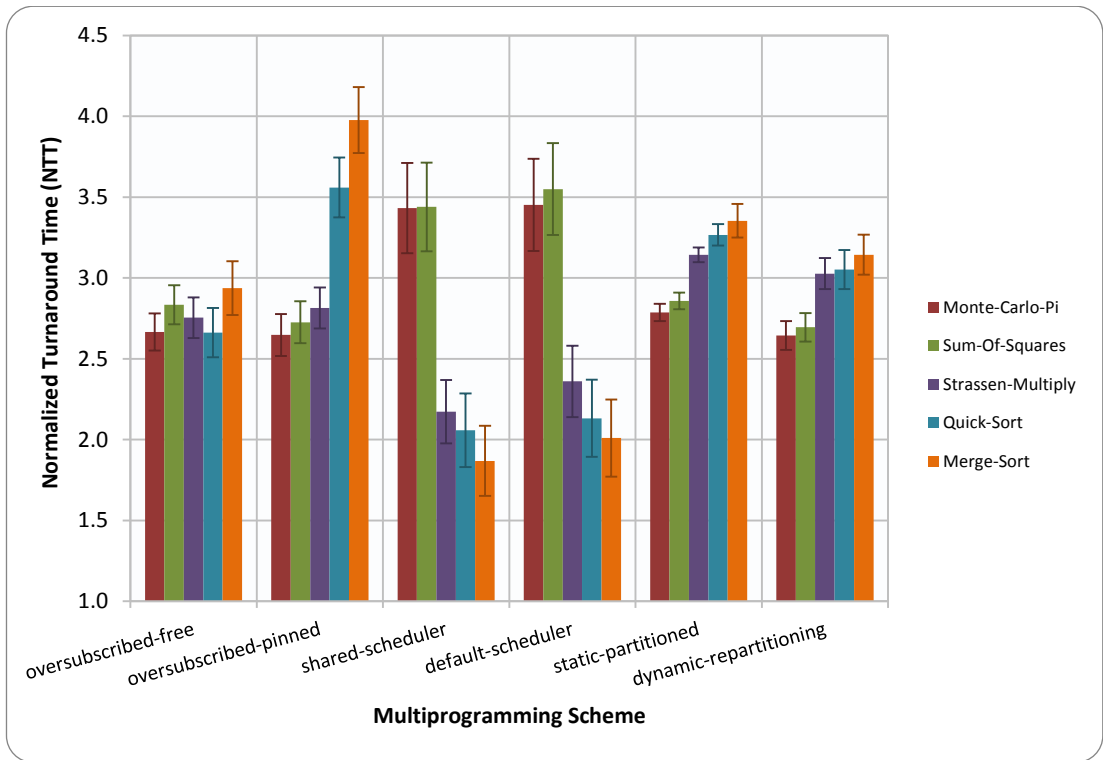


Figure 7.8. NTTs of our 5-program workload (lower is better). Shared-scheduler schemes favour low-scalability programs and suffer the worst variances. All other schemes favour high-scalability programs.

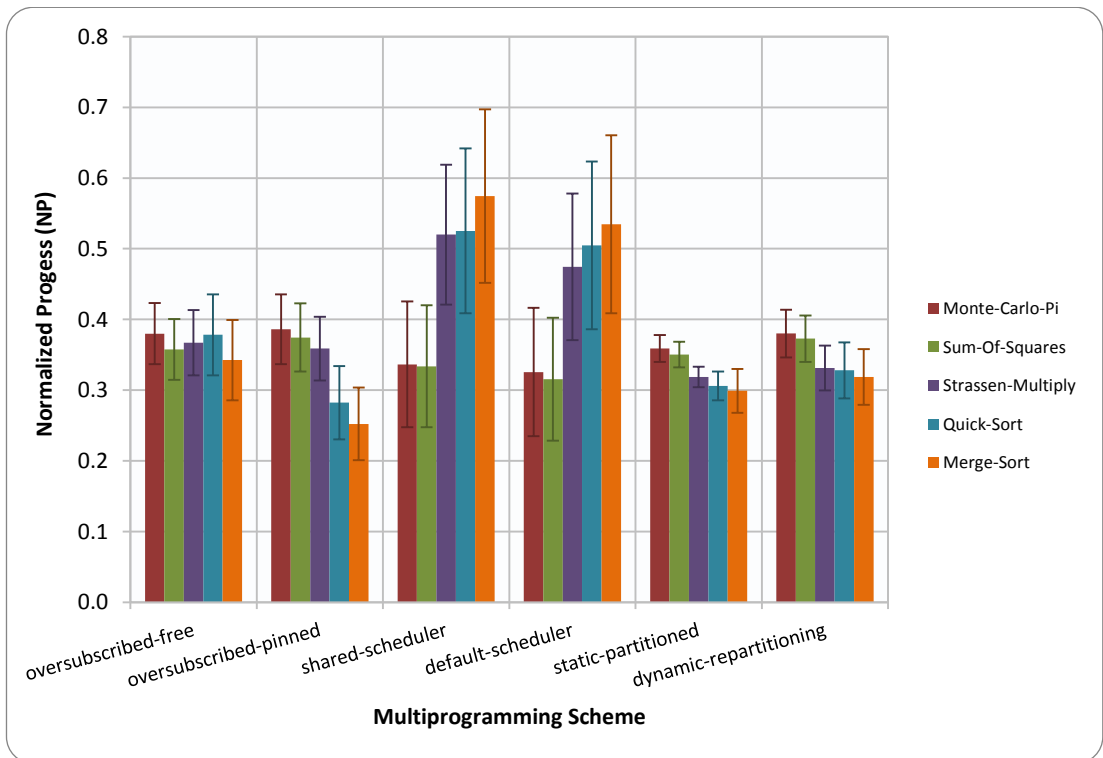


Figure 7.9. NPs of our 5-program workload (higher is better)

Figure 7.8 and Figure 7.9 (above) present the NTTs and NPs of our respective programs under this multiprogram workload. In the thread-oversubscription schemes, as well as our processor-partitioning schemes, higher-scalability programs perform better than lower-scalability ones. However, the trend is completely reversed in the shared-scheduler schemes (which include the default scheduler), where the low-scalability programs perform best.

Another interesting observation pertains to the variances among consecutive runs of the same program under the various schemes, as depicted through the error bars. Shared-scheduler schemes incur the worst variances, with the relative standard deviations (RSD) of the various programs' NTTs averaging 38%. This is reduced to 18% in the thread-oversubscription schemes, 13% in our dynamically-repartitioning scheme, and merely 7% in our statically-partitioned scheme.

This result can be explained through a number of factors. Shared-scheduler schemes suffer from inherently volatile performances, since minor timing fluctuations during the execution of the task scheduler's work-stealing logic can lead to large differences in the overall task assignment among threads. Oversubscription schemes benefit from the thread scheduler's goal of promoting fairness. Finally, processor partitioning ensures that each program has its own unique set of processors, making its performance much more deterministic. Dynamic repartitioning introduces some unpredictability, but only insofar as there is an insufficiency of available tasks in some programs.

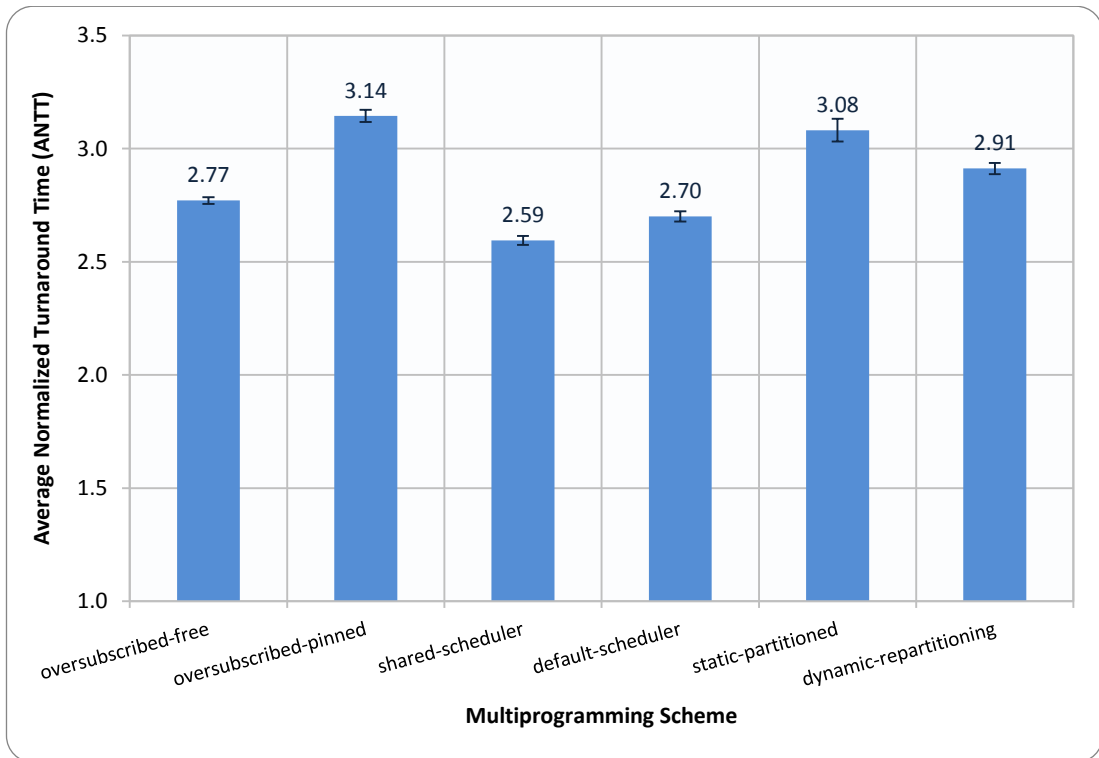


Figure 7.10. ANTT of our 5-program workload (lower is better). Best system-level performance is achieved by the shared-scheduler schemes.

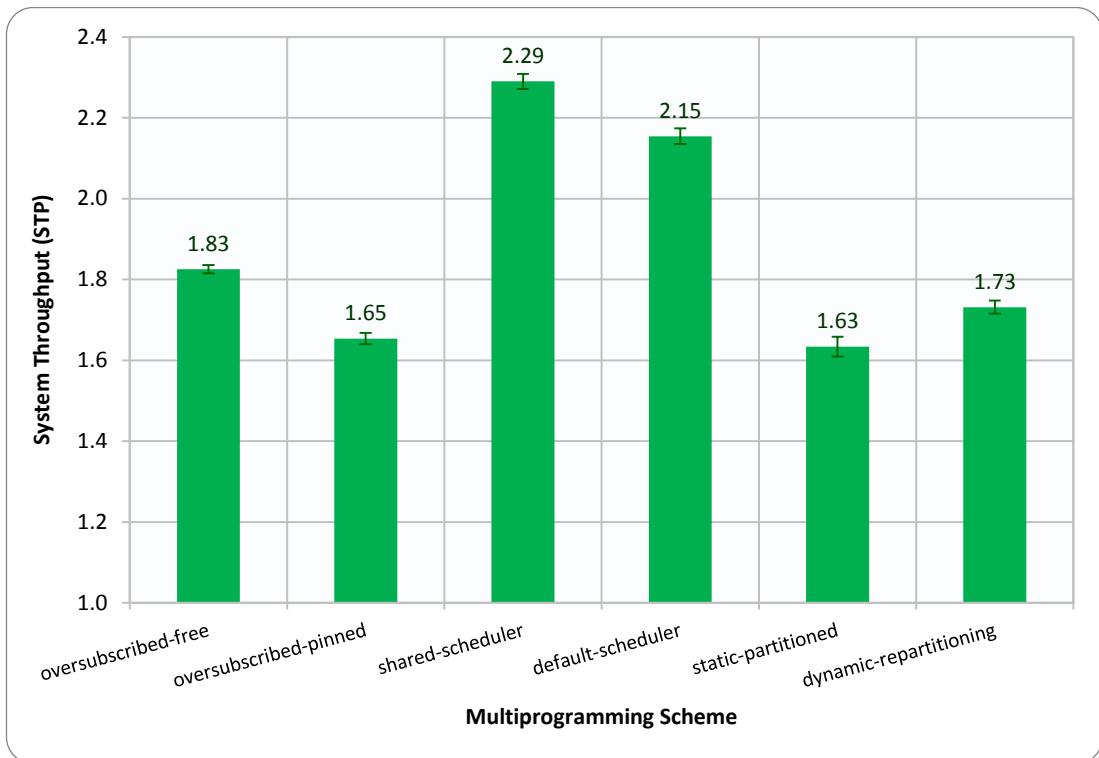


Figure 7.11. STP of our 5-program workload (higher is better)

Figure 7.10 and Figure 7.11 present the system-level metrics for our 5-program workload. The best performance is given by the shared-scheduler schemes, in which our `AffinityTaskScheduler` outperforms the default scheduler by 6% in terms of STP. The thread-oversubscription scheme with free threads is 10% faster than the one which pins them, again showing that the operating system’s thread scheduler makes beneficial decisions on thread migration. Notwithstanding, the free oversubscription scheme is still 20% slower than the shared scheduler, which is strong evidence that work-stealing task scheduling can significantly outperform the thread scheduler for compute-intensive programs. Our dynamically-repartitioning scheduler is 6% faster than when statically partitioned. However, it is 24% slower than the shared scheduler, thereby seemingly rebutting our hypothesis.

7.5.2 Low- to Moderate-Scalability Subset

Sasaki et al. [13], citing Bhadauria & McKee [53], observe that:

“Minimizing ANTT means that all the programs are fairly achieving high performance compared to its peak performance (achievable only when occupying the whole system by itself). Therefore, programs that scale almost linearly [...] are removed from the [multiprogram coscheduler] and run in isolation, or gang-scheduling.” — Sasaki et al. [13]

On this basis, we altered our experiment to remove the highly-scalable programs (Monte Carlo Pi and sum-of-squares), and rerun the cyclic tests with just the moderate- and low-scalability ones (Strassen multiplication, quicksort, and mergesort).

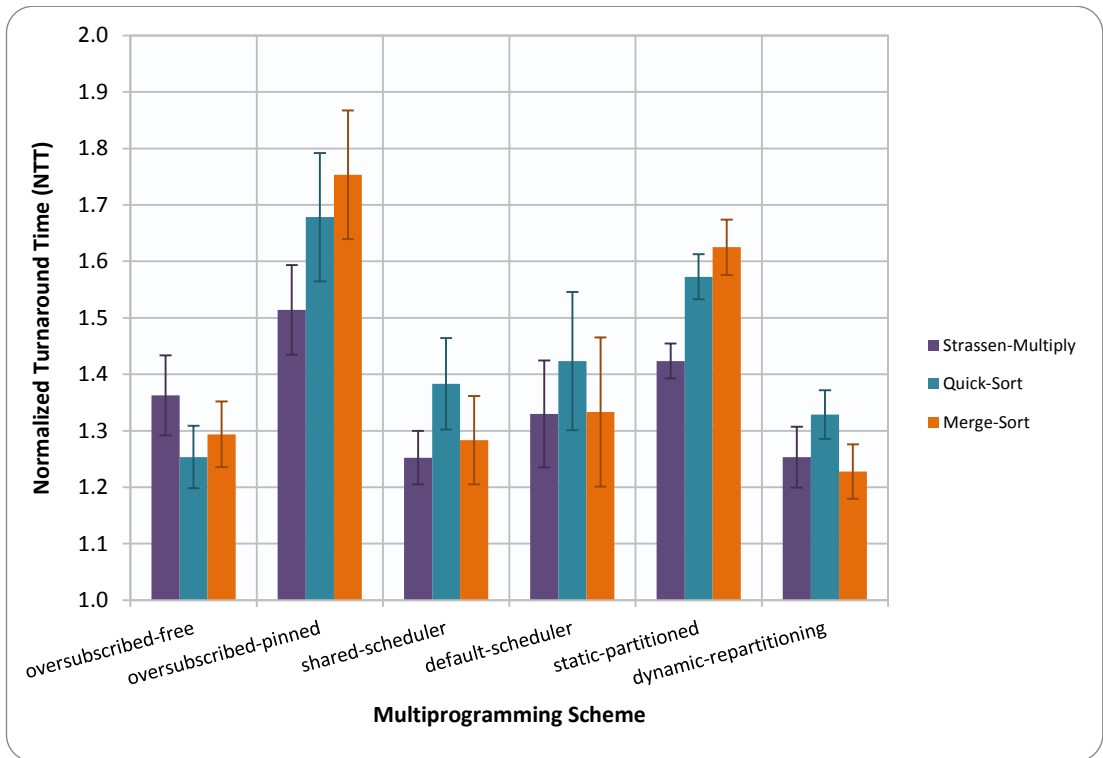


Figure 7.12. NTTs of our 3-program workload (lower is better)

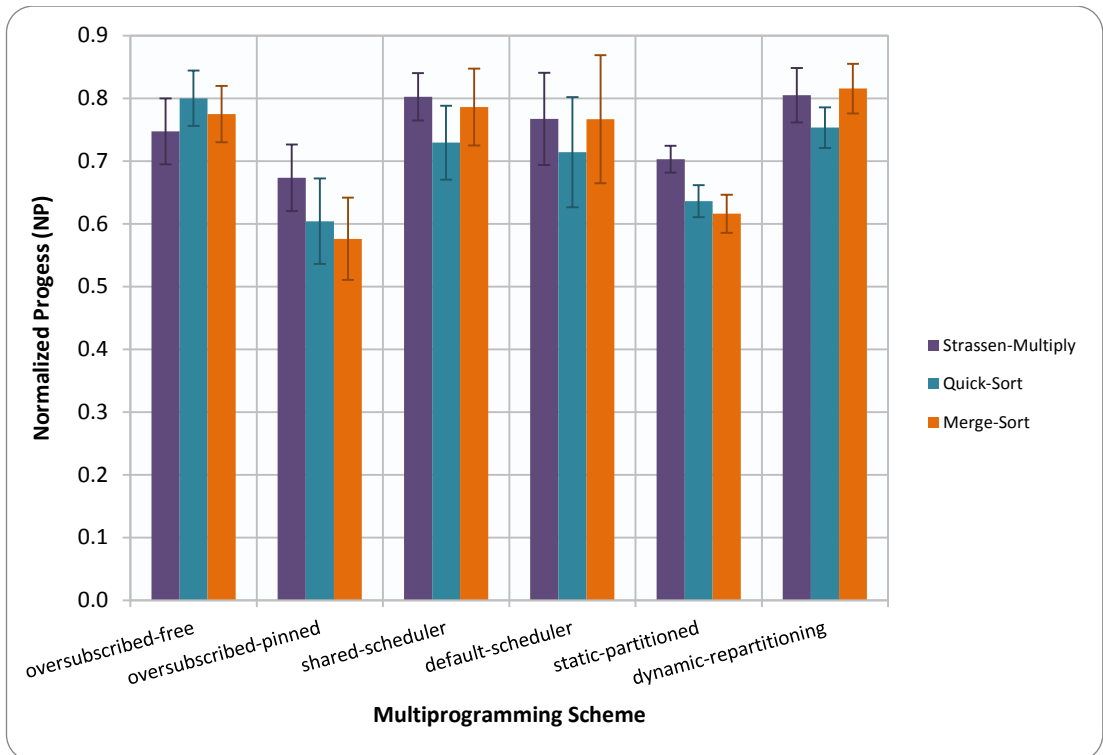


Figure 7.13. NPs of our 3-program workload (higher is better)

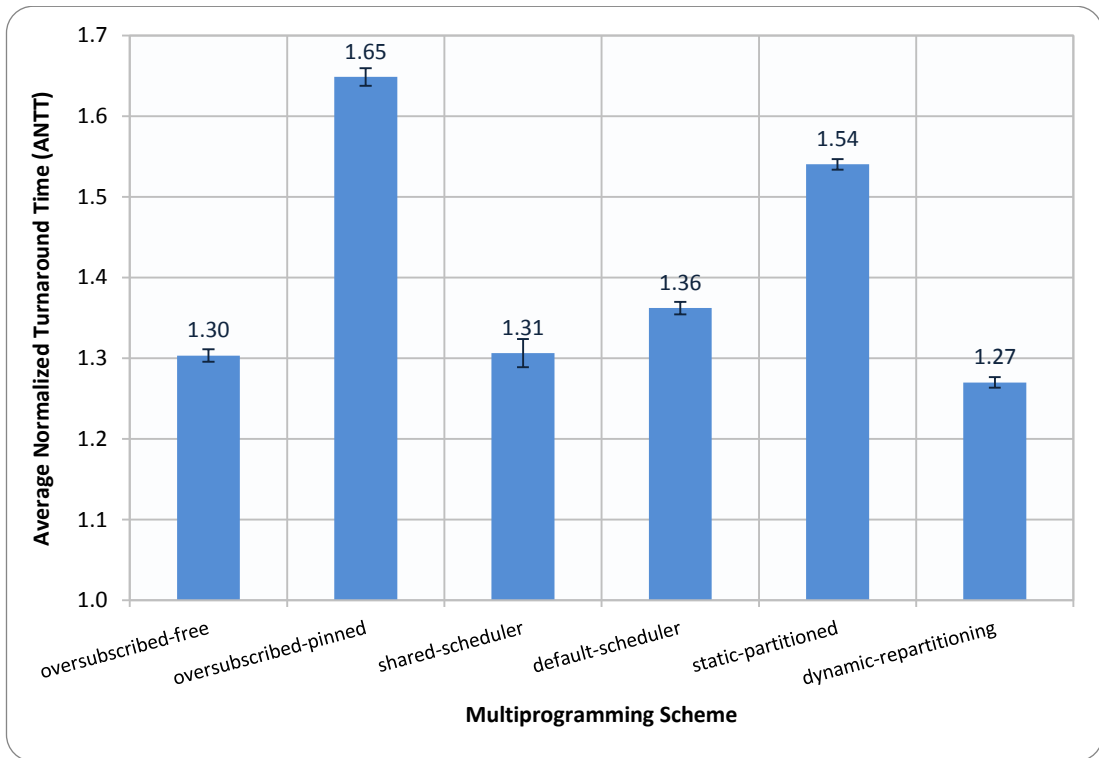


Figure 7.14. ANTT of our 3-program workload (lower is better).
Best system-level performance is now achieved by our repartitioning multiprogram scheduler.

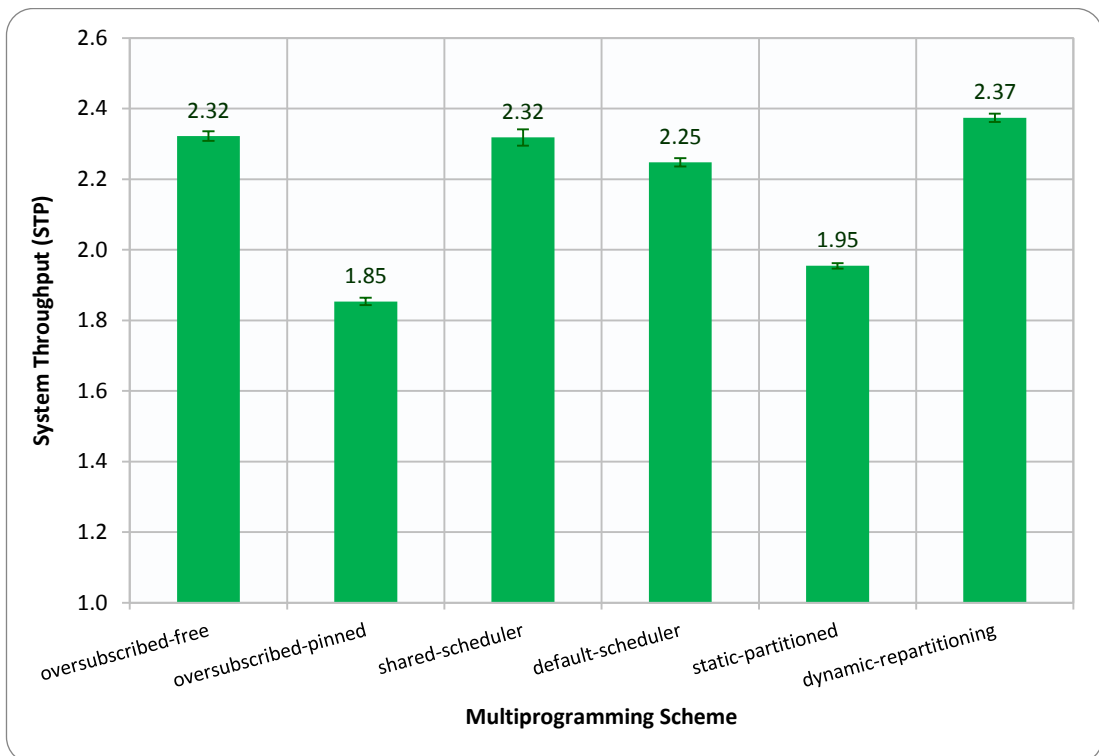


Figure 7.15. STP of our 3-program workload (higher is better)

Figure 7.12 to Figure 7.15 (above) present the results of this workload involving just moderately- and poorly-scalable programs. In this case, our repartitioning scheduler does achieve the best performance from among all multiprogramming schemes, outperforming the default scheduler by a solid 5½% in terms of STP, thereby confirming our hypothesis that scalability-based partitioning can be beneficial for task-parallel programs. Meanwhile, its performance also constitutes a 2% improvement over both the shared scheduler and the free oversubscription scheme, with the result being statistically significant at 99% confidence (when averaged over 21 runs).

7.5.3 Comparison with Related Work

One of our initial goals was to reproduce the results obtained by Sasaki et al. [13], but applied to task-parallel programs. Our experiments are most directly comparable to the results presented in Section 5.2 of their paper, where they investigate the performance of their scalability-based manycore partitioning (SBMP) scheduler, with phase prediction enabled, on multiprogram workloads involving parallel programs with multiple phases. Their phase prediction capability plays a role quite similar to our task-availability-based repartitioning (discussed in Section 5.3.4, p. 72).³⁰ Their SBMP scheduler “significantly outperforms” the Linux thread scheduler by 6% in terms of ANTT.

Our repartitioning scheduler, on the other hand, only outperforms the Linux thread scheduler (as leveraged within the free oversubscription scheme) by 2.6% in ANTT. There are several factors that could account for this shortcoming. Due to the efficiency of our D&C skeleton’s asynchronous execution, all our programs exhibit relatively clean scalabilities, with speedups continuing to increase monotonically up to 64 cores (as shown in Figure 7.3, p. 85). This is not the case for the PARSEC benchmark used by Sasaki et al. [13], several of whose programs suffer from erratic scalabilities (see Figure 7.16 below), particularly within the “Yellow” and “Red” groups, which are the focus of their SBMP scheduler. This poor performance of the benchmark programs affords the SBMP scheduler more room for their improvement.

³⁰ The core donation technique suggested by Sasaki et al. [13] is not relevant to our experiments, since it is intended for applications exhibiting “low CPU utilization”.

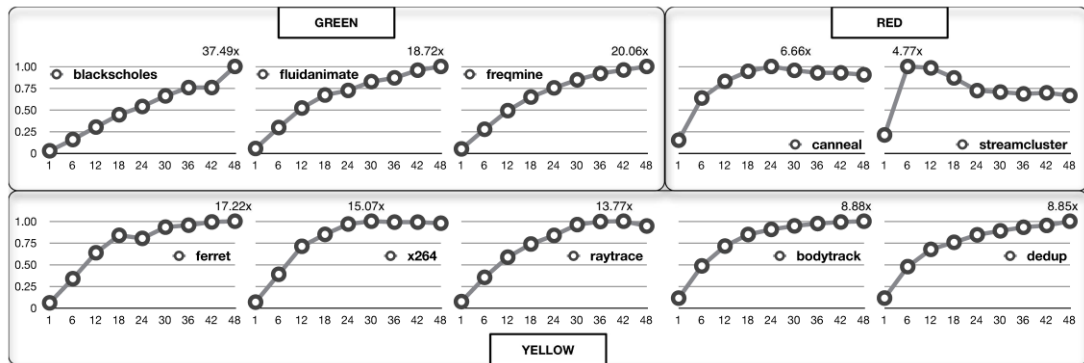


Figure 7.16. Scalabilities of PARSEC benchmarks. Copied from Sasaki et al. [13].

Furthermore, given that our D&C skeleton is implemented to use task parallelism rather than explicit threading, all our tests run on top of a work-stealing task scheduler. This includes the free oversubscription scheme that we have adopted as our baseline, which simply initializes such a scheduler (with a full contingent of threads) for each program in the workload. The performance benefits of work-stealing task schedulers (discussed in Section 3.1.2, p. 22) are well-established, both in research and in industry [7], [8], [46]. (For example, Tsogkas [48] observed that a task-parallel mergesort could outperform an explicitly-threaded one by 45% on 16 cores.³¹) Thus, it is reasonable to assume that this work-stealing logic reaps most of the benefits to be had from cache reuse (due to the data locality arising from its LIFO pushing/popping) and dynamic load-balancing (from its FIFO work-stealing), leaving us limited opportunity for further improvement. Sasaki et al. [13], on the other hand, use explicitly-threaded applications [54], which are likely to be more amenable to performance boosts from improved data locality.

Several of our programs – particularly, sum-of-squares, quicksort, and mergesort – work on sample problems that have large memory footprints (256 or 512 MB). Since they would be continuously scanning arrays at least an order of magnitude larger than the processor caches, their working sets are transient, making them unlikely to enjoy significant benefits from the cache reuse promoted by processor partitioning.

³¹ For their task-parallel tests, Tsogkas used the Skandium library, whose task scheduler uses a centralized queue, meaning that even better results might have been possible through work-stealing.

Finally, our system was built on the high-level programming platform presented by the .NET Framework, and therefore needs to run on Mono when tested on our Linux-based manycore machine. This introduces overheads and unpredictability due to the mechanisms supporting the higher level of abstraction, including the garbage collector, which can kick in and temporarily stall the system at arbitrary points in time. The PARSEC benchmark suite, being written in C/C++ [54], can largely avoid these issues.

Chapter 8: Conclusion

To close off our dissertation, we shall recapitulate the main achievements of our project. We then present a number of potential improvements and future work for our system, and conclude with some final remarks.

8.1 Achievements

By architecting a fresh implementation of the divide-and-conquer skeleton atop the task infrastructure of the Task Parallel Library (TPL) in the .NET Framework, we have demonstrated the design benefits of layered parallel abstractions (Section 2.2, p. 9). In particular, this layering permits our skeleton to implicitly reap the performance gains of the separately-developed work-stealing task scheduler. By contrast, Skandium adopts a stovepipe approach, whereby it implements its own task-scheduling logic using a centralized queue [25], enabling our implementation to outperform it by 75%.³²

By optimizing our D&C skeleton to use asynchronous parallelism (Section 4.2.2, p. 35), we effectively eliminated almost all instances of blocking from our system. The skeleton uses an atomic decrement (rather than a blocking primitive) to identify when to spawn the merge tasks. The programs themselves do not use any explicit synchronization at all, relying on the skeleton to direct their entire execution flow. This execution efficiency has given us a 17% improvement over traditional fork–join implementations for divide-and-conquer algorithms on TPL [8].

Our combination of structured parallelism (offered by the D&C skeleton) with functional-style programming (through C# lambda expressions and LINQ) engendered a declarative programming model whose expressive power was demonstrated through our succinct 8-line parallel quicksort implementation (Section 4.1.4, p. 31).

Through our comparison of multiprogramming schemes, we have established that sharing a single work-stealing task scheduler among all concurrent programs (and thereby

³² As has been already mentioned on p. 86, the comparisons in this section do not account for experimental variations, such as problem sizes, and should only be treated as casual observations.

eliminating thread oversubscription altogether) can outperform the Linux thread scheduler by 25% for multiprogram workloads containing variable scalabilities (Section 7.5.1, p. 92).

Our main achievement lies in our novel multiprogram scheduler, which unifies the designs of work-stealing task-scheduling and scalability-based processor-partitioning into a single component that can draw on the performance benefits of both (Section 5.3, p. 64). This scheduler has been engineered to work for any arbitrary multiprogram workload, including unseen programs (other than the five we have considered). However, it is best-suited for workloads comprised of low- to moderate-scalability programs, where it has been shown to outperform the TPL default task scheduler by 5½%, and the Linux thread scheduler by 2% (Section 7.5.2, p. 96).

8.2 Potential Improvements and Future Work

The most significant shortcoming of our scalability-based processor-partitioning technique is the simplified manner in which it evaluates scalability (Section 5.3.2, p. 66). Whilst this was, to an extent, acceptable for our suite of cleanly-scalable programs, it would give rise to issues if employed over programs exhibiting erratic scalabilities, such as the PARSEC benchmark (Figure 7.16, p. 100). Therefore, our partitioning strategy should be upgraded to use a hill-climbing heuristic over a populated scalability table, as is done by Sasaki et al. [13].

High-performance computing on manycore machines would inevitably benefit when the software developers align their systems according to the physical characteristics of the underlying hardware platform. For example, Sasaki et al. [13], running their experiments on a 48-core machine comprised of eight 6-core processor dies, restricted their scheduler to use the said 6-core die “as a minimum unit of allocation to programs” [13]. Since our manycore consisted of just four 16-core processor chips, we could not afford to impose such a chip-level restriction; however, this left us susceptible to negative interference among programs executing on cores belonging to the same chip. If our scheduler were to be run on a manycore machine that has a finer distribution of cores, then it should be modified to perform such alignment along processor chip boundaries.

A design limitation concerns the rigidity of the D&C skeleton. When developing the Strassen matrix multiplication program (Section 4.4.3, p. 45), we found that the skeleton’s interface

did not allow us to parallelize the submatrix additions (specified within the *I–VII* computations) as part of the spawned subtasks; rather, these had to be performed sequentially within the parent split operation. This restriction caused a performance penalty, with the split phase accounting for 14% of the program’s overall execution time (Section 7.3, p. 88). By extending the skeleton’s interface to permit multiple definitions of the recursive case (in this case, one for the first level, and one for all subsequent levels), the issue could have been avoided.

In Section 4.5.3 (p. 53), we presented the reasons why we did not employ nested parallelism in our skeletal designs. In production scenarios, it would obviously be desirable to maximize machine utilization at all levels of the recursion, by parallelizing the split and/or merge operations as well, as depicted in Figure 8.1 (below).

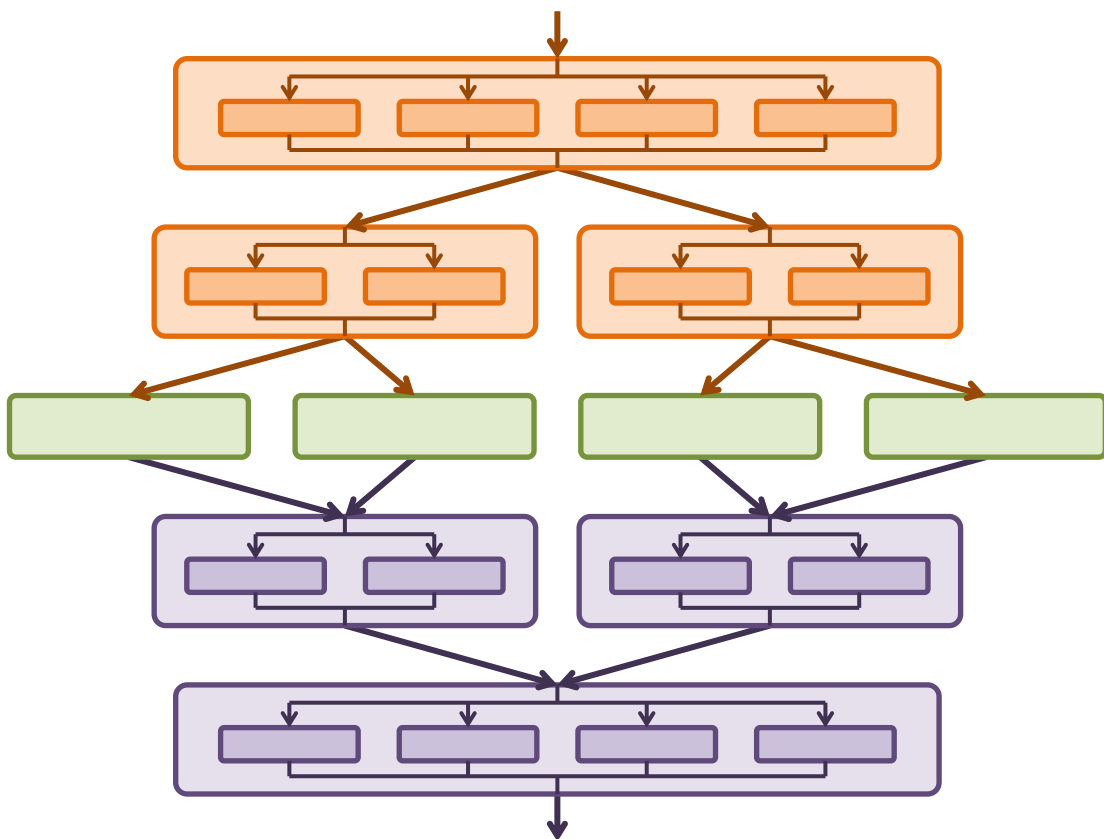


Figure 8.1. Nested parallelism attained by embedding sub-skeletons within the split and merge operations of the top-level D&C skeleton. Using this approach, full concurrency may be achieved at all levels of the recursion.

Nested parallelism may be achieved by: replacing the split and/or merge muscles with nested sub-skeletons [25] (typically of other skeleton types, such as map–reduce);

developing “fused” skeletons that support this composite parallelism natively [31]; or introducing *ad hoc* parallelism at the program level [35] (such as through explicitly-spawned subtasks).

8.3 Closing Remarks

At its core, the design of this project was an exercise in architectural layering through structured parallelism. We principally worked at three layers: the task scheduler (for distributing tasks onto processors), the D&C skeleton evaluator (for generating task graphs), and the actual program instances (for expressing the problem-domain logic). One of the outcomes that struck us during this experience was the ease and confidence with which extensive changes could be applied to the components of a given layer without affecting the other layers, despite their close interdependence for the run-time execution.

The object-oriented programming (OOP) paradigm that currently pervades the software industry encourages applications to be designed using a functional layering that is often orthogonal to parallelism. Any necessary parallelization would typically be bolted-on late in the development process, giving rise to “spaghetti code”-like behaviour, with threads darting sporadically across different objects. This often breaks the notion of encapsulation, since any threading issues that need to be debugged (such as deadlocks) would require the developer to trace the threads’ entire flow across the otherwise-unrelated classes.

Through this project, we have demonstrated that layered parallelism can serve not only as an adequate multi-level abstraction for aiding software developers writing parallel programs, but also as a rich source of structural information for transparently boosting the system’s parallel performance.

Bibliography

- [1] H. Sutter, 'The free lunch is over: A fundamental turn toward concurrency in software', *Dr. Dobbs's Journal*, vol. 30, no. 3, pp. 202–210, 2005.
- [2] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiawicz, N. Morgan, D. Patterson, K. Sen, and J. Wawrzynek, 'A view of the parallel computing landscape', *Communications of the ACM*, vol. 52, no. 10, pp. 56–67, 2009.
- [3] B. Catanzaro, A. Fox, K. Keutzer, D. Patterson, B.-Y. Su, M. Snir, K. Olukotun, P. Hanrahan, and H. Chafi, 'Ubiquitous parallel computing from Berkeley, Illinois, and Stanford', *IEEE Micro*, vol. 30, no. 2, pp. 41–55, Apr. 2010.
- [4] H. Sutter and J. Larus, 'Software and the concurrency revolution', *Queue*, vol. 3, no. 7, pp. 54–62, 2005.
- [5] E. A. Lee, 'The problem with threads', *Computer*, vol. 39, no. 5, pp. 33–42, 2006.
- [6] T. Willhalm and N. Popovici, 'Putting Intel Threading Building Blocks to work', in *Proceedings of the 1st International Workshop on Multicore Software Engineering*, 2008, pp. 3–4.
- [7] E. Ayguade, N. Coptly, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang, 'The design of OpenMP tasks', *IEEE Transactions on Parallel and Distributed Systems*, vol. 20, no. 3, pp. 404–418, 2009.
- [8] D. Leijen, W. Schulte, and S. Burckhardt, 'The design of a task parallel library', in *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, New York, NY, USA, 2009, pp. 227–242.
- [9] S. Borkar, 'Thousand core chips: a technology perspective', in *Proceedings of the 44th annual Design Automation Conference*, New York, NY, USA, 2007, pp. 746–749.
- [10] A. Heinecke, M. Klemm, and H.-J. Bungartz, 'From GPGPU to Many-Core: Nvidia Fermi and Intel Many Integrated Core Architecture', *Computing in Science Engineering*, vol. 14, no. 2, pp. 78–83, 2012.
- [11] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Paillet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van der Wijngaart, and T. Mattson, 'A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS', in *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, 2010, pp. 108–109.
- [12] E. Frachtenberg and Y. Etsion, 'Hardware parallelism: Are operating systems ready? (Case studies in mis-scheduling)', in *Proceedings of Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA'06)*, 2006.

- [13] H. Sasaki, T. Tanimoto, K. Inoue, and H. Nakamura, 'Scalability-based manycore partitioning', in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, New York, NY, USA, 2012, pp. 107–116.
- [14] M. K. Emani, Z. Wang, and M. F. P. O'Boyle, 'Smart, adaptive mapping of parallelism in the presence of external workload', presented at the 2013 International Symposium on Code Generation and Optimization (CGO '13), 2013.
- [15] M. Rajagopalan, B. T. Lewis, and T. A. Anderson, 'Thread scheduling for multi-core platforms', in *Proceedings of the 11th USENIX workshop on Hot topics in operating systems (HOTOS '07)*, 2007.
- [16] G. Moore, 'Cramming more components onto integrated circuits', *Electronics*, vol. 38, no. 8, pp. 114–117, Apr. 1965.
- [17] K. Olukotun and L. Hammond, 'The future of microprocessors', *Queue*, vol. 3, no. 7, pp. 26–29, 2005.
- [18] M. J. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. I. August, 'Revisiting the sequential programming model for multi-core', in *40th Annual IEEE/ACM International Symposium on Microarchitecture, 2007 (MICRO 2007)*, 2007, pp. 69–84.
- [19] M. Flynn and P. Hung, 'Microprocessor design issues: thoughts on the road ahead', *IEEE Micro*, vol. 25, no. 3, pp. 16–31, 2005.
- [20] L. Hammond, B. A. Nayfeh, and K. Olukotun, 'A single-chip multiprocessor', *Computer*, vol. 30, no. 9, pp. 79–85, 1997.
- [21] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger, 'Clock rate versus IPC: The end of the road for conventional microarchitectures', *ACM SIGARCH Computer Architecture News*, vol. 28, no. 2, pp. 248–259, 2000.
- [22] A. C. Sodan, J. Machina, A. Deshmeh, K. Macnaughton, and B. Esbaugh, 'Parallelism via multithreaded and multicore CPUs', *Computer*, vol. 43, no. 3, pp. 24–32, Mar. 2010.
- [23] D. Koufaty and D. T. Marr, 'Hyperthreading technology in the NetBurst microarchitecture', *IEEE Micro*, vol. 23, no. 2, pp. 56–65, 2003.
- [24] D. Burger and J. R. Goodman, 'Billion-transistor architectures', *Computer*, vol. 30, no. 9, pp. 46–48, 1997.
- [25] M. Leyton and J. M. Piquer, 'Skandium: Multi-core programming with algorithmic skeletons', in *2010 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, 2010, pp. 289–296.
- [26] S. Okur and D. Dig, 'How do developers use parallel libraries?', in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, New York, NY, USA, 2012, pp. 54:1–54:11.
- [27] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation*. Cambridge, MA, USA: MIT Press, 1991.

- [28] L. F. Bic and A. C. Shaw, 'Chapter 5: Process and thread scheduling', in *Operating Systems Principles*, 1st ed., Prentice Hall, 2002.
- [29] L. Dagum and R. Menon, 'OpenMP: an industry standard API for shared-memory programming', *IEEE Computational Science Engineering*, vol. 5, no. 1, pp. 46–55, Mar. 1998.
- [30] E. W. Dijkstra, 'Go to statement considered harmful', *Communications of the ACM*, vol. 11, no. 3, pp. 147–148, 1968.
- [31] M. D. McCool, 'Structured parallel programming with deterministic patterns', in *Proceedings of the 2nd USENIX Conference on Hot Topics in Parallelism*, Berkeley, CA, USA, 2010.
- [32] J. Rapp, 'Oversubscription: a classic parallel performance problem', *MSDN Blogs*. [Online]. Available: <http://blogs.msdn.com/b/visualizeparallel/archive/2009/12/01/oversubscription-a-classic-parallel-performance-problem.aspx>. [Accessed: 14-Aug-2013].
- [33] Y. Ling, T. Mullen, and X. Lin, 'Analysis of optimal thread pool size', *SIGOPS Operating Systems Review*, vol. 34, no. 2, pp. 42–55, Apr. 2000.
- [34] C. Campbell, R. Johnson, A. Miller, and S. Toub, *Parallel Programming with Microsoft .NET: Design Patterns for Decomposition and Coordination on Multicore Architectures*. Redmond: Microsoft Press, 2010.
- [35] M. Cole, 'Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming', *Parallel Computing*, vol. 30, no. 3, pp. 389–406, Mar. 2004.
- [36] E. Ayguadé, B. Blainey, A. Duran, J. Labarta, F. Martínez, X. Martorell, and R. Silvera, 'Is the "schedule" clause really necessary in OpenMP?', in *OpenMP Shared Memory Parallel Programming*, M. J. Voss, Ed. Springer Berlin Heidelberg, 2003, pp. 147–159.
- [37] H. González-Vélez and M. Leyton, 'A survey of algorithmic skeleton frameworks: High-level structured parallel programming enablers', *Software: Practice and Experience*, vol. 40, no. 12, pp. 1135–1160, Nov. 2010.
- [38] F. M. David, J. C. Carlyle, and R. H. Campbell, 'Context switch overheads for Linux on ARM platforms', in *Proceedings of the 2007 workshop on Experimental computer science*, New York, NY, USA, 2007.
- [39] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry, and C. Wilkerson, 'Scheduling threads for constructive cache sharing on CMPs', in *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, New York, NY, USA, 2007, pp. 105–115.
- [40] M. Crovella, P. Das, C. Dubnicki, T. LeBlanc, and E. Markatos, 'Multiprogramming on multiprocessors', in *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing, 1991*, 1991, pp. 590–597.

- [41] S. Eyerman and L. Eeckhout, 'System-level performance metrics for multiprogram workloads', *IEEE Micro*, vol. 28, no. 3, pp. 42–53, May 2008.
- [42] V. Donaldson, F. Berman, and R. Paturi, 'Program speedup in a heterogeneous computing network', *Journal of Parallel and Distributed Computing*, vol. 21, no. 3, pp. 316–322, Jun. 1994.
- [43] D. L. Eager, J. Zahorjan, and E. D. Lazowska, 'Speedup versus efficiency in parallel systems', *IEEE Transactions on Computers*, vol. 38, no. 3, pp. 408–423, 1989.
- [44] S. R. Sarangi, B. Greskamp, and J. Torrellas, 'CADRE: Cycle-accurate deterministic replay for hardware debugging', in *International Conference on Dependable Systems and Networks, 2006 (DSN 2006)*, 2006, pp. 301–312.
- [45] D. Skinner and W. Kramer, 'Understanding the causes of performance variability in HPC workloads', in *Proceedings of the IEEE International Workload Characterization Symposium, 2005*, 2005, pp. 137–149.
- [46] W. Kim and M. Voss, 'Multicore desktop programming with Intel Threading Building Blocks', *IEEE Software*, vol. 28, no. 1, pp. 23–31, 2011.
- [47] G. M. Amdahl, 'Validity of the single processor approach to achieving large scale computing capabilities', in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, New York, NY, USA, 1967, pp. 483–485.
- [48] P. Tsogkas, 'Evaluating Skandium's Divide-and-Conquer Skeleton', University of Edinburgh, Edinburgh, 2010.
- [49] V. Strassen, 'Gaussian elimination is not optimal', *Numerische Mathematik*, vol. 13, no. 4, pp. 354–356, Aug. 1969.
- [50] C. A. R. Hoare, 'Quicksort', *The Computer Journal*, vol. 5, no. 1, pp. 10–16, Jan. 1962.
- [51] B. M. Rogers, A. Krishna, G. B. Bell, K. Vu, X. Jiang, and Y. Solihin, 'Scaling the bandwidth wall: challenges in and avenues for CMP scaling', *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3, pp. 371–382, 2009.
- [52] G. E. P. Box, J. S. Hunter, and W. G. Hunter, *Statistics for Experimenters: Design, Innovation, and Discovery*. Wiley-Interscience, 2005.
- [53] M. Bhaduria and S. A. McKee, 'An approach to resource-aware co-scheduling for CMPs', in *Proceedings of the 24th ACM International Conference on Supercomputing*, New York, NY, USA, 2010, pp. 189–199.
- [54] C. Bienia and K. Li, *Benchmarking Modern Multiprocessors*. Princeton University, 2011.