Deconstructing the Overhead in Parallel Applications

Mark Roth
Simon Fraser University
Burnaby, British Columbia
Email: mroth@cs.sfu.ca

Micah J Best
University of British Columbia
Vancouver, British Columbia
Email: mjbest@cs.ubc.ca

Craig Mustard
Simon Fraser University
Burnaby, British Columbia
Email: cam14@cs.sfu.ca

Alexandra Fedorova
Simon Fraser University
Burnaby, British Columbia
Email: fedorova@cs.sfu.ca

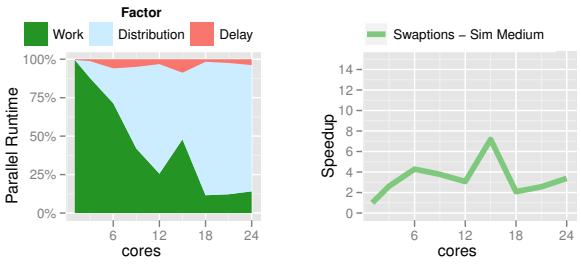
***Abstract*—Performance problems in parallel programs manifest as lack of scalability. These scalability issues are often very difficult to debug. They can stem from synchronization overhead, poor thread scheduling decisions, or contention for hardware resources, such as shared caches. Traditional profiling tools attribute program cycles to different functions, but do not generate immediate insight into issues limiting scalability. Profiling information is very program-specific and is usually processed manually by a human expert in a time-consuming and cumbersome process.**

Our experience in tuning performance of parallel applications led us to discover that performance tuning can be considerably simplified, and even to some degree automated, if profiling measurements are organized according to several intuitive *performance factors* common to most parallel programs. In this work we present these factors and propose a hierarchical framework composing them. We present three case studies where analyzing profiling data according to the proposed principle led us to improve performance of three parallel programs by a factor of 6-20×. Our work lays foundation for new ways of organizing and visualizing profiling data in performance tuning tools.

I. Introduction

As parallel computing becomes more and more prevalent, diagnosis of scalability problems in parallel programs becomes increasingly important. In the recent literature, limiting factors to parallel performance are often deduced based on aggregate and general metrics such as overall speedup, rather than being concretely identified and measured. There is a lack of formal mechanisms for this type of performance analysis and a corresponding lack of automatic tools to aid programmers.

A program is said not to scale if when executed on N cores its speedup is less than a factor of N. There are many factors that can contribute to a lack of scaling including contention for locks, contention for cache and memory system resources, overhead in the runtime work scheduler, inefficient work distribution leaving processors idle, or very simply, there could be not enough available parallelism. Profiling tools measure where a program spends its time and attribute the overhead to functions, synchronization primitives and even to data structures [1], but fail to generate immediately *actionable* insight into issues limiting scalability. Profiling data is specific to a given program and is not organized in a way that allows us to process it *automatically* and diagnose performance issues. Instead, data must be examined manually by an expert. Building diagnosis and optimization solutions that are interactive, automatic or online remain a challenge in most cases.



(a) Factor Breakdown (b) Speedup Chart

Fig. 1. Performance profile of Swaptions from 1 - 24 cores for sim medium input. (a) Factor decomposition (b) Speedup chart

Our experience in tuning parallel performance led us to derive several intuitive *performance factors* that are common to most parallel programs, and the corresponding *hierarchical framework* to organize these factors. In this work, we present a method for decomposing the program overhead according to these factors and demonstrate, using three case studies, how this decomposition method helped us identify performance bugs and improve performance of three PARSEC applications by 6-20× on a 24-core AMD platform. Our work lays the foundation for more effective organization and visualization of profiling data in performance tuning tools.

At the top of our hierarchy are three key factors: *Work* (time spent on 'useful' instructions), *Distribution* (the overhead of distributing work to processors and any load imbalance or idleness), and *Delay* (the slow-down due to contention, or cpu resources spent on 'non-useful' instructions such as failed transactions). These three factors are aggregate values and can be decomposed into finer-grained factors, as will be shown in Section II. A concrete preview of applying the framework can be found in Figure 1(a), which shows the breakdown of the performance factors for the `Swaptions` program in the PARSEC 2.1 suite [2]. The performance factors we chose account for the entire execution time of the program, so charts such as Figure 1(a) succinctly summarize program behaviour as the number of threads increases. We see that in `Swaptions`, *Distribution* constitutes a very large fraction of the overhead. This breakdown immediately tells us that there

are some inefficiencies in how the work is distributed among the cores. Analyzing work distribution lets us quickly pinpoint the problem: load imbalance. Compare this to a typical speedup chart such as in Figure 1(b) which does show a scaling issue, but does not immediately tell us where to look. Addressing the load imbalance reduced the *Distribution* factor and improved performance of `Swaptions` by as much as a factor of six. This process and the results are detailed in Section V.

The primary contributions of our work can be summarized as follows:

- Elicitation of key performance factors contributing to scalability issues in parallel programs.
- A new framework for hierarchically organizing these factors. The hierarchical organization is critical as it allows the inference of some factors when they cannot be measured directly.
- Three case studies showing the framework applied to parallel scalability issues that led us to improve performance of PARSEC benchmarks by as much as 20× in one case.
- Design and implementation of an automatic tuning algorithm *Partition Refinement* that dynamically selects the best partition size in a data-parallel program based on the measurements supplied by the framework.

Categorization of performance overhead is not entirely new. In 1994 Crovella and LeBlanc [3] introduced the concept of *lost cycle analysis* to categorize and account for aspects of parallel performance loss. Our contribution is the hierarchical organization of performance factors (the importance of the hierarchical view is described in the following section), and the definition of the factors that accounts for the realities of modern hardware and software, which have changed drastically in the intervening decades.

The rest of the paper is organized as follows. Section II presents the performance factors and the framework. Section III introduces visual representation of performance factors and explains their hierarchical composition. Section IV explains what changes must be made to a parallel program or runtime library in order to categorize the overhead according to the framework. Sections V-VII present case studies. Section VII also presents the partition refinement algorithm. Section VIII discusses related work and Section IX presents our conclusions.

II. Factors of Parallel Performance Overhead

The premise underlying the overhead decomposition method is that most parallel programs spend their time in one of the following states: doing actual work, scheduling activities (both scheduling and waiting for work), and resource competition. This leads to the highest level categories of our performance factor hierarchy which are as follows:

Work: cycles spent on executing the actual logic of the program; and is the same as the number of cycles the program would execute in a pure serial implementation.
Distribution: cycles spent on distributing the work across processors or waiting for work when none is available.

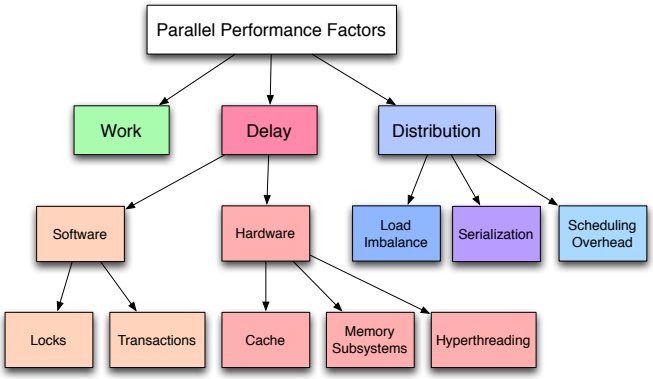


Fig. 2. Deconstruction Framework

Delay: Cycles wasted when the components of the program that run in parallel compete for resources. The resources can be software-constructed (*e.g.*, locks) or can be actual hardware resources (*e.g.*, caches, memory controllers, etc.). These can also be cycles wasted on superfluous calculations such as a failed transaction.

Figure 2 expands on these factors for deconstructing overhead and we now elaborate on the factors comprising *Distribution* and *Delay*.

A. Distribution

Distribution is the process of dealing work to processors, rebalancing the work when needed, and waiting for its completion. These tasks are usually performed by a parallel runtime system such as OpenMP [4], Intel TBB, Cilk [5] or Map/Reduce [6]), which are responsible for creating and mapping tasks or threads to processors, pre-empting processors when needed, and supplying work to idle processors. These scheduling actions can add to the runtime of the program. They are identified in our framework as *Scheduling Overhead*.

Serial sections in the algorithm will affect parallel speedup, so it is crucial for the programmer to be aware of them. This overhead is labeled in our framework as *Serialization*.

If the scheduler assigns work such that some processors are working while others are idle and not waiting on synchronization primitives, performance of the parallel program may suffer. In that case, it is important to know about the number of cycles that are unnecessarily idle when work is available. We refer to this class of overhead as *Load Imbalance*.

Serialization and *Load Imbalance* have a high degree of similarity as they both manifest as idle time on some processors while others are doing work, but it is important that they be separated as each requires a different class of remedies. *Serialization* overhead dictates changes to the algorithm to produce more parallelism. *Load Imbalance* can often be addressed entirely by the scheduler, without changing the underlying algorithm, through better distribution of work.

B. Delay

When work is performed in parallel, performance may be limited by availability of hardware and software resources. When tasks or threads compete for these resources they are unable to make as much progress as when they are running in isolation and so parallel scalability is limited.

Delay in our framework is subdivided into two components: *Software* and *Hardware*. *Software* delay accounts for time spent waiting on synchronization primitives (*e.g.,* locks) or re-executing aborted software transactions. *Hardware* Delay accounts for cycles wasted on contention for resources such as the processor pipeline in hyperthreaded processors, shared caches when the cache miss rate increases because the data is evicted by another core, or other memory subsystem components, such as memory buses, memory controllers, system request queues or hardware pre-fetchers [7].

The category *Memory subsystem* also includes the memory access overhead and communication latencies on systems with non-uniform memory (NUMA) hierarchies. On NUMA systems the latency of data exchange depends on relative locations of the cores involved in the exchange. Furthermore, accesses to a remote memory node take more time than local accesses.

III. Visual Representation of Performance Factors

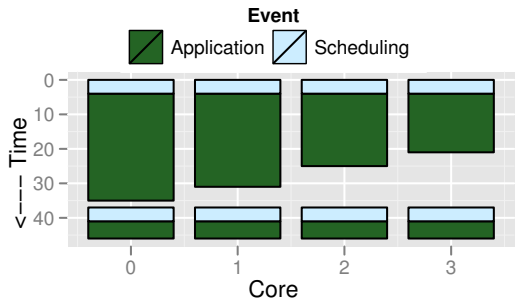


Fig. 3. Activity Graph example

Before we explain how we measured the performance factors introduced in the previous section and how we used the resulting analysis to track scalability issues, we introduce the concept of an *activity graph*, which visually demonstrates how the running time is subdivided according to various components of the program.

Figure 3 shows an example of an activity graph. The y-axis denotes time and the x-axis shows the physical cores running the application. Colour-coded blocks show how the running time is subdivided between *Application logic*, *Scheduling*, and *Idleness*. Application logic in this context corresponds to the combined *Work* and *Delay* factor as the application logic may be slowed down due to contention. Scheduling corresponds to the *Scheduling overhead* factor in the *Distribution* category, and idleness corresponds to *Load Imbalance* (also in the *Distribution* category).

The total time accounted by the activity graph, T_{total}, is:

$$T_{total} = T_p \times P \tag{1}$$

Where T_p is the absolute start to finish time of the application and P is the number of cores. Each of the performance factors in the framework accounts for a fraction of T_{total}, and so we naturally have the following relationship:

$$Work + Delay + Distribution = 100\% \tag{2}$$

The real benefit of expressing each of these performance factors as a fraction of T_{total} is that it creates a standard metric. Using this standard metric allows us to make performance and overhead comparisons between different platforms, implementations, program regions, and core counts. It also allows for the direct comparison between different sources of parallel overhead.

Ideally, we would like to see very little idle and scheduling time on an activity graph. The portion that these two components cover the activity graph constitutes the *Distribution* factor of the parallel run time and is calculated as:

$$\textit{Distribution} = \frac{Scheduling + Idle}{T_p \times P} \tag{3}$$

As mentioned earlier, the *Work* time is equal to the amount of time that application would take to run serially, and therefore the *Work* factor is:

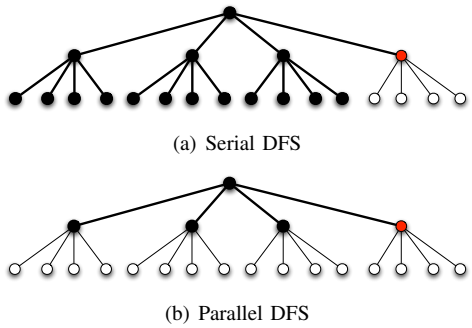
$$\textit{Work} = \frac{T_s}{T_p \times P} \tag{4}$$

Where T_s is the serial run time.

Likewise, the *Delay* component can be computed by taking the difference between the sum of the *Application logic* in the parallel and serial executions. Alternatively, any one of these three components can be inferred if the other two are known since the sum of all three must add to 100%. This rule generalizes to all levels of the hierarchical framework. The overheads represented by a set of sibling nodes add up to the overheads represented by their parent. For instance, *Scheduling Overhead*, *Load Imbalance* and *Serialization* must add up to the overhead accounted by *Distribution*. The identification of these subcomponents in some cases can be done with the use of finer grained labelling in the activity graph. In other cases such as distinguishing cache contention from memory controller contention, heuristic measures would currently need to be used. However, the use of a hierarchical system allows us to place bounds on how much those factors are limiting scalability of the program. These complexities are explained further in Section IV

Before concluding this section, we bring up two important points about negative delay and super-linear speedup.

An interesting characteristic of *Delay* is that it can be negative under some circumstances. The parallel version may execute less code by finding an early exit condition as with the parallel depth first search as seen in Figure 4. Properties of hardware may also produce this effect as when cores that share the same cache exhibit a cooperating behaviour by sharing data



(a) Serial DFS

(b) Parallel DFS

Fig. 4. Example showing how a parallel depth first search could execute less code by finding an early termination condition. Nodes checked are highlight black. Target node in red.

or when the application is too large to fit in a single CPU's cache but is small enough to fit into the aggregate cache of multiple CPU's.

If the *Delay* is negative but the *Distribution* is zero, then the *Work* must be greater than one. A *Work* greater than one indicates that super-linear speedup occurred. However, having negative delay does not guarantee super-linear performance as the performance gain can be offset by performance loss of *Distribution*. This highlights the need for fine-grained performance factors. It is possible that an application may exhibit P times improvement on P cores, but be capable of having a greater than P times improvement. A situation that would be undetectable without looking for cases such as negative *Delay*.

IV. Implementation

In this section we describe how to measure and categorize the overhead according to the framework in a practical implementation. Some of the overhead sources, especially those induced by the software, can be measured directly. Those stemming from the hardware are difficult to measure directly on modern systems and thus need to be inferred. Experiments were done on a 4 socket AMD Opteron 8435 running Gentoo Linux with Kernel 3.2.1. Each socket contained 6 cores for a total of 24 cores.

A. Measuring software-induced overhead

Measuring overhead related to software (*Distribution* and *Software Delay*) is relatively simple. The software needs to be instrumented to measure the timings of all program components (functions) that define the boundaries of various performance factors related to work distribution and software contention: e.g., *Scheduling Overhead*, *Load Imbalance*, etc.

Load Imbalance and *Serialization* can be measured by counting the cycles when a core is idle or busy-waiting for work (and not waiting on synchronization primitives) while another core (or cores) are busy doing work[1]. Distinguishing between *Load Imbalance* and *Serialization* is tricky, as they both show up as processor idle cycles while a thread is either busy-waiting or sleeping. The way to address this issue is to label the idle cycles as *Load Imbalance* or *Serialization* depending on *how* the parallel work is generated in a given section of code. For example, in the parallel section of code created from a *parallel-for* loop in OpenMP or a *map* directive in a Map/Reduce framework, idle cycles would be labeled as *Load Imbalance*, because there is a high probability (although not always) that the idle cycles could be created by inefficient distribution of work by the schedulers. For instance, when scheduling data-parallel operations, if the scheduler gives an equal number of work items to each processor, but the amount of computation per work item is not fixed or if some processors are running slower than others due to system asymmetry, some processors will end up being idle while others are doing work. In that case, work must be re-distributed, as is commonly done in Map/Reduce frameworks. Labelling these idle cycles as *Load Imbalance* will signal the programmer (or the auto-tuning algorithm) that the parameters affecting work distribution may need to be altered.

The easiest way to implement the overhead profiling and categorization is inside a parallel runtime, where parallelism is extracted automatically by the parallelizing runtime or compiler. In this case, parallelism is harvested by way of directives in the serial code (e.g., parallel-for in OpenMP) or by explicitly invoking the interfaces that assist in parallelization (e.g., in Intel TBB). This gives us a very clear separation of application code (code required in a serial implementation) from parallel library code. For categorization of parallelization overheads, it is relatively easy for the developer implementing the library to add the instrumentation and labelling according to the desired categories.

When parallelization is performed directly by the programmer using low-level tools such as pthreads, instrumentation can be inserted automatically, by typical profiling tools or by a binary instrumentation tool such as Pin. The programmer, however, needs to provide a mapping between a function name and the overhead type. This can be a cumbersome process, so our hope is that the proposed overhead-deconstruction framework will be primarily implemented *de facto* in massively emerging parallel programs and frameworks, as opposed to being added to existing programs as an afterthought.

Identifying synchronization-contention overhead deserves special discussion. Overall, it is trivial to measure the time spent while waiting on locks, barriers or other synchronization primitives. It is worth noting, however, that some of that overhead comes from the hardware, such as coherence protocol latency when multiple cores are using the lock. However, for the purposes of fixing scalability issues it is more convenient to classify this overhead as *Lock Contention* as opposed to *Hardware*, and so we treat it as such.

Another interesting point related to synchronization is how to treat busy-waiting and blocking. In the implementation of synchronization primitives, blocking is sometimes used to give up the processor when waiting for the primitive takes too long. While blocking is definitely a part of lock contention, it is also arguably a part of scheduling, as effectively argued in

[1]We assume that the program is not I/O-bound.

the work by Johnson [8]. Essentially, the action of giving up the processor to make way for other threads is a scheduling activity, and it may be more convenient, for performance debugging purposes, to treat it as such. This is what we chose to do in our implementation (although another implementation may make a different choice), and so processor-idle cycles occurring because a thread blocks on a lock show up as *Load Imbalance*.

The key take-away from this section is that all performance factors induced by software can be measured directly by automatic or manual instrumentation. Next we discuss how to infer hardware-related overheads.

B. Inferring hardware-related overheads

Hardware overheads are difficult to quantify, because the additional cycles that they generate cannot be measured directly. Despite hundreds of hardware performance events available for monitoring on modern processors, it is very difficult to determine precisely how many cycles are being wasted due to contention for caches or other hardware in the memory system. Performance modelling can be used to estimate hardware-related overhead, as was done in [9], but given complexity of a typical system no model can be completely accurate.

We observe that hardware contention related to parallelization will show up as the increase in the time attributed to the total *Delay* factor. If both total delay and software delay factors are known, then the hardware overhead is simply the difference between the two. However, to compute the total delay, we must know the values of *Work* factor and *Distribution* factor. Since the *Distribution* factor is entirely software related, we can compute this value. And the *Work* factor is the serial time divided by T_{total} (time parallel $\times$ number of cores). We can therefore infer the hardware contention portion.

Although this implies that *Hardware* overhead cannot be measured precisely without having a serial execution time as a reference point, this does not discount implementation of auto-tuning algorithms for "online" scenarios where this reference point is hard to obtain. For long-running parallel programs that *iteratively* execute parallel sections many times (e.g., animation, simulation, image analysis and many others), the runtime system can search the parameter configuration space by varying their settings and observing how they affect the *changes* in the *Hardware* overhead. Recent work has shown that after enough repetitions, we are statistically likely to arrive at the optimal configuration with a high probability [10].

C. Our implementation

In order to perform our evaluations of this model, we were required to hand instrument the timing events into the code base. This process is simple for programs which spawn one worker thread per core as the chance of a thread being pre-empted by the OS is low. If a thread were to be interrupted while a timer is running, then the time the thread was spent suspended would also need to be taken into account. We instrumented selected benchmarks from PARSEC suite that fit the one-worker-per-core model. For our timing implementation, we measured the duration of each code segment which would also have been required to be in the serial implementation. The time accumulated by all these code segments would equal *Work* + *Delay*. Since *Work* is the serial completion time, we can infer both the *Distribution* and *Delay* components with the given timing information. The program completion time is measured as the time to complete the Region of Interest (ROI) [2]. All reported measurements were averaged over five runs. The overhead induced by the added timing code was less than standard deviation of the run, unless otherwise noted.

Sections V-VII will show how deconstructing parallel performance overheads even at a coarse level can provide valuable insight into program behaviour. Sections V and VI will show how the factor analysis can be used in "manual" performance debugging. Section VII will show how it can be used to implement an algorithm that automatically chooses the best program configuration parameters based on repeated measurement.

V. Load Imbalance in *Swaptions*

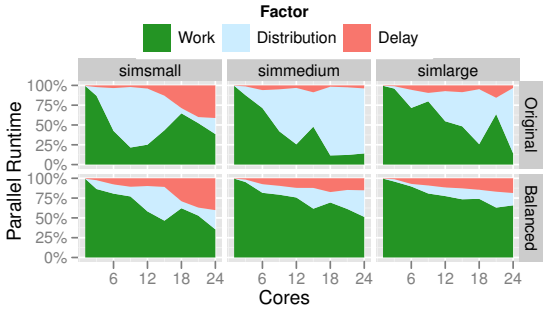


Fig. 5. Breakdown of overhead in *Swaptions* according to performance factors: original pthread implementation and the improved load-balanced implementation (bottom)

Our first case study looks at how the framework helps us manually tune the *Swaptions* benchmark from the PARSEC [2] suite. In this benchmark, Monte Carlo simulations are used to price swaptions in a financial portfolio. The only synchronization mechanism for the program is locks. Running a large simulation with 8 cores requires only 23 lock acquisitions in total [2]. This program scales well with a small number of cores. However, with a larger number, the parallel performance suffers greatly; obtaining only about a 3.5 times speedup with 24 cores on the large input set. As the only synchronization in the program is locks, a programmer may naively conclude that the poor parallel performance is due to lock contention. However, by looking at the parallel factor analysis (top half of figure 5) we see that poor scalability is mostly a *Distribution* issue.

If fine-granular timing information were incorporated into the parallel runtime environment (pthreads, in this case), then we would be able to automatically derive how much of the distribution factor was due to overhead, serialization or imbalance. Even without this detailed breakdown, analyzing the swaptions activity graph (Figure 6) tells us that the work

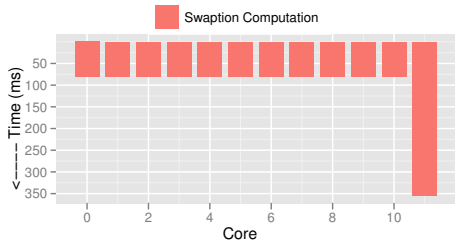


Fig. 6. *Swaptions* activity graph showing load imbalance of original pthread implementation

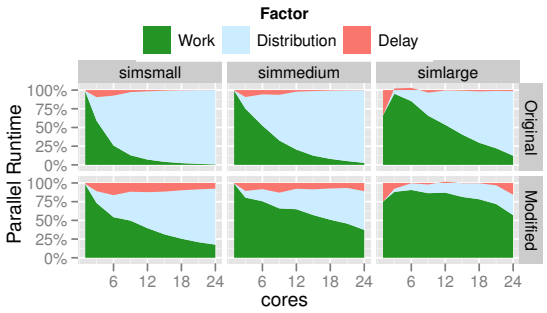


Fig. 7. Parallel performance decomposition of Streamcluster with simlarge, simmedium, and simsmall inputs

is not being distributed evenly amongst the cores. Taking advantage of our knowledge that there is no sizeable serial region in this program, we conclude that the culprit is load imbalance, and looking at the code that is responsible for distributing chunks of work in the pthread implementation, we see why the load imbalance is occurring.

```
int chunksize = items/nThreads;
int beg = tid*chunksize;
int end = (tid+1)*chunksize;
if (tid == nThreads -1 )
    end = items;
for(int i=beg; i < end; i++) {
...
}
```

What the code attempts to do is evenly distribute N swaptions to P threads by picking a chunk size N/P and giving each thread N/P swaptions to compute. The very last thread handles any odd swaptions left over. This works fine if there is a large number of swaptions and a few cores, but that is not the case for this benchmark. For example, the *simmedium* input set contains only 32 swaptions. If there were 16 threads, each thread would compute two swaptions each and so the work would be evenly distributed among threads. But if there were, for instance 17 threads, the first 16 would compute one swaption and the 17th thread would compute the remaining 16, resulting in a very large load imbalance. Moreover, this explains why we see the scaling performance zigzag with the number of cores, as the workloads become more or less balanced. Fixing this imbalance and distributing work more evenly across the threads (as shown the top part of the figure) improves performance by as much as a factor of six in some cases.

This same load imbalance issue was also discovered in the work of *Thread Reinforcer* [11]. However, our methodology clearly identified a scheduling issue from the very start, whereas *Thread Reinforcer* determined the best number of cores to run on.

VI. Inefficient Barrier Implementation

Streamcluster is a data-mining application from the PARSEC benchmark suite. We observed that this particular benchmark had very poor scaling when run on a large number of cores and that parallel performance gains were only realized when using a very large input. Previous work on evaluating the performance of PARSEC note that for *Streamcluster*, "95% of compute cycles are spent finding the Euclidean distance between two points" and that "Scaling is sensitive to memory speeds and bus contention" [12]. However, our analysis reveals that the performance issue for *Streamcluster* does not stem from where compute cycles are being spent, but rather where they are *not* being spent.

It is immediately apparent from the top part of Figure 7 that *Distribution* is the primary cause of performance loss. Examining the activity graph of *Streamcluster*, Figure 8, gives us a further insight and reveals a striking lack of activity, as indicated by the empty portions of the activity graph. This idleness could indicate either blocking or executing in the OS, as these cycles would not be captured by our simple user-level implementation.

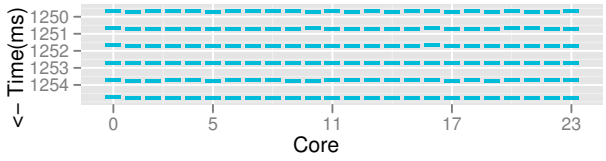
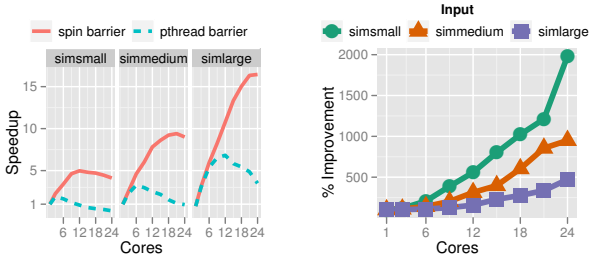


Fig. 8. Activity graph of streamcluster covering a 5 millisecond interval. Visible execution chunks are associated with calculating a distance cost function (streamcluster.cpp:1036-1067). Additional execution chunks are present, but are too small to be visible.

Observing that *Streamcluster* uses barriers for synchronization we suspected that this may be the cause of the large amounts of inactivity. Code analysis revealed that the pthreads library uses a *yielding* implementation of a barrier, where a thread voluntarily gives up the CPU if it is unable to execute the barrier after initial spinning. Performance begins to suffer when many successive barriers are used in a short timeframe. As more threads are added to the system, the time they take to synchronize increases, and it becomes increasingly likely that a thread will yield after a time-out at the barrier. When a thread yields and resumes, it will be delayed in starting the next stage and arrive at the next barrier late, causing the other

threads to spin too long and eventually yield, creating a vicious cycle.

As explained earlier, in our implementation we chose to represent blocking events (even those resulting from failed synchronization attempts) as *Load Imbalance*, which falls under the *Distribution* factor. Therefore, the inefficiencies associated with excessive yielding on the barrier show up in Figure 7 under the *Distribution* category.



(a) Speed up (b) Improvement over original

Fig. 9. Performance comparison of pthread barriers and spin barriers in the Streamcluster benchmark.

The detrimental effects of pthread barriers on this particular application can be alleviated by replacing the yielding pthread barriers with a spinning implementation. The bottom of Figure 7 shows the improved breakdown of execution time, as the *Work* factor takes a much larger portion of time than *Distribution* and *Delay*. Figure 9(b) shows the performance difference between the two implementations. The 24-core performance of the `simsmall` input set represents a twenty times performance improvement as the pthread implementation is many times slower than the serial version. Further investigation reveals that the remaining *Distribution* factor that appears even with the spin barrier is due to serial sections of the application.

Even though the spin barrier implementation shows tremendous improvements over the pthread barrier for this benchmark in this case it is certainly not a solution for all barrier related problems. If, for example, the number of threads exceeds the number of available cores, then the performance of a spin barrier can degrade drastically. A dynamic locking primitive that switches between spinning and blocking implementation depending on the number of runnable threads has been proposed by Johnson [8] and could be used in this case.

VII. Partition Refinement

In the previous sections we showed how the performance analysis framework can be used for the manual tuning of parallel programs. In this section we show how it can be used for automatic tuning. We incorporate it into an online algorithm to find an optimal configuration for runtime parameters.

Our partition refinement algorithm addresses the problem of dynamically determining the right size of a data partition in a data-parallel program. If the data partition is too small, then the cost of creating and scheduling the tasks is large relative to the execution time of the task. In other words, we

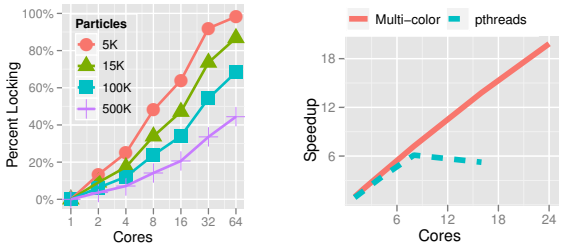


Fig. 10. (a) Percentage of cells requiring locking as number of participating cores increases. (b) Speedup comparison of the implementation using *Multi-colour Partitioning* vs. the original pthread implementation (simlarge). Pthread version is restricted to core counts of powers of two.

suffer excessive *Scheduling Overhead*. On the other hand, if the task is too large, we may find that some processors are idle and unable to steal work, while others are working on very large tasks. This stuation would show up as *Load Imbalance*. During the runtime, our tuning algorithm will measure the *Scheduling Overhead* compared to the *Load imbalance* and will dynamically change the size of the data partition to arrive at the optimal task size.

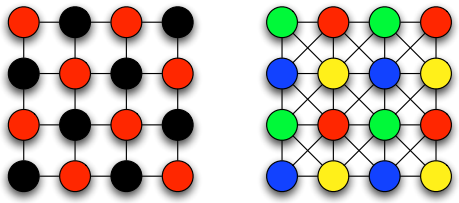
Since the partition refinement algorithm works by observing the performance factors and iteratively adjusting the partition size, it is applicable to programs that repeatedly re-execute data-parallel code sections. The repeated processing pattern occurs often in simulation algorithms, video games, multimedia and many other interactive and soft-realtime applications.

A. Fluidanimate

To illustrate the algorithm, we use the *Fluidanimate* benchmark from the PARSEC suite. This application operates on a 3D grid to simulate an incompressible fluid by modelling the movements of particles in space. The fluid space is divided into cells, each holding a maximum of 16 particles.

The cell size is chosen in such a way that the particles in one cell can only influence the particles in the adjacent cells. If the cells are divided for parallel processing among the threads, we must make sure to avoid race conditions, as it is possible that some cells could be modified concurrently. Since each cell has a fixed 'influence radius' mutual exclusion requirements for a cell are predictable from its coordinates alone. A common technique to deal with the mutual exclusion requirement can be seen in the pthread implementation of *Fluidanimate*. The cells of the simulation are divided into roughly P equal partitions where P is the number of available threads. Cells that lie on the border of these partitions must be locked before they can be modified. Using locks in this scenario can severely limit scalability. In Figure 10(a) we can see that as we increase the number of partitions, the percentage of cells that require locking increases to the point where almost every cell requires a lock.

We applied a multi-color partitioning method [13], an extension of red-black partitioning, which we use to eliminate the locks from the original version included with PARSEC.



(a) red black partitioning (b) Multi-colour partitioning

Fig. 11. Unlike red-black partitioning, *Multi-colour Partitioning* applies to problems where exclusive access constraint includes diagonal cells.

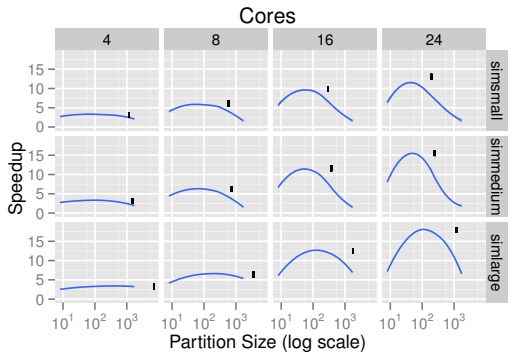


Fig. 12. Speedup values are highly dependent on partition size. No single partition size will be optimal for all input and core counts. Black ticks indicate partition sizes required to give each core 8 partitions.

Multi-colour partitioning is a variant and generalization of red-black partitioning, applicable for data in any number of dimensions and with diagonal dependencies of the cells in the grid. With *Multi-colour partitioning*, the cells are divided into small partitions with a minimum size of $2 \times 2 \times 2$ when considering three dimensions. These partitions are then coloured such that no partition is adjacent to another partition with the same colour. The computation is then carried out in a sequence of stages where each stage processes all the partitions of the same colour. Each colour partition can be computed independently without any synchronization. In order to satisfy the exclusive access constraint of all neighbour cells, including diagonal cells, eight colours are required with 3D grids. Figure 11 shows the difference between the *red-black* colouring and *Multi-colour Partitioning* for a 2D grid.

We applied *Multi-colour Partitioning* to Fluidanimate using OpenMP and the consequent elimination of locks allowed for distribution of work at a finer granularity. The speedup of parallel regions using *Multi-colour Partitioning* is shown in Figure 10(b). Values are shown for combined times of `Compute Densities` and `Compute Forces` sections of the application as *Multi-colour Partitioning* was not applicable to other regions. These two sections make up the bulk of the execution time for the benchmark.

As mentioned previously, there is a trade-off between *Scheduling Overhead* and *Load Imbalance*. To highlight this effect, we magnified the possible work imbalance by padding the simulation space with empty cells. This was done by increasing the simulation space by a factor of 2 in each dimension, thereby increasing the total number of cells by a factor of 8. This enlarged simulation space is used for the partition refinement experiments. Figure 12 shows combined speedup values, relative to the serial implementation, of the `Compute Densities` and `Compute Forces` regions of the code over varying partition sizes, program inputs, and core counts.

We observe that there is not a single fixed partition size that achieves the best speedup across all core counts and input sizes. The black tick marks on the graph show what partition size would have been chosen if each stage of the *Multi-colour Partitioning* had been divided into $8 \times P$ partitions, where P is the number of cores. This value is chosen to demonstrate that simply dividing the work into some multiple of P cannot find the optimal value across a wide range of parameters.

B. Partition Refinement Algorithm

As mentioned previously, we focus on applications that exhibit periodic behaviour. We are therefore able to take advantage of performance metrics measured in one iteration to inform what changes need to be made in the next. The two metrics that we are interested in are *Scheduling Overhead* and *Load Imbalance*. If the *Scheduling Overhead* is large, this is a signal that the partition size should be increased. On the other hand, if the *Load Imbalance* is large then this indicates that partition size should be decreased. However, these adjustments must be made intelligently in order for the algorithm to converge quickly and avoid large oscillations.

To satisfy these objectives, we imagine the worst case imbalanced scenario where there are P cores and N items, but all the work items are assigned to only one core. In order to rectify the situation, we must, at the very least, divide N items into P partitions. Another way to perform that operation is to decrease the partition size by a factor of P (assuming that the partition size is less than or equal to N). *Load Imbalance* is, therefore, considered as a force that decreases the partition size. If there is a maximum imbalance, then the partition size is reduced by a factor of P. If there is no imbalance, then partition size is not reduced. We decrease the partition size in proportion to the measured *Load Imbalance* as the fraction of the maximum possible *Load Imbalance*.

The maximum possible *Load Imbalance* is simply a function of the number of cores P and can be computed as follows. Suppose that all work is done by a single core. We can compute the *Work* factor by summing the execution of all the work kernels and dividing it by T_{total}. Assuming conservatively that load imbalance is the only cause of overhead, all the execution time T_p will be used to execute the work kernels (load imbalance shows up as idle time). So the aggregate time spent in work kernels is simply T_p. The *Work* factor, in this case, becomes $\frac{T_p}{T_p \times P}$, or simply $\frac{1}{P}$. Given our conservative assumption that *Load Imbalance* is the only factor besides *Work*, then the maximum *Load Imbalance* is $(1 - \frac{1}{P})$.

Just as *Load Imbalance* signals the need for a decrease in

the partition size, *Scheduling Overhead* signals a need for an increase. The equations below summarize how the partition size is adjusted depending on the the values of *Load Imbalance* and *Scheduling Overhead*:

$$decrease\% = P \cdot \frac{LoadImbalance}{(1 - \frac{1}{P})} \tag{5}$$

$$increase\% = P \cdot \frac{Sch.Overhead}{(1 - \frac{1}{P})} \tag{6}$$

$$size_{new} = size_{old} \frac{1 + increase\%}{1 + decrease\%} \tag{7}$$

$$size_{new} = size_{old} \cdot \frac{1 + P \cdot \frac{Sch.Overhead}{(1-\frac{1}{P})}}{1 + P \cdot \frac{LoadImbalance}{(1-\frac{1}{P})}} \tag{8}$$

Simplification yields the formula:

$$size_{new} = size_{old} \cdot \frac{P^2 \cdot Sch.Overhead + P - 1}{P^2 \cdot LoadImbalance + P - 1} \tag{9}$$

This removes the possibility of dividing by 0 when *Load Imbalance* is 0, except when the core count is 1; in which case the partition refinement algorithm would not be needed.

C. Results

Since the partition refinement algorithm requires an initial partition size, we tried two different initial starting sizes: the smallest (2x2x2) and the size obtained by dividing the data into $2 \times P$ partitions, where P is the number of cores.

For both starting configurations, we executed with all given input files and core counts. These results are compared to the best measured speedup for each input and core configuration, which was determined experimentally. The summary of the results is shown as a histogram in Figure 13. For the majority of the input configurations, the partition refinement algorithm works well and is able to converge to a value within 10% of the best achievable performance as measured in Figure 12. Figure 14 summarizes the performance improvements that the

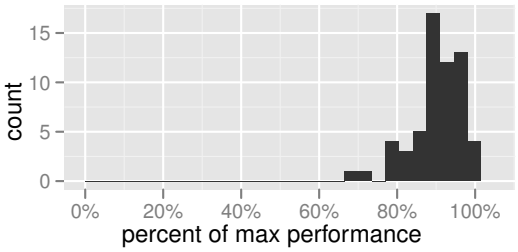


Fig. 13. Histogram of experimental data points under partition refinement. In most experiments partition refinement performs very closely to the optimal configuration.

partition refinement algorithm achieves over the implementation that partitions the data statically. We compare to the two static partitioning scenarios that are used as the baseline for the partition refinement algorithm: the smallest possible (*small start*) and $2 \cdot P$. We observe that in most cases performance improvements are very substantial: 20-80%.

Effective decomposition of profiling measurements into actionable performance factors, *Scheduling Overhead* and *Load Imbalance* in this case, enabled us to quickly isolate performance-limiting factors and design a simple algorithm that finds the best setting for a tuneable parameter across many inputs and core counts.

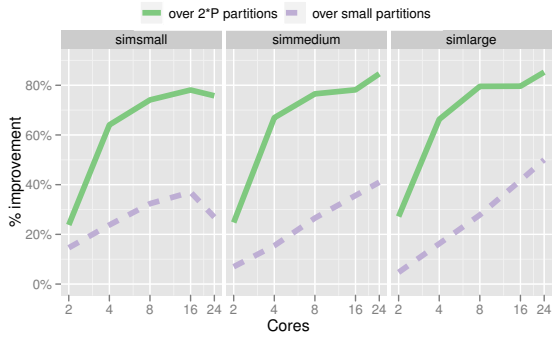


Fig. 14. Summary of improvement of dynamic partition refinement over static partitioning.

VIII. Related Work

There is a plethora of profiling tools that allow gathering and examining performance data, but to the best of our knowledge none of them offers the analysis similar to what we proposed in this work. Profilers can tell us how much time the program spends executing different functions, waiting on locks, or idling the CPU, but come short of organizing the data according intuitive performance factors. The closest works that comes to achieving this goal are Lost Cycle Analysis [3] (LCA), by Crovella et al, and Speedup Stacks [14].

LCA decomposes the execution time into the following components: load imbalance, insufficient parallelism (serialization), synchronization contention, cache contention, and resource contention. While these factors are similar to some used in our framework, our contribution is a *complete* hierarchical framework that accounts for all sources of work and overhead and can be thus used as the basis for automatic performance tuning. Completeness and hierarchical organization are crucial, because not all of the performance factors can be measured directly, and must be inferred from others. LCA also required completeness, however, the authors were fortunate enough to work with a platform [15] that allowed for direct measurement of all hardware delays, which cannot be done on current hardware.

Speedup Stacks is a contemporary paper that identifies the lost scaling performance using approximately the same categories which we identify. This work exemplifies what what our framework would like to measure if we had perfect knowledge about the system. Speedup Stacks was able to directly attribute cycles to fine grained components of the application with the use of a simulator. Our method is limited

in that only some performance categories can be measured. However, with the use of a hierarchy, we were able to infer other performance metrics. Furthermore, we demonstrated that the information gathered is sufficient to drive an adaptive algorithm which balances synchronization overhead with load imbalance.

Representing all performance factors as the fraction of total time, as is done in our framework, helps design automatic tuning algorithms, such as partition refinement, that examine the relative contribution of each factor and tune parameters based on this relation.

Another area of related work includes algorithms that automatically discover the right configuration parameters for the parallel program. Examples of the more recent work in this area include Thread Reinforcer [11] and Feedback-Driven Threading [9]. In both cases, the algorithms aim to find the optimal number of threads in a parallel program, and show a good example of the kind of optimization that could be built on top of our framework. The strength of our framework is that it can be used to tune many parameters that are responsible for various sources of overhead.

A more general approach to parameter tuning is via machine learning. Brewer investigated machine learning techniques that find good configuration parameters for the application [16]. Ganapathi, et al. also apply a machine learning technique to tune application parameters [17]. Ganapathi's technique reduces a large search space of 4×10^7 parameters down to 1500 random samples, and finds configurations for two programs that are within 1% and 18% of the version optimized by an expert. However, this is still an offline technique that takes hours. Our factor decomposition framework could be used to further guide machine learning techniques to reduce the search space and identify the most crucial tuning parameters.

IX. Conclusions

Through our experience of finding and fixing scalability bottlenecks in parallel applications, we discovered that performance debugging can be substantially simplified if profiling measurements are organized according to several intuitive performance factors, common to most parallel programs. The key performance factors we proposed are *Work*, *Delay* and *Distribution*; each of them is further decomposed into additional sub-factors, forming a hierarchy. As the key contribution of our work we presented and described this hierarchy.

We further showed how the performance factor analysis can be used in practice for fixing scalability issues in parallel applications. We discovered and eliminated an inefficient barrier implementation in one application and improved a work distribution algorithm in another. These changes led to performance improvements of 6-20×. Finally, to demonstrate how the framework can be used for *automatic* performance analysis and tuning, we presented a partition refinement algorithm that repeatedly compares *Scheduling Overhead* and *Load Imbalance*, the components of *Distribution*, to balance between the two and obtain the optimal partition size in data-parallel applications. This algorithm performs 20-80% better, in most cases, than simple static partitioning and is robust across different inputs and core counts.

Our hope is that the instrumentation required to measure the proposed performance factors is incorporated in future parallel libraries, facilitating performance debugging and enabling proliferation of automatic performance tuning techniques.

References

[1] A. e. a. Pesterev, "Locating cache performance bottlenecks using data profiling," in *Proceedings of the 5th European conference on Computer systems*, ser. EuroSys '10, 2010, pp. 335–348.

[2] C. e. a. Bienia, "The parsec benchmark suite: Characterization and architectural implications," in *PACT*. ACM, 2008, pp. 72–81.

[3] M. Crovella and T. LeBlanc, "Parallel performance using lost cycles analysis," in *Proceedings of the 1994 conference on Supercomputing*. IEEE Computer Society Press, 1994, pp. 600–609.

[4] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *Computational Science & Engineering, IEEE*, vol. 5, no. 1, pp. 46–55, 1998.

[5] R. e. a. Blumofe, *Cilk: An efficient multithreaded runtime system*. ACM, 1995, vol. 30, no. 8.

[6] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[7] S. e. a. Zhuravlev, "Addressing shared resource contention in multicore processors via scheduling," in *ASPLOS 15*, 2010, pp. 129–142.

[8] F. e. a. Johnson, "Decoupling contention management from scheduling," in *ASPLOS 15*, 2010, pp. 117–128.

[9] M. e. a. Suleman, "Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on cmps," *ACM SIGPLAN Notices*, vol. 43, no. 3, pp. 277–286, 2008.

[10] P. e. a. Radojkovic, "Optimal Task Assignment in Multithreaded Processors: A Statistical Approach," in *ASPLOS 17*, 2012.

[11] K. e. a. Pusukuri, "Thread reinforcer: Dynamically determining number of threads via os level monitoring," in *Workload Characterization (IISWC), 2011 IEEE International Symposium on*. IEEE, 2011, pp. 116–125.

[12] M. e. a. Bhadauria, "Understanding parsec performance on contemporary cmps," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. IEEE, 2009, pp. 98–107.

[13] L. Adams and J. Ortega, "A multi-color sor method for parallel computation," in *Proceedings of the International Conference on Parallel Processing*, vol. 3. Citeseer, 1982, p. 53.

[14] S. E. et al., "Speedup stacks: Identifying scaling bottlenecks in multi-threaded applications," in *ISPASS'12*, 2012, pp. 145–155.

[15] T. Dunigan, "Kendall square multiprocessor: Early experiences and performance," *Oak Ridge National Laboratory Technical Report ORNL/TM-12065*, 1992.

[16] E. A. Brewer, "High-level optimization via automated statistical modeling," in *PPoPP 5*. New York, NY, USA: ACM, 1995, pp. 80–91. [Online]. Available: http://doi.acm.org/10.1145/209936.209946

[17] A. e. a. Ganapathi, "A case for machine learning to optimize multicore performance," in *HotPar 1*. Berkeley, CA, USA: USENIX Association, 2009, pp. 1–1. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855591.1855592