THE Multicore ASSOCIATION™

MULTICORE PROGRAMMING PRACTICES

Working Group Chairs: **David Stewart & Max Domeika**
President: **Markus Levy**

Table of Contents

Chapter 4: Implementation and Low-Level Design

Chapter 5: Debug

Chapter 6: Performance

FOREWARD: MOTIVATION FOR THIS MULTICORE PROGRAMMING PRACTICES GUIDE

Markus Levy, Multicore Association President

The Multicore Association was founded in May 2005, back in the pioneering days of multicore technology. To this day, the primary goal of the MCA is to develop an extensive set of application programming interfaces (APIs) and the establishment of an industry-supported set of multicore programming practices and services.

When the Multicore Programming Practices working group (MPP) formed 4+ years ago, it was (and still is) a known fact that C/C++ was going to be the predominant programming language for at least the better part of the next decade. While the industry was waiting for long-term research results to change the programming infrastructure, the multicore programmability gap was continuously expanding, pointing out an obvious need for a group of like-minded methodology experts to create a standard "best practices" guide.

Through the guidance of Max Domeika (Intel) and David Stewart (CriticalBlue), the Multicore Association created this one-of-a-kind document, providing best practices for writing multicore-ready software using C/C++ without extensions, describing a framework of common pitfalls when transitioning from serial to parallel code, and offering insight to minimizing debugging efforts by reducing bugs.

Of course, the only problem with creating this kind of guide is that it's really a work in progress. As a matter of fact, even at the onset of this monumental project, we realized that the biggest challenge was deciding what not to include (at least in the first version). So therefore, on one hand I personally wish to thank the working group members, and on the other hand, I'm inviting you to step forward and join the Multicore Association effort to help evolve this MPP Guide.

Primary contributors to the Multicore Programming Practices Guide include:

Hyunki Baik (Samsung)
François Bodin (CAPS entreprise)
Ross Dickson (Virtutech/Wind River)
Max Domeika (Intel)
Masato Edahiro (Nagoya University)
Scott A. Hissam (Carnegie Mellon University)
Skip Hovsmith CriticalBlue)
James Ivers (Carnegie Mellon University)
Markus Levy (EEMBC and The Multicore Association)
Ian Lintault (nCore Design)
Stephen Olsen (Mentor Graphics)
Tobias Schuele (Siemens Corporate Technology)
David Stewart (CriticalBlue)
Peter Torelli (EEMBC)

CHAPTER 1: INTRODUCTION & BUSINESS OVERVIEW

1.1 Introduction

Multicore processors have been available for many years; however, their widespread use in commercial products and across multiple market segments has only recently occurred. This ‘multicore era’ transpired because processor architects ran into power & performance limits of single core processors. The multicore era shifts more of the responsibility for performance gains onto the software developer who must direct how work is distributed amongst the cores. In the future, the number of cores integrated onto one processor is expected to increase, which will place even greater burden on the software developer. Like many new eras in the computer industry, numerous supporting software development tools and technologies have been introduced with the aim of helping developers obtain maximum performance benefit with minimal effort. One of the potential challenges that may hinder developers is the conflict between an intrinsic quality of developing for multicore processors and the inertia of existing software.

Prior to the multicore processor era, developers obtained performance increases through processor upgrades and the inherent clock frequency boosts and microarchitecture improvements. These upgrades required minimal software changes. Obtaining performance increases in the multicore era requires developers to invest in significant software modifications to, in effect, transform current sequential applications into parallel ones. This modification is nontrivial and introduces new challenges, spanning the traditional development phases of program analysis, design, implementation, debug, and performance tuning.

In stark contrast, the inertia of existing software is undeniable. To meet tight deadlines, development projects rarely have the freedom to make wholesale changes to applications, and instead limit risk by minimizing them. This means that a developer considering a move to multicore processors may not have the luxury of taking on a new parallel programming language or even re-architecting the application to support widespread concurrency. Instead, many development projects in the computing industry are adopting an evolutionary approach to enabling multicore processors. This approach employs existing programming tools and technology and a systematic method of introducing and debugging concurrency in existing software.

The Multicore Association (MCA) Multicore Programming Practices (MPP) guide is a detailed set of best practices for employing this evolutionary approach to multicore development. The following sections detail the goal of the MPP guide, the target audience, how to apply the MPP guide, and areas not covered.

1.2 Goal of MPP

The Multicore Programming Practices guide details a concise set of best practices in software development for multicore processors using existing technology and existing software. In writing this document, the MPP working group made decisions and compromises in *what and how much* this document targets.

The following are thoughts on some of the higher level decisions made:

- API choices – Pthreads and MCAPI were selected for several reasons. These two APIs cover the two main categories of multicore programming – shared memory parallel programming and message passing. Second, Pthreads is a very commonly used API for shared memory parallel programming. MCAPI is a recently introduced API, but is novel in being a message passing-based multicore API that specifically targets embedded homogeneous and heterogenuous applications.
- Architecture categories – The architecture categories (homogeneous multicore with shared memory, heterogeneous multicore with mix of shared and non-shared memory, and homogeneous multicore with non-shared memory) are in priority order and reflect the group's assessment of use in embedded applications. Admittedly, the second category is a superset of the first and third. The motivation for listing these three categories separately is due to the widespread and common use of techniques associated with the first and third, namely shared memory parallel techniques and process-level parallelism. The following paragraphs dissect this goal and offer more detail.

1.3 Detailed Description

A concise set of best practices is a collection of best known methods for accomplishing the task of development for multicore processors. The intent is to share development techniques that are known to work effectively for multicore processors thus resulting in reduced development costs through a shorter time-to-market and a more efficient development cycle for those employing these techniques.

The phases of software development discussed in MPP and a summary of each follows:

- Program analysis and high level design is a study of an application to determine where to add concurrency and a strategy for modifying the application to support concurrency.
- Implementation and low level design is the selection of design patterns, algorithms, and data structures and subsequent software coding of the concurrency.
- Debug comprises implementation of the concurrency in a manner that minimizes latent concurrency issues, enabling of an application to be easily scrutinized for concurrency issues, and techniques for finding concurrency issues.

- Performance concerns improving turnaround time or throughput of the application by finding and addressing the effects of bottlenecks involving communication, synchronization, locks, load balancing, and data locality.
- Existing technology includes the programming models and multicore architectures detailed in the guide and are limited to a few that are in wide use today.
- Existing software, also known as legacy software, is the currently used application as represented by its software code. Customers using existing software have chosen to evolve the implementation for new product development instead of re-implementing the entire application to enable multicore processors.

1.4 Business Impact

Readers of the MPP guide, who apply the detailed steps, may save significant costs in development projects. Cost savings can be applied to developer efficiency in analyzing and designing the implementation, a reduced number of bugs in the implementation, and increased likelihood of the implementation meeting performance requirements. These savings lead to faster time-to-market and lower development cost.

1.5 Target Audience

The MPP guide is written specifically for engineers and engineering managers of companies considering or implementing a development project involving multicore processors and favoring the use of existing multicore technology.

Specifically, the benefits of this guide to the specific target audience are summarized below:

- Software developers who are experienced in sequential programming will benefit from reading this guide as it will explain new concepts which are essential when developing software targeted at multicore processors.
- Engineering managers running technical teams of hardware and software engineers will benefit from reading this guide by becoming knowledgeable about development for multicore processors and the learning curve faced by their software developers.
- Project managers scheduling and delivering complex projects will benefit from reading this guide as it will enable them to appreciate, and appropriately schedule, projects targeting multicore processors.
- Test engineers developing tests to validate functionality and verify performance will benefit from reading this guide as it will enable them to write more appropriate and more efficient tests for multicore-related projects.

1.6 Applying MPP

The MPP guide is organized into chapters based upon software development phases. Following a brief introduction, each chapter includes either a process description or topic-by-topic coverage containing the technical details or both. The MPP guide provides more than a cursory review of a topic and strives to offer a succinct, but detailed enough description targeting the average developer.

1.7 Areas Outside the Scope of MPP

The MPP guide does not address every possible mechanism for parallel programming - there are simply too many different technologies and methods. The guide constrains coverage to existing and commonly used technology. Many of the techniques detailed in the guide can be generally applied to other programming models, however, this guide does not comment on the specifics necessary in a re-mapping. Chapter 7 contains comments on many of the technologies and methods, but the discussion is intended to be informational and not specific to the best practices documented in the previous chapters.

In particular, the areas outside the scope of the best practices portion of the MPP guide are:

- Languages other than C and C++ and extensions to C and C++ that are not part of the C and C++ ISO standard or are proprietary extensions to the standards;
- Coding style guidelines; and
- Architectures and programming models other than those specified in Chapter 2. This includes C & C++ compatible parallel libraries other than those specified in Chapter 2.

CHAPTER 2: OVERVIEW OF AVAILABLE TECHNOLOGY

2.1 Introduction

Multicore programming tools and models include items such as software development tools, programming languages, multicore programming APIs, and hardware architectures. While the programming practices covered in this guide are exemplified on these specific tools and models, our intent is for these practices to be generally applicable. This chapter specifies the programming languages, multicore programming APIs, and hardware architectures assumed in the programming practices documented in the later chapters.

2.2 Programming Languages

The MPP guide employs standard C and C++ as the implementation languages because these are the predominant languages used in embedded software development. The C language as specified in "ISO/IEC 9899:1999, Programming Language C" is used. The C++ language as specified in "ISO/IEC 14882, Standard for the C++ Language" is used. For both languages, we did not use any nonstandard extensions, either commercial or research oriented.

2.3 Implementing Parallelism: Programming Models/APIs

There are several types of parallelism used in multicore programming, namely Task Level Parallelism (TLP), Data Level Parallelism (DLP), and Instruction Level Parallelism (ILP). The MPP guide focuses on TLP and DLP, however all three types of parallelism play a role in design and implementation decisions.

The MPP guide uses two multicore programming APIs, Pthreads[i] and Multicore Communications API (MCAPI[ii]), for its examples. This will allow us to provide coverage of two categories of multicore software development - shared memory programming and message-passing programming. We chose these two APIs because they are commonly used APIs within these two software development paradigms.

2.4 Multicore Architectures

The MPP guide targets three classes of multicore architectures: homogeneous multicore with shared memory, heterogeneous multicore with mix of shared and non-shared memory, and homogeneous multicore with non-shared memory[iii]. These three classes are the predominant architectures employed in embedded projects and they are different enough so that the set of practices that apply to each are substantially disjointed. The underlying communication fabric is outside the scope of this document and in most cases is irrelevant to the software developer.

A homogeneous multicore processor with shared memory is a processor comprised of multiple processor cores (2 to n cores) implementing the same ISA and having access to the same main memory.

A heterogeneous multicore processor with a mix of shared and non-shared memory is a processor comprised of multiple processor cores implementing different ISAs and having access to both main memory shared between processor cores and local to them. In this type of architecture, local memory refers to physical storage that is only accessible from a subset of a system. An example is the memory associated with each SPE in the Cell BE processor from Sony/Toshiba/IBM. It is generally necessary to state what portion of the system the memory is local to, as memory may be local to a processor core, a processor, an SoC, an accelerator, or a board.

A homogeneous multicore processor with non-shared memory is a processor comprised of multiple processor cores implementing the same ISA and having access to local, non-shared main memory.

2.5 Programming Tools

The software tools employed in this guide and details of use are summarized as follows:

- GNU gcc version 4.X – compiler used for all C examples
- GNU g++ version 4.X – compiler used for all C++ examples

CHAPTER 3: ANALYSIS AND HIGH LEVEL DESIGN

3.1 Introduction

This chapter discusses program analysis techniques used to uncover opportunities to exploit parallel behavior and the dependencies which constrain that behavior. Analysis results are strongly affected by the use of representative benchmarks and realistic workloads. High level design decisions, including choice of algorithm, selection of platform architecture, and identification of appropriate parallel design patterns, are then made which will optimize the multicore implementation to meet critical application metrics such as performance, power, footprint, and scalability.

3.2 Analysis

To effectively exploit multicore resources, you must parallelize your program. However, several steps should first be taken before introducing parallelism. This section describes two such activities—improving the program's serial performance[iv] and understanding the program.

3.3 Improving Serial Performance

The first step in redesigning your program to exploit multicore resources is to optimize its serial implementation because serial performance tuning is typically easier, less time consuming, and less likely to introduce bugs. Serial improvement will reduce the gap between where you are and your performance goal, which may mean that less parallelization is needed. It also allows parallelization to focus on parallel behavior, rather than a mix of serial and parallel issues.

It is important to remember, though, that serial optimization is *not* the end goal. We want to optimize carefully, applying only changes that will facilitate parallelization and the performance improvements that parallelization can bring. In particular, serial optimizations that interfere with, or limit parallelization should be avoided. For example, avoid introducing unnecessary data dependencies or exploiting details of the single core hardware architecture (such as cache capacity).

Developers are familiar with many techniques that can be used to improve a program’s serial performance; in fact, there is a wealth of public material (e.g., articles, books, tools, and accumulated wisdom). But haphazard application of such improvements is generally a bad idea.

"Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.”[v]

Instead, discipline and an iterative approach are the keys to effective serial performance tuning (Figure 1). Use measurements and careful analysis to guide decision making, change one thing at a time, and meticulously re-measure to confirm that changes have been beneficial.

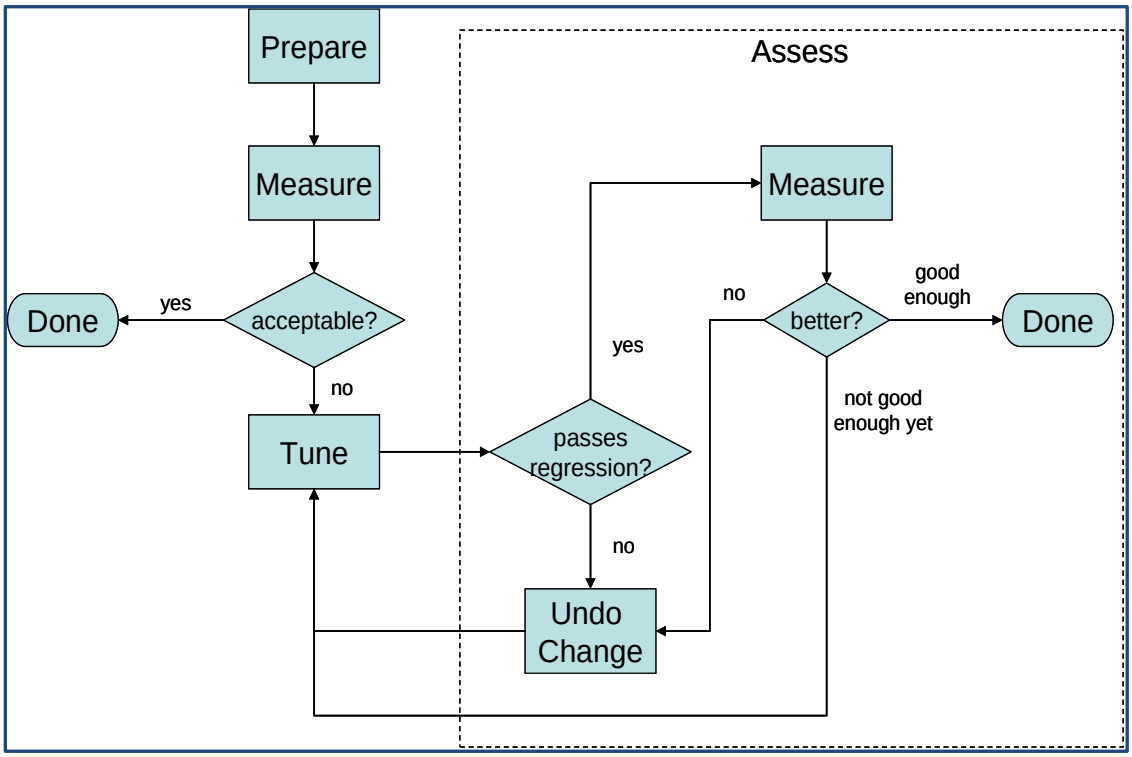


Figure 1. Serial performance tuning approach. Four key steps: prepare, measure, tune, assess.

The following sections provide short descriptions of the steps in this approach, but each topic is richer than can be reasonably covered in this document. There has been a great deal of useful work in performance analysis and performance tuning and most of it is applicable when preparing for multicore (though the effects of parallelization changes are more sensitive to the order in which the changes are applied).

3.3.1 Prepare

Assemble a collection of regression tests and benchmarks against which to improve the program's performance. Before making any changes, think about your objectives and assemble the resources that will help you achieve them efficiently and reliably.

- What are your goals? Improving a program’s performance can be a complicated, time consuming, and error prone task. Stop when you've reached "good enough" results.
- What operational scenarios are most important? Any improvements made will be based on data from observation of the program's execution, but which scenarios are most important? You could focus on steady state operations, heavy load conditions, error recovery scenarios, or some balance of such scenarios.
- What performance metric makes sense for your program? Perhaps it's the time to process a unit of work, or maybe the number of units of work completed in a unit of time. Know what to measure so you can quantify goals or know that you've met them.

Construct a set of benchmarks (uses of the program over specific data and operations) over which you will collect performance data. All optimizations will be relative to the benchmarks, so carefully choose the scenarios for which performance most needs to be improved. Also include cases that vary the workload so that you can see how performance scales (at least three data points are needed to differentiate linear from non-linear scaling).[vi]

Performance optimizations are often trade-offs with some other concern, such as memory consumption, code maintainability, or portability. Know how much you are willing to compromise in any of these areas.

Serial performance tuning is an iterative process that involves repeated execution of your program, both to measure performance and ensure that correct functionality is preserved, so make sure you have a good regression suite before you start (it should go without saying that the program passes all tests before starting).

3.3.2 Measure

After preparations are complete, the next task is to utilize your benchmarks and understand the baseline performance of your program. Development tools, such as profilers, are typically used to measure performance. A profiler is a tool that observes the execution of a program and collects measures of its performance. Several different approaches are used for profiling:

- Sampling: A sampling profiler takes measurements at regular intervals.[vii]
- Instrumentation: An instrumention-based profiler typically compiles or links additional instructions into the program for measurement.
- Emulation or simulation: These approaches execute the program in an environment that emulates or simulates (e.g., through interpretation) the execution environment.

The quality of the performance data varies in several ways, two of which are completeness and resemblance of "real performance." Sampling only measures at regular intervals, generating incomplete insight into program behavior. Non-sampling approaches produce complete data, with respect to the measurement granularity. Any profiling approach, however, has some effect on how closely the measurements resemble execution in the absence of measurement. Low overhead approaches more closely resemble typical execution than those with high overhead, such as instrumentation or interpretation, which may alter the timing properties of the program in significant ways.

Output differs from tool to tool in both granularity and format. Profilers typically measure data relative to functions, with simple measures being the time at which each function call and return occurs. Some profilers capture information at a finer granularity or allow the user add instrumentation points (e.g., encompassing interesting loops).

Profiler output varies considerably, from simple reports to sophisticated GUIs with controls for filtering and sorting. Two common forms of output are:

- A flat profile is generally a tabular report of statistics organized by function, such as how much execution time was spent in a function (by percent and units of time) and the number of times the function was called.
- A call graph shows execution time, again by function, relative to call chains. A call graph will show which functions were called by a given function, how many times, and how much execution time was used.

Typically, you want to use both forms of output. A flat profile is a good way to quickly find where large portions of time are being spent (e.g., a single function that consumes 60% of the time). A call graph is a good way to see where a function is being used.

Profiler(s) choice is based on a number of the usual criteria, such as availability/cost, target platform, experience, and so on. Use at least one profiler that measures how much of your execution time is spent outside of your program (e.g., in system calls) to assemble a more complete performance picture. When developing an embedded system, you may need to use an emulation- or simulation-based profiler before target hardware is available or to compensate for lack of a persistent store for measurements in the target device.

3.3.3 Tune

An important rule of thumb in performance tuning is "only change one thing at a time." Each tuning iteration, use the measurements you gathered through profiling to look for the biggest improvement you can reasonably make.[viii] Remember that you will also be making substantial changes when parallelizing your program, so it's usually best to stick with the "low-hanging fruit" or big wins when improving the serial program's performance. Changing an algorithm that consumes a large portion of time to scale linearly rather than logarithmically, for instance, would be a good example of a change worth making (for clarification, this sentence only refers to the scaling, which depends on the number of cores, and not to the algorithm's complexity, which depends on the size of the input data). In general, favor changes that reduce computation over those that minimize data footprint. At this stage, we want to focus on reducing the program to the essential computations needed to function correctly.

Start by looking through your measurements for hotspots. Hotspots are areas in your program that use disproportionate percentages of your execution time, and can be found using flat profiles. Start with hotspots for a simple reason: a performance improvement in frequently executed code will yield better overall improvement than a similar improvement in infrequently executed code. That is, a 10% improvement in a function using 50% of total execution time is an overall improvement of 5%; a 10% improvement in a function using 5% of total execution time is only an overall improvement of 0.5%. Beware of diminishing returns.

Once you select a hotspot for examination, think about why it is consuming so much time and whether that is reasonable.

- Computing: executing the instructions is the first thing we tend to think about.
- Is an appropriate algorithm used; review the profiling data for different workloads to see if it's scaling as you would expect.
- Are all computations needed. You may be able to hoist calculations out of a loop or use lookup tables to reduce computation.
- Are you using the right types and data structures? Moving larger chunks of memory around typically takes longer (e.g., using floats is often faster than using doubles). Consider the impact of different language constructs on your platform (e.g., the overhead that is necessary to support exception handling).
- Is your program decomposition good? Use inlining or refactoring if you have functions whose execution time isn't significantly larger than call/return overhead.
- Think about your persistence strategies. Re-initializing and re-using a data structure (or class) can be more efficient than creating a new one.
- Waiting: some operations just have to spend time waiting, which is particularly problematic in serial programs. I/O operations like network or file system access take a long time, as does user interaction. Examine your caching strategy for some improvements, but using concurrency is the typical solution. Avoid introducing concurrency at this point, but note the dependency. This will factor into your parallelization strategy.
- Working with large data sets can cause programs to frequently move data in and out of memory. Reducing unnecessary memory moves can be a big help. Memory profilers are good tools for understanding and improving your memory use.
- Though it could improve serial performance, avoid reorganizing your data for locality (i.e., keeping data used together close in memory, which could allow a single move instead of several smaller moves). This kind of optimization is better performed as part of parallelization, when factors like cache width and policy are considered.

As you examine each hotspot, use several sources of information. Beyond the code itself, you can also get a lot of quick information from the call graph generated by a profiler. By looking at calls made by or to a function, you gain more insight into how it's used and what it does. In particular, walking back up the call graph (callers of the function) may lead you to a hotspot that is not a leaf node of the call graph.

3.3.4 Assess

Each time you make a change to your program, stop and assess the effect. A "good change" is one that does not introduce bugs and that either directly or indirectly enables additional changes which improve the performance metrics you identified in the *Prepare* step. Assess these criteria by re-running your regression suite and re-measuring the program using your benchmarks.

If the regression fails or your performance is worse than before the change, then you misunderstood something about the program or your change. Study the program again, before and after the change, to understand the cause of the failure. If it cannot be fixed, back out the change and return to the Tune step. Re-examine the hotspot and try to apply different changes before moving on to the next hot spot.

If the regression passed and your performance improved, make sure the change is also acceptable with regards to your other concerns. For example, it might be important to preserve the portability properties that your program started with. Such criteria are often more difficult to validate in an automated way, but should be part of your mental checklist.

3.4 Understand the Application

Regardless of the technique used to transform the application from one that processes sequentially to one that processes concurrently, having a thorough understanding of the problem being solved cannot be avoided. This understanding can come from a variety of sources, including documentation, discussions with the application's developers, and profiling data.

As discussed in the following sections, some aspects of an application's behavior are particularly relevant to parallelization, such as a thorough understanding of computational and data dependencies. A good application analysis will help to determine which dependencies are inherent in solving the problem and which are artificial, perhaps introduced as part of a serial optimization.

3.4.1 Setting Speed-up Expectations

Generally there are two reasons for splitting up and re-architecting an application for parallelization: solving large problems and speed-up. Solving large problems is achieved not only by distributing the work across cores, but (sometimes) also by distributing the work across processors due to the sheer size of the data which makes the problem too large for a single processor[ix]. Speed-up will be the primary driver for the discussion here (based on the assumption that the legacy application being ported to the multicore platform is already properly sized for data and memory requirements).

Speed-up is achieved by effectively load balancing work across the cores to minimize execution time, but there are limits to what is reasonable. For example, some code segments cannot be effectively parallelized. Amdahl's Law (see Section 6.2) expresses the maximum expected benefit for improving a portion of an application. If the serial execution time of a portion accounts for 80% of the total time, and that portion will be spread across eight cores, the result would be at best 3.3 times faster. If that same portion only accounted for 20% of serial execution time, the same parallelization would be at best 1.2 times faster. These are maximum expected improvements - various overhead factors (e.g., for coordination) will further reduce the benefit.

The effort to parallelize the application should be justified by an objective analysis of the application, the effort required to make the change, and the realistic expectations from the effort.

3.4.2 Task or Data Parallel Decomposition

From the application's profile it should be possible to find the more computationally-intensive sections of the application (e.g., hotspots identified from the flat profile generated from measures of the serial application). The computational nature of these sections will likely fall into one of two general forms of parallel decomposition: task or data parallel. (Note: Task parallel is sometimes referred to as function parallel).

Task parallel decomposition is where the solution to a problem could be envisioned as a sequence of steps which could be accomplished concurrently, which are absolutely or relatively independent of each other, to achieve the same outcome regardless of the order in which those tasks are carried out. Absolutely independent of each other means that the tasks share no data and that no synchronization between the steps must occur (very uncommon). Relatively independent of each other, the more common situation, means that such tasks share data and at certain times the tasks must synchronize their activities. One example of task parallel decomposition is an image processing and capture application in which a bitmap from a camera is received and the application performs three operations:

- Count the number of unique colors in the image
- Convert the bitmap to a JPEG
- Save the image to disk.

In the serialized application, these activities are sequential (Figure 2). In the task parallelized version of this application, two tasks can be done in parallel since there are no dependencies between them, but the third task to save the image can only be done after the bitmap is converted to JPEG—thus requiring synchronization between the task that performs the encoding and the task that saves the image. If the application were only required to save the original bitmap captured from the camera and not the converted JPEG, there would be absolutely no dependencies between the three tasks, and all three could be performed in parallel.

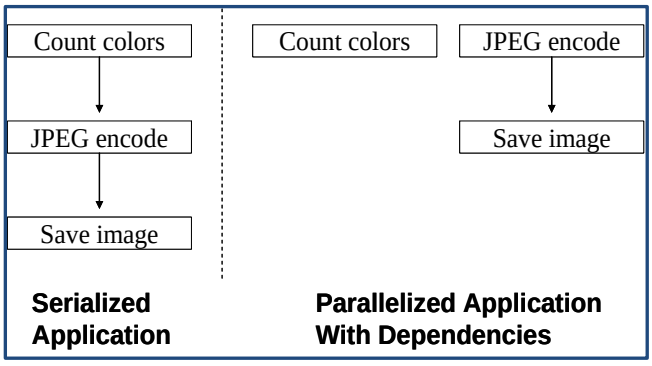


Figure 2. Task parallel decomposition example.

Data parallel decomposition is where the solution to a problem involves segmenting and decomposing major data structures (e.g., arrays) into smaller, independent blocks of data which can be processed

independently and concurrently from one another. An example of this form is a scalar multiplication operation on a matrix (Figure 3). In a serialized application, such operations are often performed sequentially through the aid of looping constructs. In this example, each cell can be operated on completely independent of other operations on other cells in the matrix. Furthermore, such operations can occur in any order and the resulting matrix from the operation will be the same in any case. The ordering imposed by the serial loop construct is simply artificial given the semantics of the loop construct; a parallelized form of this same matrix operation is not bound by such constraints. As such, each cell of the matrix could be the decomposed form of a parallelization effort. If the matrix is 2x2, then there could be four functions assigned to perform the scalar multiplication to each corresponding cell.

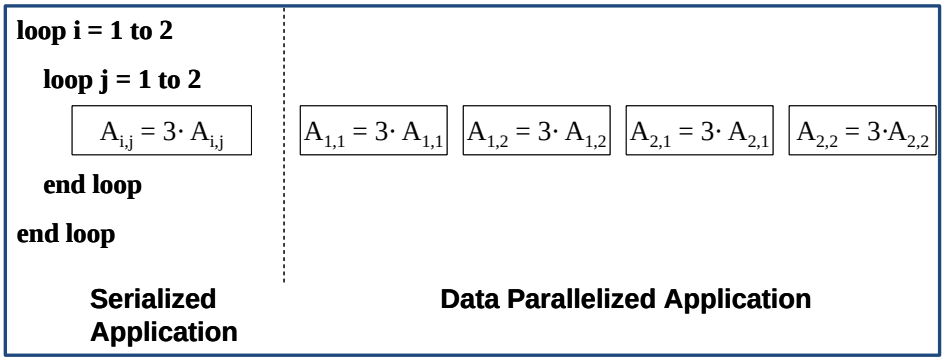


Figure 3. Data parallel decomposition example.

In task or data parallelization, it is often that one type will impact the other; in other words, task and data parallelism are not mutually exclusive. Most application problems are amenable to both forms of decomposition—knowing which form is the most appropriate can only be reasoned by thoroughly understanding the problem. Some problems, such as the image processing and capture application problem, are more amenable to task decomposition, whereas others (e.g. the scalar multiplication problem) are more amenable to data decomposition. In either type of parallelism, data structures must be designed so that the tasks from task parallel decomposition can perform their computations, or tasks must be created to manipulate the independent blocks of data resulting from the data parallel decomposition.

3.4.3 Dependencies, Ordering, and Granularity

Dependencies can be in the form of either ordering between operations within the serialized application or data dependencies (e.g., shared memory) between operations. Some profiling tools can generate call graphs which can aid in discovering the ordering and the frequencies of operations and tasks within an application. The serialized form of the application may yield some insight into what that ordering will be, but some compromises may have been made to optimize the serialized application for a single core processor and that the ordering could be artificial to achieve other (prior) objectives. Having insight of the potential for parallelization of the application, separate from the

serialized form, may help identify false ordering (ordering which is not required) between operations and present opportunities for parallelization. In the image processing and capture application example above, counting the number of unique colors in the bitmap did not necessarily have to occur first.

Shared data dependencies between different (perhaps large) portions of code may also be discernible from call graphs, but it is more likely that inspection of the source code will be necessary to either confirm or refute such data dependencies. Static analysis tools and memory profiling tools may be available to aid in the identification of data dependencies for large regions of code (for example looking for access to global variables, shared memory regions, etc. across many lines of code or source code files). Such tools will likely be language dependent and possibly even platform dependent.

Hotspots that are the result of a computationally-intensive loop may create dependencies within an operation. These hotspots are candidates for parallelization because if the loop can be designed to be executed in parallel some speed-up can be achieved. However, in order to perform this activity, the loop must be analyzed and redesigned so that each loop iteration can be made independent—that is, each iteration can be safely executed in any order and any dependencies are not carried from each iteration. On the other hand, some loops used in serialized applications must have dependencies between iterations, such as performing accumulations or other associative operations which are more difficult to remove.

To identify speed-up and achieve efficient use of all the cores, the decomposition, ordering, design, and implementation of the tasks needed to perform the overall workload (task parallel or data parallel decomposition) must consider the target platform. This is to ensure that the work created for each of the tasks will be of sufficient size to keep the cores busy and be correctly balanced between the multiple cores. Previously, we illustrated in the prior scalar multiplication operation that a task could be created for each element of the matrix to perform the design operation. For this specific example, this would likely never be done here for various reasons:

- The allocation of a task for each cell of the matrix to one of the cores would simply not scale for a matrix of any practical size[x]
- The overhead introduced by parallelization would likely not outperform the serialized code

This underscores the point that determining the proper sizing (or granularity) of the tasks is non-trivial. For software engineers performing this activity for the first time, this might involve some trial and error after the parallelization of the application is shown to be correct. There are also heuristics which can aid in estimating granularity which may be available for the target platform, parallelization libraries, or languages used for the parallelization effort. For example, Thread Building Blocks[xi] recommends a “grainsize” (an indicator of granularity for the number of instructions to execute in the body of a task) of 10,000 to 100,000 instructions.

After conducting a parallelization exercise on the source code, revisit the assessment loop (Figure 1) to confirm that the expected results for speed-up occurred and that the regression test for the application passed.

3.5 High-Level Design

After comprehending the algorithmic intent and serial implementation of a program, it is appropriate to begin the high-level design for a parallel implementation of the application. The intent of the high-level design phase is to partition the application into separate regions which expose work to run in parallel. Dependencies between regions will limit this parallelism. Additionally, characteristics of the target run time will practically limit the amount of work which can be done in parallel. Choose an appropriate granularity of decomposition which can be efficiently mapped onto the parallel run-time environment.

3.5.1 Task Parallel Decomposition

Task parallel decomposition enables different tasks to run in parallel. The different tasks may read shared data, but they produce independent results. In an example image recognition application (Figure 4), four separate identification tasks share the same input image data. Each task is specialized to identify different objects in the image. Once all four tasks have been completed, the aggregate data may be processed by downstream operations.

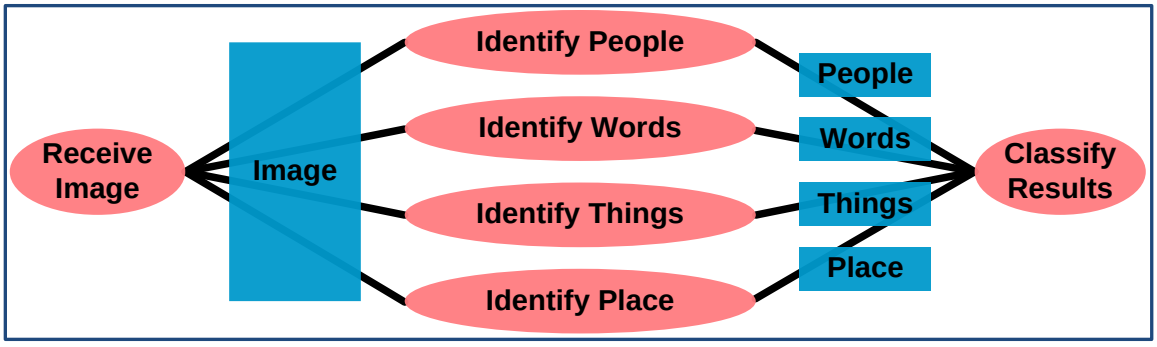


Figure 4.Task parallel image processing example.

For a given processing platform, the different ‘identify’ tasks may have different execution times. Though the four tasks may all be run in parallel, if identifying words, things, and places are each three times faster than identifying people, it may make sense to run the word, thing, and place identification tasks one after another in parallel with the slower people identification task. This type of load balancing is an example of how task parallel decomposition may need to be scheduled properly to most effectively utilize the available processing resources.

3.5.2 Data Parallel Decomposition

Data parallel decomposition focuses on partitioning input data into separate regions and the same task is run in parallel on the different data regions. In another example image recognition application (Figure 5), each pixel in the output image is calculated using some corresponding combination of input pixels. An output pixel computation may share input pixels with its neighbors, but each output pixel can be computed independently of all others.

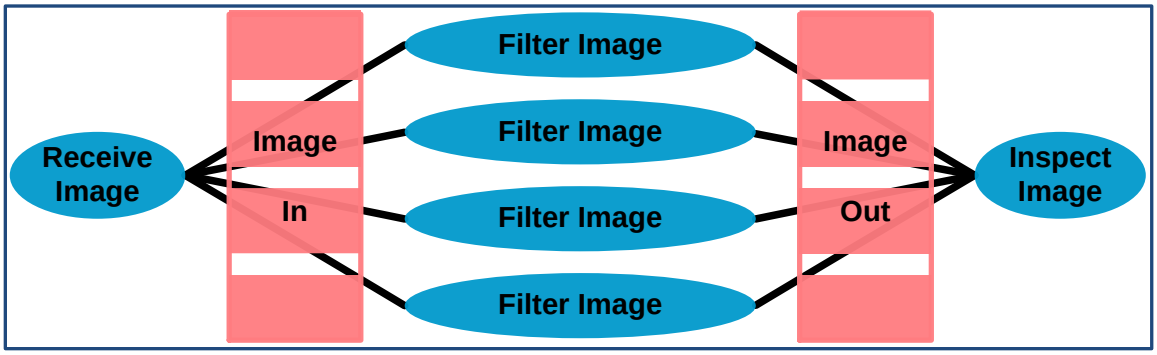


Figure 5. Data parallel image processing example.

Though each pixel may be computed in parallel, the granularity of decomposition must match the available resources. For example, if there are 8 logical processor cores available, the image could be sliced into 8 independent regions. Within a region, the pixels would be computed serially, but the regions themselves would all be computed in parallel. Data decomposition can inherently deliver a natural load balancing process. When the task being computed takes the same amount of time per pixel, which represents a static workload, then the data can be divided into equal segments to balance the load.

3.5.3 Pipelined Decomposition

Pipelined decomposition may be considered a type of hybrid task-data decomposition whereby each task may be decomposed into separate functional stages. Each stage processes a subset of the overall data and passes its results to the next stage. Though each stage processes its data serially, all stages run in parallel to increase the processing throughput.

In an example imaging edge-detection application, it consists of a number of stages: pixel correction, image smoothing, and Sobel[xii] edge detection (Figure 6). With pipelined decomposition, a pixel block enters the pipeline at the correction stage and emerges after Sobel. As a data block moves from the first stage to the second, the next data block may enter the first stage. All three stages can run in parallel, each working on a different block of data at a different stage in the algorithm.

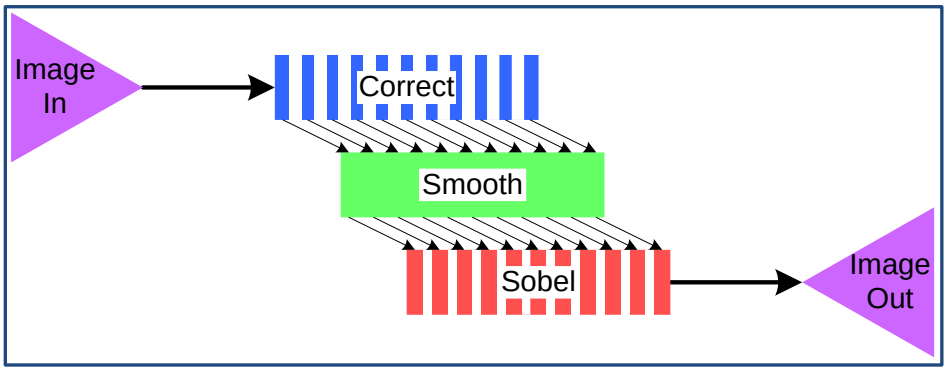


Figure 6. Pipelined edge detection

Ideally, pipeline decomposition requires the number of stages to be matched to the available processing resources. The overall throughput of the pipeline is also limited by the slowest pipeline stage. In the example above, the smoothing function requires the most work, and for each data block, the correction and Sobel functions must idle until the smoothing function completes. The smoothing function's speed determines the overall performance of the parallel implementation. Recursively parallelizing the smoothing function may shorten its execution time and better balance the pipeline.

3.5.4 SIMD Processing

Most modern processors include some type of vector operations, termed single instruction, multiple data (SIMD) operations. A SIMD instruction enables the same operation to be applied to multiple data items in parallel. An example instruction might multiply four pairs of eight-bit data values, storing the result as four sixteen-bit data values. This type of low-level parallelism is often used within loops to increase the work per iteration. Where possible, this instruction-level parallelism should be applied first followed by the task level data decomposition previously discussed.

3.5.5 Data Dependencies

When algorithms are implemented serially, there is a well-defined operation order which can be very inflexible. In the edge detection example, for a given data block, the Sobel cannot be computed until after the smoothing function completes. For other sets of operations, such as within the correction function, the order in which pixels are corrected may be irrelevant.

Dependencies between data reads and writes determine the partial order of computation. There are three types of data dependencies which limit the ordering: true data dependencies, anti-dependencies, and output dependencies (Figure 7).

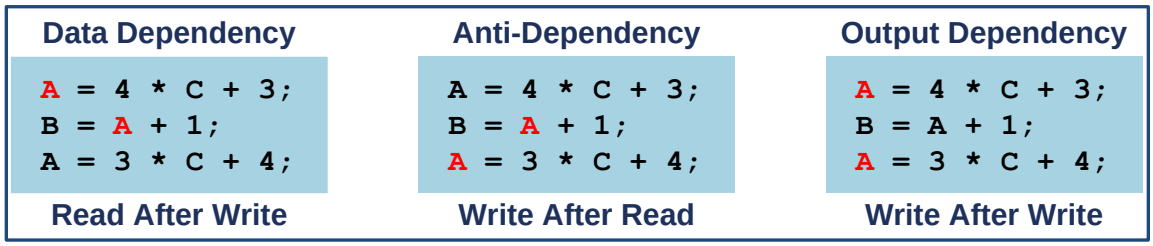


Figure 7. Three types of data dependency examples (the dependencies are highlighted in red): a) data dependency; b) anti-dependency; and c) output dependency.

True data dependencies imply an ordering between operations in which a data value may not be read until after its value has been written. These are fundamental dependencies in an algorithm, although it might be possible to refactor algorithms to minimize the impact of this data dependency.

Anti-dependencies have the opposite relationship and can possibly be resolved by variable renaming. In an anti-dependency, a data value cannot be written until the previous data value has been read. In Figure 7b, the final assignment to A cannot occur before B is assigned, because B needs the previous value of A. In the final assignment, variable A is renamed to D, then the B and D assignments may be reordered.

Renaming may increase storage requirements when new variables are introduced if the lifetimes of the variables overlap as code is parallelized. Anti-dependencies are common occurrences in sequential code. For example, intermediate variables defined outside the loop may be used within each loop iteration. This is fine when operations occur sequentially. The same variable storage may be repeatedly reused. However, when using shared memory, if all iterations were run in parallel, they would be competing for the same shared intermediate variable space. One solution would be to have each iteration use its own local intermediate variables. Minimizing variable lifetimes through proper scoping helps to avoid these dependency types.

The third type of dependency is an output dependency. In an output dependency, writes to a variable may not be reordered if they change the final value of the variable that remains when the instructions are complete. In Figure 7c, the final assignment to A may not be moved above the first assignment, because the remaining value will not be correct.

Parallelizing an algorithm requires both honoring dependencies and appropriately matching the parallelism to the available resources. Algorithms with a high amount of data dependencies will not parallelize effectively. When all anti-dependencies are removed and still partitioning does not yield acceptable performance, consider changing algorithms to find an equivalent result using an algorithm which is more amenable to parallelism. This may not be possible when implementing a standard with strictly prescribed algorithms. In other cases, there may be effective ways to achieve similar results.

3.6 Communication and Synchronization

Data dependencies establish a set of ordering requirements which must be enforced to ensure proper operation. Diverse parallel programming models use different techniques to ensure proper ordering.

3.6.1 Shared Memory

In a shared memory system, the individual processing units may locally cache memory, but the system must maintain a consistent view of shared memory between the processors (Figure 8). With caches, the silicon cost of maintaining a coherent view of memory can become very high if cache lines must be shared between processors. Proper data layout in memory can minimize this penalty.

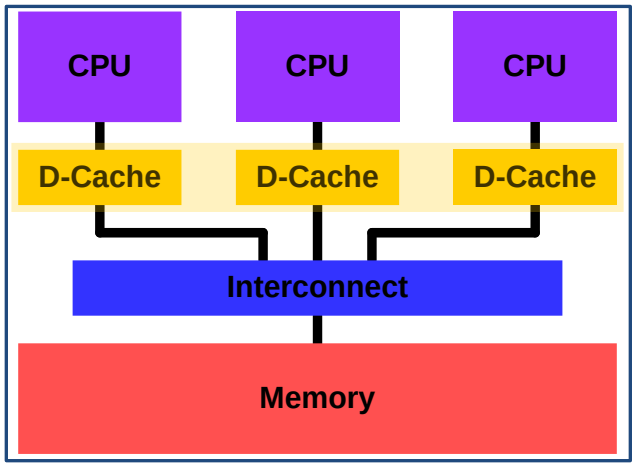


Figure 8. Processing units share the same data memory.

Multiple threads of execution are used to run multiple tasks simultaneously. Data which is shared between tasks must also be properly synchronized by the developer in the application (i.e. locks) to ensure dependency relationships are properly maintained. In an example, threads T0 and T1 may both need to read and write variable ‘A’ (Figure 9). Without synchronization, the order of reads and writes between the two threads is unpredictable, and three different outcomes are possible.

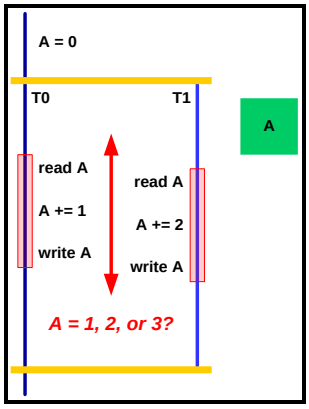


Figure 9. Shared data synchronization.

Mutual exclusion through locks and semaphores is a common technique to ensure that only one thread at a time may execute code which encompasses certain critical dependencies. While one thread holds a lock for a critical section, other threads are blocked from entering, which if allowed, could violate dependencies involving data shared between the sections. In the example of Figure 9, properly locking the critical sections ensures that 'A' always receives 3 as a final value.

Generally, each processor runs one thread at a time. More threads may be created across the processor cores. When one thread blocks while waiting for the lock to clear, the operating system may wake up another ready thread to take its place.

The cost of acquiring and releasing a lock can be significant. Locks serialize code, so locking large critical sections will inhibit parallelism. On the other hand, using frequent, low-level locking may impose a large penalty for synchronization. The cost of creating and destroying completed tasks is also significant, so once again, the granularity of tasks and locking should match the available resources.

3.6.2 Distributed Memory

With distributed memory systems, memory is not shared between systems and each processor manages its own local memory (Figure 10). Communication of data between tasks running on different processors is accomplished by sending and receiving data between them, often termed message-passing. While the thread programming model is appropriate for shared memory, the message passing model may be used with either distributed or shared memory systems.

In the distributed memory model, data must be *explicitly* shared between tasks. Synchronization between tasks can be achieved by synchronous send-receive semantics. A receiving task will block until the sending data is available. Alternatively, asynchronous send-receive semantics can be used where the receiving task can check or be notified when data is available without blocking. This permits overlap of computation and communication leading to significant performance gains.

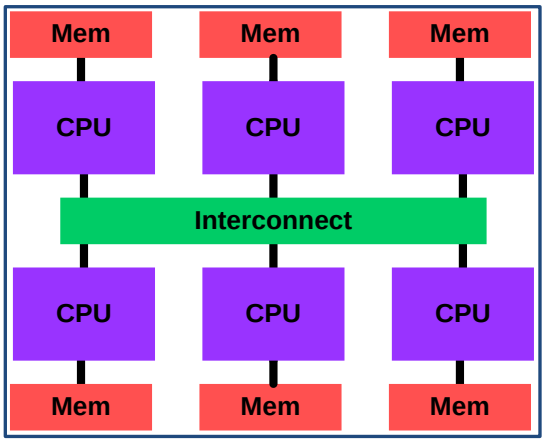


Figure 10. Distributed memory architecture.

Relative to shared memory, distributed memory represents a higher overhead for communication, both in the setup and tear down of a message and in explicit copying of data, so message passing should be optimized for both functions.

3.7 Load Balancing

The amount of work per computation may be fixed or it will depend on the input data. Work which is fixed, is termed a static workload, and may be partitioned during design time. Consider the edge detection example discussed previously. In data decomposition, the workload is the same for all pixels, and the parallelism scales easily across available processors.

What would happen if the run time of the smoothing function was dependent on the input data? For example, regions with slowly varying pixel values might take fewer compute cycles to smooth, while regions with rapidly changing pixel values might require extra processing. This type of workload varies dynamically depending on the pixel values - equally-sized pixel regions may contain very different workloads. Consider an image with widely varying pixels on the top half and a uniform color in the bottom half. If the work is split between two tasks, top and bottom and each running on separate processors, the bottom half will finish well in advance of the top half, and the bottom processor will idle.

With unpredictable workloads, the schedule can only be statistically optimized. In this case, dividing the work into smaller regions might improve the processing efficiency because the differences between neighboring workloads will be less. As the work is divided into smaller tasks, the cost of communication can become a significant portion of the task run time and must be considered in determining the best task sizes. Run times using thread pools or work stealing provide built-in support for dynamic load-balancing.

Task, data, and pipeline decompositions should always consider how to balance the workloads between regions. Static workloads are easy to analyze, while dynamic workloads will require

statistical considerations to find a good level of granularity which both matches the available resources and alleviates the workload mismatches.

3.8 Decomposition Approaches

The high-level design process decomposes the algorithm into separate processing regions, taking the data dependencies, communication, and synchronization requirements into consideration, and then performs load balancing to establish an efficient partitioning for implementation.

3.8.1 Top-Down or Bottoms-Up

There are two main approaches to decompose an algorithm - top-down and bottom-up. In a top-down approach, the algorithm is decomposed into separate regions. Some regions will remain serial, while other regions may be decomposed using task, data, and pipelined decompositions. Use profiling information from high-level analysis to identify the most beneficial regions to be decomposed. After each decomposition effort, the parallel performance can be estimated. If not enough performance is achieved, then regions can be further subdivided trying to find additional parallelism.

Some regions will easily decompose into many fine-grained regions, and there will be a tradeoff of granularity and overhead. Data decomposing an image into pixels might be an extreme example of fine-grained implementation. The number of available cores, overhead of creating and destroying tasks, and communication and synchronization overhead become a significant cost.

As we described previously, profiling information from high-level analysis identifies the most important hot spot regions. To gain the most benefit, concentrate on parallelizing the hot spots that consume relatively large amounts of execution time. A true bottom-up approach will start with the finest grained partitioning for hot spot regions and progressively move up to encompass larger regions, maintaining a balance between partitioning and overhead. Once a suitable granularity is established, a larger region can be assembled in parallel from the smaller regions until achieving a satisfactory estimated performance level.

3.8.2 Hybrid Decomposition

Decompositions may be applied hierarchically to best match the available and type of processing resources. For example, consider a two-stage pipeline decomposition in which the first stage is 6 times slower than the second (Figure 11). Therefore, the initial pipeline decomposition will focus on the slower first stage.

Using data decomposition and assuming sufficient processing resources, it may be possible to move six data blocks in parallel through the first stage, which can then run serially through the second stage in the same amount of time, effectively balancing the pipeline and optimizing the throughput.

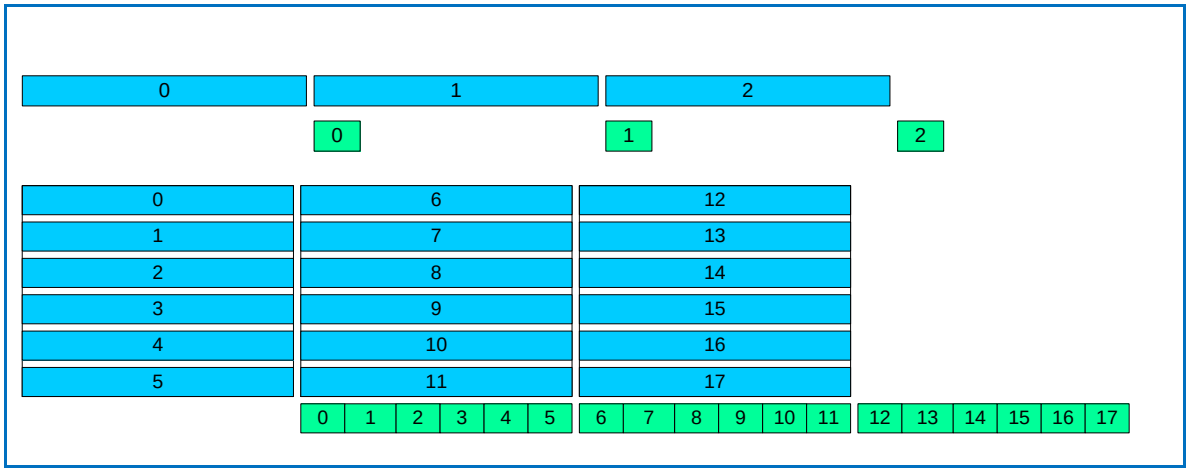


Figure 11. Example depicts 2-stage pipeline combining data and pipeline decomposition.

3.9 Parallel Design in Modeling

In some application areas, it may be necessary to undertake the analysis and high-level design of parallelism in the modeling phase because algorithms are designed on special modeling tools that automatically generate C code. For example, in recent control systems, mathematical modeling tools such as MATLAB/Simulink® are utilized. Owing to the use of these tools, Model-Based Development (MBD) has become prevalent, wherein algorithms are designed and simulated using the modeling tools, and implementation is performed with the automatically generated code by using these tools. In this design flow, given that high-level design and analysis with automatically generated code are sometimes futile, it is indispensable to design parallelism in modeling.

Model-Based Parallelization (MBP) can be used for this purpose. In many modeling tools, data flow block diagrams are utilized to represent models. Figure 12 shows an example of Simulink model for the Fibonacci function ($f_{n+1} = f_n + f_{n-1}$) and comparison of f_n/f_{n-1} with its convergence value, with is also called the golden ratio. In this diagram, the blocks marked by $1/z$ are unit delays that keep values of f_n and f_{n-1}. At each step, the calculation initializes with the reading of the values of f_n and f_{n-1} at the unit delays marked (S). Then, the values of f_{n+1} and f_n at the unit delays marked (T) and the calculated values at output ports marked (T) are stored and outputted. The diagram explicitly shows data dependency; task parallel decomposition creates a model partitioning problem, which considers load balancing, inter-core communication, and computation orders with start and end points (Figure 13). In addition, there is a way to draw data parallelism using special blocks such as “For Iterator” blocks in Simulink. Therefore, MBP might be useful for the analysis and high-level design of parallelism in modeling.

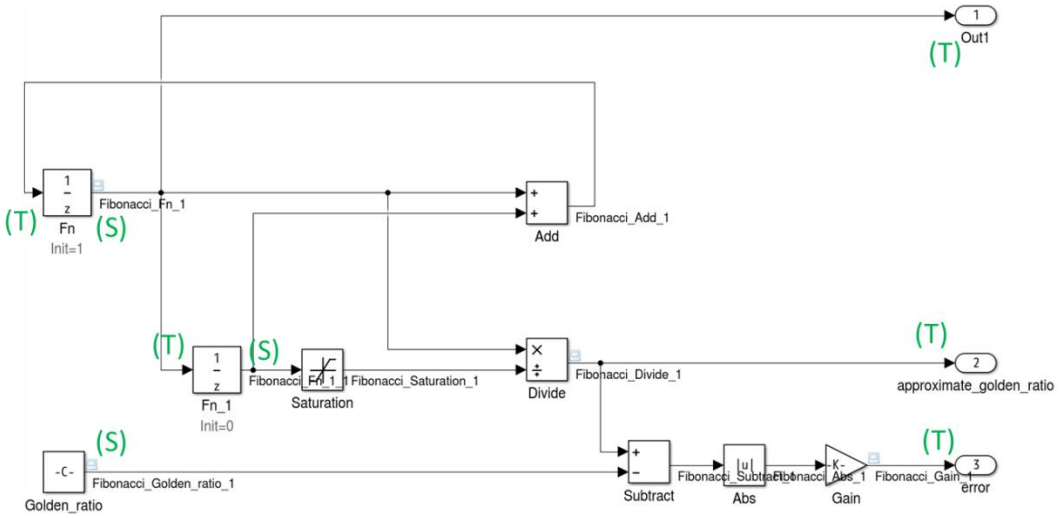


Figure 12. Model for Fibonacci function

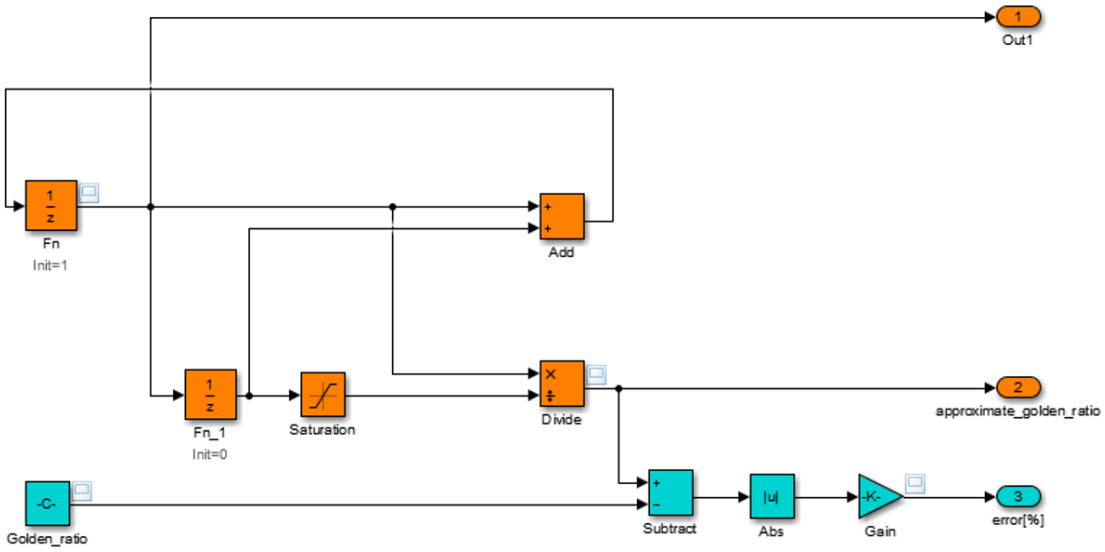


Figure 13. Model partitioning.

CHAPTER 4: IMPLEMENTATION AND LOW-LEVEL DESIGN

4.1 Introduction

Parallel implementations are highly platform architecture dependent. For a set of representative architectures, this chapter discusses the choice of parallel implementation technology (such as threading or message-passing) and recommended implementation techniques (such as incremental refinement and serial consistency). Good implementations properly realize high-level design metrics, ease debugging, and reduce latent bugs.

4.2 Thread Based Implementations

Threads are a basic means of expressing parallelism in a software program and it can be viewed as a lightweight process that the operating system creates, manages, and destroys via a specific mechanism. Independent of the mechanism, the goal is to improve application responsiveness, use machines with multiple processors more efficiently, and improve the program structure and efficiency. Historically, many implementations of threading systems are usually centered around specific platforms or user requirements. The two basic threading models can be described as user level threads (ULT) and kernel level threads (KLT).

For user-level threads, the user is responsible for scheduling and managing the threads, although the kernel manages the process but is unaware of the threads. However, if a ULT makes a call into the kernel, the entire process will block. For KLTs, the kernel must schedule each thread and maintains information about each thread. Process blocking is done on a per thread basis. Hybrid ULT/KLT threading is implemented on platforms such as Linux and Solaris (Figure 14).

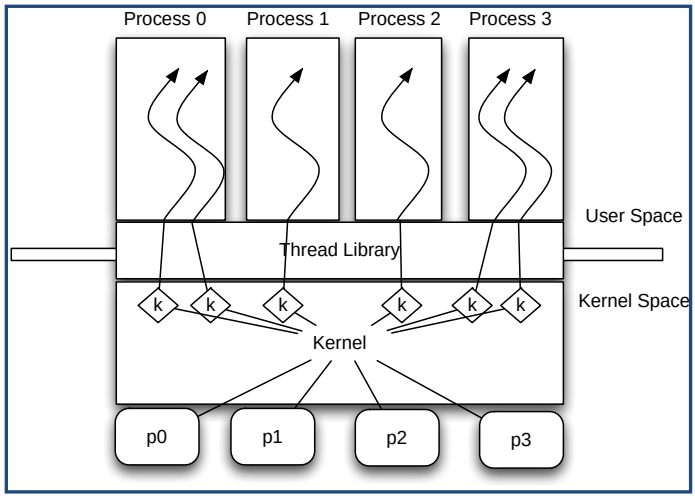


Figure 14. Hybrid threading model.

Threads are created and controlled via function calls for creation, destruction, synchronization, scheduling, and process interaction.

Other basic properties of threads include:

- A thread does not know what thread created it.
- All threads in the same process share the same address space.
- Threads in the same process, share instructions, data, file descriptors, signals and signal handlers.
- Each thread has a unique thread ID, set of registers and stack pointer, stack for local variables and return addresses, a signal mask, priority and its return value.

4.3. Kernel Scheduling

The kernel is programmed to schedule the processes and threads on a machine according to its internal scheduling algorithm. Certain effects from the kernel scheduling threads can be observed:

- The kernel will switch control between threads based on the scheduling algorithm and the state of the machine
- The kernel will save and restore the context of the thread - known as a context switch. This has a detrimental effect on the performance of the software because of the inherent overhead, adversely affecting the behavior of the underlying microprocessor and cache state.
- Kernel scheduling is non-deterministic and does not reliably schedule threads in any particular order
-

4.4 About Pthreads

POSIX threads (Pthreads) is an application programming interface (API) for creating and manipulating threads. The standard defines roughly 100 primitives with optional components and implementation dependent semantics. Implementations of this API are available on most POSIX-compliant platforms including Solaris, MAC OS X, HP-UX, FreeBSD, GNU/Linux, and others. Windows also benefits from a pthreads-w32 project that supports a subset[xiii] of the API. The API is usually provided as a library that forms a layer between the user and the kernel. Although this MPP guide features C++ language examples, the Pthreads library can be accessed from many other languages such as C and Fortran.

Even though POSIX compliant platforms feature Pthreads, some implementations are incomplete or don't conform to the specification. Some API calls may not have underlying operating system kernel

support and do not appear in the library of a given platform - check the platform-specific documentation while developing threaded software.

Beyond this document there are several classic texts[xiv] [xv] [xvi] that cover Pthreads in great depth and even more on-line tutorials.

4.5 Using Pthreads

pthread programs written in C or C++ must include the Pthreads header file to use any of the API functions (*#include <pthread .h>*). On most Unix-like platforms, '*−lpthread*' is typically used to link the library with the user binary file.

Each Pthreads function call takes an opaque variable (object) as a function parameter. The thread library uses this parameter to track the state of that object/resource. For instance the *pthread_create* call takes a *pthread_t* type that contains, amongst other things, the ID of the thread. This ID can be retrieved via the *pthread_self* call.

Most Pthreads functions return a value indicating the success, failure or status of the call. Check these values and deal with them appropriately.

The Pthreads function categories are outlined in Table 1. We will introduce most of this API and its applicability later in the chapter.

Function prefix	Function
pthread_	Thread Management
pthread_attr_	Thread attributes
pthread_mutex_	Mutexes
pthread_mutexattr_	Mutex Attributes
pthread_cond_	Condition variable
pthread_condattr_	Condition variable attributes
pthread_key_	Thread specific data
pthread_rwlock_	read/write locks
pthread_barrier_	Barriers

Table 1. Pthreads functions.

4.6 Dealing with Thread Safety

In order for any code to be thread safe, take specific precautions to prevent multiple threads from accessing a shared resource[xvii]. Some precautions include a) all accesses must have no effect on the resource; b) all accesses are idempotent - the order of operations does not affect the outcome; and c) only one access is allowed at a time.

Any shared memory location that will be encountered by more than one thread of execution must be protected by a synchronization mechanism for writes where all but one thread are mutually excluded from accessing the resource. When more than one thread accesses a resource (seemingly) simultaneously and there is one writer, a race condition will arise because the outcome depends upon the relative execution order of each thread.

Any portion of code that must be synchronized and protected by mutual exclusion is called a critical section. Furthermore, construct the protection mechanism (Figure 15) so that all threads execute the entry and exit routines before entering and exiting the critical section.

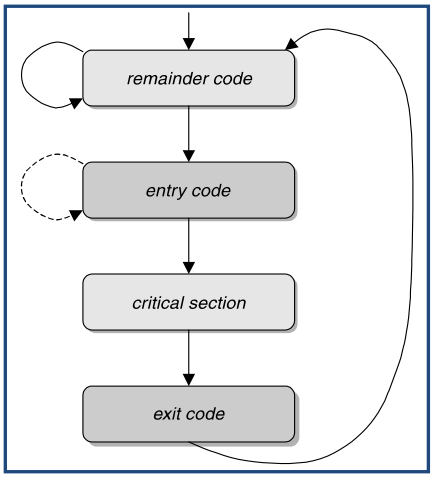


Figure 15. Critical section protection requirements.

For Pthreads, unprotected global or static variables are particularly troublesome and should be avoided. Threads that allocate memory on the stack can take advantage of the thread's stack space offering automatic privacy. Many functions return a pointer to static data, which can be a problem, but this can be remedied by allocating memory on the heap and returning a pointer. Using thread-safe variants of library function calls is also recommended. Furthermore, any third-party libraries should be vetted for thread safety and thread-safe calls should be used.

4.7 Implementing Synchronizations and Mutual Exclusion

Mutual exclusion is enforced when no two threads are in their critical section at the same time. Additionally, this means that deadlock freedom and starvation freedom must be maintained (Table 2[xviii]). Many initial attempts were made to solve this problem (Dijkstra and Dekker Algorithms)[xix xx xxi] and later efforts are popular[xxii xxiii xxiv] with Petersen's Algorithm [xxv] being most popular (there are also some interesting myths about mutex algorithms[xxvi]).

Mutual Exclusion	No two processes are in their critical sections at the same time
Deadlock-freedom	If a process is trying to enter its critical section, then some process, not necessarily the same one, eventually enters its critical section
Starvation-freedom	If a process is trying to enter its critical section, then this process must eventually enter its critical section

Table 2. Mutual Exclusion Properties.

In normal operation, threads compete for resources, cooperate to share them, or cooperate amongst themselves via some type of communication method to guarantee a certain outcome. Depending on the interaction between threads, there are many opportunities for the programmer to fall prey to race conditions, deadlocks, and starvation.

Deadlock[xxvii] is a situation that arises when two threads are both holding a resource that the other needs to continue and they are both waiting on the other to release it. However, without external interference there is no possibility to release the resource and it ends up in a situation termed circular waiting. This can usually be prevented by verification of the lock ordering - the order in which locks are acquired and released. One rule of thumb is to acquire locks in ascending order and release in descending order. When using C++, concepts such as RAII (Resource Acquisition Is Initialization), leveled locking, and lock guards/scope locks can be helpful to avoid missing locks[xxviii]. Figure 16 illustrates a common C++ scope-locking technique. When an object goes out of scope, it's destroyed. The `ScopeLock` class destructor ensures that the mutex is unlocked when the object is destroyed because the mutex unlock is called in the destructor of the object as the stack is wound down.

```
class ScopeLock {
private:
  pthread_mutex_t &lock_;

public:
  ScopeLock(pthread_mutex_t &lock) : lock_(lock) {
    pthread_mutex_lock(&lock_);
  }
  ~ScopeLock() {
    pthread_mutex_unlock(&lock_);
  }
};
```

Figure 16. A common C++ scope-locking technique.

Starvation can happen when attempting to enforce mutual exclusion in a heavily contended critical section. The kernel may schedule two threads in such a way that a third thread must wait indefinitely for a shared resource, thereby starving it and making no progress. Thus, one of the goals of certain locks is to be fair, thereby preventing starvation.

4.8 Mutex, Locks, Nested Locks

Using Pthreads, one can implement synchronization and mutual exclusion using specific Pthreads calls. This mutex, which will effectively protect the critical section, however, without proper implementation, the mutex is still subject to deadlock and starvation. The programmer is responsible for guaranteeing that all uses of the shared resource are protected by a mutex. In Pthreads, only one thread may lock the mutex object at the same time. Any thread that tries to lock the mutex object will block until the thread that holds the mutex releases the object. At this point, one of the threads - not necessarily the thread that arrived first - will be allowed to enter the critical section.

Rules of thumb for critical sections:

- Time - Make the critical section as short as possible – some instructions take longer to execute.
- Space - Execute the minimal amount of instructions during the lock.
- Do not lock any section of code that executes undeterministically.
- When dealing with containers that hold data, lock data items and structures whenever possible, as opposed to the entire container - balance this against granularity concerns.

4.9 Using a Mutex

To use a mutex, it’s necessary to first initialize it and destroy it when finished. This resource initialization/destruction paradigm is common throughout Pthreads (Figure 17).

```
#include <pthread.h>
....

pthread_mutex_t mutex;
int global;

main()
{
pthread_mutex_init(&mutex,NULL); // dynamic initialization

// some code to create and join threads

pthread_mutex_destroy(&mutex);
}

void thread_one()
{
// some code for thread one
pthread_mutex_lock(&mutex);
++global;
pthread_mutex_unlock(&mutex);
// some other code for thread one
}

void thread_two()
{
// some code for thread two
pthread_mutex_lock(&mutex);
--global;
pthread_mutex_unlock(&mutex);
// some other code for thread two
}
```

Figure 17. A partial example of initializing and locking a critical section with a mutex.

4.10 Condition Variables

Condition variables are a method of enforcing mutual exclusion based on a change in a variable. For example, one can create a bounded buffer, shared queue, or software FIFO using two condition variables and a mutex (Figure 18). A C++ class is created that wraps a standard library `queue` object and uses the *ScopeLock* class to enforce lock discipline.

```
class IntQueue
{
private:
  pthread_mutex_t mutex_;
  pthread_cond_t more_;
  pthread_cond_t less_;
  std::queue<int> queue_;
  size_t bound_;

public:
  IntQueue(size_t bound) : bound_(bound) {
    pthread_mutex_init(&mutex_, NULL);
    pthread_cond_init(&less_, NULL);
    pthread_cond_init(&more_, NULL);
  }
  ~IntQueue() {
    pthread_mutex_destroy(&mutex_);
    pthread_cond_destroy(&more_);
    pthread_cond_destroy(&less_);
  }
  void enqueue(int val) {
        pthread_mutex_lock(&mutex);
    while(queue_.size() >= bound_)
        pthread_cond_wait(&less_,&mutex_);
    queue_.push(val);
    pthread_cond_signal(&more_);
    pthread_mutex_unlock(&mutex);
  }
  int dequeue() {
        pthread_mutex_lock(&mutex);
    while(queue_.size() == 0) {
      pthread_cond_wait(&more_, &mutex_);
    }
    int ret = queue_.front();
    queue_.pop();
    pthread_cond_signal(&less_);
        pthread_mutex_unlock(&mutex);
    return ret;
  }
  int size() {
    ScopeLock lock(mutex_);
    return queue_.size();
  }
};
```

Figure 18. Simple concurrent queue using two condition variables and a mutex.

4.11 Levels of Granularity

Granularity is the ratio of computation to communication in a parallel program. A program will typically have sections that are computing and sections that are communicating, whereby communication means anything related to controlling, locking, and destroying threads. There are two basic types of granularity in parallel computing; fine-grained and coarse-grained.

Fine-grained parallelism is small amounts of work done or that there is a low computation to communication ratio. This can facilitate load balancing, but relatively large amounts of time will be spent coordinating the events and communicating their outcome to the other threads. It's also possible to produce a situation where there is more communication than computation, such as when the program spends most of its time in the kernel dealing with mutexes or condition variables.

Coarse-grained parallelism is where there is a high computation to communication ratio. This might indicate that the performance would increase, but depending on the algorithm's implementation, it may require more effort to achieve a reasonable load balance across a given machine.

For deciding on the appropriate granularity, a useful rule of thumb is that there must be enough work to amortize the cost of communication over the computation. Start with a coarse-grained approach and look for opportunities to implement fine-grained parallelism. More parallelism is usually better, but this alone is not the determining factor for performance. This balance is usually dependent on the scheduling strategies, algorithm implementation, the computation to communication ratio[xxix], as well as the available processing resources. Maximum performance is only realized after considerable experimentation and applied performance tuning. Additionally, the number of concurrent work items should be close to the number of cores in the machine to avoid over-subscription.

When using most Pthreads primitives, the library will ask the kernel to help it perform the task, which in turn causes its own set of performance problems. Have an idea of the relative overhead for Pthreads calls. This knowledge combined with understanding the potential concurrency in the software (Amdahl's/Gustavson's Law)[xxx] will help with the optimization process.

4.12 Implementing Task Parallelism

Task parallelism implies that each task is executed with a separate thread and the threads are mapped to different processors or nodes on a parallel computer. The threads may communicate with each other to pass data as part of the program structure. It's likely that tasks are different and that the algorithm has imposed control and data flow dependencies, thereby ordering the tasks and limiting the amount of parallelism.

Implementing task parallelism with Pthreads is:

- Creating or identifying a group of program functions that will be used as tasks - called thread functions
- Creating the threads that will execute the thread functions and passing the necessary data to the functions
- Joining with and retrieving the results returned from the thread functions, if any
- Minimally, all critical sections must be protected – this pertains to locations where shared resources will be changed by at least one thread and accessed by others

4.13 Creation and Join

Threads are created and joined using the `pthread_create` and `pthread_join` calls ("joining" is one way to accomplish synchronization between threads). Threads can also pass to and receive information from the calling thread, allowing threads to communicate. In the following example (Figure 19), illustrating the creating and joining of threads, we are passing a thread number to the thread. With a small modification, we could retrieve the value returned using the `pthread_exit` call in the thread function using the second parameter of `pthread_join`.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define NUM_THREADS 5

void *print_hello( void *threadid )
{
        printf( "\n%d: Hello World!\n", ( int ) threadid );
        pthread_exit( NULL ); // potenitally return some value here
}

int main()
{
        static pthread_t threads[ NUM_THREADS ];
        int rc, t;
        for ( t = 0; t < NUM_THREADS; t++ ) {
                printf( "Creating thread %d\n", t );
                rc = pthread_create( &threads[ t ], NULL, print_hello, ( void * ) t );
                if ( rc ) {
                        printf( "ERROR; pthread_create() returned %d\n", rc );
                        printf( "Error string: \"%s\"\n", strerror( rc ) );
                        exit( -1 );
                }
        }
        for ( t = 0; t < NUM_THREADS; t++ ) {
                printf( "Waiting for thread %d\n", t );
                rc = pthread_join( threads[ t ], NULL); // potentially get a return value here
                if ( rc ) {
                        printf( "ERROR; pthread_join() returned %d\n", rc );
                        printf( "Error string: \"%s\"\n", strerror( rc ) );
                        exit( -1 );
                }
        }
        return 0;
}
```

Figure 19. Pthreads creation and join.

4.14 Parallel Pipeline Computation

As previously discussed, a pipeline is a collection of sequentially-executed tasks, whereby the input of the first task is obtained from some data source and the result of one stage is fed to the next. Each pipeline stage represents one parallel task. The pipeline consists of three separate phases: 1) Receive (a task); 2) Compute (perform some computation on the task); and 3) Send (send the task to the next stage).

A parallel pipeline system has temporal and data dependencies (Figure 20), therefore synchronization at the output of each stage might need to be considered.

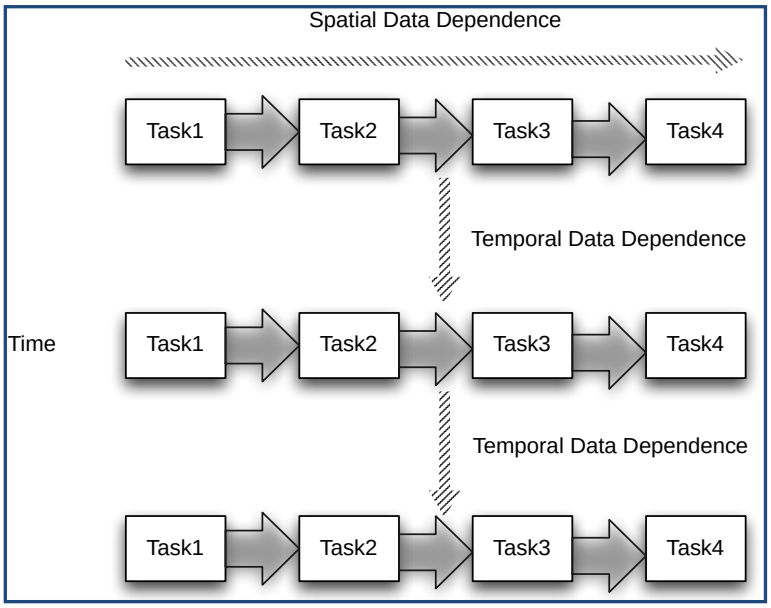


Figure 20. Parallel pipeline system with temporal and data dependencies.

A pipeline can be created using the Queue concept introduced earlier, where each stage in the pipeline is separated by a queue object. Each stage has one or more threads that remove a data item from the previous queue, perform some work and then place the data object in the queue of the next block.

4.15 Master/Worker Scheme

The master/worker scheme is a technique that can be used for a task pool with dynamic task distribution (Figure 21). A master thread passes work to worker threads and collects the results. In this scheme, the many-to-one communication system enables the master to evenly distribute the work by supplying a worker with a new task each time a result is collected. It's also possible to give many tasks to a worker, provided this does not affect load balancing.

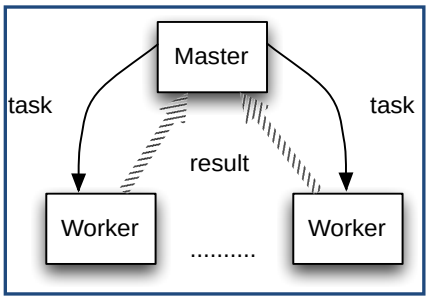


Figure 21. Master/worker scheme supports dynamic task distribution.

The master/worker system relies on the relatively smooth operation of the master thread which is responsible for collecting results and emitting tasks. When a large number of workers are used, the master thread may become overloaded. Conversely, allowing the worker threads to perform more

work (coarse-grained approach) might limit the dynamic nature of the system and affect load balancing. To combat these problems, it's possible to set up a hierarchy of master/worker thread systems as a tree, with worker processes as the tree's leaves and sub-masters as the inner nodes.

4.16 Divide and Conquer Scheme

The divide and conquer scheme is an important algorithmic expression used to solve many sorting, computational geometry, graph theory, and numerical problems. It has three distinct phases.

1) Divide Phase - A problem is split into one or more independent sub-problems of smaller size
2) Conquer Phase - Each sub-problem is solved recursively or directly
3) Combine Phase - The sub-solutions are combined to a solution of the original problem instance

In the divide-and-conquer model, one or more threads perform the same task in parallel (SPMD model). There is no master thread and all run independently. Recursive calls can be made concurrently only if the calls write to different parts of the program's memory. Divide and conquer algorithms are subject to load-balancing problems when using non-uniform sub-problems, but this can be resolved if the sub-problems can be further reduced.

4.17 Task Scheduling Considerations

The determination of task granularity is a partitioning problem. The program must be decomposed into a set of tasks suitable for parallel execution[xxxi]. The 'grain size' can be defined as a series of instructions organized as a task. Depending on the language, this may be a function or a loop body.

An efficient use of the available resources on a parallel system requires enough tasks to spread over the processor cores allowing tasks to execute in parallel. The programmer must determine the optimal task size that will give the shortest execution time. The larger the task size, the less parallelism that is available on a given machine. A smaller task size will result in greater parallel overhead (communication and synchronization)

The general solution to the granularity problem is NP-complete[xxxii], but it's possible to find a near optimal solution to a sub-problem[xxxiii]. Techniques such as task packing[xxxiv] [xxxv] and work stealing[xxxvi] [xxxvii] are also beneficial to arrive at a tuned scheduling algorithm.

4.18 Thread Pooling

A thread pool is a collection of threads that will perform a given function or task. Primarily, this technique allows the programmer to reuse threads and avoid the overhead of creating and destroying

threads multiple times. The programmer may control the pool dynamics and size, although it frequently depends on the system state, such as the number of tasks or available processors. The pool is tuned to provide the best performance given the load and machine characteristics.

4.19 Affinity Scheduling

Depending on the algorithm, resource acquisition/management scheme, and the workload, certain threads may benefit from having their affinity set to specific processors. In some instances the operating system kernel will have support for scheduling threads based on power, cache, CPU, or other resources. For example, the kernel may be able to detect that a thread will benefit from continued execution on a given processor based on the cache usage (i.e. to avoid cache flushing and refilling). This cache-aware version is known as cache affinity scheduling.

On most processor architectures, migration of threads across cache, memory, or processor boundaries is expensive (i.e. TLB flush, cache invalidation) and can reduce program performance. The programmer is able to set affinities for certain threads to take advantage of shared caches, interrupt handing, and to match computation with data (locality). Additionally, by setting the affinity to a single CPU and excluding other threads from using that CPU, and that processor is dedicated to running that thread only, and takes that processor out of the pool of resources that the OS can allocate. When the affinity scheduling is manually controlled, the programmer should carefully design the affinity scheduling scheme to ensure efficiency.

Each operating system will use different calls to set affinity, binding the thread to the processor - for Linux, it's *sched_setaffinity* and *pthread_setaffinity_np* (np refers to non-portable); for Solaris it's *processor_bind*. Both have similar semantics.

In the following example that shows the function call for cache affinity scheduling (Figure 22), the mask is a bitmap of all the cores in the machine (Figure 23). If *sched_setaffinity* is called again later with a different mask value, the OS migrates the thread to the requested processor.

```
unsigned long mask = 1; /* processor 0 */

/* bind the calling process to processor 0 */
if (sched_setaffinity(0, sizeof(mask), &mask) <0)
        {
        perror("sched_setaffinity");
        }
```

Figure 22. Code snippet using bitmap mask for cache affinity scheduling.

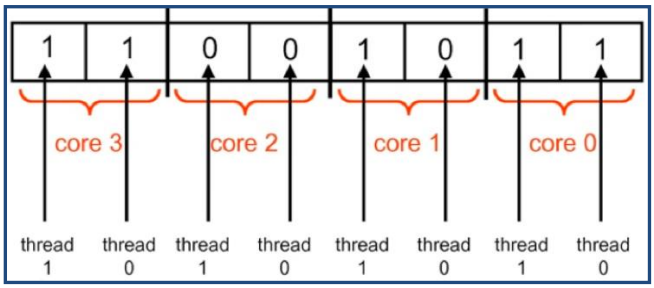


Figure 23. Affinity mask.

4.20 Event Based Parallel Programs

Use an event based coordination approach if an application can be decomposed into groups of semi-independent tasks that interact in an irregular manner and if the interaction between them is determined by the flow of data between them (which implies ordering constraints between the tasks). These tasks, and their interaction, can be implemented to execute concurrently. This approach does not have a restriction to a linear structure or a restriction that the flow of data is uni-directional, however, the interaction can take place at irregular and sometimes unpredictable intervals.

Using a laundromat as an example, where there are a number of washers and dryers available for use, people arrive with their laundry at random times. Each person is directed by an attendant to an available washing machine (if one exists) or queued if all are busy. Each washing machine processes one load of laundry at a time. The goal is to compute, for a given distribution of arrival times, the average time a load of laundry spends in the system (wash time plus any time waiting for an available machine) and the average length of the queue. The "events" in this system include loads of laundry arriving at the attendant, loads of laundry being directed to the washing machines, and loads of laundry leaving the washing machines. Think of this as "source" and "sink" objects to make it easier to model loads of laundry arriving and leaving the facility. Also, the attendant must be notified when loads of laundry leave the washing machines so that it knows whether the machines are busy.

Viewing this from a programming perspective, as opposed to conventional programming where a program is written as a sequence of actions, in event-based programming the application developer writes a set of event handlers. Incoming events activate these event handlers. When the number of interacting processes is large, understanding a program that is coded as a series of responses to many different events is difficult. At any moment, one single process may be interacting with dozens of other processes, and each of these interactions may require its own state information. This necessitates the need for abstractions to coordinate interacting processes.

4.21 Implementing Loop Parallelism

Loop parallelism (Data Parallelism) is potentially the easiest type of parallelism to implement, while achieving the best speed-up and scalability (Figure 24). By dividing the loop iteration space by the number of processors, each thread has an equal share of the work. If the loop iterations do not have any dependencies and the iteration space is large enough, good scalability can be achieved. Additionally, this implies that each loop iteration takes relatively the same amount of time and the program might be free of load-balancing problems.

```
#include <pthread.h>

struct ThreadParam {
        int startIndex_;
        int endIndex_;
        int threadNb_;  // thread number
        pthread_t threadID_; // pthreads thread object
        VectorsStruct *vectors_; // vectors we're going to calculate
};

static void *threadFunction(void *paramPtr) {
   ThreadParam *param = (ThreadParam *)paramPtr; // get our "instructions"
   std::cout<<"Thread: "<<param->threadNb_<<" from "<<param->startIndex_<<" to "
               << param->endIndex_<< std::endl;
   compute(param->vectors,param->startIndex_, param->endIndex_); // compute the vectors
   return NULL;
}

main{}
{
  int nthreads = 8; // Adjusted for the witdth of the machine
  ThreadsParam* paramPtr[nthreads]; // each thread needs it's set of "instructions"
  VectorsStruct vectors(A, B, C); // need something to calculate

  /* Creation of the threads. */
  for(int i = 0; i < nthreads; ++i)
  {
        int startIndex = (size * i) / nthreads; // each thread gets its own start index
        int endIndex = (size * (i + 1)) / nthreads; // each thread gets its own end index
        ThreadParam *paramPtr[i] = new ThreadParam(startIndex, endIndex, i + 1, vectors);
        int status = pthread_create(&paramPtr->threadID_, NULL, threadFunction, paramPtr);
        // check status
  }

  // join on the threads and delete ThreadsParam array.
}
```

Figure 24. An example of loop parallelism using Pthreads.

4.22 Aligning Computation and Locality

Locality is one of the key considerations for performance oriented codes. Application performance degrades as memory access latency becomes high relative to processor cycle times. To avoid this, keep data close to the processor that is performing the computation. Furthermore, the program must be structured in such a way as to allow the microprocessor caches to make most efficient use of memory. There are two types of locality involved: 1) Temporal locality (a memory location used is likely to be used again); and 2) Spacial locality (if a data item is referenced, the neighboring data locations may also be used).

There are many techniques for locality optimization of codes. For example, cache blocking is reorganizing data objects to fit in the microprocessor cache. Loop tiling is also a technique to break a loop's iteration space into smaller blocks thereby insuring that the data of interest stays in the cache.

4.22.1 NUMA Considerations

Non-uniform Memory Access (NUMA) architectures are popular in many modern microprocessors. A NUMA architecture is where the entire memory space is shared but each processor is connected to a node with a dedicated memory controller (Figure 25). This allows system designers to build larger systems, but the programmer must take greater care with data and memory affinity because off-node data access times will be greater than accesses time within a node. Since access time to memory is not uniform, specific code optimizations are necessary to guarantee performance.

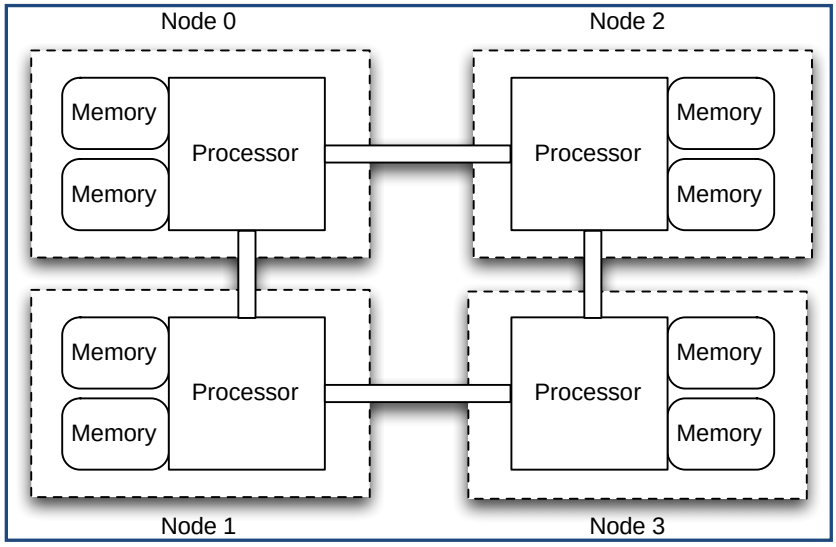


Figure 25. With NUMA architecture, each processor core connects to a node with dedicated memory.

4.22.2 First-Touch Placement

First-touch placement allocates memory on the NUMA architecture containing the processor that first touches (writes to) the memory. A first-touch memory placement policy can greatly improve the performance of applications where data accesses are made mostly to the memory local to each processor. First-touch placement is the default on most Unix like operating systems and it usually supports an API that allows for the manipulation of the placement policies.

Programmers must take care to initialize their data structures using the thread that will also perform the computation. This will ensure that memory is placed closest to the node when the thread is executing. Additionally, to guarantee the best memory performance, threads should be pinned to the processor on the node where the memory of interest is located.

4.23 Message Passing Implementations

The Multicore Association (MCA) has created two specifications that address the area of inter-process communications (IPC) between CPUs in a multicore system. These specifications include the Multicore Communications API (MCAPI®) and the Multicore Resource Management API (MRAPI®).

To understand how MCAPI might fit into your system, look at how to divide the system and its resources among the available operating systems and CPUs. The system itself must be designed both with the hardware and software in mind. What are the system requirements? What is the minimum latency needed to process a datagram? Are there any deterministic requirements that must be adhered to? Can the application be parallelized?

It's important to note the differences between asymmetric multiprocessing (AMP) and symmetric multiprocessing (SMP). In some situations, SMP hardware is best used by running a single operating system across all cores. However, sometimes it can be advantageous to distribute several operating system instances (similar to AMP systems) across cores in an SMP system. In addition, it might be optimal to run an application directly on a CPU without an operating system, otherwise known as a "bare-metal" application.

After determining that using AMP is advantageous, or deemed a requirement in your system design, there must be an inter-processor communication (IPC) mechanism for passing messages between those instances. MCAPI specifically deals with message passing between nodes in the system. MCAPI can be used at a very low layer to pass messages between operating systems, making it possible to abstract away the hardware differences and uniqueness from the application.

4.23.1 MCAPI

There are three primary concepts to understand with MCAPI: a node, an endpoint, and a channel. An MCAPI implementation passes messages between nodes; where a node can be a CPU, an operating

system instance, a thread, or a process. Initialization software establishes each node and creates a set of endpoints for inter-node communication. The system designer must decide the appropriate configuration; your system can have different types of nodes. For example, two operating systems can run side-by-side on separate CPU combinations. In this example, the first operating system could support five processes running on that CPU. On the second operating system, you might consider the entire operating system as a single node.

There are two types of inter-node communication: connection-less and connected. A connection-less endpoint can receive communication from many different nodes at any time. A connected channel is when communication passes between two distinct endpoints; no other nodes can connect to that endpoint.

A channel can be either “socket-like” in its message passing, where all messages appear in their transmitted order. Alternatively, channels can be “scalar-based” where a simple integer message will be passed between endpoints. For example, a video frame would be passed in its entirety as a datagram, whereas a command can be passed as a scalar.

By using MCAPI at a very low layer to pass messages between operating systems, you can abstract away all of the hardware differences and uniqueness from the application. Check out the MCAPI programming source code example in Appendix. This example involves a printer function with two operating systems (Android and Nucleus), and includes a flow chart with initialization and the message process.

4.23.2 MRAPI

MRAPI deals with resources, such as sharing memory and synchronizing objects between processing nodes or operating system instances. Memory sharing APIs allow a single block of memory, owned by one operating system, to be shared among several nodes in a system. These APIs can enforce only one writer or multiple readers, which can access the memory block at any time.

Synchronization APIs provide applications running on different nodes with the ability to synchronize their activities. An MRAPI semaphore is a system-wide semaphore that can be accessed by any node in the system, preventing access by multiple nodes to a shared resource.

4.23.3 MCAPI and MRAPI in Multicore Systems

Let's look at how MCAPI/MRAPI can be used in a multicore system to provide the necessary intercommunication framework between nodes. In this example we have four cores, each of which is allocated a set amount of memory from main memory. In addition, there are memory regions shared by all cores.

In the first step, MRAPI initializes any shared memory region that will be accessible between the system's nodes. Next, during MCAPI initialization, the MRAPI shared memory is distributed to each of the endpoints or channels that have been established between each of node. These nodes can be local or remote, but the shared memory region must be used in order to pass messages between nodes. Interrupts may be used to interrupt current node processing in order to minimize system latency for higher priority communications.

The combination of using MCAPI and MRAPI together is a compelling approach that allows full-featured APIs to be implemented in an AMP system across any of the system's nodes. This even includes nodes that are remote to all other nodes, but must be passed through intermediate nodes to get data to its destination by setting up a route. The way your system is architected will either enable or impair the ability to pass messages efficiently throughout the system.

4.23.5 Using a Hybrid Approach

Using standard POSIX threads inside a node and using MCAPI with MRAPI between nodes, you can develop a system architecture that allows high portability between varied systems. However, when load balancing between nodes (a thread of execution migrates to another node), the API would change because what once was local is now remote, or vice versa. The solution is to use MCAPI/MRAPI for all messaging between threads as well as nodes. If the communications do not leave the node, the MCAPI/MRAPI layer would translate into native message passing such as POSIX on the node. While this adds another layer of abstraction in using MCAPI/MRAPI on a single node, it adds the benefit of being able to load balance across nodes, using MCAPI/MRAPI as a common API for all types of inter-/intra-node communications. However, if you need to maximize CPU efficiency, then while using MCAPI/MRAPI as a communications layer allows for future flexibility in system architecture, its benefit may be superseded by the need for speed.

During development, you can use MCAPI/MRAPI APIs on a single operating system instance, with either a single or multiple nodes, with all the appropriate endpoints, and with all the appropriate tasks, and the algorithm will run. This is an easy way to test the application on a single core operating system node, prior to the multicore hardware being available. Although it does not test all the parallel threading, it does allow the logical nature of the algorithms to be tested.

4.24 Task-based Implementations

4.24.1 MTAPI

Besides MCAPI and MRAPI, the Multicore Association defined MTAPI, a set of interfaces for task management in embedded systems. MTAPI allows to split complex computations into fine-grained tasks that can be executed in parallel. Threads are usually too heavy-weight for that purpose, since context switches consume a significant amount of time. Moreover, programming with threads is complex and error-prone due to concurrency issues such as race conditions and deadlocks.

While task schedulers are nowadays widely used, especially in desktop and server applications, they are typically limited to a single operating system running on a homogeneous multicore processor. System-wide task management in heterogeneous systems must be realized explicitly via communication mechanisms. MTAPI addresses these issues by providing an API which allows parallel software to be designed in a straightforward way, covering homogeneous and heterogeneous multicore architectures. It abstracts from the hardware details and lets software developers focus on the application.

MTAPI provides two basic programming models, tasks and queues, which can be implemented in software or hardware (Figure 26). Tasks are fine-grained pieces of work executed locally (on a multicore processor) or on a remote node, e.g. an accelerator. For that purpose, MTAPI provides mechanisms to pass data to the remote node and back to the calling node. Queues can be used to control the scheduling policy for related tasks. For example, order-preserving queues ensure that subsequent tasks are executed sequentially.

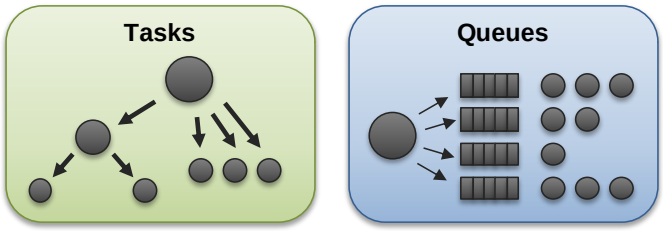


Figure 26. Basic MTAPI programming models

Figure 27 depicts a sample architecture based on MTAPI utilizing the CPU, a DSP, and a GPU. Each of them is represented by one or more MTAPI nodes (one core of the CPU is reserved for special purposes, e.g. processing real-time tasks, and constitutes an own node). From an application perspective, executing a task on such a system is accomplished via unified interfaces hiding the specific characteristics of the different hardware units.

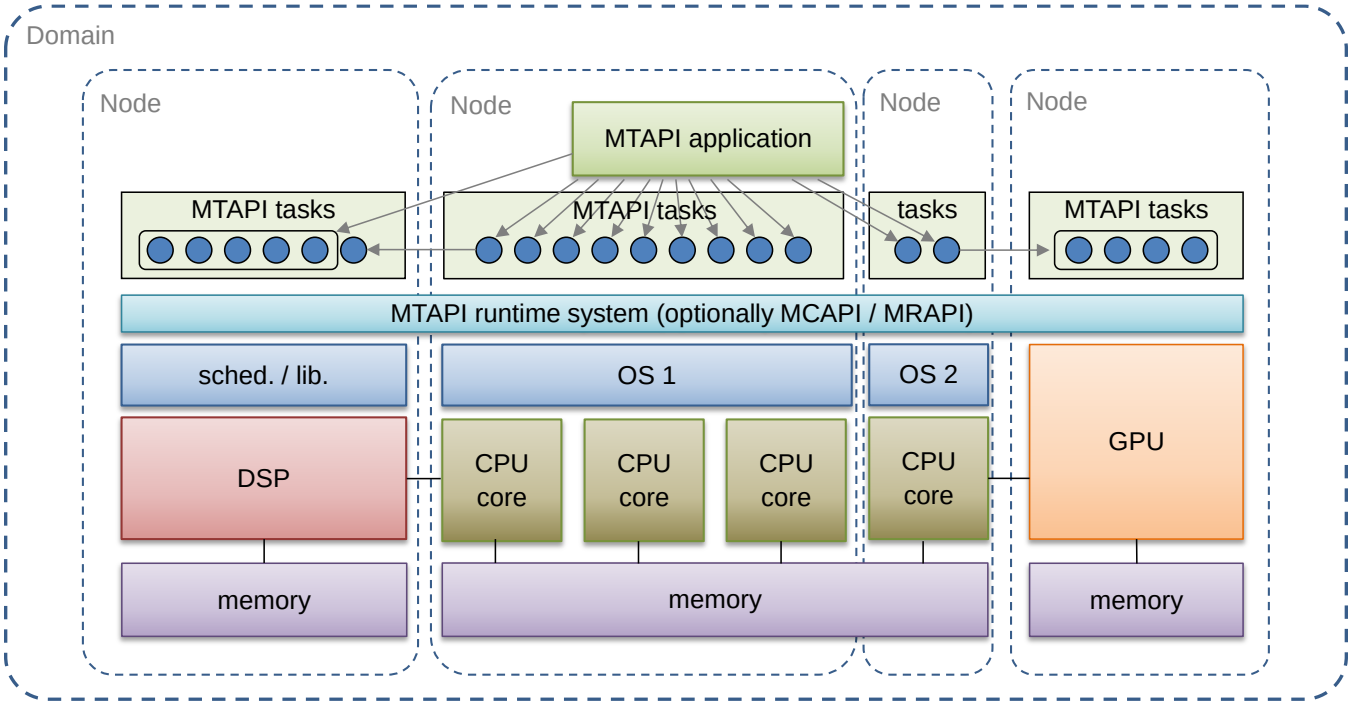


Figure 27. Example for MTAPI-based architecture

In the following, we describe the main concepts of MTAPI in more detail:

Node: An MTAPI node is an independent unit of execution. A node can be a process, a thread, a thread pool, a general purpose processor or an accelerator.

Job and *Action*: A job is an abstraction representing the work and is implemented by one or more actions. The MTAPI system binds tasks to the most suitable actions during runtime.

Task: An MTAPI task is an instance of a job together with its data environment. Since tasks are fine granular pieces of work, numerous tasks can be created, scheduled, and executed in parallel. A task can be offloaded to a neighboring node other than its origin node depending on the action binding.

Group: MTAPI groups are defined for synchronization purposes. A group is similar to a barrier in other task models. Tasks attached to the same group must be completed before the next step by calling an API function (`mtapi_group_wait`).

An MTAPI-compliant scheduler distributes tasks among the available processing units and combines the results after synchronization. However, during creation, the task does not know with which action it will be associated. MTAPI provides a dynamic binding policy between tasks and actions. This allows it to schedule jobs on more than one hardware type, where the scheduler is responsible for balancing the load. Depending on where a task is located, it is either a local task (if it is implemented by an action residing on the same node) or a remote task. Figure 28 shows an example for the relationship between tasks and actions. In the example, Tasks 1 and 2 refer to Job A, which is implemented by Actions I and II on

remote nodes. In contrast, Task 3 accomplished by Job B is executed on the same node as the application (Action III).

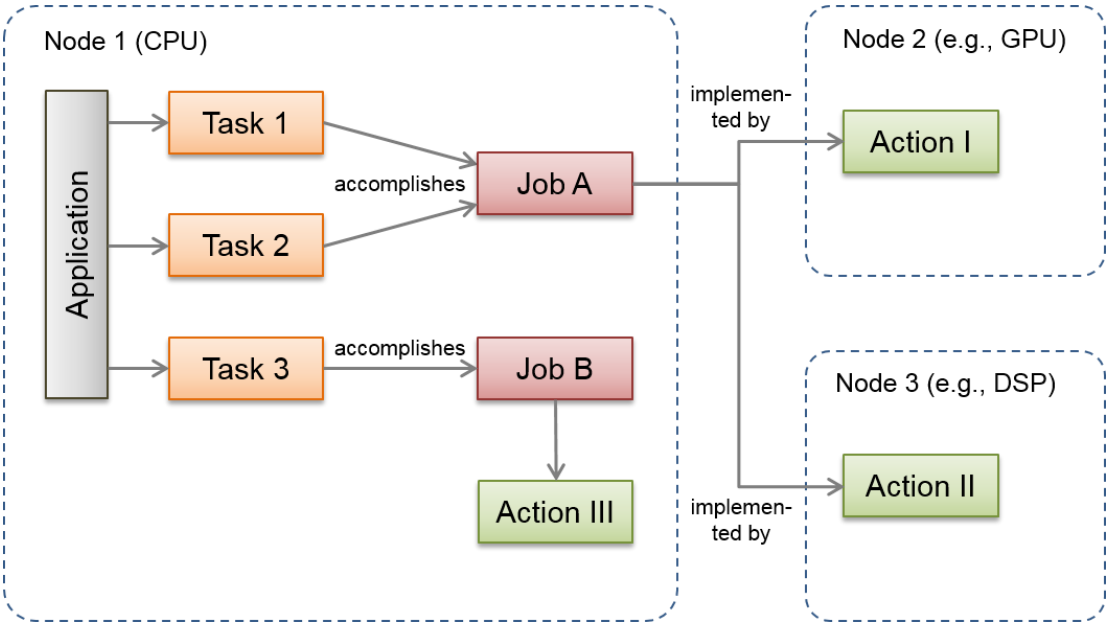


Figure 28. MTAPI tasks, jobs, and actions

4.24.2 EMB²

In this section, we give a brief overview of the Embedded Multicore Building Blocks[xxxviii] (EMB²), an open source C/C++ library for the development of parallel applications based on MTAPI. EMB² has been specifically designed for embedded systems and the typical requirements that accompany them, such as predictable memory consumption and support for timing-critical applications. Besides a task scheduler, EMB² provides basic parallel algorithms, concurrent data structures, and skeletons for implementing dataflow-based applications (Figure 29).

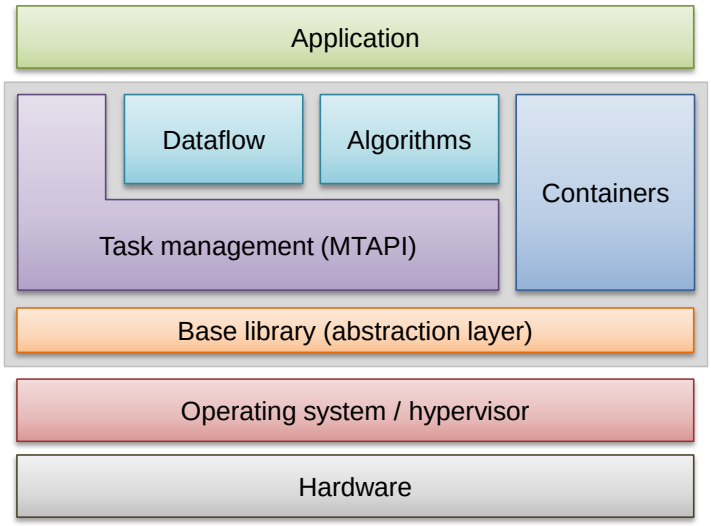


Figure 29. EMB² components

Figure 30 shows an example for task creation and synchronization (for convenience, we use C++ wrappers for the corresponding MTAPI functions). The main function first gets a reference to the current node. By calling `Start`, it then creates a task that executes the function `work` in parallel to `main`. Finally, it waits for the task to finish. Instead of an ordinary function, we could also pass a job to `Start` in order to execute an action on an accelerator. Note that the function `work` takes as parameter a reference to an object of type `TaskContext`, which provides information about the status of the task and can be used to set an error code to be returned by `Wait`.

```
using namespace embb::mtapi;

void work(TaskContext& task_context) {
  // do work ...
}

int main() {
  Node& node = Node::GetInstance();
  Task task = node.Start(&work);
  // do something...
  mtapi_status_t status = task.Wait();
  // check status ...
}
```

Figure 30. Example for task creation and synchronization

The algorithms provided by EMB² can be used for more complex computations such as parallel loops. They split the computations to be performed into tasks, which are executed by the MTAPI task scheduler. Suppose, for example, we are given a range of integers and want to double each of them. In principle, one could apply the operation to all elements in parallel as there are no data dependencies. However, this results in unnecessary overhead if the number of elements is greater than the number of available processor cores. A better solution is to partition the range into blocks and to process the

elements of a block sequentially. With the `ForEach` construct, users do not have to care about the partitioning of the range into chunks of reasonable size, since this is done automatically. Similar to the C++ Standard Library's `for_each` function, it is sufficient to pass the operation in form of a function object or a lambda function (Figure 31).

```
std::vector<int> v;

// initialize v ...

embb::algorithms::ForEach(v.begin(), v.end(), [] (int& x) {x *= 2;});
```

Figure 31. Example for parallel loop

Similar to the previous example (Figure 30), we could also pass a job to the loop instead of a lambda function. This way, it is even possible to process the elements of a range on different compute units, e.g. the CPU and a GPU, where the task scheduler automatically distributes the load by means of dynamic action binding.

For further information and examples, the reader is referred to the MTAPI specification[xxxix] and the EMB² website[xl].

CHAPTER 5: DEBUG

5.1 Introduction

The adoption of multicore systems requires programmers to have a better understanding of how to effectively design, write, and debug parallel programs. Parallelism introduces complexities that are exacerbated by potentially non-deterministic and asynchronous tasks interleaved by the system scheduler. This makes debugging difficult, since concurrency related bugs may not manifest themselves in every application execution and consecutive runs of the same application may yield different results.

5.2 Parallel Processing Bugs

In addition to requiring proficiency in parallel and multi-threaded software design and development, parallel programming on multicore platforms requires developers to better understand and coordinate between available resources to ensure application stability and enhanced performance.

Threads in a multi-threaded parallel application may be arbitrarily interleaved in time, potentially resulting in different possible instruction interleaving with different threads of execution. The issues encountered in multi-threaded applications include[xli]:

1. Thread stalls caused by a thread locking a resource for a long period which causes other threads requiring the locked resource to wait for a long time.
2. Thread convoying results from thread stalls, where multiple threads may wait to acquire a lock. As threads wait for locks to be released, the system switches through all runnable, blocked threads.
3. Data races (Figure 32): result from two concurrent threads performing conflicting accesses with no explicit mechanism implemented to prevent simultaneous access. Synchronization and critical sections may be used to correct data races. Unnecessary synchronization must be avoided to prevent negative impact on application performance.
4. Deadlocks caused by locking hierarchies, resulting from all threads being blocked as each thread waits on an action by another thread.
5. Livelocks : caused by threads detecting and recovering from deadlock, as processes/threads constantly change with respect to one another, trigger the deadlock detection algorithm.

The code sample in Figure 32 shows two functions, where one function contains a data race and the second function shows how to fix the data race. The first function, thread_broken, is spawned twice by the main routine. This function increments the global variable, global_broken. Since the read and increment is not atomic, it is possible for one thread to read the original value (zero) while the other thread is in the process of incrementing it and writing the value back to memory. The latter increment occurs on a stale value with the resulting write not showing the fact that the variable global_broken should have incremented twice.

The second function, thread_fix, shows how to resolve the data race issue. A mutex is created and protects access to the increment of global_fix. The mutex serializes access to the code between the pthread_mutex_lock call where the mutex is acquired and the pthread_mutex_unlock call, where the mutex is released.

```
   1  int global_broken;


   2  static void* thread_broken(void* p) {
   3    global_broken++;
   4  }
   5
   6  int global_fix;
   7  static pthread_mutex_t mutex;
   8
   9  static void* thread_fix(void* p) {
   10   pthread_mutex_lock(&mutex);
   11   global_fix++;
   12   pthread_mutex_unlock(&mutex);
   13 }
   14
   15 int main(int argc, char *argv[])
   16 {
   17   pthread_t threadID1, threadID2, threadID3, threadID4;
   18
   19   global_broken = 0;
   20   pthread_create(&threadID1, NULL, thread_broken, NULL);
   21   pthread_create(&threadID2, NULL, thread_broken, NULL);
   22
   23   global_fix = 0;
   24   pthread_mutex_init(&mutex, NULL);
   25   pthread_create(&threadID3, NULL, thread_fix, NULL);
   26   pthread_create(&threadID4, NULL, thread_fix, NULL);
   27
}
```

Figure 32. Code sample shows data race condition.

The code sample in Figure 33 shows an occurrence of deadlock. The function, deadlock, acquires a mutex, but does not release it. The main function attempts to acquire the lock and stalls indefinitely because the deadlock function never releases.

```
1  int global;
2  static pthread_mutex_t mutex;
3
4  static void* deadlock(void* p) {
5    pthread_mutex_lock(&mutex);
6    global++;
7  }
8
9  int main(int argc, char *argv[])
10 {
11   pthread_t threadID1;
12
13   global = 0;
14
15   pthread_mutex_init(&mutex, NULL);
16   pthread_create(&threadID1, NULL, deadlock, NULL);
17
18   pthread_mutex_lock(&mutex);
19   global++;
20
}
```

Figure 33. Code sample showing a Deadlock condition.

Livelocks are similar to deadlocks, and are a special case of resource starvation. In livelocks, there is a constant change in the states of the processes with respect to one other, preventing either one of them from progressing. Livelocks occur in algorithms which detect and recover from deadlocks, repeatedly triggering the deadlock detection algorithm. Livelocks can be prevented by allowing only one process to take action.

A livelock occurs when a request for an exclusive lock for a shared resource is repeatedly denied, since a series of overlapping shared locks, continue to interfere with each other. At the end two or more threads may continue to execute, with neither one making any progress. One example of livelock occurs when two or more processes or threads are set to repeat a set of actions until a given condition tests true. However, the two processes or threads may cancel each other's actions before testing for the condition, causing all of them to perpetually re-start. The threads (or processes) are not blocked, they simply change their respective states, in response to changes in the other thread(s) (or process(es)), preventing all of them from making any progress.

Another example of livelocks may occur where atomic thread safety is enforced. For example, one thread locks the system and goes to sleep. Another thread attempting to access data, spins over the same instructions, until it automatically goes to sleep.

5.3 Debug Tool Support

Tools supporting multi-threaded code debugging, should at a minimum, support switching between threads and examination of thread state during a debugging session. The enhanced complexity of multithreaded applications results from factors such as non-deterministic thread scheduling and pre-emption, and dependencies between control flow and data[xlii]. Non-deterministic execution of multiple instruction streams, from runtime thread scheduling and context switching, generally stems from the operating system's scheduler. Debuggers may mask issues caused by thread interactions, such as deadlocks and race conditions. Factors such as thread priority, processor affinity, thread execution state, and starvation time can affect the resources and execution of threads.

Thread-related bugs are difficult to find due to their non-determinism and certain bugs may only be observed when threads execute in a specific order. The more common threading bugs include data races, deadlocks, and thread stalls. Synchronization may solve data races, but may result in thread stalls and deadlocks.

A number of approaches are available for debugging concurrent systems including traditional debugging and event based debugging. Traditional debugging (breakpoint debugging) has been applied to parallel programs, where one sequential debugger is used per parallel process. These debuggers can only provide limited information when several processes interact. Event-based debuggers (or monitoring debuggers) provide some replay functionality for multi-threaded applications, but can result in high overhead. Debuggers may display control flow, using several approaches such as textual presentation of data, time process diagrams, or animation of program execution[xliii].

The use of a threading API may impact the selection of a debugger. Using OpenMP, for example, requires the use of an OpenMP-aware debugger, which can access information such as constructs and types of OpenMP variables (private, shared, thread private) after threaded code generation.

In general, the use of scalar optimizations may negatively impact the quality of debug information. In order to improve the quality of available debugging information, it may be desirable to deselect some of the advanced optimizations. Examples of the impact of serial optimization on debugging include[xliv]:

- Sequential stepping through inlined source code may display different source locations and lead to confusion.

- Code motion (where the code is moved out of the loop) may lead to interspersed code from different locations.
- The use of base pointer by optimized x86 applications may lead to incorrect back tracking.

Commonly used Linux based application debuggers include **dbx** and **gdb**, which are thread aware.

5.4 Static Code Analysis

Static code analysis is performed on source code without executing the application or considering a specific input and without requiring any instrumentation of the code or development of test cases. Static code analysis exhaustively explores all execution paths, inclusive of all data ranges to ensure correctness properties such as absence of deadlock and livelock. Static code analysis tools cannot model absolute (wall-clock) time but can model relative time and temporal ordering. One method of implementing static code analysis uses a directed control flow graph developed from the program's syntax tree. The constraints associated with variables are assigned to the nodes of the tree. Nodes represent program points and the flow of control is represented by edges. Typical errors detected by using analysis based on the control flow graph include:

- Illegal number or type of arguments
- Non-terminating loops
- Inaccessible code
- Un-initialized variables and pointers
- Out of bounds array indices
- Illegal pointer access to a variable or structure

Static deadlock detection tools look for violations of reasonable programming practices, and approaches have been based on type systems, dataflow analysis and model checking. Type systems are syntactic frameworks for classifying phrases according to types of values computed. Type checkers evaluate well typed programs for deadlocks. These require annotations and they do not scale well to large systems.

In dataflow analysis call graph analysis is used to check for deadlock patterns. A static lock order graph is computed and cycles are reported as possible deadlocks. The analysis provides information such as type information, constant values, and nullness. Dataflow algorithms generally use information from conditional tests.

Model checking systematically checks all schedules and assumes the input program has finite and tractable state-space. Modeling software applications use the source code to detect potential errors. These models can be analyzed for behavioral characteristics. Due to the large number of possible interleavings in a multithreaded application, model checking is computationally expensive and does not scale well to large programs, thereby limiting its applicability[xlv] [xlvi] [xlvii].

The use of static code analysis tools helps maintain code quality. Their integration in the build process is recommended to help identify potential issues earlier in the development process. When using a static code analysis tool:

1. Use verbosity options if possible. Analyzing source code with many instances of multiple issues can be overwhelming. Research phasing in new checks as more critical issues are addressed.
2. Beware of false positives. Static analysis cannot guarantee 100% accuracy so confirm reported issues before enacting source code changes.
3. Static code analysis complements the development and debugging process, but it is not intended to be a replacement for a mature development and testing process - use it in conjunction with other debug techniques.

5.5 Dynamic Code Analysis

Dynamic code analysis is performed by executing programs built on physical hardware or a virtual processor. Issues can be detected much more precisely using code instrumentation and memory operation analysis. Dynamic code analysis tools are generally easy to automate with a low rate of false positives. For dynamic testing to be effective, the test input must be selected to exercise proper code coverage.

In the context of multithreading, the following recommendations should help in identifying thread-related bugs when using a dynamic thread analysis tool[xlviii]:

1. Benchmarks should exercise the application's threaded sections - tools cannot identify and analyze threading issues unless the threaded code is executed.
2. Use a code coverage tool to ensure the test suite runs through and the dynamic analysis tool adequately exercises key code regions.
3. Use a non-optimized debug version of the application that includes symbol and line number information.
4. Limit instrumentation added to the application. If threading is restricted to a specific area of the application, it may not be necessary to instrument the entire application.
5. If possible, test both optimized and non-optimized versions of the application. If certain bugs are only detected in the optimized version of the application then selected optimizations should be re-visited to ensure that none are contributing to the spurious bugs.
6. If binary instrumentation is used the binary should be relocatable.

Following are some tips for using dynamic code analysis tools:

- Limit working set size
- Start by analyzing frequently observed variables in the tool output.
- Beware of false positives

Examples of open source dynamic thread analysis tools based upon Valgrind include ThreadSanitizer, Helgrind, and DRD[xlix].

5.6 Active Testing

Active testing consists of two phases. In the first phase, static and/or dynamic code analyzers are used to identify concurrency related issues, such as atomicity violations, data races, and deadlock. This information is then provided as input to the scheduler to minimize false positives from the concurrency issues identified during the static and dynamic code analysis. One such tool that uses this approach is CalFuzzer[l].

5.7 Software Debug Process

Software debug of embedded multi-processing and multi-threaded applications is a difficult problem and demands a rigorous software development process that maximizes the detection of bugs with minimal effort. The following steps propose a phased debug process that evolves initially from a serial version of the application to one containing parallel tasks and execution on the targeted multicore processor:

1. Debug a serial version of the application.
2. Use defensive coding practices when parallelizing serial applications.
3. Debug a parallel version executing serially.
4. Debug a parallel version using an increasing number of parallel tasks.

5.7.1 Debug a Serial Version of the Application

In applications that take advantage of multicore processors, bugs can be divided into two categories: 1) general bugs unrelated to parallel processing; and 2) parallel processing bugs. Since debugging parallelized applications is inherently more difficult than debugging a serial application, start by debugging the application's serial version to find and fix all bugs unrelated to the parallel implementation. Traditional debug techniques should suffice to find these issues; detailing these techniques is out of this document's scope.

It may be more difficult to create a serial version for applications that will ultimately execute on heterogeneous multicore processors. In these cases, a library that emulates the execution in software can be used.

5.7.2 Use Defensive Coding Practices

The same well-known coding practices that minimize general bugs also help to minimize parallel processing bugs and should be used.[li] Three specific multicore development recommendations are 1) enable logging, 2) parameterize the parallelism, and 3) add synchronization points. Enabling logging involves adding source code that tracks the application's state and associates it with time and task. Parameterizing the application's parallelism involves adding code that allows the easy modification of the number of parallel tasks used in the application's execution. In general, this recommendation helps future attempts at scaling the performance of your application as well as the debug benefits mentioned here. Synchronization points are detailed later in this chapter.

5.7.3 Debug Parallel Version While Executing Serially

Once completing the initial parallel implementation, debug it while using minimal parallelism. This step relies upon the ability to easily parameterize the parallelism, as discussed in the previous step. The topic of serial consistency is discussed later in this chapter.

This step may not apply to applications which use threading for latency, such as the case when threading a GUI for responsiveness executing on a single core processor.

For applications that execute on heterogeneous multicore processors, it may not be possible to execute serially. Alternatively, limiting the parallel execution as much as possible can be effective. For example, limit the parallel execution to one task per type of processor core.

5.7.4 Debug Parallel Version Using an Increasing Number of Parallel Tasks

After debugging the serial execution of the parallel version, incremental increase the amount of parallelism. Again, this step assumes the ability to parameterize the parallelism. Debug a version that executes on a dual-core processor, then a quad-core processor, and so on.

5.8 Code Writing and Debugging Techniques

This section details several of the specific recommendations discussed in the proceeding section:

- Serial consistency
- Logging
- Synchronization Points
- Thread Verification
- Simulation
- Stress Testing

5.8.1 Serial Consistency

Enforce serial consistency to ease the transition from a serial version to a parallel version. Serial consistency is a property of code where the serial and parallel versions of the code are similar enough that they can be easily swapped in and out. The following example represents both a serial version and parallel version of the image filtering application (Figure 34). The serial and parallel versions of the program are respectively enabled through the following compilation commands:

- *gcc –DNUM_THREADS=1 app.c*
- *gcc –DNUM_THREADS=2 app.c*

In the parallel version (*NUM_THREADS=2*), the number of threads used is two, set by the command line definition of *NUM_THREADS* during compilation. The function, *FilterImage()*, only includes the code to divide the data space based upon the total number of threads.

```
char **image;
int size_x, size_y;

void *CreateThreads(void *arg)
{
 pthread_t threads[NUM_THREADS];
 int i = 0;
 for (i = 0; i < NUM_THREADS; ++i) {
  if (pthread_create(&(threads[i]), NULL, FilterImage,
  (void*)i)){
   ErrorMessage("pthread_create failed!\n");
  }
 }
 for (i = 0; i < NUM_THREADS; ++i) {
  if (pthread_join(threads[i], NULL)) {
   ErrorMessage("pthread_join failed!\n");
  }
 }
 CleanExit(0);
 return NULL;
}
void* FilterImage(void *id)
{
 int cpu_num = (int)id;
 int start_y, end_y;
 /* Assume size_y % NUM_THREADS = 0 */
 start_y = (size_y / NUM_THREADS) * cpu_num;
 end_y = (start_y + (size_y / NUM_THREADS)) - 1;
 /* Filter Image */

}
```

Figure 34. Serial consistency example.

5.8.2 Logging (Code Instrumentation for Meta Data Send and Receive)

A good technique for tracking the status and ordering of events in an application is the use of trace buffers in your code to create logs during execution time. The practice involves creation of a trace buffer which allows recording of an identifier, timestamp, and a message into a log that can be retrieved after execution and used to trace application execution in case of a problem. When using trace buffers, be careful where access to the trace buffer is placed in the application - synchronization controlling access to the trace buffer may mask concurrency bugs.

Figure 35 implements a log of size LOG_SIZE and lists the function, *event_record()*, which adds a log entry. Notice the mutex around the increment of the array increment, *id_count*. Time is obtained via a *system time()* function, which is assumed to be thread safe. Clock cycle information may be used if finer granularity is desired.

```
#define LOG_SIZE 1024
#define MSG_SIZE 64
typedef struct log_entry_s {
 unsigned int task_id;
 time_t timestamp;
 char message[MSG_SIZE];
} log_entry;
log_entry event_log[LOG_SIZE];

int id_count = -1;

int event_record(char *message) {
 time_t temp_time = time(NULL);
 pthread_mutex_lock (&mut);
 id_count++;
 id_count = id_count % LOG_SIZE;
 pthread_mutex_unlock (&mut);
 event_log[id_count].task_id = (unsigned int)pthread_self();
 event_log[id_count].timestamp = temp_time;
 strncpy(event_log[id_count].message, message, MSG_SIZE);
 event_log[id_count].message[MSG_SIZE-1] = '\0';
 return 1;
}
```

Figure 35. Logging example.

5.8.3 Synchronization Points

Synchronization points are specified locations in your application where a given task waits until other tasks have reached another specified location, exemplified in Figure 34. The thread executing *CreateThreads()* uses *pthread_join()* to wait on the other threads that are executing *FilterImage()*.

Synchronization points and logging can be used throughout your application to help the programmer understand the state of an executing application. Beware: use of synchronization points in program hotspots should be avoided because of potential performance impacts.

5.8.4 Dynamic Analysis Techniques Summary

Dynamic analysis tools use a dynamic analysis of an application as it executes. In the context of multithreading and finding data races, for example, the tool watches all memory accesses during threaded execution. By comparing the addresses accessed by different threads, and whether or not those accesses are protected by some form of synchronization, read-write and write-write conflicts can be found. Dynamic analysis will catch obvious errors of accessing the variables visible to multiple threads as well as memory locations accessed indirectly through pointers.

Typical dynamic analysis tools use instrumentation which can be inserted directly into the binary file (binary instrumentation) just before the analysis run or inserted at compilation time (source instrumentation). Further information on instrumentation and verification technology can be found in the paper by Banerjee et al.[lii]

5.8.5 Simulation Techniques Summary

A simulator is a software tool that mimics the final device hardware and can help find potential problems at two points during development. First, many embedded projects will need software developed in advance of hardware availability and thus use a simulator for the hardware platform. Early use of simulators can serve as a platform to test and debug your concurrent applications. Second, simulators are easily reconfigured and can be used to perform stress testing.

5.8.6 Stress Testing

As previously stated, concurrency bugs are nondeterministic. Therefore, an application that takes advantage of multicore processors may execute correctly on a dual-core processor, but then fail on a quad-core processor, or vice versa. Stress testing increases the chances of finding concurrency problems in your code and involves varying the number of threads or tasks used in executing the application. In addition, system parameters such as number of cores or latency of various system components can be varied.

Effective use of stress testing involves varying the above mentioned parameters and seeing if the application behaves incorrectly. This can help catch anomalies that may rarely occur on the actual hardware due to timing dependencies.

CHAPTER 6: PERFORMANCE

6.1 Performance

Multicore architectures have important sensitivity to the structure of codes. Parallel execution incurs many overheads that limit the expected execution time benefits. In this chapter we describe software application tuning practices to avoid or reduce the performance impact of the main bottlenecks. However, through the use of performance benchmarks we will discuss the bottlenecks that are associated with the implementation of the processor architecture as well as operating system dependencies. We will not be addressing parallel I/O, OS level (e.g. dynamic memory allocations, process scheduling), and low-level communications (e.g. DMA transfer efficiencies) optimizations since these are usually peculiar to each embedded system - there is typically considerable room for improvement at these levels. This chapter also does not explicitly address hybrid parallel programming that mixes two or more parallel programming models, although these should be considered to optimize performance.

As you will see, performance improvements with the use of a multicore processor depend on the software algorithms and their implementations. Ideally, parallel problems may realize speedup factors near the number of cores, or possibly even more if the problem is split up to fit within each core's cache(s), which avoids the use of the much slower main system memory. However, many applications cannot be greatly accelerated unless the application developer spends a large amount of effort to re-factor the entire application.

6.2 Amdahl's Law, Speedup, Efficiency, Scalability

Performance measurement of parallel systems is stated in terms of speedup, which is the ratio

$$S_P = \frac{T_1}{T_P}$$

where T_1 is the execution time using one core, T_P is the execution time using P cores, and the goal is to obtain a speedup of P. To characterize a parallel program's behavior, we usually refer to its efficiency (E_P):

$$E_P = \frac{T_1}{P * T_P}$$

A parallel program makes full use of the hardware when the efficiency is 1. It is said to be scalable when the efficiency remains close to 1 as the number of cores increases. Scalability is rarely achieved when increasing the number of cores without increasing the amount of work to perform.

Amdahl's law states that the sequential part of the execution limits the potential benefit from parallelization. The execution time T_P using P cores is given by:

$$T_P = seq * T_1 + (1 - seq) * \frac{T_1}{P}$$

where *seq* (in *[0,1]*) is a percentage of execution that is inherently sequential. For instance, if *seq=0.5* the potential speedup of a parallel execution is limited to 2. Usually, scalability is achieved by increasing the problem size because the ratio *seq* decreases with the growth of the problem to process.

6.3 Using Compiler Flags

Before considering parallelism to achieve higher performance, the sequential execution must first be efficient. The following sections will address sequential code tuning, and we'll start with a look at utilizing compiler flags.

Compilers implement many optimizations that must be activated using compilation flags, with the most common and straightforward ones being –O1, -O2, -O3, going from local to more extensive optimizations. These flags activate a set of optimizations that are also set or unset by more specific flags. The most aggressive optimization mode is usually not set by default since it can greatly increase compilation time, as well as debug effort.

More complex specific flags allow for inter-procedural optimizations, profile-guided optimizations, automatic vectorization or parallelization, or to help program analysis (no alias flags, etc…). Some flags deal with specific functional areas (floating point, function inlining, etc.). All these flags must be tested because their impact on performance may vary significantly from one application to another.

Here are a few global remarks:

1. When setting the debugging flag '-g', some compilers limit the extent of optimization performed to preserve some debugging capabilities. Alternatively, some compilers may reduce debugging information when increasing the optimization level.
2. Finding the right combination of optimization flags can be burdensome because it's a manual process.
3. The highest level of optimizations (e.g. -O3) does not always produce the fastest code because in this mode, the compiler tends to make tradeoffs between optimizations, sometimes making incorrect choices (especially due to the lack of runtime data).
4. Optimizations may change the program results (e.g. different floating-point rounding).
5. Compilers must produce correct code. When an optimization is not proven safe, it is discarded. After a slight rewriting (e.g. removing pointer aliasing), the compiler may be able to apply an optimization.

6.4 Serial Optimizations

Serial optimizations consist of rewriting the application code to help the compiler by highlighting instruction level parallelism and data level parallelism for vectorization as well as providing information on data structures. The first part of this section is devoted to aliasing information. Then we present a large class of restructuring techniques that mainly apply to loops. Finally, we address SIMD instructions.

6.4.1 Restrict Pointers

Many compiler optimizations are inhibited by spurious data dependencies due to potential memory aliasing between pointers that should not be used unnecessarily. When it is not possible to avoid these pointers, use restrict pointers (introduced in the ISO/IEC 9899:1999 standard) to declare that a pointer is the only one pointing to its memory region. This assists the compiler in its 'alias analysis'. Figure 36 provides an example showing that without the restricted pointer indication ('restrict' keyword) the compiler would have to assume that parameters dest and src can overlap (i.e. are aliased). As a consequence, many optimizations involving reordering of reads and writes will not be performed

```
/* File: restrict.c */

void f (int* restrict dest, int * restrict src, int n) {
 int i;
 for (i=0;i < n; i++){
  dest[i]=src[i];
 }
}
```

Figure 36. Code sample restricting pointers.

Here are a few global remarks:

1. C++ doesn't support restrict yet. Restriction on aliasing can also be indicated using compiler flags (e.g. 'fno-alias' or 'fargument-noalias'), but these options should be used with care since they may break your application software.
2. If aliasing exists and restricted pointers indicate otherwise, this may introduce difficulties in finding bugs (especially if the activation of debug reduces optimization and hides the bug).
3. Some compilers generate multiple code variants (one assuming aliasing and another without aliasing) in the absence of aliasing knowledge. This may have an impact on code size.

6.4.2 Loop Transformations

Loops are typically the hotspots of most applications and many loop transformations have been proposed to organize the sequence of computations and memory accesses to better fit the processor internal structure.

Figure 37 shows '*unroll-and-jam*', a powerful transformation to exhibit instruction level parallelism (ILP) as well as decrease the number of memory accesses. The outer loop is unrolled inside the inner loop: this exhibits a common sub-expression *v[j]* and increases the number of independent operations inside the loop body. Be aware that loop transformations cannot always be safely applied. For example, ensure that you preserve dependencies (see Chapter 3) that constrain execution ordering.

```
/* File: unrollandjam.c */

//original code
void fori (int dest[100][100], int src[100][100], int v[]){
 int i,j;
 for (i=0;i < 100; i++){
  for (j=0;j < 100;j++){
   dest[i][j] = src[i][j] * v[j];
  }
 }
}
// after unroll-and-jam
void ftrans (int dest[100][100], int src[100][100], int v[]){
 int i,j;
 for (i=0;i < 100; i = i+2){
  for (j=0;j < 100;j++){
   int t = v[j];
   dest[i+0][j] = src[i+0][j] * t;
   dest[i+1][j] = src[i+1][j] * t;
  }
 }
}
```

Figure 37. Sample code showing the unroll-and-jam loop transformation.

Here are a few global remarks:

1. Some transformations can be implemented using compiler-specific pragmas (e.g. *#pragma unroll(2)*).
2. Hand-checking the validity of loop transformations is necessary but not always trivial.
3. Performing loop transformations by hand may make the code target specific and thus difficult to read and maintain.

6.4.3 SIMD Instructions, Vectorization

SIMD instructions, such as SSE and AltiVec, provide a powerful way to improve performance because they process in parallel a vector of data compacted into registers. For example, a 128-bit register can be seen as a vector of two 64-bit words or a vector of four 32-bit words.

Methods to exploit such instructions may include writing inline assembly code, using a C intrinsic library, or letting the compiler automatic vectorization technology generate the SIMD instructions.

Figure 38 shows example code using the SSE intrinsic library ('m128d' are vectors of 2 double floating-point data). The SSE instructions are used via the intrinsic functions *mm_load_pd*, *mm_mul_pd*, *mm_add_pd*, and *mm_store_pd*. The SSE version produces two results per iteration instead of one.

```
/* File: simd.c */

//original code
void daxpy( unsigned int N, double alpha, double *X, double *Y ) {
 int i;
 for( i = 0 ; i < N ; i++ ) {
  Y[i] = alpha*Y[i] + X[i];
 }
}
//SSE code
void daxpySSE( unsigned int N, double alpha, double *X, double *Y) {
// m128d is two 64 bit float.
m128d alphav = _mm_set1_pd(alpha);
int i;
for( i = 0 ; i < N ; i+=2 ) {
  __m128d Yv   = _mm_load_pd(&Y[i]);
  __m128d Xv   = _mm_load_pd(&X[i]);
  __m128d mulv  = _mm_mul_pd(alphav, Xv);
  __m128d computv = _mm_add_pd(mulv, Yv);
  _mm_store_pd(&Y[i], computv);
 }
}
```

Figure 38. Example code using the SSE intrinsic library.

Here are a few global remarks:

1. On some processors, SIMD memory access instructions have data alignment requirements that can be specified using compiler pragmas.
2. In the simplest cases, the compiler automatically generates vector code that use SIMD instructions.
3. It's difficult to maintain hand-written assembly code. The use of intrinsics in high-level language is preferable.
4. Saturated arithmetic, predicated operations, and masked operations, when available, can be very useful to enhance performance.

6.5 Adapting Parallel Computation Granularity

As discussed in Chapter 3, parallel execution always incurs some overhead resulting from functions such as task start-up time, inter-task synchronization, data communications, hardware bookkeeping (e.g. memory consistency), software overhead (libraries, tools, runtime system, etc.), and task termination time.

As a general rule, small tasks (fine grain) are usually inefficient. In most implementations, there is a tradeoff between having enough tasks to keep all cores busy and having enough computation in each task to amortize the overhead. For example, Figure 39 shows the computation structure of 'Divide and Conquer' approaches for algorithms such as sorting, computational geometry, or FFT. The function *divideAndConquer* recursively spawns tasks (*parallel_combine*) and when reaching a small enough *basecase*, no new tasks are created. The granularity is tuned by fixing the size of the *basecase*.

```
/* File: granularity.c */

void divideAndConquer (int *p){
 pthread_t t1,t2;
 if (basecase(p)){
   p[3] = basesolve(p);
 } else {
  int p1[3],p2[3];
  get_part1(p1,p);
  pthread_create(&t1,NULL,(void *(*)(void *)) divideAndConquer,p1);
  get_part2(p2,p);
  pthread_create(&t2,NULL,(void *(*)(void *)) divideAndConquer,p2);
  pthread_join(t1,NULL);
  pthread_join(t2,NULL);
  p[3] = parallel_combine(p1,p2);
 }
}
```

Figure 39. Divide and Conquer approach.

When dealing with sequences of loop nests, many techniques can be used to tune the granularity. Conceptually, these techniques are easy to understand, but in practice it is complex to get the implementation details correct.

The following transformations can be used to increase task computation load:

- Transformations called 'loop fusion' (a.k.a. loop merging) is where consecutive loops with independent computation are gathered into a unique loop. This transformation decreases the loop overhead and allows for better instruction overlap. Although it can potentially increase cache misses due to conflicts and cause register spilling, fused loops using the same arrays can improve data locality.
- Loop tiling transformations (a.k.a. loop blocking) divide the iteration space of a loop nest in blocks that usually make better use of cache memories.
- Loop interchange transformations can be used to make the innermost parallel loop the outermost one, but should be carefully used in order to adversely affect data locality (see Section 6.11 Improving Data Locality).

The following transformations can be used to increase the number of tasks:

- With a loop distribution transformation (a.k.a. loop splitting**)**: The body of a loop is split to create numerous loops. This transformation is also used to separate serial computation from the parallel ones gathered in a single loop body.
- Loop coalescing are where nested loops are gathered into a single loop.

Here are a few global remarks:

1. Better data locality is usually achieved by using larger tasks.
2. Larger tasks can result in more load unbalancing.
3. When there are more threads/tasks than cores, many synchronizations (e.g. barrier) or global communication operations have degraded behavior.

6.6 Improving Load Balancing

Good load balancing is imperative to achieve scaling. When tasks have variable duration times (for example, because the amount of computations is input data dependent), static scheduling may result in load unbalance. A first step to balance the load may be to split the work into a number of tasks that is significantly larger than the number of cores. This can be followed by choosing a dynamic task scheduling strategy (e.g. work stealing scheduling) to allocate the work on the idle cores while the parallel computation continues.

Here are a few global remarks:

1. Achieving adequate load balancing may conflict with improving data locality, and may require a tradeoff.
2. With OpenMP, dynamic scheduling strategies are available under the “schedule” clause for parallel loops.
3. Dynamic task scheduling strategies usually incur more overhead and are less scalable than static scheduling strategies since they require some global synchronization.

6.7 Removing Synchronization Barriers

A synchronization barrier causes a thread to wait until the other threads have reached the barrier and are used to ensure that variables needed at a given execution point are ready to be used. Over-synchronizing can negatively impact performance and barriers should be placed to ensure that data dependencies are respected and/or where the execution frequency is the lowest.

Here are a few global remarks:

1. Removing synchronization barriers can introduce race conditions when done improperly.

2. A barrier can be used to replace creating and destroying threads multiple times when dealing with a sequence of tasks (i.e. replacing join and create threads).
3. Barriers are also used to flush memory transactions to ensure that the memory contents are up-to-date.
4. Some barriers are implicit in some programming APIs such as OpenMP.

6.8 Avoiding Locks/Semaphores

Typically, locks are used to prevent multiple threads' simultaneous access to some shared data or code sections (i.e. mutual exclusion). Locks are usually implemented using semaphores (aka mutexes). Locks are expensive and should be avoided as much as possible. There are two main strategies when using locks:

1. Fine granularity locking consists of protecting atomic data with a mutex in the low-level access functions (e.g. each record of a database). Using this approach, high-level functions are not concerned about locking, however, the number of mutexes can be very high thereby increasing the risk of deadlocks.
2. High-level locking is where data is organized in large areas (e.g. a database) protected by a single mutex. Although there is less risk of deadlock, the performance penalty can be large since accesses to the atomic data of the entire area are serialized.

An efficient tradeoff can be achieved by organizing a global data structure into buckets and using a separate lock for each bucket. With an efficient partitioning, the contention on locks can be significantly reduced. Another approach is to make each thread compute on private copies of a value and synchronize only to produce the global result.

Here are a few contexts where locks can be avoided:

1. Use of locks to implement global operations such as reductions.
2. Use of locks to mutex on shared data that can be privatized.
3. Spinning on shared variables to wait for an event may waste memory bandwidth.

Here are a few global remarks:

1. Trying locks in a non-blocking way before entering a mutex code section (and switching to a different job) can help avoid thread idle time.
2. Too many locks may be the cause of tricky deadlock situations.
3. Some architectures provide atomic memory read/write that can be used to replace locks.

6.9 Avoiding Atomic Sections

An atomic section is a set of consecutive statements that can only be run by one thread at a time, and they may be used to implement reductions. Atomic sections have similar issues as mutexes.

6.10 Optimizing Reductions

Global commutative and associative reductions (such as additions and multiplications) can be implemented in an efficient parallel scheme in replacement to the simple use of an atomic section (Figure 40).

```
/* File: reduction.c */

// ON ALL CORES/THREADS
void worker1(int *dp, int src1[], int src2[], int bg, int ed) {
 int i,j;
 for (i=bg;i < ed; i++){
  int t;
  t += src1[i] * src2[i];
// START ATOMIC SECTION
  pthread_mutex_lock(dp_mutex);
  *dp += t;
  pthread_mutex_unlock(dp_mutex);
// END ATOMIC SECTION
 }
}

//ALTERNATIVE IMPLEMENTATION

// ON ALL CORES/THREADS
void worker2 (int t[],int src1[], int src2[], int bg, int ed){
 int i,j;
 for (i=bg;i < ed; i++){
  t[MYCOREorTHREADNUMBER] += src1[i] * src2[i];
 }
}

// ON THE MAIN CORE/THREAD
. . .
  for (i=0; i < NBCORE; i++){
   dp = dp + t[i];
  }
. . .
```

Figure 40. Parallel reductions.

Here are a few global remarks:

1. Most parallel programming APIs provide efficient operations such as reduction and prefix.
2. Parallel reductions change rounding computations, which may generate non-reproducible results depending on the number of processors cores.

6.11 Improving Data Locality

This section describes the issues concerning caches and covers both coherent and non-coherent cases. Because the types of optimizations presented here may limit the application range or affect the trade-off of portability, the interface design and the operation strategy must be carefully considered. The reader should refer to Section 4.19 Affinity Scheduling, as this is also an important topic related to data locality and optimization of cache usage.

6.11.1 Cache Coherent Multiprocessor Systems

At a high level, multicore processors can be implemented with a shared memory model and/or a distributed memory model. The cache memories in shared memory systems can have copies of main memory contents and therefore, a cache coherence protocol must be used to keep every cache's behavior consistent, or *coherent*, as if the processors were directly attached to the main memory.

6.11.2 Improving Data Distribution and Alignment

Data distribution can be used to improve data alignment for better resource utilization and performance. To reduce communication cost, keep data as close as possible to the associated processor core. In other words, the data to be processed should be kept "local" to the processor core. This may involve the programmer's effort to implement data redistribution, data restructuring, and data prefetching.

In multicore processors, we can consider data distribution and alignment for both the shared and distributed memory models. In either case, the programmer should consider the strategy of data distribution in the main memory to reduce extra communication.

For example, assume an unaligned data structure comprised of 16 bytes. This data structure would require two transfers with a shared-memory processor whose coherence unit (cache line) is 16 bytes wide. This unaligned data structure would also causes extra transfers in heterogeneous multicore systems whose minimal DMA transfer granularity is 16 bytes, since most of these systems require that DMA transfers are aligned to a minimum size.

As an example, in codec applications, where image frames are processed in rectangular block units, transferring each block may involve more transfers than would be needed if the blocks are aligned. Sometimes programmers reorganize the entire image frame so that a rectangular block of the image can be transferred in a minimal number of transfers. This approach has a trade-off on performance because the reorganization requires extra overhead. Heterogeneous multicore systems that use a small scratchpad memory can use software techniques to reduce the communication cost.

Programmers should be aware of prefetching effects when designing for data distribution. Since the data is transferred to the processor in units larger than single word blocks, this could result in extra words being transferred at the same time. If the requesting processor uses the extra words before the words are changed by other processors or replaced by the requesting processor, additional data transfer is not needed; this is referred to as the "prefetching effect" (this also applies to single-core systems but the effect is more significant in multicore systems).

6.11.3 Avoiding False Sharing

In the situation described above, if the extra pre-fetched words are not needed and another processor in this cache-coherent, shared memory system must immediately change the words, this extra transfer has a negative impact on system performance and energy consumption. This symptom is called "false sharing".

In cache-coherent, shared memory systems, in order to reduce the occurrence of coherence operations and to exploit data locality (fetch once, reuse many times), cores share the memory in a coherence unit with specific size (typically anywhere from 32 to 256 bytes, depending on the processor and system architecture).

In a cache coherent system, which adopts one of the popular cache coherent protocols such as MSI and MESI[liii], only one core at a time can write to a specific cache line. The copies of a cache line cached by other cores in read-only modes ("Shared" state in MESI protocol) must be invalidated before being written, while a cache line can be shared among cores as long as all accesses to it are read.

False sharing means that coherence operations occur between two or more cores, for data not actually shared by the cores. For example, Core0 tries to write to a shared memory location *x* and Core1 tries to read from a shared memory location *y*, where both *x* and *y* addresses happen to be on the same cache line (coherence unit). When Core0 performs write operations to location *x* and Core1 tries to read from location *y*, and if the write operations and read operations are interleaved in time, the system can suffer from false sharing.

False sharing, and its performance-degrading effect, can be illustrated in three steps (Figure 41): 1) the cache line *A*, which contains both words *x* and *y*, is read by Core1 and the copy of *A* becomes read-only on Core0; 2) the cache line *A* is invalidated on Core1 when Core0 performs a write operation on cache line *A;* and 3) when the Core1 wants to read word *y*, a cache miss occurs in the closest level cache to Core1, and while the latest copy of cache line *A* is transferred to Core1, Core0's cache line *A*

is downgraded to read-only mode. If word *x* and word *y* had resided in different cache lines, then this could have avoided cache misses caused by false sharing.

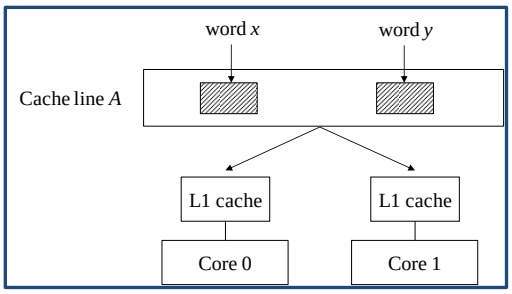


Figure 41. False sharing situation.

Since false sharing only occurs when one or more cores tries to write to the same cache line, care must be taken when the frequently written data locations reside on the same cache line with other less related data. Specifically, avoid collecting synchronization variables in one compact data structure. While this may forego readability, it will result in better performance.

A simple way to resolve false sharing is to pad the cache line so that frequently written data can reside on different cache lines. For example, suppose that in '*struct test { int x; int y; }*' that *x* is frequently written by Core0 (Figure 42). We can pad the structure to avoid false sharing at the cost of extra memory space.

```
struct test

int x;
unsigned char
padding[CACHE_LINE_SIZE -
sizeof(int)];
int y;
```

Figure 42. Cache-line padding example.

Figure 42 demonstrates a conceptual example and the actual system-dependent directives for alignment are not shown. Compilers may provide pragmas or library functions for memory alignment, such as *memalign()*, but the expression and behavior are system-dependent. To avoid performance degradation and waste of memory space, it is desirable to collect data accessed by a specific core (thread) into the same or neighboring cache lines.

The programmer should consider the effect of automatic prefetching, a feature supported by modern processor architectures, whereby neighboring cache lines are prefetched according to the program's access pattern. However, false sharing avoidance, while meeting other criteria, is not always an easy task.

6.11.4 Cache Blocking (or Data Tiling) Technique

In cache-coherent systems, the memory can be reorganized to maximize temporal and spatial locality. This technique, called ‘cache blocking’, improves program performance by attempting to decrease the cache-miss rates by increasing the reuse of data present in the cache. When the data size exceeds the cache size, loading new data evicts older data, thereby increasing the cache miss rates. Using cache blocking, loop nests are restructured such that the computation proceeds in contiguous chunks chosen to fit the cache.

In an example of cache blocking (Figure 43), the original loop computes the total sum of a 4-Mbyte *input_data* array 100 times, and the loop with cache blocking breaks the 4-Mbyte *input_data* into multiple chunks that fit within the 32KB cache block size. Due to the large size *input_data*, the cache hit rate is significantly improved.

Original Loop

```
#define TOTAL_SIZE 4194304 /* 4 MB */
#define LOOP_NUM 100
int i, j;
int sum = 0;
int input_data[TOTAL_SIZE] = { 0, 1, };
/* Before cache blocking */
for(j = 0; j < LOOP_NUM; j++)
{
   for(i = 0; i < TOTAL_SIZE; i++)
   {
       sum += input_data[i];
   }
}
```

Loop with Cache Blocking

```
#define TOTAL_SIZE 4194304 /* 4 MB */
#define LOOP_NUM 100
#define CACHE_BLOCK_SIZE 32768 /* CACHE_BLOCK_SIZE divides evenly into
TOTAL_SIZE */
#define BLOCK_NUM (TOTAL_DATA_SIZE/CACHE_BLOCK_SIZE) /* 128 */
int i, j, k;
int sum = 0;
int input_data[TOTAL_SIZE] = { 0, 1, };
/* Cache blocking */
for(k = 0; k < BLOCK_NUM; k++)
{
   for(j = 0; j < LOOP_NUM; j++)
   {
     for(i = k * CACHE_BLOCK_SIZE; i < ( k + 1 ) * CACHE_BLOCK_SIZE;
i++)
     {
     sum += input_data[i];
     }
   }
}
```

Figure 43. Cache blocking example.

6.11.5 Software Cache Emulation on Scratch-Pad Memories (SPM)

Most heterogeneous multicore systems do not support cache coherence on the hardware accelerator (i.e. DSP), but instead, typically have a scratchpad memory local to the accelerator core. This approach can reduce energy consumption and complex cache coherence circuits. The data transfer between the scratchpad memory and the system's main memory is typically handled by means such as DMA. The core waiting for the completion of the DMA transfer can suffer from degraded performance and should use latency hiding techniques such as double buffering (See 6.1 Performance). A compiler can also be used to analyze the access pattern and make the appropriate data mapping. These techniques are appropriate for regular data access patterns, such as sequential or stride accesses. For indirect references or irregular access patterns that cannot be predicted by simple calculation, these techniques may fail to reduce the latency.

Emulating a cache using software (henceforth referred to software cache[liv]) is a method to attempt to hide these communication latencies involved in irregular access patterns. To maximize spatial locality in irregular access patterns, the software cache maintains the already fetched copies of memory for reuse. In other words, software can be used to emulate a direct-mapped cache or n-way set associative cache which behaves logically similar to a hardware cache. However, unlike hardware caches which do not incur a penalty on a cache hit, the software cache must check if the requested cache line is kept in local storage (scratchpad memory) even with a cache hit, making it difficult to achieve performance comparable to hardware caches.

For example, assume a software cache invoked as:

X = SOFTWARE_CACHE_READ(SYSTEM_MEMORY_ADDRESS)

Even when there is no change in *SYSTEM_MEMORY_ADDRESS*, the cache hit check routine is called whenever executing this code fragment. For best performance, when developing an efficient cache-hit check mechanism or access-localization technique, treat any subsequent accesses after a cache hit as local memory accesses (scratchpad memory accesses). To achieve better performance based on access patterns, the cache algorithm should analyze the access pattern and prefetch the cache lines that will be used with high probability in the future.

As an example, consider the processing of a 2-dimensional image in a rectangular block unit. A software cache can be designed to exploit 2-dimensional spatial locality so that the cache hit check can be performed only once per rectangular block.

As implemented in software, the cache's structure (associativity, line size, indexing scheme) and replacement algorithms can be flexibly decided at compile- or run-time. Since these parameter types are closely related to performance, it is possible to use several software caches with different parameter sets within one application.

6.11.6 Scratch-Pad Memory (SPM) Mapping Techniques at Compile Time

Mapping techniques for architectures with SPM are very powerful, but require compile-time analyzable code. Most importantly, array data flow analysis (geometrical/polyhedral analysis) is applied to identify and explore the application's data reuse. Combined with task level parallelization, this can provide a viable mapping solution for multicore architectures with SPMs for statically analyzable applications. Such a solution can avoid false sharing by identifying the exact array locations used by each core and making local copies to its SPM.

6.12 Enhancing Thread Interactions

Because of insufficient shared resources (e.g. off-chip memory bandwidth[lv]) and the overhead of data synchronization, a parallelized application can perform worse when running on a multicore processor, as compared to a single-core processor. Although future multicore processors will have more available memory bandwidth, these systems can still suffer performance degradation because of the improper distribution of memory bandwidth to the cores. Techniques, generally known as 'Quality of Service' (QoS), that control the activity of bandwidth-consuming threads, can be considered so that higher priority threads can gain access to memory. However, most QoS proposals require hardware support and when not available, a programmer should consider using software to control memory access behavior. For example, with producer-consumer memory access patterns, a software-based memory access controlling mechanism is proposed M. Alvarez, A. Ramirez, A. Azevedo, C.H. Meenderinck, B.H.H. Juurlink, M. Valero[lvi]. This technique inspects the shared queue between a producer and consumers and limits the number of active cores for the consumer task in order to allocate more bandwidth to the producer which is a higher-priority task in terms of memory access.

If the shared queue length is too short, the producer may lack the memory bandwidth and the mechanism will reduce the number of cores for the consumer task. On the other hand, if the shared queue length is too long, the mechanism increases the number of cores for the consumer task to accelerate the consumer-side processing. Figure 44 shows the effect of applying a shared queue on a parallel H.264 decode running on a quad-core processor (the solid line represents the enhanced speedup delivered by this technique).

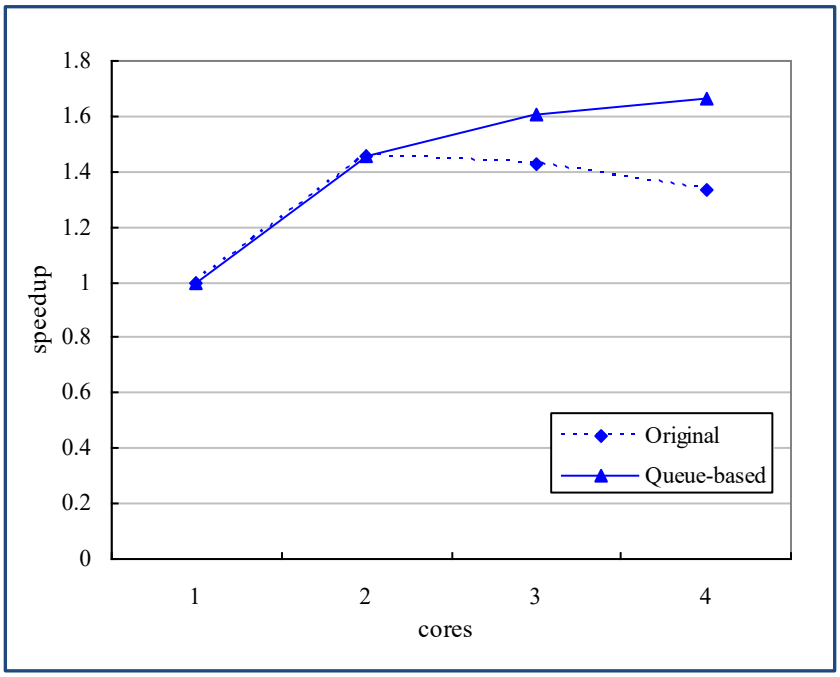


Figure 44. Performance comparison utilizing queue-based memory access.

Many applications benefit from using multi-threading; but when the performance of an application is limited by the contention for the shared data or bus bandwidth, additional threads do not improve performance. Instead, these threads only waste on-chip power. Thus, for power-efficient and high-performance execution, it is important to choose the right number (and distribution) of threads.

6.13 Reducing Communication Overhead

We have already discussed how communications (i.e. message passing) can severely slow down parallel programs, and should be avoided when possible, even at the cost of extra computing. Communications time, T_{com}, based on message passing can usually be expressed as:

$$T_{com}(n) = \alpha + \beta * n$$

where n is the size of the message, α the startup time due to the latency, and β the time for sending one data unit limited by the available bandwidth. Because of the non-trivial, start-up time, it is usually better to gather many small messages into larger ones when possible to increase the effective communications bandwidth.

Here are a few general remarks on communication:

1. Sending non-contiguous data is usually less efficient than sending contiguous data.
2. Do not use messages that are too large. Some APIs (e.g. MPI implementation) may switch protocols when dealing with very large messages.
3. In some cases, the layout of processes/threads on cores may affect performance due to the communication network latency and the routing strategy.

6.14 Overlapping Communication and Computation

Using asynchronous communication primitives or double buffering techniques to overlap communication and computation can increase performance (up to 2x). Synchronous communications (also referred to as blocking) make the sender wait until the communication has completed. On the other hand, asynchronous communication (non-blocking) primitives do not require the sender and receiver to "rendezvous". The data transfer starts as soon as the sender orders it, which can continue to work while the communication is taking place. This communication mode achieves interleaving of computation and communication but is expensive in terms of memory space because the data sent must potentially be stored on the receiver or sender side before being used.

Here are a few general remarks on message passing:

1. Message-passing techniques on shared memory systems are not always efficient compared with direct memory-to-memory transfers; this is due to the extra memory copies required to implement the sending and receiving operations. State-of-the-art implementations avoid memory copies for large messages by using zero-copy protocols.
2. When the receive operation is ready before the corresponding post, it enables the message passing API (e.g. MCAPI) to store received data directly in the destination, thus saving the cost of buffering the data.

Double buffering uses two memory buffers to mask communication latency. The computation is performed on one buffer while the other is filled by asynchronous communication with the next data. The buffers are swapped at each step.

6.15 Collective Operations

Point-to-point communication involves two tasks, a sender and a receiver. When dealing with global communication schemes, a point-to-point communication implementation may be very inefficient. Most communication APIs propose collective operations that involve a group of tasks (Figure 45). Optimizing communications may consist of using common collective operations such as scan, reductions, broadcast, scatter, or gather. Many collective operations have at least *log2(P)* steps (*P* is the number of cores), and as a consequence, they are expensive operations that should be avoided.

```
/* File: collectiveOperations.c */
//NAÏVE BROADCASTING OF DATA WITH POINT TO POINT MESSAGE
…
if (me == sender){
// start sending data to every other tasks
 for (i=0;i<nbtask;i++){
 if (i != me){
   send(data);
 }
 }
} else {
// receive data from sender
 for (i=0;i<nbtask;i++){
 if (i != sender){
   receive(data);
 }
 }
}

#include "mpi.h"
//BROADCASTING DATA WITH MPI
…
MPI_Bcast (&data, 1, MPI_INT, sender, MPI_COMM_WORLD);
```

Figure 45. Example code broadcasting data with MPI.

6.16 Using Parallel Libraries

This section provides a list of parallel libraries which can help reduce programmer development time, promote code portability and reuse, and improve code performance.

- Native thread libraries
  - Microsoft Win32 and COM Threads
  - POSIX threads
  - Various UNIX, Linux flavors, including Apple MacOS- Wind River VxWorks API, Enea OSE API
- Explicit control parallel APIs
  - MPI
  - MCAPI, MTAPI, TIPC, LINX
  - Apple Grand Central fit
- Managed runtime systems
  - Microsoft .NET Task Parallel Library
  - Java Concurrency – java.lang.Thread
  - Python thread library
- Libraries already threaded and/or thread-safe
  - Intel Math Kernel Library
  - Intel Integrated Performance Primitives
- Threading abstractions
  - Intel Threading Building Blocks
  - OpenMP
  - Microsoft Parallel Patterns Library & Concurrency Runtime
  - Boost thread library
  - Embedded Multicore Building Blocks (EMB^2)
- Esoteric
  - Haskell, Erlang, Linda, F#, Cg, HLSL, RapidMind
  - OZ/Mozart
  - CodePlay's C++ compilers
  - Scala
- Parallel Linear Algebra Libraries
  - ATLAS, BLACS, PBLAS, LAPACK, ScaLAPACK

6.17 Multicore Performance Analysis

At the high level, there are really two types of multicore performance analysis – homogeneous and heterogeneous. For the latter, you can synthetically analyze each of the individual compute elements separately. But to get the more realistic behavior, you would have to utilize more real-world applications

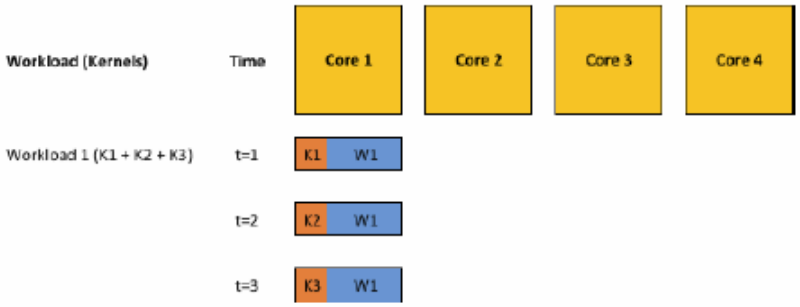
that would allow the heterogeneous compute elements to work cooperatively in parallel. For the former approach, you want to use benchmarks that can show scalability by ramping up the core utilization from single core to x cores (where x is the maximum number of cores in the processor). The industry organization, EEMBC, has produced several suites of standardized benchmarks to support this analysis (these are referred to as MultiBench, CoreMark-Pro, AutoBench 2.0, and FPMark), but they are functionally similar except for the specific code kernels that they contain (for simplicity, we'll just reference the MultiBench). This section will focus on the overall functionality of these benchmarks, supported by a series of results obtained.

The key element of MultiBench is the Multi-Instance Test Harness (MITH) that provides both a framework for coordinating scalability analysis, as well as an abstraction layer to facilitate porting to different platforms. Using the code kernels (such as angle-to-time computation, finite impulse response filtering, IP packet checking, image rotation, and video encoding), MITH provides a method to analyze compute scalability through a platform framework that can link together these individual kernels into workloads with varying degrees of complexity.

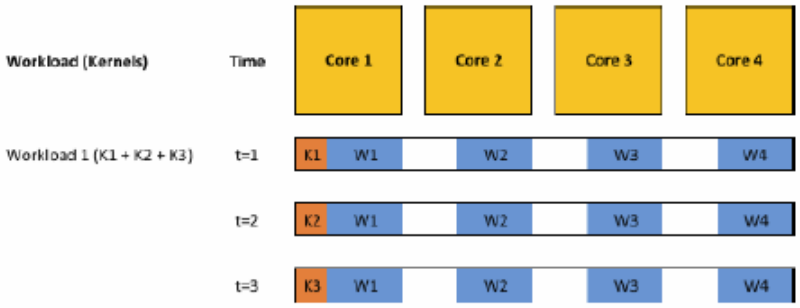
MITH supports two types of parallelism: worker and context. Worker parallelism is analogous to data decomposition. In MITH, kernels are referred to as work items. Parallel elements in a work item are referred to as workers. For example, the image rotation kernel is parallelized by data decomposition: splitting up the scan-lines across a specified number of workers. Context level parallelism allows the user to set the number of workload instances to run concurrently. For example, a context of 8 would cause MITH to launch 8 concurrent instances of the same workload, which may or may not be the optimal set up for an 8-core processor. Contexts are synonymous with task-level parallelism. If one context is used, the kernels inside the workload execute serially on one logical core. If the user chooses to run more than one context, the kernels will execute in parallel across the logical cores assigned by the O/S scheduler[lvii]. However, even if the contexts execute serially, the workers inside the benchmark may execute in parallel.

The two types of parallelism may also be combined. For example, on a four-core machine, it is possible to launch two image rotation kernel contexts, and then instruct MITH to use two workers per kernel. The following diagrams illustrate the relationship between contexts and workers on a four-core machine. (NOTE: The XCMD= statements above the diagrams refer to the corresponding Makefile configuration options for the benchmark run.) In the diagram, each workload is comprised of three kernels (or work items), K1, K2 and K3, and each Kernel may have one or more workers, [W1 … Wn].

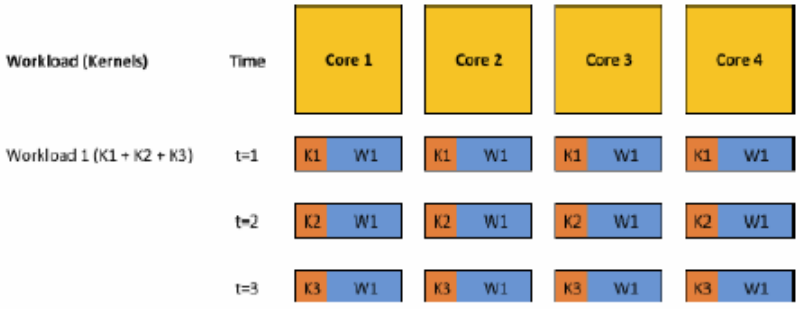
A. One context and one worker per context (`XCMD="-c1 -w1"`):



B. One context and four workers per context (`XCMD="-c1 -w4"`):



C. Four contexts and one worker per context (`XCMD="-c4 -w1"`):



D. Two contexts and two workers per context (`XCMD="-c2 -w2"`):

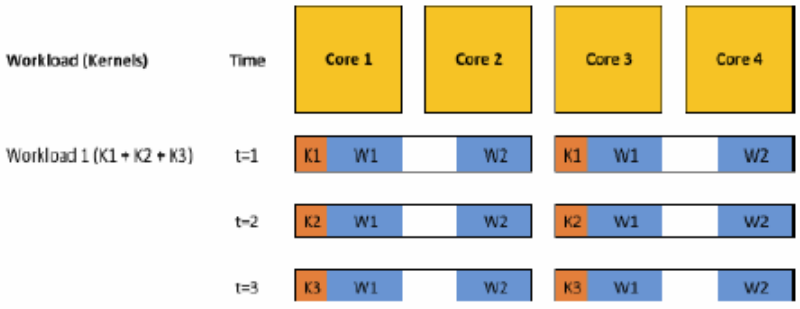


Figure 46. The relationship between contexts and workers on a four-core machine.

All MITH-based benchmark suites utilize two performance metrics for comparing configurations and platforms:

Throughput - defined in iterations per second, each platform executes a workload a specific number of times per second. The user adjusts the number of contexts to find the optimal throughput for each workload for the given hardware.

Performance Scaling - this unit-less value defines how well performance scales with more computing resources assigned to the workloads. This is done by comparing a workload running with a single worker to that same workload with multiple workers. The user adjusts the number of workers to find the optimal scaling for each workload.

Using MultiBench as an example, it consists of three separate marks: MultiMark, ParallelMark, and MixMark. Each of these marks are comprised of approximately two dozen workloads.

MultiMark

MultiMark consolidates the best throughput using workloads with only one work item, each of which uses only one worker. The calculated throughput factor is 10 times the geometric mean of the iterations per second achieved with the best configuration for each workload (Note: 10 is a multiplication factor). Each work item uses only one worker (-w1), and multiple copies of the task can be performed in parallel to take advantage of concurrent hardware resources (-cN). All workloads in this mark use a 4MByte dataset[lviii].

ParallelMark

This mark consolidates the best throughput of workloads with only one work item that each use multiple workers. The calculated throughput factor is 10 times the geometric mean of the iterations per second achieved with the best configuration for each workload (Note: 10 is a multiplication factor). Only one work item may be executed at a time, and multiple workers may be used to take advantage of concurrent hardware resources (-wN). All workloads in this mark use a 4MB dataset.

MixMark

MixMark is perhaps the most telling mark, it consolidates the best throughput of workloads with multiple different work items. These workloads are closest to workloads run on actual systems. The calculated throughput factor is 10 times the geometric mean of the iterations per second achieved with the best configuration for each workload (Note: 10 is a multiplication factor). All workloads in this mark use a 4MB dataset.

Sample scores on a simulated platform with 16 cores (Figure 47) show some interesting information even without diving into the details of specific workloads. For example, consider the fact that the MultiMark Scaling Factor is 8.9 rather than a number closer to 16, which one might hope for with 16

cores. This factor strongly hints that the system can use only about half of the computing resources available to it on any particular problem.

	Throughput Factor	Scale Factor
MultiMark	10.9	8.9
ParallelMark	10.5	4.7
MixMark	4.5	8.8

Figure 47. Sample scores on a simulated 16-core platform.

This limitation is likely related to memory bottlenecks, as single-worker workloads have very little synchronization and thus are less dependent on the synchronization efficiency of the platform and operating system. A more detailed examination of the individual results may be needed to yield more answers and highlight the trends better than a single-number representation can.

Figure 48 compares the results from two dual-core platforms. Note the significant difference in the Performance Factors between these two platforms. While both platforms use Linux and GCC and both have the same core frequency, they are based on different processor architectures with different memory hierarchies. Platform #1 has a shared L2 cache while Platform #2 has separate L2 caches.

	Platform #1		Platform #2	
	Performance Factor	Scale Factor	Performance Factor	Scale Factor
MultiMark	33.3	1.9	24.7	1.8
ParallelMark	24.8	1.9	13.8	1.7
MixMark	11.8	1.7	7.2	1.5

Figure 48. Sample scores from two dual-core platforms running at 2 GHz

Basic analysis of the results show that platform #1 is 30% faster on MultiMark and 80% faster on ParallelMark. Insights into why the performance is so different don't necessarily require sophisticated performance analysis tools. Often, simply looking deeper into the specific scores of different workloads and correlating those with architectural differences can be enough to figure out where the differences come from. For example, synchronization overhead and cache coherency traffic both result in the significantly lower ParallelMark for Platform #2.

Additional Throughput Results With Scaling

The combined effect of using all these workloads provides a comprehensive view of multicore processor behavior. For this purpose, the following will demonstrate the combined throughput using the AutoBench 2.0. The chart below provides some of the basic parameters of two processors, neither of which are automotive processors by any stretch of the imagination, but they are interesting to analyze because of the relatively high number of cores).

	NXP QorIQ T2040	Intel Xeon CPU E5-2676 v3
Operating Frequency	1800 MHz	2400 MHz
Number of cores	4 (8 virtual)	12 (24 virtual)
L1 (kbytes)	32/32	32/32
L2	1Mbyte	12x256kbytes

The first set of results (Figure 49) shows the number of iterations/second for the overall AutoBench 2.0 (taking the geometric mean of all workloads in this suite).

Where in Cx-4y, X = number of contexts launched (1, 2, 4, or 8); controls core utilization; Y = size of dataset per context (4k or 4M). The 'P' suffix indicates the perfect scaling result from a single core. The goal here is not to compare these two processors, but to demonstrate how much different architectures can scale. In Figure 50 with the 4Mbyte data sizes, the overall performance is significantly reduced as expected, as is the overall scaling as the number of contexts launched reaches 4 or greater.

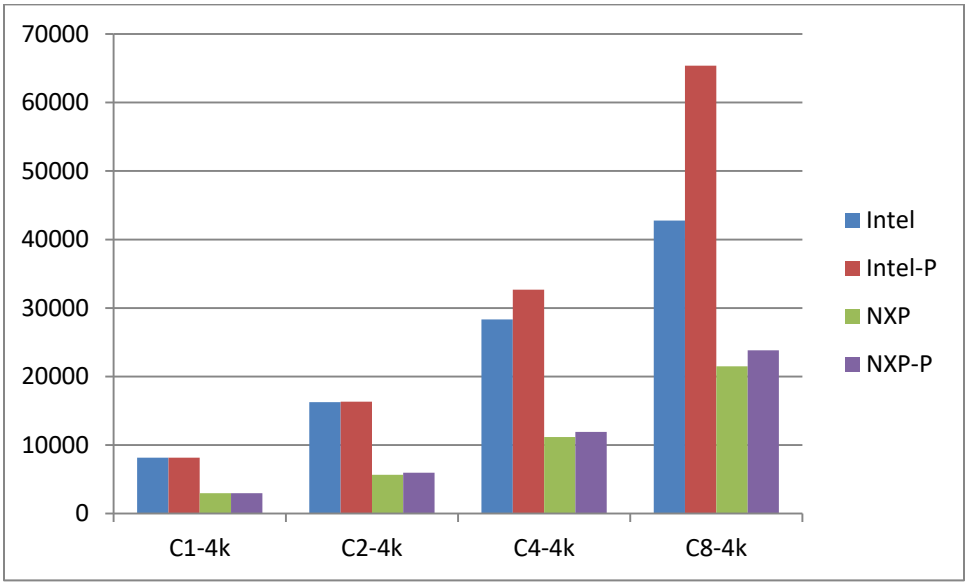


Figure 49. Scaling with 4kbyte data sizes.

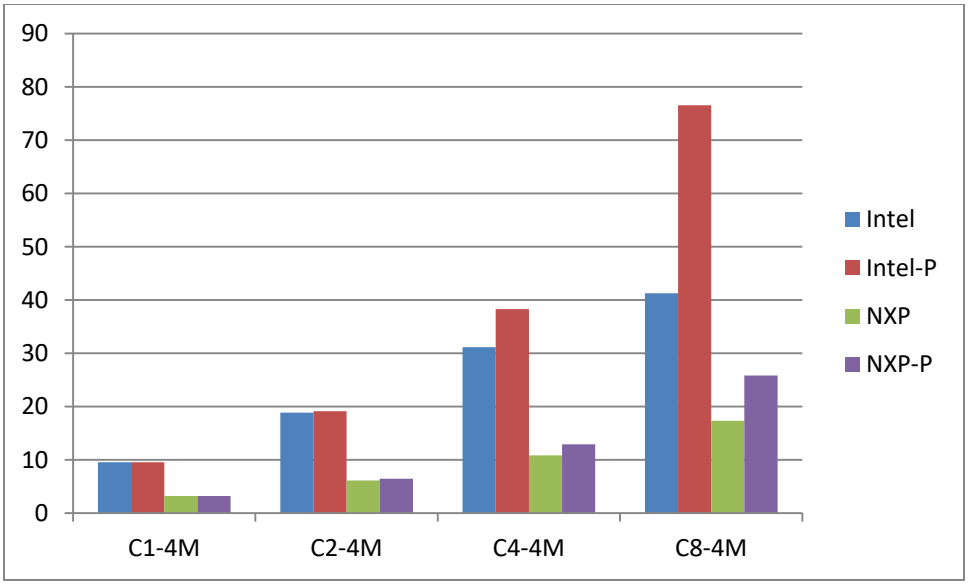


Figure 50. Scaling with 4Mbyte data sizes.

CHAPTER 7: FUNDAMENTAL DEFINITIONS

7.1 Fundamental Definitions Introduction

The diverse communities involved in multicore programming present a challenge when communicating guidelines and practices, as each community generally has its own set of vocabulary and definitions. Unfortunately, most communities use the same vocabulary but with few overlaps in definitions. To avoid unnecessary confusion, this section presents the most commonly reused terms and their definitions. All terms are presented alphabetically in this section, which also presents additional terms that may be unfamiliar but are less likely to suffer from multiple definitions.

7.2 Fundamental Multicore Definitions (Hardware)

These definitions refer only to the static hardware that makes up a system and do not imply anything about the software or runtime configuration of the system.

Accelerator: integrated circuit block implementing some specific functionality. Generally, an accelerator is designed to run faster than a software implementation of the same functionality run on a processor core. Accelerators may be programmable or configurable but do not implement a general purpose ISA.

Asymmetric Homogeneous Multi-*: homogeneous multi-* with more than one micro-architecture implemented in different processor cores that use the same ISA.

Coarse-grain Multi-threading (CMT): processor core with more than one context that share resources but only one context can execute code at a time. Execution of the multiple contexts is time sliced by cycle, memory access, or some other mechanism. An example of a CMT processor is the Sun UltraSPARC T2 (“Niagara 2”) processor. The C in CMT may also stand for Cooperative or Chip without change in definition.

Context: program-counter and set of registers within a processor core capable of executing a program. Contexts are frequently referred to as threads or strands in hardware documentation and can be thought of as a “logical” processor. Contexts may be defined to only support some levels in the privilege hierarchy (user-level to kernel-level to hypervisor-level).

Heterogeneous Multi-*: multi-* implementing more than one ISA.

Homogeneous Multi-*: multi-* where all processor cores implement the same ISA. This does not imply that the software is AMP, SMP, or a combination thereof, only that that hardware is homogeneous.

Many-Core Processor: multicore processor with a large number of cores (typically greater than 16).

Multi-*: multicore processor, multicore system, or multi-processor system. Multi-* systems are often referred to as Parallel Systems and their use as Parallel Computing.

Multicore Processor: computer chip comprised of two or more processor cores. Also referred to in hardware literature as a Chip Multi-Processor (CMP).

Multicore System: computer with one or more multicore processors.

Multi-Processor System: computer with more than one processor core available to the end user.

N-way System or **N-core System:** computer with N contexts consisting of either more than one processor or one processor with more than one context or core.

NxM-way System or **NxM-core System:** computer with N processors and M contexts or cores per processor. The next logical step of an NxMxL-way system with N processors and M cores per processor and L contexts per core is not used.

Processor (CPU): one or more processor cores implemented on one or more chips, in a single package from the end user perspective.

Processor Core: integrated circuit block implementing an instruction set (ISA), producing a unit that is user-programmable.

Simultaneous Multi-threading (SMT): processor core with more than one context that share resources within the processor core where multiple contexts can execute code at the same time. A single-core SMT processor and a multicore processor differ in that the SMT contexts share resources within the processor core while the multicore contexts do not. Either may share resources outside of the processor core. An example of a SMT processor is the Intel Pentium with Hyperthreading. An example of a Multicore SMT processor, meaning a processor where there are multiple processor cores each of which is individually SMT, are the Intel Core i7 based processors.

Symmetric Homogeneous Multi-*: homogeneous multi-* where all the processor cores have the same micro-architecture (and the same ISA).

System on a Chip (SoC): single chip containing at least one processor and at least one non-processor device of relevance to the system design, such as computation accelerators, memory controllers, timers, or similar. In general, SoC tends to refer to highly integrated chips that can indeed work as a complete or almost complete system in their own right (adding external memory should be sufficient).

7.3 Fundamental Multicore Definitions (Configuration)

These definitions refer only to the dynamic configuration of a system and do not necessarily imply anything about the hardware making up the system or about software running on the system.

Local Memory: physical storage that is only accessible from a subset of a system. An example is the memory associated with each SPE in the Cell BE processor from Sony/Toshiba/IBM. It is generally

necessary to state what portion of the system the memory is local to as memory may be local to a processor core, a processor, an SoC, an accelerator, or a board.

Shared Memory: physical storage that is accessible from multiple parts of a system. As with local memory, it is generally necessary to state what system portions share the memory.

7.4 Fundamental Multicore Definitions (Software)

These definitions refer only to the software that runs on a system and may not imply anything about the hardware or runtime configuration of the system.

Application: program or programs that combine to compute a value or provide a service.

Asymmetric Multiprocessing (AMP): MP or MP and MT application running on a heterogeneous multi-* or a homogeneous multi-* where not all processor cores have a similar view of the system or share the same main memory.

Concurrent Tasks: two or more tasks operating at the same time. No relation among the tasks other than that they execute at the same time is implied.

Multi-threaded (MT) Application: application with at least one process containing more than one thread.

Multiprocessing (MP) Application: application consisting of more than one process, each with its own mapping from virtual to physical addresses. The processes may use an operating system to dynamically share one or more contexts or may be statically mapped to separate contexts.

Parallel Tasks: concurrent tasks from the same application.

Process: program running on one or more contexts with a specific mapping between virtual and physical addresses. Conceptually, a process contains an instruction pointer, register values, a stack, and a region of reserved memory that stays active throughout the life of a process.

Program: compiled code that can run on a processor.

Symmetric Multiprocessing (SMP): MP or MT application running on a homogeneous multi-* where all processor cores have a similar view of the system and share the same main memory.

Task: unit of work with in a process. Depending on coding style, a task may be performed by a process, a thread, or the use of an accelerator.

Thread: part of a process running on exactly one context.

APPENDIX A: MPP ARCHITECTURE OPTIONS

The appendices provide additional summary information, focusing more on open source or non-commercial solutions.

Assumptions:

- An open standard and/or implementation is available.
- Implementations are available for more than one platform or operating system.

A.1 Homogeneous Multicore Processor with Shared Memory

- Widely used architecture, with support for a number of APIs.
- Uses same processor cores with the same instruction sets and same data structures; memory is used for inter-process communication.
- Shared memory is typically accessed through a bus, which generally uses a locking mechanism to control simultaneous access to a memory location by multiple cores.
- Shared memory facilitates inter-core communication with minimal overhead, and allows passing data by reference.
- Shared memory may become a bottleneck as the number of cores simultaneously accessing the shared memory increases. This bottleneck makes this memory architecture less scalable, compared to non-shared memory architectures.
- Thread scheduling is easier on simultaneous multithreaded homogeneous core platforms with shared memory.
- Shared memory systems are suited for array-based multithreaded applications which typically share data heavily, which generally makes thread scheduling easier and efficient on nearby CPUs with common caches.
- Homogeneous Multicores are more suited for dynamic task mapping and data parallelism.
- Shared Memory Parallel nodes have several well-established parallel programming models, such as Posix Threads (Pthreads) and OpenMP.
- Until recently, this model was not as common across embedded devices.

A.2 Heterogeneous multicore processor with a mix of shared and non-shared memory

- Heterogeneous architectures with a mixed memory model have until recently, been common in embedded systems.
- Shared memory is used in addition to message exchange by cores for synchronization.
- These architectures may require a higher learning curve, since programming on such systems may be platform specific.
- In general, one processor is primarily used to handle serial and administrative tasks, the other cores are generally of the same type, with a path to the serial processor, and their own memory network. These systems have historically evolved from multimedia computing, with each having its own programming environment.

A.3 Homogeneous Multicore Processor with Non-shared Memory

- This architecture is well suited for data parallel applications due to the relative independence of threads which generally share only a small amount of data at certain communication points.
- This model is common in clusters for high performance computing.

A.4 Heterogeneous Multicore Processor with Non-shared Memory

- Until recently, this architecture has been more common in embedded systems.
- These are generally specialized application-specific processors, with each processor having a different instruction set and data structures.
- Static task mapping is generally used on these systems.
- Message passing is used between the cores.

A.5 Heterogeneous Multicore Processor with Shared Memory

- The choice and use of this architecture varies between different industry segments and target applications.
- Heterogeneity of cores may add the complexity of having different instruction sets and data structures.

APPENDIX B: PROGRAMMING API OPTIONS

Assumptions:

- An open standard and/or implementation is available.
- Implementations are available for more than one platform or operating system.

B.1 Shared Memory, Threads-based Programming

B.1.1 Pthreads (POSIX Threads)

- A standard for threads which defines an API for creating, manipulating, and managing threads, and for synchronizing between threads using mutexes and signals.
- Defined as a set of C language programming types and procedure calls. It is implemented with a *pthread.h* header/include file and a thread library. Special compiler support is not required.
- Most commonly used on Unix-like POSIX systems such as Linux and Solaris. A subset of Pthreads APIs is also available for Microsoft Windows (pthreads-w32).

B.1.2 GNU Pth (GNU Portable Threads)

- A POSIX/ANSI-C based library for Unix platforms.
- It provides non-preemptive, priority-based scheduling for multithreading inside event-driven applications. All threads are run in server application address space. Each thread has its own program counter, run-time stack, signal mask, and errno (error number) variable. It also provides optional Pthreads API.
- Threads are managed by a priority and event based non-preemptive scheduler. It uses an M:1 mapping to kernel space threads. Scheduling is done by the GNU Pth library and the kernel is not aware of the M threads in user space. This precludes utilization of SMP, as kernel dispatching would be necessary.
- GNU Pth is not very widely used.

B.1.3 OpenMP (Open Multiprocessing)

- An API which supports multi-platform, shared memory, multiprocessing programming in C, C++, and Fortran on multiple architectures.
- Consists of a set of compiler directives, library routines, and run time environment variables.
- Uses fork-join parallelism, whereby master thread spawns a team of threads as needed. Uses synchronization to impose order and protect shared data access.
- Global variables are shared among threads. Stack variables in functions/subprograms called from parallel regions, and automatic variables in a statement block are private.
- Use of OpenMP requires an OpenMP compatible compiler and thread-safe library routines.
- Using OpenMP requires the use of an OpenMP aware debugger which can access information related to OpenMP constructs and types.

B.1.4 Threading Building Blocks (TBB)

- Also known as Intel Threading Building Blocks
- A portable C++ template library, with higher level, task-based parallelism which abstracts platform details and threading mechanisms for performance and scalability.
- Uses templates, relying on compile-time polymorphism.
- Implements task-stealing to balance parallel workloads across processing cores.
- Consists of data structures and algorithms for accessing multiple processors by treating operations as tasks using dynamic allocation to cores with the use of a run-time engine.

B.1.5 Protothreads (PT)

- Extremely lightweight, stackless threads designed for memory-constrained systems. Provides linear code execution for event-driven systems implemented in C.
- Can be used with or without an underlying OS for blocking event handlers. Provides sequential control flow without complex state machines or full multi-threading
- There is no thread-local data in Protothreads and scheduling is done manually. Threads are scheduled by calling the *thread* function with appropriate thread state.

-

B.1.5 Embedded Multicore Building Blocks (EMB²)

- Open source C/C++ library for implementing compute-intensive applications
- Designed for embedded systems (key requirements: predictable memory consumption, support for timing-critical applications)

- Based on the Multicore Task Management API (MTAPI)
- Provides scheduler for fine-grained tasks, generic parallel algorithms, templates for stream processing, and concurrent containers

Supports heterogeneous systems consisting of different kinds of compute units (e.g., CPU, GPU, DSP, FPGA).

B.2 Distributed Memory, Message-Passing Programming

B.2.1 Multicore Communications API (MCAPI)

- A message-passing standardized API for communication and synchronization between closely distributed cores and/or processors in closely distributed embedded systems. Target systems will span multiple dimensions of heterogeneity.

B.2.2 Message Passing Interface (MPI)

- A widely available library specification for message-passing, designed for high performance on massively parallel machines and workstation clusters. Supports point-to-point and collective communication, and provides a message-passing API, with protocol/semantic specifications of feature functionality in an implementation.
- Provides essential virtual topology, synchronization, and inter-process communication in a language-independent manner, with language-specific bindings, and some language-specific features.
- Running the same program on each node requires explicit control logic.

B.2.3 Web 2.0

- Web 2.0 infrastructure includes server software, content syndication, messaging protocols, standards-oriented browsers with plugins /extensions and various client applications.
- Allows users to build on Web 1.0 iterative facilities, to provide “network as platform”, enabling software applications’ execution through a browser.

B.3 Platform-specific Programming

B.3.1 Open Computing Language (OpenCL)

- C-based framework for programs executing on heterogeneous platforms. Includes a language for writing kernels and APIs for defining/controlling a heterogeneous platform. It defines hardware and numerical precision requirements.
- Uses task-based and data-based parallelism and implements a relaxed consistency, shared memory model. It provides distinct address spaces which can be collapsed.
- The standard is expected to continue to mature over time.

APPENDIX C: PARALLEL PROGRAMMING DEVELOPMENT LIFECYCLE

C. 1 Introduction to Parallel Tool Categories

A number of tools can be used to assist programmers in migrating existing sequential applications to multicore platforms. Figure 51 provides a high level view of the workflow between the various categories of tools[lix]. Performance bottlenecks which impact both serial and parallel applications should be removed prior to parallelizing an application. Appendix B contains additional information about threading APIs. A brief introduction to the tool categories identified in Figure 51 is provided below.

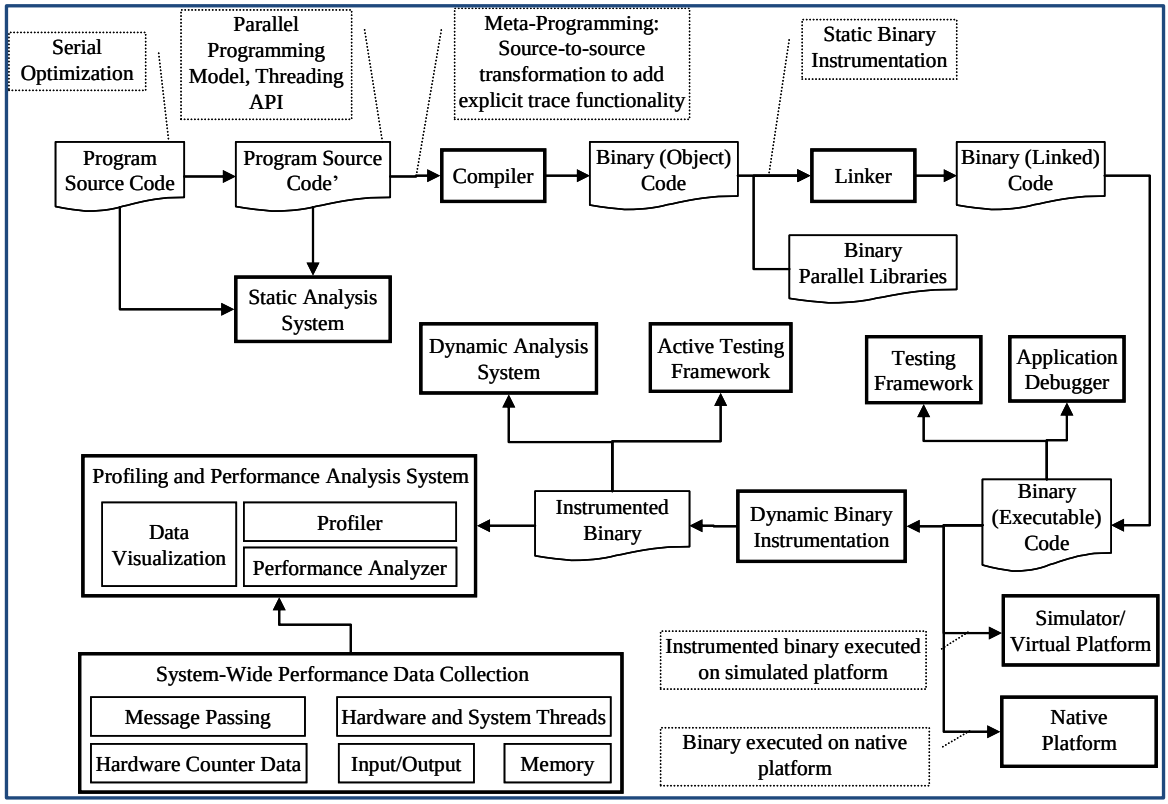


Figure 51. Workflow between the various tool categories.

C.1.1 Compilers

A number of compilers are available for compiling applications, depending upon the choice of the threading API. As indicated in Figure 51, the compiler may modify code for generating trace information, using either source-to-source, static binary, or dynamic instrumentation. Source-to-source instrumentation modifies source code prior to pre-processing and compilation. Static binary instrumentation modifies compiled binary code prior to execution.

C.1.2 Static Code Analyzers

Static code analyzers, used to detect issues which may be missed during functional testing, analyze source code without executing the application. The analyzers model the software applications using the source code and exhaustively explore all execution paths. Typical errors detected include illegal number or type of arguments, non-terminating loops, inaccessible code, un-initialized variables and pointers, out-of-bound array indices, and illegal access to a variable or structure. Integration of static

code analyzers in the build process is recommended to help identify issues early in the development process.

C.1.3 Debuggers

Complexities resulting from non-deterministic thread scheduling and pre-emption, and dependencies between control flow and data flow, can make debugging multithreaded applications more complicated. In addition, issues caused by thread interactions, such as deadlocks and race conditions, may also be masked by the use of debuggers.

Debuggers for concurrent systems may use a number of approaches, including traditional debugging and event-based debugging. In traditional debugging (or break-point debugging), one sequential debugger is used per parallel process, which limits the information that can be made available by these debuggers. Event-based (or monitoring debuggers) provide better replay functionality for multi-threaded applications, but may have more overhead. The use of a given threading API may impact the choice of debuggers.

C.1.4 Dynamic Binary Instrumentation

Dynamic binary instrumentation (DBI) is used to gain insight in the runtime behavior of a binary application. Instrumentation code is injected which executes as part of the application instruction stream. DBI only explores executed code paths. Two common approaches are light-weight DBI and heavy-weight DBI. Architecture specific instruction stream and state are used in light-weight DBIs, whereas, an abstraction of instruction stream and state is used by heavy-weight DBIs. Compared to light-weight DBIs, heavy-weight DBIs are more portable across architectures.

C.1.5 Dynamic Program Analysis

Dynamic program analyzers check program properties at runtime, which helps identify problem sources much faster than extensive stress testing. Dynamic analysis may be done using either hardware or a virtual processor. These tools are generally easy to automate and have a low false positive rate. For dynamic analysis to be effective, selected test input should be used to exercise good code coverage.

C.1.6 Active Testing

Active testing uses both static and dynamic code analyzers to identify concurrency related issues (e.g. atomicity violations, data races, and deadlocks). The output from static and dynamic code analyzers is provided as input to the scheduler to minimize false positives from concurrency related issues identified during analysis.

C.1.7 Profiling and Performance Analysis

Profilers are used to facilitate efficient utilization of system resources and to optimize program decomposition by inspecting the behavior of running programs, and to help identify issues which may impact performance and execution. Either active or passive profiling approaches can be used. In active profiling, execution behavior may be recorded using callbacks to the trace collection engine. Passive profiling uses external entities, such as a probe or a modified runtime environment, to inspect control flow and execution state. In general, modification of the measured system is not required for passive profiling.

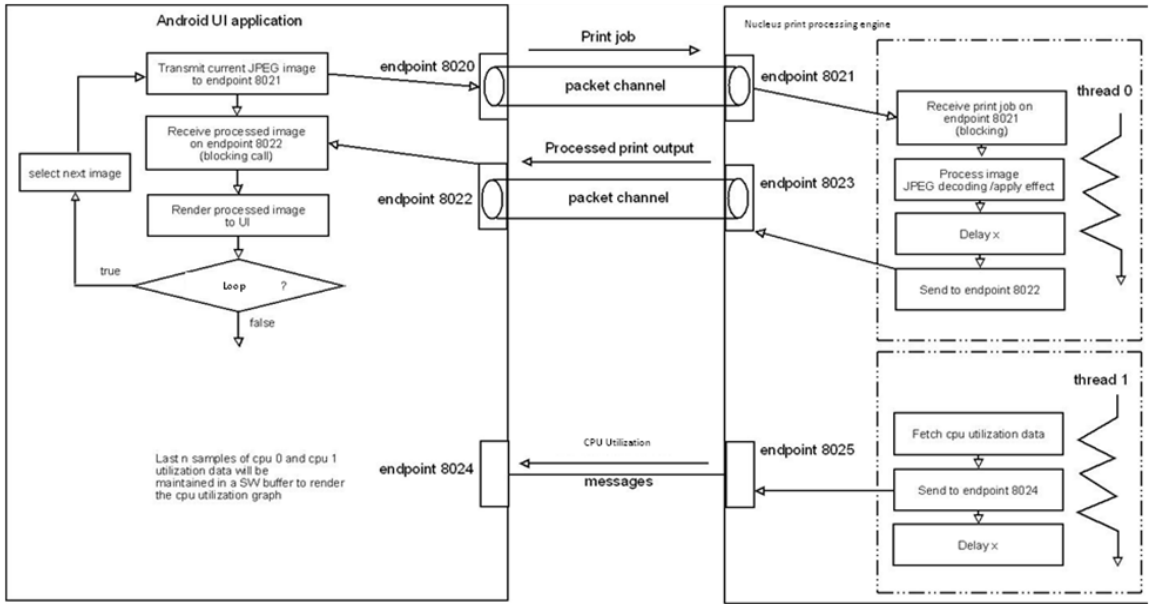
Profilers can provide behavioral data for the executed control paths. Code coverage may be improved by using multiple application runs with carefully selected input data and artificial fault injection.

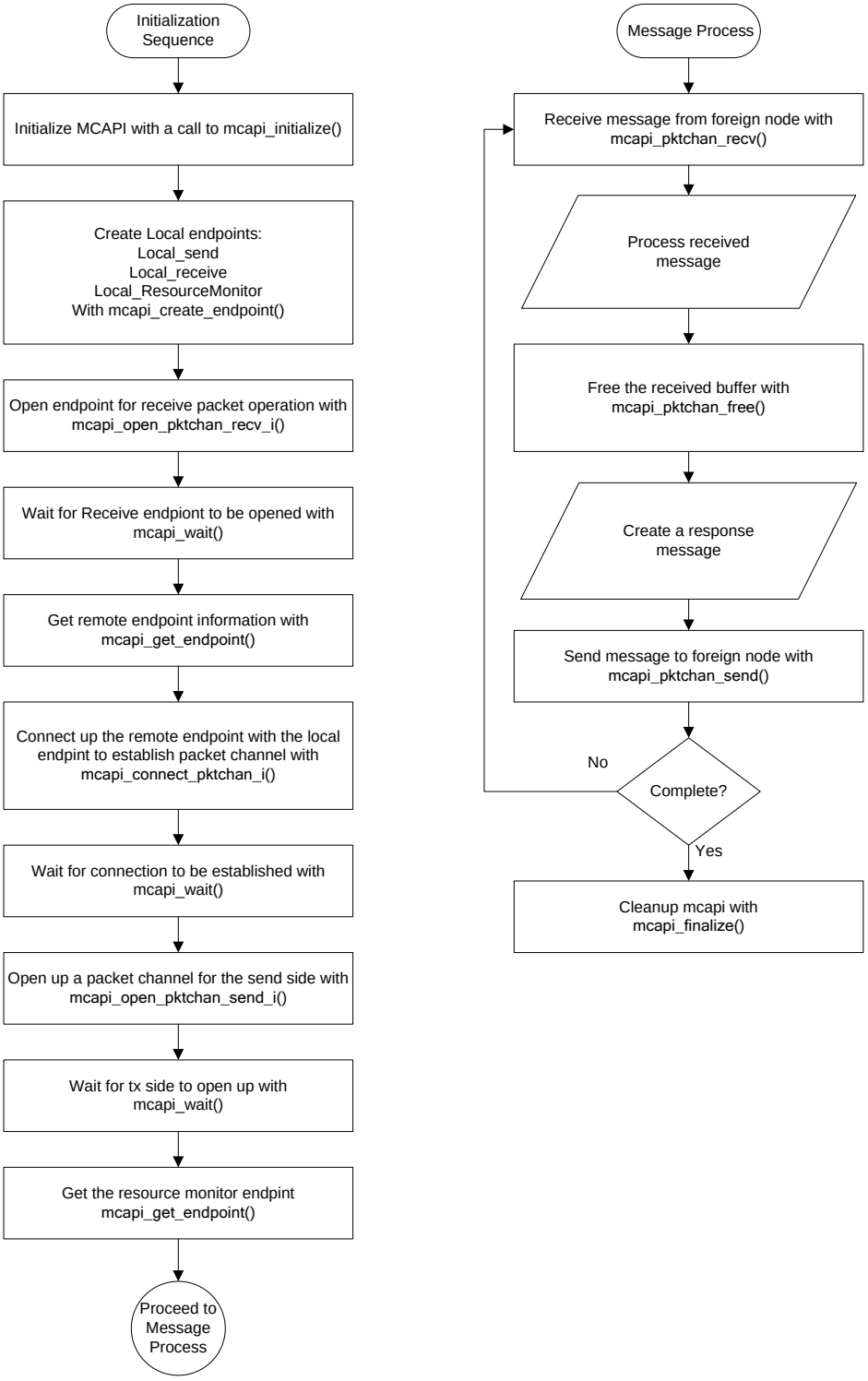
When selecting a profiler, the programmer must understand if the profiler requires specific hardware support, whether the profiler does system-wide profiling, or if the focus is only on user applications. In addition, the profilers may be designed to analyze utilization of select system resources, such as thread profiling, call stack sampling, memory profiling, cache profiling, and heap profiling.

C.1.8 System-wide Performance Data Collection

In order to best utilize system resources, programmers must be able to access system-wide resource usage data and relate this information to application performance. Standardized APIs can help facilitate access to low level system data by performance analyzers and profilers.

APPENDIX D


Android UI application
Transmit current JPEG image to endpoint 8021
Receive processed image on endpoint 8022 (blocking call)
Render processed image to UI
select next image
Loop ?
true
false
Last n samples of cpu 0 and cpu 1 utilization data will be maintained in a SW buffer to render the cpu utilization graph
endpoint 8020
endpoint 8022
endpoint 8024
Print job
packet channel
Processed print output
packet channel
CPU Utilization
messages
Nucleus print processing engine
endpoint 8021
endpoint 8023
endpoint 8025
thread 0
Receive print job on endpoint 8021 (blocking)
Process image JPEG decoding /apply effect
Delay x
Send to endpoint 8022
thread 1
Fetch cpu utilization data
Send to endpoint 8024
Delay x


Initialization Sequence
Initialize MCAPI with a call to mcapi_initialize()
Create Local endpoints:
Local_send
Local_receive
Local_ResourceMonitor
With mcapi_create_endpoint()
Open endpoint for receive packet operation with mcapi_open_pktchan_recv_i()
Wait for Receive endpiont to be opened with mcapi_wait()
Get remote endpoint information with mcapi_get_endpoint()
Connect up the remote endpoint with the local endpint to establish packet channel with mcapi_connect_pktchan_i()
Wait for connection to be established with mcapi_wait()
Open up a packet channel for the send side with mcapi_open_pktchan_send_i()
Wait for tx side to open up with mcapi_wait()
Get the resource monitor endpint mcapi_get_endpoint()
Proceed to Message Process
Message Process
Receive message from foreign node with mcapi_pktchan_recv()
Process received message
Free the received buffer with mcapi_pktchan_free()
Create a response message
Send message to foreign node with mcapi_pktchan_send()
No
Complete?
Yes
Cleanup mcapi with mcapi_finalize()

```
/***************
* Include Files
****************/
…
/* IPC */
#include          "mcapi/inc/mcapi.h"

/**************
IPC connection
***************/

/* MCAPI definitions */

#define           LOCAL_NODEID            1
#define           REMOTE_NODEID           0

static const struct


[**UNRESOLVED****UNRESOLVED****UNRESOLVED****UNRESOLVED**]ports[2] = {
[**UNRESOLVED**],[**UNRESOLVED**]};

mcapi_pktchan_recv_hndl_t     send_handle;
mcapi_pktchan_recv_hndl_t     recv_handle;
mcapi_endpoint_t              local_rm_endpoint;
mcapi_endpoint_t              remote_rm_endpoint;

/*********************************************************************
*
*   FUNCTION
*
*       Initialization
*
*   DESCRIPTION
*
*       This task initializes the IPC.
*
*********************************************************************/
void  Initialization()


      {mcapi_version_tmcapi_version;mcapi_status_tmcapi_status;mcapi_endpoint_tloca
l_send_endpoint;mcapi_endpoint_tlocal_recv_endpoint;mcapi_endpoint_tremote_recv_end
point;mcapi_request_trequest;size_tsize;/* Initialize MCAPI
*/mcapi_initialize(LOCAL_NODEID, &mcapi_version, &mcapi_status);/* Create a local
send endpoint for print job processing. */if(mcapi_status ==
MCAPI_SUCCESS)[**UNRESOLVED**]

      /* Create a local receive endpoint for print job processing. */
      if(mcapi_status == MCAPI_SUCCESS)


[**UNRESOLVED**]

      /* Create a local send endpoint for resource monitor. */
      if(mcapi_status == MCAPI_SUCCESS)
```

```
[**UNRESOLVED**]

      /* Open receive side */
      if(mcapi_status == MCAPI_SUCCESS)


[**UNRESOLVED**]

      /* Wait for the rx side to open. */
      mcapi_wait(&request, &size, &mcapi_status, 0xFFFFFFFF);

      /* Wait till foreign endpoint is created */
      if(mcapi_status == MCAPI_SUCCESS)


[**UNRESOLVED**]

      /* Connect node 0 transmitter to node 1 receiver */
      if(mcapi_status == MCAPI_SUCCESS)


[**UNRESOLVED**r]

      /* Wait for the connect call to complete. */
      mcapi_wait(&request, &size, &mcapi_status, 0xFFFFFFFF);

      printf("Connected \r\n");

      /* Open transmit side */
      if(mcapi_status == MCAPI_SUCCESS)


[**UNRESOLVED**]

      /* Wait for the tx side to open */
      mcapi_wait(&request, &size, &mcapi_status, 0xFFFFFFFF);

      /* Get the remote resource monitor endpoint */
      remote_rm_endpoint = mcapi_get_endpoint(REMOTE_NODEID,
ports[REMOTE_NODEID].rm_rx, &mcapi_status);

      if(mcapi_status == MCAPI_SUCCESS)


[r**UNRESOLVED**]
}
```

```
/*********************************************************************
*
*   FUNCTION
*
*       Sample
*
*   DESCRIPTION
*
*       This task receives a job and responds to job over MCAPI.
*
*
*********************************************************************/
void  Sample()


      {STATUSstatus;mcapi_status_tmcapi_status;UINT32size;UINT32bytesReceived,type,
cmd;unsigned char *in_buffer;unsigned char
out_buffer[MAX_SIZE];mcapi_uint64_ttmp;/* Receive image from foreign node
*/mcapi_pktchan_recv(recv_handle, (void **)in_buffer,(size_t*)&size,
&mcapi_status);/* Do something with MCAPI buffer */… /* Free MCAPI buffer
*/if(mcapi_status == MCAPI_SUCCESS)[**UNRESOLVED**]

      …
      /* Respond to message with data to send in out_buffer */
      mcapi_pktchan_send(send_handle, out_buffer, MAX_SIZE, &mcapi_status);
      if(mcapi_status != MCAPI_SUCCESS)


[n]

      if ( /* Complete then clean up */ )


            {/* Finalize current nodes MCAPI instantiation
*/mcapi_finalize(&mcapi_status);if(mcapi_status != MCAPI_SUCCESS)[n]
      }
}
```

References

[i]POSIX Threads as specified in 'The Open Group Base Specifications Issue 6, IEEE Std 1003.1 is used with no nonstandard extensions, either commercial or research oriented.

[ii]No nonstandard extensions either commercial or research oriented are used. http://www.multicore-association.org/workgroup/mcapi.php

[iii]Some homogeneous multicore architectures employ cores with a mix of shared & non-shared local memory, however this document does not focus on these.

[iv] Throughout this section, we assume that the primary purpose of migrating a program to multicore is to improve its performance (end-to-end time). There are, of course, other reasons to move to multicore, such as improving power consumption. While much of the guidance in this section can be applied with different improvement goals in mind, performance is used as our examples as the primary metric.

[v] Donald Knuth. "Structured Programming with go to Statements." Computing Surveys, vol 6, no 4, Dec 1974, p 268.

[vi] When parallelization begins, it's also important to select a set of machine configurations. It may be a good idea to choose a variety of architectures and different core counts to better understand the effect of changes in different environments. In particular, long-lived programs can be expected to be run in a variety of environments. Understanding a program's behavior across a range of environments helps prevent over-optimization for a particular environment.

[vii] Grotker, Holtmann, Keding, and Wloka recommend that an area of interest be run often enough that its total runtime is at least two orders of magnitude greater than the sampling interval to compensate for variance in measurement results.

[viii] This is one common approach used in serial performance tuning, but not the only option. In some cases, it is difficult to isolate a single, significant hot spot. In such cases, it may be possible to "polish everywhere" by addressing a large number of smaller inefficiencies throughout a program.

[ix] This could be true for a multi|many-processor system or a multi-core system even though it is not in the traditional HPC realm which is not the focus of this guide.

[x] This assumes that a no specialized work queue and thread pool is created to keep processors busy.

[xi] http://www.threadingbuildingblocks.org/

[xii] http://en.wikipedia.org/wiki/Sobel_operator

[xiii] R. Johnson, Open Source POSIX Threads for Win32, Redhat, 2006.

[xiv] D.B. Bradford Nichols and J.P. Farell, Pthreads Programming, O'Reilly & Associates, 1996.

[xv] C.J. Northrup, Programming with UNIX Threads, John Wiley & Sons, 1996.

[xvi] D.R. Butenhof, Programming with POSIX Threads, Addison-Wesley, 1997.

[xvii] G. Taubenfeld, Synchronization Algorithms and Concurrent Programming, Prentice Hall, 2006.

[xviii] G. Taubenfeld, Synchronization Algorithms and Concurrent Programming, Prentice Hall, 2006.

[xix] E.W. Dijkstra, "Cooperating Sequential Processes", (Techniche Hogeschool, Eind- hoven, 1965) Reprinted in: F. Genuys (ed.), Programming Languages, vol. 43, 1968.

[xx] G.L. Peterson, "Myths about the Mutual Exclusion Problem", Information Processing Letters, vol. 12, 1981, p. 115-116.

[xxi] E. Dijkstra, "Solution of a Problem in Concurrent Programming Control", Communications of the ACM, Jan. 1965.

[xxii] J.H. Anderson, "A Fine-Grained Solution to The Mutual Exclusion Problem", Acta Informatica, vol. 30, 1993, p. 249-265.

[xxiii] M. Raynal, Algorithms for Mutual Exclusion, MIT Press, 1986.

[xxiv] J.H. Anderson, "Lamport on mutual exclusion: 27 years of planting seeds", PODC: 20th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, vol. 12, 2001, p. 3-12.

[xxv] G.L. Peterson, "Myths about the Mutual Exclusion Problem", Information Processing Letters, vol. 12, 1981, p. 115-116.

[xxvi] K. Alagarsamy, "Some Myths About Famous Mutual Exclusion Algorithms", SIGACT News, vol. 34, Sep. 2003, p. 94-103.

[xxvii] T.A. Cargill, "A Robust Distributed Solution to the Dining Philosophers Problem", Software---Practice and Experience, vol. 12, Oct. 1982, p. 965-969.

[xxviii] M. Suess and C. Leopold, "Generic Locking and Dead-Prevention with C++", Parallel Computing: Architectures, Algorithms and Applications, vol. 15, Feb. 2008, p. 211-218.

[xxix] H.M.S.D.K. Chen and P.C. Yew, "The Impact of Synchronization and Granularity on Parallel Systems", Proceedings of the 17th International Symposium on Computer Architecture, June. 1990, p. 239-249.

[xxx] M.D. Hill and M.R. Marty, "Amdahl's Law in the Multicore Era", IEEE Computer, Jan. 2008, p. 1-6.

[xxxi] V. Sarkar, Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors, London, UK: Pitman, 1987.

[xxxii] O. Sinnen, Task Scheduling for Parallel Systems, NJ: John Wiley & Sons, 2007.

[xxxiii] M. Garey and D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, Freeman, 1979.

[xxxiv] J. Beck and D. Siewiorek, "Automated Processor Specification and Task Allocation for Embedded Multicomputer Systems: The Packing-Based Approaches", Symposium on Parallel and Distributed Processing (SPDP '95), Oct. 1995, p. 44-51.

[xxxv] J.E. Beck and D.P. Siewiorek, "Modeling Multicomputer Task Allocation as a Vector Packing Problem", ISSS, 1996, p. 115-120.

[xxxvi] K. Agrawal, C. Leiserson, Y. He and W. Hsu, "Adaptive Work-Stealing with Parallelism Feedback", Transactions on Computer Systems (TOCS, vol. 26, Sep. 2008.

[xxxvii] M. Michael, M. Vechev and V. Saraswat, "Idempotent Work Stealing", PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming, Feb. 2009.

[xxxviii] Tobias Schuele, "Embedded Multicore Building Blocks", Embedded World, 2015.

[xxxix] “Multicore Task Management API (MTAPI) Specification V1.0“, Multicore Association, 2013.

[xl] EMB² website, https://embb.io.

[xli] Shameem Akhter, Jason Roberts, "Multi-Core Programming: Increasing Performance through Software Multithreading," Intel Press, 2006.

[xlii] Daniel G. Waddington, Nilabya Roy, Douglas C. Schmidt, "Dynamic Analysis and Profiling of Multi-threaded Systems," http://www.cs.wustl.edu/~schmidt/PDF/DSIS_Chapter_Waddington.pdf, 2007.

[xliii] Yusen Li, Feng Wang, Gang Wang, Xiaoguang Liu, Jing Liu, "MKtrace: An innovative debugging tool for multi-threaded programs on multiprocessor systems," Proceedings of the 14th Asia Pacific Software Engineering Conference, 2007

[xliv] Domeika, Max, "Software Development for Embedded Multicore Systems," Newnespress.com, 2008.

[xlv] Brutch, Tasneem, "Migration to Multicore: Tools that can help," USENIX ;login:, Oct. 2009.

[xlvi] M. Naik, C. S. Park, K. Sen, and D. Gay, "Effective Static Deadlock Detection", ICSE 2009, Vancouver, Canada.

[xlvii] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, and J. Penix, "Using Static Analysis to Find Bugs", IEEE Software, Sept. 2008.

[xlviii] Domeika, Max, "Software Development for Embedded Multicore Systems," Newnespress.com, 2008.

[xlix] Valgrind Instrumentation Framework website, http://valgrind.org/.

[l] Pallavi Joshi, Mayur Naik, Chang-Seo Park, Koushik Sen, "CalFuzzer: An Extensible Active Testing Framework for Concurrent Programs," http://berkeley.intel-research.net/mnaik/pubs/cav09/paper.pdf.

[li] Thorsten Grotker, Ulrich Holtman, Holger Keding, Markus Wloka, "The Developer's Guide to Debugging," Springer, 2008.

[lii] U. Banerjee, B. Bliss, Z. Ma, P. Petersen, "A Theory of Data Race Detection," International Symposium on Software Testing and Analysis, Proceeding of the 2006 workshop on Parallel and distributed systems: testing and debugging, Portland, Maine, http://support.intel.com/support/performancetools/sb/CS-026934.htm

[liii] MSI (Modified, Shared, Invalid) and MESI (Modified, Exclusive, Shared, Invalid) protocols are named after the states that cache lines can have, and used commonly for many shared memory multiprocessor systems. They allow only one "Exclusive" or "Modified" copy of a cache line in the system to serialize the write accesses to the cache line, thus keep the cache *coherent*. Variants of these protocols such as MOSI and MOESI also have been researched.

[liv] This should be distinguished from software-controlled hardware cache, where cache control like flushing and coherence protocol is done by software, and the other operations are performed by hardware.

[lv] We assumed that the programmer has applied all possible techniques for data locality.

[lvi] Scalability of Macroblock-level parallelism for H.264 decoding. The IEEE Fifteenth International Conference on Parallel and Distributed Systems (ICPADS), December 8-11, 2009, Shenzhen, China. http://alvarez.site.ac.upc.edu/papers/parallel_h264_icpads_2009.pdf.

[lvii] The default implementation of MITH uses the POSIX pthread library and launches all contexts and workers as threads, thereby making it easy to port this benchmark suite when using a system that contains an operating system that supports the POSIX library. For advanced users, the MITH abstraction layer provides a method to implement custom execution scheduling.

[lviii] All MITH-based benchmarks support options for a small (4kbytes) and a large (4Mbyte) data size.

[lix] Brutch, Tasneem, “Parallel Programming Development Life Cycle: Understanding Tools and their Workflow when Migrating Sequential Applications to Multicore Platforms,” USENIX ;login:, Oct. 2009.