Characterizing Concurrency of Java Programs

by

Chenwei Wang

Master of Computer Science, Harbin Institute of Technology, 2007

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master's in Computer Science

In the Graduate Academic Unit of Computer Science

Supervisor(s):	Eric Aubanel, PhD, Computer Science
	David Bremner, PhD, Computer Science
Examining Board:	Kenneth Kent, PhD, Computer Science, Chair
	Rainer Herpers, PhD, Computer Science
	Eduardo Castillo, PhD, Electrical and Computer Engineering

This thesis is accepted by the

Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

February, 2016

Abstract

With the emergence of multi-core processors, concurrent programs are becoming the commonplace, as they can provide better responsiveness and higher performance. Java usually uses shared memory as the concurrent programming model. Moreover, Java provides various concurrency related features at the language level to support concurrent programming, such as thread creation and synchronization. However, because of the managed runtime environment of the Java Virtual Machine (JVM), it is hard to understand the performance of concurrent programs written in Java.

Currently available metrics—the number of spawned threads, execution time and throughput—are not enough to characterize a Java program's concurrency behavior. More metrics are needed, such as how many threads contribute to the workload significantly and concurrently and how threads use shared memory. We present a set of metrics to characterize concurrency for Java programs.

IBM's J9 JVM has been instrumented to trace concurrency behaviors, such as thread starts/ends and shared object access. Trace files are dumped when

the VM shuts down. Post processing programs are used to process these trace files to generated these metrics.

We also characterize concurrency for micro benchmarks with different concurrency patterns, as well as commercial and academic benchmarks. The results show that we can characterize concurrency for Java programs by using these metrics we presented.

Acknowledgements

I would like to acknowledge my supervisors Eric Aubanel and David Bremner for their persistent support during my whole study.

Furthermore, I would also like to thank all colleagues of mine at the Centre for Advanced Studies–Atlantic. They have been very helpful by providing valuable technical suggestions and improving my English.

A special acknowledgment to my parents for supporting my overseas study spiritually and financially.

Finally I would like to thank Michael Dawson from the IBM Ottawa team for his technical assistance during all the phases of the project.

Table of Contents

List of Tables

List of Figures

List of Symbols, Nomenclature or Abbreviations

GC	Garbage Collection
JVM	Java Virtual Machine
JIT	Just-In-Time Compiler
JUC	`java.util.concurrent.*` package
PPTD	Parallel Periodic Thread Density
PTD	Periodic Thread Density
PTD_A	Periodic Thread Density for Alternating Operations
PTD_S	Periodic Thread Density for Shared Objects
RA	Alternating Operation Rate
RR	Shared Read Rate
RW	Shared Write Rate
S(N)	Maximum Speedup
TD	Thread Density
TD_A	Thread Density for Alternating Operations
TD_S	Thread Density for Shared Objects
#T	The Number of Spawned User Threads

Chapter 1

Introduction

While multi-threaded programs written in traditional programming languages like C, C++ and Fortran have been extensively studied, understanding the performance of multi-threaded programs written in Java has received little attention. One of the reasons is the complexity of Java programs' managed workloads. Java programs run on top of a virtual machine, which includes its own service threads such as garbage collection, profiling, signal dispatcher and compilation threads. These service threads interact with application threads in many complex, non-deterministic ways, even when the Java program itself is single-threaded. These management threads can stop the application threads (stop-the-world), e.g., to reclaim memory in the heap. They can also replace application threads' stacks, e.g., changing the bytecode instruction to machine code. In addition, they compete with application threads at the micro-architectural level, sharing cores, caches and

off-chip bandwidth. Considering the ubiquity of multi-threaded Java programs, it is crucial to understand their concurrency behavior.

Existing Java benchmarks either report throughput or execution time without telling users how concurrent these Java benchmarking programs really are. The only concurrency related metric is the number of spawned threads. However, the number of spawned threads, including JVM service threads and Java program threads, is a poor metric to measure concurrency.

Therefore, we need to characterize concurrency for Java programs by using a range of concurrency related metrics. We created a tool (implemented on IBM's J9 JVM) to characterize concurrency for Java programs in which a set of concurrency related metrics were provided. These concurrency metrics we obtain can help us understand Java programs' concurrency behavior, such as how many threads contribute significantly and concurrently and how threads use shared memory. They can also be used for future in-depth work, such as distinguishing different concurrency patterns using a subset of metrics (feature selection) and classifying Java concurrency applications into different concurrency patterns.

In Chapter 2 we introduce the necessary background which includes a brief introduction to Java and Java virtual machines, a presentation of parallel programming, a short description of the prevalent concurrency patterns for Java programs, a small introduction to Java benchmarks we used for our project, a short discussion of instrumentation tools and a brief survey of related work on metrics of Java programs.

We continue in Chapter 3 introducing the metrics we will use to characterize concurrency for Java programs. More specifically, we divide the metrics into two groups: level of concurrency metrics and shared memory metrics.

In Chapter 4, we present the tool to obtain metrics presented in Chapter 3. The tool consists of an on-line data-collection part and an off-line post processing part. We describe the instrumentations we performed to IBM's J9 JVM to collect trace data and programs to process trace data to calculate these metrics introduced before. We also discuss the portability of these metrics on other platforms.

We continue with Chapter 5 where we use the tool presented in Chapter 4 to characterize concurrency for micro benchmarks with different concurrency patterns as well as commercial and academic benchmarks like SPECjvm–2008 and DaCapo benchmark suite.

We conclude in Chapter 6, where we give a brief summary of the thesis and the conclusions drawn from it as well as a discussion of possible improvements to our tool and future work based on these concurrency metrics.

Chapter 2

Background

2.1 Java and the Java Virtual Machine

Java is a high level general-purpose programming language that is concurrent, class-based and object oriented. It is platform independent—so called "write once, run everywhere"—by running on top of a platform dependent virtual environment. It also includes garbage collection to reclaim memory from the heap automatically, so there are no safety issues caused by improper explicit deallocation [7].

Java programs run on top of the Java Virtual Machine (JVM), which is the foundation of the Java platform. The JVM executes Java bytecode and realizes Java language's hardware and operating system independency [7]. There are two widely used commercial JVMs: IBM's J9 [8] and Oracle's HotSpot [20]. This thesis presents experiments using an instrumented version

of the IBM's J9 JVM.

2.2 Parallel Programing and Patterns

2.2.1 Parallel Programming

In the single core processor age, there were many techniques to improve single core performance, like increasing the clock frequency and introducing the instruction pipeline, the multi-level cache memory and the hierarchical bus structure. Therefore, applications could gain performance improvement without any change. However, with the emergence of multi-core processors and parallel platforms, traditional applications cannot take the free ride of CPU frequency increases. In order to gain performance, they need to explicitly take advantage of multiple cores/processors. The evolution of code written for single-core platforms into code that can take advantage of multi-core technology is challenging. A sequential program has one thread accessing data structures, main memory, cache and processor. On the other hand a parallel program has multiple threads running at the same time and communicating with each other. Communication may lead to contention, both for software-level resources such as locks, and hardware-level resources such as cache lines. In addition to that, partitioning the workload among the processors in a balanced way is often not easy.

Meade et al. [17] summarized four challenges of parallel programming: parallel programmers require more knowledge; insufficient tools available to aid

parallel programming, testing and debugging; managing data dependencies in parallel code is challenging; and the dependency between programming languages, compilers, libraries, middleware and operating systems.

Concurrency is a software engineering concept with multiple threads running even on a single core processor to provide better responsiveness. However on parallel hardware, these threads are essentially running at the same time in parallel to achieve improved throughput. Nowadays, parallel hardware is the trend and we perform all our tests on multi-core processors, therefore in this thesis, concurrency programs always stand for parallel programs.

2.2.2 Parallel Pattern Language

Because of the challenges in parallel programming, in Mattson et al.'s book [16], a pattern language for parallel programming was presented which aims at lowering the barrier to parallel programming by providing a catalog of methodologies to recurring parallel problems. The pattern language is organized hierarchically into four design spaces from top to bottom: Finding Concurrency, Algorithm Structure, Supporting Structures, and Implementation Mechanisms.

The Finding Concurrency layer structures the problem to exploit concurrency. The Algorithm Structure layer structures the algorithm to take advantage of potential concurrency. The Supporting Structures layer represents an intermediate stage between the Algorithm Structure and Implementation Mechanisms layers with program structuring approaches and data structures.

The Implementation Mechanisms layer deals with how the patterns of the higher layers are mapped into particular programming environments.
Based on Mattson et al.'s work [16], Kurt Keutzer et al. presented a parallel programming framework for the Parallel Computing Laboratory at U.C.Berkeley [2] [15]. They are developing parallel applications in five areas as their research prototypes: music/hearing, speech understanding, content-based image retrieval, intraoperative risk assessment for stroke patients, and a parallel browser.

2.2.3 Concurrency Patterns for Java

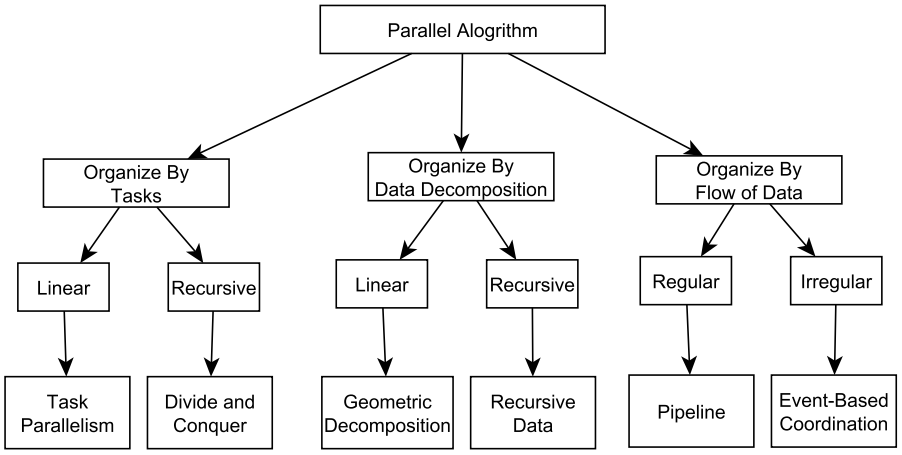

Figure 2.1: Classification of parallel algorithms [16]

Figure 2.1 presents different parallel algorithms based on three organizations: organization by tasks, organization by data decomposition, and organization by flow of data.

According to Mattson et al. [16], in terms of program structure, the Supporting Structures layer includes SPMD, Loop Parallelism, Master/Worker and Fork/Join patterns. Different program structure patterns can be applied in multiple Algorithm Structure patterns. The relationship between the patterns in the Algorithms Structure and Supporting Structures layers is shown in Table 2.1. The black dots stand for the likelihood that the given program structure pattern is useful in the implementation of the Algorithm Structure pattern [16].

Table 2.1: Relationship between supporting structures patterns and algorithm structure patterns [16]

	SPMD	Loop Parallelism	Master/ Worker	Fork/ Join
Task parallelism	● ● ● ●	● ● ● ●	● ● ● ●	● ●
Divide and conquer	● ● ●	● ●	● ●	● ● ● ●
Geometric Decomposition	● ● ● ●	● ● ●	●	● ●
Recursive Data	● ●		●	
Pipeline	● ● ●		●	● ● ● ●
Event-Based Coordination	● ●		●	● ● ● ●

Table 2.2 shows the relationship between the Java programming environment and the program structure patterns [16].

Table 2.2: Relationship between supporting structures patterns and Java [16]

	Java
SPMD	● ●
Loop Parallelism	● ● ●
Master/ Worker	● ● ●
Fork/ Join	● ● ● ●

If we choose three program structure patterns suitable for Java (with at least three dots): Loop Parallelism, Master/Worker and Fork/Join for the corresponding algorithm structure patterns, we also choose the high likelihood with at least three dots, then we get the concurrency patterns for Java shown in Table 2.3.

We realized micro benchmarks with these patterns, which are presented in Chapter 5.

Table 2.3: Concurrency patterns for Java

Algorithm	**Program Structure**	**Examples**
Task Parallelism	Loop Parallelism + Master/Worker	Image-construction program
Divide and Conquer	Fork/Join	Merge Sort, Matrix Diagonalization
Geometric Decomposition	Loop Parallelism	Matrix-Multiplication
Pipeline	Fork/Join	Fourier-transform computations which have 3 stages
Event-Based Coordination	Fork/Join	Coordination:discrete event simulations

2.2.4 Parallel Programming Environments

A parallel programming environment supports the construction of parallel programs by providing language features, parallel directives and APIs. The parallel programming models can be shared memory, message passing in a distributed memory environment or the combination of the two. For traditional programming languages like C, C++ and Fortran, the predominating parallel programming environments are OpenMP [19] for shared memory and MPI [18] for message passing [16].

Java usually uses the shared memory programming model. It has built-in features to support parallel programming[1]. Since Java 5, it includes support for concurrent programming, most significantly in the `java.util.concurrent.*` packages (JUC) [22], which provide a variety of concurrency related functionalities. These include concurrent data structures, management of thread pools, asynchronous task execution, synchronization mechanisms, atomics and locks [6]. It helps writing robust, scalable, and responsive parallel Java applications. In Java, JUC is an important tool for programmers to write concurrent programs. However, a corresponding mechanism is needed to measure the performance of concurrent Java programs.

[1] We do not distinguish parallel and concurrent programming here

2.3 Java Benchmarks

In order to assess the performance and optimize the run-time system, researchers usually rely on a suite of benchmarks.

2.3.1 DaCapo Benchmark suite

The DaCapo benchmark suite is an open source and free tool for Java benchmarking, mainly for academic purposes [23]. It consists of a set of real world, open source applications [3]. The latest version is DaCapo-9.12-bach, released in 2009. DaCapo measures execution time of each sub-benchmark based on operations done over a fixed workload with three sizes: small, default and large. By default, DaCapo uses a number of driver threads equal to the number of logical processors on the target machine.

2.3.2 SPECjvm2008

SPECjvm2008 is a commercial benchmark suite mainly used for Java Runtime Environment (JRE) performance measurement [24]. It includes several real world applications targeting core Java functionality. The SPECjvm2008 workload simulates a variety of common general purpose application computations including text processing, numerical computations, and bitwise computations. SPECjvm2008 scores each sub-benchmark based on operations done over a fixed time rather than elapsed time for a fixed workload.

2.4 Instrumentation

In order to gather profiling information used to measure the performance of concurrent Java programs, the code can be instrumented accordingly.
The instrumentation can be performed in two different ways:

- **Static instrumentation:** extra statements are applied to the source code before compilation, e.g, adding `System.out.println`.
- **Dynamic instrumentation** code is instrumented directly at the runtime, also called runtime instrumentation.

Static instrumentation is the easiest way to add instrumentation code and collect data, however, a Java program usually imports other packages whose source code may not be available. With dynamic instrumentation, there is no need to change the source code: the program can be evaluated in a black box view. There are three widely-used dynamic instrumentation tools.

- **ASM:**

  ASM is a bytecode instrumentation framework. It is used to implement Java bytecode transformation dynamically, like adding a new field/method in a class, or inserting code before method execution and exit [21].

- **BTrace:**

Btrace is a dynamic tracing tool for Java applications. In order to see what happens in a specific area of the code, it manipulates the classes of the target program to inject tracing code [14].

- **JVMTI:**

  Java Virtual Machine Tools Interface (JVMTI) is a native interface of the JVM. It is event-based; when some specified events happen, like threads start/end, a callback function is called. It is helpful in debugging, profiling and analyzing by providing APIs to access the state of the JVM.

2.5 Related Work

This section introduces related work in metrics for Java programs.

Dufour et al. proposed some dynamic metrics for Java programs. They mentioned that "to be useful, dynamic metrics should provide a concise, yet informative, summary of different aspects of the dynamic behavior of programs" [5]. They defined five types of dynamic metrics which characterize a Java program's runtime behavior in terms of data structures, memory use, size and control structure, polymorphism, concurrency and synchronization. In order to measure concurrency, they proposed thread density and lock intensity metrics. They used the Java Virtual Machine Profiler Interface (JVMPI) agent (now replaced by JVMTI [9]) to generate event traces and another Java program to process the event traces and compute various

metrics.

Kalibera et al. provided a group of concurrency related metrics [13]. They used BTrace [14] and ASM [21] bytecode instrumentations and inside Jikes RVM [12] instrumentation to get these metrics they proposed. They devised new metrics to characterize concurrent programs. In order to characterize concurrency level, instead of only reporting the number of spawned threads, they devised *thread density* to measure how many threads contribute to the workload significantly (95% of the operations), and *periodic thread density* to measure how many threads contribute to the workload concurrently. For the latter, they divided the execution time into small periods (100 ms) and in each period they computed thread density and reported the median. In Java, threads typically communicate through shared memory and synchronization, which include synchronized states, barriers with wait/notify, volatile variables and atomic classes [6]. Therefore, they also characterized the shared memory access by computing thread density and periodic thread density on shared objects.

Chapter 3

Concurrency Metrics

In this chapter, metrics to characterize concurrency for Java programs are presented.

In the previous chapter, we discussed related work on metrics for Java applications. In order to characterize concurrency, we adopt the concepts of concurrent threads and shared memory related metrics proposed by Dufour et al. [5]. and Kalibera et al. [13]. However, the metrics they obtained are not accurate. According to Dufour et al.'s conclusion, the profiling tool they used was not versatile and powerful enough, and some metrics were not accurately measured or impossible to compute in a virtual machine without access to the source code. They said the most serious problem with JVMPI is that it is impossible to obtain the state of the execution stack information, thus all the memory related metrics cannot be computed. Kalibera et al. mentioned that the metrics they obtained are not accurate as they cannot

exclude JVM threads. For periodic metrics, they set the period length arbitrarily to 100ms without any explanation. In addition to that, the way they obtain these metrics has a huge overhead as they fully trace all objects' read/write operations.

We refine these metrics and obtain them more accurately with less overhead and use them as a starting point to characterize concurrency for Java programs.

3.1 Level of Concurrency Metrics

For the sake of finding the level of concurrency, the number of spawned threads is not enough, we need other metrics to measure how parallel and multi-threaded a Java program really is.

3.1.1 The Number of Spawned User Threads (#T)

Even if a Java program is single threaded, the number of total spawned threads is more than one, as the JVM spawns a number of service threads like JIT threads and garbage collection threads. We devise this metric to measure the number of user threads spawned by a Java program. JVM's service threads are not counted in this metric.

$$\#T = \left| \{thread_i \mid \forall thread_i \notin (JVM\,Service\,Thread)\} \right| \tag{3.1}$$

3.1.2 Thread Density (TD)

We adopt the concept of the *Thread Density* metric from Kalibera et al.'s work [13] to measure how many threads do a meaningful amount of work. It counts the number of threads that contribute together a significant proportion of the workload. Similar to Kalibera et al.'s work, we consider 95% of the workload to be significant[1]. Consider the case of a thread pool with only a few threads that contribute to the workload whereas others are runnable but actually are waiting for work. The thread density helps us identify this situation. The difference between our thread density metric and Kalibera et al.'s is that to measure a thread's contribution, we use a thread's CPU time instead of total read/write operations on all objects performed by a thread for the sake of less overhead.

Let T_i be the CPU time for user thread i; w.l.o.g. suppose that $T_1 \geq T_2 \geq \ldots T_m$. The thread density (TD) is calculated by the following formula:

$$TD = min\{j \mid \sum_{i=1}^{j} T_i >= 0.95 * \sum_{i=1}^{m} T_i\} \tag{3.2}$$

3.1.3 Periodic Thread Density (PTD)

We adopt the concept of the *Periodic Thread Density* metric from Kalibera et al.'s work [13] to measure how many threads contribute to the workload concurrently. It divides execution into small periods, computes the thread

[1]For all the other metrics with thread density in their names, we also use 95% as the threshold

density in each period and reports the median. The reason for using median rather than mean is to avoid the deviation impact of each period's thread density. For example, consider 100 periods, with 99 periods' thread density equal to 1, but only one period's thread density of 301. The median is 1, whereas mean is 4, which means there are four threads contributing to the workload concurrently. However it is not the case; during the majority of periods there is only one thread working instead of four. When applications have the value of this metric greater than 1, this means multiple mutator threads run concurrently. Therefore, they can be expected to scale on multi-core systems. The choice of period length is very important—either too long or too short cannot reflect the real concurrency level correctly. Kalibera et al. set the period length arbitrarily to 100ms, whereas we performed experiments to determine the proper period length. In Chapter 4, we introduce related experiments and conclude 60ms is a proper period length on our testing machine.

$$\text{PTD} = \text{median}(\text{TD in each period}) \tag{3.3}$$

3.1.4 Maximum Speedup (S(N))

Parallel programs are expected to scale on multi-core machines with a predictable maximum speedup. Counting periods with thread density greater than 1 allows us to obtain the proportion of a Java program's parallel part. Therefore, according to Amdahl's law [1], the maximum speedup can be com-

puted. Let P be the proportion of parallel periods, and N be the number of threads to drive the workload. Amdahl's law can predict the maximum speedup by the following formula:

$$S(N) = \frac{1}{(1\text{-}P)+P/N} \tag{3.4}$$

3.1.5 Parallel Periodic Thread Density (PPTD)

A period can be considered to be parallel when thread density is more than one, in other words, at least two threads contribute in this period. We devise the *Parallel Periodic Thread Density* metric to measure the concurrent level in the parallel stages. It computes the thread density in each parallel period and reports the median.

$$\text{PPTD} = \text{median}(\text{TD in parallel period}) \tag{3.5}$$

3.2 Shared Memory Metrics

The level of concurrency metrics above only tell us how many threads contribute to the workload significantly and concurrently. They can help us identify multi-threaded but non-parallel programs (TD>1, PTD<=1), which are actually running sequentially. However, there are also some highly concurrent Java programs (PTD>1) that are embarrassingly parallel, which means threads are independent of each other and have little communication among

them.

Memory is shared when it is accessible by multiple threads. Some programs are read intensive whereas others have more write operations. Shared reading does not impose too much pressure on threads as more than one thread can read at the same time, but only one thread is allowed to write.

In Java, threads communicate via shared memory and shared memory access poses a great challenge to concurrency. Thus, we need other metrics to measure shared memory access.

3.2.1 Metrics for Shared Objects

3.2.1.1 Thread Density for Shared Objects (TD_S)

The *Thread Density for Shared Object* metric computes the thread density for shared objects. We adopt the concept of this metric to measure how many threads contribute to the shared read/write operations significantly.

Let S_i be the shared read/write operations performed by user thread i; w.l.o.g. suppose that $S_1 \geq S_2 \geq \ldots S_m$. The thread density for shared objects (TD_S) is calculated by the following formula:

$$TD\_S = min\{j \mid \sum_{i=1}^{j} S_i >= 0.95 * \sum_{i=1}^{m} S_i\} \tag{3.6}$$

3.2.1.2 Periodic Thread Density for Shared Objects (PTD_S)

The *Periodic Thread Density for Shared Objects* metric computes the periodic thread density for shared objects. We adopt it to measure how many threads contribute to shared operations concurrently.

$$\text{PTD\_S} = \text{median}(\text{TD\_S in each period}) \tag{3.7}$$

3.2.2 Metrics for Alternating Operations

An operation is considered to be alternating if the immediately following access is performed by a different thread, in other words, if the object's ownership changes. An object's ownership change requires synchronization—an object cannot be modified by two threads at the same time—which limits the concurrent running thread number to one. As for alternating operations, we consider W-R, W-W and R-W patterns, which means an object is written by a thread, and then is read or is written by another thread, or an object is read by a thread, and then is written by another thread.

3.2.2.1 Thread Density for Alternating Operations (TD_A)

The *Thread Density for Alternating Operations* metric computes the thread density for shared objects that have alternating operations. We devise this metric to measure how many threads contribute to alternating operations significantly.

Let A_i be the alternating operations performed by user thread i; w.l.o.g. suppose that $A_1 \geq A_2 \geq \ldots A_m$. The thread density for alternating operations (TD_A) is calculated by the following formula:

$$TD\_A = min\{j \mid \sum_{i=1}^{j} A_i >= 0.95 * \sum_{i=1}^{m} A_i\} \tag{3.8}$$

3.2.2.2 Periodic Thread Density for Alternating Operations (PTD_A)

The *Periodic Thread Density for Alternating Operations* metric computes the periodic thread density for shared objects that have alternating operations. We devise this metric to measure how many threads contribute to alternating operations concurrently. An object's ownership change requires synchronization. When PTD_A is greater than 1, it means the program poses a challenge to shared memory. The higher the value of this metric, the more synchronization is needed between threads. Programs with PTD_A less than 1 are candidates for the embarrassingly parallel pattern, as there is not much communication between threads.

$$\text{PTD\_A} = \text{median(TD\_A in each period)} \tag{3.9}$$

3.2.3 Rate metrics

For all rate measurement metrics, execution time is obtained on the non-instrumented JVM.

3.2.3.1 Shared Read Rate (RR)

We adopt the *Shared Read Rate* metric to measure read operations on shared objects per millisecond.

$$\text{RR} = \frac{\text{total number of shared reads}}{\text{execution time}} \tag{3.10}$$

3.2.3.2 Shared Write Rate (RW)

We adopt the *Shared Write Rate* metric to measure write operations on shared objects per millisecond.

$$\text{RW} = \frac{\text{total number of shared writes}}{\text{execution time}} \tag{3.11}$$

3.2.3.3 Alternating Operation Rate (RA)

We adopt the *Alternating Operation Rate* metric to measure alternating operations per millisecond. If a program has a high shared write rate, but all these write operations are performed by only one thread which requires no synchronization at all — then the Alternating Modification Rate helps us recognize this situation.

$$\text{RA} = \frac{\text{total number of ownership changes of shared objects}}{\text{execution time}} \tag{3.12}$$

3.3 Summary

In this chapter, we introduced metrics which are used to characterize concurrency of Java programs. The summary of these metrics is shown in Table 3.1. Some metrics are adopted from Kalibera et al [13], other are devised by us. The last column of Table 3.1 indicates whether a metric is adopted or introduced by us.

Table 3.1: Metrics summary

Type	**Metric Name**	**Abbreviation**	**New**
Level of Concurrency	The Number of Spawned User Threads	#T	Yes
	Thread Density	TD	No
	Periodic Thread Density	PTD	No
	Parallel Periodic Thread Density	PPTD	Yes
	Speedup	S(N)	Yes
Shared Memory	Thread Density for Shared Objects	TD_S	No
	Periodic Thread Density for Shared Objects	PTD_S	No
	Thread Density for Alternating Operations	TD_A	Yes
	Periodic Thread Density for Alternating Operations	PTD_A	Yes
	Shared Read Rate	RR	No
	Shared Write Rate	RW	No
	Alternating Operation Rate	RA	No

Chapter 4

Concurrency Metrics Tool

In this chapter, we present the tool that is used to obtain the metrics shown in the previous chapter. We also discuss the portability of the metrics we presented previously.

4.1 Overview

In Figure 4.1 we present an illustration of the high-level architecture of our metrics tool. The tool consists of an on-line data-collection part and an off-line post processing part. In the data-collection stage, we run Java programs on the instrumented IBM J9 JVM and a JVMTI agent which generate trace files for the post processing stage. Later in the post processing stage, we obtain the metrics to characterize concurrency for Java programs.

Because of the managed environment of Java program execution, there are

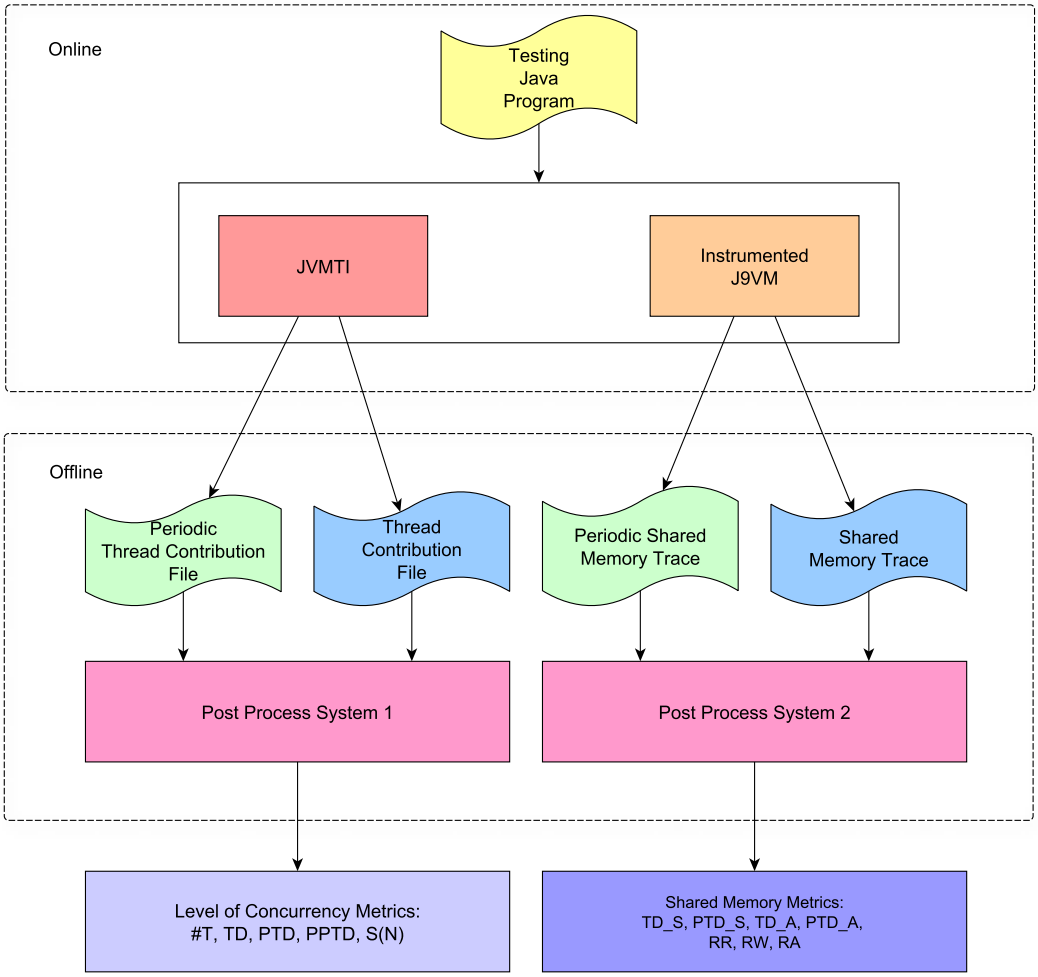

Figure 4.1: Metrics tool architectural overview with online data collection and offline post processing stages

several JVM service threads running. In our tool, all these JVM service threads are excluded from all metrics.

4.2 JVMTI Agent

A JVMTI agent was developed to collect data for the level of concurrency metrics.

4.2.1 Methodology Overview

In Figure 4.2, the JVMTI architecture is presented. The JVMTI agent is a native interface compiled as a dynamic library and is loaded when the VM starts. In the JVMTI agent, we register a group of events relevant to the level of concurrency metrics generation. When a registered event happens, the corresponding callback function is invoked to perform relevant operations. We use the CPU time spent by a thread in both user and system mode to measure a thread's contribution. The operating system provides relevant APIs to get a thread's CPU time. When a thread ends, we get its CPU time and write it to a file. We also set up a timer to capture all user threads' CPU time every 60ms for the purpose of periodic metrics. When garbage collection starts, the timer is paused, and when garbage collection ends, the timer resumes. Here we use stop-the-world garbage collection policy, and during GC periods all user threads are halted.

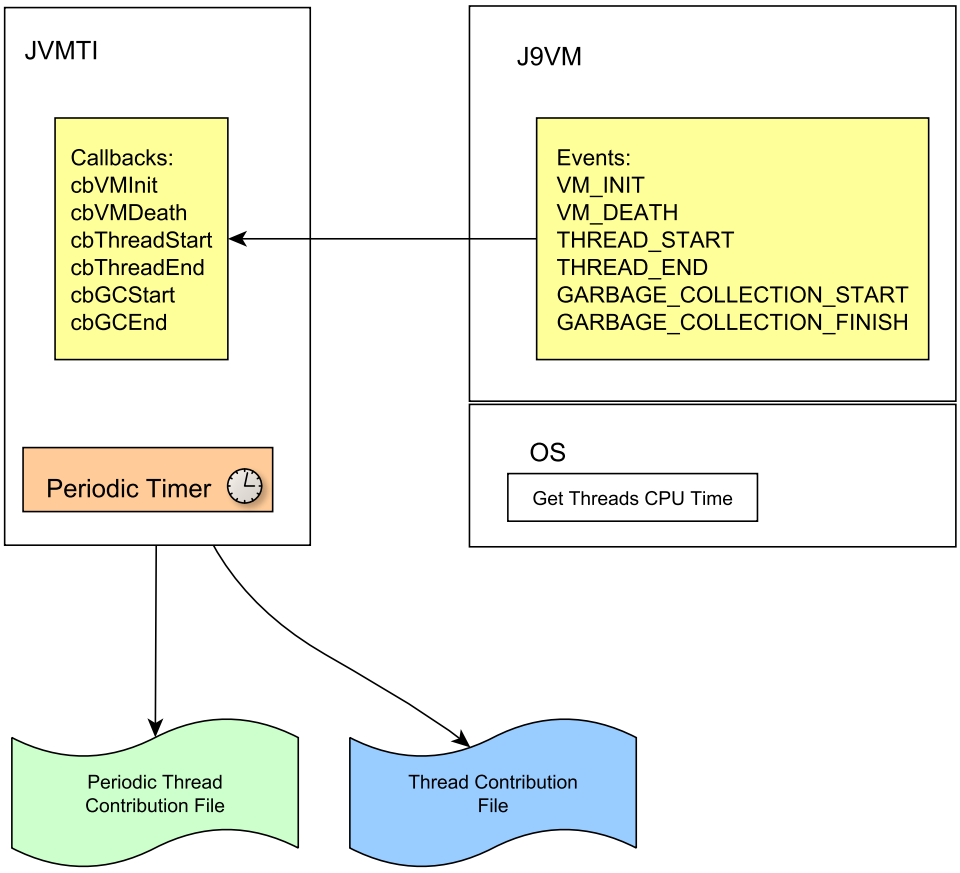

Figure 4.2: Overview of the JVMTI agent implementation with registered events, callbacks and periodic timer

4.2.2 Events and Callbacks

The registered events and the corresponding callback functions' work are shown in Figure 4.2 and include:

- **JVMTI_EVENT_VM_INIT:**

  This event notifies the completion of VM initialization. The corresponding callback function starts a timer which triggers an alarm every 60ms.

- **JVMTI_EVENT_THREAD_START:**

  This event is triggered by a new thread creation. In the corresponding callback function, we add logic to determine whether the thread is a VM service thread or a user thread. If it is a user thread, we record it in a look-up table.

- **JVMTI_EVENT_THREAD_END:**

  This event is triggered by a thread termination.

  In the corresponding callback function, we check whether it is a user thread. If it is a user thread, we record its CPU time, write to the trace file and remove it from the look-up table.

- **JVMTI_EVENT_VM_DEATH:**

  This event signals the JVMTI agent that the VM has terminated. No events will be generated after this event.

In the corresponding callback function, we obtain the CPU time of user threads that are in the look-up table. Not all user threads trigger the JVMTI_EVENT_THREAD_END event. The reason is when the VM dies, daemon threads are still alive, like the main thread and threads in the thread pool. Thus we need to capture these threads' CPU time when the VM shuts down. We also kill the timer to prevent further periodic thread contribution trace generation.

- **JVMTI_EVENT_GARBAGE_COLLECTION_START**

  This event is generated when a garbage collection pause begins.

  To avoid periodic thread contribution trace generation during the garbage collection stage, in the callback function we pause the timer.

- **JVMTI_EVENT_GARBAGE_COLLECTION_FINISH**

  This event is generated when a garbage collection pause ends.

  To resume periodic thread contribution trace generation after the garbage collection stage, in the callback function we resume the timer.

4.2.3 Samples

A snippet of a thread contribution file is attached in Listing 4.1. Each line contains the following information: event, thread's name, thread id and thread's CPU time. The unit of CPU time is nanoseconds.

Listing 4.1: Sample thread contribution file

```
ThreadEnd;Thread-20;18544;12123258
ThreadEnd;Thread-17;18541;11314805
ThreadEnd;Thread-19;18543;13816269
ThreadEnd;Thread-15;18539;11331508
ThreadEnd;Thread-13;18537;14851697
ThreadEnd;Thread-21;18545;12793984
ThreadEnd;Thread-10;18534;12001135
ThreadEnd;Thread-12;18536;11966654
ThreadEnd;Thread-14;18538;14745523
ThreadEnd;Thread-11;18535;13205930
...
ThreadEnd;Thread-24;18548;12452020741
ThreadEnd;Thread-34;18558;12283960277
ThreadEnd;Thread-36;18560;12324358925
ThreadEnd;Thread-26;18550;12367223937
ThreadEnd;Thread-37;18561;12456969497
ThreadEnd;Thread-1;18563;176991
ThreadEnd;Thread-6;18564;1148329
ThreadEnd;Thread-5;18566;468774
vmdeath;main;18478;1723832127
vmdeath;Java2D Disposer;18529;295898
```

A snippet of a periodic thread contribution file is attached in Listing 4.2. The keyword *CPU:* is the delimiter for different periods. Since we need to get threads' CPU time every 60ms, to reduce the size of the trace file, we only write crucial data, which are indispensable in the post processing stages. Each periodic thread contribution line contains the following data: thread id, and thread's CPU time. The unit of CPU time is nanoseconds.

Listing 4.2: Sample periodic thread contribution file

```
...
CPU:
18478;1450747733;
18529;295898;
CPU:
18478;1489413586;
18529;295898;
18531;24296870;
18532;898502;
18533;673354;
18534;614830;
18535;486674;
...
```

4.3 JVM Instrumentation

The IBM's J9 JVM is instrumented to collect data for shared memory metrics.

4.3.1 Methodology Overview

The overview of our instrumented J9 JVM is shown in Figure 4.3. The object model is extended for the purpose of shared object tracing. We use the object access barrier to capture shared objects' access and store the records in a hash table for fast look-up. In the J9 JVM's implementation, the object access barrier is based on the bytecode. The JIT compiler changes some bytecode instructions to machine code, which cannot be captured by the object access barrier. So the JIT must be disabled. When the VM shuts down, we dump the data from the hash table to a file, which is a full trace of all shared objects' access records. We also set up a timer to capture shared objects' access records every 60ms for the purpose of periodic metrics. When garbage collection starts, the timer is paused, and when garbage collection ends, the timer resumes. Here we use stop-the-world garbage collection policy, during GC there is no shared object access as the world is stopped.

4.3.2 Extension of the VMObject

Each object in the JVM is represented by a VMObject. For shared memory trace collection, the VMObject struct needs to be extended to indicate

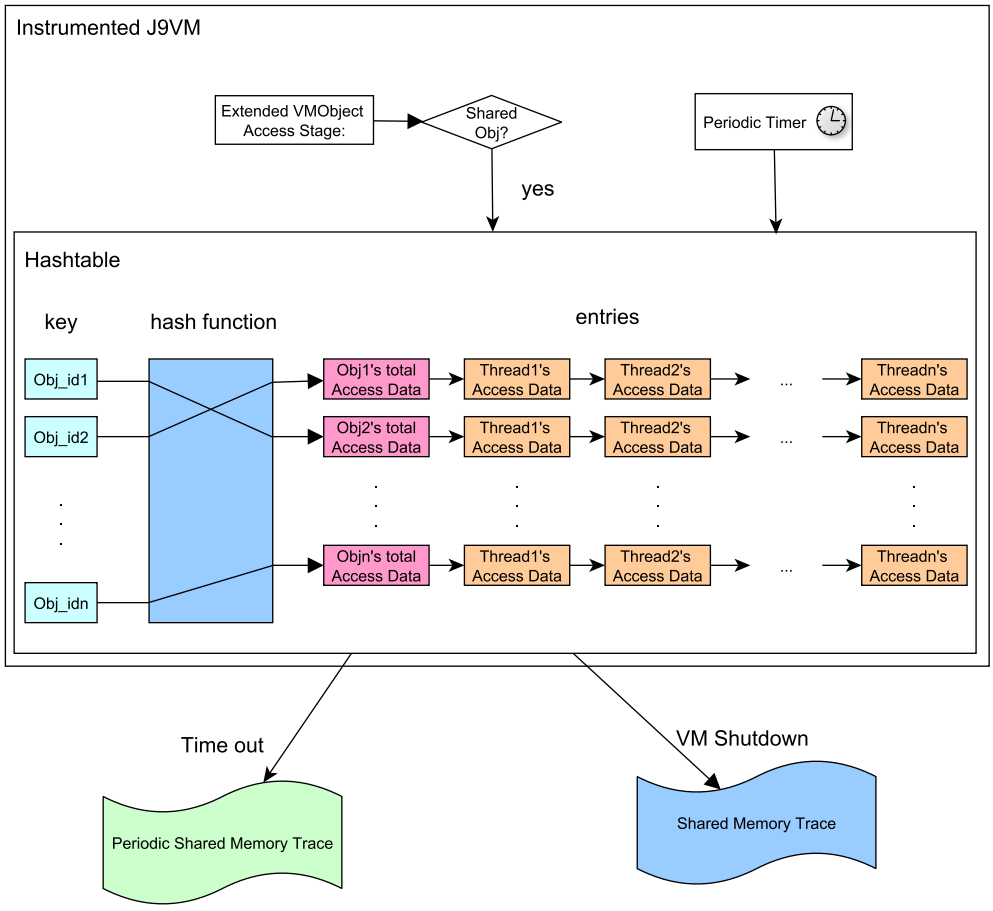

Figure 4.3: J9 JVM instrumentation with shared object check, hash table and periodic timer

whether an object is shared or not. The extended VMObject model is shown in Figure 4.4.

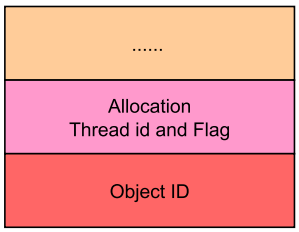

Figure 4.4: Extended VMObject with two new fields

The new field *allocThreadAndFlag* is initialized with allocation thread id at the allocation of an object. Moreover, for the sake of keeping the memory footprint small, we use the least significant bit of the new field to store the flag indicating whether the object is shared. When an object is accessed by a thread other than its allocation thread, we mark it as a shared object by setting the flag to 1.

To simplify the access to the flag and allocation thread id, the corresponding bit shifting operations are encapsulated in the following macros:

- **SHARED_OBJECT_GET_ALLOC_THREAD:**

  Returns the object's allocation thread's id.

- **SHARED_OBJECT_GET_FLAG:**

  Returns shared object flag. If the object is shared, returns 1, otherwise returns 0.

- **SHARED_OBJECT_SET_ALLOC_THREAD:**

  Sets the allocation thread's id.

- **SHARED_OBJECT_SET_FLAG:**

  Sets shared object flag to 1 indicating this is a shared object.

To distinguish one object from another, we also extended VMobject with another field storing an object's unique id. We do not use an object's hashcode as the id because an object' hashcode may be duplicated.

4.3.3 Data Structure

The core data structure is the hash table which stores shared objects' access records. To be specific, we use shared objects' ids, which are integers, as the key, and the hash function is simply using these ids modulo the hash table's size. Each shared object's access record is an entry in the hash table, which is shown in Figure 4.5. The entry has the shared object's general access record followed by per-thread access records. The general access data contains the following:

- The shared object's id.
- The number of total read operations on this shared object
- The number of total write operations on this shared object.
- The number of total alternating operations on this shared object.

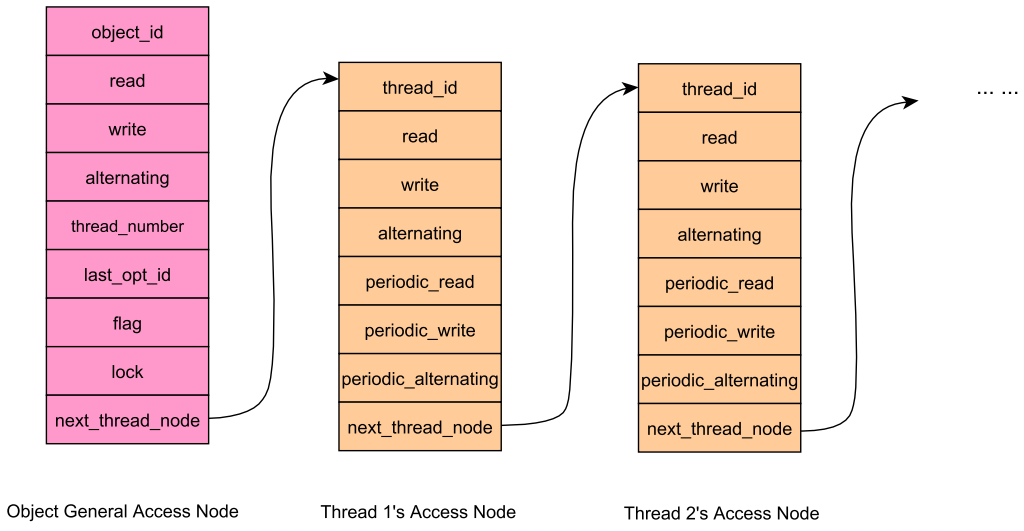

Figure 4.5: Data structure for the hash table entry with general access node and per-thread access nodes

- The number of threads working on this shared object.
- The id of the thread that performs the previous operation. If the value is greater than 0, the previous operation is a read, otherwise it is a write.
- A flag indicating whether there are shared operations on this object in the current period.
- A lock to prevent race conditions caused by concurrent updates.
- A pointer to the shared object's per-thread access records.

To keep the general access record small, there is no need to keep the allocation thread id as it is not used to obtain the metrics.
Each thread has its own shared object access record. The shared object's per-thread access records are connected in a linked-list fashion. Each record contains the following:

- Thread id.
- The number of read operations the thread did on this shared object.
- The number of write operations the thread did on this shared object.
- The number of alternating operations the thread did on this shared object.
- The number of periodic read operations the thread did on this shared object in the current period.
- The number of periodic write operations the thread did on this shared object in the current period.
- The number of periodic alternating operations the thread did on this shared object in the current period.
- A pointer to the shared object access records performed by another thread.

4.3.4 Algorithms

We use an object access barrier to capture shared object access traces. When an object is accessed, we check its flag in the extended *allocThreadAndFlag* field. If the flag is 1, we write the access data into the hash table. If the flag is 0, we compare the accessing thread id with the stored allocation thread id in the extended *allocThreadAndFlag* field. We set the flag to 1 when the accessing thread is not the allocation thread.

For thread-safety, we use a hash table level lock to synchronize threads. But there is no need to lock the whole hash table every time we access it. The synchronization only happens when a new shared object access record is inserted into the hash table. For example, if two threads happened to read an object concurrently that is not in the hash table yet (first time access), without synchronization, two entries would be created in the hash table.

For records that are already in the hash table, each thread updates its own record block, so there is no race condition raised by a shared object's per-thread access record update. Moreover, each shared object's general access record has a lock to prevent race conditions caused by concurrent updates. This is a fine-grained, per entry lock for better efficiency.

For alternating operations' statistics, each shared object's general access record has a field name *last_opt_tid* to keep the immediately preceding thread that manipulated this object. To save memory, we do not use an extra field to denote what type of operation was performed by the immediately preceding thread. We reuse the *last_opt_tid* field with positive and negative values.

If *last_opt_tid* is less than 0, the corresponding immediately preceding thread performed a write operation, otherwise it performed a read operation. We capture all the W-R, W-W and R-W patterns performed by different threads. For the periodic shared object trace, we dump the shared objects' per-thread periodic data every 60ms to a file. The periodic data dump is paused during the garbage collection stage, because the world is stopped and there is no shared object access perform by user threads. The periodic data dump is resumed after the garbage collection stage. Every 60ms, we do not dump the whole hash table containing the access records for all the shared objects, but only objects having shared operations in the current period. To this end, each shared object's general access record has a flag to denote whether there are operations on this object in the current period. The flag is reset to 0 whenever we dump periodic data. Meanwhile, *periodic_read*, *periodic_write* and *periodic_alternating* fields in the per-thread access records are reset to 0. The flag is set to 1 when there are threads manipulating that object in the current period. We only dump shared objects' per-thread periodic data with flag equal to 1.

For shared object trace, we dump all shared objects' access records from the hash table to a file when the VM shuts down.

4.3.5 Samples

A snippet of shared object trace is attached in Listing 4.3. Each line has the following values: object id, the number of threads that operate on the

object, the number of total read operations, the number of total write operations, the number of total alternating operations followed by the detailed per-thread access data on the object. The per-thread access data contains the following data: thread id, the number of read operations, the number of write operations and the number of alternating operations.

Listing 4.3: Sample shared object trace

```
...
423398:3:0R:17W:17A:6745,0,6,6;6746,0,6,6;6748,0,5,5;
423401:4:649728R:0W:0A:6750,164109,0,0;6751,170082,0,0;6749,165747,0,0;6752,149790,0,0;
423409:4:522846R:0W:0A:6750,134955,0,0;6751,124785,0,0;6749,134883,0,0;6752,128223,0,0;
423418:4:857067R:0W:0A:6749,213291,0,0;6752,213405,0,0;6750,212934,0,0;6751,217455,0,0;
423419:4:857070R:0W:0A:6749,213291,0,0;6752,213405,0,0;6750,212934,0,0;6751,217455,0,0;
423444:4:857071R:0W:0A:6749,213291,0,0;6752,213405,0,0;6750,212934,0,0;6751,217455,0,0;
423445:4:857074R:0W:0A:6749,213291,0,0;6752,213405,0,0;6750,212934,0,0;6751,217455,0,0;
...
```

A snippet of a periodic shared object trace file is attached in Listing 4.4. The keyword *sigroutine:* is the delimiter for different periods. Since we need to get shared object access data every 60ms, to reduce the size of the trace file, we only write crucial data, which are indispensable in the post processing stages. Each periodic shared object access data line has an object id and a list of detailed per-thread access data on the object. The per-thread access data contains the following data: thread id, the number of read operations, the number of write operations and the number of alternating operations.

Listing 4.4: Sample periodic shared object trace

```
...
674:31078,50,0,0;31079,3,1,0;
686:31078,13,0,0;
```

```
733:31078,3,1,0;
1095:31078,28,0,0;
sigroutine:
674:31078,50,0,0;31080,3,2,0;
686:31078,13,0,0;
733:31078,3,1,0;
1095:31078,28,0,0;
sigroutine:
674:31078,50,0,0;
686:31078,13,0,0;
...
```

4.4 Post Process

The post processing system processes trace files generated by the JVMTI agent and the instrumented J9 JVM and generates the metrics.

4.4.1 Level of Concurrency Metrics Generation

For the thread contribution file, we calculate the whole workload by summing up all user threads' CPU time. Then we sort threads in order of descending CPU time. Finally, we label each thread according to cumulative CPU time and throw away all of the threads after 95% cumulative, as in our metrics, we

consider 95% of the work to be significant. The TD metric can be calculated by counting the number of remaining threads.

A C++ program is used to process the periodic thread contribution file to calculate the number of significantly contributing threads in each period and report the median as the PTD metric.

The proportion of parallel periods is calculated by the number of parallel periods over the number of total periods obtained by using a single thread to drive the workload. Then the S(N) metric can be calculated according to Amdahl's law.

We also compute the thread density in each parallel period and report the median as the PPTD metric.

4.4.2 Shared Memory Metrics Generation

A C++ program is used to traverse the shared object trace dumped from the hash table to obtain each thread's shared read/write operations performed on all shared objects. Similar to what we did to obtain the TD metric, we sort threads in the order of descending shared operations. We label each thread according to cumulative shared operations and throw away all of the threads after 95% cumulative, as in our metrics, we consider 95% of the work to be significant. The TD_S metric and the TD_A metric can be calculated by counting the number of remaining threads. The difference is that for the TDS metric we use read/write operations, whereas for the TD_A metric we only use alternating operations.

Another C++ program is used to process the periodic shared object trace to calculate the number of significantly contributing threads in each period and report the median. For PTD_S we consider shared read/write operations as a thread's contribution. For PTD_A we consider alternating operations as a thread's contribution.

For all the rate related metrics:RR, RW and RA, the execution time is based on the uninstrumented version of the VM.

4.5 Period Length Determination

In the previous chapter, we mentioned that the choice of period length is very important. If it is too short, the number of contributing threads is too low in each period, the limiting case is only one thread. If it is too long, the number of contributing threads is too high in each period, the limiting case is the number of all user threads.

We performed experiments to determine how long the period should be on our testing machine. The testing machine has 16 physical Intel Xeon Nehalem-based cores running at 1.87GHz with 2-way hyperthreading (SMT2) to 32 logical cores. This system has 64GB of DDR3 800MHz RAM, 16GB per socket, and runs CentOS 6.3 with kernel version 2.6.32.

To determine the lower bound of period length, we use a highly concurrent micro benchmark with a small critical section to make sure the time period we choose is not too small. The expected PTD_S should be equal to the

number of user threads.

The critical section is small, which only includes an addition and a read/write operation on a shared object. The code snippet is shown in Listing 4.5.

Listing 4.5: Code to determine the lower bound of period length

```
static class InnerClass{
  public int x;
  public InnerClass(){ x=1; }
}
private static class TestThread extends Thread {
  InnerClass inner;
  public TestThread(InnerClass inner) {
    this.inner=inner;
  }
  @Override
  public void run() {
    while (intrinsic()) {}
  }
  public boolean intrinsic() {
    Random rand = new Random();
    int n = rand.nextInt(100);
    synchronized(lock){
      operation++;
      if (n < 50) { // read operation
```

```
            int value = n;
            value += inner.x;
            read++;
        } else {// write operation
            inner.x=n;
            write++;
        }
        return operation <total_opertion;
    }
  }
}
```

After our experiments, we found that when the period length was less than 20ms, PTD_S was less than the number of user threads. So our period should be longer than 20ms on our testing machine.

To determine the upper bound of period length, we use a micro benchmark with a large critical section to make sure the time period we choose is not too large. The expected PTD_S should be less than the number of user threads. To make our critical section larger, we implemented a busy loop with 2^{15} iterations of addition. The code snippet is shown in Listing 4.6.

Listing 4.6: Code to determine the upper bound of period length

```
@Override
public void run() {
```

```
  Random rand = new Random();
  int n = rand.nextInt(100);
  synchronized(lock){
    operation++;
    if (n < 50) { // read operation
      int value = n;
      value += inner.x;
      read++;
    } else {// write operation
      inner.x=n;
      write++;
    }
    //busy loop for large critical section
    for (int i = 0; i < N1 ; i++)
      counter++;
  }
}
```

After our experiments, we found that when the period length was longer than 100ms, PTD_S reached the number of user threads.

We also rewrote the *xalan* benchmark using the Fork-Join tool in the JUC package. The rewritten *xalan* has a maximum 18X speedup. We varied PDT_S with different period lengths. When the period length was longer

than 100ms, the PD_S metric exceeded the maximum speedup. Thus our period should be no longer than 100ms on our testing machine.

So the period length should be between 20ms and 100ms. The middle value of 60ms would be a proper period length with equal distance to the lower and the upper bounds.

4.6 Metrics Portability

In chapter 3 we introduced the metrics used in this thesis to characterize concurrency behaviors for Java programs. Then in chapter 4 we presented the tool to obtain these metrics. These metrics are platform independent, but our tool has platform dependencies. The concurrency metrics tool is implemented on IBM's J9 JVM running on Linux. In this section, we discuss the portability of these metrics.

For level of concurrency metrics, we used a JVMTI agent which can be applied to other JVMs on different platforms. The agent used a Linux system call to get a thread's CPU time. When porting to other platforms, the corresponding system call should be used.

For shared memory metrics, we extended the J9's VMObject model. Moreover, we used IBM's J9 JVM's access barrier to capture shared object access traces. These approaches can only be ported to JVMs with access to source code and having an implementation of the object access barrier.

For periodic related metrics, a timer is set up by a Linux system call. When

porting to other platforms, the corresponding system call should be used.

For all metrics with thread density in their names, we ruled out JVM service threads by checking a special flag in J9. When porting to other JVMs, other approaches should be applied to differentiate user threads and JVM service threads. However, other JVMs may not be able to rule out JVM service threads.

In Table 4.1, we summarize the necessary changes when porting our metrics tool to other JVMs and other platforms.

Table 4.1: Metrics portability

Metrics Type	**Port to Other JVMs**	**Port to Other Platforms**
Level of Concurrency	rule out VM threads	thread CPU time system call set timer system call
Shared Memory	extend VMObject model object access barrier rule out VM threads	set timer system call

4.7 Overhead Discussion

The metrics tool introduces some overheads, but it does not significantly interfere with the normal execution of the threads and does not change their concurrency behavior.

The overhead introduced by the JVMTI agent is quite small, as we obtain user threads' CPU time by a Linux system call that is extremely inexpensive.

However, we obtain shared memory metrics by an object access barrier, so the JIT must be off, which makes our Java programs run at least 10 times slower. But JIT-off does not interfere with the normal execution of user threads. Moreover, we only trace shared objects rather than all allocated objects. In Chapter 5, we will demonstrate that the percentage of shared objects out of total allocated objects is extremely low in existing widely used benchmarks. We dump shared objects access data from the hash table containing the access records for all the shared objects when the VM shuts down which involves huge I/O operations, but it is performed when a Java program is finished. We also dump data from the hash table every 60ms, but not the whole hash table, only objects having shared operations in the current period. According to the sample periodic shared object trace shown in Listing 4.4, few objects are shared in each period, and we only write crucial data to a file. So there is not much I/O involved in every periodic dump.

Chapter 5

Measurements

In this chapter, we characterize concurrency for different Java programs including micro benchmarks with different concurrency patterns as well as commercial and academic benchmarks.

The testing machine has 16 physical Intel Xeon Nehalem-based cores running at 1.87GHz with 2-way hyperthreading (SMT2) to 32 logical cores. In our experiments, we use four threads to drive the workload to obtain metrics mentioned previously. The reason that we use four threads is Kalibera et al. tested DaCapo using 4 threads [13], so we can compare some results. We also perform scalability tests to get the maximum speedup on the target machine. Moreover, we try to make the impact of garbage collection as small as possible by using a huge Java heap (32G). In addition, JIT is disabled as the object access barrier is used.

5.1 Micro Benchmarks Measurement

In this section, we characterize concurrency for a set of micro benchmarks with different concurrency patterns.

5.1.1 Micro Benchmarks

In Chapter 2, we discussed concurrency patterns for Java, which are shown in Table 2.3. In this section, we present micro benchmarks with different concurrency patterns: Divide and Conquer/Fork-Join, Geometric Decomposition/Loop Parallelism, Event-Based Coordination/Fork-Join and Pipeline/Fork-Join.

5.1.1.1 Divide and Conquer/Fork-Join Pattern

We use two micro benchmarks for Divide and Conquer/Fork-Join pattern: *Maximum Finder* and *Merge Sort*. For the underlying threads fork-join program structure, we use Fork-Join framework in the JUC package. The Divide and Conquer/Fork-Join Pattern is shown in Figure 5.1. In this pattern, we divide a problem into two sub problems recursively until the problem size is small enough. In JUC's Fork-Join framework, a thread pool is used to prevent creating too many threads recursively. In these two benchmarks, we set the capacity of the thread pool to the number of driver threads specified by the command line option.

Maximum Finder uses `java.util.concurrent.RecursiveTask`, whereas *Me-*

rge Sort uses `java.util.concurrent.RecursiveAction`. Both `RecursiveTask` and `RecursiveAction` extend `java.util.concurrent.ForkJoinTask`. The difference is that `RecursiveTask` returns a value, but `RecursiveAction` does not.

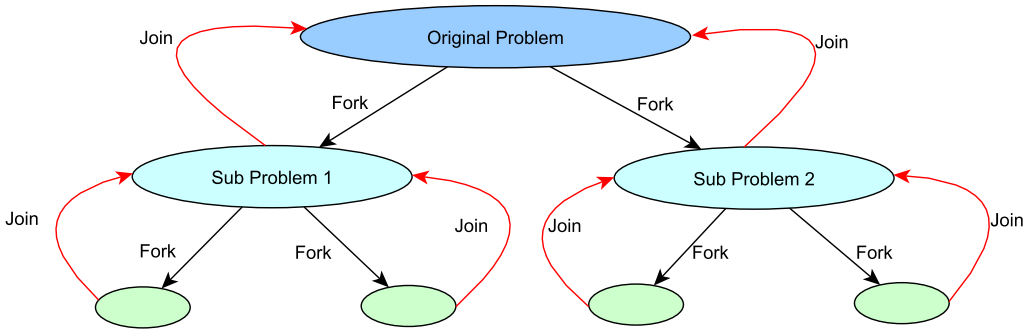

Figure 5.1: Divide and conquer/Fork-Join Pattern with recursive division into two sub-problems

The *Maximum Finder* micro benchmark finds the maximum number out of 20,000,000 numbers. We specify the sequential threshold to be 10 which means when there are 10 numbers, instead of splitting into two tasks, we find the maximum number directly.

The pseudo code is shown in Listing 5.1

Listing 5.1: Maximum Finder

```
//find the maximum number in data[start...r]
MaximumFinder(data,start,end){
  if ((end-start) < SEQUENTIAL_THRESHOLD)
    return findDirectly();
  else {
     split = (end-start) / 2 ;
```

```
    spawn: left=MaximumFinder(data,start,start+split))
           right=MaximumFinder(data,start+split,end)
    sync
    return max(left, right);
  }
}
```

The complete code of the *Maximum Finder* micro benchmark is shown in Listing A.1 in Appendix A.

The *Merge Sort* micro benchmark sorts 20,000,000 numbers. We also parallelize the merge stage by using Cormen's parallel merge algorithm [4]. The pseudo code is shown in Listing 5.2

Listing 5.2: Merge Sort

```
//Sort the elements in A[p...r] and store them in B[s...s+r-p]
MergeSort(A, p, r, B, s){
  n = r-p+1
  if (n == 1)
    B[s] = A[low]
  else {
    let T[1...n] be a new array
    q=(p+r)/2
    q2=q-p+1
    spawn: MergeSort(A, p, q, T, 1)
           MergeSort(A, q+1, r, T, q2+1)
    sync
      parallel_merge(T, 1, q2, q2+1, n, B, s);
  }
}

/* Merge two ranges of source array T[ p1 .. r1 ] and T[ p2 .. r2 ]
  into destination array B starting at index p3 */
parallel_merge(T, p1, r1, p2, r2, B, p3) {
  n1=r1-p1+1
```

```
  n2=r2-p2+1
  if(n1<n2)
    swap(n1, n2), swap(p1, p2), swap(r1, r2)
  if (n1==0) return
  else{
    q1=(p1+r1)/2
    q2=Binary_Search(T[q1],T, p2, r2)
    q3=p3+(q1-p1)+(q2-p2)
    B[q3]=T[q1]
    spawn: parallel_merge(T, p1, q1-1, p2, q2-1, B, p3)
           parallel_merge(T, q1+1, r1, q2, r2, B, q3+1)
    sync
  }
}
```

The complete code of the *Merge Sort* micro benchmark is shown in Listing A.2 in Appendix A.

5.1.1.2 Event-Based Coordination/Fork-Join Pattern

We use a micro benchmark simulating a supermarket with several cashiers to serve a large number of customers.

The arrival of customers is subject to a Poisson distribution. We specify 10 customer arrival events within 100ms by setting λ to 100 in a Poisson distribution. This is a simulation program rather than a real supermarket, so we make it run faster with customer arrival rates beyond the normal human movement speed.

Each customer has a shopping list which determines how long a customer will go shopping and how long a cashier will process a customer. The length

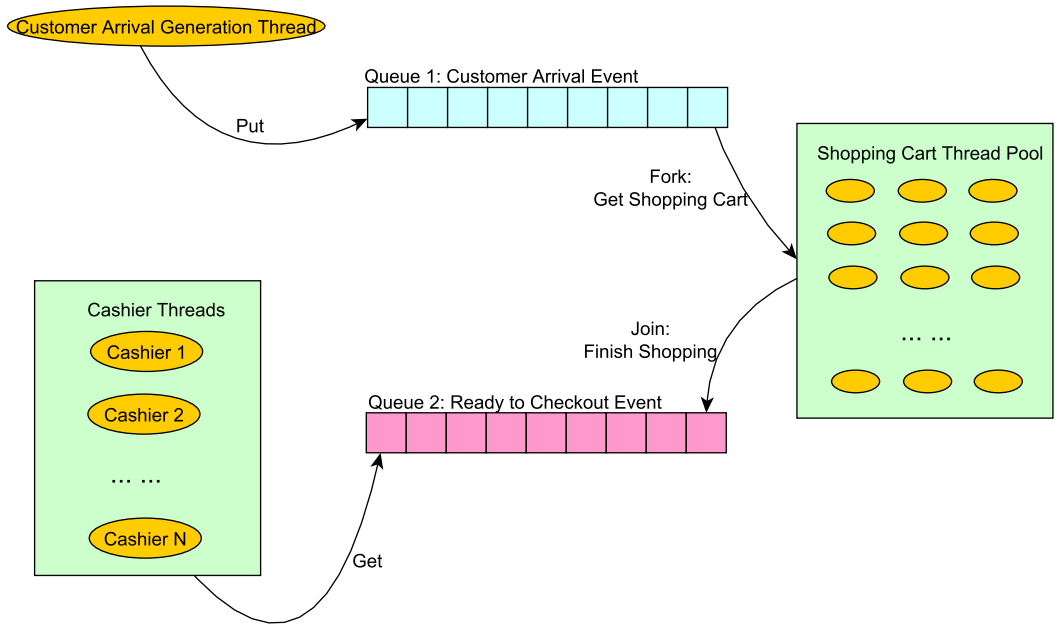

Figure 5.2: Supermarket micro benchmark with specified number of cashiers and shopping cart thread pool

of a customer's shopping list is subject to Uniform distribution from 100 to 1000.

The micro benchmark uses a thread to generate the events of customer arrival. Instead of creating a new thread for each customer, which results in the creation of too many threads, a thread pool is used. Each thread in the pool simulates a shopping cart. A customer stays in the "Customer Arrival" queue until a shopping cart is available. Then the customer can enter into the supermarket and start shopping. The customer is put into the "Ready to Checkout" queue when shopping is finished. The micro benchmark has

a configurable number of cashiers to service customers in the "Ready-to-Checkout" queue.

We use `java.util.concurrent.LinkedBlockingQueue` to coordinate among cashiers and customer shopping threads.

The design of *Supermarket* micro benchmark is shown in Figure 5.2.

The *Supermarket* micro benchmark uses 4 cashier threads to serve 50,000 customers with the size of shopping cart thread pool of 32, which is the number of available logical CPUs on our testing machine.

The pseudo code is shown in Listing 5.3.

Listing 5.3: Supermarket

```
Supermarket(){
  spawn: customer_arrival()
  for (i=0; i<cashier_number; i++)
    spawn: cashier(i)
  while(customer_arrival_queue not empty){
    customer=customer_arrival_queue.take()
    spawn: customer_shopping(customer)
  }
  sync
}

customer_arrival() {
  while(customer_number < maximum_customer) {
    customer=customer_generation()
    customer_arrival_queue.put(customer)
    customer_number++
  }
}

customer_shopping(customer){
```

```
  for(i=0; i<customer.shoppinglist; i++)
    shopping(i);
  customer_checkout_queue.put(customer)
}

cashier() {
  customer=customer_checkout_queue.take(customer)
  for(i=0; i<customer.shoppinglist; i++)
    process(i)
}
```

The complete code of the *Supermarket* micro benchmark is shown in Listing A.3 in Appendix A.

5.1.1.3 Pipeline/Fork-Join Pattern

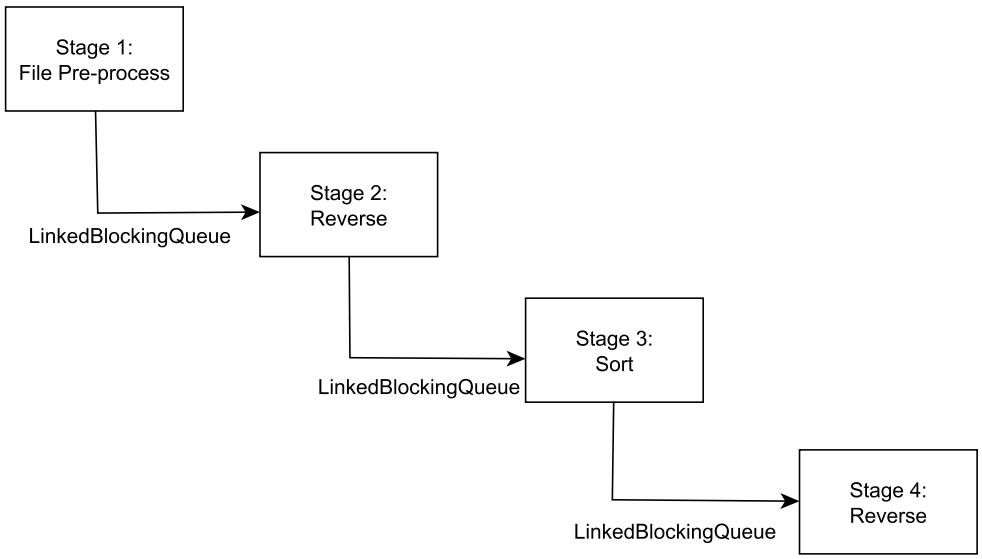

Figure 5.3: Four stages rhyming words pipeline

For Pipeline/Fork-Join pattern, we use a *Rhyming Words* micro benchmark. In this micro benchmark, there are four stages: pre-process file list, reverse each word in a list, sort the words, and then reverse each word again to create a list of rhyming words. These four stages are connected linearly by three pipes implemented by `java.util.concurrent.LinkedBlockingQueue`. The pipeline is shown in Figure 5.3.

Each stage is single threaded without spawning other threads. In the sort stage we use the sequential version of quick sort. We use 2,000,000 files with each having 35 words.

The pseudo code is shown in Listing 5.4.

Listing 5.4: Rhyming Words

```
Rhyming_Words_Pipeline(){
  totalStages = 4;
  stages[0] = File_Preprocess_Stage();
  stages[1] = Reverse1_Stage();
  stages[2] = Sort_Stage();
  stages[3] = Reverse2_Stage();

  //create the queues connecting stages
  for(i=0; i<totalStages-1; i++) {
    queues[i] = BlockingQueue();

  BlockingQueue in = null;
  BlockingQueue out = queues[0];

  //connect stages with queues.
  for(int i=0; i<totalStages; i++) {
    stages[i].init(in, out, s);
    in = out;
    if(i < totalStages-2) out = queues[i+1];
```

```
    else out=null;
  }
  for(i=0; i<totalStages; i++)
    spawn: Stage_Start(i);
  sync
}
```

The complete code of the *Rhyming Words* pipeline micro benchmark is shown in Listing A.5 in Appendix A.

5.1.1.4 Geometric Decomposition/Loop Parallelism Pattern

We use a *Matrix Multiplication* micro benchmark for Geometric Decomposition/Loop Parallelism Pattern.

In the algorithm, we calculate C=A*B by partitioning the matrix C by rows with a thread for each row. We can specify an arbitrary number of threads to partition C. If the dimension of C is not divisible by the number of driver threads, the last thread gets the leftover rows of C.

We use 4 threads for the 10,000*10,000 matrix multiplication.

The pseudo code is shown in Listing 5.5.

Listing 5.5: Matrix Multiplication

```
matrix_multiplication(C, A, B, dimension){
len = dimension/thread_number;
for(i=0; i<thread_number-1; i++){
j = i*len
spawn: multiplication_task(C, A, B, j, j+len);
}
spawn:multiplication_task(C, A, B, (thread_number-1)*len, dimension);
sync
```

```
}
multiplication_task(C, A, B, start, end){
for(int i=start; i<end; i++)
for(int j=0;j<dimension; j++)
for(int k=0; k<dimension; k++)
C[i][j] += A[i][k]*B[k][j];
}
```

The complete code of the *Matrix Multiplication* micro benchmark is shown in Listing A.4 in Appendix A.

5.1.2 Results and Analysis

In Java, JUC is an important tool for programmers to write concurrent programs. The JUC features employed in each micro benchmark are shown in Table 5.1.

Table 5.1: JUC features for micro benchmarks

Concurrency Pattern	**Micro Benchmark**	**JUC Features**
Divide and Conquer/Fork-Join	Maximum Finder	Fork-Join Framework
	Merge Sort	Fork-Join Framework
Event-Based Coordination/Fork-Join	Supermarket	Thread Pool LinkedBlockingQueue
Pipeline/Fork-Join	Rhyming Words	LinkedBlockingQueue CountDownLatch
Geometric Decomposition/Loop Parallelism	Matrix Multiplication	N/A

The micro benchmarks have the following features:

- None of the micro benchmarks spawn their own threads, except *Supermarket* which spawns an additional 32 threads for the thread pool.
- All the micro benchmarks are allowed to use an arbitrary number of driver threads, except *Rhyming Words* which uses a fixed number of four threads for four stages in the pipeline.
- All the micro benchmarks' workloads are configurable.

We characterize concurrency for the micro benchmarks above. Except *Rhyming Words*, the number of driver threads for other micro benchmarks can be configured to an arbitrary number. The corresponding level of concurrency metrics and shared memory metrics are shown in Table 5.2 and Table 5.3.

Table 5.2: Level of concurrency metrics for micro benchmarks

Micro Benchmark	**#T**[1]	**TD**[2]	**PTD**[3]	**PPTD**[4]	**S/S(32)**[5]
Maximum Finder	5	5	4	4	21.6/21.9
Merge Sort	5	4	4	4	14.5/15.2
Supermarket	37	35	33	33	4.46/11.3
Rhyming Words	5	4	4	4	N/A
Matrix Multiplication	5	4	4	4	19.3/25.8

[1] The Number of Spawned User Threads [2] Thread Density
[3] Periodic Thread Density [4] Parallel Periodic Thread Density [5] Actual Maximum Speedup/Predicted Maximum Speedup on the 32-core Target Machine

The level of concurrency metrics in Table 5.2 indicate that all these micro benchmarks are multi-threaded (TD>1) and parallel programs (PTD>1).

Table 5.3: Shared memory metrics for micro benchmarks

Micro Benchmark	**A:R:S:T**[1]	**A%**[2]	**R%**[3]	**S%**[4]	**TD_S**[5]	**PTD_S**[6]	**TD_A**[7]	**PTD_A**[8]	**RR**[9]	**RW**[10]	**RA**[11]
Maximum Finder	55: 31: 87: 14844072	63.22%	35.63%	0.00%	4	4	4	4	10646.1	1556.6	1556.6
Merge Sort	1086: 1397: 2484: 160009993	43.7%	56.2%	0.00%	4	4	4	4	38349.1	9998.8	9998.8
Supermarket	159966: 100546: 261409: 93636083	61.19%	38.46%	0.28%	35	12	36	14	111.0	3.1	2.9
Rhyming Words	204052: 264064: 468143: 1043394	43.59%	56.41%	44.87%	4	4	4	4	22015.8	2262.6	2088.7
Matrix Multiplication	1006: 2004: 3010: 12732	33.42%	66.58%	23.64%	4	4	4	4	207738.3	14837.7	14837.6

[1] Alternating Objects : Shared Read Only Objects : Shared Objects : Total Allocated Objects [2] Percentage of Alternating Objects out of Shared Objects [3] Percentage of Shared Read-only Objects out of Shared Objects [4] Percentage of Shared Objects out of Total Objects
[5] Thread Density for Shared Objects [6] Periodic Thread Density for Shared Objects [7] Thread Density for Alternating Operations
[8] Periodic Thread Density for Alternating Operations [9] Shared Read Rate [10] Shared Write Rate [11] Alternating Operation Rate

For *Maximum Finder*, *Merge Sort*, *Rhyming Words* and *Matrix Multiplication*, all the four driver threads contribute to their workload significantly and concurrently. For *Supermarket*, 35 out of 36 threads (4 driver threads plus 32 threads in the thread pool) contribute to the workload significantly and 33 threads contribute concurrently over the whole periods.

In addition, the actual speedup is close to the predicted maximum speedup. *Rhyming Words* is not suitable for the scalability test as it cannot use an arbitrary number of threads to drive the workload.

The shared memory metrics in Table 5.3 indicate that for all the micro benchmarks, multiple threads contribute to shared memory usage significantly (TD_S>1) and concurrently (PTD_S>1).

Moreover, all the user threads contribute to alternating operations significantly, as their TD_A equals the number of user threads. In addition, all the micro benchmarks pose a challenge to shared memory, as their PTD_A is greater than 1. All the micro benchmarks' user threads contribute to alternating operations concurrently (PTD_A equals to the number of user threads), except *Supermarket* whose PTD_A is less than the number of user threads. According to PTD_A, we can see that the contention on shared objects is quite high in these micro benchmarks. In other words, there are many communications between threads. So none of these micro benchmarks are embarrassingly parallel.

Maximum Finder has the fewest shared objects, under 100; compared with the number of total allocated objects, it is negligible. The other four micro

benchmarks have a large number of shared objects. As for the percentage of shared objects out of total objects, *Rhyming Words* and *Matrix Multiplication* are relatively high with 44.87% and 23.64% respectively, the others are extremely low—less than 0.3%.

Shared ready-only objects do not pose any challenge on shared memory, as multiple threads can read concurrently without any synchronization. In all of these micro benchmarks, the number of shared read-only objects makes up a high proportion, with *Maximum Finder* having the least (35.63%) and *Matrix Multiplication* having the most (66.58%).

These micro benchmarks are shared read dominated, especially for *Supermarket* with 36 times more shared read operations. Shared write rate and alternating operation rate are basically equal.

5.1.3 Micro Benchmark Summary

These micro benchmarks employ different JUC features such as Fork-Join Framework, Thread Pool, etc.

All these micro benchmarks are multi-threaded and parallel. The predicted speedup is close to the actual speedup.

In these micro-benchmarks, multiple threads contribute to shared memory usage significantly and concurrently. These micro benchmarks pose a challenge to shared memory, and multiple threads contribute to object ownership change operations concurrently. None of them are embarrassingly parallel.

For *Maximum Finder, Merge Sort* and *Supermarket*, the percentage of shared

objects out of total objects is extremely low. All these micro benchmarks are shared read dominated.

5.2 DaCapo Measurement

5.2.1 Introduction

DaCapo [23] measures execution time of each sub-benchmark based on operations done over a fixed workload. However DaCapo's workload is not arbitrarily configurable. It has only three sizes: small, default and large.
DaCapo 2009 supports scaling of seven benchmarks (*h2, lusearch, sunflow, tomcat, tradebeans, tradesoap, xalan*) to an arbitrary number of driver threads. So these sub-benchmarks are suitable for the scalability tests to get the maximum speedup. However, *tradesoap* and *tradebeans* have exceptions in our tests. In *h2*, instead of speedup, there is slow down because of a highly contended global lock, which is a defect in the H2 database engine.
The following sub-benchmarks in the DaCapo suite are used in our experiments:

- **batik:** A Scalable Vector Graphics (SVG) images tool based on the unit tests in Apache Batik.
- **eclipse:** An Eclipse IDE (non-gui) performance test tool.
- **fop:** An XSL-FO file parser and formatter.

- **jython:** A Python interpreter.
- **luindex:** A text index tool using Apache Lucene which is a text search engine library.
- **lusearch:** A text search tool using Apache Lucene.
- **pmd:** A Java classes analyzer.
- **sunflow:** An image render using ray tracing.
- **tomcat:** A web service simulator using Tomcat.
- **xalan:** An XML to HTML transformer.

5.2.2 Results and Analysis

We use four threads to drive the workload for *lusearch, sunflow, tomcat, xalan* sub-benchmarks to characterize concurrency. The results are shown in Table 5.4 and Table 5.5

Table 5.4: Level of concurrency metrics for DaCapo 1

Sub-benchmark	**#T**[1]	**TD**[2]	**PTD**[3]	**PPTD**[4]	**S/S(32)**[5]
sunflow	12	4	4	4	19.5/28.1
xalan	5	4	4	4	23.1/30.2
lusearch	5	5	4	4	10.6/11.2
tomcat	52	10	9	9	7.6/8.2

[1] The Number of Spawned User Threads [2] Thread Density
[3] Periodic Thread Density [4] Parallel Periodic Thread Density [5] Actual Maximum Speedup/Predicted Maximum Speedup on the 32-core Target Machine

Table 5.5: Shared memory metrics for DaCapo

Sub-benchmark	**A:R:S:T**[1]	**A%**[2]	**R%**[3]	**S%**[4]	**TD_S**[5]	**PTD_S**[6]	**TD_A**[7]	**PTD_A**[8]	**RR**[9]	**RW**[10]	**RA**[11]
sunflow	44: 4972: 5019: 26527010	0.88%	99.06%	0.02%	4	4	4	1	30528.1	0.5521	0.5521
xalan	17563: 4445: 22015: 6527010	79.78%	20.19%	0.33%	4	4	4	3	6790.5	104.9	466.3
lusearch	76: 603: 698: 26542021	10.89%	86.39%	0.00%	4	4	4	1	449.1	0.1126	0.1122
tomcat	75786: 100197: 229048: 63007547	33.09%	43.74%	0.36%	9	9	10	6	3984.9	477.8	482.4

[1] Alternating Objects : Shared Read Only Objects : Shared Objects : Total Allocated Objects [2] Percentage of Alternating Objects out of Shared Objects [3] Percentage of Shared Read-only Objects out of Shared Objects [4] Percentage of Shared Objects out of Total Objects
[5] Thread Density for Shared Objects [6] Periodic Thread Density for Shared Objects [7] Thread Density for Alternating Operations
[8] Periodic Thread Density for Alternating Operations [9] Shared Read Rate [10] Shared Write Rate [11] Alternating Operation Rate

From Table 5.4, we can see that all these four sub-benchmarks are multi-threaded (TD>1) and parallel (PTD>1). Sub-benchmarks *tomcat* and *sunflow* spawn additional threads from the specified four driver threads, as the number of spawned user threads is greater than five (four driver threads plus a main thread). *tomcat* spawns 52 threads, but only 10 contribute significantly, and nine contribute concurrently.

For *lusearch* and *tomcat*, the predicted maximum speedup is quite close to the actual speedup on our testing machine. For *sunflow* and *xalan*, the predicted maximum speedup is a little far from the actual speedup, but still complies with Amdahl's law.

From Table 5.5, we can see that multiple threads contribute to shared memory usage significantly (TD_S>1) and concurrently (PTD_S>1).

Moreover, multiple threads contribute to alternating operations significantly (TD_A>1). But for *sunflow* and *lusearch*, only one thread contributes to alternating operations concurrently (PTD_A=1), which means the contention on shared objects is quite low. So they are candidates for the embarrassingly parallel pattern. For *xalan* and *tomcat*, multiple threads contribute to alternating operations concurrently (PTD_A>1), which means the contention on shared objects is high.

For all these four sub-benchmarks, the percentage of shared objects out of total objects is extremely low—less than 0.4%. Moreover, the percentage of shared read-only objects out of shared objects is quite high for *sunflow* and *lusearch*—99.06% and 86.39% respectively. Meanwhile, for *sunflow* and

lusearch, the percentage of alternating objects out of shared objects and the alternating operation rate are also quite low. To be specific, for *sunflow*, 0.88% shared objects have ownership change operations and 0.5521 ownership change operations per millisecond, and for *lusearch*, 10.89% shared objects have ownership change operations and 0.1122 ownership change operations per millisecond.

All these four sub-benchmarks are shared read dominated. To be specific, *sunflow* has the highest, 55,300 times more shared read than shared write and alternating operations, and *tomcat* has the lowest, 8.26 times more shared read than shared write and alternating operations. The *tomcat* sub-benchmark allocates the most objects and has the most shared objects and alternating objects and it also has the fastest shared write rate and alternating operation rate. The *xalan* sub-benchmark allocates the fewest objects, but the highest percentage of alternating objects, and the alternating operation rate is quite high. The *lusearch* sub-benchmark has the fewest shared objects and *sunflow* has the fewest alternating objects.

Sub-benchmark *eclipse, luindex, pmd, batik, jython* and *fop* in the DaCapo suite do not support using arbitrary numbers of threads to the drive their workloads. Their level of concurrency metrics are shown in Table 5.6.

From Table 5.6, we can see that *eclipse, luindex, pmd, batik, jython* and *fop* are multi-threaded (TD>1) but non-parallel (PTD=1). Using the number of spawned threads to measure concurrency is a poor metric. The number of threads performing a meaningful amount of work is small and even smaller is

Table 5.6: Level of concurrency metrics for DaCapo 2

Sub-benchmark	**#T**[1]	**TD**[2]	**PTD**[3]	**PPTD**[4]
eclipse	851	23	1	2
luindex	2	2	1	0
pmd	34	31	1	29
batik	9	2	1	0
jython	3	1	1	0
fop	2	1	1	0

[1] The Number of Spawned User Threads [2] Thread Density
[3] Periodic Thread Density [4] Parallel Periodic Thread Density

the number of threads doing that concurrently. For example, *eclipse* spawns a large number of 851 threads, but only 23 threads contribute significantly and only one thread contributes concurrently. So in *eclipse*, the majority of threads do not have enough work to do, and basically these threads are running sequentially. For *pmd*, 31 out 34 threads contribute to the workload significantly, but only one thread contributes concurrently. Its PPTD is 29, which means 29 threads working concurrently in the parallel periods. However, its PTD is 1 which means the majority of periods are sequential. Even though there are 29 threads working concurrently in parallel periods, there are few such periods. For *luindex, batik, jython, fop*, their concurrency level is extremely low, as PPTD is 0, which means the parallel periods are few or there is no parallel period at all.

Since these sub-benchmarks are actually running sequentially (PTD=1), there is no need to characterize their shared memory behavior. They are not expected to scale on multi-cores.

5.2.3 DaCapo Summary

In the DaCapo benchmark suite, *sunflow, xalan, lusearch* and *tomcat* are multi-threaded and parallel. In these four parallel sub-benchmarks, multiple threads contribute to shared memory significantly and concurrently. Threads in *sunflow* and *lusearch* do not have too much communication, so they are candidates for the embarrassingly parallel pattern. The percentage of shared objects out of total objects is extremely low in these benchmarks and they are shared read dominated.

Sub-benchmark *eclipse, luindex, pmd, batik, jython* and *fop* are multi-threaded but non-parallel.

5.3 SPECjvm2008 Measurement

5.3.1 Introduction

SPECjvm2008 [24] supports using an arbitrary number of threads to drive the workload. By default, SPECjvm2008 measures throughput by scoring each benchmark on operations done over a fixed time rather than elapsed time for a fixed workload. To characterize concurrency for the SPECjvm2008 suite, we need some special settings. Since we need to measure elapsed time for a fixed workload, we use the `-ops` option to specify how many operations in an iteration, which makes it a fixed workload. By default, SPECjvm2008 generates a raw result file, a text report and a html report

when benchmarking finishes, which has heavy disk operations. We disable them with the option `-crf 0 -ctf 0 -chf 0`.
The following sub-benchmarks in the SPECjvm2008 suite are used in our experiments:

- **compress:** A compress tool using modified LZW algorithm.
- **crypto:** A cryptographic tool with three sub benchmarks: *aes*, *rsa* and *signverify*.
- **derby:** A tool using an open source database stressing the BigDecimal library.
- **mpegaudio:** An mp3 decoder with floating-point heavy operations.
- **serial:** A serialization and deserialization tool using data from the JBoss benchmark.
- **sunflow:** A graphics visualization tool using an open source global illumination system.
- **xml:** A tool that exercises JRE's implementation of *javax.xml.transform* and *javax.xml.validation*.

5.3.2 Results and Analysis

The level of concurrency metrics are shown in Table 5.7. The shared memory metrics are shown in Table 5.8 and Table 5.9.

Table 5.7: Level of concurrency metrics for SPECjvm2008

Sub-benchmark	**#T**[1]	**TD**[2]	**PTD**[3]	**PPTD**[4]	**S/S(32)**[5]
serial	6	4	4	4	9.5/28.5
compress	6	4	4	4	19.4/18.2
xml.validation	6	4	4	4	17.1/25.6
xml.transform	6	5	4	4	15.7/3.89
crypto.signverify	6	4	4	4	18.4/21.8
crypto.rsa	6	4	4	4	12.1/15.9
crypto.aes	6	4	4	4	18.8/20.7
sunflow	328	152	16	16	5.8/28.0
derby	10	5	4	4	17.5/2.29
mpegaudio	6	4	4	4	20.3/31.9

[1] The Number of Spawned User Threads [2] Thread Density
[3] Periodic Thread Density [4] Parallel Periodic Thread Density [5] Actual Maximum Speedup/Predicted Maximum Speedup on the 32-core Target Machine

According to Table 5.7, all these sub-benchmarks are multi-threaded (TD>1) and parallel (PTD>1). According to SPECjvm2008's description, the benchmark launcher creates an additional service thread for each sub-benchmark. Furthermore, according to our trace file, the service thread only does a tiny amount of work, as the functionality of the result file generation is disabled. Thus, *serial, compress, xml.validation, xml.transform, crypto.signverify, crypto.rsa, crypto.aes* and *mpegaudio* do not spawn additional threads for their workload, as the number of spawned threads is six. Sub-benchmark *derby* spawns four additional threads, and *sunflow* spawns 322 additional threads. Except *sunflow*, other sub-benchmarks' driver threads contribute to their workload significantly (TD>4) and currently (PTD=4). For *sunflow*, 328 threads are spawned, but only 152 threads do a meaningful amount of work,

Table 5.8: Shared memory metrics for SPECjvm2008(1)

Sub-benchmark	**A:R:S:T**[1]	**A%**[2]	**R%**[3]	**S%**[4]	**TD_S**[5]	**PTD_S**[6]	**TD_A**[7]	**PTD_A**[8]	**RR**[9]	**RW**[10]	**RA**[11]
serial	575: 5689: 6336: 6216288130	9.07%	89.79%	0.00%	4	4	4	3	5685.8	179.7	212.1
compress	105: 644: 767: 88269	13.69%	83.96%	0.87%	5	0	5	0	0.0909	0.0015	0.0015
xml. validation	2672: 4611: 7350: 81905375	36.35%	62.73%	0.01%	4	4	4	4	32640.8	5110.7	5110.8
xml. transform	4471: 12456: 16962: 2107437900	26.36%	73.43%	0.00%	4	4	4	3	13669.3	3091.7	3091.7
sunflow	441: 21867: 22471: 86939618	1.96%	97.31%	0.03%	16	16	20	1	11182.5	0.1158	0.1155

[1] Alternating Objects : Shared Read Only Objects : Shared Objects : Total Allocated Objects [2] Percentage of Alternating Objects out of Shared Objects [3] Percentage of Shared Read-only Objects out of Shared Objects [4] Percentage of Shared Objects out of Total Objects
[5] Thread Density for Shared Objects [6] Periodic Thread Density for Shared Objects [7] Thread Density for Alternating Operations
[8] Periodic Thread Density for Alternating Operations [9] Shared Read Rate [10] Shared Write Rate [11] Alternating Operation Rate

Table 5.9: Shared memory metrics for SPECjvm2008(2)

Sub-benchmark	**A:R:S:T**[1]	**A%**[2]	**R%**[3]	**S%**[4]	**TD_S**[5]	**PTD_S**[6]	**TD_A**[7]	**PTD_A**[8]	**RR**[9]	**RW**[10]	**RA**[11]
crypto. signverify	213: 8599: 8861: 13106824	2.40%	97.04%	0.07%	4	2	4	1	2544.4	0.1308	0.1561
crypto.rsa	214: 28010: 28308: 61296925	0.76%	98.95%	0.05%	4	4	3	1	5932.7	2.7	2.8
crypto.aes	209: 2440: 2797: 307113	7.47%	87.24%	0.91%	4	3	5	1	8041.8	0.0071	0.0073
derby	174127: 337008: 511771: 1086132585	34.02%	65.85%	0.05%	5	4	5	4	12339.7	1322.9	1323.3
mpegaudio	215: 4425: 4686: 1155340	4.59%	94.43%	0.41%	4	4	5	1	36502.0	0.0019	0.0019

[1] Alternating Objects : Shared Read Only Objects : Shared Objects : Total Allocated Objects [2] Percentage of Alternating Objects out of Shared Objects [3] Percentage of Shared Read-only Objects out of Shared Objects [4] Percentage of Shared Objects out of Total Objects
[5] Thread Density for Shared Objects [6] Periodic Thread Density for Shared Objects [7] Thread Density for Alternating Operations
[8] Periodic Thread Density for Alternating Operations [9] Shared Read Rate [10] Shared Write Rate [11] Alternating Operation Rate

and only 16 threads do that concurrently.

For *compress, xml.vlidation, crypto.sigverify, crypto.rsa, crypto.aes, mpegaudio*, the actual speedup matches the predicted speedup. For *serial* and *sunflow* the actual speedup is much less than the predicted speedup. For *derby* and *xml.transform* the actual speedup is far more than the predicted speedup. We measure the speedup according to the execution time reported by SPECjvm2008, which is a black-box to us. It seems that SPECjvm2008 only reports a partial time for the workload. But our tool predicts the speedup based on the whole workload, excluding VM initialization and shutdown. If SPECjvm2008 provided a better time measurement of the whole workload from start to end—like DaCapo does, we might not get an inconsistency of Amdahl's law.

According to Table 5.8 and Table 5.9, we can see that multiple threads contribute to shared memory significantly (TD_S>1). Except for *compress*, multiple threads contribute to shared memory concurrently (PTD_S>1).

Moreover, multiple threads contribute to alternating operations significantly (TD_A>1). But for *compress, sunflow, crypto.signverify, crypto.rsa, crypto.aes* and *mpegaudio*, only one or zero threads contribute to alternating operations concurrently (PTD_A<=1), which means the contention on shared objects is quite low in these sub-benchmarks. There is not too much communication between threads, so these six sub-benchmarks are candidates for the embarrassingly parallel pattern. For these sub-benchmarks, the rate of alternating operations is quite low, ranging from 0.0015 to 2.8. For *serial,*

xml.validation, xml.transform, derby, threads have more communications, as PTD_A is greater than 1. For these sub-benchmarks, the rate of alternating operations is much higher than that of those sub-benchmarks with PTD_A less than 1, ranging from 212.1 to 5110.8.

For all these sub-benchmarks, the percentage of shared objects out of the total objects is quite low, with a maximum of 0.91%. Moreover, the percentage of shared read-only objects out of shared objects is quite high, with *crypto.rsa* having the most shared read-only objects, 98.95%, and *xml.validation* having the fewest shared read-only objects, 62.73%. All these sub-benchmarks are shared read dominated. To be specific, *mpegaudio* has the most, 19,211,578 times more shared read than shared write and alternating operations and *xml.transform* has the fewest 4.42 times more shared read than shared write and alternating operations.

5.3.3 SPECjvm2008 Summary

Sub-benchmarks *serial, compress, xml.validation, xml.transform, crypto.signverify, crypto.rsa, crypto.aes, sunflow, derby* and *mpegaudio* in the SPECjvm2008 benchmark suite are multi-threaded and parallel. In these sub-benchmarks, multiple threads contribute to shared memory usage significantly and concurrently, except for *compress* where no thread contributes to shared memory usage concurrently.

In sub-benchmarks *compress, sunflow, crypto.signverify, crypto.rsa, crypto.aes* and *mpegaudio*, there is not too much communication between threads, so

they are candidates for the embarrassingly parallel pattern. The percentage of shared objects out of total objects is extremely low in the SPECjvm2008 benchmark suite. All the sub-benchmarks are shared read dominated.

5.4 Summary

In this chapter we first presented micro benchmarks with different concurrency patterns. Then we characterized concurrency using the metrics presented in Chapter 3 for these micro benchmarks, as well as two widely used benchmarks: SPECjvm2008 and DaCapo benchmark suites.

The level of concurrency metrics measure how parallel a Java program is. With these metrics we can do the following things:

- Measure how many threads contribute to the workload significantly according to TD.
- Measure how many threads contribute to the workload concurrently accordng to PTD.
- Identify multi-threaded and parallel programs (TD>1, PTD>1).
- Identify multi-threaded but non-parallel programs (TD>1, PTD<=1).
- Identify multi-threaded but concurrency level is extremely low programs (TD>1, PPTD<=1).
- Predict maximum speedup according to S(N).

The shared memory metrics measure how a Java program uses shared memory. With these metrics we can do the following things:

- Measure how many threads contribute to shared memory significantly according to TD_S.
- Measure how many threads contribute to shared memory concurrently according to PTD_S.
- Measure how many threads contribute to the object ownership changing operations significantly according to TD_A.
- Measure how many threads contribute to the object ownership changing operations concurrently to PTD_A.
- Identify embarrassingly parallel programs according to PTD_A.

Using the number of spawned threads to measure concurrency is a poor metric. The number of threads performing a meaningful amount of work is small, and even smaller is the number of threads doing that concurrently. This can be explained in the following three equations:

$$\#T \geq TD \geq PTD \tag{5.1}$$

$$\#T \geq TD\_S \geq PTD\_S \tag{5.2}$$

$$\#T \geq TD\_A \geq PTD\_A \tag{5.3}$$

Kalibera et al. and we both measured TD, PTD, TD_S and PTD_S for the DaCapo benchmark suite using four driver threads [13]. They also reported the number of spawned threads, which includes JVM service threads. For TD and TD_S, our result is equal to or lower than the number they reported, as we exclude JVM service threads and we use CPU time to measure a thread's contribution. For PTD and PTD_S, we achieved the same results as Kalibera et al.

Both of us came to the same conclusion that *eclipse, luindex, pmd, batik, jython* and *fop* in the DaCapo benchmark suite are not parallel, *sunflow, xalan, lusearch* and *tomcat* are parallel.

But we provided better understanding of the concurrency behavior of a Java program by the following insights:

- More precise value for metrics with thread density in their names, as JVM service threads are excluded.
- Further determine concurrency level for multi-threaded, non-parallel programs (TD>1, PTD<=1) according to their PPTD metrics.
- Predict maximum speedup.
- Identify embarrassingly parallel programs according to PTD_A.
- Measure how many objects are allocated and shared, how many objects

are shared read only and how many objects have ownership changing operations.

Moreover, the total number of allocated objects is extremely huge, and the percentage of shared objects out of total allocated objects is extremely small —in the SPECjvm2008 and DaCapo benchmark suites, it is less than 1%. According to Jendrosch's escape analysis [10, 11], the percentage of sharable objects (escape rate) of the majority of classes is relatively low, whereas some classes like *String* have almost 100% escape rate. Kalibera et al. traced all objects' access operations. But in our concurrency metrics tool, for the level of concurrency metrics, we use CPU time to measure a thread's contribution, and for shared memory metrics we only trace shared objects' access operations. Since the percentage of shared objects is quite low, the cost is relatively lower than that in Kalibera et al.'s work [13].

Chapter 6

Conclusions

6.1 Overview

The goal of this thesis was to characterize concurrency for Java programs using a range of concurrency related metrics instead of only execution time and the number of spawned threads.

First, we introduced a set of metrics used to characterize concurrency for Java programs. These metrics are divided into two groups: level of concurrency metrics and shared memory metrics. The level of concurrency metrics measure how parallel a Java program is, including the following metrics: The Number of Spawned User Threads (#T), Thread Density (TD), Periodic Thread Density (PTD), Parallel Periodic Thread Density (PPTD) and Maximum Speedup (S(N)). The shared memory metrics measure how a Java program uses shared memory, including the following metrics: Thread Den-

sity for Shared Objects (TD_S), Periodic Thread Density for Shared Objects (PTD_S), Thread Density for Alternating Operations (TD_A), Periodic Thread Density for Alternating Operations (PTD_A), Shared Read Rate (RR), Shared Write Rate (RW) and Alternating Operation Rate (RA).

Then, we presented a tool based on the J9 JVM that is used to obtain these metrics. To be specific, a JVMTI agent is developed to obtain the level of concurrency metrics. The J9 JVM is instrumented by extending the VMObject model to capture the shared memory metrics. We also discussed the corresponding changes required to port these metrics to other JVMs and other platforms.

Finally, we used these metrics to characterize concurrency for micro benchmarks with different concurrency patterns: Divide and Conquer/Fork-Join, Geometric Decomposition/Loop Parallelism, Event-Based Coordination/Fork-Join, Pipeline/Fork-Join. We also characterized concurrency for SPECjvm2008 and DaCapo benchmark suites.

In conclusion, we can characterize concurrency by the metrics obtained by our tool. Compared with execution time and the number of spawned threads, these metrics give us a better understanding of the concurrency behaviour of the Java programs. From these metrics, we can know whether a multi-threaded Java program is really parallel, and how parallel it is. In addition, for a concurrent Java program, we can predict the maximum speedup and identify whether it is embarrassingly parallel.

6.2 Future Work

So far, we only have 12 metrics to characterize concurrency of Java programs. There are further metrics that should be considered. For example, the locality of reference, as programs with stronger locality may exhibit higher performance, due to fewer cache misses, independent of the number of threads issued. Other metrics like the primary data structures and types, and dynamic memory use should also be considered. Then we can apply a feature selection strategy to select the subset of these metrics that can be used to characterize concurrency and differentiate different Java concurrency patterns.

We realized our metrics tool on the J9 JVM on Linux, so another enhancement could be porting these metrics to other platforms and JVMs.

Bibliography

[1] Gene M Amdahl, *Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities, Reprinted from the AFIPS Conference Proceedings, Vol. 30 (Atlantic City, NJ, Apr. 18–20), AFIPS Press, Reston, Va., 1967, pp. 483–485, when Dr. Amdahl was at International Business Machines Corporation, Sunnyvale, California*, Solid-State Circuits Society Newsletter, IEEE **12** (2007), no. 3, 19–20.

[2] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, et al., *The Landscape of Parallel Computing Research: A View from Berkeley*, Tech. report, Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[3] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al., *The DaCapo Benchmarks:*

Java Benchmarking Development and Analysis, ACM SIGPLAN Notices, vol. 41, ACM, 2006, pp. 169–190.

[4] Thomas H Cormen, *Introduction to Algorithms*, Third Edition, MIT press, pp. 797-805, 2009.

[5] Bruno Dufour, Karel Driesen, Laurie Hendren, and Clark Verbrugge, *Dynamic Metrics for Java*, ACM SIGPLAN Notices, vol. 38, ACM, 2003, pp. 149–168.

[6] Brian Göetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea, *Java Concurrency in Practice*, Addison-Wesley, 2006.

[7] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley, *The Java Language Specification, Java SE 7 Edition*, 2013.

[8] IBM, *J9 Virtual Machine (JVM)*, `http://pic.dhe.ibm.com/infocenter/java7sdk/v7r0/index.jsp?topic=%2Fcom.ibm.java.lnx.70.doc%2Fuser%2Fjava_jvm.html`.

[9] Java Virtual Machine Tool Interface, *JVMTI*, `http://w3.hursley.ibm.com/java/docs/java7/technotes/guides/jvmti/index.html`.

[10] Manfred Jendrosch, *Master's Thesis: Runtime Escape Analysis in a Java Virtual Machine*, Fredericton, University of New Brunswick, 2013.

[11] Manfred Jendrosch, Gerhard W Dueck, Charlie Gracie, and André Hinkenjann, *PC Based Escape Analysis in the Java Virtual Machine*, Lecture Notes on Software Engineering **2** (2014), no. 1, 16.

[12] Jikes, `http://www.jikesrvm.org/`.

[13] Tomas Kalibera, Matthew Mole, Richard Jones, and Jan Vitek, *A Black-box Approach to Understanding Concurrency in DaCapo*, Proceedings of the ACM international conference on Object oriented programming systems languages and applications, ACM, 2012, pp. 335–354.

[14] Kenai, *Btrace*, `https://kenai.com/projects/btrace`.

[15] Kurt Keutzer and Tim Mattson, *A Design Pattern Language For Engineering (Parallel) Software*, Intel Technology Journal **13** (2010), no. 4.

[16] Timothy G Mattson, Beverly A Sanders, and Berna L Massingill, *Patterns for Parallel Programming*, Pearson Education, 2004.

[17] Anne Meade, Jim Buckley, and JJ Collins, *Challenges of Evolving Sequential to Parallel Code: an Exploratory Review*, Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution, ACM, 2011, pp. 1–5.

[18] MPI, `http://www.mcs.anl.gov/research/projects/mpi/`.

[19] OpenMP, `http://openmp.org/wp/`.

[20] Oracle, *Java SE HotSpot at a Glance*, http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136373.html.

[21] OW2, *Asm*, http://asm.ow2.org/.

[22] Java Package, *java.util.concurrent*, http://w3.java.ibm.com/java/docs/java7/api/java/util/concurrent/package-summary.html.

[23] DaCapo research project, *Dacapo benchmark suite*, http://www.dacapobench.org/.

[24] The Standard Performance Evaluation Corporation (SPEC), *Java client/server benchmarks*, http://spec.org/benchmarks.html\#java.

Appendix A

Micro Benchmarks

Listing A.1: Maximum Number Finder

```
public class MaximumFinder extends RecursiveTask<Integer> {
  private static final int SEQUENTIAL_THRESHOLD = 10;
  private final int[] data;
  private final int start;
  private final int end;

  public MaximumFinder(int[] data, int start, int end) {
    this.data = data;
    this.start = start;
    this.end = end;
  }

  public MaximumFinder(int[] data) {
    this(data, 0, data.length);
  }

  @Override
  protected Integer compute() {
    final int length = end - start;
```

```
    if (length < SEQUENTIAL_THRESHOLD) {
      return computeDirectly();
    }
    final int split = (start + end) / 2;
    final MaximumFinder left = new MaximumFinder(data, start, split);
    left.fork();
    final MaximumFinder right = new MaximumFinder(data, split, end);
    return Math.max(right.compute(), left.join());
  }

  private Integer computeDirectly() {
    int max = Integer.MIN_VALUE;
    for (int i = start; i < end; i++) {
      if (data[i] > max) {
      max = data[i];
      }
    }
    return max;
  }

  public static void main(String[] args) {
    if (args.length != 2) {
      System.out.println("Usage: MaximumFinder problem_size threads");
      System.exit(1);
    }

    final int size = Integer.parseInt(args[0]);
    final int threads = Integer.parseInt(args[1]);

    // create a random data set
    final int[] data = new int[size];
    final Random random = new Random(759123751834L);

    for (int i = 0; i < size; i++)
      data[i] = random.nextInt();

    // submit the task to the pool
```

```
    final ForkJoinPool pool = new ForkJoinPool(threads);
    MaximumFinder finder = new MaximumFinder(data);
    long start = System.currentTimeMillis();
    pool.invoke(finder);
    long time = System.currentTimeMillis() - start;
    System.out.println("size:"+size+",threads:"+threads+","+time);
  }
}
```

Listing A.2: Merge Sort

```
public class MergeSort {
  private final ForkJoinPool pool;
  private  int[] array;
  private  int[] result;

  private  class MergeSortTask extends RecursiveAction {
    private  int p;
    private  int r;
    private  int s;
    private  int[] B;
    protected MergeSortTask(int p, int r, int[] B, int s) {
      this.p = p;
      this.r = r;
      this.s = s;
      this.B = B;
    }

    @Override
    protected void compute() {
      int n = r-p+1;
      if(n == 1)
        B[s]= array[p];
      else {
        int[] T = new int[n];
        int q = (p+r)/2;
        int q2 = q-p +1;
```

```
      MergeSortTask left = new MergeSortTask(p, q,T,0);
      MergeSortTask right = new MergeSortTask(q+1, r, T, q2);
      invokeAll(left, right);
      left.join();
      right.join();
      par_merge(T, 0, q2, q2+1, n-1, B, s);
    }
  }

  private void par_merge( int[] T, int p1, int r1,
                          int p2, int r2, int[] B,  int p3){
    ParMergeTask parmerge=new ParMergeTask(T, p1, r1, p2, r2, B, p3);
    pool.invoke(parmerge);
  }
}

private  class ParMergeTask extends RecursiveAction {
  int p1, r1, p2, r2, p3;;
  int T[];
  int B[];
  public ParMergeTask(int[] T, int p1, int r1, int p2,
                      int r2, int[] B, int p3) {
    this.p1=p1;
    this.r1=r1;
    this.p2=p2;
    this.r2=r2;
    this.p3=p3;
    this.T=T;
    this.B=B;
  }

  @Override
  protected void compute() {
    int n1=r1-p1+1;
    int n2=r2-p2+1;
    if(n1<n2){
      int temp=n1; n1=n2; n2=temp;
```

```
      temp=p1; p1=p2; p2=temp;
      temp=r1; r1=r2; r2=temp;
    }

    if (n1<=0)
      return;
    else {
      int q1=(p1+r1)/2;
      int q2=binary_search(T[q1], p2, r2);
      int q3= p3+(q1-p1)+(q2-p2);
      B[q3] = T[q1];
      ParMergeTask left=new ParMergeTask(T,p1,q1-1,p2,q2-1,B,p3);
      ParMergeTask right=new ParMergeTask(T,q1+1,r1,q2,r2,B,q3+1);
      invokeAll(left, right);
      left.join();
      right.join();
    }
  }

  private int binary_search(int value, int start, int end) {
    int low =  start;
    int high = start>(end+1)?start:(end+1);
    while ( low<high) {
     int mid = (low+high)/2;
     if(value<=T[mid]) high=mid;
     else low=mid+1;
    }
    return high;
  }
}

public MergeSort(int parallelism) {
  pool = new ForkJoinPool(parallelism);
}

public void sort() {
  ForkJoinTask<Void> job =
```

```
        pool.submit(new MergeSortTask(0, array.length-1,result,0));
    job.join();
  }

  public void generateArray(int elements) {
    Random random = new Random(759123751834L);
    array = new int[elements];
    result = new int[elements];
    for (int i = 0; i < elements; ++i)
    array[i] = random.nextInt(100000);
  }

  public static void main(String[] args) {
    if (args.length != 2) {
      System.out.println("Usage: MergeSort problem_size threads");
      System.exit(1);
    }
    final int size = Integer.parseInt(args[0]);
    final int threads = Integer.parseInt(args[1]);
    MergeSort mergeSort = new MergeSort(threads);
    mergeSort.generateArray(size);
    long start = System.currentTimeMillis();
    mergeSort.sort();
    long time = System.currentTimeMillis() - start;
    System.out.println("size:"+size+",threads:"+threads+","+time);
  }
}
```

Listing A.3: Supermarket

```
class Event_customer {
  private int shoppinglist;
  Event_customer(int list) {
    shoppinglist = list;
  }
  public int getShoppinglist() {
    return shoppinglist;
```

```
  }
}

class Customer_Gen extends Thread {
  int customer_number;
  LinkedBlockingQueue<Event_customer> _queue;
  Customer_Gen(int customer,
               LinkedBlockingQueue<Event_customer> queue) {
    customer_number = customer;
    _queue=queue;
  }
  private int poisson(int lamda){
    int k = 0;
    double b=1, c=Math.exp(-lamda), u;
    do {
      u=Math.random();
      b *=u;
      if(b>=c)
      k++;
    }while(b>=c);
    return k;
  }
  @Override
  public void run() {
    int min = 100;
    int max = 1000;
    Random random = new Random(759123751834L);
    int total_customer = 0;
    while (total_customer < customer_number) {
      int customer_arrived = poisson(100);
      for (int i = 0; i < customer_arrived; i++)
        _queue.put(new Event_customer(
                        random.nextInt(max) % (max - min + 1) + min));
      total_customer += customer_arrived;
      sleep(100);
   }
    _queue.put(new Event_customer(0));
```

```
  }
}

class CustomerShopping implements Runnable {
  LinkedBlockingQueue<Event_customer> _queue_arrvial;
  LinkedBlockingQueue<Event_customer> _queue_checkout;
  Event_customer _event;
  public CustomerShopping(
        LinkedBlockingQueue<Event_customer> queue_arrival,
        LinkedBlockingQueue<Event_customer> queue_checkout,
        Event_customer event){
    _queue_arrvial = queue_arrival;
    _queue_checkout = queue_checkout;
    _event = event;
  }
  private void single_item_shopping() {
    Random ran = new Random();
    int[] ran_list =new int[10];
    for(int i=0; i<10; i++)
      ran_list[i]=ran.nextInt();
  }
  @Override
  public void run() {
    int list = _event.getShoppinglist();
    for(int i=0; i<list; i++)
      single_item_shopping();
    _queue_checkout.put(_event);//done shopping
  }
}

class Cashier extends Thread {
  LinkedBlockingQueue<Event_customer> _queue;
  int _id;
  public Cashier(LinkedBlockingQueue<Event_customer> queue, int id) {
    _queue = queue;
    _id = id;
  }
```

```
  private void process() {
    Random ran = new Random();
    int[] ran_list =new int[10];
    for(int i=0; i<10; i++)
      ran_list[i]=ran.nextInt();
  }
  @Override
  public void run() {
    while (true) {
      Event_customer event = _queue .take();
      int list = event.getShoppinglist();
      if (list == 0)
        break;
      for(int i=0; i<list; i++)
        process();
    }
  }
}

public class Supermarket {
  LinkedBlockingQueue<Event_customer> customer_arrival_queue = null;
  LinkedBlockingQueue<Event_customer> checkout_queue = null;
  Cashier[] _cashiers = null;
  Customer_Gen customer_event_gen=null;
  ExecutorService executor = Executors.
    newFixedThreadPool( Runtime.getRuntime().availableProcessors());
  Supermarket() {
    customer_arrival_queue = new LinkedBlockingQueue<Event_customer>();
    checkout_queue = new LinkedBlockingQueue<Event_customer>();
  }
  public void start(int cashier_number, int customer_number)
                                    throws InterruptedException {
    customer_event_gen = new Customer_Gen(customer_number,
                                          customer_arrival_queue);
    customer_event_gen.start();
    _cashiers = new Cashier[cashier_number];
    for (int i = 0; i < cashier_number; i++) {
```

```
      _cashiers[i] = new Cashier(checkout_queue, i);
      _cashiers[i].start();
    }
    while (true){
      Event_customer event = customer_arrival_queue.take();
      if(event.getShoppinglist() == 0)
        break;
      Runnable shoppingcart = new CustomerShopping(
                      customer_arrival_queue, checkout_queue, event);
      executor.execute(shoppingcart);
    }
    for(int i=0; i<cashier_number; i++)
    checkout_queue.put(new Event_customer(0));
    executor.shutdown();
    customer_event_gen.join();
    for (int i = 0; i < cashier_number; i++)
      _cashiers[i].join();
  }

  public static void main(String[] args) throws InterruptedException {
    if (args.length != 2) {
      System.out.println("Usage: Supermarket cashier# consumer# ");
      System.exit(1);
    }
    final int cashier_number = Integer.parseInt(args[0]);
    final int customer_number = Integer.parseInt(args[1]);
    Supermarket myMarket = new Supermarket();
    long start = System.currentTimeMillis();
    myMarket.start(cashier_number, customer_number);
    long diff = System.currentTimeMillis() - start;
    System.out.println(cashier_number + "Cashier:" + diff+" msec");
  }
}
```

Listing A.4: Matrix Multiplication

```
public class MatrixMultiplication {
```

```
int dimension;
int threads;
double[][] A = null;
double[][] B = null;
double[][] C = null;
MatrixMultiplication(int dimension, int threads) {
  this.dimension = dimension;
  this.threads = threads;
  A = new double[dimension][dimension];
  B = new double[dimension][dimension];
  C = new double[dimension][dimension];
}
public void initilization() {
  Random rnd = new Random();
  for (int i = 0; i < dimension; i++) {
    for (int j = 0; j < dimension; j++) {
      A[i][j] = rnd.nextDouble();;
      B[i][j] = rnd.nextDouble();;
      C[i][j] = 0;
    }
  }
}
public void paral_compute() throws InterruptedException {
  final int length = dimension/threads;
  final int last_thread = threads-1;
  ThreadTask[] task = new ThreadTask[threads];
  for (int i = 0; i < threads-1; i++) {
    int j=i*length;
    task[i] = new ThreadTask(j, j+length);
    task[i].start();
  }
  task[last_thread] = new ThreadTask((last_thread)*length,dimension);
  task[last_thread].start();
  for(int i = 0; i < threads; i++)
    task[i].join();
}
private class ThreadTask extends Thread {
```

```
    int start, end;
    ThreadTask(int start, int end){
      this.start=start;
      this.end=end;
    }
    @Override
    public void run() {
    for(int i=start; i<end; i++)
      for(int j=0;j<dimension; j++)
        for(int k=0; k<dimension; k++)
          C[i][j] += A[i][k]*B[k][j];
    }
  }
  public static void main(String[] args) throws InterruptedException {
    if (args.length != 2) {
      System.out.println(
         "Usage: MatrixMuliplication Matrix_Dimension threads");
      System.exit(1);
    }
    final int dimension = Integer.parseInt(args[0]);
    final int threads = Integer.parseInt(args[1]);
    MatrixMultiplication matrixmulitipication =
                        new MatrixMultiplication(dimension, threads);
    matrixmulitipication.initilization();
    long start = System.currentTimeMillis();
    matrixmulitipication.paral_compute();
    long time = System.currentTimeMillis() - start;
    System.out.println("size:"+dimension+",threads:"+threads+","+time);
  }
}
```

Listing A.5: Rhyming Words Pipeline

```
abstract class PipelineStage implements Runnable {
  BlockingQueue in;
  BlockingQueue out;
  CountDownLatch s;
```

```
  boolean done;
  abstract void firstStep() throws Exception;
  abstract void secondStep() throws Exception;
  abstract void lastStep() throws Exception;
  void handleComputeException (Exception e) {
    e.printStackTrace();
    Thread.dumpStack();
  }
  @Override
  public void run() {
    try{
      firstStep();
      while(!done) {secondStep();}
      lastStep();
   }
   catch(Exception e) {handleComputeException(e);}
     finally {s.countDown();}
  }
  public void init(BlockingQueue in,
                   BlockingQueue out,CountDownLatch s){
    this.in=in;
    this.out=out;
    this.s=s;
  }
}

abstract class LinearPipeline {
  PipelineStage[] stages;
  BlockingQueue[] queues;
  int numStages;
  CountDownLatch s;
  abstract PipelineStage[] getPipelineStages(String[] args);
  abstract BlockingQueue[] getQueues(String[] args);
  LinearPipeline(String[] args) {
    stages = getPipelineStages(args);
    queues = getQueues(args);
    numStages = stages.length;
```

```
    s = new CountDownLatch(numStages);
    BlockingQueue in = null;
    BlockingQueue out = queues[0];
    for(int i=0; i<numStages; i++) {
      stages[i].init(in, out, s);
      in = out;
      if(i < numStages-2) out = queues[i+1];
        else out=null;
    }
  }
  public void start() {
    for(int i=0; i<numStages; i++)
      new Thread(stages[i]).start();
  }
}

class FileStage extends PipelineStage {
  static int FILENUMBER;
  FileReader wordsfiles;
  int filecount=0;
  FileStage(int filenumber) {
    FILENUMBER = filenumber;
  }
  @Override
  void firstStep() throws Exception {
  }
  @Override
  void secondStep() throws Exception {
    wordsfiles = new FileReader("words.txt");
    BufferedReader in = new BufferedReader(wordsfiles);
    String input;
    filecount++;
    done = filecount > FILENUMBER ? true:false;
    if(!done) {
      while(!(input = in.readLine()).equals("EOF"))
        out.put(input);
      out.put("EOF");
```

```
      wordsfiles.close();
      in.close();
    }
  }
  @Override
  void lastStep() throws Exception {
    out.put("DONE");
  }
}

class ReverseStage1 extends PipelineStage {
  String input;
  @Override
  void firstStep() throws Exception {
  }
  @Override
  void secondStep() throws Exception {
    input = (String)in.take();
    done = (input.equals("DONE"));
    if(!done) {
      if(input.equals("EOF"))
        out.put("EOF");
      else
        out.put(reverseIt(input));
    }
  }
  @Override
  void lastStep() throws Exception {
    out.put("DONE");
  }
  private String reverseIt(String source) {
    int i, len = source.length();
    StringBuffer dest = new StringBuffer(len);
    for (i = (len - 1); i >= 0; i--)
      dest.append(source.charAt(i));
    return dest.toString();
  }
```

```
}

class SortStage extends PipelineStage {
  String input;
  @Override
  void firstStep() throws Exception {
  }
  @Override
  void secondStep() throws Exception {
    input = (String)in.take();
    done = (input.equals("DONE"));
    if(!done) {
      int MAXWORDS = 100;
      String[] listOfWords = new String[MAXWORDS];
      int numwords = 1;
      listOfWords[0]=input;
      while (!(listOfWords[numwords] =
                                  (String)in.take()).equals("EOF"))
        numwords++;
      quicksort(listOfWords, 0, numwords-1);
      for (int i = 0; i < numwords; i++)
        out.put(listOfWords[i]);
    }
  }
  @Override
  void lastStep() throws Exception {
    out.put("DONE");
  }
  //implement quicksort for string objects
  private static void quicksort(String[] a, int lo0, int hi0) {
    int lo = lo0;
    int hi = hi0;
    if (lo >= hi)
      return;
    String mid = a[(lo + hi) / 2];
    while (lo < hi) {
```

```
      while (lo<hi && a[lo].compareTo(mid) < 0)
        lo++;
      while (lo<hi && a[hi].compareTo(mid) > 0)
        hi--;
      if (lo < hi) {
        String T = a[lo];
        a[lo] = a[hi];
        a[hi] = T;
        lo++;
        hi--;
      }
    }
    if (hi < lo) {
      int T = hi;
      hi = lo;
      lo = T;
    }
    quicksort(a, lo0, lo);
    quicksort(a, lo == lo0 ? lo+1 : lo, hi0);
  }
}

class ReverseStage2 extends PipelineStage {
  String input;
  FileWriter outfile;
  @Override
  void firstStep() throws Exception {
    outfile = new FileWriter("wordsout.txt");
  }
  @Override
  void secondStep() throws Exception {
    input = (String)in.take();
    done = (input.equals("DONE"));
    if(!done)
          outfile.write(reverseIt(input)+"\n");
  }
  @Override
```

```
  void lastStep() throws Exception {
    outfile.close();
  }
  private String reverseIt(String source) {
    int i, len = source.length();
    StringBuffer dest = new StringBuffer(len);
    for (i = (len - 1); i >= 0; i--)
      dest.append(source.charAt(i));
      return dest.toString();
  }
}

class MyPipeline extends LinearPipeline {
  static final int STAGES=4;
  MyPipeline(String[] args) {
    super(args);
  }
  @Override
  PipelineStage[] getPipelineStages(String[] args) {
    PipelineStage[] stages = new PipelineStage[STAGES];
    stages[0] = new FileStage(Integer.parseInt(args[0]));
    stages[1] = new ReverseStage1();
    stages[2] = new SortStage();
    stages[3] = new ReverseStage2();
    return stages;
  }
  @Override
  BlockingQueue[] getQueues(String[] args) {
    int totalStages = stages.length;
    BlockingQueue[] queues = new BlockingQueue[totalStages-1];
    for(int i=0; i<totalStages-1; i++)
      queues[i] = new LinkedBlockingQueue<String>();
    return queues;
  }
  public static void main(String[] args) throws InterruptedException {
    LinearPipeline mypipe = new MyPipeline(args);
    long start = System.currentTimeMillis();
```

```
    mypipe.start();
    mypipe.s.await(); //terminate threads when all stages terminates
    long time = System.currentTimeMillis() - start;
    System.out.println("All threads terminated:"+time+" msec");
  }
}
```

Vita

Candidate's full name: Chenwei Wang

Universities attended:
Harbin Institute of Technology (2000-2004)
Bachelor of Software Engineering

Harbin Institute of Technology (2005-2007)
Master of Computer Science