Modern Software Engineering Concepts and Practices:
Advanced Approaches

Ali H. Doğru
Middle East Technical University, Turkey

Veli Biçer
FZI Research Center for Information Technology, Germany

INFORMATION SCIENCE REFERENCE
Hershey · New York

Senior Editorial Director: Kristin Klinger
Director of Book Publications: Julia Mosemann
Editorial Director: Lindsay Johnston
Acquisitions Editor: Erika Carter
Development Editor: Joel Gamon
Production Coordinator: Jamie Snavely
Typesetters: Keith Glazewski & Natalie Pronio
Cover Design: Nick Newcomer

Published in the United States of America by
Information Science Reference (an imprint of IGI Global)
701 E. Chocolate Avenue
Hershey PA 17033
Tel: 717-533-8845
Fax: 717-533-8661
E-mail: cust@igi-global.com
Web site: http://www.igi-global.com

Library of Congress Cataloging-in-Publication Data

Modern software engineering concepts and practices : advanced approaches / Ali H. Doğru and Veli Biçer, editors.
p. cm.
Includes bibliographical references and index.
Summary: "This book provides emerging theoretical approaches and their practices and includes case studies and real-world practices within a range of advanced approaches to reflect various perspectives in the discipline"-- Provided by publisher.
ISBN 978-1-60960-215-4 (hardcover) -- ISBN 978-1-60960-217-8 (ebook) 1. Software engineering. I. Doğru, Ali H., 1957- II. Biçer, Veli, 1980-
QA76.758.M62 2011
005.1--dc22
2010051808

British Cataloguing in Publication Data
A Cataloguing in Publication record for this book is available from the British Library.

Chapter 16
Analyzing Concurrent Programs Title for Potential Programming Errors

Qichang Chen
University of Wyoming, USA

Liqiang Wang
University of Wyoming, USA

Ping Guo
University of Wyoming, USA

He Huang
University of Wyoming, USA

ABSTRACT

Today, multi-core/multi-processor hardware has become ubiquitous, leading to a fundamental turning point on software development. However, developing concurrent programs is difficult. Concurrency introduces the possibility of errors that do not exist in sequential programs. This chapter introduces the major concurrent programming models including multithreaded programming on shared memory and message passing programming on distributed memory. Then, the state-of-the-art research achievements on detecting concurrency errors such as deadlock, race condition, and atomicity violation are reviewed. Finally, the chapter surveys the widely used tools for testing and debugging concurrent programs.

DOI: 10.4018/978-1-60960-215-4.ch016

INTRODUCTION

The development in the computing chip industry has been roughly following Moore's law in the past four decades. As a result, most classes of applications have enjoyed regular performance gains even without real improvement on the applications themselves, because the CPU manufacturers have reliably enabled ever-faster computer systems. However, the chip industry is now facing a number of engineering challenges associated with power consumption, power dissipation, slower clock-frequency growth, processor-memory performance gap, etc. Instead of driving clock speeds and straight-line instruction throughput ever higher, the CPU manufacturers are instead turning to multi-core architectures.

With the prevalence of multi-core hardware on the market, the software community is witnessing a dramatic shift from the traditional sequential computing paradigm to the parallel computing world. Parallel computing exploits the inherent data and task parallelism and utilizes multiple working processes or threads at the same time to improve the overall performance and speed up many scientific discoveries. Although threads have certain similarities to processes, they have fundamental differences. In particular, processes are fully isolated from each other; threads share heap memory and files with other threads running in the same process. The major benefits of multithreading include faster inter-thread communication and more economical creation and context switch.

Here, we use "concurrent" and "parallel" interchangeably, although there is a little difference between them. Usually, "parallel programming" refers to a set of tasks working at the same time physically, whereas "concurrent programming" has a broader meaning, *i.e.*, the tasks can work at the same time physically or logically.

Although for the past decade we have witnessed increasingly more concurrent programs, most applications today are still single-threaded and can no longer benefit from the hardware improvement without significant redesign. In order for software applications to benefit from the continued exponential throughput advances in new processors, the applications will need to be well-written concurrent software programs.

However, developing concurrent programs is difficult. Concurrency introduces many new errors that are not present in traditional sequential programs. Recent events range from failing robots on Mars to the year 2003 blackout in northeastern United States, which were both caused by a kind of concurrency error called race condition. Debugging concurrent programs is also difficult. Concurrent programs may behave differently from one run to another because parallelism cannot be well determined and predicted beforehand. Existing debugging techniques that are well adopted for sequential programs are inadequate for concurrent programs. Specialized techniques are needed to ensure that concurrent programs do not have concurrency-related errors. Detecting concurrency errors effectively and efficiently has become a research focus of software engineering in recent years.

In the rest of the chapter, we review the state-of-the-art research achievements on detecting concurrency errors as well as the corresponding parallel programming models. Major debugging tools are also introduced and compared with regard to their usability and capability.

PARALLEL COMPUTING PLATFORMS

Advances on Architecture: Multi-Core Processor

Due to the physical limitations of the technology, keeping up with Moore's Law by increasing the number of transistors on the limited chip area has been becoming a more difficult challenge for the CPU industry. In the past decade, we have

witnessed an increasing number of hardware architectures that shift towards parallelism instead of clock speed. The industry has gradually turned to parallelism in computational architectures with the hope of living up to Moore's law in terms of GFLOPS (1 GFLOPS = 109 floating-point-operations per second) performance growth per chip area instead of transistors growth per chip area. The prominent CPU vendors (namely, Intel and AMD) are packing two or more cores into each processor unit while the clock speed no longer grows at the rate of as Moore law dictates the transistor density growth. The multi-core processor architecture has become an industrial standard and is gradually changing the way people write programs.

In essence, a multi-core processor is a single chip containing more than one microprocessor core, which multiplies the potential performance with the number of cores. Some components, such as the bus interface and second level cache, are shared between cores. Because the cores are physically very close, they communicate at much higher bandwidth and speed compared to conventional discrete multiprocessor systems, which significantly reduces the communication overhead and improves overall system performance.

In order to fully utilize the multi-core capability, the programmers need to explicitly migrate their sequential code into parallel version. This opens the door to introduce many subtle concurrent programming errors that could be very difficult to detect and uncover. Those concurrent programming errors are introduced in details in Section 4.

Advances on Architecture: Accelerator for General Purpose Computing

As the scale of computing increases dramatically, coprocessor is becoming a popular attracting technique to accelerate general purpose computing. At this point, there are two types of widely used accelerators, i.e., GPU (Graphic Processing Unit) and Cell Broadband Engine (BE).

GPU is specifically designed to provide high throughput on parallel computation using many-core chips. Unlike conventional CPU that supports a much larger set of general instructions, GPU supports only graphics-related computations, which can be adopted for general-purpose scientific computations. More specifically, unlike a conventional CPU, in a GPU, much more transistors are devoted to data processing rather than data caching and flow control, which is especially well suited to address problems that can be expressed as data-parallel computations, where the same program is executed on many data elements in parallel with high arithmetic intensity. Because the same program is executed for multiple data elements, there is a lower requirement for sophisticated flow control; and because it is executed on many data elements and has high arithmetic intensity, the memory access latency can be hidden with calculations instead of big data caches.

Due to its high arithmetic computation power, GPU is becoming an attractive co-processor option for general-purpose computing. A typical GPU (e.g., NVIDIA GeForce GTX 295) can reach a peak processing rate of 1788 GFLOPS and a peak memory bandwidth of 224 GB/s, which are not possible to be achieved on a typical current-generation CPU.

As another type of well-known accelerator, Cell BE is a hybrid processor between conventional desktop processors (e.g., Intel Core 2) and more specialized high-performance processors (e.g., GPU). A Cell BE chip consists of a main processor called Power Processor Element (PPE), which can run general operating systems (e.g., Linux) and functions, and eight Synergistic Processor Elements (SPE), which are designed to accelerate data-parallel computing. The latest Cell BE, PowerXCell 8i, supports a peak performance of 102 GFLOPS double-precision calculations. Cell BE accelerators have been deployed on the supercomputer Roadrunner designed by IBM at

the Los Alamos National Laboratory, which is the world's first petaflops system.

Super, Cluster, Grid, and Cloud Computing

Supercomputers are specialized computers that rely on innovative and throughput-oriented design in order to obtain high-performance and high-throughput computing gain over conventional computers. For example, the memory hierarchy in supercomputer is carefully designed to minimize the idle time of processors. The I/O systems are designed to support large-scale parallel I/O operations with high bandwidth. Many CPUs with specific tuned instructions set (*e.g.*, Complex Instruction Set Computer/ CISC) and advanced pipeline mechanisms have been invented for the purpose of high-throughput computing. Vector processor, or array processor, is such a CPU design where the instructions can perform mathematical operations on multiple data elements simultaneously. Supercomputers are usually specialized for certain types of computation, such as numerical calculations.

As the most popular parallel computing platform, a computing cluster is a collection of computers that are located physically in a small area and are linked through very fast local area network. Clusters are usually deployed for higher performance and availability, while typically being much more cost-effective than supercomputers with the comparable speed or availability. Cluster computing typically incorporates fault tolerance, data redundancy, load balancing and other features to ensure the high availability and reliability.

Grid computing is an extension of cluster computing. Computers in grid can be geographically dispersed and do not fully trust each other. Hence, the communication in grid may suffer much higher latency than cluster. A grid combines various compute resources from multiple domains to solve a common scientific problem that requires a lot of compute processing cycles and large amount of data. It is a form of distributed computing in practice.

As an emerging parallel computing platform, cloud computing has recently gained wide attention. A cloud encompasses the cluster of machines as well as the system and application software that work together to provide some kind of service. Cloud computing is more scalable than the classical cluster, and can typically offer higher data availability and throughput as well as virtualized services. These services are broadly divided into three categories: Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), and Software-as-a-Service (SaaS). Amazon Web Services is such an example of IaaS, where virtual server instances are provided according to the capacity that users purchase. Google Apps is an example of PaaS, where users can create their own applications based on the provider's platform over the Internet. In the SaaS cloud model, services can be very broad, ranging from Web-based email to database processing. This type of cloud computing delivers a client-side application through the Internet browser to thousands of customers with much lower maintenance cost. In addition, cloud computing is more reliable and scalable than cluster and grid computing as it allows redundancy and can easily incorporate more machines, more processors, and more storage space to improve the overall performance without affecting the end users. All traditional and new concurrent programming models ranging from MPI, OpenMP, to CUDA might be adopted on cloud computing. Specifically, cloud computing is extremely amenable to the MapReduce concurrent programming model as most applications running on the cloud are data-parallel.

PARALLEL PROGRAMMING MODELS

In order to fully utilize the power of underlying parallel computing platforms, various parallel

programming models have been developed. Most parallel programming models are derived from the traditional sequential programming. The sequential programming is embodied through an imperative or functional program that executes on a single processor. The behavior of the sequential program is predictable and the outcome is expectable every time it runs. In order to improve the performance of sequential programs, people resort to the processor clock frequency and other hardware optimization improvements which have grown relatively slowly recently. Concurrent programming models allow parts of the sequential program to run in parallel on multiple processors or concurrently on a single processor. The current widely used concurrent programming models include multithreaded programming on shared memory and message passing programming on distributed memory.

In multithreaded programming, multiple threads share the memory. Synchronization among threads is mainly enforced through lock (mutex), semaphore, spinlock (spinmutex), and monitor. A semaphore is an object that carries a value and uses a blocking mechanism to enforce the mutual exclusion among multiple threads. A lock or mutex is a special case of semaphore (i.e., binary semaphore) whose values can be only true or false. A spinlock or spinmutex is also a mutually exclusive object that instead uses a non-blocking mechanism to enforce the mutual exclusive property. Spinlock differs from the usual lock in that it utilizes busy waiting/checking without enforcing context switches. A monitor is a mutually exclusive code region that can be executed by one thread at a time. It is achieved through the acquisition and release of the lock or spinlock immediately before the entrance and after the exit of the code region. The major multithreaded programming paradigms include Java/C# Threads, Pthreads, Windows threads, OpenMP, TBB (Intel Threading Building Block), as well as CUDA and openCL for GPU computing.

On distributed memory, each compute node has its own private memory. Communication and synchronization among processes are carried out by sending and receiving messages. The main message passing programming model is MPI (Message Passing Interface).

Some hybrid parallel programming models exists. UPC is such a model which combines the programmability advantages of the shared memory programming paradigm and the control over data layout of the **message passing programming paradigm**.

This section introduces the major parallel programming models for shared memory and distributed memory, with focusing on comparing their synchronization mechanisms.

Multithreaded Programming on Shared Memory Model

The traditional parallel programming model on shared memory is multiprocessing, where multiple processes share the same memory. Compared to forking or spawning new processes, threads require less overhead because the system does not initialize a new system virtual memory space and environment for the process. In addition, the overhead of context switch on threads is also much less than on processes. In contrast to parallel programming paradigms such as MPI in a distributed computing environment, threads are usually limited to a single computer system. All threads within a process share the same address space.

Java Threads

Multithreaded execution is an essential feature of Java platform. Multithreaded Java programs start with the main thread, which then spawns additional threads. The conventional synchronization mechanisms supported by Java include *monitor, wait/notify, semaphore,* and *barrier*.

A *monitor* is an object implementation where at most one thread can simultaneously execute

Algorithm 1.

synchronized(this){
this.balance += depositAmount;
}
	synchronized(this){
	this.balance -= withdrawAmount;
	}

any of its methods (*i.e.*, *critical sections*). If one thread is inside the monitor (*i.e.*, holding the lock associated to the monitor), the other threads must wait for that thread to exit the monitor (*i.e.*, release the lock) before they can enter the monitor. The lock of monitor in Java is reentrant, which means that a thread holding a lock can request it again. To use monitor, programmer can explicitly label an entire method or a well-defined code block inside a method as critical sections using the keyword "synchronized". As "synchronized" enforces paired lock acquire and lock release, the "Lock" class in Java allows more flexible structuring by providing Lock.lock() and Lock. unlock(). Algorithm 1 shows that two threads use "synchronized" to keep the integrity of the "balance" for a bank account.

A thread can hang itself and wait for another thread by calling "wait", which will release the corresponding lock if in critical section. When the other thread invokes "notify" to wake up the suspended thread, it will acquire the lock and resume its execution. Algorithm 2 shows how to enforce the deposit/withdraw order using wait/ notify. In this case, a deposit should always occur before a withdraw can take place.

In addition, Java introduces a few specialized synchronization mechanisms. Java supports the keyword "volatile", which is used on variables that may be modified concurrently by multiple threads. Compiler will enforce fetching fresh volatile variables each time, rather than caching them in registers. Note that volatile does not guarantee atomic access, *e.g.,* i++. Java provides a series of classes (such as AtomicInteger and AtomicIntegerArray) to support atomic operations. When a thread performs an atomic operation, the other threads see it as a single operation. The advantage of atomic operations is that they are relatively quick compared to locks, and do not suffer from deadlock. The disadvantage is that they support only a limited set of operations, and are often not enough to synthesize complex operations efficiently.

In addition, Java JDK also contains a set of collection classes (*e.g.,* HashTable, Vector) to support safe concurrent modification. They may run slightly slower than their counterparts (*e.g.,* HashMap, ArrayList) that may throw exceptions or have incorrect results when performing concurrent operations.

C# Threads

Threads in C# behave similarly to Java threads. However, instead of using "synchronized", C# provides its own synchronization keywords. To enforce critical sections, Monitor class can be used to acquire a lock at the beginning of the code section by calling Monitor.Enter(object). Any other

Algorithm 2.

synchronized(this){
this.wait();
this.balance -= withdrawAmount;
}
	synchronized(this){
	this.balance += depositAmount;
	this.notify();
	}

Algorithm 3.

bool acquiredLock = false;
try{
Monitor.Enter(lockObject,
ref acquiredLock);
this.balance -= withdrawAmount;
} finally {
if (acquiredLock)
Monitor.Exit(lockObject);
}
	bool acquiredLock = false;
	try{
	Monitor.Enter(lockObject,
	ref acquiredLock);
	this.balance += depositAmount;
	} finally {
	if (acquiredLock)
	Monitor.Exit(lockObject);
	}

thread wanting to execute the same code would need to acquire the same lock and will be paused until the first thread releases the lock by calling Monitor.Exit(object). Algorithm 3 illustrates the usage of Monitor to protect the integrity of the field "balance".

C# also provides a keyword "lock", a syntactic shortcut for a paired call to the methods Monitor.Enter and Monitor.Exit, which is converse compared to Java Synchronized and Lock. Algorithm 4 illustrates the similar use of lock in C# as "synchronized" in Java.

The mutual exclusion enforced by lock and monitor in C# is only among threads inside a process. To enforce mutual exclusion across multiple processes, Mutex can be used, which works in the same way as lock inside a process. Besides Monitor, Lock, and Mutex, C# also supports Semaphore, Barrier, Volatile, and atomic operations (by calling System.Threading.Interlocked), whose meanings and usages are similar to these in Java.

Pthreads

Pthreads stands for POSIX Threads libraries for C/C++. Thread operations provided by Pthreads include thread creation, termination, synchronization, scheduling, data management, and process interaction. Threads in the same process share process instructions, most data, open files (descriptors), signals and signal handlers, current working directory, user and group identifies. However, each thread has its own thread ID, set of registers, stack pointer, stack for local variables, return addresses, signal mask priority, and return value.

Pthreads provides three synchronization mechanisms: *join*, *mutex*, and *condition variables*. After a thread has been spawned, programmers can perform join operation by calling pthread_join. The calling thread will suspend its execution until the thread to be joined has finished its execution. This allows certain cooperation between different tasks, for example, one thread is waiting for the results from other threads for further execution.

Mutex is mainly used to protect memory access thus prevent data races. A mutex variable is declared by "pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER". After a mutex variable is created, acquiring and releasing locks can be performed by calling pthread_mutex_lock(&mutex) and pthread_mutex_unlock(&mutex), respectively.

Algorithm 4.

lock(this){
this.balance -= withdrawAmount;
}
	lock(this){
	this.balance += depositAmount;
	}

Algorithm 5.

pthread_mutex_lock(&mutex);
while (!cond)
thread_cond_wait(&cond,&mutex);
do_something();
pthread_mutex_unlock(&mutex);
	pthread_mutex_lock(&mutex);
	… //make condition TRUE
	if (cond)
	pthread_cond_signal(&cond);
	pthread_mutex_unlock(&mutex);

A condition variable is a variable in the type of pthread_cond_t and offers a more flexible way for threads to suspend/resume execution than the "join" mechanism. A condition variable should be protected with a mutex in order to avoid race conditions. Without mutex, the signaling thread and the waiting thread may access the condition variable as well as other shared variables simultaneously, which results in race conditions. Algorithm 5 shows how to use condition variable and mutex together.

To force the current thread to wait on a condition variable, pthread_cond_wait is called. At the same time, the mutex currently held is also released. pthread_cond_timedwait works similarly except for waiting for a specific time period then resuming that thread's execution. Other threads can wake up a waiting thread by calling pthread_cond_signal or pthread_cond_broadcast (which wakes up all waiting threads). After receiving waking up signal, the waiting thread will acquire mutex again and resume its execution.

OpenMP

OpenMP (Open Multi-Processing) is another multithreaded programming model that supports C, C++, and Fortran on many platforms including Linux and Microsoft Windows. It is composed of a set of compiler directives, library routines, and environment variables that influence program's run-time behavior.

The OpenMP compiler directives are manually inserted into programs and indicate how to execute the code sections in parallel. For example, consider the following loop to sum the elements of two arrays, the directive indicates that the iterations of the loop can be executed in parallel, *i.e.*, a few concurrent threads will be spawned at runtime and each thread handles some iterations.

```
#pragma omp parallel for
for (i=0; i<n; i++){
    c[i] = a[i] + b[i];
}
```

In order to be parallelized, the loop must obey certain patterns. OpenMP does not detect dependencies between loop iterations, which may incur race conditions. A way to avoid such race conditions is to use critical introduced below.

Other widely-used OpenMP directives include "master", "critical" and work-sharing "for" and "section directives.

The "master" directive specifies a region that is to be executed only by the master thread of the thread group. All other threads on the group skip this section of code. The example below allows only the master thread to initialize the counter and other threads to skip the initialization.

```
#pragma omp master
int counter = 0;
```

The "critical" directive specifies a region of code that must be executed by only one thread at a time. The following example shows how to allow multiple threads to update the variable "balance" in a concurrent way without incurring race conditions.

```
#pragma omp critical
balance += depositAccount;
```

The work-sharing "for" directive specifies that the iterations of the for-loop are executed in parallel. The granularity of the parallelism can be decided statically using a keyword "static" or run time using the keyword "dynamic". Optionally one can provide a chunk size if she/he wants to assign more than one iteration to a thread. This following example shows an example to assign iterations evenly among the threads in compile time (statically).

```
#pragma omp for schedule static
for (i=0; i<n; i++){
    c[i] = a[i] + b[i];
}
```

The work-sharing "sections" directive allows multiple threads to execute different code blocks only once. In the example below, three threads will execute the first, second, third block labeled with "#pragma omp section" independently once.

```
#pragma omp sections
{
    #pragma omp section
    { int a = 0; }
    #pragma omp section
    { int b = 0; }
    #pragma omp section
    { int c = 0; }
}
```

In principle, one of advantages for the compiler directive strategy is that the code can run as ordinary sequential code if the directives are ignored. Unfortunately, some of the OpenMP directives, such as these managing memory consistency and local copies of variables, affect the semantics of the sequential code, compromising this desirable property unless the code avoids these directives.

Intel TBB

Intel TBB (Threading Building Block) is a runtime-based parallel programming paradigm. It is a C++ template library that consists of data structures and algorithms aiming to reduce the complexity arising from the use of the more primitive threading packages such as Pthread and Windows threads. Like the high-level concurrency objects in Java, TBB library simplifies thread-level parallelism further into task-level. Programmers only need to specify the intended parallel code as tasks and do not need to explicitly control the underling scheduling and synchronization technicalities. The library's runtime engine will take care of the rest which prevents programmers from making potential concurrency errors. This offers an alternative way for developers to leverage multi-core processors without being an expert on multithreading. However, this might limit the type of applications that can be ported to TBB since many tightly-coupled programs require more fine-grained synchronization between threads.

Intel TBB provides mutual exclusion and atomic operations for synchronization among different threading blocks. TBB has several kinds of mutex objects (spin_mutex, queuing_mutex, spin_rw_mutex, queuing_rw_mutex, mutex) for different performance, fairness, and reentrant, *etc*. For example, a thread trying to acquire a lock on spin_mutex is in busy wait till acquiring the lock. A spin_mutex is appropriate when the lock is held for a short time (*e.g.* a few instructions). However incorrect uses may incur huge performance penalty. As a cheaper alternative to mutex, atomic operations can be used. The Class atomic<T> implements atomic operations. It supports three popular atomic non-blocking operations: fetch_and_store, fetch_and_add, and compare_and_swap.

CUDA and OpenCL

GPU software development tools have evolved rapidly with the dramatic advances of GPU hardware. In the early stage, people struggled with the implementation of scientific computing using graphics primitives. Then several high-level abstractions for streaming programming, such as BrookGPU (Buck, et al., 2004) and Sh (McCool, et al., 2004), were designed to hide the graphics-specific details of GPU programming. Recently, commercially supported GPU program toolkits, in particularly CUDA (Cuda), RapidMind (RapidMind), and OpenCL (Open Computing Language (OpenCL)), dramatically promote general computing on GPU and help leverage GPU capabilities and manage data parallel computations on high-level programming languages, such as C/C++.

NVIDIA's CUDA is the leader of programming interfaces for GPU computing. The CUDA software stack is composed of several layers including a hardware driver, an application programming interface (API) and its runtime environment, as well as high-level mathematical libraries. The main synchronization mechanism in CUDA is barrier (through calling "__syncthreads()"). CUDA also provides atomic operations that are performed without interference from any other threads in order to prevent race conditions. The following example (NCSA-CUDA) shows how to use CUDA to conduct addition operations on two vectors. The keyword label "__global__" declares a function to be a kernel function which is executed on CUDA device and called from the host only. In this case, the kernel code computes the sum of two float vectors in parallel on the CUDA device. The host code (the rest code in the example) allocates the device memory and provides some essential parameters such as the grid dimension, number of blocks, number of thread per block when calling the kernel function. In this case, the host code first calls CUDA memory functions to allocate the device memory for the two input vectors A and B and the sum vector C and then specifies the number of block to be 1 and the number of threads per block to be 10 when calling the CUDA kernel function "vecAdd". At the end of the program, the host code calls CUDA memory functions to release the allocated memory.

```
__global__ void vecAdd(float* A,
float* B, float* C) {
      int i = threadIdx.x;
      A[i]=0;
      B[i]=i;
      C[i] = A[i] + B[i];
}
int main() {
      int N=10, SIZE=10;
      float A[SIZE], B[SIZE],
C[SIZE];
      // Kernel invocation

      float *devPtrA;
      float *devPtrB;
      float *devPtrC;
      int memsize= SIZE *
sizeof(float);

      cudaMalloc((void**)&devPtrA,
memsize);
      cudaMalloc((void**)&devPtrB,
memsize);
      cudaMalloc((void**)&devPtrC,
memsize);
      cudaMemcpy(devPtrA, A, memsize,
cudaMemcpyHostToDevice);
      cudaMemcpy(devPtrB, B, memsize,
cudaMemcpyHostToDevice);
      vecAdd<<<1, N>>>(devPtrA, devP-
trB, devPtrC);
      cudaMemcpy(C, devPtrC, memsize,
cudaMemcpyDeviceToHost);

      for (int i=0; i<SIZE; i++)
        printf("C[%d]=%f\n",i,C[i]);

      cudaFree(devPtrA);
```

```
    cudaFree(devPtrA);
    cudaFree(devPtrA);
}
```

Another open GPGPU program interface is OpenCL (OpenCL) which is similar to CUDA in many aspects. OpenCL programs are a mixed form of host code and device code. It uses keyword labeling to express data parallelism for device code and the host code. The execution of an OpenCL program involves simultaneous execution of multiple instances of a kernel on the OpenCL devices as they are queued and controlled by the host application. Each instance of a kernel is referred to as a work-item. The data parallelism lies in that each work item executes the same code on different portions of the data. Each work-item runs independently on a single core of OpenCL device.

OpenCL supports two forms of synchronization between work-items in the same workgroup: barriers and memory fences. The barrier operation barrier() allows the work-items in the same group to have the same progress before starting the next stage. The fence operation mem_fence() forces all outstanding loads and stores on the OpenCL device memory to be completed before execution proceeds, and disallows the compiler and runtime system from reordering any loads and stores. This can be used to ensure that all data produced in a work-group are flushed to global OpenCL device memory before proceeding, which prevents other work-groups from reading premature results.

Parallel Programming on Distributed Memory Model: MPI

MPI (Message Passing Interface) is currently the *de facto* standard programming model for high-performance scientific computing. MPI defines the syntax and semantics of a core of library routines for writing portable message-passing programs. Besides supporting distributed memory, MPI can also utilize the shared memory for faster data communication between processes on the same node. MPI supports many popular languages such as C, C++, Fortran, Python, and Java. There are several open source implementations of MPI like MPICH, LAM MPI, and OpenMPI, and commercial implementations from Portland Group, HP, Intel, Sun, IBM, and Microsoft.

Unlike multithreading, message passing is a form of distributed memory programming paradigm based on the sending and receiving messages. In message passing, multiple processes coordinate their progress and communicate their immediate results by sending messages to one or more designated receivers and receiving messages from one or more designated senders. A send/receive can be a blocking or non-blocking operation. A blocking operation blocks the process from executing next instruction until it completes. A non-blocking operation does not need to wait for the finish of designated events, the process will continue without suspending. For example, in MPI, mpi_send() is a blocking send and mpi_isend() is a non-blocking one. The blocking operations are unsafe operations as improper use of them would lead to deadlock and other type of concurrency errors. The nonblocking ones can be safe but more difficult for coding.

A send/receive can target at a specific group of processes by specifying a communicator. A communicator designates a group of processes for sending/receiving message. For example, in MPI, the macro MPI_COMM_WORLD is the initially defined universe intracommunicator for all processes to conduct various communications. A send/receive can also use a message tag to indicate what kind of message it expects to send/receive. Tags are used to distinguish different message types a process might send/receive.

There are a number of message passing patterns that are commonly used in MPI programs. *Point-to-point* is the most basic communication pattern in MPI. One process uses MPI_Send() or its variant to send a message to a designated receiving process which uses MPI_Recv() or

its variant to receive the message. A *collective communication* is a communication pattern that involves all the processes in a communicator. Consequently, a collective communication is usually associated with more than two processes. *Barrier* is a program point where all processes in the same communicator should reach before proceeding.

Most MPI routines also enforce some kind of synchronization. For example, MPI_Bcast() broadcasts a message to the whole communication group; a node will not continue until it receives the message. In addition, MPI explicitly provides barrier operation to synchronize the progress of different processes. A process that calls MPI_Barrier() will block itself until all processes in the same group have reach the barrier point. The following example taken from (ANL MPI) shows how to use MPI to calculate π on parallel machines. Every process on each machine will execute the same code except they use "myid" to distinguish from each other. MPI_Comm_size stores the total number of processes in the variable "numprocs" for further use. Since MPI programs can be started with any number of processes using the option "-np", it is very useful to take into account the total number of processes in the computation as the number may vary each time. In this program, it asks user to provide the number of intervals (steps) for computing π which is broadcasted to each process from the root process. Later, each process except the root process (process with rank 0) does its own computation and calls MPI_Reduce to sum up their results. Finally, the root process prints out the computed pi against a 25-digit π.

Algorithm 6.

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>
int main(int argc, char *argv[]){
int n, myid, numprocs, i;
double PI25DT = 3.14159265358979323846264 3;
double mypi, pi, h, sum, x;
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&myid);
while (1) {
if (myid == 0) {
printf("Enter the number of intervals: (0 quits) ");
scanf("%d",&n);
}
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
      if (n == 0)
      break;
else {
h = 1.0 / (double) n;
sum = 0.0;
for (i = myid + 1; i <= n; i += numprocs) {
x = h * ((double)i - 0.5);
sum += (4.0 / (1.0 + x*x));
}
mypi = h * sum;
MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);
if (myid == 0)
printf("pi is approximately %.16f, Error is
%.16f\n",pi, fabs(pi - PI25DT));
}
}
MPI_Finalize();
return 0;
}
```

MapReduce

MapReduce (Dean 2008) is a loosely-coupled parallel computation model that designs to handle data-intensive computations. MapReduce derives its name from the map and reduce combinators from the functional programming languages. In functional programming languages, a *map* takes a function and a sequence of values as input. It then applies the function to each value in the sequence. A *reduce* combines all the elements of a sequence using a binary operation. One example of MapReduce is that Map takes a function which breaks the input string into characters and Reduce uses a 'count' function to count the total number of characters in the sequence. This model is specifically suitable for many data-intensive applications because those applications usually only apply simple operations on large data which can be easily split into multiple chunks for independent processing. **(Hadoop) is a prominent open source MapReduce implementation that has been adopted for many data-intensive computations.**

UPC

Unified Parallel C (UPC) extends C programming language with a few additional structs to enable Single Program Multiple Data (SPMD) parallel computing model on large-scale machines. The language provides a uniform and integrated programming model for both shared and distributed memory hardware.

UPC provides the following synchronization mechanisms: *barrier, wait/notify, lock, fence*, and *spinlock*. Among them, barrier, wait/notify, and lock work similarly to other multithreaded programming paradigms. The *fence* struct (*i.e.*, upc_fence) in UPC ensures that all the shared references issued ahead of the fence are complete. *Spinlock* is a non-blocking alternative to lock, which is designed for relatively fast operations on shared variables with less performance penalty.

CONCURRENCY ERRORS

With the introduction of the above concurrent programming models, there also come concurrency errors, such as deadlock, race condition, and atomicity violation, that are not present in traditional sequential programming. Like most traditional programming errors such as memory leak, buffer overrun, null pointer dereferencing, *etc.* Preventing and/detecting concurrency errors may suffer performance loss. For example, data race can be prevented using a lock to guard every shared variable's access. This inevitably brings down the performance of concurrent program as too many lock contentions will affect the performance dramatically.

Deadlock

Deadlock is perhaps the most common concurrency error that might occur in almost all parallel programming paradigms including both shared-memory and distributed memory. A *deadlock* occurs when a chain of processes/threads are involved in a cycle in which each process is waiting for resources/locks that are held by some other processes. When a deadlock happens, none of the processes/threads can proceed, which in turn causes the whole or part of the program to halt.

In shared-memory programming models, such as Java threads and Pthreads, when programmers use multiple locks to coordinate the accesses to shared variables from multiple threads, it may result in deadlock if two locks are acquired in different orders in multiple threads. Hence, to avoid such deadlock, successive locks should be locked in the same order. An example of deadlock in Java is shown in Algorithm 7.

Another kind of deadlock may occur in Pthreads is that a thread tries to reacquire a lock that it already owns, as shown in Algorithm 8.

Such kind of deadlock will not happen in Java threads because both "synchronized" and "lock" support reentrant locking. However, it is recommended that locking and unlocking are performed in the same scope. Otherwise, we should use try-finally or try-catch to ensure that unlocking is conducted finally, as shown below.

```
Lock.lock();
try {
    …
}  finally  {
    Lock.unlock();
}
```

Message passing programs can also be victims of deadlock. The features of MPI (such as blocking communication and non-deterministic scheduling) and different implementations of MPI would potentially lead to deadlock. For example, an intuitive deadlock scenario is that some processes are awaiting messages, but these messages may never be sent out because the sending processes are blocked or unable to send. This scenario causes part of processes or even the whole MPI program to be blocked forever.

Algorithm 7.

thread-1:	thread-2:

synchronized(lock1){
synchronized(lock2){}
}
	synchronized(lock2){
	synchronized(lock1){}
	}

Algorithm 8.

g() {
pthread_mutex_lock (&mutex);
…
pthread_mutex_unlock(&mutex);
}
	f() {
	pthread_mutex_lock (&mutex);
	g();
	pthread_mutex_unlock(&mutex);
	}

Figure 1, which is taken from (Hilbrich et al. 2009), shows three kinds of MPI deadlocks. As shown in the example of Figure 1(a), an MPI deadlock would occur when two blocking receives wait for each other, the program cannot continue.

Another example is shown in Figure 1(b). Point-to-point blocking routines may incur deadlock when their executions do not succeed. Point-to-point blocking routines, such as MPI_Send() and MPI_Recv(), do not return to the program until the message data have been safely stored (in message storage or buffer). However, if some problems arise in message storage (*e.g.*, it is full), this may cause some processes to infinitely wait for messages or responses from other processes, although such kind of MPI deadlock happens very rarely.

Point-to-point non-blocking routines can also lead to deadlock, although functions like MPI_Irecv() and MPI_Isend() can return to the program immediately without waiting the messages copying into buffers. But programmers often have to associate a request to a non-blocking routine and later invoke MPI_Wait(), which is also a blocking point-to-point routine.

Many collective MPI routines, such as MPI_Bcast() and MPI_Barrier(), can also cause deadlock if not used correctly. For example, programmers sometimes make mistakes like incomplete barrier operations, which means that not all the processes in the same communicator

Figure 1. Deadlock scenario for MPI programs

process 0	process 1
Recv:(from 1)	Recv:(from 0)
Send:(to 1)	Send:(to 0)

(a). Simple deadlock scenario

process 0	process 1
Send:(to 1)	Send:(to 0)
Recv:(from 1)	Recv:(from 0)

(b). Another possible deadlock scenario

process 0	process 1	process 2
SSend:(to 1)	Recv:(from ANY)	SSend:(to 1)
SSend:(to 2)	Recv:(from 2)	Recv:(from 0)

(c). Possible deadlock scenario involving 3 processes

invoke MPI_Barrier() which leads to a deadlock situation where the whole program can not proceed if the "barrier" point has not been reached by all processes in the same communicator group.

Incorrect use of wildcard receive is another scenario leading to deadlock. As an example shown in Figure 1(c), the labels MPI_ANY_SOURCE and MPI_ANY_TAG allow a process to receive any message from any source or any tag, respectively, but only the first incoming message is matched. If there is any subsequent MPI_Recv() on a specific process whose MPI_Send() is just matched by the previous wild card receive, then the process is blocked.

Data Race

Data race (also called race condition) is present only in the parallel programming models based on shared memory. *Data race* happens when two or more accesses from different threads access the same shared variable without proper synchronization and at least one of the accesses is a write to the variable.

Data race may or may not affect the correctness of the executing program depending on the context in which it occurs. That is, there are two types of data races: *harmful* and *benign*. For example, if thread 1 and thread 2 execute "x=1; y=x+1" and "x=2" (where x is a shared variable by both threads), respectively, a harmful data race may occur because the value of y depends on which statement of "x=1" and "x=2" runs first. A benign data race is shown below, where a boolean variable "flag" controls the ordering of thread 1 and thread 2. When thread 2 is done with its task, it notifies thread 1 by setting "flag" to be true. Thread 1 is constantly checking "flag" to wait for thread 2. In this situation, the data race on the variable "flag" is benign.

Similar to data race in multithreaded programs, message race may occur on programs that utilize message passing mechanisms such as MPI. *Message race* occurs when a process receives messages from other processes at a non-deterministic order. Due to various process schedulings and communication latencies, messages may reach a process at various orders, and thus incur different schedulings and results. In the following example, the process P2 receives its message from process P1 and process P3. There is a message race in the first wildcard receive of P2. Depending on the order, P2 might receive the message from P1 with the value 1 or receive the message from P3 with the value of 0.

Atomicity Violation

Atomicity violation, which is caused when concurrent execution unexpectedly violates the atomicity of a code segment, is another kind of common concurrency errors. Atomicity is well known in the context of transaction processing, where it is sometimes called *serializability*. An *atomicity violation* occurs when an interleaved execution of a set of code blocks (expected to be atomic) by multiple threads is not equivalent to any serial execution of the same code blocks. Figure 2 illustrates two examples in Java from (Chen et al. 2009), where Program 1 contains obvious data races on the shared variable "bal", and Program 2 eliminates the data races in Program 1

Algorithm 9.

Thread 1	Thread 2

…
while(!flag){
…
}
	…
	flag = true;
	…

Algorithm 10.

Process 1(P1)	Process 2(P2)	Process 3(P3)
MPI_Isend(P2, data = 1);	MPI_Irecv(*, x);	MPI_Isend(P2, data = 0);

by adding a lock o. However, it is still incorrect. In this example, the deposit method is expected to be atomic otherwise it would cause the bank account balance to be inconsistent. An atomicity violation would occur when the two synchronization blocks in Thread 2 execute between the two synchronization blocks in Thread 1. From this example, we can observe that the occurrence of atomicity violation depends on thread scheduling.

Other Concurrency-Related Programming Errors

Starvation describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress. It occurs when shared resources are made unavailable for long periods by other "greedy" and higher priority threads.

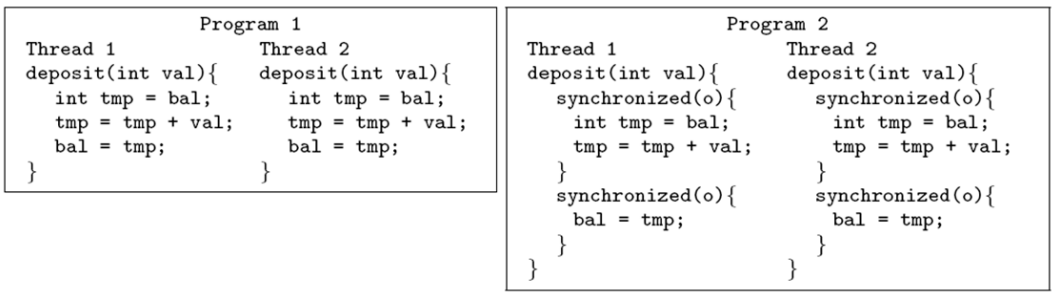
Livelock occurs when two or more processes/threads are busy in responding to each other's request while none of them can make further progress. Same as deadlock, it causes the whole or part of program to block indefinitely.

Lost Wait-Notify is another kind of common concurrency error. In multithreaded programming such as Java threads, two or more threads can use the "wait/notify/notifyAll" methods to synchronize between each other. In Pthreads, "pthread_cond_wait/pthread_cond_signal" are used instead, which work similarly. If a notifying thread calls "notify()" before the thread to be notified calls "wait()", the signal will be missed by the waiting thread. This may not be a problem if there are subsequent calls to "notify()". But if no thread calls "notify()" again, the waiting thread may wait forever because the waking up signal will never be received. Such kind of error highly depends on the thread scheduling and may not repeat in subsequent executions.

DETECTING CONCURRENCY ERRORS

In this section, we introduce the state-of-the-art research progress on detecting various concurrency errors including deadlock, data race, and atomicity violation. Most techniques for detecting concurrency errors fall into some categories of program analysis: dynamic analysis, static analysis, hybrid analysis, or model checking. We first introduce these four kinds of program analysis,

Figure 2. Examples in Java demonstrating data races and atomicity violations



then survey the detection approaches for deadlock, race condition, and atomicity violation.

Overview of Program Analysis

Dynamic Analysis

Dynamic analysis reasons about behavior of a program through observing its executions. It is usually performed by instrumenting source code (like (Wang and Stoller 2006b), bytecode (like (O'Callahan and Choi 2003)), or binary code (like (Savage et al. 1997)), and monitoring the programs' executions. The observed events can be analyzed on-line (*i.e.*, during executions) or off-line (*i.e.*, after executions terminate). To detect concurrency errors, dynamic analysis extends the traditional testing techniques. It tries to look for potential concurrency errors by searching specific patterns based on the current observed events, even the errors do not show up in the current execution paths. For example, to detect deadlock, the approaches in (Havelund 2000, Bensalem and Havelund 2005, Agarwal et al. 2005b)) search all lock acquires and releases for a potential cyclic chain. To detect data races, the approaches in (Savage et al. 1997) keep track of common locks for each shared variable, and a warning is issued when the common lock becomes empty. To improve the accuracy, the approaches in (O'Callahan and Choi 2003, Yu et al. 2005, Ratanaworabhan et al. 2009, Flanagan and Freund 2009) integrate the associated lock-set with happen-before relationships for events. The approaches to detect race condition on OpenMP (Kang et al. 2009] and CUDA (Hou et al. 2009, Boyer et al. 2008)) are very similar to the previous approach. To detect atomicity violations, the approaches in (Flanagan and Freund 2004a, Xu et al. 2005, Wang and Stoller 2006b, Wang and Stoller 2006a, Lu et al. 2006, Flanagan et al. 2008, Chen et al. 2008) search all events related with shared variable accesses and synchronization for specified violation patterns. Randomized dynamic program analysis (Sen 2008, Park and Sen 2008, Joshi et al. 2009) use two stages to detect and confirm real deadlocks, races, and atomicity violations: in the first stage, it uses an imprecise dynamic analysis to find potential errors; in the second stage, it controls a random thread scheduler to create these potential errors with high probability.

In addition, the monitoring overhead is another problem of dynamic analysis, which usually slows down the speed of programs by a factor of 2 to 100. One approach to reduce overhead is to use random sampling (Liblit et al. 2003). This approach works only when a large set of sample executions are available. Another approach is to perform selective monitoring on program region, avoid remonitoring the same code region under the same context. However, checking the equivalence of program context is also expensive. The work in (Fei and Midkiff 2006) takes approximation for variables and pointers.

Static Analysis

Static analysis makes predictions about a program's runtime behavior based on analyzing its source code. Static analysis tools like Codesurfer (Anderson et al. 2003), PREfix and PRE-fast (Bush et al.2000), ESP (Das et al. 2002), ESC/Java (Detlefs et al. 1998), and LockSmith (Pratikakis et al. 2006) aim to detect potential errors by analyzing the source code (or byte/binary code) without actually executing the programs. Type systems are proposed to avoid deadlocks (Boyapati et al. 2002, Agarwal et al. 2005b), data races (Flanagan and Freund 2000, Boyapati and Rinard 2001, Boyapati et al. 2002, Flanagan and Freund 2004b, Agarwal et al. 2005a, Sasturkar et al. 2005, Naik and Aiken 2007), and atomicity violations (Flanagan and Qadeer 2003, Agarwal et al. 2005a, Sasturkar et al. 2005, Flanagan et al. 2005, Wang and Stoller 2005). A type system is a system of programmer added types that express some correctness requirement on the variables or functions that can be involved in concurrent

error. For example, the deadlock types express a partial order on the locks, and the type rules ensure that whenever a thread holds multiple locks, the thread acquires the locks in a descending order (Boyapati et al. 2002, Agarwal et al. 2005b). However, it is a big burden for programmers to manually annotate programs with extra type information. Moreover, even the very expressive type systems may report many false positives. Inter-procedural static analysis is also used to detect potential concurrency errors (Choi et al. 2002, Engler and Ashcraft 2003). Compared to type systems, these inter-procedural analyses do not need annotations for types, but still produce numerous false positives. Static analysis can be sound, but it sacrifices accuracy and reports many false positives.

Hybrid Analysis

Static and dynamic analyses can be combined in various ways. Static analysis can be used to reduce the overhead of dynamic analysis. For example, static analysis can show that some statements are not involved in any data races or atomicity violations and hence do not need to be instrumented; this can significantly reduce the overhead of dynamic analysis by up to a factor of 20 (von Praun and Gross 2001, Choi et al. 2002, Agarwal et al. 2005a, Sasturkar et al. 2005, Agarwal et al. 2005b, Elmas et al. 2007). Dynamic analysis can help static analysis by providing more accurate runtime information. Daikon (Ernst et al. 2001) examines program executions to determine invariants to assist static analysis such as theorem proving. Static analysis and dynamic analysis can be performed interactively. Synergy (Gulavani et al. 2006) combines testing (*i.e.*, dynamic analysis) and verification (*i.e.*, static analysis) to simultaneously search for bugs and proofs. Concolic testing (Godefroid et al. 2005, Cadar et al. 2006, Majumdar and Sen 2007) runs symbolic execution simultaneously with concrete executions to generate new test inputs for better path coverage. (Chen et al. 2009) designs a hybrid approach that integrates static and dynamic analyses to attack this problem. It first performs static analysis to obtain summaries of synchronizations and accesses to shared variables. The static summaries are then instantiated with runtime values during dynamic executions to speculatively approximate the behaviors of branches that are not taken. Compared to dynamic analysis, the hybrid approach is able to detect atomicity violations in unexecuted parts of the code.

Model Checking

Model checking is a formal method for proving that a finite-state model satisfies a temporal logic property. Explicit state model checkers, such as SPIN (Holzmann 2003), enumerate the reachable states explicitly. They also utilize additional techniques such as partial order reduction (Holzmann and Peled 1994). Symbolic model checking (McMillan 1994) avoids an explicit enumeration of the state space using symbolic representations of sets of states and transitions based on Binary Decision Diagrams (BDDs) or Boolean Satisfiability Solving. Model checking can also be applied to real programs. CHESS (Musuvathi and Qadeer 2008), Java PathFinder (Visser et al. 2003), Bogor (Dwyer et al. 2005) and VeriSoft (Godefroid 1997) are such tools. Although the most rigorous automatic method to verify software, model checking faces a combinatorial blow up of the state space, commonly known as the state explosion problem. Hence it cannot handle large-scale software systems.

Approaches to Detect Deadlock

Detecting deadlock has been a decades-long problem. Recall that a deadlock occurs when all threads are blocked, each waiting for some action by one of the other threads. Thus dependences among threads and resources can be modeled by a resource allocation graph, where nodes denote

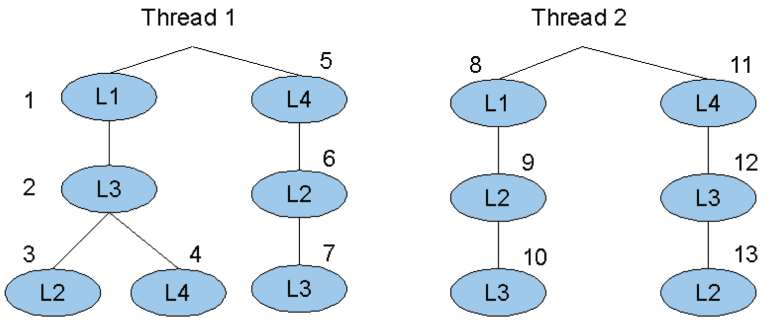
threads and exclusive resources, and edges denote allocation or wait-for relations between threads and resources. A common way to detect deadlock is to check whether the resource allocation graph contains a cycle (Silberschatz et al. 2008). Most approaches are based on it by checking against cycles, just in different ways. However, detecting deadlock thoroughly is very expensive. Large-scale software systems such as operating systems take an ostrich way, *i.e.*, assume that deadlock will not happen, hence never detect or prevent it in order to keep the performance to be efficient.

Detect Deadlocks in Multithreaded Programs

To introduce how to detect potential deadlocks in multithreaded programs, we use the GoodLock algorithm (Havelund 2000) as a typical algorithm. The GoodLock algorithm assumes that all locks are acquired and released in nested pairs, like "synchronized" in Java threads. It records a run-time lock tree for each thread as shown in Figure 3. A lock tree is a tree that represents the lock acquire order and relation for each thread as the control flows in each thread. An edge from a parent node to a child node in the lock tree indicates that the thread is currently holding a lock represented by the parent node when acquiring another lock denoted by the child node. The lock tree for a thread represents the nested pattern in which locks are acquired by the thread. Each node of the lock tree is labeled with a lock. If a thread re-acquires a lock that it already holds, its run-time lock tree does not contain a node representing the re-acquire. At the end of the execution of the program, if there exist threads $t1$ and $t2$ and locks $l1$ and $l2$ such that $t1$ acquires $l2$ while holding $l1$, and $t2$ acquires $l1$ while holding $l2$, then a warning of potential deadlock is issued, unless there is a common lock, called a *gate lock*, that is held by both threads when they acquire $l1$ and $l2$; the gate lock prevents the acquires of $l1$ and $l2$ from being interleaved in a way that leads to deadlock. For example, in Figure 3, the left branches in thread 1 and thread 2 denote two lock acquiring sequences, where *L2* and *L3* are acquired in the reverse order. However, there is no deadlock in this example because of the gate lock *L1*.

The GoodLock algorithm was extended in (Agarwal and Stoller 2006), which presents a runtime detection approach for potential deadlocks in Java programs that involves locks, semaphore, and condition variables. They extended the runtime lock tree in the GoodLock algorithm into a directed graph $G = (V;E)$, where V contains all the nodes of all the run-time lock trees, and the set E of directed edges contains (1) tree edges, which are the directed (from parent to child) edges in each of the lock trees, and (2) inter edges, which

Figure 3. A lock tree example. The small superscript numbers identify each unique lock acquire event.



are the bidirectional edges between nodes labeled with the same lock in different run-time lock trees. In order to detect potential deadlocks, they use a modified DFS (Depth First Search) to traverse the graph to look for cycles. To check deadlocks involving semaphore and condition variables, they check possible permutations of the program execution trace and report if any feasible permutation would result in a deadlock.

However, most deadlock approaches suffer from false positives which might baffle the programmers from distinguishing real bugs from false alarms. To improve the quality of deadlock checking, (Agarwal et al.) proposes more extensions that help eliminate possible false positives or label them as low severity deadlocks in the lock graph generated by static or dynamic analysis. In addition, they present a technique that effectively combines information from multiple runs of the program into a single lock graph. Such a technique may help find deadlock potentials that might not be revealed by one arbitrary run of the program because of nondeterministic scheduling. Finally they describe the use of static analysis to automatically reduce the overhead of dynamic checking for deadlock potentials.

There are many other approaches to detect deadlock. For example, (Li et al.2005) implements a tool that is integrated with operating systems and dynamically detects various types of deadlocks in application programs. Their tool runs as a system daemon and periodically scans the system for processes that have been blocked for a long time. To determine if these processes are deadlocked, the tool speculatively executes them ahead to discover their dependences. Based on this information, it constructs a general resource graph and detects deadlock by checking whether the graph contains cycles. (Williams et al. 2005) applies a flow-sensitive, context-sensitive interprocedural static analysis on detecting deadlock in Java libraries. Their analysis builds a single lock-order graph that captures locking information for an entire Java library source code and checks for cycles in the graph.

Detecting Deadlock in MPI Programs

The simplest way to detect deadlock in MPI program is to use timer when MPI program is running. If the blocking time of some process exceeds the pre-defined threshold, those processes are announced to be in deadlock. Timer approach is easy to implement and does not impose too much overhead, but is difficult to set and adjust the threshold, and it may potentially report many false positives.

Dependency graph is one of the major approaches to detect deadlocks in MPI programs. It is usually implemented as a dynamic approach. In (Hilbrich et al. 2009), based on the concept of AND model, OR model and the combination of AND-OR model, dependency graph expresses the waiting relation between various processes at specific time. If part of the dependency graph satisfies the pre-defined deadlock conditions such as a circle or some kind of knot, then that part of the dependency graph is considered in a deadlock. This approach can only find deadlock happening during execution for specific schedules, but will not report all potential deadlock for all schedules.

(Luecke et al. 2002) uses a dynamic handshaking approach to detect potential and actual deadlocks. Handshaking code (handshake_send and handshake_recv) is statically instrumented before each MPI send/receive call in the source program and when the instrumented program is compiled and run, the dynamic monitor tracks the handshaking code to match a send and a receive. A handshake is a matching pair of instrumented handshake method calls for a send event and a receive event. If a handshake is not observed for each send or receive call after a user-defined time, then it reports a potential or actual deadlock warning for that send or receive call depending on the scenarios. They summarize a collection of situations where a actual deadlock will occur

Algorithm 11.

Let *held(t)* be the set of locks held by thread *t*.
For each v, initialize *C(v)* to the set of all locks.
On each access to *v* by thread *t*,
C(v):= C(v) ∩ locks_held(t);
if *C(v)* == { }, then issue a warning.

and a set of possible deadlock situations if the handshake is not observed. In (Vo et al. 2008), a dynamic formal verification approach is proposed to detect potential deadlocks in MPI programs. In the proposed approach, the execution of MPI program is under control of an interleaving scheduler where nondeterministic constructs are explored for all possible interleavings.

Approaches to Detect Race Condition

Detecting Race Condition in Multithreaded Programs

Many static and dynamic approaches have been proposed to detect race conditions in multithreaded programs. They are based on either lockset analysis or happen-before order. However, detecting race conditions is not an easy problem, which has been proved to be NP-hard (Netzer and Miller 1992).

We use the Eraser algorithm (Savage et al. 1997) shown above as a typical algorithm of lockset analysis. As a dynamic approach, Eraser checks all shared-memory accesses against a simple locking policy, *i.e.*, all accesses to a shared variable should be protected by a common lock. As shown above, for each shared variable *v*, Eraser maintains a set *C(v)* of candidate locks for *v*. This set contains those common locks that have protected *v* in the execution so far. That is, a lock *l* is in *C(v)* if, in the execution up to that point, every thread that has accessed *v* was holding *l* at the moment of the access. When *v* is initialized, its candidate set *C(v)* is considered to hold all possible locks. Whenever the variable is accessed, Eraser updates *C(v)* with the intersection of *C(v)* and the set of locks held by the current thread. This process, called lockset refinement, ensures that any lock consistently protecting *v* is contained in *C(v)*. If some lock *l* consistently protects v, it will remain in *C(v)* during the refinement. If *C(v)* becomes empty, which indicates that there is no lock consistently protecting *v*, Eraser will report a warning of race condition on *v.*

Figure 4 illustrates how the Eraser algorithm is applied to detect potential data races. The left two columns contain two threads. Thread 2 runs after thread 1. The third and forth column reflect the corresponding locks held by the current thread and the set of candidate locks *C(v),* respectively. This example has two locks, so *C(v)* starts containing both of them. When *v* is accessed by thread 1 while holding lock *o1*, *C(v)* is refined to contain that lock. Later, *v* is accessed again by thread 2 while holding only *o2*. The intersection of the singleton sets {o1} and {o2} is the empty set, which indicates that no lock protects *v*. Hence, a race condition is reported on *v.*

However, the simple locking discipline may report many false positives. More improvements are designed for better accuracy. The extended Eraser algorithm proposed in (Savage et al. 1997) distinguishes states such as variable initialization (*i.e.*, *Exclusive*), initialized then read-only (*i.e.*, *Shared*), and read/write by multiple threads (*i.e.*, *Shared-Modified*). The state transitions are shown in Figure 5. When a variable is first allocated, it will be in the *Virgin* state, which implies that the variable is not shared among multiple threads yet. Once it has been accessed by the first thread, it enters the *Exclusive* state. Any following reads and writes from the same thread do not change the variable's state and do not update *C(v)*. With a read access from a different thread, the state is changed to *Shared* from *Exclusive*. In the *Shared* state, *C(v)* is updated, but no data race will be reported because this is a read-shared situation. Alternatively, a write access from a different thread changes the state from *Exclusive* or *Shared* to the

Figure 4. An example illustrating the Eraser algorithm

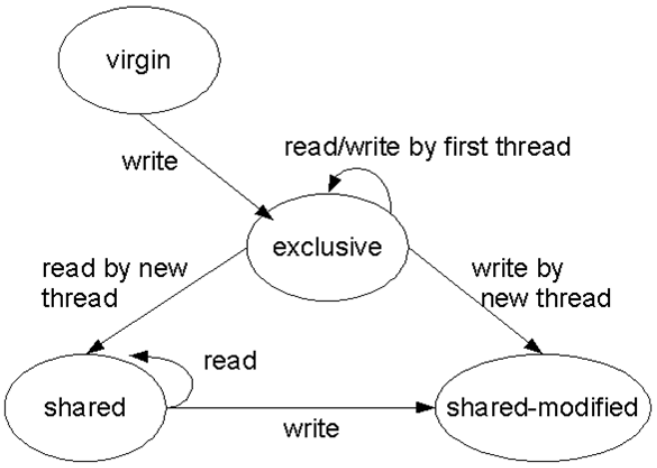
Thread 1	Thread 2	Locks held	C(v)
			{o1, o2}
lock(o1);		{}	{o1, o2}
v:= v + 1;		{o1}	{o1}
unlock(o1);		{}	{o1}
	lock(o2);	{}	{o1}
	v:= v -1;	{o2}	{}
	unlock(o2);	{}	{}

Shared-Modified state, in which *C(v)* is updated and races are reported as the original Eraser algorithm would.

To improve accuracy and achieve lower runtime overhead of dynamic analysis, (Choi et al. 2002) proposes an efficient and precise dynamic detection approach on multithreaded programs. They take into account a *weaker-than* relationship that allows dynamic analysis to consider only portion of memory accesses rather than monitor all memory accesses. Given two memory access events e_i and e_j, if for every subsequent access e_k, isRace(e_j, e_k) implies isRace(e_i, e_k), then e_i is more *weakly protected* from data race than e_j, or in another word, e_i is weaker than e_j. With the weaker-than relation, they only need to store information about the weaker one of two events, which reduces both space and time overhead. In addition, caching technique is used to detect and remove redundant accesses thus further reduce space overhead. Before running the dynamic analysis, a static analysis is performed to identify all possible statements involving data races. Thus, the dynamic analysis will not need to monitor the irrelevant statements. The static analysis uses inter-thread control flow graph, points-to analysis, and extended escape analysis to help identify data races more accurately.

The hybrid approach in (O'Callahan and Choi 2003) further extends the Eraser algorithm (Savage et al.1997) and the work in (Choi et al. 2002) with happens-before order relationship to reduce false positives. "Hybrid" means that the approach integrates the lockset-based analysis and the happen-before order relationship. The happens-before relationship was originally defined by

Figure 5. State transition of the extended Eraser algorithm



Lamport as a partial order on events occurring in a distributed system (Lamport1978). Informally, a pair of events (e_i, e_j) has happen-before relationship if (1) e_i and e_j are events in the same thread, and e_i occurs before e_j; or (2) If e_i is the sender of a message and e_j is the receiver of the message; or (3) e_i and e_j have transitive happen-before order. The hybrid race detection helps reduce many false positives reported by the Eraser algorithm alone.

Based on the lockset analysis and happen-before orders, there are many other approaches to detect race conditions. (Engler and Ashcraft 2003) proposes a static technique that uses flow-sensitive, inter-procedural static analysis to check race conditions and deadlocks. One of the contributions is to rank all warnings using various criteria such as simple checking, simple statistical measure, and precise statistical measure. (Yu et al. 2005) proposes an adaptive tracking scheme that can reduce the runtime monitoring overhead to at most 3x slowdown of the original program. In addition, a post-processing step is performed to rank race warnings with the most likely ones on top. Their implementation on Microsoft.NET platform exploits the benefits of Common Language Runtime so that the instrumentation happens on the virtual machine level and no modification on the original programs is needed. They track the happen-before order through the vector clock attached to each memory access. Many other related work have been introduced in Section 5.1.

Detecting Message Race in MPI Programs

(Netzer et al. 1996) is one of the first papers that explore the problem of detecting message races in MPI programs. The proposed approach is a dynamic algorithm with two passes that can handle long running MPI programs regardless their execution length. Specifically, it uses a vector timestamp to track the happen-before relation between different send events then determines the possible concurrent send events. In the first pass, for each send event that has been matched by a receive event in the real execution, it tries to find out all possible send events that could be matched by that receive event. The second pass uses the information reported by the first pass to validate the message races.

(Park et al., 2007) detects all potential race conditions by checking concurrent communication between processes. It uses vector timestamps to determine possible concurrency between send/receive events in MPI programs. To capture all points-to-points MPI function calls, it replaces all the original calls with the profiling calls defined by MPI profiling interface.

Approaches to Detect Atomicity Violation

The approaches to detect atomicity violations root in the detection of serializability in database systems. The algorithms can be classified into two main categories: the approaches based on Lipton's reduction theorem (Lipton 1975), and the approaches based on detecting unserializable patterns.

The approach in (Flanagan and Qadeer 2003) is a typical algorithm based on Lipton reduction theorem for analyzing atomicity in multithreaded programs. The theory of reduction is based on the notion of right-mover and left-mover actions. In the reduction algorithm, events are classified according to their commutativity properties. An event is a *right-mover* if, whenever it appears immediately *before* an event of a different thread, the two events can be swapped (*i.e.*, they can be executed in the opposite order without blocking) without changing the resulting state. A *left-mover* is defined analogously. For example, if an event e_1 of a thread is a lock acquire, its immediate successive event e_2 from another thread cannot be a successful acquire or release of the same lock because an acquire would block and a release would fail (in Java, it would throw an exception). Hence, e_1 and e_2 can be swapped without

affecting the result, so e_1 is a right-mover. Lock release events are left-movers for similar reasons. An event is a *both-mover* if it is both a left-mover and a right-mover. For example, if there are only read events (no write) on a given variable, the read events commute in both directions with all events, so these read events are both-movers. Events not known to be left or right movers are non-movers. For Java programs, a classification of events can be conveniently obtained based on synchronization operations. Lock acquire events are right-movers. Lock release events are left-movers. Race-free reads and race-free writes are both-movers. An execution path is considered to be atomic if it contains sequence of right-movers, followed by at most one non-mover action and then a sequence of left-movers.

The approach in (Wang and Stoller 2006b) is a typical algorithm based on detecting unserializable patterns. The algorithm checks atomicity violations by permuting the order of events that are consistent with the synchronization events. Explicitly enumerating these permutations would be prohibitively expensive. Instead, they look for unserializable patterns of operations from these events. An unserializable pattern is a sequence in which operations from different threads are interleaved in an unserializable way. As an example, the following table shows four unserializable patterns when multiple threads share exactly one variable. The more complex cases, such as multiple shared variables, are introduced in (Wang and Stoller 2006b).

From top left to bottom right, these four patterns shown in Algorithm 12 are described below.

- A read in one transaction occurs between two writes in another transaction.
- A write in one transaction occurs between two reads in another transaction.
- A write in one transaction occurs between a write and a subsequent read in another transaction.
- The final write in one transaction occurs between a read and a subsequent write in another transaction.

Many other approaches have been proposed to check atomicity violations. (Vaziri et al. 2006) takes a similar approach to check atomicity problems by searching non-serializable interleaving scenarios. In addition, they present a language extension called *atomic set of locations* to allow programmers to specify existence of properties between fields in objects. They use an inter-procedural static analysis technique that automatically infers those points where synchronization is missing. (Xu et al. 2005) proposes a tool to detect serializability violation (*i.e.*, atomicity violation). It can automatically infer atomic regions where serializability criterion must be met. (Lu et al. 2006) proposes an approach to detect atomicity violations based on access interleaving invariants that are observed in multiple runs of the concurrent program. The access interleaving invariants imply the programmers' assumptions about the atomicity of certain code regions. (Farzan and Madhusudan 2008) proposes a space-efficient monitoring algorithm for checking atomicity violations. The algorithm builds a conflict-graph through dynamic monitoring the program and then reduces the conflict graph into a summarized conflict graph to check atomicity problem. (Chen et al. 2009) presents a hybrid approach that complements dynamic analysis with static speculation to detect potential atomicity violations in concurrent Java programs. Their approach first performs static analysis to obtain summaries of synchronizations

Algorithm 12.

W(x) R(x) W(x)	R(x) W(x) R(x)
W(x) W(x) R(x)	R(x) FW(x) W(x)

and accesses to shared variables. The static summaries are then instantiated with runtime values during dynamic executions to speculatively approximate the behaviors of branches that are not taken. Compared to dynamic analysis, the hybrid approach is able to detect atomicity violations in unexecuted parts of the code. Compared to static analysis, the hybrid approach produces fewer false alarms. More approaches about checking atomicity violations appear in Section 5.1.

TOOLS TO DETECT CONCURRENCY ERRORS

There are many methods and tools that have been developed for ensuring the correctness of multithreaded and MPI programs. Here we give a brief overview of the widely used or well-known tools. The commercial debugging tools include PGI Tools, TotalView, Intel Message Checker, Allinea Distributed Debugging Tool (DDT), and Nvidia Nexus and CUDA GDB. The open source community offers Eclipse Parallel Tools Platform (PTP), MPI-CHECK, Umpire, MARMOT, ISP, MPI-Spin, and MS CHESS.

Commercial Debugging Tools

PGDBG (pgi2009) is developed by the Portland Group, Inc. (PGI) as a symbolic debugger for Fortran, C/C++, and assembly language programs. It provides most typical debugger features such as breakpoint setting, single instruction stepping, visualization of application variables, memory locations, and registers. It supports debugging parallel applications using Pthreads or Windows threads, OpenMP and MPI, as well as hybrid programming paradigms that combine two or more of aforementioned parallel programming interfaces.

Intel Message Checker (DeSouza et al. 2005) is an MPI correctness tool that helps ensure the correctness of MPI programs. It can detect many MPI errors such as mismatched arguments and buffers (size and type), race conditions, resources leaking, overlapped read/write to the same message buffers, message checksum errors, and potential deadlocks. It comes with a user-friendly graphical user interface. However, the trace files generated by Intel Message Checker can be very large thus it may be inefficient to analyze the trace files.

Intel Thread Checker (inta) is a data race and deadlock detection tool for 32-bit and 64-bit multithreaded and OpenMP applications in Windows and Linux. However, its overhead could be as high as 200x of the original program's performance which makes it hard to be adopted on long-running server programs (Sack et al. 2006).

Intel Trace Analyzer and Collector (intb) contains an MPI correctness checking library that can dynamically detect many communication errors including deadlocks, data corruption, or errors regards to MPI parameters, data types, buffers, communicators, point-to-point messages, and collective operations. The tool supports setting debugger breakpoints to greatly help the analysis. It can also instrument the original source code of MPI programs to monitor data types and MPI calls with their wrapper calls and compile the instrumented programs with their checking library. It can scale to large systems with many processes running concurrently.

TotalView (Kingsbury 2007) is a commercial MPI tool providing industrial level of debugging support. It can debug one or many processes and/or threads with complete control over program execution. In addition, it has the capability of reproducing programs crashes. It can visualize the state of the running program for efficient debugging of memory errors and leaking and diagnosing subtle problems like deadlocks and race conditions. It works with C, C++, and FORTRAN applications. Its latest extension supports debugging CUDA programs.

Allinea Distributed Debugging Tool (DDT) (DDT) is another commercial debugging tool with graphical user interface that supports both centralized and distributed debugging. It supports

C/C++, FORTRAN, OpenMP, MPI, Pthreads, Windows threads, and CUDA. DDT supports fine-grained control over the target program to examine the program states in more effective ways during execution. In addition, with the support of controlling individual threads and/or processes separately or collectively, it allows programmer to examine data across threads/processes. The programmable STL Wizard that comes with DDT enables the programmers to view C++ Standard Template Library structures such as lists, maps, sets, pairs, and strings.

Nvidia introduces **Nexus** (NVIDIA Nexus, 2009) in October, 2009, which is a tool integrated into Microsoft Visual Studio 2008 to debug, profile, and analyze CUDA programs. **CUDA-GDB** (CUDA-GDB) is an extension of the GNU Project Debugger (GDB) to debug CUDA programs on both 32-bit and 64-bit Linux. CUDA-GDB supports debugging both host and GPU code. CUDA-GDB runs only on CUDA-capable GPUs with the compute capability later than 1.1.

Open-source Debugging Tools

Eclipse Parallel Tools Platform (PTP) (Watson et al. 2006) is a plug-in to Eclipse IDE and contains many productivity tools to help programmers launch, control, monitor, and debug MPI programs. It is also a framework for developers to integrate external tools so that they can take advantage of the user interface components and services provided by both PTP and Eclipse. The latest version is 4.0 published at the end of June, 2010.

PTP manages MPI source programs as projects. With PTP, programmers can utilize all the productivity features in Eclipse to develop their programs such as syntax highlighting, static code checking, automatic build, and error location. PTP uses the resource manager to manage and control the resources required for launching a parallel job. For example, given a cluster with Open MPI installed, the Open MPI runtime system would be considered the resource manager. Once the programmers configure the resource manager, they can launch, monitor, and control their programs on the target resource regardless whether the target resource is remote or local. Programmers can also launch a MPI program in debug mode of PTP. In the debug mode, PTP switches Eclipse to Parallel Debug View which allows programmers to suspend processes, and visualize the detailed information about the suspended processes such as stack frame content and local variables values. Parallel break point is another feature supported in PTP. Programmers can either set global breakpoints in the source program that apply to all processes in any job or set local breakpoints that apply only to a specific set of processes (which can include the root set) for a single job. The difference is that a global breakpoint remains in effect between job launches while local breakpoints are removed when the job completes.

MARMOT (Krammer et al. 2004) is a runtime detection tool that samples the MPI-calls invoked in the runtime and subsequently checks the correct usage of these calls and their arguments. It can be used in conjunction with traditional sequential debuggers such as GDB to help the programmers pinpoint the bugs. It supports both C and FORTRAN languages. After the runtime monitoring, it generates a human-readable log file which can be analyzed for reporting the violations of MPI specification. It can also check the call stack trace for potential deadlocks based on a time-out mechanism. However, the deadlock detection could report false positives since some calls would take longer than expected due to physical network problems or other reasons.

UMPIRE (Vetter and de Supinski 2000) is another dynamic tool to analyze MPI programming errors using a profiling interface like MARMOT. It can detect deadlocks by combining time-out mechanism and dependency graphs together.

ISP (Vo et al. 2008) is a tool that dynamically verifies MPI programs. It consists of three parts: profiler, scheduler, and checker. The profiler wraps the MPI-related function calls inside their wrap-

per functions and intercepts these MPI function call events for later processing. The scheduler carries out all possible schedulings while using POE (Partial Order Reduction) to remove the redundant states and minimize the state space. The scheduler explicitly considers and handles MPI-specific properties including wildcard receives, barriers, which are important to scheduling. The checker checks each permuted ordering for possible violations of properties such as deadlocks and resource leaking.

MPI-Spin (Siegel 2007) is an extension to the popular model checker Spin (Holzmann 2003). It adds to Spin's input language a number of functions, types, and constants for modeling MPI programs. By default, MPI-Spin checks a number of generic correctness properties in MPI programs. These properties include (1) the program cannot deadlock, (2) there are never two incomplete requests whose buffers intersect non-trivially, (3) the total number of outstanding requests never exceeds a specified bound, (4) when MPI_Finalize is called, there are no request objects allocated for and there are no buffered messages destined for the calling process, and (5) the size of an incoming message is never greater than the size of the receive buffer. In addition, MPI-Spin can check application-specific user-written properties that are formulated in temporal logic. It provides extensive support for symbolic execution, making it possible to verify that a program behaves correctly on all possible inputs.

MPI-CHECK (Luecke et al. 2003) is an open-source tool developed for checking MPI programs in FORTRAN and C/C++ languages. It provides both compile-time and runtime checks on the target MPI programs. With macro and the wrappers for MPI routines, compiler can invoke MPI-CHECK to statically check the data type of each argument, the intent of each argument, and the number of arguments in the routines. For the runtime checking, MPI-CHECK first instruments the original source program and links it with their modules to produce an instrumented executable. When the resulting executable runs, instrumented code emits events to MPI-CHECK and possible error/warning messages are reported if found. MPI-CHECK run-time checker checks all MPI-1 and MPI-2 routines for problems such as buffer data type inconsistency, buffer out of bounds, improper placement of MPI_INIT, illegal message length, and invalid MPI rank. However, MPI-CHECK places significant performance overhead for the target programs under test which could prevent it from being deployed to large scale real-world programs.

Library-Based MPI Debugging Tools

Some level of debugging support is also provided in many MPI implementations such as OpenMPI (Graham et al. 2005), LAM-MPI (Burns et al. 1994) and MPICH (Worringen et al.2002). They either provide additional compile-time flags for checking MPI function calls, or come with a separate profiler/monitor for runtime testing. OpenMPI provides several compiler flags for statically checking some properties of function calls such as null parameter passing, checking potential resource leaking, displaying runtime configuration such as MCA parameters and their values during MPI_INIT call, and printing stack trace when MPI_ABORT() is invoked. LAM-MPI implementation has a GUI-based tool called XMPI that allows programmers to debug and visualize the running MPI programs. It can take snapshots of runtime synchronization events and retrieve detailed information about MPI events such as communicator, data type, tag, and message content and length. MPICH provides support for external debuggers such as TotalView and DDT.

Summary and Comparison of Tools

To summarize the differences and commonality between these tools, we present a comparison in Table 1.

Table 1. Comparison of MPI error prevention and detection tools. **S** *denotes using static analysis;* **D** *denotes using dynamic analysis which includes both online analysis and offline trace-based analysis;* **H** *denotes using both Static and Dynamic; × denotes Not Available. The column "MPI-specific problems" include resource leak, mismatched buffer size and type, null parameter passing etc. None of the tools is able to support detecting atomicity violations so far.*

	Deadlock Detection	**MPI-specific problems**	**Message Race Detection**	**Data Race Detection**	**Runtime Debugging Support**
Allinea Distributed Debugging Tool (DDT)	×	×	×	×	Pthreads Windows Threads OpenMP MPI in C MPI in Fortran CUDA
Intel Message Checker	×	MPI in C: **D** MPI in Fortran: **D**	MPI in C: **D** MPI in Fortran: **D**	×	×
Intel Thread Checker	OpenMP: **D** Pthreads: **D**	×	×	OpenMP: **D** Pthreads: **D**	×
Intel Trace Analyzer and Collector	MPI in C: **D** MPI in Fortran: **D**	MPI in C: **D** MPI in Fortran: **D**	MPI in C: **D** MPI in Fortran: **D**	×	MPI in C MPI in Fortran
ISP	MPI in C: **D**	MPI in C: **D**	×	×	×
MARMOT	MPI in C: **D** MPI in Fortran: **D**	MPI in C: **D** MPI in Fortran: **D**	×	×	×
MPI-CHECK	MPI in C: **D** MPI in Fortran: **D**	MPI in C: **H** MPI in Fortran: **H**	×	×	×
MPI-Spin	MPI in C: **S**	MPI in C: **S**	×	×	×
Nvidia Nexus	×	×	×	×	CUDA
PGDBG	×	×	×	×	Pthreads Windows Threads OpenMP MPI in C MPI in Fortran
Eclipse Parallel Tools Platform (PTP)	×	×	×	×	OpenMP MPI in C MPI in Fortran
TotalView	×	×	×	×	Pthreads WindowsThreads OpenMP MPI in C MPI in Fortran
UMPIRE	MPI in C: **D** MPI in Fortran: **D**	×	×	×	×
LAM-MPI	×	×	×	×	MPI in C MPI in Fortran
MPICH	×	×	×	×	×
Open MPI	×	MPI in C: **S** MPI in Fortran: **S**	×	×	×

As we can see from this Table 1, most current tools are using dynamic analysis to find out and correct potential errors. This could be partly attributed to the precision of dynamic analysis which leads to much fewer false positives. However, due to the nature of dynamic analysis, users might find difficulty in adopting them on many large programs.

CONCLUSION

We give a comprehensive introduction to multithreaded and message passing programming, including the approaches to detect deadlock, race condition, and atomicity violation, as well as the widely used tools to debug concurrent programs. With the prevalence of multi-core CPU and many-core co-processor, concurrent programming is becoming more popular and bringing significant effect on the practice and research of software engineering. The research on detecting concurrency errors is attracting more and more attentions. With the effort of industry and academia, we expect that the next generation of concurrent programming will be easier for coding and debugging.

REFERENCES

Agarwal, R., Bensalem, S., Farchi, E., Havelund, K., Nir-Buchbinder, Y., & Stoller, S. D. (in press). Detection of deadlock potentials in multi-threaded programs. *IBM Journal of Research and Development*.

Agarwal, R., Sasturkar, A., Wang, L., & Stoller, S. D. (2005a). Optimized run-time race detection and atomicity checking using partial discovered types. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM Press.

Agarwal, R., Wang, L., & Stoller, S. D. (2005b). Detecting potential deadlocks with static analysis and runtime monitoring. *In Proceedings of the Parallel and Distributed Systems: Testing and Debugging (PADTAD)*. Springer-Verlag.

Anderson, P., Reps, T., Teitelbaum, T., & Zarins, M. (2003). Tool support for fine-grained software inspection. *IEEE Software*, *20*(4), 42–50.

ANL MPI Using MPI in Simple Programs http://www.mcs.anl.gov/research/projects/mpi/usingmpi/examples/simplempi/main.htm

Bensalem, S., & Havelund, K. (2005). Scalable deadlock analysis of multi-threaded programs. In S. Ur (Ed.), *IBM Verification Conference*, (LCNS 3875), Haifa, Israel. Springer.

Borthakur, D. (2007). *The Hadoop Distributed File System: Architecture and design*. The Apache Software Foundation.

Boyapati, C., Lee, R., & Rinard, M. (2002). Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the 17th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA),* (pp. 211–230). ACM Press.

Boyapati, C., & Rinard, M. C. (2001). A parameterized type system for race-free Java programs. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, (pp. 56–69). ACM Press.

Boyer, M., Skadron, K., & Weimer, W. (2008). Automated dynamic analysis of CUDA programs. In *Proceedings of the Third Workshop on Software Tools for Multi-Core Systems*.

Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., & Houston, M. (2004). Brook for GPUs: Stream computing on graphics hardware. *ACM Transactions on Graphics*, *23*(3), 777–786.

Burns, G., Daoud, R., & Vaigl, J. (1994). LAM: An open cluster environment for MPI. In *Proceedings of Supercomputing Symposium*, (pp. 379–386).

Bush, W., Pincus, J. D., & Sielaff, D. J. (2000). A static analyzer for finding dynamic programming errors. *Software, Practice & Experience*, *30*(7), 775–802.

Cadar, C., Ganesh, V., Pawlowski, P. M., Dill, D. L., & Engler, D. R. (2006). EXE: Automatically generating inputs of death. In *Proceedings of the 13th ACM conference on Computer and communications security (CCS)*. ACM Press.

Chen, F., Serbanuta, T. F., & Rosu, G. (2008). jPredictor: A predictive runtime analysis tool for Java. In *Proceedings of the 30th international conference on Software engineering (ICSE '08),* pages 221–230. ACM.

Chen, Q., Wang, L., Yang, Z., & Stoller, S. D. (2009). HAVE: Integrated dynamic and static analysis for atomicity violations. *In Proceedings of International Conference on Fundamental Approaches to Software Engineering (FASE)*, (LNCS 5503), (pp. 425–439). Springer.

Choi, J.-D., Lee, K., Loginov, A., O'Callahan, R., Sarkar, V., & Sridharan, M. (2002). Efficient and precise datarace detection for multi-threaded object-oriented programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI),* (pp. 258–269). ACM Press.

Cuda, N. V. I. D. I. A. *CUDA Compute Unified Device Architecture Programming Guide.* http://www.nvidia.com/object/cuda_home.html

CUDA-GDB. CUDA-GDB: The NVIDIA CUDA Debugger. http://developer.download.nvidia.com/compute/cuda/2_1/cudagdb/CUDA_GDB_User_Manual.pdf

Das, M., Lerner, S., & Seigle, M. (2002). Esp: Path-sensitive program verification in polynomial time. In *proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

DDT Allinea Software, Allinea DDT The Distributed Debugging Tool. http://www.allinea.com/index.php?page=48.

Dean, J., & Ghemawat, S. (2008). MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, *51*(1), 107–113.

DeSouza, J., Kuhn, B., de Supinski, B. R., Samofalov, V., Zheltov, S., & Bratanov, S. (2005). Automated scalable debugging of MPI programs with Intel message checker. In *Proceedings of the Second International Workshop on Software Engineering for High Performance Computing System Applications*, (pp. 78–82). New York: ACM.

Detlefs, D. L., Leino, K. R. M., Nelson, G., & Saxe, J. B. (1998). Extended static checking. Research Report 159, Compaq SRC. Retrieved from http://www.research.compaq.com/SRC/esc/

Dwyer, M., Hatcliff, J., Hoosier, M. & Robby (2005). *Building your own software model checker using the Bogor extensible model checking framework*. Computer Aided Verification (CAV).

Elmas, T., Qadeer, S., & Tasiran, S. (2007). Goldilocks: A race and transaction-aware Java runtime. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM Press.

Engler, D., & Ashcraft, K. (2003). RacerX: Effective, static detection of race conditions and deadlocks. In *Proceedings of ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. ACM Press.

Ernst, M. D., Cockrell, J., Griswold, W. G., & Notkin, D. (2001). Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, *27*(2), 99–123.

Fei, L., & Midkiff, S. P. (2006). Artemis: Practical run-time monitoring of applications for execution anomalies. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, (pp. 84–95). ACM Press.

Flanagan, C., & Freund, S. (2000). Type-based race detection for Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, (pp. 219–232). ACM Press.

Flanagan, C., & Freund, S. N. (2004a). Atomizer: A dynamic Atomicity checker for multithreaded programs. *In Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, (pp 256–267). ACM Press.

Flanagan, C., & Freund, S. N. (2004b). *Type inference against races*. In Static Analysis Symposium, (LNCS 3148). Springer-Verlag.

Flanagan, C., & Freund, S. N. (2009). FastTrack: Efficient and precise dynamic race detection. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI),* (pp. 121–133). ACM Press.

Flanagan, C., Freund, S. N., & Qadeer, S. (2005). Exploiting purity for Atomicity. *IEEE Transactions on Software Engineering*, *31*(4).

Flanagan, C., Freund, S. N., & Yi, J. (2008). Velodrome: A sound and complete dynamic Atomicity checker for multithreaded programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, (pp. 293–303), New York: ACM.

Flanagan, C., & Qadeer, S. (2003). A type and effect system for Atomicity. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM Press.

Godefroid, P. (1997). Model checking for programming languages using Verisoft. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*. ACM Press.

Godefroid, P., Klarlund, N., & Sen, K. (2005). DART: Directed Automated Random Testing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM Press.

Graham, R. L., Woodall, T. S., & Squyres, J. M. (2005). Open MPI: A flexible high performance MPI. In *Proceedings, 6th Annual International Conference on Parallel Processing and Applied Mathematics*, Poznan, Poland.

Gulavani, B. S., Henzinger, T. A., Kannan, Y., Nori, A. V., & Rajamani, S. K. (2006). Synergy: A new algorithm for property checking. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '06/FSE-14)*. ACM Press.

Hadoop Hadoop Open Source MapReduce Platform. http://lucene.apache.org/hadoop/

Havelund, K. (2000). Using runtime analysis to guide model checking of Java programs. In *Proceedings of the 7th Int'l. SPIN Workshop on Model Checking of Software,* (LNCS 1885), (pp. 245–264). Springer-Verlag.

Hilbrich, T., de Supinski, B. R., Schulz, M., & Müller, M. S. 2009. A graph based approach for MPI deadlock detection. In *Proceedings of the 23rd International Conference on Supercomputing,* Yorktown Heights, NY, (pp. 296-305). New York: ACM.

Holzmann, G. J. (2003). *The SPIN model checker*. Addison-Wesley.

Holzmann, G. J., & Peled, D. (1994). An improvement in formal verification. In *Proceedings of International Conference on Formal Description Techniques (FORTE'94), volume 6 of IFIP Conference Proceedings*, (pp. 197–211). Chapman & Hall.

Hou, Q., Zhou, K., & Guo, B. (2009). Debugging GPU stream programs through automatic dataflow recording and visualization. In *ACM SIGGRAPH papers* (pp. 1–11). New York: ACM.

inta Intel Thread Checker. http://software.intel.com/en-us/intel-thread-checker

intb Intel Trace Analyzer. http://software.intel.com/en-us/intel-trace-analyzer

Joshi, P., Park, C.-S., Sen, K., & Naik, M. (2009). A randomized dynamic program analysis technique for detecting real deadlocks. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, (pp. 110–120). ACM Press.

Kang, M.-H., Ha, O.-K., Jun, S.-W., & Jun, Y.-K. (2009). A tool for detecting first races in OpenMP programs. In *Proceedings from the 10th International Conference Parallel Computing Technologies*, (pp. 299–303).

Kingsbury, B. (2007). Organizing processes and threads for debugging. In *Proceedings of the 2007 ACM Workshop on Parallel and Distributed Systems: Testing and Debugging*, (pp. 21–26), New York: ACM.

Krammer, B., Muller, M. S., & Resch, M. M. (2004). MPI application development using the analysis tool MARMOT. In *Proceedings from the International Conference on Computational Science*, (pp 464–471).

Liblit, B., Aiken, A., Zheng, A., & Jordan, M. I. (2003). Bug isolation via remote program sampling. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, (pp. 141–154). ACM Press.

Lu, S., Tucek, J., Qin, F., & Zhou, Y. (2006). AVIO: Detecting Atomicity violations via access interleaving invariants. In *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM Press.

Luecke, G., Chen, H., Coyle, J., Hoekstra, J., Kraeva, M., & Zou, Y. (2003). MPI-CHECK: A tool for checking FORTRAN 90 MPI programs. *Concurrency and Computation*, *15*(2), 93–100.

Luecke, G., Zou, Y., Coyle, J., Hoekstra, J., & Kraeva, M. (2002). Deadlock detection in MPI programs. [John Wiley & Sons.]. *Concurrency and Computation*, *14*(11), 911–932.

Majumdar, R., & Sen, K. (2007). Hybrid concolic testing. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*. IEEE Press.

McCool, M., Du Toit, S., Popa, T., Chan, B., & Moule, K. (2004). Shader algebra. *ACM Transactions on Graphics*, *23*(3), 787–795.

McMillan, K. L. (1994). *Symbolic model checking*. Boston: Kluwer Academic Publishers.

Musuvathi, M., & Qadeer, S. (2008). Fair stateless model checking. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, (pp. 362–371). ACM Press.

Naik, M., & Aiken, A. (2007). Conditional must not aliasing for static race detection. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM Press.

NCSA-CUDA NCSA CUDA Tutorial. http://www.ncsa.illinois.edu/UserInfo/Training/Workshops/CUDA/presentations/tutorial-CUDA.html

Netzer, R. H. B., Brennan, T. W., & Damodaran-Kamal, S. K. (1996). Debugging race conditions in message-passing programs. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, (pp. 31–40). New York: ACM.

Nexus, N. V. I. D. I. A. 2009 - Visual Studio-based GPU Development, http://developer.nvidia.com/object/nexus.html

O'Callahan, R., & Choi, J.-D. (2003). Hybrid dynamic data race detection. In *Proceedings of ACM SIGPLAN 2003 Symposium on Principles and Practice of Parallel Programming (PPoPP)*, (pp. 167–178). ACM.

OpenCL Open Computing Language (OpenCL). http://www.khronos.org/opencl/.

Park, C.-S., & Sen, K. (2008). Randomized active Atomicity violation detection in concurrent programs. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'08)*, (pp. 135–145). New York: ACM.

pgi2009 (2009). PGI Tools Guide. http://www.pgroup.com/doc/pgitools.pdf

Pratikakis, P., Foster, J. S., & Hicks, M. (2006). Locksmith: Context-sensitive correlation analysis for race detection. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, (pp. 320–331). ACM.

RapidMind RapidMind. http://www.rapidmind.com/.

Ratanaworabhan, P., Burtscher, M., Kirovski, D., Zorn, B., Nagpal, R., & Pattabiraman, K. (2009). Detecting and tolerating asymmetric races. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP)*, (pp. 173–184). ACM Press.

Sack, P., Bliss, B. E., Ma, Z., Petersen, P., & Torrellas, J. (2006). Accurate and efficient filtering for the Intel thread checker race detector. In *Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, (pp. 34–41) New York: ACM.

Sasturkar, A., Agarwal, R., Wang, L., & Stoller, S. D. (2005). Automated type-based analysis of data races and Atomicity. In *Proceedings of the ACM SIGPLAN 2005 Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM Press. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P. & Anderson, T.E. (1997). Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, *15*(4), 391–411.

Sen, K. (2008). Race directed random testing of concurrent programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, (pp. 11–21). ACM Press.

Siegel, S. F. (2007). Verifying parallel programs with MPI-SPIN. In *Proceedings of the 14th European PVM/MPI User's Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, (pp. 13–14). Berlin/Heidelberg: Springer-Verlag.

Silberschatz, A., Galvin, P. B., & Gagne, G. (Eds.). (2008). *Operating system concepts* (8th ed.). John Wiley & Sons.

UPC UPC. UPC language specifications. http://upc.gwu.edu/docs/upc_spec_1.1.1.pdf

Vetter, J. S., & de Supinski, B. R. (2000). Dynamic software testing of MPI applications with UMPIRE. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing (CDROM)*, (p. 51). Washington, DC: IEEE Computer Society.

Visser, W., Havelund, K., Brat, G., Park, S., & Lerda, F. (2003). Model checking programs. *Automated Software Engineering*, *10*(2), 203–232.

Vo, A., Vakkalanka, S., DeLisi, M., Gopalakrishnan, G., Kirby, R. M., & Thakur, R. (2008). Formal verification of practical MPI programs. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, (pp. 261–270). New York: ACM.

von Praun, C., & Gross, T. R. (2001). Object race detection. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, (pp. 70–82). ACM Press.

Wang, L., & Stoller, S. D. (2005). Static analysis of Atomicity for programs with non-blocking synchronization. In *Proceedings of the ACM SIGPLAN 2005 Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM Press.

Wang, L., & Stoller, S. D. (2006a). Accurate and efficient runtime detection of Atomicity errors in concurrent programs. In *Proceedings of the ACM SIGPLAN 2006 Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM Press.

Wang, L., & Stoller, S. D. (2006b). Runtime analysis of Atomicity for multi-threaded programs. *IEEE Transactions on Software Engineering*, *32*(2), 93–110.

Watson, G., Rasmussen, C., & Tibbitts, B. (2006). Application development using eclipse and the parallel tools platform. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, (p. 204). New York: ACM.

Worringen, J., Scholtyssik, K., Dr, P. & Bemmerl, T. (2002). *MP-MPICH: User documentation technical notes*.

Xu, M., Bodik, R., & Hill, M. D. (2005). A serializability volation detector for shared-memory server programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM Press.

Yu, Y., Rodeheffer, T., & Chen, W. (2005). RaceTrack: Efficient detection of data race conditions via adaptive tracking. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*. ACM Press.

ADDITIONAL READING

Chen., et al. 2008 Chen, F., Serbanuta, T. F., and Rosu, G. (2008). jPredictor: A Predictive Runtime Analysis Tool for Java. In Proceedings of the 30th international conference on Software engineering (ICSE '08), pages 221–230. ACM.

Chen., et al. 2009 Chen, Q., Wang, L., Yang, Z., and Stoller, S. D. (2009). HAVE: Integrated Dynamic and Static Analysis for Atomicity Violations. In Proceedings of International Conference on Fundamental Approaches to Software Engineering (FASE), volume 5503 of LNCS, pages 425–439. Springer.

Choi., et al. 2002 Choi, J.-D., Lee, K., Loginov, A., O'Callahan, R., Sarkar, V. and Sridharan, M. (2002). Efficient and Precise Datarace Detection for Multi-threaded object-oriented programs. In Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pages 258–269. ACM Press.

Engler and Ashcraft 2003 Engler, D. and Ashcraft, K. (2003). RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In Proceedings of ACM SIGOPS Symposium on Operating Systems Principles (SOSP). ACM Press.

Ernst, (2001). Ernst, M. D., Cockrell, J., Griswold, W. G., and Notkin, D. (2001). Dynamically Discovering Likely Program Invariants to Support Program Evolution. *IEEE Transactions on Software Engineering*, *27*(2), 99–123.

Fei and Midkiff 2006 Fei, L. and Midkiff, S. P. (2006). Artemis: Practical Run-time Monitoring of Applications for Execution Anomalies. In Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pages 84–95. ACM Press.

Flanagan, (2005). Flanagan, C., Freund, S. N., and Qadeer, S. (2005). Exploiting Purity for Atomicity. *IEEE Transactions on Software Engineering*, *31*(4).

Flanagan., et al. 2008 Flanagan, C., Freund, S. N., and Yi, J. (2008). Velodrome: A Sound and Complete Dynamic Atomicity Checker for Multithreaded Programs. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pages 293–303, New York, NY, USA. ACM.

Flanagan and Freund 2000 Flanagan, C. and Freund, S. (2000). Type-based Race Detection for Java. In Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pages 219–232. ACM Press.

Flanagan and Freund 2004a Flanagan, C. and Freund, S. N. (2004a). Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs. In Proc. ACM Symposium on Principles of Programming Languages (POPL), pages 256–267. ACM Press.

Flanagan and Freund 2004b Flanagan, C. and Freund, S. N. (2004b). Type Inference Against Races. In Static Analysis Symposium (SAS), volume 3148 of LNCS. Springer-Verlag.

Flanagan and Freund 2009 Flanagan, C. and Freund, S. N. (2009). FastTrack: Ffficient and Precise Dynamic Race Detection. In Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pages 121–133. ACM Press.

Flanagan and Qadeer 2003 Flanagan, C. and Qadeer, S. (2003). A Type and Effect System for Atomicity. In Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). ACM Press.

Godefroid 1997 Godefroid, P. (1997). Model Checking for Programming Languages Using Verisoft. In Proc. ACM Symposium on Principles of Programming Languages (POPL). ACM Press.

Graham., et al. 2005 Graham, R. L., Woodall, T. S., and Squyres, J. M. (2005). Open MPI: A flexible high performance MPI. In Proceedings, 6th Annual International Conference on Parallel Processing and Applied Mathematics, Poznan, Poland.

O'Callahan and Choi 2003 O'Callahan, R. and Choi, J.-D. (2003). Hybrid Dynamic Data Race Detection. In Proc. ACM SIGPLAN 2003 Symposium on Principles and Practice of Parallel Programming (PPoPP), pages 167–178. ACM.

Park and Sen 2008 Park, C.-S. and Sen, K. (2008). Randomized Active Atomicity Violation Detection in Concurrent Programs. In Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'08), pages 135–145, New York, NY, USA. ACM.

Wang and Stoller 2005 Wang, L. and Stoller, S. D. (2005). Static Analysis of Atomicity for Programs with Non-blocking Synchronization. In Proc. ACM SIGPLAN 2005 Symposium on Principles and Practice of Parallel Programming (PPoPP). ACM Press.

Wang and Stoller 2006a Wang, L. and Stoller, S. D. (2006a). Accurate and Efficient Runtime Detection of Atomicity Errors in Concurrent Programs. In Proc. ACM SIGPLAN 2006 Symposium on Principles and Practice of Parallel Programming (PPoPP). ACM Press.

Wang and Stoller 2006b Wang, L. and Stoller, S. D. (2006b). Runtime Analysis of Atomicity for Multi-threaded Programs. IEEE Transactions on Software Engineering, 32(2):93–110.

Xu., et al. 2005 Xu, M., Bodik, R., and Hill, M. D. (2005). A Serializability Volation Detector for Shared-memory Server Programs. In Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). ACM Press.

Yu., et al. 2005 Yu, Y., Rodeheffer, T., and Chen, W. (2005). RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking. In Symposium on Operating Systems Principles (SOSP). ACM Press.

KEY TERMS AND DEFINITIONS

Atomicity Violation: An atomicity violation refers to a program error that an interleaved execution of a set of code blocks (expected to be atomic) by multiple threads is not equivalent to any serial execution of the same code blocks.

Benign Warning: A benign warning is a false warning about some code that actually does not affect the correctness of the program but matches the definition of a specific bug. Examples include the benign data race on the busy-wait and compare-and-swap flag.

Concurrent Programs: Concurrent programs are programs that contain portion of code that can run concurrently on a machine or a collection of machines.

Data Race: A data race refers to a scenario that two concurrent threads perform conflicting accesses (*i.e.*, accesses to the same shared variable and at least one access is a write) and the threads use no explicit mechanism to prevent the accesses from being simultaneous.

Deadlock: A deadlock occurs when a chain of processes/threads are involved in a cycle in which each process is waiting for resources/locks that are held by some other processes.

Dynamic Analysis: Dynamic analysis is a program analysis technique that observes and analyzes the actual behaviors of a program by executing it.

False Positive: A false positive is a false bug warning that has been erroneously reported by the bug detection tool due to the imperfect or inaccurate algorithm or approach that is used by the tool.

Hybrid Analysis: Hybrid analysis refers to a hybrid program analysis technique that combines both dynamic and static analysis to analyze the program.

Static Analysis: Static analysis is a program analysis methodology that examines the program source code without running the program.