# Parallel Programming

Chapter 4

## Primality Testing

A simple computation that will take a long time.

**Whether a number x is prime:** Decide whether a number x is prime using the trial division algorithm.

**Trial Division Algorithm:** The function tries to divide x by 2 and by every odd number p from 3 up to the square root of x.

If any remainder is 0, then p is a factor of x and x is not prime; otherwise x is prime.

**Primality Testing Function**

|  |
| --- |
| public static boolean isPrime(long x) {  if (x % 2 == 0) {  return false;  }  long p = 3;  long xsqrt = (long) Math.ceil(Math.sqrt(x));  while (p <= xsqrt) {  if (x % p == 0) {  return false;  }  p += 2;  }  return true;  } |

## Sequential Program

First, call this method sequentially for many numbers and keep the running times.

**Measuring Time in Java**

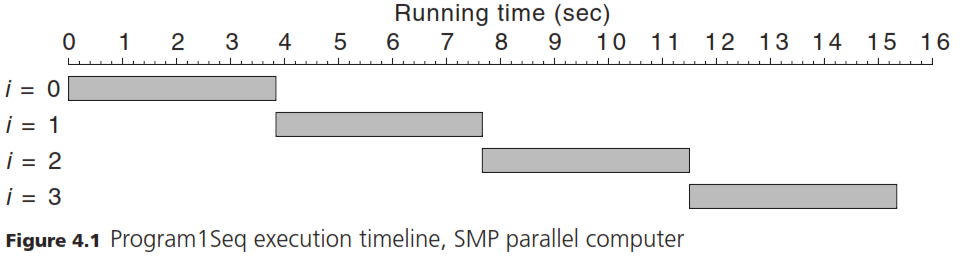
To measure times, we use Java’s System.currentTimeMillis() method, which returns the wall clock time in milliseconds (msec) since 1970.

**No printing while measuring time**

We record each instant in a variable, and postpone printing the results, so as to disturb the timing as little as possible while the program is running. It can take several msec to call println(), and we don’t want to include that time in our measurements.

|  |
| --- |
| /\*  \* @author lenovo  \*/  public class PrimeTestSeq {  /\*\*  \* only test odd divisors.  \* even numbers can not be primes  \* start with 3 and test with every odd number: 3, 5, 7, 9,  \* go up to the square root of the tested number  \* the numbers larger than the square root can not divide be sole divisors  \*/  private static boolean isPrime(long x) {  if (x % 2 == 0) {  return false;  }  long p = 3;  long xsqrt = (long) Math.ceil(Math.sqrt(x));  while (p <= xsqrt) {  if (x % p == 0) {  return false;  }  p += 2;  }  return true;  }  public static void main(String[] args) {  // testOneNumber();  testManyNumbers();  }  public static void testOneNumber() {  long m = 1289237867378231L;  System.out.println("started testing: " + m);  long t1 = System.currentTimeMillis();  boolean positive = isPrime(m);  long t2 = System.currentTimeMillis();  long duration = t2 - t1;  if (positive) {  System.out.println(m + " is prime");  } else {  System.out.println(m + " is not prime");  }  System.out.println("the time it takes: " + duration + " ms");  }    public static void testManyNumbers() {  long m[] = {1000000000000037L, 1000000000000091L, 1000000000000159L, 1000000000000187L};  System.out.println("started testing: ");  boolean positives[] = new boolean[m.length];  long startTimes[] = new long[m.length];  long endTimes[] = new long[m.length];    long start = System.currentTimeMillis();    for (int i = 0; i < m.length; i++) {  startTimes[i] = System.currentTimeMillis();  positives[i] = isPrime(m[i]);  endTimes[i] = System.currentTimeMillis();  }  for (int i = 0; i < m.length; i++) {  if (positives[i]) {  System.out.println(m[i] + " is prime");  } else {  System.out.println(m[i] + " is not prime");  }  System.out.println(i + " start: "+ (startTimes[i]-start) );  System.out.println(i + " start: "+ (endTimes[i]-start) );  }  }  } |

**Iterative Program Running Times**



## Parallel Programming with Java Threads

We can write a multi threaded java program.

**Number of Threads**

We construct the same number of threads as the number of cores. Each thread runs on one core.

**Constructing and Starting the Threads**

* **Construction:** First we create the threads. This part is sequential.
* **Start Parallel Execution**: When the start method of the threads is called, the parallel execution starts.

**Extending Thread Class**

User defined Thread class extends the Thread class in java.lang package.

**Run method**

Run method of the Thread class runs in parallel.

**End of Thread Execution**

When the run method of a thread finishes, that thread ends its execution.

**Main thread**

When we start 4 threads, there are actually 5 threads. One is the main thread that starts the other ones.

Main thread ends when the main method finishes the execution.

**Index of Threads**

Usually we assign a unique id to each thread. By using this id, we assign different jobs.

**Thread to Core Assignments**

Java Virtual Machine (JVM) and the operating system performs the thread to core assignments.

Programmer does not know which thread is running on what core.

**static variables**

The variables that need to be accessed by all threads are defined as static outside of **main** method.

**Array Variables**

If we need one variable for each thread, we can define a shared array. Each thread uses one element of that array as its own variable.

Each thread can use the element residing on its index on the array.

**Example:**

For each thread, we need a startTime, endTime and number variable. We can define these variables as shared array variables.

The size of the array will be equal to the number of threads.

|  |
| --- |
| /\*\*  \* Primality test with threads and without using Parallel Java library  \*  \* @author lenovo  \*/  public class PrimeTestThread extends Thread{    int index;    public PrimeTestThread(int index){  this.index = index;  }    public void run() {  startTimes[index] = System.currentTimeMillis();  PrimeTestParallel02.isPrime(numbers[index]);  endTimes[index] = System.currentTimeMillis();  }    /\*\*  \* only test odd divisors.  \* even numbers can not be primes  \* start with 3 and test with every odd number: 3, 5, 7, 9,  \* go up to the square root of the tested number  \* the numbers larger than the square root can not divide be sole divisors  \*/  public static boolean isPrime(long x) {  if (x % 2 == 0) {  return false;  }  long p = 3;  long xsqrt = (long) Math.ceil(Math.sqrt(x));  while (p <= xsqrt) {  if (x % p == 0) {  return false;  }  p += 2;  }  return true;  }  static long start;  static long startTimes[];  static long endTimes[];  static long numbers[] = {1000000000000037L, 1000000000000091L};  // static long numbers[] = {1000000000000037L, 1000000000000091L, 1000000000000159L, 1000000000000187L};  public static void main(String[] args) throws Exception {  start = System.currentTimeMillis();  startTimes = new long[numbers.length];  endTimes = new long[numbers.length];  PrimeTestThread threads[] = new PrimeTestThread[numbers.length];  for (int i = 0; i < threads.length; i++) {  threads[i] = new PrimeTestThread(i);  }    for (int i = 0; i < threads.length; i++) {  threads[i].start();  }    // wait until all threads are done  boolean notDone = true;  while(notDone){    notDone = false;  for (int i = 0; i < threads.length; i++) {  if(threads[i].isAlive())  notDone = true;  }  try{  Thread.sleep(100);  }catch(Exception e){    }  }    for (int i = 0; i < numbers.length; i++) {  System.out.println(i + " start: " + (startTimes[i] - start));  System.out.println(i + " end: " + (endTimes[i] - start));  }  }  } |

**Waiting the main thread**

We can instruct the main thread to wait others.

When each thread is done, they increase a counter.

When the final thread is done, it notifies the main thread to wake up.

Main thread wakes up and proceeds.

|  |
| --- |
| /\*\*  \* Primality test with threads and without using Parallel Java library  \*  \* @author lenovo  \*/  public class PrimeTestThread2 extends Thread{    int index;    public PrimeTestThread2(int index){  this.index = index;  }    public void run() {  startTimes[index] = System.currentTimeMillis();  PrimeTestParallel02.isPrime(numbers[index]);  endTimes[index] = System.currentTimeMillis();    oneThreadDone();  System.out.println("thread "+index+" is done.");  }    /\*\*  \* only test odd divisors.  \* even numbers can not be primes  \* start with 3 and test with every odd number: 3, 5, 7, 9,  \* go up to the square root of the tested number  \* the numbers larger than the square root can not divide be sole divisors  \*/  public static boolean isPrime(long x) {  if (x % 2 == 0) {  return false;  }  long p = 3;  long xsqrt = (long) Math.ceil(Math.sqrt(x));  while (p <= xsqrt) {  if (x % p == 0) {  return false;  }  p += 2;  }  return true;  }  static long start;  static long startTimes[];  static long endTimes[];  static long numbers[] = {1000000000000037L, 1000000000000091L};  // static long numbers[] = {1000000000000037L, 1000000000000091L, 1000000000000159L, 1000000000000187L};  public static PrimeTestThread2 threads[];    public static void main(String[] args) throws Exception {  start = System.currentTimeMillis();  startTimes = new long[numbers.length];  endTimes = new long[numbers.length];  threads = new PrimeTestThread2[numbers.length];  for (int i = 0; i < threads.length; i++) {  threads[i] = new PrimeTestThread2(i);  }    for (int i = 0; i < threads.length; i++) {  threads[i].start();  }    // wait until all threads are done  waitMain();    for (int i = 0; i < numbers.length; i++) {  System.out.println(i + " start: " + (startTimes[i] - start));  System.out.println(i + " end: " + (endTimes[i] - start));  }  }    public static boolean allDone = false;  public static Object obj = new Object();    public static void waitMain() {  if(allDone)  return;  synchronized (obj) {  try {  System.out.println("main thread waiting");  obj.wait();  System.out.println("main thread woke up");  } catch (InterruptedException e) {  e.printStackTrace();  }  }    }  public static int doneCounter = 0;  public static synchronized void oneThreadDone(){    doneCounter++;  if(doneCounter < threads.length)  return;    allDone = true;  doneCounter=0;  System.out.println("waiking up main thread");  synchronized (obj) {  obj.notify();  }  }  } |

## Parallel Program with Parallel Java Library

**Parallel Java library**

Parallel java library jar file needs to be downloaded and included into the project.

The file to be downloaded: **pj20120620.jar**

Address: <http://www.cs.rit.edu/~ark/pj.shtml#download>

Add the jar file to: Project | Properties |Libraries

**ParallelTeam object**

A ParallelTeam object is constructed. This object runs the threads in parallel. We provide the number of threads to be run to this object.

**ParallelRegion: run method**

ParallelRegion object has the code that will run in parallel in the run method.

**Anonymous class**

ParallelRegion object constructed from an anonymous class.

**getThreadIndex() method**

the index of the calling thread is retreived by ParallelRegion’s getThreadIndex()

|  |
| --- |
| /\*  \* @author lenovo  \*/  import edu.rit.pj.ParallelRegion;  import edu.rit.pj.ParallelTeam;  public class PrimeTestParallel {  /\*\*  \* only test odd divisors.  \* even numbers can not be primes  \* start with 3 and test with every odd number: 3, 5, 7, 9,  \* go up to the square root of the tested number  \* the numbers larger than the square root can not divide be sole divisors  \*/  private static boolean isPrime(long x) {  if (x % 2 == 0) {  return false;  }  long p = 3;  long xsqrt = (long) Math.ceil(Math.sqrt(x));  while (p <= xsqrt) {  if (x % p == 0) {  return false;  }  p += 2;  }  return true;  }    static long start;  static long startTimes[];  static long endTimes[];  static long numbers[] = {1000000000000037L, 1000000000000091L};  // static long numbers[] = {1000000000000037L, 1000000000000091L, 1000000000000159L, 1000000000000187L};    public static void main(String[] args) throws Exception {    start = System.currentTimeMillis();  startTimes = new long[numbers.length];  endTimes = new long[numbers.length];    ParallelTeam team = new ParallelTeam(numbers.length);  team.execute(new ParallelRegion() {  public void run() {  int i = getThreadIndex();  startTimes[i] = System.currentTimeMillis();  isPrime(numbers[i]);  endTimes[i] = System.currentTimeMillis();  }  });    for (int i = 0; i < numbers.length; i++) {  System.out.println(i + " start: "+ (startTimes[i]-start) );  System.out.println(i + " end: "+ (endTimes[i]-start) );  }  }  } |

**How Parallel Program Works**

**Main program thread:** The main program begins with one thread, the “main thread,” executing the main() method.

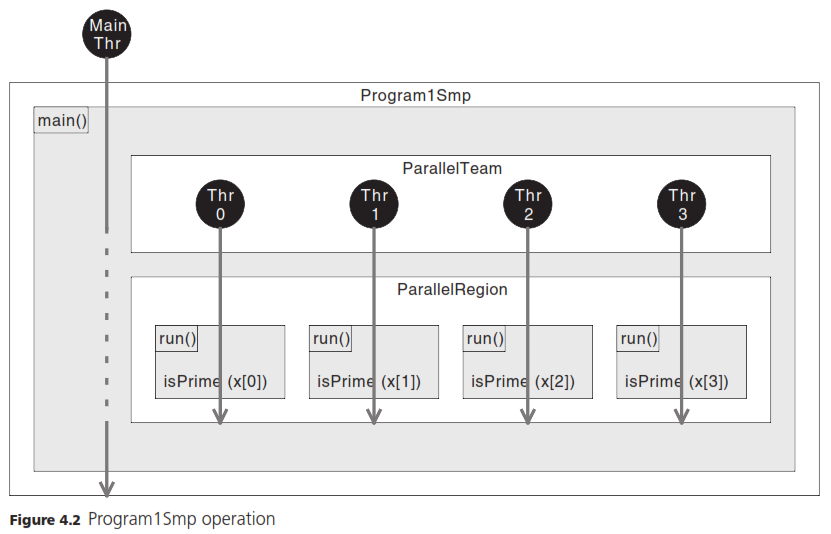
**Parallel Team Threads:** When the main thread creates the parallel team object, the parallel team object creates additional hidden threads; the constructor argument specifies the number of threads.

**Team work:** These form a “team” of threads for executing code in parallel. When the main thread calls the parallel region’s execute() method, the main thread suspends execution and the parallel team threads take over.

**All team threads call ParallelRegion’s run method simultaneously:** Each thread retrieves a value for x, and each thread calls isPrime().

**Concurrent execution:** Thus, the isPrime() subroutine calls happen at the same time, and each subroutine call is performed by a different thread with a different argument.

**Back to main thread:** When all the subroutine calls have finished executing, the main thread resumes executing statements after the parallel region and prints the timing measurements.



**No Loop, Multiple Threads**

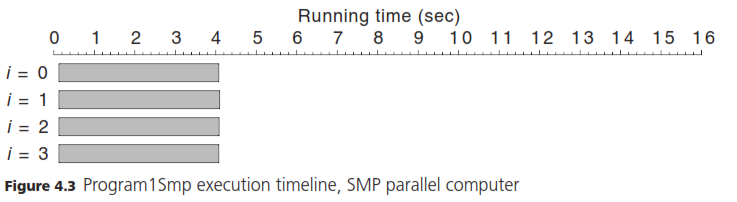
The parallel program illustrates a central theme of parallel program design: Repetition does not necessarily imply sequencing (looping).

The sequential program used a loop to get n repetitions (4 repetitions) of a subroutine call (isPrime).

However, for this program there is no need to do the repetitions in sequence in a loop. Each method call is assigned to a thread.

So a loop is not the only way to do a repeated calculation.

**Running Timelines**



Each thread runs in parallel. There needs to be at least 4 cores.

The speedup from the book:

Sequential running time/parallel running time = 15342/4098 = **3.744**

Almost linear increase.

**Startup Cost for parallel programs**

With Program1Smp, the first subroutine call didn’t begin until 125 msec after the program started.

During this time, the program was occupied in creating the parallel team and parallel region objects, starting up the parallel team threads, and executing the parallel region’s run() method—work that the sequential program didn’t have to do.

**Parallelism is not free**

This illustrates another central theme of parallel program design: Parallelism is not free. The benefit of speedup or sizeup comes with a price of extra overhead that is not needed in a sequential program.

The name of the game is to minimize this extra overhead.

## Running Parallel Programs on a Single CPU Machine

A parallel program can easily run on a single CPU machine.

In a single CPU machine, parallel version may outperform the sequential version.

This is surprising since parallel program also have threading overhead.

**The reason for this is that:**

JVM Just-In-Time compiler converts the isPrime function into machine code faster in parallel version. Since all threads are calling the same function.

In sequential version, it takes longer time for JVM to understand that isPrime function is a hotspot and its code needs to be converted into the machine code.