CONCURRENCY IN JAVA

Course “Parallel Computing”

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC)
Wolfgang.Schreiner@risc.jku.at
http://www.risc.jku.at
Java on a NUMA Architecture

■ Loading Java 11 (default is Java 6):
  zusie> module avail
  ...
  zusie> module load jdk/11.0.1+13
  Module for jdk 11.0.1+13 loaded.
  zusie> java
  Picked up _JAVA_OPTIONS: -XX:+UseParallelGC -XX:ParallelGCThreads=4
  ...

■ Advanced Runtime Options:
  -XX:+UseParallelGC
    Enables the use of the parallel scavenger garbage collector
    (also known as the throughput collector) to improve the performance
    of your application by leveraging multiple processors. ...
  -XX:ParallelGCThreads=N
    Sets the number of threads used for parallel garbage collection in
    the young and old generations. ...
  -XX:+UseNUMA
    Enables performance optimization of an application on a machine
    with nonuniform memory architecture (NUMA) by increasing the
    application’s use of lower latency memory. ...

Additional threads are created for garbage collection.
Java on a NUMA Architecture

■ Pinning threads to cores:

zusie> man 1 dplace
...

Dplace is used to bind a related set of processes to specific cpus or nodes to prevent process migrations. In some cases, this will improve performance since a higher percentage of memory accesses will be to the local node.

...

OPTIONS

-c Cpu numbers. Specified as a list of cpus, optionally strided cpu ranges, or a striding pattern. Example: "-c 1", "-c 2-4", "-c 1,4-8,3", "-c 2-8:3", ...

...

In some cases, version 2 of numatools will give better performance than version 1. ... In version 2, this memory is usually allocated local to the task’s node.

...

■ Pin Java threads to physical cores in current CPU set:

zusie> dplace -c 16-31 java ... // all threads on second blade
Java on a NUMA Architecture

■ Control NUMA policy for processes or shared memory:

```
zusie> man 1 numactl
...
numactl runs processes with a specific NUMA scheduling or memory placement policy. ...
...
OPTIONS
-physcpubind=cpus, -C cpus
  Only execute process on cpus. ... Physical cpus may be specified as N,N,N or N-N or N,N-N or N,N,N-N and so forth. Relative cpus may be specified as +N,N,N or +N-N or +N,N-N and so forth. The + indicates that the cpu numbers are relative to the process’ set of allowed cpus in its current cpuset. ...
...
```

■ Place Java threads on physical cores in current CPU set:

```
zusie> numactl -C +16-31 java ... // all threads on second blade
```

■ No pinning: threads may migrate among cores.
Java on a NUMA Architecture

top -H -u login: press f j <ENTER>

top - 08:17:23 up 8 days, 17:01, 12 users, load average: 2.34, 0.53, 0.18
Tasks: 16842 total, 1 running, 16840 sleeping, 1 stopped, 0 zombie
Cpu(s): 0.8%us, 0.0%sy, 0.0%ni, 99.2%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Mem: 2051061M total, 1958678M used, 92382M free, 0M buffers
Swap: 262143M total, 0M used, 262143M free, 1952269M cached

<table>
<thead>
<tr>
<th>PID</th>
<th>USER</th>
<th>PR</th>
<th>NI</th>
<th>VIRT</th>
<th>RES</th>
<th>SHR</th>
<th>S</th>
<th>%CPU</th>
<th>%MEM</th>
<th>TIME+</th>
<th>P COMMAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>331708</td>
<td>k313270</td>
<td>20</td>
<td>0</td>
<td>62444</td>
<td>15m</td>
<td>1972</td>
<td>R</td>
<td>20.0</td>
<td>0.0%</td>
<td>0:08.14</td>
<td>529 top</td>
</tr>
<tr>
<td>331467</td>
<td>k313270</td>
<td>20</td>
<td>0</td>
<td>106m</td>
<td>2536</td>
<td>1476</td>
<td>S</td>
<td>0.0</td>
<td>0:00.02</td>
<td>513 sshd</td>
<td></td>
</tr>
<tr>
<td>331468</td>
<td>k313270</td>
<td>20</td>
<td>0</td>
<td>55824</td>
<td>5704</td>
<td>2776</td>
<td>S</td>
<td>0.0</td>
<td>0:00.10</td>
<td>10 bash</td>
<td></td>
</tr>
<tr>
<td>331633</td>
<td>k313270</td>
<td>20</td>
<td>0</td>
<td>106m</td>
<td>2536</td>
<td>1476</td>
<td>S</td>
<td>0.0</td>
<td>0:00.02</td>
<td>513 sshd</td>
<td></td>
</tr>
<tr>
<td>331634</td>
<td>k313270</td>
<td>20</td>
<td>0</td>
<td>55824</td>
<td>5724</td>
<td>2796</td>
<td>S</td>
<td>0.0</td>
<td>0:00.10</td>
<td>580 bash</td>
<td></td>
</tr>
<tr>
<td>331709</td>
<td>k313270</td>
<td>20</td>
<td>0</td>
<td>32.8g</td>
<td>116m</td>
<td>12m</td>
<td>S</td>
<td>0.0</td>
<td>0:00.01</td>
<td>64 java</td>
<td></td>
</tr>
<tr>
<td>331710</td>
<td>k313270</td>
<td>20</td>
<td>0</td>
<td>32.8g</td>
<td>116m</td>
<td>12m</td>
<td>S</td>
<td>0.0</td>
<td>0:00.29</td>
<td>65 java</td>
<td></td>
</tr>
<tr>
<td>331711</td>
<td>k313270</td>
<td>20</td>
<td>0</td>
<td>32.8g</td>
<td>116m</td>
<td>12m</td>
<td>S</td>
<td>0.0</td>
<td>0:00.00</td>
<td>66 java</td>
<td></td>
</tr>
<tr>
<td>331712</td>
<td>k313270</td>
<td>20</td>
<td>0</td>
<td>32.8g</td>
<td>116m</td>
<td>12m</td>
<td>S</td>
<td>0.0</td>
<td>0:00.00</td>
<td>67 java</td>
<td></td>
</tr>
<tr>
<td>331713</td>
<td>k313270</td>
<td>20</td>
<td>0</td>
<td>32.8g</td>
<td>116m</td>
<td>12m</td>
<td>S</td>
<td>0.0</td>
<td>0:00.00</td>
<td>68 java</td>
<td></td>
</tr>
<tr>
<td>331714</td>
<td>k313270</td>
<td>20</td>
<td>0</td>
<td>32.8g</td>
<td>116m</td>
<td>12m</td>
<td>S</td>
<td>0.0</td>
<td>0:00.00</td>
<td>69 java</td>
<td></td>
</tr>
<tr>
<td>331715</td>
<td>k313270</td>
<td>20</td>
<td>0</td>
<td>32.8g</td>
<td>116m</td>
<td>12m</td>
<td>S</td>
<td>0.0</td>
<td>0:00.00</td>
<td>70 java</td>
<td></td>
</tr>
</tbody>
</table>

Column “P”: the core executing the thread.
public class HelloRunnable implements Runnable {
    public void run() {
        System.out.println("Hello!");
    }
}

public static void main(String args[]) {
    Thread t =
        new Thread(new HelloRunnable());
    t.start();
    try { t.join() }
    catch(InterruptedException e) { }
}

public class HelloThread extends Thread {
    public void run() {
        System.out.println("Hello!");
    }
}

public static void main(String args[]) {
    Thread t =
        new HelloThread();
    t.start();
    try { t.join() }
    catch(InterruptedException e) { }
}

Creating threads and waiting for their termination.
public class MatMultThreads {

    private static int N;
    private static int T;
    private static double[][] A;
    private static double[][] B;
    private static double[][] C;

    private static final class MultThread
        extends Thread {
        private int begin; private int end;
        public MultThread(int begin, int end) {
            this.begin = begin; this.end = end;
        }
        public void run() {
            for (int i = begin; i < end; i++) {
                for (int j = 0; j < N; j++) {
                    C[i][j] = 0;
                    for (int k = 0; k < N; k++)
                        C[i][j] += A[i][k]*B[k][j];
                }
            }
        }
    }

    private static void multiply() {
        int n = N/T;
        Thread[] thread = new MultThread[T];
        for (int i = 0; i < T; i++) {
            thread[i] = new MultThread(i*n, Math.min((i+1)*n,N));
            thread[i].start();
        }
        try {
            for (int i = 0; i < T; i++)
                thread[i].join();
        } catch(InterruptedException e) { }
    }

    public static void main(String[] args) {
        try {
            N = Integer.parseInt(args[0]);
            T = Integer.parseInt(args[1]);
        } catch(NumberFormatException e) { return; }
        A = new double[N][N];
        B = new double[N][N];
        C = new double[N][N];
        multiply();
    }
}
Synchronization of Threads

■ Synchronized methods:

```java
public class SynchronizedCounter {
    private int c = 0;
    public synchronized void increment() { c++; }
    public synchronized int value() { return c; }
}
```

■ Synchronized statements:

```java
public static void push(List<String> list, String name) {
    synchronized(list) { list.add(name); }
}
public static void pop(List<String> list) {
    synchronized(list) { list.remove(list.size()-1); }
}
```

The executions of two synchronized methods/statements on the same lock object do not overlap.
public class MatMultWorkers {

    private static int N;
    private static int T;
    private static double[][] A;
    private static double[][] B;
    private static double[][] C;
    private static int rows;

    private static final class MultWorker extends Thread {
        public void run() {
            while (true) {
                int i;
                synchronized (C) {
                    i = rows;
                    rows++;
                }
                if (i >= N) return;
                for (int j = 0; j < N; j++) {
                    C[i][j] = 0;
                    for (int k = 0; k < N; k++)
                        C[i][j] += A[i][k]*B[k][j];
                }
            }
        }
    }

    private static void multiply() {
        int n = N/T;
        Thread[] thread = new MultWorker[T];
        for (int i = 0; i < T; i++) {
            thread[i] = new MultWorker();
            thread[i].start();
        }
        try {
            for (int i = 0; i < T; i++)
                thread[i].join();
        } catch(InterruptedException e) { }
    }

    public static void main(String[] args) {
        try {
            N = Integer.parseInt(args[0]);
            T = Integer.parseInt(args[1]);
        } catch(NumberFormatException e) { return; }
        A = new double[N][N];
        B = new double[N][N];
        C = new double[N][N];
        rows = 0;
        multiply();
    }
}
Memory Consistency Properties

Be careful: the effect of a write action by one thread is only guaranteed to be seen by the read action of another thread, if the actions are in the (transitive) happens-before relationship:

- Each action in a thread happens-before every later action (in program order) in the same thread.
- A synchronized method or statement exit happens-before every subsequent synchronized entry on the same lock object.
- A write to a volatile field happens-before every read to the same field.
- The start of a thread happens-before all actions of the thread.
- All actions of a thread happen-before every join of the thread.

The constructs synchronized, volatile, start and join define the happens-before relationship of a program.
The High-Level Concurrency API

Package `java.util.concurrent`.

- **Lock objects**
  - Package `java.util.concurrent.locks`
- **Executors**
  - Executor interfaces, thread pools, the Fork/Join framework.
- **Concurrent collections**
  - Interfaces `BlockingQueue`, `ConcurrentMap`, `ConcurrentNavigableMap`.
- **Atomic variables**
  - Package `java.util.concurrent.atomic`
- **Pseudorandom numbers from multiple threads.**
  - Class `ThreadLocalRandom`

We will investigate the “executors” in more detail.
Executors

- Core idea: separate “tasks” from “threads”.
  - Tasks: computations to be performed.
  - Threads: the unit of execution mapped to a processor core.

- Executors: an object that executes tasks.
  - Receives tasks and schedules them on a pool of threads.

- Tasks may or may not return a result:
  - interface Executor:
    - void execute(Runnable command)
    - interface Runnable { void run(); }
  - interface ExecutorService:
    - <T> Future<T> submit(Callable<T> task)
    - Future<?> submit(Runnable task)
    - interface Callable<T> { T call(); ... }
    - interface Future<T> { T get(); ... }
Thread Pools

■ Factory methods of class Executors:
  static ExecutorService newFixedThreadPool(int nThreads)
  Creates a thread pool that reuses a fixed number of threads operating
  off a shared unbounded queue.
  static ExecutorService newSingleThreadExecutor()
  Creates an Executor that uses a single worker thread
  operating off an unbounded queue.
  static ExecutorService newWorkStealingPool(int parallelism)
  Creates a thread pool that maintains enough threads to support given
  parallelism level, and may use multiple queues to reduce contention.

■ Manual creation of a ThreadPoolExecutor:
  ThreadPoolExecutor(int corePoolSize, int maximumPoolSize,
  long keepAliveTime, TimeUnit unit,
  BlockingQueue<Runnable> workQueue)
  Creates a new ThreadPoolExecutor with the given initial parameters
  and default thread factory and rejected execution handler.

Creation may be also parameterized by a “thread factory”.

import java.util.*;
import java.util.concurrent.*;

public class MatMultPool {

    private static int N;
    private static int T;
    private static double[][] A;
    private static double[][] B;
    private static double[][] C;

    private static final class MultTask
            implements Runnable {
        private int i;
        public MultTask(int i) {
            this.i = i;
        }
        public void run() {
            for (int j = 0; j < N; j++) {
                C[i][j] = 0;
                for (int k = 0; k < N; k++)
                    C[i][j] += A[i][k]*B[k][j];
            }
        }
    }

    private static void multiply() {
        ExecutorService pool =
            Executors.newFixedThreadPool(T);
        Vector<Future<?>> result =
            new Vector<Future<?>> (N);
        for (int i = 0; i < N; i++)
            result.add(pool.submit(new MultTask(i)));
        try {
            for (int i = 0; i < N; i++)
                result.get(i).get();
        } catch (InterruptedException e) { }
        catch (ExecutionException e) { }
    }

    public static void main(String[] args) {
        ...
        try
        {
            N = Integer.parseInt(args[0]);
            T = Integer.parseInt(args[1]);
        }
        catch(NumberFormatException e) { return; }
        A = new double[N][N];
        B = new double[N][N];
        C = new double[N][N];
        multiply();
    }
}
import java.util.*;
import java.util.concurrent.*;

public class MatMultFuture {

    private static int N;
    private static int T;
    private static double[][] A;
    private static double[][] B;
    private static double[][] C;

    private static final class MultResult
        implements Callable<double[]> {
            private int i;
            public MultResult(int i) {
                this.i = i;
            }
            public double[] call() throws Exception {
                double[] C = new double[N];
                for (int j = 0; j < N; j++)
                    C[j] = 0;
                for (int k = 0; k < N; k++)
                    C[j] += A[i][k]*B[k][j];
                return C;
            }
        }
    
    private static void multiply() {
        ExecutorService pool = Executors.newFixedThreadPool(T);
        Vector<Future<double[]>> result =
            new Vector<Future<double[]>> (N);
        for (int i = 0; i < N; i++)
            result.add(pool.submit(new MultResult(i)));
        try {
            for (int i = 0; i < N; i++)
                C[i] = result.get(i).get();
        } catch(InterruptedException e) { }
        catch(ExecutionException e) { }
    }

    public static void main(String[] args) {
        ... try
        {
            N = Integer.parseInt(args[0]);
            T = Integer.parseInt(args[1]);
        } catch(NumberFormatException e) { return; }
        A = new double[N][N];
        B = new double[N][N];
        C = new double[N][];
        multiply();
    }
}
The Fork/Join Framework

A framework for recursive tasks.

- Class ForkJoinPool
  
  ForkJoinPool(int parallelism)
  
  <T> ForkJoinTask<T> submit(ForkJoinTask<T> task)

- Abstract class ForkJoinTask<T>:
  
  ForkJoinTask<T> fork()
  
  public final T join()
  
  static void invokeAll(ForkJoinTask<?>... tasks)

  - Abstract subclass RecursiveAction:
    
    protected abstract void compute()

  - Abstract subclass RecursiveTask<T>:
    
    protected abstract T compute()

Applies work stealing: when one thread runs out of tasks, it steals tasks created by another thread.
private static void multiply() {
    ForkJoinPool pool = new ForkJoinPool(T);
    ForkJoinTask<Void> task =
        pool.submit(new MultRec(0, N));
    task.join();
}

class MultRec extends RecursiveAction {
    private int begin; private int end;
    public MultRec(int begin, int end) {
        this.begin = begin; this.end = end;
    }
    public void compute() {
        if (begin == end - 1) {
            int i = begin;
            for (int j = 0; j < N; j++) {
                C[i][j] = 0;
                for (int k = 0; k < N; k++)
                    C[i][j] += A[i][k] * B[k][j];
            }
        } else if (begin < end) {
            int mid = (begin+end)/2;
            forAll(new MultRec(begin, mid), new MultRec(mid, end));
        }
    }
}
Distributed Memory Programming

- Use networking API for “message passing” programming.
  - TCP-based sockets for transferring streams of bytes.
- On a remote node a server process has to be started.
  - For instance, by “secure shell”.
  - Process waits on some port for connection requests.
  - By accepting a request, server receives socket to client.
- Client processes may request connections to the server.
  - Server identified by IP address and port number.
  - Upon acceptance, client receives socket to server.
- Sockets provide conventional input/output streams.
  - Standard I/O operations may be used for communication.
  - Output has to be (explicitly/automatically) flushed.

Low-level approach; there also exist high level alternatives, e.g., Java Remote Method Invocation (RMI).
Example: A Client/Server Program

import java.io.*;
import java.net.*;

public class MatMultNet {

    private final static String URL = "localhost";
    private final static int port = 9999;
    private static int N;
    private static int T;
    private static double[][] A;
    private static double[][] B;
    private static double[][] C;

    private static final class MultThread
        extends Thread {
        private int begin; private int end;
        public MultThread(int begin, int end) {
            this.begin = begin; this.end = end;
        }
        public void run() {
            for (int i = begin; i < end; i++) {
                for (int j = 0; j < N; j++) {
                    C[i][j] = 0;
                    for (int k = 0; k < N; k++)
                        C[i][j] += A[i][k]*B[k][j];
                }
            }
        }
    }

    private static void multiply() {
        int n = N/T;
        Thread[] thread = new MultThread[T];
        for (int i = 0; i < T; i++) {
            thread[i] =
                new MultThread(i*n, Math.min((i+1)*n,N));
            thread[i].start();
        }
        try {
            for (int i = 0; i < T; i++)
                thread[i].join();
        }
        catch(InterruptedException e) { }
    }

    public static void main(String[] args) {
        ...  
        if (args[0].equals("-client"))
            client();
        else
            server();
    }
}
Example: A Client/Server Program

```java
public static void server() {
    try {
        ServerSocket server = new ServerSocket(port);
        while (true) {
            Socket socket = server.accept();
            BufferedReader in =
                new BufferedReader(new InputStreamReader(socket.getInputStream()));
            PrintWriter out =
                new PrintWriter(new OutputStreamWriter(socket.getOutputStream()), true);
            String line = in.readLine();
            if (line == null) return;
            ...
            try {
                N = Integer.parseInt(args[0]);
                T = Integer.parseInt(args[1]);
            } catch(NumberFormatException e) { ... }
            A = new double[N][N];
            B = new double[N][N];
            C = new double[N][N];
            long t1 = System.currentTimeMillis();
            multiply();
            long t2 = System.currentTimeMillis();
            out.println((t2-t1) + " ms");
        }
    } catch(IOException e) { System.exit(-1); }
}
```

```java
static void client() {
    try {
        BufferedReader console =
            new BufferedReader(new InputStreamReader(System.in));
        while (true) {
            String line = console.readLine();
            if (line == null) return;
            Socket socket = new Socket(URL, port);
            BufferedReader in =
                new BufferedReader(new InputStreamReader(socket.getInputStream()));
            PrintWriter out =
                new PrintWriter(new OutputStreamWriter(socket.getOutputStream()), true);
            out.println(line);
            String answer = in.readLine();
            if (answer == null) return;
            System.out.println(answer);
        }
    } catch(IOException e) { System.exit(-1); }
}
```