Second International Conference
Modelling and Development of Intelligent Systems
Sibiu - Romania, September 29 - October 02, 2011

Playing with threads in Java 7

Ernest Scheiber

Abstract

The new Java 7 version introduce the *fork-join* technique and the class *Phaser.* These techniques are compared with other tools introduced in earlier Java versions (`join, CountDownLatch, CyclicBarrier, ExecutorService`) in terms of the time to solve a simple embarrassing parallel test problem based on a synchronous algorithm for the successive approximation method.

In the same way a comparison is made between Java 6 and Java 7.

1 Introduction

The new Java 7 version introduce the *fork-join* technique that will allow recurrence definitions in parallel-concurrent programs as well as an implicit support for multi-core processors. There are some good tutorials for this Java technique [4],[3]. In [3] there is stated that, for the *fork-join* technique, the algorithmic speed increase linearly relative to the number of cores.

The class *Phaser* offers a flexible barrier synchronization technique based on phases associated to registered parties, i.e. threads.

We are interested to compare

- *fork-join* technique with `Phaser` and other tools introduced in earlier versions of Java (`join, CountDownLatch, CyclicBarrier, ExecutorService`) and
- the behavior of these tools between Java 6 and Java 7

in terms of the time to solve a simple embarrassing parallel test problem based on a synchronous algorithm for the successive approximation method.

2 A test problem

The Jacobi method to solve a linear system $Ax = b,\ A = (a_{i,j})_{1\leq i,j\leq n} \in M_n(\mathbb{R}),\ b = (b_i)_{1\leq i\leq n} \in \mathbb{R}^n$ is well known, [1]. For parallel computing, it is known a block version, too, [2]. We shall use the most simple version, with the iterations given by

$$u_i^{k+1} = \frac{1}{a_{i,i}}(b_i - \sum_{\substack{j=1\\ j\neq i}}^{n} a_{i,j}u_j^k), \quad i \in \{1,\ldots,n\},\ k \in \mathbb{N}.$$

There are known several convergence theorems. Our test data will satisfy the conditions of a convergence theorem. Starting with an arbitrary vector $u^0 \in \mathbb{R}^n$, the stopping rule will be

$$\|u^{k+1} - u^k\|_\infty < \varepsilon \qquad \text{or} \qquad k > k_{\max}, \tag{1}$$

where ε is a given tolerance to be satisfied in given fixed number of iterations $k_{\max}$.

Each u_i^{k+1} component will be computed within a thread. There is a synchronization problem: before starting a new iteration, all launched threads must finish their activities and the stopping test must be applied.

Our test data are defined in the class

```java
public class Data{
  public double a[][]={{4,1,1},{1,4,1},{1,1,4}};
  public double b[]={9,12,15};
  public int n=a.length;
  public double x[]=new double[n];
  public double eps=1.0e-5;           // tollerance
  public int nmi=50;                  // maximum permissible number of iterations
  public int error;                   // error code
  public double y[]=new double[n];    // the solution
}
```

3 Java parallel-concurrent programming templates

The threads act in a pool. The most simple case considers the threads as elements of an array, collection, etc. Java 5 introduces specialized pools that implements the interface `java.util.concurrent.ExecutorService.` Also there is introduced the class `java.util.concurrent.CyclicBarrier` to synchronize the threads that have arrived at a specific point of their activities. The *fork-join* technique uses a specialized pool, `java.util.concurrent.ForkJoinPool.`

We have developed 6 classes to solve the linear system as it was described in the previous section:

1. The threads will be elements of an array and the required synchronization is obtain using the `join` method of the class `Thread.`
   The computation required by (1) is perform by the thread

```java
class JacobiThread1 extends Thread{
    int myindex;

    public JacobiThread1(int myindex){
      this.myindex=myindex;
    }

    public void run(){
      double suma=0;
      for(int i=0;i<d.n;i++)
        if (i!=myindex)
          suma+=d.a[myindex][i]*d.x[i];
      d.y[myindex]=(d.b[myindex]-suma)/d.a[myindex][myindex];
    }
  }
```

   The instantiation of the threads, their starting and the synchronization is done by the method

```java
void solve(){
    int ni=0;
    double nrm;
    JacobiThread1 t[]=new JacobiThread1[d.n];
    do{
      ni++;                         // A new iteration
      for(int i=0;i<d.n;i++){       // For any equation
        t[i]=new JacobiThread1(i);     // a new thread is instantiated
        t[i].start();               // and is launched to work
      }
      try{                                      // Synchronization
        for(int i=0;i<d.n;i++) t[i].join();   // on the condition that the threads
      }                                         // are terminated
      catch(InterruptedException e){}
      nrm=0;
      for(int i=0;i<d.n;i++){                 // Stopping rule
        if(nrm<Math.abs(d.x[i]-d.y[i]))
            nrm=Math.abs(d.x[i]-d.y[i]);
        d.x[i]=d.y[i];
      }
    }
    while ((nrm>=d.eps)&&(ni<d.nmi));
    if (nrm<d.eps)
      d.error=0;
    else
      d.error=1;
  }
```

At each iteration there are created and started a new set of threads corresponding to the components of the new approximation to be computed.

2. Using the same thread code, the usage of the `java.util.concurrent.Phaser` class is based on a template presented in the *docs/api* of the Java 7 release. In this case the code of the *solve* method is

```java
void solve(){
  List<Runnable> tasks=new ArrayList<Runnable>(d.n);
  for(int i=0;i<d.n;i++)
    tasks.add(new JacobiThread1(i));
  runTasks(tasks);
}
```

where the code based on the template is

```java
static void runTasks(List<Runnable> tasks) {
  int ni=0;
  double nrm;
  final Phaser phaser = new Phaser(1); //"1" to register self
  do{
    ni++;
    for (final Runnable task : tasks) {
      phaser.register();
      new Thread() {
        public void run() {
           task.run();
           phaser.arriveAndAwaitAdvance();
        }
      }.start();
    }
   try{Thread.sleep(5);}catch(InterruptedException e){}
    nrm=0;
    for(int i=0;i<d.n;i++){
      if(nrm<Math.abs(d.x[i]-d.y[i])) nrm=Math.abs(d.x[i]-d.y[i]);
      d.x[i]=d.y[i];
    }
    if (nrm<d.eps)
       d.error=0;
    else
       d.error=1;
   }
   while((nrm>=d.eps)&&(ni<d.nmi));
   phaser.arriveAndDeregister();
 }
```

3. Very close to the previous template is as follows. The class `java.util.concurrent.CountDownLatch` introduced in Java 5 is used for synchronization. The code of the thread reproduced above has an additional line code (after the line 13)

```java
countDownLatch.countDown();
```

In the solver method, at the beginning of each iteration it is instantiated an instance of the class `CountDownLatch`

```java
 countDownLatch=new CountDownLatch(d.n);
```

and the lines relating to the `join` method are replaced by

```java
try{
  countDownLatch.await();
} catch (InterruptedException e){}
```

4. The synchronization will be done through a `CyclicBarrier` object. In this case the following thread is used to compute the attached component to each iterations:

```
class JacobiThread3 extends Thread{
   int myIndex;

  JacobiThread3(int index){
    myIndex=index;
  }

  public void run(){
    double s;
    while(!sfarsit){
       ni++;
       s=0;
       for(int i=0;i<d.n;i++){
          if(i!=myIndex){
            s+=d.a[myIndex][i]*d.x[i];
          }
       }
       d.y[myIndex]=(d.b[myIndex]-s)/d.a[myIndex][myIndex];
       try{
          barrier.await();
       }
       catch(Exception e){}
    }
  }
}
```

The solver method is

```
public void solve(){
    StoppingRule test=new StoppingRule();
    barrier=new CyclicBarrier(d.n, test);
    for(int i=0;i<d.n;i++)
      (new JacobiThread3(i)).start();
    while(!sfarsit);
  }
```

As the name, the *StoppingRule* class is a thread required by the `CyclicBarrier` class containing the stopping rule:

```
class StoppingRule extends Thread{
    public void run(){
      double nrm=0,dif;
      for(int i=0;i<d.n;i++){
        dif=Math.abs(d.y[i]-d.x[i]);
        if(dif>nrm) nrm=dif;
        d.x[i]=d.y[i];
      }
      if((nrm<d.eps)||(ni/d.n>d.nmi))
        sfarsit=true;
      if(nrm<d.eps)
        d.error=0;
      else
        d.error=1;
    }
  }
```

5. Instead of the array containing the threads, a specialized thread pool is used. The pool implements the interface `java.util.concurrent.ExecutorService.` In this case the code of the solver method is

```
  public void solve(){
    StoppingRule test=new StoppingRule();
    barrier=new CyclicBarrier(d.n, test);
    ExecutorService executor=Executors.newFixedThreadPool(d.n);
    for(int i=0;i<d.n;i++){
      Runnable action=new JacobiThread2(i);
      executor.execute(action);
    }
    try{
      executor.shutdown();
      while(!executor.isTerminated());
    }
    catch(Exception e){}
  }
```

6. Finally, we use the *fork-join* technique. The threads pool is an instance of the class `java.util.concurrent.ForkJoinPool.` Instead of pure thread objects there are using descendants of the class `java.util.concurrent.ForkJoinTask,` especially `RecursiveAction` or `RecursiveTask.`
   Because we do not have a recurrence formula, a trick is used. A fake `RecurentAction` class is introduced:

```java
static class JacobiTask extends RecursiveAction{
    private int index;

    JacobiTask(int index){
      this.index=index;
    }

    @Override
    protected void compute(){
      if(index==-1){
        for(int i=0;i<d.n;i++){
          JacobiTask action=new JacobiTask(i);
          action.fork();
        }
      }
      else{
        double s=0;
        for(int i=0;i<d.n;i++){
          if(i!=index){
            s+=d.a[index][i]*d.x[i];
          }
        }
        d.y[index]=(d.b[index]-s)/d.a[index][index];
      }
    }
  }
```

   The invocation of the object *JacobiTask(-1)* is used to spawn recursively the needed tasks and to launch them asynchronously. The solver method is

```java
public void solve(){
    int processors=Runtime.getRuntime().availableProcessors();
    ForkJoinPool pool=new ForkJoinPool(processors);

    int ni=0;
    JacobiTask task=null;
    double nrm,dif;
    do{
      ni++;
      task=new JacobiTask(-1);
      pool.invoke(task);
      while(pool.isTerminated());
      nrm=0;
      for(int i=0;i<d.n;i++){
        dif=Math.abs(d.y[i]-d.x[i]);
        if(dif>nrm) nrm=dif;
        d.x[i]=d.y[i];
      }
      if(nrm<d.eps)
        d.error=0;
      else
        d.error=1;
    }
    while(nrm>=d.eps && ni<=d.nmi);
  }
```

4 Results and conclusions

Each example is solved several times with the same data. For each run the beginning and ending time are obtained and hence the duration of the computation. Finally the average time is computed:

```java
long averageTime=0,duration,beginTime,endTime;
for(int i=0;i<testsNumber;i++){
   beginTime=System.currentTimeMillis();
   //       Instantiates the main class
   //       Calls the method to solve the problem
```

```
    endTime=System.currentTimeMillis();
    duration=endTime-beginTime;
    averageTime+=duration;
}
duration/=testNumber;
```

In our experiences we have $testsNumber = 100$. The obtained computation time is only indicative because the time can't be measured with the same computer used to do the computation.

Instead of calling the `currentTimeMillis` method, the usage of the *perf4j* framework offers a more elaborate approach, but we will to keep the code as simple as possible.

On a 64 bit PC with an Intel CORE2 Duo CPU and Windows 7 Home Premium OS, we have run the classes using Java 6 and Java 7. The results of our computing experiences are given in the table

Java distribution	Time(ms)/Example					
	1	2	3	4	5	6
jdk-6u27-windows-x64	16.70	—	21.37	62.71	5.78	—
jdk-7-windows-x64	7.62	102.92	7.98	0.99	1.1	0.81

Of course the class based on the *Phaser* class (example 2) and the *fork-join* technique (example=6) can't be run using Java 6. Our conclusion is that the best results are obtained using the `ExecutorService` with *CyclicBarrier.* The discrepancy for *CyclicBarrier* technique between Java 6 and Java 7 is due to the need to introduce a waiting period after the launch of the threads (after the line 6 of the *solve* method). Java 7 doesn't require such a waiting time. A waiting period is needed by the usage of the *Phaser* class, too. We must also admit that `ExecutorService` and `CyclicBarrier` offer a simpler programming template and that the *fork-join* technique allows the use of recurrence formulas in parallel-concurrent programs.

The present Java 7 release is more efficient than the last Java 6 release. The relations between the older tools are unchanged.

References

[1] D. Kincaid, W. Cheney, *Numerical Analysis Mathematics of Scientific Computing.* Brooks/Cole Publishing Company, Pacific Grove, California, 1991.

[2] J.M. Bahi, S. Contassot-Vivier, R. Couturier, *Parallel Iterative Algorithms. From Sequential to Grid Computing.* Chapman & Hall/CRC, Boca Raton, 2007.

[3] J. Ponge, Fork and Join: Java can Excel at Painless Parallel Programming Too! `http://www.oracle.com/technetwork/articles/java/fork-join-422606.html`.

[4] * * *, Java Fork/Join for Parallel Programming. `http://www.javacodegeeks.com/2011/02/java-forkjoin-parallel-programming.html`.

Scheiber Ernest
Transilvania University of Braşov
Department of Computer Science
Str. I. Maniu 50
ROMANIA
E-mail: *scheiber@unitbv.ro*