Concurrent Programming

Bernard Sufrin

Hilary Term 2016-17

3: Synchronous Data Parallel Programming

Reading: Andrews, Chapter 11.

Introduction

In this chapter we will study a particular style of data parallel programming, where the processes proceed *synchronously*; i.e. they perform some kind of global synchronisation at the end of each iteration (a.k.a. *round*).

Typically, each process will operate on one section of the data. However, the data calculated by one process might need to be read by another process on the next round; this data can be distributed:

* By writing to shared variables; in this case, there needs to be a global synchronisation part way through each iteration, after processes have finished reading the shared variables, and before they start writing to them.
* By sending messages; this works well when each piece of data has to be passed to only a few other processes.

These algorithms are sometimes known as heart-beat algorithms.

Applications

* Cellular automata
* Image processing
* Solving differential equations, for example in weather forecasting or fluid dynamics
* Matrix calculations

When the data is organised as a matrix of cells it can sometimes be natural to allocate a horizontal strip of rows to each process. At the end of each round, each process communicates the state of its top row to the process above it, and communicates the state of its bottom row to the process below it.

NB: This section is intended to demonstrate the *architecture* of synchronous data parallel programs as much as to be a piece of advocacy for implementing such programs in Scala or Java.

Barrier synchronisation

The global synchronisation at the end of each round is sometimes known as a *barrier synchronisation*, since it represents a barrier than no process may pass until all have reached that point.

A barrier synchronisation object is created by:

```
val barrier = new Barrier(p)
```

where `p` is the number of processes (usually with `p>1`).

Each process performs the barrier synchronisation by executing

```
barrier.sync()
```

No call to `sync` will return until all `p` processes have called it.

We will see later how to implement barrier synchronisation.

Simulating a simple cellular automaton

* The simulation is modelled by an array of N `Cell`s
* Each cell is initialized and then controlled (forever) by a distinct process
* In each round the next state of each cell is computed from the current states of the others
* Barrier synchronization ensures controllers update the model *in phase* with each other.

```
val model = Array.ofDim[Cell](N)
val barrier = new Barrier(N)
def Controller(i: Int) = proc {
    model(i) = computeInitialState(i)
    while (true)                                  // invariant:  model states are in phase
    { barrier.sync                                // model is consistent
      val next = computeNextState(i, model) // reading (all) current model states
      barrier.sync                                // (all) next states computed
      model(i) = neext                            // write (my) next model state
    }
}
val Simulation = || (for (i ← 0 until N) yield Controller(i))
```

Termination using a Combining Barrier

* Sometimes we need a way for worker/controller processes to agree terminate a calculation
* We need a way of combining local judgments about termination and distributing the combined conclusion to all participants.
* A *combining* barrier synchronisation is like a normal barrier synchronisation, except each process i contributes some piece of data x_i, and they all end up with the value

$$f(x_0, f(x_1, f(x_2, \ldots, f(x_{N-1}, e))\ldots)$$

for some associative and commutative function f with identity e.

* For example, the boolean combining barrier declared by

```
val b = CombiningBarrier[Boolean](N, true, (_ && _))
```

has method `sync(vote: Boolean): Boolean` that yields the conjunction of all termination votes to each participant

Example: smoothing an image

* An image will be represented as an array $image$ holding $M \times COLS$ pixels (represented as integers).
* We want to *smooth* the image by repeatedly setting $image(i, j)$ to the average of its immediate neighbours computed by: `average(image, i, j)`
* We are going to use `WORKERS` worker processes; each responsible for updating a region of width $COLS$ and height (about) $(M / WORKERS)$ rows.
* We will use an ordinary barrier, to synchronize the (end) of the computing phase and a combining barrier, to synchronize the (end) of the updating phase, and to aggregate the termination votes of the individual workers.

```
val computed = new Barrier(WORKERS)
val updated  = new CombiningBarrier[Boolean](WORKERS, true, (_ && _))
```

* The smoothing should terminate after at most `maxRounds` iterations, or when the image pixels have "stabilised".

```
def worker( startrow: Int, ROWS: Int ) = proc
{ val local    = makeImage(ROWS, COLS)  // local copy of the image region
  var rounds   = 0
  var finished = false                  // result of global termination vote
  var stable   = true                   // no local pixel has changed
  while (!finished)                     // recompute the local copy
  { for (row←startrow until startrow+ROWS; col ←0 until COLS)
    { val v = average(image, row, col)  // READ neighbours from global image
      local(row-startrow)(col) = v      // update the local pixel
      stable = stable && v == image(row)(col)
    }
    computed·sync()                                          // sync ends of READ phase

    writeImage(ROWS, COLS)(local, 0, 0)(image, startrow, 0) // publish image region

    finished = updated·sync(stable || rounds>= maxRounds)   // sync ends of WRITE phase
    rounds += 1
    stable = true
  }
}
```

Discussion: What happens if nearly but not all regions have become stable?

Particle computations

We will consider the problem of simulating the evolution of a (large) collection of N particles (*e.g.* stars or planets) that are subject to gravitational forces. We are going to implement a discrete time simulation, with time quantum `deltaT`.

We will implement it as a concurrent program that will record the mass, position and velocity of each particle in the simulation. The latter quantities are vectors in 3-space.

For reasons of engineering hygiene we will make these the attributes of a single type.

```
trait AbstractParticle
{ def mass:     Double
  def position: Position                    // a Vector.Variable
  def velocity: Velocity                    // a Vector.Variable
  ...
}

val allParticles: Seq[AbstractParticle] = ··· // allParticles.size == N
```

Appropriate types and methods have been defined in the `Vector` package to implement vector arithmetic, scaling, summation, etc.

Simple physics

A particle p_1 will exert a force on a particle p_2 of magnitude

$$G \times p_1.mass \times p_2.mass / distance^2$$

where $G \approx 6.67 \times 10^{-11}$ is the gravitational constant, and $distance$ is the distance between them. This is an attractive force, directed along the vector from the position of p_1 to that of p_2

It is computed by a method of `AbstractParticle`

```
def attractionTo(that: Particle): Force =
{ val magnitude = G * this·mass * that·mass / (this·position squareTo that·position)
  (this·position directionTo that·position) * magnitude
}
```

After each stage of the simulation we will need to calculate the total force exerted on each particle by *all* other particles.

We can then update the particle's position and velocity using another method of `AbstractParticle`.

```
def nextState(totalForce: Force): Unit =
{
  velocity += totalForce * deltaT / mass
  position += velocity * deltaT
}
```

Calculating the total force on each particle sequentially

If we are content to use a single process for the simulation, then we can re-calculate, at each stage of the simulation:

```
val totalForce: Seq[ForceVariable] = ··· // force.size == N
```

The recalculation is:

```
for (i ← 0 until N) totalForce(i)·setZero
for (i ← 0 until N) for (j ← 0 until N if j != i)
    totalForce(i) += allParticles(i) attractionTo allParticles(j)
```

and the final step of each stage is to update the position and velocity of each particle:

```
for (i ← 0 until N) allParticles(i)·nextState(totalForce(i))
```

* But the sequential algorithm is quadratic!

Towards a concurrent algorithm

The magnitude of the attraction of particle i towards particle j is the same that of the attraction of particle j towards particle i (but in the opposite direction), and it need not be calculated twice.

We propose to use P worker processes, and will allocate each process its own set $mine$ of particles. For each particle $i \in mine$, the process will calculate the forces between i and all particles j with $j > i$.

The obvious way to go about this in each process is:

```
for (i ← mine) for (j ← i+1 until N)
{ val force = allParticles(i) attractionTo allParticles(j)
  totalForce(i) += force
  totalForce(j) -= force
}
```

But this has an obvious race-condition: *several* processes might be trying to write at the same index of `totalForce` simultaneously.

Avoiding race conditions

To avoid the race condition we arrange for each of the P processes to work on its own, private, array of forces.

```
val localForces = Array.ofDim[Vector](P,N)
// localForces.size = P
// ∀ p ∈ [0..P) · localForces(p).size == N
```

In process me with particles $mine$ the initialization is:

```
localForce = localForces(me)                                  // initialize me^th row
for (i ← 0 until N) localForce(i) = new ForceVariable()       // magnitude = 0
```

.. and at each stage of the simulation the private data is updated by:

```
for (i ← 0 until N) localForce(i).setZero                     // magnitude = 0
for (i ← mine) for (j ← i+1 until N)
{ val force = allParticles(i) attractionTo allParticles(j)
  localForce(i) += force
  localForce(j) -= force
}
```

Calculating the total forces

* When all the `localForce` values have been calculated, the processes will perform a barrier synchronisation.
* Then the processes will calculate the *total* force on each particle they were allocated, and update the states of those particles.
* Process w with set of particles $mine$ calculates as follows:

```
for (i ← mine)
{ val totalForce = new ForceVariable()        // magnitude = 0
  for (w ← 0 until P) totalForce += localForces(w)(i)
  allParticles(i)·nextState(totalForce)
}
```

* $P \ll N$, so the cost of the "extra" summation in the worker is comparatively small.
* At this point the state of each particle has been determined, and the processes perform another barrier synchronisation before the next round.

Detailed implementation of a Worker process

```
def worker(me: Int, mine: Seq[Int]): PROC = proc(s"worker($me)")
  { val localForce = localForces(me)
    for (pid ← 0 until N) localForce(pid) = new ForceVariable() // magnitude = 0 †
    val totalForce = new ForceVariable()                        // magnitude = 0
    barrier·sync()

    while (true)
    { for (pid ← 0 until N) localForce(pid)·setZero()          // magnitude = 0 †
      for (pid ← mine) for (other ← pid + 1 until N)
      { val force = allParticles(pid) attractionTo allParticles(other)
        localForce(pid)   += force
        localForce(other) -= force
      }
      barrier·sync()

      for (pid ← mine)
      { totalForce·setZero
        for (w ← 0 until P) totalForce += localForces(w)(pid)
        allParticles(pid)·nextState(totalForce)
      }
      barrier·sync()
    }
  }
```

The pattern of synchronisation

* Workers are responsible for disjoint parts of the shared global state of the computation.
* They have a local "private" workspace in which the new state of their part can be computed without interference from the shared global state.
* They then use some (or all) of the disjoint local parts to update the shared global state.

```
// initialise private workspace
barrier.sync
while(true)      // Invariant: all parts of the global state are in phase
{ // reading shared global state, writing private workspace
  barrier.sync  // all private workspaces are up-to-date
  // reading possibly-all private workspaces, writing my part of the shared global state
  barrier.sync  // all parts of the shared global state are in phase
}
```

The final synchronisation on each iteration could be replaced by a synchronisation using a combining barrier, to decide whether to continue.

Load balancing

* We want to choose the sets $mine$ allocated to different processes so as to balance the total load (between available CPUs).

* The cost of calculating all the forces for particle i is $\Theta(N - i)$ – so not all particle computations are equal!

* One way to balance the load is to split the N particles into $2P$ segments, each of size $segSize = N/2P$, and allocate process me the segments me and $2P - me - 1$, *i.e.*

$$me \times segSize \text{ until } (me + 1) \times segSize$$

and

$$(2P - me - 1) \times segSize \text{ until } (2P - me) \times segSize$$

* This tactic is adopted in the published program.

Displaying the state of the simulation

* In the published program an extra $Display$ process shares a $Barrier(P+1)$ with the workers.

```
// initialize the GUI and tell the display about the collection of particles
...
display = new Display[Particle](allParticles, ···)
barrier·sync()

// run the simulation
while (true)
   { display·draw()   // draw the particles
     barrier·sync()   // local updates overlap with the drawing
     barrier·sync()   // global updates complete, so particles are drawable
   }
```

* The call $display.draw$ returns only when all particles have been drawn
* Local workspace updates overlap the drawing
* Only when global updates are complete can the particles be drawn again
* Problem: Displaying one drawing frame per cycle might be too fast for the eye!

Slowing the drawing-rate down

The following method takes *at least* time t, and runs $body$. Evaluation might overrun the time, and if so this is reported:

```
def takeTime(t: Nanoseconds)(body: ⇒ Unit): Unit =
  {  val deadline = nanoTime + t
     body
     val ahead = deadline-nanoTime
     if (ahead<=0) reportOverrun else sleep(ahead)
  }
```

We now modify the main loop of the $Display$ process so that the pace of showing successive "frames" of the simulation can be controlled:

```
while (true)
   { display.draw()              // draw the particles
     takeTime (seconds(1.0/FPS))
     { barrier.sync()            // local updates overlap with the drawing
       barrier.sync()            // global updates complete, so particles are drawable
     }
   }
```

So the frame remains visible for *at least* the specified time.

Barrier Implementation with Semaphores

* We define `class Barrier(n: Int){ def sync = { ···} }` to implement barrier synchronisation.

* We say that a process calling `barrier.sync` is "enlisting in $barrier$'s next round".

  The implementation's job is to make the first $n-1$ enlisting processes wait for the nth enlisting process, then let them all "into the round" (by leaving `sync`).

* The variable `waiting` records the number of processes currently waiting to leave `sync`.

* The boolean semaphore `gate`, initially closed (down, unavailable), is used to block the first $n-1$ "enlisting" processes in `sync`.

* The nth enlisting process raises the semaphore to unblock a waiting process. This in turn raises the semaphore to unblock the next waiting process, and so on ("passing the baton"), until they have all successfully enlisted in the round (by returning from `sync`).

* The following intended implementation has a bug:

```
class Barrier(n:Int)
{ private var waiting = 0          // # processes waiting
  private val gate = new Sema(available=false)
  def sync = {
    if (waiting==n-1)
       gate·release                // last enlisting process opens gate
    else
    {  waiting+=1
       gate·acquire                // enlistee waits at the gate
       waiting-=1
       if (waiting>0) gate·release // enlistee opens gate for waiting successor
    }
  }
}
```

* The variable `waiting` can be accessed by several processes simultaneously. Enlistment in a round is not atomic.

  While processes are leaving sync, a new process may enlist (possibly on its next round); it will wait on `gate`, but may subsequently be woken up and leave sync instead of one of the processes that were there earlier.

* Solution: add a mutex to ensure atomicity of enlistment
* All enlisting processes except the last wait at gate
* Last process to leave sync reopens mutex for the next round

```
class Barrier(n:Int)
{ private var waiting = 0                  // # processes waiting
  private val gate  = new Sema(available=false)
  private val mutex = new Sema(available=true)

  def sync = {
    mutex.acquire
    if (waiting==n-1)
       gate.release                  // last enlisting process opens gate
    else
    {  waiting+=1
       mutex.release; gate.acquire  // enlistee waits at the gate
       waiting-=1
       if (waiting>0)
          gate.release              // enlistee opens gate for a waiting successor
       else
          mutex.release            // allow next round to start
    }
  }
}
```

* The first $n-1$ processes to enter end up blocked on `gate.acquire`;
* The nth process performs `gate.release`, to unblock one process, and exits;
* Now $n-2$ processes of the processes stalled at `gate` are woken in turn, decrement `waiting`, and wake up the following process;
* The last of these processes to be woken performs `mutex.release` to allow the following round of synchronisation to proceed.

After the nth process has entered, no other process can enter until the last process exits, since `mutex` is down.

The implementation of the barrier synchronisation illustrates a technique known as *passing the baton*.[1] By releasing a semaphore, a process "passes the baton" to another process that's waiting, or about to wait, on it and allows it to continue.

* During the first stage, while `waiting < n`, each process passes the baton to a process that's starting (or yet to start) the `sync` method, by performing `mutex·release`.
* The nth process passes the baton to one of the processes waiting on `gate` to leave `sync`, by performing `gate·release`.
* The next $n - 2$ processes to leave pass the baton to another process waiting on `gate`, by performing `gate·release`.
* The last process to leave passes the baton to a process that's starting (or yet to start) the `sync` method, by performing `mutex·release`.

[1] By analogy with a relay race!

Supplement: Jacobi Iteration – Specification

We[2] now study a program to find an approximate solution to a large system of simultaneous linear equations.

Given an N by N matrix $A = (a_{ij})_{i,j=0,\ldots,N-1}$, and a vector b of size N, we want to find a vector x of size N such that $Ax = b$.

[2] Gavin Lowe wrote this section

We decompose A as $A = D + R$ where D contains the diagonal entries of A, and R contains the rest of the entries. Then

$$\begin{aligned} & Ax = b \\ \Leftrightarrow\ & Dx + Rx = b \\ \Leftrightarrow\ & x = D^{-1}(b - Rx) \end{aligned}$$

provided $a_{ii} \neq 0$ for all i (so D^{-1} exists).

This suggests calculating a sequence of approximations to the solution by taking $x^{(0)}$ arbitrary (say all 0s), and

$$x^{(k+1)} = D^{-1}\left(b - Rx^{(k)}\right).$$

It can be shown that this iteration will converge on a solution if

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}|.$$

It will be convenient, when doing the implementation, to have the "point-free" equation

$$x^{(k+1)} = D^{-1}\left(b - Rx^{(k)}\right),$$

expressed in component form:

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)}\right),\ i = 0, \ldots, N-1.$$

Towards a sequential implementation

Given

```
val N = ···; // size of array
val A = Array·ofDim[Double](N,N)
val b = Array·ofDim[Double](N)
```

we will calculate the solution in

```
val x = Array·ofDim[Double](N)
```

We start by considering a sequential program to do Jacobian iteration. What needs to be done on each iteration is *simultaneously* set each `x(i)` by

```
var sum: Double = 0
for (j ← 0 until N) if(j!=i) sum += A(i)(j)*x(j)
x(i) = (b(i)-sum) / A(i)(i)
```

It is evident that doing this *sequentially* for each `i` in turn does not have the required effect!

Using a second array

Instead, we could have another array

```
val newX = Array.ofDim[Double](N)
```

and initially do

```
var sum: Double = 0
for (j ← 0 until N) if(j!=i) sum += A(i)(j)*x(j)
newX(i) = (b(i)-sum) / A(i)(i)
```

for each `i`; and then copy the values back from `newX` into `X`.

But the copy-back is inefficient!

A two-stage algorithm

To remedy the inefficiency split each iteration into two stages.

In the first stage each `newX(i)` is calculated by

```
var sum: Double = 0;
for (j ← 0 until N) if(j!=i) sum += A(i)(j)*x(j);
newX(i) = (b(i)-sum) / A(i)(i);
```

And in the second stage each `x(i)` is calculated by

```
var sum: Double = 0;
for (j ← 0 until N) if(j!=i) sum += A(i)(j)*newX(j);
x(i) = (b(i)-sum) / A(i)(i);
```

Termination

We will consider the process to have converged when, for all `i`, the difference between `x(i)` and `newX(i)` is less than

```
val EPSILON = 0.000001;
```

We can check this while updating the `x(i)`.

A complete sequential implementation

```
def Solve =
{ var finished = false
  while (!finished) {
    for (i ← 0 until N) {
      var sum: Double=0
      for(j ← 0 until N) if (j!=i) sum += A(i)(j)*x(j)
      newX(i) = (b(i)-sum) / A(i)(i)
    }
    finished = true
    for (i ← 0 until N) {
      var sum: Double=0
      for (j ← 0 until N) if (j!=i) sum += A(i)(j)*newX(j)
      x(i) = (b(i)-sum) / A(i)(i)
      finished = finished && Math·abs(x(i)-newX(i)) < EPSILON;
    }
  }
}
```

Towards a concurrent implementation

For the concurrent solution we will use `W` workers.

We will split `x` and `newX` into `W` disjoint segments, and arrange for each worker to complete one segment.

For simplicity, we will assume $N \bmod W = 0$, and take each segment to be of height

```
val height = N/W
```

The concurrent solution will proceed in rounds, each round corresponding to one iteration of the sequential solution. Each round will have two stages. During the first stage, all workers can read all of x and each worker can write its own segment of newX; and during the second stage all workers can read all of newX and each worker can write its own strip of x.

We need to avoid race conditions, so we perform a barrier synchronisation at the end of each stage.

We will use a conjunctive combining barrier:

```
val conjBarrier = new CombiningBarrier[Boolean](W, true, (_ && _))
```

and each process will execute

```
finished = conjBarrier.sync(finishedLocally)
```

to cast its vote for termination and retrieve the global aggregation of votes

We also need a standard barrier synchronisation at the end of the first stage of each round:

```
val barrier = new Barrier(W)
```

A worker process, parameterised by `start,end`, calculates the slice `x[start:end)`

```
def Worker(start: Int, end: Int) = proc
{ var finished = false
  while (!finished)
  { // calculate our slice of newX from x
    for (i ← start until end)
    { var sum: Double=0
      for (j ← 0 until N) if (j!=i) sum += A(i)(j)*x(j)
      newX(i) = (b(i)-sum) / A(i)(i)
    }
    barrier·sync()
    // all workers have written their own slice of newX

    // calculate our slice of x from newX
    var finishedLocally = true
    for (i ← start until end)
    { var sum: Double=0
      for (j ← 0 until N) if( j!=i) sum += A(i)(j)*newX(j)
      x(i) = (b(i)-sum) / A(i)(i)
      finishedLocally = finishedLocally && Math·abs(x(i)-newX(i)) < EPSILON
    }

    // cast our vote for termination, retrieve the aggregated votes
    finished = conjBarrier·sync(finishedLocally)
    // all workers have written their own slice of x for this iteration
}}
```

The concurrent algorithm

The concurrent algorithm is the parallel composition of W workers:

```
def System =
    || (for (i ← 0 until W) yield Worker(height*i, height*(i+1)));
```

Experimental results

For small values of N, the sequential version is faster; but for larger values of N, the concurrent version is faster. Why?

The computation time for each round is $\Theta(N^2)$ for the sequential version, or $\Theta(N^2/W)$ for the concurrent version. But there is a communication time of $\Theta(N)$ for the concurrent version, to keep the values in the caches up to date, and this is overwhelming for small values of N.

Contents

Introduction . . . 2
Introduction . . . 2
Applications . . . 3
Barrier synchronisation . . . 4
Simulating a simple cellular automaton . . . 5
Termination using a Combining Barrier . . . 6
Example: smoothing an image . . . 7
Particle Computations . . . 9
Particle computations . . . 9
Simple physics . . . 10
Calculating the total force on each particle sequentially . . . 12
Towards a concurrent algorithm . . . 13
Avoiding race conditions . . . 14
Calculating the total forces . . . 15
Detailed implementation of a Worker process . . . 16
The pattern of synchronisation . . . 17
Load balancing . . . 18
Displaying the state of the simulation . . . 19
Slowing the drawing-rate down . . . 20
Barrier Implementation . . . 21
Barrier Implementation with Semaphores . . . 21
Supplement: Jacobi Iteration . . . 26
Supplement: Jacobi Iteration – Specification . . . 26
Towards a sequential implementation . . . 29
Using a second array . . . 31
A two-stage algorithm . . . 32
Termination . . . 33
A complete sequential implementation . . . 34
Towards a concurrent implementation . . . 35
The concurrent algorithm . . . 39
Experimental results . . . 39

Note 1: 2 ☞
We will not discuss the message-passing forms of data-parallel programming here.

Note 2: 7 ☞
(inventing "virtual" rows and columns to act as the neighbours of the columns on the boundary)

Note 3: 9 ☞
In order to make our code more intelligible we have incorporated the following type definitions in our particle simulator:

```
type Position      = Vector·Variable
type Velocity      = Vector·Variable
type Force         = Vector·Value
type ForceVariable = Vector·Variable
```

A `new Vector·Variable()` has zero magnitude, as does a `new Vector·Constant()`

$v.setZero$ re-zeros a $Vector.Variable\, f$.

See the published `Vector` package, and the published file `Particles·scala` for more detail.

Note 4: 10 ☞
The methods `squareTo` and `directionTo` respectively compute the square of the Euclidean disance between two vectors, and the normalized direction vector between them. The vector method $*(s : Double) : Vector.Value$ returns vector value scaled by s.

Note 5: 16 ☞
Note, at †, that in finalizing the detailed implementation we moved the allocation of the force variable $totalForce$ out of the main loop into the initialization, and replaced it with $totalForce.setZero$. This tiny "strength reduction" optimization saves one reallocation per simulation cycle – a worthwhile increase in efficiency.

Note 6: 25 ☞
Note that each awakened process passes the baton to precisely one other process.