

FAST GUIDE TO USING THE RIT PARALLEL JAVA LIBRARY

An introduction to
Java Parallel Programming
using an API

Jonathan Jude

1st Edition

© Jonathan Jude 2008

All rights reserved by the Author. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means electronic, mechanical, photocopying, recording or otherwise without the prior permission of the Author, the copyright holder.

Cover: Jude / MALM

Publisher : www.guideline.ch

**Published & Printed by: Books on Demand
GmbH, Norderstedt,
Deutschland.**

ISBN 978-3-8370-2439-5

Programming Task example

All programmers have at some point needed to write some code or a function that performs a task on a table. Tables are everywhere in computing. They could be tables in a data-base, or an electronic picture (which is just a table of pixel color values). In scientific computing one of the simplest cases is tables of results. In the field of computer systems management, gathering data on systems performance is one of creating and analyzing data tables. Let us then take the following example as the basis for a parallel program:

- A monitor process gathers data at regular intervals on the CPU usage of each of the applications that run on the system. ¹
- A table is built where each row contains a list of values of CPU usage over time for an application. ²
- The task is to calculate the average CPU usage for each application.

On corporate or departmental systems, there can be many hundreds of processes running.

(1) This is not about how the data is generated. There are a number of ways to do it depending on the computer system used. On a UNIX system this could be a *cron* job that runs every ten minutes and on each run will gather data on the current CPU usage of the top load generating processes.

(2) For the sake of argument, we'll assume that the timestamps and the application names are collected separately in arrays where the timestamp array has the same number of entries as there are columns in the data table, and the application name array contains the same number of entries as there are rows in the data table.

In our case here we'll scale the table to be one that contains data for 512 processes, and the monitor process has collected data every 10 minutes for four whole days (96 hours).

This gives a total of:

$$\frac{(96 \times 60)}{10} = 576 \text{ Entries}$$

The base table of data is then 512 rows, 576 Columns, being a total of 294912 data values.

Average Calculation algorithm

Knowing that Java stores data in Row-major format in memory (See Basic Concepts chapter) tells us how to get speed in processing. We need to work on the data row by row. This means each row of the CPU statistics table has to contain consecutive “time-stamped” CPU readings for a given application. The calculation of the average CPU usage for each application is then to add all of the entries for a given row together and divide by the number of entries in the row.

If we let ***E*** be the total number of entries per row (columns in the table), and ***N*** be a numeric identifier for a process (the row index for a given application), we can describe this mathematically, as shown in Figure 2 below.

$$AvgeCPU_N = \frac{\left(\sum_{j=0}^{(E-1)} Table[N][j] \right)}{E}$$

- Where E is the total number of columns in the table
- Where N is index of the row for a given application

Figure 2 : Mathematical representation of Average Calculation

To put this into a programming context, a simple non-parallel pseudo-code program to calculate the average could be as follows:

```
for each application
  average_value=0
  cpu_sum=0
  for each CPU usage value
    cpu_sum = cpu_sum + next CPU usage value
  endfor_cpu
  average_value = cpu_sum / number of CPU usage values
  save average_value to results
endfor_application
```

Because Java stores data in Row-major order the two dimensional for-loop of the code from **Example 2** (P.22) above can be used. It will run over each element in a row, row by row.

At this point a further specification can be made that a requirement for the programming of the parallel task is that the parallel part uses all of the available processors to perform the average calculation task. How to split up the task is discussed in the next section.

Decomposing for Parallel execution

Because accessing tables in Java uses indexes we can say that for however many processes we might want to have working on the calculation of average CPU values, we can give each process part of the table to work on. If we have two processors we can give each half of the table (and by this we mean a non-overlapping half). If we have four processors, we can give each process a quarter of the table to work on, etc.

To make things as easy as possible and to ensure we don't have to intervene to manually set the list of rows each process will work on, we need some kind of standard way of working out which indices each process should use. We want to work out indices that will let each individual process decide itself on where to start and where to stop.

Here we can specify the requirements of the table and what we want to do:

- The table contains values for 512 processes having 572 CPU usage values each.
- From the Row-major nature of Java we know that we want calculate an average for each row.

The solution we will use here is to give each process sets of rows, being a horizontal slice of the table.

Each process will be working on Row-major data in a table slice (a given number of rows), and each will be working on contiguous memory sections (see Figure 3).

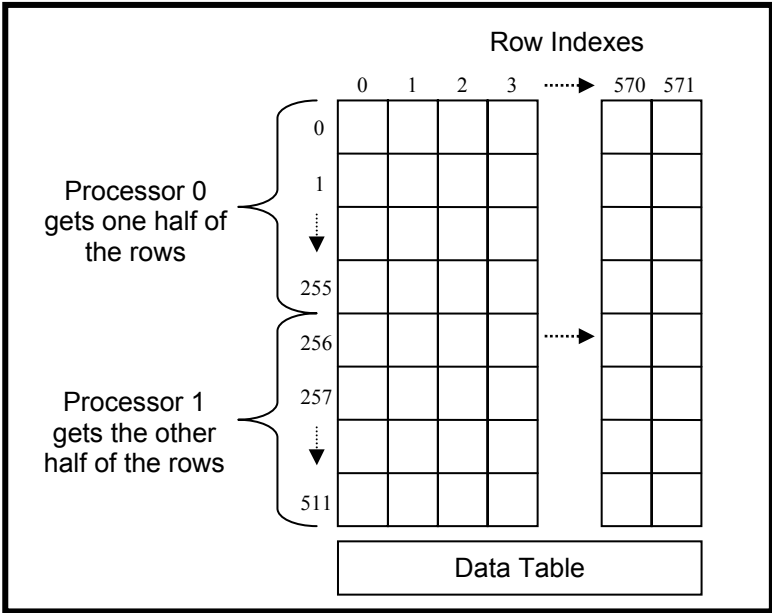


Figure 3 : Example table decomposition for two processors

If the starting index of the rows is 0 then the indices for the first process of two processes are the following:

$$\begin{aligned}
 \text{Processor0_Startrow} &= 0 \\
 \text{Processor0_slicesize} &= \left(\frac{\text{Number_of_Rows}}{\text{Number_of_processors}} \right) \\
 \text{Processor0_Endrow} &= (\text{Processor0_slicesize} - 1)
 \end{aligned}$$

But we also need to be *specific* about how to stop. If we start at 0 and perform calculations on 256 rows, where the index of the first row (here called the *Startrow*) is 0, the index of the last row to work on will be 255. But the process will have worked on 256 items in total, which is what we want.

We also need to say that the condition for stopping is that we arrive at the last row but that we process it too. So we need to say that the process will work up to and including the *Endrow*. If we were to write this in code, the outer for statement of our two dimensional **for** loop, it will look like the following:

```
int r1=0;
for (r1=Processor0_Startrow ; r1<=Processor0_Endrow ; r1++)
{
    ...
}
```

For the second process of two processes, again we need to be careful about the indices. The number of rows to work on should be the same as that of the first process, but the *Start row* value for the second process will be the one after the last row of the first process. In this case the *Start row* will be $255+1 = 256$.

The *Endrow* for the second process will be the end of the table, which (again taking 0 as the start index) will have the index 511:

$$\begin{aligned} \text{Processor1_Startrow} &= (\text{Processor0_Endrow} + 1) \\ \text{Processor1_slicesize} &= \left(\frac{\text{Number_of_Rows}}{\text{Number_of_processors}} \right) \\ \text{Processor1_Endrow} &= (\text{NumberOfRows} - 1) \end{aligned}$$

The outer for loop code for the second process is then:

```
int r2=0;
for ( r2=Processor1_Startrow ; r2<=Processor1_Endrow ; r2++)
{
    ...
}
```

But here we can already detect overlap between the two index calculations. We want to avoid if possible the reliance on a previously calculated value from the first process in order to get the indices for the second process. We also want to still keep the program as simple as possible, and abstract out those common parts of index calculation. We need then to make the *Startrow* value be calculated as a function of the Process ID.

Indices based on Processor ID

Each process or processor having its own ID gives us the possibility to calculate indices for the start and end of each set of rows that a process will work on. If we let each parallel process (running each on a processor) calculate its own set of indices, this is what the index calculation then becomes (PID = Processor ID):

$$\begin{aligned} \text{Slice_Size} &= \left(\frac{\text{Number_of_Rows}}{\text{Number_of_Processors}} \right) \\ \text{MyStartrow} &= \text{Slice_size} * \text{PID} \\ \text{MyEndrow} &= (\text{Slice_size} * (\text{PID}+1)) - 1 \end{aligned}$$

We can apply these formulae to the problem in hand for a table of 512 rows.

So for the processor of ID 0 (of two processors, of ID 0 and 1) this will give:

$$\begin{aligned} \text{Slice_Size} &= (512/2) &&= 256 \\ \text{MyStartrow} &= \text{Slice_size} * \text{PID} &&= 0 \\ \text{MyEndrow} &= (\text{Slice_size} * (\text{PID}+1)) - 1 &&= 255 \end{aligned}$$

For the processor of ID 1 (of two processors, of ID 0 and 1) this will give:

slice_size	= (512/2)	= 256
MyStartrow	= Slice_size*PID	= 256
MyEndrow	= (Slice_size * (PID+1)) - 1	= 511

This works very well for numbers of processors and problem sizes that are factorized using powers of 2.

The Base-2 number system is the underlying nature of all things pertaining to the internals of computers, and this from the basic logic of 0 or 1, through address and register sizes, to caches sizes and memory size etc.

What then if neither the number of processors nor the problem size is a power of two?

Let's take the example of 3 processors, working on the same example. For processor 0:

$$\text{Slice Size} = \left(\frac{512}{3} \right) = 170.6666\dots$$

Now we have a problem. We cannot have 0.6666 of a line, or worse, a floating-point table index! We need to rationalize this using functions that will give us not only a whole number of rows in a slice, but also, if there are any rows left over once the slice size is determined as a whole integer, let us give these remaining rows to a processor so as to cover all rows. In Java we need to use the "*floor ()*" function for floating point numbers and the modulus operator "%" inherent in the language.

Our index calculation for three processors is thus:

```
nprocessors=3
numberofrows=512
Slice_Size = (int) floor((float) numberofrows / (float) nprocessors)
```

That is to say

```
Slice size = (int) floor((float)512/(float)3) = 170
```

The number of remaining rows is calculated thus:

```
extra_rows = numberofrows % nprocessors
```

In the example this is:

```
extra_rows = 512 % 3 = 2
```

(i.e. 3 into 512 is one hundred and seventy times + a remainder of two).

So we can say that two of the three processors in a 3 processor machine will work on 170 rows each, and one of them will have to work on 172. For the sake of argument, and simplicity we will work on the principle that the processor with the highest ID will get the extra rows to work on. We can again use the PID to let the process decide if it will work on the extra rows. In the worked examples below we'll let the variable **myPID** contains the physical processor ID value.

For example:

```
Numprocessors =3
Numberofrows =512
Slice_Size   = (int)floor((float)numberofrows/(float)nprocessors)
MyStartrow   = (slice_size*myPID)
MyEndrow     = (slice_size*(PID+1)) - 1
// if last processor then add the remaining rows
if (PID==(nprocessors-1)) then MyEndrow
    += (numberofrows%nprocessors)
```

Let's try this with 1, 2, and 3 as the number of processors just to be sure. Figures 4, 5 and 6 below and on the following pages show the results if we apply this method.

```
// 1 Processor Example

numprocessors = 1                // (myPID = 0)
numberofrows = 512
slice_size = (int) floor((float)512/(float)1)    // = 512

MyStartrow= (slice_size*myPID)    // = 0
MyEndrow= (slice_size*(myPID+1)) - 1    // = 511

if (myPID==(numprocessors-1)) then MyEndrow
    += (numberofrows%numprocessors)

// The test returns true, remainder is 0 so MyEndrow := 511
```

Figure 4 : Index Calculation : 1 Processor example

```
// 2 Processor Example  
// Some Global parameters – these are the same  
// for all processors  
  
numprocessors = 2 // (myPID = 0,1)  
numberofrows = 512  
slice_size = (int) floor((float)512/(float)2) //= 256  
  
//Processor 0  
MyStartrow= (slice_size*myPID) //= 0  
MyEndrow= (slice_size*(myPID+1)) - 1 //= 255  
  
if (myPID==(numprocessors-1)) then MyEndrow  
    += (numberofrows%numprocessors)  
// The test returns false so MyEndrow := 255  
  
//Processor 1  
MyStartrow= (slice_size*myPID) //= 256  
MyEndrow= (slice_size*(myPID+1)) - 1 //= 511  
  
if (myPID==(numprocessors-1)) then MyEndrow  
    += (numberofrows%numprocessors)  
  
// The test is true, but remainder is 0 so MyEndrow := 511
```

Figure 5 : Index Calculation : 2 Processor example

Yes, it seems to work, and we now have a way to allow each individual process to decide which part of the table it will work on, and if it has the highest processor ID, to do extra rows. We can now bring all of this together and look at an initial version of what our parallel program code shall look like, and the parts that will run in parallel.

```
// 3 Processor Example  
// Some Global parameters – these are the same  
// for all processors  
  
numprocessors = 3                // (myPID = 0,1 or 2)  
numberofrows = 512  
slice_size = (int) floor((float)512/(float)2)    //= 170  
  
//Processor 0  
MyStartrow= (slice_size*myPID)                //= 0  
MyEndrow= (slice_size*(myPID+1)) - 1          //= 169  
  
if (myPID==(numprocessors-1)) then MyEndrow  
    += (numberofrows%numprocessors)  
// The test is false so MyEndrow :=169  
  
//Processor 1  
MyStartrow= (slice_size*myPID)                //= 170  
MyEndrow= (slice_size*(myPID+1)) - 1          //= 339  
  
if (myPID==(numprocessors-1)) then MyEndrow  
    += (numberofrows%numprocessors)  
// The test is false so MyEndrow := 339  
  
//Processor 2  
MyStartrow= (slice_size*myPID)                //= 340  
MyEndrow= (slice_size*(myPID+1)) - 1          //= 509  
  
if (myPID==(numprocessors-1)) then MyEndrow  
    += (numberofrows%numprocessors)  
  
// The test is true, and remainder is 2, so MyEndrow := 511
```

Figure 6 : Index Calculation : 3 Processor example

Index

A

Acknowledgements · 8
Amdahl's Law · 23
Annexes · 82
Average Calculation
algorithm · 27

B

Basic Concepts · 20
Bibliography · 80
Blinkinglights · 10, 80

C

C, C++ · 21
CM-2 · 10
Column-major · 20, 21,
22
compute intensive
problems · 9
Conclusions · 71
Contents · 5
Conventions · 8

D

Data Table load
example · 82
Data Table loading
example · 82
Decomposing the
example · 29

E

Eclipse · 17, 81
Inserting Sources · 66
Running the
examples · 68
Examples of Parallel
Processing · 18
Execution Results · 71

F

Feedback · 83

G

Glossary · 82
Guideline Series Books
· 86

H

Henry Ford · 9

I

Increasing complexity · 50

Index · 84

Indices · 32

Initial Java Program · 38

Initial Parallel Program · 38

Introduction · 16

L

List of Figures · 7

M

Massively Parallel Processing · 9

MPP · 9, 11

P

parallel languages · 10

ParallelRegion · 43, 44, 45, 47, 48, 49

ParallelTeam · 43, 44, 46, 47, 48

personal computer · 9

PJ Parallel Java Library · 38, 42, 44, 46

Preface · 9

Process ID · 23, 32

Programming Task example · 26

R

RIT PJ Parallel Java · 15, 42

Rochester Institute of Technology · 42

Row-major · 20, 21, 22, 27, 28, 29

S

Setting up Eclipse · 57

T

Thinking Machines · 10

W

What Next? · 77