Timing Comparisons for FaceReplace using async, Actors, and Sequential Implementations

Tony Tuttle
January 21, 2014

Abstract
We wrote three implementations of the same program, one using async sections of code, one using Actor objects, and one sequential version. We compared the running times for these implementations. The implementation using Actor objects was consistently the fastest, followed by the one using async sections, then the sequential version.

Contents
1 Introduction 2
2 Experiment Details 2
  2.1 What the FaceReplace Program Does ......................... 2
  2.2 Setting up and Running the Timing Tests ..................... 2
3 Results 4
4 Conclusion 5

List of Figures
1 Example of the FaceReplace running with the GUI ............... 3
2 Running times with 64 × 64px subimages ....................... 5
3 Running times with 128 × 128px subimages ..................... 5
4 Running times with 256 × 256px subimages ..................... 6
5 Running times with 512 × 512px subimages ..................... 6

List of Tables
1 Results of frAsync timing tests ................................. 3
2 Results of frActors timing tests ............................... 4
3 Results of HJSep timing tests ................................ 4
4 Results of JavaSeq timing tests ............................... 4
1 Introduction

We performed timing tests on three different implementations of the FaceReplace program. We wrote two concurrent implementations and one sequential implementation, all using Habanero Java (HJ). HJ is a language developed at Rice University designed to allow parallel programming constructs to be added to standard Java programs in a relatively easy and safe manner.[3]

The first program mainly used the async construct. The second used the Actor model. The third was a sequential version of the program that was compiled and run using both the HJ compiler and the Java compiler.

Our purposes were to learn several things. First, we aimed to determine what sort of speedup we could expect with relatively simple use of the constructs offered in HJ. We also wanted to compare the running time required for the async construct and the Actor model for this particular program. Lastly, we wanted to test whether a sequential Java program compiled using the HJ compiler would have a similar running time to that compiled with the standard Java compiler.

2 Experiment Details

2.1 What the FaceReplace Program Does

The FaceReplace program takes as input two JPEG files. The images in these files are made up of a collection of smaller subimages, all of which have identical dimensions. Each subimage is a face. The two large images both contain the same faces, but they are permuted so that they appear in different areas of either of the large images. See Figure 1 for a visual representation. For clarity we refer to one of the large images as the source and the other as the target.

The program iterates through each subimage in the the source and the target, storing information about the color of the pixels that make up the subimage. Once this is done for every subimage in both images, the program matches the correct face from the source to that of the target. In Figure 1 you can see that the subimage that the cursor is hovering over, as well as the matching subimage on the right side, is highlighted green.

2.2 Setting up and Running the Timing Tests

Once all three implementations (frAsync, frActors, HJSeq) were written and tested for correctness, each was tested over 16 different problem sizes. The programs were run without the GUI component. This decision was made because none of the HJ code was located in the GUI, and because there were calls to Thread.sleep() injected into the GUI. We were interested specifically in how much faster HJ code would run than standard Java code and therefore isolated the part of the program where this could best be observed.

For each of the 16 problem sizes, the program was run eight times in order to get an accurate average execution time. There were two components to each problem size: the number of subimages and the size of each subimage. The number of subimages represented how many pieces (or faces) each image was broken into. The size of each subimage was the length, in pixels, of one side of a subimage. For

1Timing tests were done not with faces, but by ‘scrambling’ large images into equally-size subimages
example, if one whole image was 128 × 128 px, we could divide that image into 16 subimages, each having a size of 32 × 32 px.

Each implementation was tested on each of the 16 problem sizes. The problem sizes were a cross-product of each possible subimage size $p \times p$ pixels and $n$ total subimages where $p, n \in \{64, 128, 256, 512\}$. When attempting to run the programs with a problem size of $512 \times 512$ px subimages and 512 subimages, every implementation ran out of heap space. The frAsync implementation ran out of heap space on three other problem sizes as well. We are not certain why this implementation ran out of heap space sooner than the others.

The raw results of the timing tests can be seen in Tables 1-4.

<table>
<thead>
<tr>
<th>number of subimages</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
</tr>
</thead>
<tbody>
<tr>
<td>size of 64</td>
<td>90</td>
<td>57</td>
<td>96</td>
<td>191</td>
</tr>
<tr>
<td>of 128</td>
<td>93</td>
<td>180</td>
<td>355</td>
<td>656</td>
</tr>
<tr>
<td>each 256</td>
<td>330</td>
<td>664</td>
<td>1369</td>
<td>*</td>
</tr>
<tr>
<td>subimage 512</td>
<td>1401</td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
</tbody>
</table>

* Java ran out of heap space.

Table 1: Results of frAsync timing tests
3 Results

As can be seen from Figures 2-5, each of the concurrent implementations were consistently faster than the sequential implementation. The frActors implementation ranged from 1.81 to 2.76 times faster than the sequential implementation. The frAsync implementation ranged from 1.65 to 1.82 times faster. Thus, in terms of speedup the frActors implementation was the best choice.

In comparing the timing results of the frAsync with the frActors implementations, we can see that the latter was consistently faster. The values for frAsync time ranged from 1.10 to 1.54. Thus, the speedup from frAsync to frActors was less pronounced than that of moving from a sequential implementation to either of the concurrent implementations.

We also wanted to compare the running times for the HJSeq implementation when it was compiled using the HJ compiler versus the same program compiled with the Java compiler. The maximum variation between these times was 6% with most discrepancies running around 2-3%. 

### Table 2: Results of frActors timing tests

<table>
<thead>
<tr>
<th>number of subimages</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
</tr>
</thead>
<tbody>
<tr>
<td>size 64</td>
<td>24</td>
<td>52</td>
<td>84</td>
<td>137</td>
</tr>
<tr>
<td>of 128</td>
<td>66</td>
<td>131</td>
<td>245</td>
<td>467</td>
</tr>
<tr>
<td>each 256</td>
<td>226</td>
<td>438</td>
<td>867</td>
<td>1722</td>
</tr>
<tr>
<td>subimage 512</td>
<td>907</td>
<td>1767</td>
<td>3645</td>
<td>*</td>
</tr>
</tbody>
</table>

* Java ran out of heap space.

### Table 3: Results of HJSeq timing tests

<table>
<thead>
<tr>
<th>number of subimages</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
</tr>
</thead>
<tbody>
<tr>
<td>size 64</td>
<td>50</td>
<td>94</td>
<td>159</td>
<td>322</td>
</tr>
<tr>
<td>of 128</td>
<td>156</td>
<td>310</td>
<td>600</td>
<td>1195</td>
</tr>
<tr>
<td>each 256</td>
<td>589</td>
<td>1193</td>
<td>2337</td>
<td>4467</td>
</tr>
<tr>
<td>subimage 512</td>
<td>2287</td>
<td>4568</td>
<td>10062</td>
<td>*</td>
</tr>
</tbody>
</table>

* Java ran out of heap space.

### Table 4: Results of JavaSeq timing tests

<table>
<thead>
<tr>
<th>number of subimages</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
</tr>
</thead>
<tbody>
<tr>
<td>size 64</td>
<td>49</td>
<td>89</td>
<td>167</td>
<td>319</td>
</tr>
<tr>
<td>of 128</td>
<td>155</td>
<td>314</td>
<td>595</td>
<td>1195</td>
</tr>
<tr>
<td>each 256</td>
<td>593</td>
<td>1150</td>
<td>2296</td>
<td>4501</td>
</tr>
<tr>
<td>subimage 512</td>
<td>2291</td>
<td>4458</td>
<td>9803</td>
<td>*</td>
</tr>
</tbody>
</table>

* Java ran out of heap space.
Conclusion

We determined that a significant speedup could be expected by using tools offered by the HJ extension over standard sequential Java code. With minimal effort and expertise in concurrent programming, the writers of this code were able to gain 1.5 to well over two times the speed of the sequential code. It should be noted that the nature of this particular program made this an easier task than may be expected with other types of programs.
We also determined that there were significant differences in the speedup gained between the \texttt{frAsync} implementation and the \texttt{frActor} implementation, with the latter being consistently faster. Again, much of this discrepancy can be explained by recognizing that the problem at hand was essentially a producer-consumer problem. Actors lend themselves very naturally to this type of problem.

We determined that there were no significant timing discrepancies running identical code when it had been compiled with the HJ compiler versus when compiled using the Java compiler.

Figure 4: Comparison of running times for all three implementations with 256x256px subimages.

Figure 5: Comparison of running times for all three implementations with 512x512px subimages.
References


