Summer School Parallel Programming
From Multi-core to Many-core and beyond

Day 5

Dataflow programming with pipes and makefiles

Raphael kena Poss
r.c.poss@uva.nl

July 12, 2012

University of Amsterdam

VU University Amsterdam

1 Unix processes and pipes

With this activity, you will learn to use Unix pipes to connect processes together and form an asynchronous dataflow graph.

1.1 Wrapping programs

Exercise 1 Create a file "`resize.sh`" with the following contents:

```
#! /bin/bash
input=${1:?}
output=${2:?}
size=${3:?}

while true; do
   read filename < "$input"
   echo "Resizing $filename..."

   convert "$filename" -geometry $size "$filename".tmp
   mv "$filename".tmp "$filename"

   echo "Done!"
   echo "$filename" > "$output"
done
```

Then make this file executable.

Exercise 2 Here you get to use the script above to process a stream of input images.

1. create two named pipes "`source`" and "`sink`" (using `mkfifo`);
2. open one terminal, to run the following command:
   `./resize.sh source sink 100x100`
3. open another terminal, to run the following command:
   ```
   while true; do read name < sink; echo "processing done for $name";
   done
   ```
4. open another terminal, to run the following command in a directory where you have some images already created:
   `for f in *.ppm; do echo $PWD/$f > /path/to/source`

Exercise 3 Measure the time to convert 10 images of a megabyte each using this pipeline. What is the maximum throughput you can observe? What are the likely factors that limit the throughput?

Exercise 4 Using the same technique, wrap other image filters (including possibly your program from the previous days) in a way suitable for stream processing.

1.2 Parallel execution

Exercise 1 Here you get to parallelize your pipeline.

1. create a script `splitter.sh` which:
    - uses one input FIFO and two output FIFOs,
    - uses `mogrify -crop` to split the input image horizontally (top and bottom parts),
    - writes the names of the parts to their respective output FIFOs.
2. create another script `merger.sh` which:
    - uses two input FIFOs and one output FIFO,
    - reads two filenames from the input FIFOs,
    - uses `mogrify -append` to merge two input images into one output image,
    - writes the name of the output image to its output FIFO.
3. create 6 FIFOs and connect the splitter, merger and 2 instances of the resize script above.
4. test your dataflow network.

Exercise 2 Compare the throughput of resizing using only one resizer (exercise 3 above) and two resizers (exercise 4) for a fixed amount of input and output bytes. Do you get any speedup? Experiment with the number of parallel resizers to maximize throughput.

Exercise 3 Repeat exercise 2, using your parallelized image processing code from the previous days instead of `convert -geometry` as an individual filter. Do you get any speedup? How can you explain the difference?

2 Dataflow programming with GNU Make

In this activity, you will learn to use GNU make to automate the scheduling of an asynchronous dataflow graph.

2.1 Image filters

Exercise 1 Write the following `Makefile` (and beware of tabs!):

```
# input file list - remove the ".ppm" extension!
INPUTS = image1 image2 ...
SIZES = $(foreach W,10 20 50 100,$(foreach H,10 20 50 100,$(W)x$(H)))
```

```
OUTPUTS = $(foreach S,$(SIZES), $(INPUTS:%=out.%.$(S).ppm))
all: $(OUTPUTS)

.SECONDEXPANSION:
# The following rule implements a resizer
out.%.ppm: $$(word 2,$$(subst ., ,$$@)).ppm
     convert $(word 2,$(subst ., ,$@)).ppm -geometry $(word 3,$(subst ., ,$@)) $@
```

Exercise 2 Run the following sequence of commands:

```
rm out.*
time make -jN
```

for different values of N. What is the throughput every time?

Exercise 3 Modify the Makefile to use other sorts of image transformations. Measure the throughput for different combinations of `make -j`.

2.2 Benchmarking (optional)

You can also use GNU make to automate the execution of a program over many input files. The task of *benchmarking* can be represented as a dataflow graph, where each test is a node and edges represent files to process and benchmarking results.

Exercise 1 Download the code and documentation from `http://staff.science.uva.nl/~poss/bench/`. Decompress the archive and ensure the makefile works with the provided files. Read the documentation.

Exercise 2

1. Modify the variable `BENCHMARKS` in `Makefile` to set it to different names of image filters.
2. Adapt the script `progrun.sh` to run your programs from the previous days using the image filter name provided as 1st argument and image name provided as 2nd argument (you can ignore the 3rd argument "profile").
3. Create a `.inputs` file with a list of multiple input image file names.
4. Use `make bench` to execute the cross-product of filters to images.

Exercise 3

1. Experiment with the following combinations:
   - `make -j1` and a sequential filter program;
   - `make -j`N (with different values of N) and a sequential filter program;
   - `make -j1` and a parallel filter program;
   - `make -j`N (with different values of N) and a parallel filter program.
2. Which combinations yield the best speedup?