The benchmark-plot tool

Umut Acar, Arthur Charguéraud, Mike Rainey

17 October 2014

Synopsis

pplot [*MODE*] [*OPTIONS_FOR_MODE*]

Description

The benchmark-plot tool is a program that generates various types of plots for given data that is generated by the benchmark-run tool. Supported types include bar plots, scatter plots, tables, and speedup plots. Various temporary files used to generate plots and tables can be found in the folder `_results/`.

Options

Plot modes

The mode specifier *MODE* is optional: if not present, the default plot to be generated is the bar plot. Otherwise, *MODE* can be one of the following: `bar`, `scatter`, `table`, or `speedup`.

Mode-specific options

Options common to all

`-width` ***n*** Specify width *n* of the plot to be generated (in pixels).
`-height` ***n*** Specify height *n* of the plot to be generated (in pixels).
`-title` ***t*** Specify the title string *t* to be displayed in the plot.
`-input` ***filename*** Specify a file *filename* where to read results (default is `results.txt`).
`-output` ***filename*** Specify a file *filename* where to plot results.

Options common to `bar` and `scatter`

`-legend-pos [topright|topleft|...]` Select an R code for the legend (see R codes for legend or file `legend.ml`).
`-chart` ***k1,k2,...*** Select a combination of keys by which to divide program runs among charts.
`-series` ***k1,k2,...*** Select a combination of keys by which to divide program runs among series.
`-group-by` ***k1,k2,...*** Select a combination of keys by which to divide program runs among groups.
`-y` ***x*** Select the key to plot on the y axis.
`-ylabel` ***lab*** Label the y axis as *lab*.
`-ymin` ***n*** Set the origin of the y axis to *n*.
`-ymax` ***n*** Set the maximum value of the y axis to *n*.
`--yzero` Set the origin of the y axis to zero.
`--ylog` Use a log scale for the y axis.

Options for `scatter` only

`-xlabel` ***lab*** Label the x axis as *lab*.
`-xmin` ***n*** Set the origin of the x axis to *n*.
`-xmax` ***n*** Set the maximum value of the x axis to *n*.
`--xzero` Set the origin of the x axis to zero.
`-drawline` ***n*** Draw a horizontal line originating from *n* on the y axis.

Options for `bar` only:

`--xtitles-vertical` Use a vertical orientation for the label of the x axis.

or:

`-xtitles-dir [horizontal|vertical]` Set the orientation to use for the labels under the bars.

Options for `table` only

`-table` ***k1,k2,...*** Select a combination of keys by which to divide program runs among tables.
`-row` ***k1,k2,...*** Select a combination of keys by which to divide program runs among rows of the table.
`-col` ***k1,k2,...*** Select a combination of keys by which to divide program runs among columns of the table.

`-cell` ***k*** Select the key whose corresponding value is to be displayed in the cells of the table.

`-group-by` ***k1,k2,...*** Select a combination of keys by which to divide program runs among groups of columns in the table.

Options for `speedup` only

`-chart` ***k1,k2,...*** Select a combination of keys by which to divide the program runs among speedup charts.

`-series` ***k1,k2,...*** Select a combination of keys by which to divide the program runs among series.

`-group-by` ***k1,k2,...*** Select a combination of keys by which to divide the program runs among groups.

`-legend-pos [topright|topleft|...]` Specify the position of the legend in the plot (see R codes for legend or file `legend.ml`).

`--log` Use a log scale for the speedup curves.

`--factored` Generate a "factored" speedup plot (see instructions below).

Alternative syntax for providing `mode`

pplot -mode [*MODE*] [*OPTIONS_FOR_MODE*]

Examples

Building and running a simple experiment

The following commands build the benchmarking tools and then gather some data from a few runs of our example program, Fibonacci.

```
make -C examples/basic fib
make prun
prun -prog examples/basic/fib -algo recursive,cached -n 39,40 -runs 2
make pplot
```

Bar plots

The first plot we generate is a simple bar plot in which bars are grouped together by arguments of `n` and apart by arguments o `algo`. The y axis represents run time.

```
pplot -x n -y exectime -series algo
```

Another way to make the same bar plot is the following.

```
pplot bar -x n -y exectime -series algo
```

This command generates two bar plots: one for each value of `algo`, namely `recursive` and `cached`.

```
pplot bar -x n -y exectime -chart algo
```

In this case, bars are separated by arguments of both `n` an `algo`.

```
pplot bar -x n,algo -y exectime --xtitles-vertical
```

Scatter plots

We generate a scatter plot for the same data set as follows. Each algorithm in the plot is represented by a labeled curve.

```
pplot scatter -x n -y exectime -series algo
```

Tables

We can generate a table showing breakdowns of run times. The columns are represented by the values of `algo` and the rows by values of `n`.

```
pplot table -row n -col algo -cell exectime
```

A slight change gives a similar result, but one with multiple tables such that each table represents a different value of `algo`.

```
pplot table -row n -table algo -cell exectime
```

Another slight changes gives a similar table where rows are represented by combinations of values of both `n` and `algo`.

```
pplot table -row n,algo -cell exectime
```

Speedup plots

Traditional speedup plots

The speedup curve for a series of runs of a given parallel program is calculated by

$$\text{speedup}(P) = \frac{T_S}{T_P}$$

where P denotes the number of processors used by the program, T_S the time taken by the sequential baseline program, and T_P the the taken by the parallel program using P processors.

In order to generate a speedup plot, we need to follow a few simple steps. First, we need to ensure that the programs that we wish to benchmark print to `stdout` the running time in the format

`exectime` t

where t is a floating-point number that expresses wall-clock time (in seconds). For example, one such running-time report is `exectime 4.32`, which reports that running time was 4.32 seconds. Second, we need to perform some runs of the programs that we want to benchmark, collecting the data as we go. The data that we need to collect is generated automatically by the "speedup" mode of the `prun` tool. Third, after the benchmarking runs complete, we need to run the `pplot` command to generate the speedup plot.

Let us consider the following example, where we are going to compare the speedup curves resulting two competing algorithms. In our sample benchmark program, the algorithm to be measured is selected by the command-line key `-algo`. Our baseline algorithm is specified by the value `foo` and our parallel algorithm by the value `bar`.

```
make prun
prun speedup -baseline "examples/others/speedup.sh -algo foo" -baseline-runs 1  -parallel "e
pplot speedup
```

We can take multiple speedup curves as well. The following example extends our previous example by introducing a new parameter, namely `n`, and two values for `n`: 4 and 5. The plot generated for this example will have two speedup curves, one for each setting of `n`.

```
prun speedup -baseline "examples/others/speedup.sh -algo foo" -baseline-runs 1  -parallel "e
pplot speedup -series n
```

Here is another example, this time using our parallel Fibonacci benchmark.

```
prun speedup -baseline "bench.baseline" -parallel "bench.opt -proc 1,2" -bench fib -n 39
```

Factored speedup plots

A "factored speedup plot" is a speedup plot that shows, in addition to an actual speedup curve of a given benchmark program, three or more "synthetic" speedup curves for the same benchmark program. Each synthetic speedup curve is just a curve that shows how the actual speedup curve might look, provided that overheads related to a given source, such as scheduling overhead, could be disregarded.

Our plotting tool currently generates three synthetic speedup curves by default and one extra synthetic curve, optionally. The first synthetic curve is the ***maximal speedup curve***. This curve is defined by the function

$$\text{maximal}(P) = \frac{P \cdot T_S}{T_1}$$

where P denotes the number of processors, T_1 the running time of the parallel program on a single processor, and T_S the running time of the sequential baseline program. The maximal speedup curve gives a realistic upper bound on the parallel speedup by taking into account any additional work that must be performed by the parallel run compared to the sequential run.

The second synthetic curve is the ***idle-time-specific curve***. This curve is defined by the function

$$\text{idletime}(P) = \frac{P \cdot T_S}{T_1 + I_P}$$

where I_P denotes the time spent by a processor when the processor is idling and waiting for work, combined across all processors. The `pplot` tool expects for the value of I_P to be reported by the benchmark program to `stdout` in the form: `total_idle_time` t, where t denotes the total idle time in seconds. This curve represents the speedup curve that we might expect to see if we consider only the performance of the parallel program on a single processor and the amount of idle time as the number of processors increase.

The third synthetic curve is the ***inflation-specific curve***. This curve is defined by the function

$$\text{inflation}(P) = \frac{P \cdot T_S}{T_1 + F_P}$$

where F_P denotes the "work inflation". The work inflation of a parallel computation is the amount of time by which the computation is slowed down due to effects that are not related to processor utilization. The most significant of these effects are the increased costs relating to memory accesses in a parallel computation where multiple processors are sharing the same memory controllers.

Because the work inflation term F_P is not readily measured in a direct fashion, we derive the inflation-specific speedup from $(P \cdot T_S)/(P \cdot T_P - I_P)$.

```
prun speedup -baseline "examples/others/speedup.sh -algo foo" -baseline-runs 1 -parallel "ex
pplot speedup --factored
```

The factored speedup mode can optionally generate one more curve: the ***elision-specific curve***. This curve is calculated by

$$\text{elision}(P) = \frac{P \cdot T_S}{T_E}$$

where T_E denotes the running time of the "sequential elision" of the parallel benchmark. The sequential elision is the program that is derived by substituting all parallel function calls in the program by sequential function calls. What remains in the elision program is just the parallel algorithm, minus the overheads imposed by parallel scheduling.

```
prun speedup -baseline "examples/others/speedup.sh -algo foo" -baseline-runs 1 -elision "exa
pplot speedup --factored
```