Parallel Algorithm Scheduling Library (PASL) Users' Guide

Deepsea

23 January 2015

Getting started

Table 1: PASL directory structure.

Folder	Contents
`sequtil`	command-line interface, printing interface, timers, control operators, container data structures
`parutil`	locks, barriers, concurrent containers, worker threads, worker-local storage, platform-specific configuration mining
`sched`	threading primitives, DAG primitives, thread-scheduling algorithms, granularity control
`example`	example programs
`test`	regression tests
`tools`	log visualization, shared PASL makefiles, ported code from Problem Based Benchmark Suite, custom malloc count, script to enable/disable hyperthreading, matrix market file IO, export scripts
`chunkedseq`	chunked-sequence library
`graph`	graph-processing library

Currently, PASL supports only x86-64 machines. Furthermore, full support is available only for recent versions of Linux. Limited support is available for Mac OS.

Package dependencies

Table 2: Summary of software dependencies.

Package	Version	Details
gcc	>= 4.9.0	Recent gcc is required because PASL makes heavy use of recent features of C++ / C++1x, such as lambda expressions and higher-order templates.
php	>= 5.3.10	PHP is currently used by the build system of PASL. In specific, the Makefiles used by PASL make calls to PHP programs for purposes of dependency analysis. As such, PHP is not required in situations where alternative build systems, such as XCode, are used.
ocaml	>= 4.00	Ocaml is required by the log-file visualization tool, but currently nothing else.

Installing dependencies on Ubuntu

```
$ sudo add-apt-repository ppa:ubuntu-toolchain-r/test
$ sudo apt-get update
$ sudo apt-get install g++-4.9 ocaml php5-cli
```

Building programs

Linux

Let us start by building the fibonacci example. First, change to the examples directory.

```
$ cd pasl/example
```

Then, build the binary for the source file, which is named `fib.cpp`.

```
$ make fib.opt -j
```

Now we can run the program. In this example, we are computing `fib(45)` using eight cores.

```
$ ./fib.opt -n 45 -proc 8
```

In general, you can add a new build target, say `foo.cpp`, by doing the following.

1. Create the file `foo.cpp` in the folder `example`.
2. Add to `foo.cpp` the appropriate initialization code. For an example, see `fib.cpp`.
3. In the Makefile in the same folder, add a reference to the new file.

   ```
   PROGRAMS=\
           fib.cpp \
           hull.cpp \
           bhut.cpp \
           ...
           foo.cpp
   ```

4. Now you should be able to build and run the new program just like we did with fibonacci.

Mac OS (XCode)

Currently, building from the command line is not yet supported. However, it is possible to build from an XCode project.

1. Create a new project.
2. Create a new target binary.
3. In the "build phases" configuration page of the new target binary, add to the "Compile Sources" box the contents of the following directories.

- `sequtil`
- `parutil`
- `sched` (except for `pasl.S`; this file should not be imported)

4. Add to the `CFLAGS` of the target binary the following flags.

- `-DTARGET_MAC_OS`
- `-D_XOPEN_SOURCE`

5. Add to your project the remaining source files that you want to compile. Remember to add one file which defines the `main()` routine. For example, you could add `examples/fib.cpp` if you want to run the fibonacci program.
6. Click the "play" button to compile and run the program.

Scheduling

Table 3: Command-line interface for scheduling algorithms.

Option	Description
`-kappa` t	the size targeted for sequentialized subtasks, expressed in microseconds (defaultly `500`)
`-delta` t	the targeted delay between every two calls to the `communicate` function (defaultly `kappa`/2, which has the effect of scheduling one call every `kappa` in most cases)

Granularity control

Using profiling data

PASL's granularity-control mechanism gathers profiling data at run time. By default, this profiling data is lost once the program terminates, in which case, the next time the program runs the data must be gathered yet again. Because the profiling can be costly, we may want to save and later reuse the profiling data across successive program runs. Let us see by example how to use this feature. In this experiment, we do an initial run which writes to the filesystem the profiling data and one run which reads the data from the filesystem.

```
fib.opt -n 30 -proc 30 --write_csts
fib.opt -n 30 -proc 30 --read_csts
```

By default, when profiling data is read from the filesystem, PASL gathers no new profiling data. If we wish to override this behavior, we can do the following:

```
fib.opt -n 30 -proc 30 --read_csts --force_controller_report
```

Table 4: Command-line interface for granularity-control profiling data.

Option	Description
`--write_csts`	write values of estimator constants to a file (defaultly, path is `fib.opt.cst`)
`--read_csts`	read values of estimator constants from a file (defaultly, path is `fib.opt.cst`)

Option	Description
`-write_csts_in` *p*	write values of estimator constants to a given file
`-read_csts_in` *p*	read values of estimator constants from a given file
`--force_controller_report`	force the controller to report measured runs

Using logging data

PASL can be configured to collect and report events that are relevant to granularity control. To obtain this logging data, first build a logging binary, for example, `fib.log`. Then, run the program with the following command line options.

```
fib.log -n 30 -proc 30 --log_estims --log_text
```

After the run completes, a file named `LOG` should appear in the same folder. In this file are three types of events. The first type establishes an association between the name of the estimator, in this case `fib`, and the unique identifier for the name, in this case `0x652440`.

```
82.000000     -1   estim_name      0x652440  fib
```

The second type of event records the results of a prediction operation that is performed by the granularity controller. The value in the fifth position, namely `12500000`, is the value that is returned by the complexity function for this particular prediction. The value in the sixth position, namely `0.026923` is the value of the estimator data structure. The value in the last position, namely `336537.500000` is the predicted running time for associated computation that is calculated by the expression `12500000.0 * 0.026923`.

```
1907.000000   27   estim_predict   0x652440  12500000  0.026923   336537.500000
```

The third type of event records the result of a measured run. The value in the fifth position, namely `61203`, is the value that is returned by the complexity function. The value in the sixth position is the current value of the estimator data structure. The value in the last position is the actual running time that was observed by the measured run.

```
2662.000000   1    estim_report    0x652440  6103      0.050988   311.182642
```

Table 5: Command-line interface for granularity-control logging.

Option	Description
`--log_estims`	log predictions and reports

The log visualizer: `pview`

We implemented `pview` to help diagnose performance issues relating to parallelism. The tool reads logging data that is generated by the PASL runtime and renders from the data a breakdown of how the scheduler spends its time during the run of the program. The tool and corresponding documentation can be found in the folder named `tools/pview/`.