Par4All User Guide
p4a

—

HPC Project

Mehdi AMINI Béatrice CREUSILLET Onil GOUBIER Serge GUELTON
Ronan KERYELL Janice ONANIAN-MCMAHON Grégoire PÉAN

April 23, 2012

Contents

1 Introduction **2**

2 p4a architecture **2**

3 Automatic parallelization using p4a **3**
3.1 Automatic parallelization with OpenMP generation 3
3.2 Automatic parallelization with CUDA generation 3
3.3 Automatic parallelization with OPENMP emulation of GPU 4
3.4 Automatic parallelization with OpenCL generation 4
3.5 Automatic task generation for the SCMP architecture 4
3.6 More on PAR4ALL automatic task generation 5

4 p4a full option list **5**
4.1 Options 5
4.2 Project (aka workspace) options 5
4.3 PIPS processing options 6
4.4 Preprocessing options 7
4.5 Back-end compilation options 8
4.6 Back-end linking options 9
4.7 CMake file generation options 9
4.8 Output options 9
4.9 General options 10

5 p4a extension for advanced users **10**

6 Tips, troubleshooting, debugging and auxiliaries tools **12**
6.1 Coding rules 12
6.2 P4A Accel runtime tweaking for CUDA 12
6.3 P4A Accel runtime for OpenCL 13
6.4 Auxiliary tools 14
6.4.1 Selective C Pre Processor 14

1 Introduction

PAR4ALL is a project aimed at easing code generation for parallel architectures from sequential source codes written in C or Fortran with almost no manual code modification required. PAR4ALL is based on multiple components, including the PIPS source-to-source compiler framework, and is developed by HPC Project, Mines ParisTech and Institut Télécom. PAR4ALL is open source to take advantage of community participation and to avoid capturing users in a proprietary short-term solution. Specialized target developments and professional support are also available on demand.

The main goal of a source-to-source compiler is to be naturally independent of the target architecture details and to take advantage of the best back-end tools, such as highly optimized vendor compilers for a given processor or platform, open-source compilers and tools, high-level hardware synthesizers, CUDA or OPENCL compilers for GPU. At the same time, some architectural aspects can be expressed or generated in special source constructs to capture architectural details when needed (SIMD or accelerators intrinsics). The source-to-source aspect makes PAR4ALL *de facto* interoperable with other tools as front-end or back-end compilers to build complete tool chains.

`p4a` is the basic script interface for users who are not interested in PIPS details but wish to produce parallel code from user sources.

This script can take C or Fortran source files and generate OPENMP, CUDA or OPENCL output to run on shared memory multicore processors or GPU, respectively.

The output is created in files with the same name as the original ones, with a *.p4a* extension added before the original extension. The generated files are extracted from a temporary *x*`.database` directory that can be kept for later inspection (standard practice for PIPS usage).

This script can also be used to call a back-end compiler such as `gcc`, `icc` or `nvcc` to generate a binary executable. Compiler and preprocessor options can be passed directly to the back-end compiler via the script.

A CMake build infrastructure can be automatically generated to ease compilation by the back-end.

The PAR4ALL documentation is available from `http://www.par4all.org` and more specifically, this document can be found in PDF from `http://download.par4all.org/doc/par4all_user_guide/par4all_user_guide.pdf` and in HTML from `http://download.par4all.org/doc/par4all_user_guide/par4all_user_guide.htdoc`.

Since PAR4ALL is a large project in progress with continuous updates, users should refer to the current release notes and the general documentation on `http://www.par4all.org` for updates on the current status and limitations of the tools.

This project is funded through various research projects : French ANR FREIA, European ARTEMIS SCALOPES, French Images and Network TransMedi@, French System@tic OpenGPU, French ANR MediaGPU, European ARTEMIS SMECY.

2 p4a architecture

The global architecture of PAR4ALL is given on figure 1.

From the high-level user point of view, the sources follow this journey:

- the source files pass through the preprocessor of PIPS. C source files also pass through `p4a_recover_includes` to instrument the `#include` processing for later recovering;
- the preprocessed files pass through a splitter that creates one file per function and a compilation unit file that keeps track of all the file-global declarations;
- each function file or compilation-unit file can be parsed on demand according to the PyPS script;
- a predefined PyPS program applies many different PIPS phases on the code and regenerate the transformed sources;

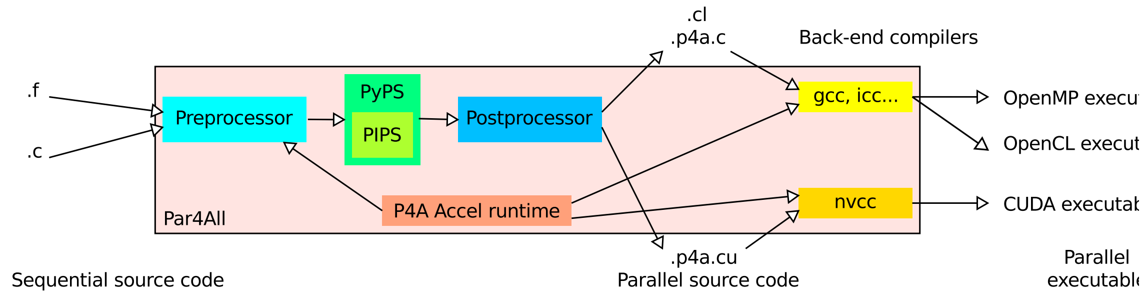

Figure 1: Intestinal view of `p4a`.

- `p4a_recover_includes` is applied again as a post-processor to recover most of the `#include` work;
- the sources are postprocessed by `p4a_post_processor.py` to cope with some of the special syntax (CUDA, OpenCL...) that can not be directly represented in PIPS;
- the generated final files are copied to the final directory;
- if requested, the final target back-end compilers are called to produce the parallelized executable program.

3 Automatic parallelization using p4a

3.1 Automatic parallelization with OpenMP generation

To generate OpenMP code from a C or Fortran program, use `p4a` with option `--openmp`, for example for a Fortran program:

```
p4a --openmp example.f
```

which produces output in the file `example.p4a.f`. Since this is the default behavior, the `--openmp` option can be omitted. The compilation process automatically data-parallel *do*-loops into OpenMP parallel loops with the correct pragmas and with the privatization of scalar variables.

3.2 Automatic parallelization with CUDA generation

To generate from a C program source a CUDA program that is also compiled into an executable, use:

```
p4a --cuda example.c -o cuda_example
```

which produces an `example.p4a.cu` CUDA program source and an `cuda_example` executable that will run on a GPU. The GPU accelerator support relies on a small Par4All Accel interface that connects to the CUDA infrastructure. Data-parallel loops are automatically transformed into CUDA kernels that execute on the GPU. *Ad hoc* communications between the host memory and the GPU memory are generated to enable kernel execution.

The native compilation scheme is to generate allocation on the GPU for each parallel loop nest, transfer to the GPU the data needed for the computation, launch the kernel and copy back the results from the GPU. For iterative algorithms for example, some data can often remain on the GPU between kernel calls because they are not touched by the host. For this, there is an optional optimization that can be used with the `--com-optimization` option: a static interprocedural dataflow analysis keep track of data movement and allocation to remove redundant communications

and allocation, leading to great performance improvements. This options supposes that you make use of static and/or C99 VLA array, it would be broken by `malloc`'ed arrays in parallel section. For example:

```
p4a --cuda --com-optimization optimized_example.c -o cuda_optimized_example
```

Par4All can also deal with C99 code sources. Indeed `nvcc` doesn't support the following C99 syntax:

```
f_type my_function(size_t n, size_t m, float array[n][m]) {
            ...
}
```

Using `--c99` option, `p4a` will automatically generate the CUDA code in new C89 files (with no VLA but pointers with linearized accesses instead) that will be compiled by `nvcc`. A simple call to each kernel will be inserted into the original files `*.p4a.c` that can be compiled with your usual C99 compiler, and linked to the object files compiled from `*.cu`, to produce an executable. For example:

```
p4a --cuda --c99 c99_example.c -o c99_example
```

generates the executable file `c99_example`.

3.3 Automatic parallelization with OpenMP emulation of gpu

To generate an OpenMP emulation executable of GPU-like accelerated code (for debugging or if a GPU is not available), use:

```
p4a --accel --openmp example.c -o accel_openmp_example
```

which produces a `accel_openmp_example` binary executable file with its corresponding `example.p4a.c` program source. This OpenMP emulation of CUDA with memory transfer may be helpful for debugging since there is no longer emulation mode in recent versions of CUDA.

3.4 Automatic parallelization with OpenCL generation

To generate from a C program source a OpenCL program that is also compiled into an executable, use:

```
p4a --opencl example.c -o opencl_example
```

which produces a host program `example.p4a.c`, a OpenCL kernel program source `p4a_wrapper_example.cl` and an `opencl_example` executable. The current version of Par4All generates OpenCL codes that are to be run on a GPU. The GPU accelerator support relies on a small Par4All Accel interface that connects to the OpenCL infrastructure. Data-parallel loops are automatically transformed into OpenCL kernels and saved as *.cl files. These files will be loaded, compiled and executed on the GPU. *Ad hoc* communications between the host memory and the GPU memory are generated to enable kernel execution.

3.5 Automatic task generation for the SCMP architecture

To generate an application for the SCMP architecture from a C program, use:

```
p4a --scmp example.c
```

The tasks must have been previously identified with labels prefixed by `scmp_task_`[1].

The compiler generates two directories:

[1] Advanced users can change this prefix by changing the PIPS `SCALOPES_KERNEL_TASK_PREFIX` property.

- `applis/` contains the automatically generated control task file (`.app` file);
- `applis_processing/project_name/` contains the buffers description header file (`scmp_buffers`), the `*.mips.c` task files and a `Makefile.arp` to compile and build the final application.

Some further post-processing may be required to limit the memory footprint of the generated application. This process has not yet been automated.

3.6 More on Par4All automatic task generation

The advantage of the source-to-source approach is that the generated code can be further improved manually or used as a starting point for other developments. When compiling the Par4All Accel generated code, different preprocessor symbols can be defined according to the expected target:

- `P4A_ACCEL_CUDA` preprocessor symbol, the source is to be compiled as CUDA;
- `P4A_ACCEL_OPENMP` preprocessor symbol, the source is to be compiled as OpenMP or sequential emulation code.
- `P4A_ACCEL_OPENCL` preprocessor symbol, the source is to be compiled as OpenCL code.

Parallelization of some functions can be avoided by using `--exclude-modules=`*`regex`*. This is preferable when the parallel part is not compute-intensive enough to compensate for the overhead of transferring data and launching a kernel. In this case, parallel execution is more expensive than sequential execution. The option `--select-modules=`*`regex`* is complementary to the previous one and can be useful to speedup the process by focusing on specific functions.

There is also an option to exclude some files from the back-end compilation, for example, to use libraries that have been optimized already. To use this option, the program is analyzed by providing stubs definitions in a source file. The stubs definitions are dummy function definitions that have similar global memory effects to the original functions so that PIPS global interprocedural analysis can proceed correctly. For subsequent compilation in the back-end stage, this file is skipped with `--exclude-file`, then the dummy calls are replaced with the real library functions by linking them with `-l` or compiling them with `--extra-file`.

Look at section 4 for a full list of options.

4 p4a full option list

The basic usage is `p4a [`*`options`*`] <`*`source files`*`>`

The following documentation is automatically generated from the `p4a` source code.

`Usage:  p4a.py [options] [files]; run p4a.py --help for options`

4.1 Options

`-h, --help`: show this help message and exit

4.2 Project (aka workspace) options

`-p NAME, --project-name=NAME, --project=NAME`: Name for the project (and for the PIPS workspace database used to work on the analyzed program). If you do not specify the project, a random name will be used.

`-k, --keep-database`: Keep database directory after processing.

`-r, --remove-first`: Remove existing database directory before processing.

4.3 PIPS processing options

`--no-pointer-aliasing`: Assume there is no aliasing in input program, thus enabling more optimizations. This option currently only controls PIPS internal processing and is not taken into account for back-end compilation.

`--pointer-analysis`: Activates a pointer analysis phase on the code (experimental !).

`-A, --accel`: Parallelize for heterogeneous accelerators by using the Par4All Accel run-time that allows executing code for various hardware accelerators such as GPU or even OpenMP emulation.

`-C, --cuda`: Enable CUDA generation. Implies --accel.

`--opencl`: Enable OpenCL generation. Implies --accel.

`-O, --openmp`: Parallelize with OpenMP output. If combined with the --accel option, generate Par4All Accel run-time calls and memory transfers with OpenMP implementation instead of native shared-memory OpenMP output. If --cuda is not specified, this option is set by default.

`--scmp`: Parallelize with SCMP output.

`--com-optimization`: Enable memory transfert optimizations, implies --accel. This is an experimental option, use with caution ! Currently design to work on plain array : you shouldn't use it on a code with pointer aliasing.

`--c99`: This option is useful when generating some CUDA code from C99 sources. Indeed nvcc doesn't support the following C99 syntax : foo (int n, int a[n]), then if the --c99 option is enabled, p4a will automatically generates the CUDA code in new C89 files (with no VLA but pointers with linearized accesses instead) that will be compiled by nvcc. A simple call to each kernel will be inserted into the original file that can be compiled with your usual C99 compiler.

`-S, --simple`: This cancels --openmp, --cuda, --scmp, or --opencl and does a simple transformation (no parallelization): simply parse the code and regenerate it. Useful to test preprocessor and PIPS intestinal transit.

`-F, --fine-grain`: Use a fine-grain parallelization algorithm instead of a coarse-grain one.

`--atomic`: Use atomic operations for parallelizing reductions on GPU (experimental).

`--kernel-unroll=KERNEL_UNROLL`: Unroll factor for loops inside kernels.

`--pocc`: Use PoCC to optimize loop nest (experimental). PoCC has to be already installed on your system. See pocc.sf.net, the Polyhedral Compiler Collection.

`--pocc-options=POCC_OPTIONS`: Options to pass to PoCC.

`--cuda-cc=CUDA_CC`: Compute capabilities of CUDA target (default is 2.0). For example if you have a message like 'P4A CUDA kernel execution failed : invalid device function' at execution time, the generated code may be incompatible with your GPU and you have to use this option to select the good architecture version.

`--select-modules=REGEXP`: Process only the modules (functions and subroutines) whith names matching the regular expression. For example 'ŝaxpy$—dgemm' will keep only functions or procedures which name is exactly saxpy or contains "dgemm". For more information about regular expressions, look at the section 're' of the Python library reference for example. In Fortran, the regex should match uppercase names. Be careful to escape special characters from the shell. Simple quotes are a good way to go for it.

--exclude-modules=REGEXP: Exclude the modules (functions and subroutines) with names matching the regular expression from the parallelization. For example '(?i)m̂y_runtime' will skip all the functions or subroutines which names begin with 'my_runtime' in uppercase or lowercase. Have a look to the regular expression documentation for more details.

-N, --no-process: Bypass all PIPS processing (no parallelizing...) and voids all processing options. The given files are just passed to the back-end compiler. This is merely useful for testing compilation and linking options.

-P NAME=VALUE, --property=NAME=VALUE: Define a property for PIPS. Several properties are defined by default (see p4a_process.py). There are many properties in PIPS that can be used to modify its behaviour. Have a look to the 'pipsmake-rc' documentation for their descriptions.

--apply-before-parallelization=PIPS_PHASE1,PIPS_PHASE2,..., --abp=PIPS_PHASE1,PIPS_PHASE2,...: Add PIPS phases to be applied before parallelization.

--apply-after-parallelization=PIPS_PHASE1,PIPS_PHASE2,..., --aap=PIPS_PHASE1,PIPS_PHASE2,...: Add PIPS phases to be applied after parallelization.

--apply-kernel-gpuify=PIPS_PHASE1,PIPS_PHASE2,..., --akg=PIPS_PHASE1,PIPS_PHASE2,...: Add PIPS phases to be applied to kernels inside the gpuify execution, for the gpu code generation

--apply-kernel-launcher-gpuify=PIPS_PHASE1,PIPS_PHASE2,..., --aklg=PIPS_PHASE1,PIPS_PHASE2,...: Add PIPS phases to be applied to kernel launchers inside gpuify, for the gpu code generation

--apply-wrapper-gpuify=PIPS_PHASE1,PIPS_PHASE2,..., --awg=PIPS_PHASE1,PIPS_PHASE2,...: Add PIPS phases to be applied to wrappers inside gpuify, for the gpu code generation

--apply-after-gpuify=PIPS_PHASE1,PIPS_PHASE2,..., --aag=PIPS_PHASE1,PIPS_PHASE2,...: Add PIPS phases to be applied after the gpuify execution, for the gpu code generation

--apply-before-ompify=PIPS_PHASE1,PIPS_PHASE2,..., --abo=PIPS_PHASE1,PIPS_PHASE2,...: Add PIPS phases to be applied before the ompify execution, for the OpenMP code generation

--apply-after-ompify=PIPS_PHASE1,PIPS_PHASE2,..., --aao=PIPS_PHASE1,PIPS_PHASE2,...: Add PIPS phases to be applied after the ompify execution, for the OpenMP code generation

--stubs-broker=broker1,broker2,...: Add a stubs broker, it's a resolver python class that is able to provide a source file based on a fonction name

4.4 Preprocessing options

--cpp=PREPROCESSOR: C preprocessor to use (defaults to gcc -E).

-I DIR: Add an include search directory. Same as the compiler -I option. Several are allowed.

-D NAME[=VALUE]: Add a preprocessor define. Same as passing the preprocessor a -D option. Several are allowed.

-U NAME: Remove a preprocessor define. Same as passing the preprocessor a -U option. Several are allowed.

--cpp-flags=FLAGS: Add additional flags for the C preprocessor. Several are allowed.

--skip-recover-includes: By default, try to recover standard #include. To skip this phase, use this option.

--native-recover-includes: Use the PyPS default #include recovery method that is less correct if you use complex CPP syntax but is faster. Since it does not rely on the preprocessor that normalized all the included file names, it may be easier to use Par4All in harder context, such as a virtual machine on Windows... By default, use the more complex method of Par4All.

4.5 Back-end compilation options

--fftw3: Use fftw3 library. Do not add -lfftw3 or -lfftw3f, p4a will add it automatically if needed. It's an experimental option, use with care !

-o FILE, --output-file=FILE, --output=FILE: This enables automatic compilation of binaries. There can be several of them. Output files can be .o, .so, .a files or have no extension in which case an executable will be built.

-X FILE, --exclude-file=FILE, --exclude=FILE: Exclude a source file from the back-end compilation. Several are allowed. This is helpful if you need to pass a stub file with dummy function definitions (FFT, linear algebra library...) so that PIPS knows how to parallelize something around them, but do not want this file to end up being compiled since it is not a real implementation. Then use the --extra-file or -l option to give the real implementation to the back-end.

-x FILE, --extra-file=FILE, --extra=FILE: Include an additional source file when compiling with the back-end compiler. Several are allowed. They will not be processed through PIPS.

--cc=COMPILER: C compiler to use (defaults to gcc).

--cxx=COMPILER: C++ compiler to use (defaults to g++).

--nvcc=COMPILER: NVCC compiler to use (defaults to nvcc). Note that the NVCC compiler is used only to transform .cu files into .cpp files, but not compiling the final binary.

--fortran=COMPILER: Fortran compiler to use (defaults to gfortran).

--ar=ARCHIVER: Archiver to use (defaults to ar).

--icc: Automatically switch to Intel's icc/xild/xiar for --cc/--ld/--ar.

-g, --debug: Add debug flags (-g compiler flag). Have a look to the --no-fast if you want to remove any optimization that would blur the debug.

--no-fast, --not-fast: Do not add optimized compilation flags automatically.

--no-openmp, --nomp: Do not add openmp compilation flags automatically. This option allows to get a sequential version of the openmp code produced by p4a. When icc is used this enable the option openmp-stubs.

--no-default-flags: Do not add some C flags such as -fPIC, -g, etc. automatically.

--c-flags=FLAGS: Specify flags to pass to the C compiler. Several are allowed. Note that --cpp-flags will be automatically prepended to the actual flags passed to the compiler.

--cxx-flags=FLAGS: Specify flags to pass to the C++ compiler. Several are allowed. By default, C flags (--c-flags) are also passed to the C++ compiler.

--nvcc-flags=FLAGS: Specify flags to pass to the NVCC compiler. Several are allowed. Note that --cpp-flags will be automatically prepended to the actual flags passed to the compiler.

--fortran-flags=FLAGS: Specify flags to pass to the Fortran compiler. Several are allowed. Note that --cpp-flags will be automatically prepended to the actual flags passed to the compiler.

-m 32|64, --arch=32|64: Specify compilation target architecture (defaults to current host architecture).

-K, --keep-build-dir: Do not remove build directory after compilation. If an error occurs, it will not be removed anyways, for further inspection.

4.6 Back-end linking options

--ld=LINKER: Linker to use (defaults to ld).

-L DIR: Add a library search directory. Same as the linker -L option. Several are allowed.

-l LIB: Specify an input library to link against. Same as the linker -l option. Several are allowed.

--ld-flags=FLAGS: Specify additional flags to pass to the linker. Several are allowed.

--extra-obj=FILE: Add an additional object file for linking. Several are allowed.

4.7 CMake file generation options

--cmake: If output files are specified (with -o), setting this flag will have p4a produce a CMakeLists.txt file in current directory (or in any other directory specified by --cmake-dir). This CMakeLists.txt file will be suitable for building the project with CMake. NB: setting --cmake alone will NOT build the project.

--cmake-flags=FLAGS: Specify additional flags to pass to CMake. Several are allowed.

--cmake-dir=DIR: Output/lookup the CMakeLists.txt file in this directory instead of the current working directory.

--cmake-gen: If output files are specified (with -o), setting this flag will make p4a try to locate a CMakeLists.txt file in current directory (or in any other directory specified by --cmake-dir), and generate Makefiles in a specific directory (--cmake-gen-dir).

--cmake-build: Implies --cmake-gen. Generate Makefiles from the found CMakeLists.txt and run 'make' on them.

4.8 Output options

--output-dir=DIR, --od=DIR: By default the sources files generated by p4a are located in the folder of the input files. This option allows to generate all the sources output files in the specified directory. When using this option, you can't have files with the same name processed by p4a. An absolute path must be provided, if the directory does not exist p4a will create it. If the output directory is not specified either the output suffix or the output prefix must be set, if not, a suffix will be automatically added to avoid source files destruction.

--output-suffix=SUF, --os=SUF: Use a suffix to easily recognize files processed by p4a. Default to "p4a"

--output-prefix=PRE, --op=PRE: Use a prefix to easily recognize files processed by p4a. Default to ""

4.9 General options

`-v, --verbose`: Run in verbose mode: each -v increases verbosity mode and display more information, -vvv will display most information.

`--log`: Enable logging in current directory.

`--report=YOUR-EMAIL-ADDRESS`: Send a report email to the Par4All support email address in case of error. This implies --log (it will log to a distinct file every time). The report will contain the full log for the failed command, as well as the runtime environment of the script like arguments and environment variables.

`--report-files`: If --report is specified, and if there were files specified as arguments to the script, they will be attached to the generated report email. WARNING: This might be a privacy/legal concern for your organization, so please check twice you are allowed and willing to do so. The Par4All team cannot be held responsible for a misuse/unintended specification of the --report-files option.

`--report-dont-send`: If --report is specified, generate an .eml file with the email which would have been send to the Par4All team, but do not actually send it.

`-z, --plain, --no-color, --no-fancy`: Disable coloring of terminal output and disable all fancy tickers and spinners and this kind of eye-candy things :-)

`--no-spawn`: Do not spawn a child process to run processing (this child process is normally used to post-process the PIPS output and reporting simpler error message for example).

`--execute=PYTHON-CODE`: Execute the given Python code in order to change the behaviour of this script. It is useful to extend dynamically Par4All. The execution is done at the end of the common option processing

`-V, --script-version, --version`: Display script version and exit.

5 p4a extension for advanced users

The main work flow in `p4a` is defined in `p4a::main` from `p4a.py`:

- load PyPS;
- normalize user source file names;
- call `p4a_process` to do the code transformation with PyPS;
- generate `Cmake` configuration files if asked;
- build the final executable with a `p4a_builder`.

Most of the interesting work is done inside `p4a_process::process`, in another process (if not using the `--no-spawn` option) so that `p4a` can do some PIPS output message post-processing. The interesting steps are:

- `p4a_process::processor::parallelize`;
- `p4a_process::processor::gpuify` generate GPU-like code if the `--accel` option is set;
- `p4a_process::processor::ompify` if OpenMP output is requested;
- `p4a_process::processor::save` regenerate the source files from PIPS internal representation with `unsplit()`, call the `#include` recovering, call the `p4a_post_processor.py` if GPU code generation asked. The generated files are saved to their destinations with the correct file extensions.

The advantage of the p4a script is that it is a simple tool that behaves well on simple programs but also allows the advanced user to go further within a practical infrastructure. To specialize p4a to the needs of particular users, it is possible to execute arbitrary Python code at the start of p4a execution with the --exec option.

The given code is executed in the main function of the p4a script just after option parsing but just before option handling and the call to p4a::main.

The p4a.execute_some_python_code_in_process Python variable can be used to inject code at the beginning of p4a_process::process. The code can access local variables of the script but not change them and can both access and change global symbols, such as global p4a methods. p4a_process::default_properties contains the PIPS properties used in p4a.

Below are some examples of p4a customization that change the behaviour of PyPS inside p4a by importing some modules for more practical modifications.

First an example to change only one method:

```
"""
This is an example of code injection in p4a_process of p4a with a direct
method modification. Can be seen as a poor man aspect programmin

This is to be used as:

p4a --execute p4a.execute_some_python_code_in_process='"import p4a_process_inject"' ...

and you should adapt the PYTHONPATH to cope with the location where is
this p4a_process_inject.py

Ronan.Keryell@hpc-project.com
"""

# Import the code we want to tweak:
import p4a_process

# Save the old definition of the p4a_process.p4a_processor.parallelize
# method for a later invocation:
old_p4a_processor_parallelize = p4a_process.p4a_processor.parallelize

# Define a new method instead:
def new_p4a_processor_parallelize(self, **args):

    # Apply for example a partial evaluation on all the functions of the
    # programm
    self.workspace.all_functions.partial_eval()

    # Go on by calling the original method with the same parameters:
    old_p4a_processor_parallelize(self, **args)

# Overide the old definition of p4a_process.p4a_processor.parallelize by
# our new one:
p4a_process.p4a_processor.parallelize = new_p4a_processor_parallelize
```

Another example showing how to change a whole class as well:

```
"""
This is an example of code injection in p4a_process of p4a with some class
inheritance.

This is to be used as:

p4a --execute p4a.execute_some_python_code_in_process='"import p4a_process_inject_class"'

and you should adapt the PYTHONPATH to cope with the location where is
this p4a_process_inject.py

Ronan.Keryell@hpc-project.com
"""

# Import the code we want to tweak:
import p4a_process


class my_p4a_processor(p4a_process.p4a_processor):
    """Change the PyPS transit of p4a.

    Since it is done in the p4a_process.p4a_processor, we inherit of this
    class
    """
```

```python
    def __init__(self, **args):
        "Change the init method just to warn about this new version"

        print "This is now done by class", self.__class__
        # Just call the normal constructors with all the named parameters:
        super(my_p4a_processor, self).__init__(**args)


    # Define a new method instead:
    def parallelize(self, **args):
        "Parallelize the code"

        # Apply for example a partial evaluation on all the functions of the
        # programm at the beginning of the parallelization
        self.workspace.all_functions.partial_eval()

        # Go on by calling the original method with the same parameters:
        # Compatible super() invocation with pre-3 Python
        super(my_p4a_processor, self).parallelize(**args)


# Overide the old class definition by our new one:
p4a_process.p4a_processor = my_p4a_processor
```

6 Tips, troubleshooting, debugging and auxiliaries tools

6.1 Coding rules

It is important to write programs that are compliant with the current state of PAR4ALL and PIPS and that maximize the precision of analyses and thus help the tool in generating efficient code. To get more information on PAR4ALL coding rules please refer to `http://download.par4all.org/doc/p4a_coding_rules`.

6.2 P4A Accel runtime tweaking for CUDA

To ease code generation and portability, PAR4ALL does not directly generate direct CUDA code and instead generates calls to functions and macrofunctions defined in the P4A Accel runtime.

There are many parameters that can be changed to better suit a given application on a given target.

The `--cuda-cc` option defines the compute capability of your target GPU. The default value is 2.0, and allowed values are 1.0, 1.1, 1.2 and 1.3.

In the case of CUDA generation, you can pass flags to the `nvcc` back-end compiler with the `--nvcc-flags=...` option.

For example:

- to debug the output with `cuda-gdb`, use `--nvcc-flags="-g -G"`;
- to optimize the CUDA code use `--nvcc-flags=-O3`;
- to generate code to Fermi board use `--nvcc-flags="-gencode arch=compute_20,code=sm_20"`;
- to generate code to both Tesla & Fermi board use `--nvcc-flags="-gencode arch=compute_10,code=sm_10 -gencode arch=compute_20,code=sm_20"`.

The P4A Accel runtime itself can be debugged by defining `P4A_DEBUG` environment variable to 1 (or more) to output debugging messages around all the CUDA kernels invocations, such as:

```
$ P4A_DEBUG=1 ./run_cuda
 P4A: Setting debug level to 1
 P4A: Invoking kernel p4a_wrapper_main (64x256x1 ; 32x81) with args ((int) i, (int) j, ..... )
 P4A: Invoking kernel p4a_wrapper_main_1 (64x256x1 ; 32x81) with args ((int) i, (int) j, ..... )

$ P4A_DEBUG=2 ./run_cuda
 P4A: Setting debug level to 2
 P4A: Calling 2D kernel "p4a_wrapper_main" of size (2048x2048)
```

```
 P4A: Invoking kernel p4a_wrapper_main (64x256x1 ; 32x81) with args ((int) i, (int) j, ..... )
 P4A: Calling 2D kernel "p4a_wrapper_main_1" of size (2048x2048)
 P4A: Invoking kernel p4a_wrapper_main_1 (64x256x1 ; 32x81) with args ((int) i, (int) j, ..... )

$ P4A_DEBUG=3 ./run_cuda
 P4A: Setting debug level to 3
 P4A: Calling 2D kernel "p4a_wrapper_main" of size (2048x2048)
 P4A: line 91 of function p4a_launcher_main in file "2mm.opt.cu":
 P4A: Invoking kernel p4a_wrapper_main (64x256x1 ; 32x81) with args ((int) i, (int) j, ..... )
 P4A: Calling 2D kernel "p4a_wrapper_main_1" of size (2048x2048)
 P4A: line 85 of function p4a_launcher_main_1 in file "2mm.opt.cu":
 P4A: Invoking kernel p4a_wrapper_main_1 (64x256x1 ; 32x81) with args ((int) i, (int) j, ..... )

$ P4A_DEBUG=4 ./run_cuda
 P4A: Setting debug level to 4
 P4A: Calling 2D kernel "p4a_wrapper_main" of size (2048x2048)
 P4A: Creating grid of block descriptor "P4A_grid_descriptor" of size 64x256x1
 P4A: Creating thread block descriptor "P4A_block_descriptor" of size 32x8x1
 P4A: line 91 of function p4a_launcher_main in file "2mm.opt.cu":
 P4A: Invoking kernel p4a_wrapper_main (64x256x1 ; 32x81) with args ((int) i, (int) j, ..... )
 P4A: Calling 2D kernel "p4a_wrapper_main_1" of size (2048x2048)
 P4A: Creating grid of block descriptor "P4A_grid_descriptor" of size 64x256x1
 P4A: Creating thread block descriptor "P4A_block_descriptor" of size 32x8x1
 P4A: line 85 of function p4a_launcher_main_1 in file "2mm.opt.cu":
 P4A: Invoking kernel p4a_wrapper_main_1 (64x256x1 ; 32x81) with args ((int) i, (int) j, ..... )
```

Some time measurements can be displayed by defining `P4A_TIMING` environment variable to 1. The output will then look like the following :

```
P4A_TIMING=1 ./2mm_cuda_opt
 P4A: Enable timing (1)
 P4A: Copied 33554432 bytes of memory from host 0x260b1e0 to accel 0x200100000 : 11.0ms - 3.06GB/s
 P4A: Copied 33554432 bytes of memory from host 0x460b1e0 to accel 0x202100000 : 10.8ms - 3.11GB/s
 P4A: Copied 33554432 bytes of memory from host 0x60b1e0 to accel 0x204100000 : 10.8ms - 3.12GB/s
 P4A: Copied 33554432 bytes of memory from host 0x660b1e0 to accel 0x206100000 : 10.8ms - 3.11GB/s
 P4A: Copied 33554432 bytes of memory from host 0x860b1e0 to accel 0x208100000 : 10.8ms - 3.11GB/s
 P4A: Time for 'p4a_wrapper_main' : 521.280396ms
 P4A: Time for 'p4a_wrapper_main_1' : 521.214355ms
 P4A: Copied 33554432 bytes of memory from accel 0x208100000 to host 0x860b1e0 : 9.8ms - 3.44GB/s
```

A runtime some error message such as:

```
CUTIL CUDA error : P4A CUDA kernel execution failed :
too many resources requested for launch
```

signifies that there are not enough registers to run all the requested threads in a block. To remove this error, relaunch `p4a` with the `P4A_MAX_TPB` environment variable to limit the number of threads per block under the default value of 256.

For efficiency reasons on NVIDIA Fermi GPU, the threads are allocated as much as possible as a square. For instance with the default 256 threads per block, it will end up with 16 along the X dimension and 16 along the Y dimension. The threads per block allocation take into account unbalanced loop nests. In the current revision, parallel loop nests are limited to the 3 outer parallel dimensions (2 with compute capabilities lower than 2.0) to cope more easily with CUDAGPU hardware limitation. The inner selected parallel loop is mapped onto the X GPU dimension.

For more tweaking, look at the Par4All Accel runtime source on `http://download.par4all.org/doc/Par4All_Accel_runtime/graph` or directly in the GIT repository.

6.3 P4A Accel runtime for OpenCL

Like Par4All generation for CUDA, Par4All does not directly generate direct OpenCL code and instead generates calls to functions and macrofunctions defined in the P4A Accel runtime.

For more information on P4A Accel runtime for OpenCL, have a look at Par4All Accel runtime source on `http://download.par4all.org/doc/Par4All_Accel_runtime/graph` or directly in the GIT repository.

The P4A Accel runtime itself can also be debugged by defining `P4A_DEBUG` environment variable to 1 (or more) to output debugging messages around all the OpenCL kernels invocations, such as:

```
$ P4A_DEBUG=3 ./run_opencl
 P4A: P4A_DEBUG environment variable is set : '3'
 P4A: Setting debug level to 3
 P4A: Available OpenCL platforms :
 P4A:  ** platform 0 **
    - CL_PLATFORM_NAME: NVIDIA CUDA
    - CL_PLATFORM_VENDOR: NVIDIA Corporation
    - CL_PLATFORM_VERSION: OpenCL 1.1 CUDA 4.1.1
    - CL_PLATFORM_PROFILE: FULL_PROFILE
    - CL_PLATFORM_EXTENSIONS: cl_khr_byte_addressable_store cl_khr_icd cl_khr_gl_sharing
        cl_nv_compiler_options cl_nv_device_attribute_query cl_nv_pragma_unroll
 P4A: OpenCL platform chosen : 0
 P4A: Available OpenCL devices for platform    - : NVIDIA CUDA
 P4A:  ** device 0 **
    - CL_DEVICE_NAME: Tesla C2050
    - CL_DEVICE_TYPE :GPU
    - CL_DEVICE_EXTENSIONS: cl_khr_byte_addressable_store cl_khr_icd cl_khr_gl_sharing
        cl_nv_compiler_options cl_nv_device_attribute_query cl_nv_pragma_unroll
        cl_khr_global_int32_base_atomics cl_khr_global_int32_extended_atomics
        cl_khr_local_int32_base_atomics cl_khr_local_int32_extended_atomics cl_khr_fp64
    - CL_DEVICE_PROFILE: FULL_PROFILE
    - CL_DEVICE_VENDOR: NVIDIA Corporation
    - CL_DEVICE_VERSION: OpenCL 1.1 CUDA
    - CL_DRIVER_VERSION: 285.05.05
 P4A: OpenCL device chosen : 0
 P4A: The kernel p4a_wrapper_main is loaded for the first time
 P4A: Program and Kernel creation from ./p4a_wrapper_main.cl
 P4A: Calling 2D kernel "p4a_wrapper_main" of size (13x17)
 P4A: line 27 of function p4a_launcher_main in file "/usr/home/me/where/is/myfile/example.p4a.c":
 P4A: Invoking kernel 'p4a_wrapper_main(i, j, a)' : (13x17x1 , NULL)
```

6.4 Auxiliary tools

6.4.1 Selective C Pre Processor

The `p4a_scpp` tool can be used to pre-process a specific `#include` directive while leaving the others untouched. See `p4a_scpp --help` for usage samples.