Par4All primer with tpips

Amaury Darsch

HPC Project

October 11, 2010

Contents

1 General concepts

2 First contact

3 Working with OpenMP

4 Special operations

5 Design flow

Overview

- Par4All is a framework designed for automatic program parallelization
- Heavily relies on PIPS source-to-source compiler framework
  - Programming Integrated Parallel System
  - In French: *Parallélisation Interprocédurale de Programmes Scientifiques*
- Par4All provide a `p4a` script for easy parallelization...
- ...But some people may want to use more expressive interfaces: `tpips`, PyPS, `ipyps`...
- This document describes basic `tpips` concepts and usage

Installation and documentation

- Installation methods
  `http://www.par4all.org/download/RELEASE-NOTES.txt`
- Documentation
  `http://www.par4all.org/documentation`
- Compilation by your own
  - git clone git://git.hpc-project.com/git/par4all.git
  - cd par4all
  - git checkout -b p4a remotes/origin/p4a
  - ./src/simple_tools/p4a_setup [--prefix]
- Activation
  source run/etc/par4all-rc.sh

Definitions of PIPS concepts

- A module is the minimum processing unit of a program to work on
  - A function or subroutine
  - File global definition are in a special module: a compilation unit
  - A source file is split in a collection of modules
- A workspace is a collection of modules from several files
  - A workspace is created with the *create* command
  - Modules are mainly created when files are added to a workspace
- A user transformation is often a succession of PIPS phases
  - A phase is an elementary operation on resources (code, dependence graph, symbol tables, call graph...)
  - A resource is produced by a phase
  - The initial resources are automatically created

Module naming conventions

Source file *ex.c*

```
// function f1
int f1 () {
  return 1;
}

// function f2
static int f2 () {
  return 2;
}
```

Module f1 *f1*

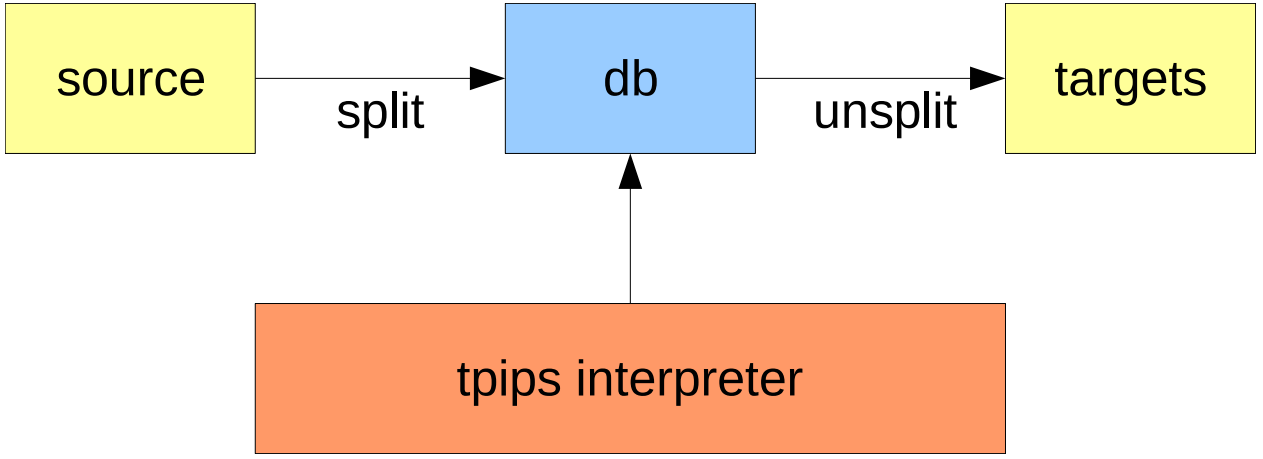
```
int f1 () {
  return 1;
}
```

Module f2 *ex!f2*

```
static int f2 () {
  return 2;
}
```

Basic flow

- Split & parse
  Split the source files in modules and generate internal representation
- Analyze & transform
  Perform one or several code analysis and transformations
- Unsplit
  Regenerate the target source code from the database



Testing your installation

- Set the Par4All path and environment variable
  - bash$ *source run/etc/par4all-rc.sh*
- Start the tpips command interpreter
  - bash$ *tpips*

Session example

```
bash$ tpips
  tpips (ARCH=SOFT_ARCH)
  running as tpips
  ...
  tpips>quit
bash$
```

First example

C function *ex1.c*

```
long max (const long x, const long y) {
  return x < y ? y : x;
}
```

Command file *ex1.tpips*

```
# Just in case there was already an ex1 workspace:
delete          ex1
create          ex1 ex1.c
display         PRINTED_FILE(max)
quit
```

First example

Session

Shell session *ex1.tpips*

```
bash> tpips ex1.tpips
tpips (ARCH=LINUX_x86_64_LL)
  running as tpips
...
PRINTED_FILE made for max.
long max(const long x, const long y)
{
   return x<y?y:x;
}
```

First example

Commands

Command file *ex1.tpips*

```
delete          ex1
create          ex1 ex1.c
display         PRINTED_FILE(max)
```

- *delete ex1*
  Remove the *ex1* workspace
- *create ex1 ex1.c*
  Create a new *ex1* workspace and adds the source file *ex1.c*
- *display PRINTED_FILE(max)*
  Display the module *max* stored in the database

First example

Database

Command file *ex2.tpips*

```
delete          ex1
create          ex1 ex1.c
apply           UNSPLIT(%ALL)
```

- *create ex1*
  Create a workspace as a *ex1.database* directory
- *apply UNSPLIT(%ALL)*
  Rebuild the source files after one or several transformations
  The files are placed in the *ex1.database/Src* directory

Generic commands

- *create name files...*
  Create a new workspace by adding source files
- *delete name*
  Delete a workspace by name
- *close*
  Close a workspace
- *echo message*
  Print a message in the terminal window
- *shell command*
  Execute a shell command
- *quit*
  Exit the interpreter (*exit* also valid)
- *# some comments...*

Resources and phases commands

- *module name*
  Select a module by name and make it the default one
- *apply phase*
  Execute a phase (transformation...) on the current module
  - *apply phase(name)*
    Execute a phase on the specific named module
  - *apply phase(%ALLFUNC)*
    Execute a phase on workspace functions
- *activate rule*
  Activate a particular rule when there are many ones to build a resource
- *display resource*
  Display a resource by name.

Module references

- *%ALLFUNC*
  All functions or subroutines in the workspace
- *%ALL*
  All modules in the workspace, that are *%ALLFUNC* plus all the compilation units describing source files
- *%MODULE*
  The current module
- *%CALLEES*
  All modules called in the given module
- *%CALLERS*
  All modules that calls the given module
- *%PROGRAM*
  The current program

Par4All and OpenMP

- Support for OpenMP 2.5
  - Automatic pragma insertion in the code
  - Need to compile with the `-fopenmp` option with GCC
- Fine Grain Parallelization
  - Based on Allen & Kennedy algorithm
  - Mostly for inner loops
  - Can find more parallelism by distributing everything
  - Put the pressure on memory interface
- Coarse Grain Parallelization
  - Based on array regions analysis
  - Mostly for outer loops
  - No loop distribution
- Both methodologies are supported
  - Difficult to choose between one or the other
  - Experiments are essential in understanding performance gains

Loop parallelization

Initial code

```
void tinit (const long size, long t[size]) {
  long i = 0;
  for (i = 0; i < size; i++) t[i] = 0L;
}
```

Loop parallelization

```
#pragma omp parallel for
   for(i = 0; i <= size-1; i += 1)
      t[i] = 0L;
```

OpenMP Generation

PIPS directives

```
# Privatize loop local scalar variables:
apply      PRIVATIZE_MODULE(tinit)

# Apply parallel code transformation
apply      INTERNALIZE_PARALLEL_CODE(tinit)

# Generate OpenMP directives
apply      OMPIFY_CODE(tinit)
```

Fine grain parallelization — complete example

PIPS script

```
delete          omp
create          omp omp.c
# Select plain style source output:
setproperty     PRETTYPRINT_SEQUENTIAL_STYLE "do"

apply           PRIVATIZE_MODULE(tinit)
apply           INTERNALIZE_PARALLEL_CODE(tinit)
apply           OMPIFY_CODE(tinit)
# Regenerate in .../Src all the transformed files:
apply           UNSPLIT(%ALL)
close
quit
```

Coarse grain parallelization — complete example

PIPS script

```
delete          omp
create          omp omp.c
setproperty     PRETTYPRINT_SEQUENTIAL_STYLE "do"

apply           PRIVATIZE_MODULE(tinit)
apply           COARSE_GRAIN_PARALLELIZATION(tinit)
apply           OMPIFY_CODE(tinit)
apply           UNSPLIT(%ALL)

close
quit
```

Coarse grain array swapping

Array Swapping

```
void tswap (const long size,
            double x[size], double y[size]) {
  long   i = 0;
#pragma omp parallel for
  for (long i = 0; i < size; i++) {
    double t = x[i];
    x[i] = y[i];
    y[i] = t;
  }
}
```

Private variable example

- The variable t has been privatized by PIPS because declared outside of the loop

Array Swapping

```
void tswap (const long size,
            double x[size], double y[size]) {
  long i = 0;
  double t;
#pragma omp parallel for private(t)
  for (long i = 0; i < size; i++) {
    t = x[i];
    x[i] = y[i];
    y[i] = t;
  }
}
```

Call graph

PIPS Script

```
delete    cg
create    cg graph.c

display   CALLGRAPH_FILE(%ALL)

close
quit
```

Dependence graph

PIPS Script

```
delete    dg
create    dg graph.c

# Compute more precise dependences by using array regions
activate  REGION_CHAINS
# Ask for exact regions when possible
activate  MUST_REGIONS

display   DG_FILE(%ALL)

close
quit
```

Automated flow

- Project structure:
  - All transformations integrated in a project build
  - Based on *Makefile*
  - Add a rule to build with PIPS optimization
- File access:
  - All files located in the database (*Src* directory)
  - Compilation in place
  - Must be cautious with dependencies
- Project testing:
  - Must have regression
  - Must compare with sequential results
  - Must exhibit performances benefits

Going further

- There are around 300 phases in PIPS ! ☺
- Look at the documentation (such as *PIPS analyses and transformations*) on `http://pips4u.org/doc/pips-technical-pages` and `http://par4all.org`
- Automate stuff by using PyPS Python front-end
- `p4a` can accept some Python injection `http://download.par4all.org/doc/simple_tools/p4a`
- Ask HPC Project for consultancy and training ☺