Contents

Chapter 1

C++

1.1 Include files

Code 1.1: driver/compiler.h

```
#include "llvm/Support/Host.h"
#include "llvm/ADT/IntrusiveRefCntPtr.h"

#include "clang/Frontend/CompilerInstance.h"
#include "clang/Basic/TargetOptions.h"
#include "clang/Basic/TargetInfo.h"
#include "clang/Basic/FileManager.h"
#include "clang/Basic/SourceManager.h"
#include "clang/Lex/Preprocessor.h"
#include "clang/Lex/Lexer.h"
#include "clang/Basic/Diagnostic.h"
#include "clang/AST/ASTContext.h"

/*
 * ---- Custom class to instantiate an object of clang::CompilerInstance
   with the options and the file
 *    passed with argv.
 */
class ClangCompiler {

private:
  clang::CompilerInstance compiler_;

public:
  ClangCompiler(int argc, char **argv);

  clang::SourceManager &getSourceManager() { return compiler_.
   getSourceManager(); }
  clang::DiagnosticConsumer getDiagnosticClient() { return compiler_.
   getDiagnosticClient(); }
  clang::LangOptions getLangOpts() { return compiler_.getLangOpts(); }
  clang::Preprocessor &getPreprocessor() { return compiler_.getPreprocessor
   (); }
  clang::ASTContext &getASTContext() { return compiler_.getASTContext(); }
  clang::FileManager &getFileManager() { return compiler_.getFileManager();
   }

};
```

Code 1.2: driver/program.h

```
#include "driver/compiler.h"
#include "utils/source_locations.h"
#include "pragma_handler/Root.h"

#include "clang/StaticAnalyzer/Core/PathSensitive/CheckerContext.h"
#include "clang/Basic/DiagnosticOptions.h"
#include "clang/Frontend/TextDiagnosticPrinter.h"
#include "clang/AST/ASTConsumer.h"
#include "clang/Parse/Parser.h"
#include "clang/Parse/ParseAST.h"
#include "clang/Rewrite/Core/Rewriter.h"
#include "llvm/Support/raw_ostream.h"
#include <string>
#include <iostream>

/*
 * ---- Instantiate a compiler object and start the parser.
 */
class Program {
/* Contains the list of all the pragmas in the source code */
std::vector<clang::OMPExecutableDirective *> *pragma_list_;
/* Contains the list of all the functions defined in the source code (for
   profiling pourpuse) */
std::vector<clang::FunctionDecl *> *function_list_;

/* To create the profiling code and the list of pragmas */
void ParseSourceCode(std::string fileName);
/* To create the final source code to be used with the scheduler */
void ParseSourceCode(std::string fileName, std::vector<Root *> *root_vect);

public:
  /* To create the profiling code and the list of pragmas */
  Program(int argc, char **argv) : ccompiler_(argc, argv), pragma_list_(NULL
   ), function_list_(NULL) {
    ParseSourceCode(argv[argc - 1]);
  }

  /* To create the final source code to be used with the scheduler */
  Program(int argc,char **argv, std::vector<Root *> *root_vect) : ccompiler_
   (argc, argv), pragma_list_(NULL), function_list_(NULL) {
    ParseSourceCode(argv[argc - 1], root_vect);
  }

  std::vector<clang::OMPExecutableDirective *> *getPragmaList() { return
   pragma_list_; }
  std::vector<clang::FunctionDecl *> *getFunctionList() { return
   function_list_; }

  ClangCompiler ccompiler_;
};


/*
 * ---- Recursively visit the AST of the source code to exctract the pragmas
    and rewrite it
 *      adding profile call.
 */
class ProfilingRecursiveASTVisitor: public clang::RecursiveASTVisitor<
   ProfilingRecursiveASTVisitor> {
```

```
53
54    /* Class to rewrite the code */
55    clang::Rewriter &rewrite_profiling_;
56
57    const clang::SourceManager& sm;
58
59    bool include_inserted_;
60    clang::Stmt *previous_stmt_;
61
62    /* Add profiling call to a pragma stmt */
63    void RewriteProfiling(clang::Stmt *s);
64    /* Given a ForStmt retrieve the value of the condition variable, to know
       how many cycles will
65       do the for */
66    std::string ForConditionVarValue(const clang::Stmt *s);
67    /* For a given stmt retrive the line of the function where it is defined
       */
68    unsigned GetFunctionLineForPragma(clang::SourceLocation sl);
69
70  public:
71    ProfilingRecursiveASTVisitor(clang::Rewriter &r_profiling, const clang::
       SourceManager& sm) :
72            rewrite_profiling_(r_profiling), sm(sm), include_inserted_(false),
       previous_stmt_(NULL) { }
73
74    /* This function is called for each stmt in the AST */
75    bool VisitStmt(clang::Stmt *s);
76    /* This function is called for each function in the AST */
77    bool VisitFunctionDecl(clang::FunctionDecl *f);
78    bool VisitDecl(clang::Decl *decl);
79    std::vector<clang::OMPExecutableDirective *> pragma_list_;
80    std::vector<clang::FunctionDecl *> function_list_;
81
82  };
83
84  /*
85   * ---- Is responible to call ProfilingRecurseASTVisitor.
86   */
87  class ProfilingASTConsumer : public clang::ASTConsumer {
88  public:
89
90    ProfilingASTConsumer(clang::Rewriter &r_profiling, const clang::
       SourceManager& sm) :
91            recursive_visitor_(r_profiling, sm) { }
92
93    /* Traverse the AST invoking the RecursiveASTVisitor functions */
94    virtual bool HandleTopLevelDecl(clang::DeclGroupRef d) {
95      typedef clang::DeclGroupRef::iterator iter;
96      for (iter b = d.begin(), e = d.end(); b != e; ++b) {
97        recursive_visitor_.TraverseDecl(*b);
98      }
99      return true;
100   }
101
102   ProfilingRecursiveASTVisitor recursive_visitor_;
103   std::vector<clang::OMPExecutableDirective *> pragma_list_;
104   std::vector<clang::FunctionDecl *> function_list_;
105 };
106
```

```
107
108 /*
109  * ---- Recursively visit the AST and repleace each pragma with a function
       call.
110  */
111 class TransformRecursiveASTVisitor: public clang::RecursiveASTVisitor<
       TransformRecursiveASTVisitor> {
112
113   clang::Rewriter &rewrite_pragma_;
114
115   const clang::SourceManager& sm;
116
117   /* Needed because the parse retrive twice each pragma stmt */
118   clang::Stmt *previous_stmt_;
119   /* Check if the inlude command has been already inserted*/
120   bool include_inserted_;
121
122   std::vector<Root *> *root_vect_;
123
124   void RewriteOMPPragma(clang::Stmt *associated_stmt, std::string
       pragma_name);
125   void RewriteOMPBarrier(clang::OMPExecutableDirective *omp_stmt);
126   std::string RewriteOMPFor(Node *n);
127
128   /* Given a pragma stmt retrive the Node object that contains all its info
       */
129   Node *GetNodeObjForPragma(clang::Stmt *s);
130   /* Called by GetNodeObjForPragma is used because the Node objs are saved
       in a tree */
131   Node *RecursiveGetNodeObjforPragma(Node *n, unsigned stmt_start_line);
132
133
134 public:
135   TransformRecursiveASTVisitor(clang::Rewriter &r_pragma_, std::vector<Root
       *> *root_vect, const clang::SourceManager& sm) :
136           rewrite_pragma_(r_pragma_), root_vect_(root_vect), sm(sm),
       include_inserted_(false), previous_stmt_(NULL) { }
137
138   bool VisitStmt(clang::Stmt *s);
139   bool VisitFunctionDecl(clang::FunctionDecl *f);
140   bool VisitDecl(clang::Decl *decl);
141 };
142
143 /*
144  * ---- Responsible to invoke TransformRecursiveASTVisitor.
145  */
146 class TransformASTConsumer : public clang::ASTConsumer {
147 public:
148
149   TransformASTConsumer(clang::Rewriter &RPragma, std::vector<Root *> *
       rootVect, const clang::SourceManager& sm) :
150           recursive_visitor_(RPragma, rootVect, sm) { }
151
152   virtual bool HandleTopLevelDecl(clang::DeclGroupRef d) {
153     typedef clang::DeclGroupRef::iterator iter;
154     for (iter b = d.begin(), e = d.end(); b != e; ++b) {
155       recursive_visitor_.TraverseDecl(*b);
156     }
157     return true;
```

```
  }

  TransformRecursiveASTVisitor recursive_visitor_;
};
```

Code 1.3: pragma_handler/Node.h

```
#include "pragma_handler/ForNode.h"

/* Contains info about function */
struct FunctionInfo {
  clang::FunctionDecl *function_decl_;

  unsigned function_start_line_;
  unsigned function_end_line_;
  std::string function_name_;
  std::string function_return_type_;
  int num_params_;
  /* Matrix Nx2. Contains the list of the parameter of the functions: type
   name */
  std::string **function_parameters_;

  std::string function_class_name_;
};

/*
 * ---- Contains all the relevant information of a given pragma.
 */
class Node {

private:

  clang::OMPExecutableDirective *pragma_stmt_;

/* Stmt start and end line in the source file */
  std::string file_name_;
  int start_line_, start_column_;
  int end_line_, end_column_;

/*Line number of the function that contains this pragma */
  FunctionInfo parent_funct_info_;

/* Variables to construct the tree */
  Node *parent_node_;

  /*Pragma name with all the parameters */
  //std::string pragma_type_;

/* Function to exctract all the parameters of the pragma */
  void setPragmaClauses(clang::SourceManager& sm);

public:
  /*Pragma name with all the parameters */
  std::string pragma_type_;

  bool profiled_ = false;

  ForNode *for_node_;

  std::vector<Node *> *children_vect_;
```

```
53
54    typedef std::map<std::string, std::string> VarList_;
55    std::map<std::string, VarList_> *option_vect_;
56
57    Node(clang::OMPExecutableDirective *pragma_stmt, clang::FunctionDecl *
        funct_decl, clang::SourceManager& sm);
58
59    void setSourceLocation(const clang::SourceManager& sm);
60
61  /*
62   * ---- Set the line, name, return type and parameters of the function
        containig the pragma ----
63   */
64    void setParentFunction(clang::FunctionDecl *funct_decl, const clang::
        SourceManager& sm);
65
66    FunctionInfo getParentFunctionInfo() { return parent_funct_info_; }
67
68    void AddChildNode(Node *n) { children_vect_->push_back(n); }
69
70    void setParentNode(Node *n) { parent_node_ = n; }
71    Node* getParentNode() { return parent_node_; }
72
73    int getEndLine() { return end_line_; }
74    int getStartLine() { return start_line_; }
75
76    void CreateXMLPragmaNode(tinyxml2::XMLDocument *xml_doc, tinyxml2::
        XMLElement *pragmas_element);
77    void CreateXMLPragmaOptions(tinyxml2::XMLDocument *xml_doc,tinyxml2::
        XMLElement *options_element);
78  };
```

Code 1.4: pragma_handler/ForNode.h

```
1  #include "xml_creator/tinyxml2.h"
2
3  #include "utils/source_locations.h"
4  #include "clang/AST/ASTConsumer.h"
5  #include "clang/Sema/Lookup.h"
6  #include "clang/Frontend/CompilerInvocation.h"
7  #include "clang/AST/ASTContext.h"
8  #include "clang/Sema/Scope.h"
9  #include "clang/Parse/ParseAST.h"
10
11 #include <iostream>
12 #include <string>
13 #include <stdio.h>
14 #include <stdlib.h>
15
16 class ForNode {
17
18 public:
19   clang::ForStmt *for_stmt_;
20
21   /* Loop variable */
22   std::string loop_var_;
23   std::string loop_var_type_;
24
25   /* Loop variable initial value: (number or variable) */
26   int loop_var_init_val_;
```

```
  bool loop_var_init_val_set_;
  std::string loop_var_init_var_;

  /* Loop condition */
  std::string condition_op_;
  int condition_val_;
  bool condition_val_set_;
  std::string condition_var_;

  /* Loop increment */
  std::string increment_op_;
  int increment_val_;
  bool increment_val_set_;
  std::string increment_var_;

  void ExtractForParameters(clang::ForStmt *for_stmt);

  void ExtractForInitialization(clang::ForStmt *for_stmt);
  void ExtractForCondition(clang::ForStmt *for_stmt);
  void ExtractForIncrement(clang::ForStmt *for_stmt);


  ForNode(clang::ForStmt *for_stmt);
  void CreateXMLPragmaFor(tinyxml2::XMLDocument *xml_doc, tinyxml2::
   XMLElement *for_element);

};
```

Code 1.5: pragma_handler/Root.h

```
#include "pragma_handler/Node.h"

/*
 * ---- It's the root node of the annidation tree of the pragmas in a
   specific function
 *    and contains the first level pragmas.
 */
class Root {
private:
  FunctionInfo function_info_;

  Node *last_node_;

public:
  Root(Node *n, FunctionInfo funct_info);

  std::vector<Node *> *children_vect_;

  void setLastNode(Node *n) {last_node_ = n; };
  Node* getLastNode() { return last_node_; };

  void AddChildNode(Node *n) { children_vect_->push_back(n); };

  void CreateXMLFunction(tinyxml2::XMLDocument *xml_doc);

  unsigned getFunctionLineStart(){ return function_info_.
   function_start_line_; }
  unsigned getFunctionLineEnd() {return function_info_.function_end_line_; }

};
```

Code 1.6: utils/source_locations.h

```
#include <string>
#include <clang/Basic/SourceLocation.h>
#include <clang/Basic/SourceManager.h>
#include <sstream>

#include <llvm/Support/raw_ostream.h>

namespace clang {
class SourceLocation;
class SourceRange;
class SourceManager;
}

namespace utils {

std::string FileName(clang::SourceLocation const& l, clang::SourceManager
    const& sm);

std::string FileId(clang::SourceLocation const& l, clang::SourceManager
    const& sm);

unsigned Line(clang::SourceLocation const& l, clang::SourceManager const& sm
    );

std::pair<unsigned, unsigned> Line(clang::SourceRange const& r, clang::
    SourceManager const& sm);

unsigned Column(clang::SourceLocation const& l, clang::SourceManager const&
    sm);

std::pair<unsigned, unsigned> Column(clang::SourceRange const& r, clang::
    SourceManager const& sm);

std::string location(clang::SourceLocation const& l, clang::SourceManager
    const& sm);

}
```

1.2 Source files

Code 1.7: main.cpp

```
int main(int argc, char **argv) {

  if(argc < 2) {
    llvm::errs() << "Usage: Source_exctractor [<options>] <filename>\n";
      return 1;
  }
/*
 * ---- Create a clang::compiler object and launch the parser saving the
    pragma stmt.
 *    Rewrite the sourcecode adding profiling call.
 */
  Program p_parser(argc, argv);

/*
```

```
 * ---- With the information exctracted by the parser create a linked list
    tree of objects containing
 *    all the necessary information of the pragmas.
 */
  std::vector<Root *> *root_vect = CreateTree(program.getPragmaList(),
   program.getFunctionList(), program.ccompiler_.getSourceManager());
/*
 * ---- Using the tree above create an xml file containing the pragma info.
    This file is used to produce the scheduler.
 */
  CreateXML(root_vect, argv[argc - 1]);

  for(std::vector<Root *>::iterator itr = root_vect->begin(); itr !=
   root_vect->end(); ++itr)
    (*itr)->VisitTree();

/*
 * ---- Parse the sourcecode and rewrite it substituting pragmas with
    function calls. This new file
 *    will be used with the scheduler to produce the final output.
 */
  Program p_rewriter(argc, argv, root_vect);

  return 0;
}
```

Code 1.8: driver/compiler.cpp

```
#include "driver/compiler.h"

using namespace clang;

ClangCompiler::ClangCompiler(int argc, char **argv) {

  DiagnosticOptions diagnosticOptions;
  compiler_.createDiagnostics();

  /* Create an invocation that passes any flags to preprocessor */
  CompilerInvocation *Invocation = new CompilerInvocation;
  CompilerInvocation::CreateFromArgs(*Invocation, argv + 1, argv + argc,
                                     compiler_.getDiagnostics());
  compiler_.setInvocation(Invocation);

  /* Set default target triple */
  llvm::IntrusiveRefCntPtr<TargetOptions> pto( new TargetOptions());
  pto->Triple = llvm::sys::getDefaultTargetTriple();
  llvm::IntrusiveRefCntPtr<TargetInfo> pti(TargetInfo::CreateTargetInfo(
   compiler_.getDiagnostics(), pto.getPtr()));
  compiler_.setTarget(pti.getPtr());

  compiler_.createFileManager();
  compiler_.createSourceManager(compiler_.getFileManager());

  /* Add default search path for the compiler */
  HeaderSearchOptions &headerSearchOptions = compiler_.getHeaderSearchOpts()
   ;

  headerSearchOptions.AddPath("/usr/local/include",
            clang::frontend::Angled,
            false,
```

```
            false);

  headerSearchOptions.AddPath("/usr/include",
            clang::frontend::Angled,
            false,
            false);

  headerSearchOptions.AddPath("/usr/lib/gcc/x86_64-linux-gnu/4.8/include",
            clang::frontend::Angled,
            false,
            false);

  headerSearchOptions.AddPath("/usr/include/x86_64-linux-gnu",
            clang::frontend::Angled,
            false,
            false);
  headerSearchOptions.AddPath("/usr/include/c++/4.8/",
            clang::frontend::Angled,
            false,
            false);

  headerSearchOptions.AddPath("/usr/include/x86_64-linux-gnu/c++/4.8/",
            clang::frontend::Angled,
            false,
            false);


  /* Allow C++ code to get rewritten */
  clang::LangOptions langOpts;
  langOpts.GNUMode = 1;
  langOpts.CXXExceptions = 1;
  langOpts.RTTI = 1;
  langOpts.Bool = 1;
  langOpts.CPlusPlus = 1;
  Invocation->setLangDefaults(langOpts,
                              clang::IK_CXX,
                              clang::LangStandard::lang_cxx0x);

  compiler_.createPreprocessor();
  compiler_.getPreprocessorOpts().UsePredefines = false;

  compiler_.createASTContext();

  /* Initialize the compiler and the source manager with a file to process
   */
  std::string fileName(argv[argc - 1]);
  const FileEntry *pFile = compiler_.getFileManager().getFile(fileName);
  compiler_.getSourceManager().createMainFileID(pFile);
  compiler_.getDiagnosticClient().BeginSourceFile(compiler_.getLangOpts(), &
   compiler_.getPreprocessor());

}
```

Code 1.9: driver/program.cpp

```
#include "driver/program.h"

void Program::ParseSourceCode(std::string file_name) {

  /* Convert <file>.c to <file_profile>.c */
```

```
6     std::string out_filename_profile (file_name);
7     size_t ext = out_filename_profile.rfind(".");
8     if (ext == std::string::npos)
9       ext = out_filename_profile.length();
10    out_filename_profile.insert(ext, "_profile");
11
12    llvm::errs() << "Output to: " << out_filename_profile << "\n";
13    std::string out_error_info;
14    llvm::raw_fd_ostream out_file_profile(out_filename_profile.c_str(),
       out_error_info, 0);
15
16    /* Create the rewriter object to create the profiling file */
17    clang::Rewriter rewrite_profiling;
18    rewrite_profiling.setSourceMgr(ccompiler_.getSourceManager(), ccompiler_.
       getLangOpts());
19
20    ProfilingASTConsumer ast_consumer(rewrite_profiling, ccompiler_.
       getSourceManager());
21    /* Parse the AST with the custom ASTConsumer */
22    clang::ParseAST(ccompiler_.getPreprocessor(), &ast_consumer, ccompiler_.
       getASTContext());
23    ccompiler_.getDiagnosticClient().EndSourceFile();
24
25    /* Save the pragma and function list */
26    pragma_list_ = new std::vector<clang::OMPExecutableDirective *>(
       ast_consumer.recursive_visitor_.pragma_list_);
27    function_list_ = new std::vector<clang::FunctionDecl *>(ast_consumer.
       recursive_visitor_.function_list_);
28
29    /*Output rewritten source code into a new file */
30    const clang::RewriteBuffer *rewrite_buf_profiling =
31        rewrite_profiling.getRewriteBufferFor(ccompiler_.getSourceManager().
       getMainFileID());
32
33    out_file_profile << std::string(rewrite_buf_profiling->begin(),
       rewrite_buf_profiling->end());
34    out_file_profile.close();
35
36  }
37
38  bool ProfilingRecursiveASTVisitor::VisitDecl(clang::Decl *decl) {
39
40    clang::SourceLocation cxx_start_src_loc = decl->getLocStart();
41    if(sm.getFileID(cxx_start_src_loc) == sm.getMainFileID()
42        && clang::isa<clang::CXXRecordDecl>(decl)
43        && include_inserted_ == false) {
44      include_inserted_ = true;
45      std::string text_include =
46        "#include \"profile_tracker/profile_tracker.h\"\n";
47      rewrite_profiling_.InsertText(cxx_start_src_loc, text_include, true,
       false);
48    }
49
50    return true;
51  }
52
53  /*
54   * ---- Insert the call to the profilefunction tracker to track the
       execution time of each funcion.
```

```
 */
bool ProfilingRecursiveASTVisitor::VisitFunctionDecl(clang::FunctionDecl *f) {

  clang::SourceLocation start_src_loc = f->getLocStart();
  unsigned funct_start_line = utils::Line(start_src_loc, sm);


  /* Skip function belonging to external include file and not defined function */
  if(sm.getFileID(start_src_loc) == sm.getMainFileID() && f->hasBody() == true) {

    function_list_.push_back(f);

    /* Include the path to ProfileTracker.h */
    if(include_inserted_ == false) {
      std::string text_include =
      "#include \"profile_tracker/profile_tracker.h\"\n";

      rewrite_profiling_.InsertText(start_src_loc, text_include, true, false);
      include_inserted_ = true;
    }

    start_src_loc = f->getBody()->getLocStart();
    unsigned start_line = utils::Line(start_src_loc, sm);
    clang::SourceLocation new_start_src_loc = sm.translateLineCol(sm.getMainFileID(), start_line + 1, 1);
    std::stringstream text_profiling;
    text_profiling << "if( ProfileTracker x = ProfileTrackParams(" << funct_start_line << ", 0)) {\n";

    /* Insert the if in the first line of the function definition */
    rewrite_profiling_.InsertText(new_start_src_loc, text_profiling.str(), true, false);

    clang::SourceLocation end_src_loc = f->getLocEnd();
    std::stringstream text_end_bracket;
    text_end_bracket << "}\n";
    /* Close the if bracket at the end of the function */
    rewrite_profiling_.InsertText(end_src_loc, text_end_bracket.str(), true, false);
  }

  return true;
}

bool ProfilingRecursiveASTVisitor::VisitStmt(clang::Stmt *s) {

  clang::SourceLocation start_src_loc = s->getLocStart();
  if(sm.getFileID(start_src_loc) == sm.getMainFileID()) {
      /* We want just the OpenMP stmt and no duplicate */
      if (clang::isa<clang::OMPExecutableDirective>(s) && s != previous_stmt_) {
        previous_stmt_ = s;
        clang::OMPExecutableDirective *omp_stmt = static_cast<clang::OMPExecutableDirective *>(s);
        pragma_list_.push_back(omp_stmt);
```

```
104
105         clang::Stmt *associated_stmt = omp_stmt->getAssociatedStmt();
106         if(associated_stmt) {
107           clang::Stmt *captured_stmt = static_cast<clang::CapturedStmt *>(
    associated_stmt)->getCapturedStmt();
108           /* In the case of #omp parallel for we have to go down two level
    befor finding the ForStmt */
109           if(strcmp(captured_stmt->getStmtClassName(), "OMPForDirective") !=
     0)
110             RewriteProfiling(captured_stmt);
111         }
112       }
113     }
114
115   return true;
116 }
117
118 void ProfilingRecursiveASTVisitor::RewriteProfiling(clang::Stmt *s) {
119
120     clang::SourceLocation start_src_loc = s->getLocStart();
121     unsigned pragma_start_line = utils::Line(start_src_loc,sm);
122     unsigned function_start_line = GetFunctionLineForPragma(s->getLocStart()
    );
123
124     std::stringstream text_profiling;
125     if(clang::isa<clang::ForStmt>(s)) {
126       std::string condition_var_value = ForConditionVarValue(s);
127       //std::string conditionVar = "";
128       text_profiling << "if(␣ProfileTracker␣x␣=␣ProfileTrackParams("
129           << function_start_line << ",␣" << pragma_start_line << ",␣" <<
    condition_var_value << "))\n";
130       rewrite_profiling_.InsertText(start_src_loc, text_profiling.str(),
    true, true);
131
132     } else {
133       text_profiling << "if(␣ProfileTracker␣x␣=␣ProfileTrackParams("
134           << function_start_line << ",␣" << pragma_start_line << "))\n";
135       rewrite_profiling_.InsertText(start_src_loc, text_profiling.str(),
    true, true);
136     }
137
138     /* Comment the pragma in the profiling file */
139     clang::SourceLocation pragma_start_src_loc =
140         sm.translateLineCol(sm.getMainFileID(), pragma_start_line - 1, 1);
141
142     rewrite_profiling_.InsertText(pragma_start_src_loc, "//", true, false);
143 }
144
145 std::string ProfilingRecursiveASTVisitor::ForConditionVarValue(const clang::
    Stmt *s) {
146
147   const clang::ForStmt *for_stmt = static_cast<const clang::ForStmt *>(s);
148   const clang::Expr *condition_expr = for_stmt->getCond();
149   const clang::BinaryOperator *binary_op = static_cast<const clang::
     BinaryOperator *>(condition_expr);
150
151   std::string start_cond_var_value, end_cond_var_value;
152
153 /*
```

```
154  *  Condition end value
155  */
156   const clang::Expr *right_expr = binary_op->getRHS();
157
158   if(strcmp(right_expr->getStmtClassName(), "IntegerLiteral") == 0) {
159     const clang::IntegerLiteral *int_literal = static_cast<const clang::
      IntegerLiteral *>(right_expr);
160     std::stringstream text_end_value;
161     text_end_value << int_literal->getValue().getZExtValue();
162     //return text.str();
163     end_cond_var_value = text_end_value.str();
164
165   } else if(strcmp(right_expr->getStmtClassName(), "ImplicitCastExpr") == 0)
       {
166     const clang::DeclRefExpr *decl_ref_expr =
167         static_cast<const clang::DeclRefExpr *>(*(right_expr->child_begin())
      );
168
169     const clang::NamedDecl *named_decl = decl_ref_expr->getFoundDecl();
170     //return nD->getNameAsString();
171     end_cond_var_value = named_decl->getNameAsString();
172   }
173
174 /*
175  * Condition start value
176  */
177
178 /*
179  *  for (int i = ...)
180  */
181   if(strcmp(for_stmt->child_begin()->getStmtClassName(), "DeclStmt") == 0) {
182     const clang::DeclStmt *decl_stmt = static_cast<const clang::DeclStmt
      *>(*(for_stmt->child_begin()));
183     const clang::Decl *decl = decl_stmt->getSingleDecl();
184
185 /*
186  *  for (... = 0)
187  */
188     if(strcmp(decl_stmt->child_begin()->getStmtClassName(), "IntegerLiteral"
      ) == 0) {
189       const clang::IntegerLiteral *int_literal =
190           static_cast<const clang::IntegerLiteral *>(*(decl_stmt->
      child_begin()));
191
192       std::stringstream text_star_value;
193       text_star_value << int_literal->getValue().getZExtValue();
194       start_cond_var_value = text_star_value.str();
195
196 /*
197  *  for (... = a)
198  */
199     }else if (strcmp(decl_stmt->child_begin()->getStmtClassName(), "
      ImplicitCastExpr") == 0) {
200       const clang::DeclRefExpr *decl_ref_expr =
201           static_cast<const clang::DeclRefExpr *>(*(decl_stmt->child_begin()
      ->child_begin()));
202
203       const clang::NamedDecl *named_decl = decl_ref_expr->getFoundDecl();
204       start_cond_var_value = named_decl->getNameAsString();
```

```
205     }
206   }
207 /*
208  *  for ( i = ...)
209  */
210   else if(strcmp(for_stmt->child_begin()->getStmtClassName(), "
      BinaryOperator") == 0) {
211     const clang::BinaryOperator *binary_op =
212         static_cast<const clang::BinaryOperator *>(*(for_stmt->child_begin()
      ));
213     const clang::DeclRefExpr *decl_ref_expr =
214         static_cast<const clang::DeclRefExpr *>(*(binary_op->child_begin()))
      ;
215 /*
216  *  for( ... = 0)
217  */
218     clang::ConstStmtIterator stmt_itr = binary_op->child_begin();
219     stmt_itr ++;
220     if(strcmp(stmt_itr->getStmtClassName(), "IntegerLiteral") == 0) {
221       const clang::IntegerLiteral *int_literal = static_cast<const clang::
      IntegerLiteral *>(*stmt_itr);
222       start_cond_var_value = int_literal->getValue().getZExtValue();
223 /*
224  *  for ( ... = a)
225  */
226     } else if (strcmp(stmt_itr->getStmtClassName(), "ImplicitCastExpr") ==
      0) {
227       const clang::DeclRefExpr *decl_ref_expr =
228           static_cast<const clang::DeclRefExpr *>(*(stmt_itr->child_begin())
      );
229       const clang::NamedDecl *named_decl = decl_ref_expr->getFoundDecl();
230       start_cond_var_value = named_decl->getNameAsString();
231     }
232   }
233   end_cond_var_value.append("␣-␣");
234   end_cond_var_value.append(start_cond_var_value);
235   return end_cond_var_value;
236 }
237
238 unsigned ProfilingRecursiveASTVisitor::GetFunctionLineForPragma(clang::
      SourceLocation sl) {
239
240   unsigned pragma_line = utils::Line(sl, sm);
241
242   unsigned start_func_line, end_func_line;
243   std::vector<clang::FunctionDecl *>::iterator func_itr;
244
245   for(func_itr = function_list_.begin(); func_itr != function_list_.end();
      ++ func_itr) {
246     start_func_line = utils::Line((*func_itr)->getSourceRange().getBegin(),
      sm);
247     end_func_line = utils::Line((*func_itr)->getSourceRange().getEnd(), sm);
248     if(pragma_line < end_func_line && pragma_line > start_func_line)
249       return start_func_line;
250   }
251
252   return 0;
253 }
254
```

```
255
256 void Program::ParseSourceCode(std::string fileName, std::vector<Root *> *
    root_vect) {
257
258   /* Convert <file>.c to <file_transformed>.c */
259   std::string out_name_pragma (fileName);
260   size_t ext = out_name_pragma.rfind(".");
261   if (ext == std::string::npos)
262     ext = out_name_pragma.length();
263   out_name_pragma.insert(ext, "_transformed");
264
265   llvm::errs() << "Output to: " << out_name_pragma << "\n";
266   std::string out_error_info;
267   llvm::raw_fd_ostream out_file_pragma(out_name_pragma.c_str(),
    out_error_info, 0);
268
269   clang::Rewriter rewrite_pragma;
270   rewrite_pragma.setSourceMgr(ccompiler_.getSourceManager(), ccompiler_.
    getLangOpts());
271
272   TransformASTConsumer t_ast_consumer(rewrite_pragma, root_vect, ccompiler_.
    getSourceManager());
273
274   /* Parse the AST */
275   clang::ParseAST(ccompiler_.getPreprocessor(), &t_ast_consumer, ccompiler_.
    getASTContext());
276   ccompiler_.getDiagnosticClient().EndSourceFile();
277
278   const clang::RewriteBuffer *rewrite_buff_pragma =
279       rewrite_pragma.getRewriteBufferFor(ccompiler_.getSourceManager().
    getMainFileID());
280   out_file_pragma << std::string(rewrite_buff_pragma->begin(),
    rewrite_buff_pragma->end());
281   out_file_pragma.close();
282 }
283
284 bool TransformRecursiveASTVisitor::VisitDecl(clang::Decl *decl) {
285
286   clang::SourceLocation cxx_start_src_loc = decl->getLocStart();
287   if(sm.getFileID(cxx_start_src_loc) == sm.getMainFileID()
288       && clang::isa<clang::CXXRecordDecl>(decl)
289       && include_inserted_ == false) {
290     include_inserted_ = true;
291     std::string text_include = "#include \"thread_pool/threads_pool.h\"\n";
292     rewrite_pragma_.InsertText(cxx_start_src_loc, text_include, true, false)
    ;
293   }
294
295   return true;
296 }
297
298 bool TransformRecursiveASTVisitor::VisitFunctionDecl(clang::FunctionDecl *f)
    {
299   clang::SourceLocation f_start_src_loc = f->getLocStart();
300
301   if(sm.getFileID(f_start_src_loc) == sm.getMainFileID() && !clang::isa<
    clang::CXXMethodDecl>(f)) {
302     if(include_inserted_ == false) {
303       include_inserted_ = true;
```

```
304
305       std::string text_include = "#include␣\"thread_pool/threads_pool.h\"\n"
      ;
306
307       rewrite_pragma_.InsertText(f_start_src_loc, text_include, true, false)
      ;
308     }
309   }
310
311   return true;
312 }
313
314 bool TransformRecursiveASTVisitor::VisitStmt(clang::Stmt *s) {
315
316   clang::SourceLocation s_start_stc_loc = s->getLocStart();
317   /* Visit only stmt in the source file (not in included file) and that are
       pragma stmt */
318   if(sm.getFileID(s_start_stc_loc) == sm.getMainFileID()
319         && clang::isa<clang::OMPExecutableDirective>(s)
320         && s != previous_stmt_) {
321
322     previous_stmt_ = s;
323     clang::OMPExecutableDirective *omp_stmt = static_cast<clang::
      OMPExecutableDirective *>(s);
324     clang::Stmt *associated_stmt = omp_stmt->getAssociatedStmt();
325     if(associated_stmt) {
326       clang::Stmt *captured_stmt = static_cast<clang::CapturedStmt *>(
      associated_stmt)->getCapturedStmt();
327       if(strcmp(captured_stmt->getStmtClassName(), "OMPForDirective") != 0)
328         RewriteOMPPragma(associated_stmt, omp_stmt->getStmtClassName());
329
330     }else if(strcmp(omp_stmt->getStmtClassName(), "OMPBarrierDirective") ==
      0
331           || strcmp(omp_stmt->getStmtClassName(), "OMPTaskwaitDirective")
      == 0){
332       RewriteOMPBarrier(omp_stmt);
333     }
334   }
335   return true;
336 }
337
338 void TransformRecursiveASTVisitor::RewriteOMPBarrier(clang::
      OMPExecutableDirective *omp_stmt) {
339   unsigned stmt_start_line = utils::Line(omp_stmt->getLocStart(), sm);
340
341   std::stringstream text_barrier;
342   text_barrier <<
343 "{\n\
344 ␣␣class␣Nested␣:␣public␣NestedBase␣{\n\
345 ␣␣public:␣\n\
346 ␣␣␣␣virtual␣std::shared_ptr<NestedBase>␣clone()␣const␣{␣return␣std::
      make_shared<Nested>(*this);␣}␣\n\
347 ␣␣␣␣Nested(int␣pragma_id)␣:␣NestedBase(pragma_id)␣{}\n\
348 ␣␣␣␣void␣callme(ForParameter␣for_param){}\n\
349 ␣␣};\n\
350 ␣␣ThreadPool::getInstance(\"" << utils::FileName(omp_stmt->getLocStart(), sm
      )
351       << "\")->call(std::make_shared<Nested>(" << stmt_start_line << "));\n\
352 }";
```

```
353
354   clang::SourceLocation pragma_start_src_loc = sm.translateLineCol(sm.
        getMainFileID(), stmt_start_line + 1, 1);
355   rewrite_pragma_.InsertText(pragma_start_src_loc, text_barrier.str(), true,
        false);
356
357   pragma_start_src_loc = sm.translateLineCol(sm.getMainFileID(),
        stmt_start_line, 1);
358   rewrite_pragma_.InsertText(pragma_start_src_loc, "//", true, false);
359 }
360
361
362 void TransformRecursiveASTVisitor::RewriteOMPPragma(clang::Stmt *
        associated_stmt, std::string pragma_name) {
363
364   clang::Stmt *s = static_cast<clang::CapturedStmt *>(associated_stmt)->
        getCapturedStmt();
365
366   clang::SourceLocation stmt_start_src_loc = s->getLocStart();
367   unsigned pragma_start_line = utils::Line(stmt_start_src_loc, sm);
368
369   Node *n = GetNodeObjForPragma(s);
370
371   std::stringstream text;
372   std::stringstream text_constructor_params;
373   std::stringstream text_class_var;
374   std::stringstream text_fx_var;
375   std::stringstream text_constructor_var;
376   std::stringstream text_constructor;
377
378 /* Insert before pragma */
379   text <<
380 "{\n\
381 ␣␣class␣Nested␣:␣public␣NestedBase␣{\n\
382 ␣␣public:␣\n\
383 ␣␣␣␣virtual␣std::shared_ptr<NestedBase>␣clone()␣const␣{␣return␣std::
        make_shared<Nested>(*this);␣}␣\n\
384 ␣␣␣␣Nested(int␣pragma_id";
385
386   text_constructor << "␣:␣NestedBase(pragma_id)";
387
388   clang::CapturedStmt *captured_stmt = static_cast<clang::CapturedStmt *>(
        associated_stmt);
389   /* Iterate over all the variable used inside a pragma but defined outside.
        These variable have to be passed to
390      the newly created function */
391   for(clang::CapturedStmt::capture_iterator capture_var_itr = captured_stmt
        ->capture_begin();
392       capture_var_itr != captured_stmt->capture_end();
393       ++capture_var_itr){
394
395     clang::VarDecl *var_decl = capture_var_itr->getCapturedVar();
396     std::string var_type = var_decl->getType().getAsString();
397
398     if(capture_var_itr != captured_stmt->capture_begin()){
399       text_fx_var << ",␣";
400       text_constructor_var << ",␣";
401       text_constructor_params << ",␣";
402     }else
```

```
403       text << ",␣";
404     std::cout << var_type << "␣-␣";
405     size_t pos_class = var_type.find("class");
406     if(pos_class != std::string::npos){
407       std::cout << "removing␣class␣-␣";
408       var_type.erase(pos_class, pos_class + 5);
409     }
410
411     size_t pos_uppersand = var_type.find("&");
412     if(pos_uppersand != std::string::npos)
413       var_type.erase(pos_uppersand - 1, var_type.size());
414
415     if(n->option_vect_->find("private") != n->option_vect_->end()) {
416       if(n->option_vect_->find("private")->second.find(var_decl->
    getNameAsString())
417             != n->option_vect_->find("private")->second.end()
418             || var_type.find("*") != std::string::npos){
419
420         text_constructor_params << var_type << "␣" << var_decl->
    getNameAsString();
421         text_class_var << var_type << "␣" << var_decl->getNameAsString() <<
    "_;\n";
422
423       }else{
424         text_constructor_params << var_type << "␣&␣" << var_decl->
    getNameAsString();
425         text_class_var << var_type << "␣&␣" << var_decl->getNameAsString()
    << "_;\n";
426       }
427     }else if(var_type.find("*") != std::string::npos) {
428       text_constructor_params << var_type << "␣" << var_decl->
    getNameAsString();
429       text_class_var << var_type << "␣" << var_decl->getNameAsString() << "_
    ;\n";
430
431     }else {
432       text_constructor_params << var_type << "␣&␣" << var_decl->
    getNameAsString();
433       text_class_var << var_type << "␣&␣" << var_decl->getNameAsString() <<
    "_;\n";
434     }
435     std::cout << var_type << std::endl;
436
437     text_constructor << ",␣" << var_decl->getNameAsString() << "_(" <<
    var_decl->getNameAsString() << ")␣";
438     text_fx_var << var_decl->getNameAsString() << "_";
439     text_constructor_var << var_decl->getNameAsString();
440   }
441
442   text << text_constructor_params.str() << ")␣" << text_constructor.str()
443       << "{}\n" << text_class_var.str() << "\n";
444
445   unsigned stmt_start_line = utils::Line(s->getLocStart(), sm);
446
447   if(text_constructor_params.str().compare("") == 0)
448       text << "void␣fx(ForParameter␣for_param)";
449     else
450       text << "void␣fx(ForParameter␣for_param,␣" << text_constructor_params.
    str() <<")";
```

```
451
452   unsigned stmt_end_line = utils::Line(s->getLocEnd(), sm);
453   if(n->for_node_ != NULL) {
454
455     std::string text_for;
456     text_for = RewriteOMPFor(n);
457
458     text << "␣{\n" << text_for;
459     clang::SourceLocation for_src_loc = sm.translateLineCol(sm.getMainFileID
      (), stmt_start_line + 1, 1);
460     rewrite_pragma_.InsertText(for_src_loc, text.str(), true, false);
461     rewrite_pragma_.InsertText(stmt_start_src_loc, "//", true, false);
462
463     clang::SourceLocation for_end_src_loc = sm.translateLineCol(sm.
      getMainFileID(), stmt_end_line + 1, 1);
464     rewrite_pragma_.InsertText(for_end_src_loc, "launch_todo_job();␣\n␣}\n",
       true, false);
465   }else {
466     rewrite_pragma_.InsertText(stmt_start_src_loc, text.str(), true, true);
467     //clang::SourceLocation stmt_end_src_loc = sm.translateLineCol(sm.
      getMainFileID(), stmt_end_line - 1, 1);
468
469     rewrite_pragma_.InsertText(s->getLocEnd(), "launch_todo_job();␣\n", true
      , false);
470   }
471
472   /* Comment the pragma */
473   clang::SourceLocation pragma_src_loc = sm.translateLineCol(sm.
      getMainFileID(), stmt_start_line - 1, 1);
474   rewrite_pragma_.InsertText(pragma_src_loc, "//", true, false);
475
476 /*
477  * ----- Insert after pragma ----
478  */
479
480   std::stringstream text_after_pragma;
481   text_after_pragma <<"\
482 void callme(ForParameter for_param) {\n";
483
484 ␣␣if(text_fx_var.str().compare("")␣==␣0)
485 ␣␣␣␣text_after_pragma␣<<␣"  fx(for_param);\n";
486 ␣␣else
487 ␣␣␣␣text_after_pragma␣<<␣"  fx(for_param, "␣<<␣text_fx_var.str()␣<<␣");\n";
488
489 text_after_pragma␣<<
490 "}\n\
491 };\n\
492 std::shared_ptr<NestedBase> nested_b = std::make_shared<Nested>("␣<<␣n->
      getStartLine();
493 if(text_constructor_var.str().compare("")␣!=␣0)
494 ␣␣␣␣␣␣text_after_pragma␣<<␣", ";
495 text_after_pragma␣<<␣text_constructor_var.str()␣<<");\n";
496 text_after_pragma␣<<
497 "if(ThreadPool::getInstance(\"" << utils::FileName(s->getLocStart(), sm) <<
      "\")->call(nested_b))␣\n";
498
499   std::cout << "classname␣" << pragma_name << std::endl;
500
501   if(pragma_name.compare("OMPParallelDirective") == 0 || pragma_name.compare
```

```
    ("OMPForDirective") == 0) {

text_after_pragma << "  nested_b->callme(ForParameter(0,1));\n";
  }else {
text_after_pragma << "  todo_job_.push(nested_b); \n";
  }
text_after_pragma << "}\n";

/* If ForDirective no need to add the if, cause everything is solved inside
    */
  stmt_end_line = utils::Line(s->getLocEnd(), sm);
  clang::SourceLocation pragma_end_src_loc = sm.translateLineCol(sm.
   getMainFileID(), stmt_end_line + 1, 1);

  rewrite_pragma_.InsertText(pragma_end_src_loc, text_after_pragma.str(),
   true, false);

}

Node *TransformRecursiveASTVisitor::GetNodeObjForPragma(clang::Stmt *s){

  clang::SourceLocation stmt_start_src_loc = s->getLocStart();
  unsigned stmt_start_line = utils::Line(stmt_start_src_loc, sm);

  std::vector<Root *>::iterator root_itr;
  for(root_itr = root_vect_->begin(); root_itr != root_vect_->end();
   root_itr ++) {
    if((*root_itr)->getFunctionLineStart() < utils::Line(stmt_start_src_loc,
    sm)
        && (*root_itr)->getFunctionLineEnd() > utils::Line(
   stmt_start_src_loc, sm))

      break;
  }

  std::vector<Node *>::iterator node_itr;
  Node * n;
  for(node_itr = (*root_itr)->children_vect_->begin();
      node_itr != (*root_itr)->children_vect_->end();
      node_itr ++) {

    n = RecursiveGetNodeObjforPragma(*node_itr, stmt_start_line);
    if(n != NULL)
      return n;
  }
  return NULL;
}

Node *TransformRecursiveASTVisitor::RecursiveGetNodeObjforPragma(Node *n,
   unsigned stmt_start_line) {
  Node *nn;
  if(n->getStartLine() == stmt_start_line){
        return n;
  }else if(n->children_vect_ != NULL) {
    for(std::vector<Node *>::iterator node_itr = n->children_vect_->begin();
        node_itr != n->children_vect_->end(); ++ node_itr) {

      nn = RecursiveGetNodeObjforPragma(*node_itr, stmt_start_line);
      if(nn != NULL)
```

```
553         return nn;
554       }
555     }
556     return NULL;
557 }
558 
559 
560 std::string TransformRecursiveASTVisitor::RewriteOMPFor(Node *n) {
561 
562     std::stringstream text_for;
563 
564     ForNode *for_node = n->for_node_;
565 
566     /* for( int i = a + for_param->thread_id_ *(b - a)/ num_threads_; .... */
567     text_for << "for(" << for_node->loop_var_type_ << " " << for_node->
        loop_var_ << " = ";
568     if(for_node->loop_var_init_val_set_)
569       text_for << for_node->loop_var_init_val_;
570     else
571       text_for << for_node->loop_var_init_var_;
572 
573     text_for << " + for_param.thread_id_*(";
574     if(for_node->condition_val_set_)
575       text_for << for_node->condition_val_ << " - ";
576     else
577       text_for << for_node->condition_var_ << " - ";
578 
579     if(for_node->loop_var_init_val_set_)
580       text_for << for_node->loop_var_init_val_;
581     else
582       text_for << for_node->loop_var_init_var_;
583 
584     text_for << ")/for_param.num_threads_; ";
585 
586 
587     /* ....; i < a + (for_param->thread_id_ + 1)*(b - a)/ num_threads_; ... */
588     text_for << for_node->loop_var_ << " " << for_node->condition_op_ << " ";
589 
590     if(for_node->loop_var_init_val_set_)
591       text_for << for_node->loop_var_init_val_;
592     else
593       text_for << for_node->loop_var_init_var_;
594 
595     text_for << " + (for_param.thread_id_ + 1)*(";
596     if(for_node->condition_val_set_)
597       text_for << for_node->condition_val_ << " - ";
598     else
599       text_for << for_node->condition_var_ << " - ";
600 
601     if(for_node->loop_var_init_val_set_)
602       text_for << for_node->loop_var_init_val_;
603     else
604       text_for << for_node->loop_var_init_var_;
605 
606     text_for << ")/for_param.num_threads_; ";
607 
608 
609     /* ...; i ++) */
610     text_for << for_node->loop_var_ << " " << for_node->increment_op_ << " ";
```

```
611   if(for_node->increment_val_set_)
612     text_for << for_node->increment_val_;
613   else
614     text_for << for_node->increment_var_;
615 
616   /* Guarantee that a "{" is inserted at the end of the for declaration line
        if necessary */
617   clang::SourceLocation for_src_loc = for_node->for_stmt_->getLocStart();
618   std::string for_string = sm.getCharacterData(for_src_loc);
619   size_t ext = for_string.find_first_of("\n");
620   for_string = for_string.substr(0, ext);
621 
622   ext = for_string.rfind("{");
623   if (ext == std::string::npos)
624     text_for << ")\n";
625   else
626     text_for << ") { \n";
627 
628   return text_for.str();
629 
630 }
```

Code 1.10: pragma_handler/Node.cpp

```
1  #include "pragma_handler/Node.h"
2 
3  Node::Node(clang::OMPExecutableDirective *pragma_stmt, clang::FunctionDecl *
       funct_decl, clang::SourceManager& sm){
4 
5    option_vect_ = new std::map<std::string, VarList_>();
6    pragma_stmt_ = pragma_stmt;
7 
8    if(pragma_stmt->getAssociatedStmt()) {
9      if(strcmp(pragma_stmt->getStmtClassName(), "OMPParallelDirective") == 0
       && utils::Line(pragma_stmt->getAssociatedStmt()->getLocStart(), sm) ==
       utils::Line(pragma_stmt->getAssociatedStmt()->getLocEnd(), sm)){
10       setPragmaClauses(sm);
11       pragma_stmt_ = static_cast<clang::OMPExecutableDirective *>(
       static_cast<clang::CapturedStmt *>(pragma_stmt->getAssociatedStmt())->
       getCapturedStmt());
12     }
13   }
14   setSourceLocation(sm);
15   setParentFunction(funct_decl, sm);
16   setPragmaClauses(sm);
17 
18   children_vect_ = new std::vector<Node *>();
19 
20   if(strcmp(pragma_stmt_->getStmtClassName(), "OMPForDirective") == 0) {
21     clang::ForStmt *for_stmt = static_cast<clang::ForStmt *>(static_cast<
       clang::CapturedStmt *>(pragma_stmt_->getAssociatedStmt())->
       getCapturedStmt());
22     for_node_ = new ForNode(for_stmt);
23   } else
24     for_node_ = NULL;
25 }
26 
27 void Node::setSourceLocation(const clang::SourceManager& sm) {
28 
29   clang::Stmt *s = pragma_stmt_;
```

```
30    if(pragma_stmt_->getAssociatedStmt())
31      s = static_cast<clang::CapturedStmt *>(pragma_stmt_->getAssociatedStmt()
      )->getCapturedStmt();
32
33    file_name_ = utils::FileName(pragma_stmt_->getLocStart(), sm);
34    if(s != NULL) {
35      start_line_ = utils::Line(s->getLocStart(), sm);
36      start_column_ = utils::Column(s->getLocStart(), sm);
37
38      end_line_ = utils::Line(s->getLocEnd(), sm);
39      end_column_ = utils::Column(s->getLocEnd(), sm);
40
41    } else {
42      start_line_ = utils::Line(pragma_stmt_->getLocStart(), sm);
43      start_column_ = utils::Column(pragma_stmt_->getLocStart(), sm);
44
45      end_line_ = utils::Line(pragma_stmt_->getLocEnd(), sm);
46      end_column_ = utils::Column(pragma_stmt_->getLocEnd(), sm);
47    }
48    return;
49  }
50
51  void Node::setParentFunction(clang::FunctionDecl *funct_decl, const clang::
      SourceManager& sm) {
52
53    parent_funct_info_.function_decl_ = funct_decl;
54    parent_funct_info_.function_start_line_ =  utils::Line(funct_decl->
      getLocStart(), sm);
55    parent_funct_info_.function_end_line_ = utils::Line(funct_decl->getLocEnd
      (), sm);
56
57    /* Name of the function containing the pragma */
58    parent_funct_info_.function_name_ = funct_decl->getNameInfo().getAsString
      ();
59
60    /* Return type of the function containing the pragma */
61    parent_funct_info_.function_return_type_ = funct_decl->getResultType().
      getAsString();
62
63    /* Parameters of the function containing the pragma */
64    parent_funct_info_.num_params_ = funct_decl->getNumParams();
65    parent_funct_info_.function_parameters_ = new std::string*[
      parent_funct_info_.num_params_];
66
67    for(int i = 0; i < parent_funct_info_.num_params_; i ++) {
68      parent_funct_info_.function_parameters_[i] = new std::string[2];
69
70      const clang::ValueDecl *value_decl = static_cast<const clang::ValueDecl
      *>(funct_decl->getParamDecl(i));
71      parent_funct_info_.function_parameters_[i][0] = value_decl->getType().
      getAsString();
72
73      const clang::NamedDecl *named_decl = static_cast<const clang::NamedDecl
      *>(funct_decl->getParamDecl(i));
74      parent_funct_info_.function_parameters_[i][1] = named_decl->
      getNameAsString();
75    }
76
77    /* If the parent function is declared in a class return the name of the
```

```
    class */
/*  if (clang::CXXMethodDecl *cxxMethodD = dynamic_cast<clang::CXXMethodDecl
    *>(funct_decl)){
      const clang::NamedDecl *nD = static_cast<const clang::NamedDecl *>(
    cxxMethodD->getParent());
      parent_funct_info_.parentFunctionClassName = nD->
    getQualifiedNameAsString();
  }
*/
    parent_funct_info_.function_class_name_ = "";
}

void Node::setPragmaClauses(clang::SourceManager& sm) {

  pragma_type_ = pragma_stmt_->getStmtClassName();
/*
 * ---- Exctract pragma options ----
 */
  clang::OMPClause *omp_clause = NULL;
  const char * clause_name;
  unsigned num_clauses = pragma_stmt_->getNumClauses();

  for(unsigned i = 0; i < num_clauses; i ++) {
    omp_clause = pragma_stmt_->getClause(i);
    clause_name = getOpenMPClauseName(omp_clause->getClauseKind());
    VarList_ *var_list = new VarList_;

    if(strcmp(clause_name, "shared") == 0 || strcmp(clause_name, "private")
   == 0 || strcmp(clause_name, "firstprivate") == 0) {

      for(clang::StmtRange stmt_range = omp_clause->children(); stmt_range;
    ++ stmt_range) {
        const clang::DeclRefExpr *decl_ref_expr = static_cast<const clang::
    DeclRefExpr *>(*stmt_range);
        if(decl_ref_expr) {
          const clang::NamedDecl *named_decl = decl_ref_expr->getFoundDecl()
    ;
          const clang::ValueDecl *value_decl = decl_ref_expr->getDecl();
          var_list->insert(std::pair<std::string, std::string>(named_decl->
    getNameAsString(), value_decl->getType().getAsString()));
        }
      }
    }else if(strcmp(clause_name, "period") == 0) {

      clang::OMPPeriodClause *omp_peroid_clause = static_cast<clang::
    OMPPeriodClause *>(omp_clause);
      const clang::IntegerLiteral *int_literal = static_cast<const clang::
    IntegerLiteral *>(omp_peroid_clause->getPeriodValue());
      char period_val[100];
      sprintf(period_val, "%lu", int_literal->getValue().getZExtValue());
      var_list->insert(std::pair<std::string, std::string>(period_val, ""));
    }else {
      var_list->insert(std::pair<std::string, std::string>("", ""));
    }

    option_vect_->insert(std::pair<std::string, VarList_>(clause_name, *
    var_list));
  }
}
```

```
125
126 void Node::CreateXMLPragmaNode(tinyxml2::XMLDocument *xml_doc, tinyxml2::
      XMLElement *pragmas_element) {
127
128   tinyxml2::XMLElement *pragma_element = xml_doc->NewElement("Pragma");
129   pragmas_element->InsertEndChild(pragma_element);
130
131   tinyxml2::XMLElement *name_element = xml_doc->NewElement("Name");
132   pragma_element->InsertEndChild(name_element);
133
134   tinyxml2::XMLText* name_text = xml_doc->NewText(pragma_type_.c_str());
135   name_element->InsertEndChild(name_text);
136
137   tinyxml2::XMLElement *position_element = xml_doc->NewElement("Position");
138
139   if(option_vect_->size() != 0) {
140     tinyxml2::XMLElement *options_element = xml_doc->NewElement("Options");
141     pragma_element->InsertEndChild(options_element);
142
143     CreateXMLPragmaOptions(xml_doc, options_element);
144
145     pragma_element->InsertAfterChild(options_element, position_element);
146   } else {
147
148 /*
149  * ---- Position ----
150  */
151     pragma_element->InsertEndChild(position_element);
152   }
153
154   tinyxml2::XMLElement *start_line_element = xml_doc->NewElement("StartLine"
      );
155   position_element->InsertEndChild(start_line_element);
156   char start_line[100];
157   sprintf(start_line, "%d", start_line_);
158   tinyxml2::XMLText* start_line_text = xml_doc->NewText(start_line);
159   start_line_element->InsertEndChild(start_line_text);
160
161   tinyxml2::XMLElement *end_line_element = xml_doc->NewElement("EndLine");
162   position_element->InsertEndChild(end_line_element);
163   char end_line[100];
164   sprintf(end_line, "%d", end_line_);
165   tinyxml2::XMLText* end_line_text = xml_doc->NewText(end_line);
166   end_line_element->InsertEndChild(end_line_text);
167
168 /*
169  * ----- If present insert info of the For stmt ----
170  */
171   if(for_node_) {
172     tinyxml2::XMLElement *for_element = xml_doc->NewElement("For");
173     pragma_element->InsertEndChild(for_element);
174     for_node_->CreateXMLPragmaFor(xml_doc, for_element);
175   }
176
177   if(children_vect_->size() != 0) {
178     tinyxml2::XMLElement *nesting_element = xml_doc->NewElement("Children");
179     pragma_element->InsertEndChild(nesting_element);
180     tinyxml2::XMLElement *new_pragmas_element = xml_doc->NewElement("Pragmas
      ");
```

```
    nesting_element->InsertEndChild(new_pragmas_element);
    for(std::vector<Node *>::iterator node_itr = children_vect_->begin();
    node_itr != children_vect_->end(); ++node_itr) {
      (*node_itr)->CreateXMLPragmaNode(xml_doc, new_pragmas_element);
    }
  }
}

void Node::CreateXMLPragmaOptions(tinyxml2::XMLDocument *xml_doc, tinyxml2::
    XMLElement *options_element) {
  if(option_vect_->size() != 0) {

    for(std::map<std::string, VarList_>::iterator options_itr = option_vect_
    ->begin(); options_itr != option_vect_->end(); ++ options_itr) {

      tinyxml2::XMLElement *option_element = xml_doc->NewElement("Option");
      options_element->InsertEndChild(option_element);

      tinyxml2::XMLElement *option_name_element = xml_doc->NewElement("Name"
    );
      option_element->InsertEndChild(option_name_element);
      tinyxml2::XMLText* name_opt_text = xml_doc->NewText((*options_itr).
    first.c_str());
      option_name_element->InsertEndChild(name_opt_text);

      if((*options_itr).second.size() != 0) {
        for(std::map<std::string, std::string>::iterator var_itr = (*
    options_itr).second.begin(); var_itr != (*options_itr).second.end(); ++
    var_itr) {
          tinyxml2::XMLElement *parameter_element = xml_doc->NewElement("
    Parameter");
          option_element->InsertEndChild(parameter_element);

          if(strcmp((*var_itr).first.c_str(), "") != 0) {
            tinyxml2::XMLElement *type_element = xml_doc->NewElement("Type")
    ;
            tinyxml2::XMLText* type_text = xml_doc->NewText((*var_itr).
    second.c_str());
            type_element->InsertEndChild(type_text);
            parameter_element->InsertEndChild(type_element);
          }
          tinyxml2::XMLElement *name_element = xml_doc->NewElement("Var");
          tinyxml2::XMLText* name_text = xml_doc->NewText((*var_itr).first.
    c_str());
          name_element->InsertEndChild(name_text);
          parameter_element->InsertEndChild(name_element);

        }
      }
    }
  }
}
```

Code 1.11: pragma_handler/ForNode.cpp

```
#include "pragma_handler/ForNode.h"


ForNode::ForNode(clang::ForStmt *for_stmt) {
  loop_var_type_ = "";
```

```
  loop_var_init_val_set_ = false;

  loop_var_init_var_ = "";

  condition_val_set_ = false;
  condition_var_ = "";

  increment_val_set_ = false;
  increment_var_ = "";

  for_stmt_ = for_stmt;
  ExtractForParameters(for_stmt);
}

void ForNode::ExtractForParameters(clang::ForStmt *for_stmt) {

  ExtractForInitialization(for_stmt);
  ExtractForCondition(for_stmt);
  ExtractForIncrement(for_stmt);

}

void ForNode::ExtractForInitialization(clang::ForStmt *for_stmt) {
/*
* Initialization of the loop variable
*/

  /* for(int i = ....) */
  if(strcmp(for_stmt->child_begin()->getStmtClassName(), "DeclStmt") == 0) {
    const clang::DeclStmt *decl_stmt = static_cast<const clang::DeclStmt
   *>(*(for_stmt->child_begin()));
    const clang::Decl *decl = decl_stmt->getSingleDecl();

    /* Return the name of the variable */
    const clang::NamedDecl *named_decl = static_cast<const clang::NamedDecl
   *>(decl);
    loop_var_ = named_decl->getNameAsString();

    /* Return the type of the variable */
    const clang::ValueDecl *vale_decl = static_cast<const clang::ValueDecl
   *>(named_decl);
    loop_var_type_ = vale_decl->getType().getAsString();

    /* for (... = 0) */
    if(strcmp(decl_stmt->child_begin()->getStmtClassName(), "IntegerLiteral"
   ) == 0) {
      const clang::IntegerLiteral *int_literal = static_cast<const clang::
   IntegerLiteral *>(*(decl_stmt->child_begin()));
      loop_var_init_val_ = int_literal->getValue().getZExtValue();
      loop_var_init_val_set_ = true;

    /* for (... = a) */
    }else if (strcmp(decl_stmt->child_begin()->getStmtClassName(), "
   ImplicitCastExpr") == 0) {
      const clang::DeclRefExpr *decl_ref_expr = static_cast<const clang::
   DeclRefExpr *>(*(decl_stmt->child_begin()->child_begin()));
      const clang::NamedDecl *named_decl = decl_ref_expr->getFoundDecl();
      loop_var_init_var_ = named_decl->getNameAsString();
    }
```

```
58
59    /* for ( i = ...) */
60    }else if(strcmp(for_stmt->child_begin()->getStmtClassName(), "
      BinaryOperator") == 0) {
61      const clang::BinaryOperator *binary_op = static_cast<const clang::
      BinaryOperator *>(*(for_stmt->child_begin()));
62      const clang::DeclRefExpr *decl_ref_expr = static_cast<const clang::
      DeclRefExpr *>(*(binary_op->child_begin()));
63
64      //Return the name of the variable
65      const clang::NamedDecl *named_decl = decl_ref_expr->getFoundDecl();
66      loop_var_ = named_decl->getNameAsString();
67
68      /* for( ... = 0) */
69      clang::ConstStmtIterator stmt_itr = binary_op->child_begin();
70      stmt_itr ++;
71      if(strcmp(stmt_itr->getStmtClassName(), "IntegerLiteral") == 0) {
72        const clang::IntegerLiteral *int_literal = static_cast<const clang::
      IntegerLiteral *>(*stmt_itr);
73        loop_var_init_val_ = int_literal->getValue().getZExtValue();
74        loop_var_init_val_set_ = true;
75
76      /* for ( ... = a) */
77      } else if (strcmp(stmt_itr->getStmtClassName(), "ImplicitCastExpr") ==
      0) {
78        const clang::DeclRefExpr *decl_ref_expr = static_cast<const clang::
      DeclRefExpr *>(*(stmt_itr->child_begin()));
79        const clang::NamedDecl *named_decl = decl_ref_expr->getFoundDecl();
80        loop_var_init_var_ = named_decl->getNameAsString();
81      }
82    }
83  }
84
85  void ForNode::ExtractForCondition(clang::ForStmt *for_stmt) {
86
87    const clang::Expr *condition_expr = for_stmt->getCond();
88    const clang::BinaryOperator *binary_op = static_cast<const clang::
      BinaryOperator *>(condition_expr);
89
90    /* Conditional funcion */
91    condition_op_ = binary_op->getOpcodeStr();
92
93    /* Conditional value */
94    const clang::Expr *right_expr = binary_op->getRHS();
95
96    if(strcmp(right_expr->getStmtClassName(), "IntegerLiteral") == 0) {
97      const clang::IntegerLiteral *int_literal = static_cast<const clang::
      IntegerLiteral *>(right_expr);
98      condition_val_ = int_literal->getValue().getZExtValue();
99      condition_val_set_ = true;
100
101   }else if(strcmp(right_expr->getStmtClassName(), "ImplicitCastExpr") == 0)
      {
102     const clang::DeclRefExpr *decl_ref_expr = static_cast<const clang::
      DeclRefExpr *>(*(right_expr->child_begin()));
103     const clang::NamedDecl *named_decl = decl_ref_expr->getFoundDecl();
104
105 /*
106  * ---- PROBLEM: If the variable is not defined inside the block (which
```

```
    block?)
107  * ----           the NameDecl * is != NULL, but when you try to exctract the
       name -> segmentation fault!!
108  */
109     condition_var_ = named_decl->getNameAsString();
110
111   }
112 }
113
114 void ForNode::ExtractForIncrement(clang::ForStmt *for_stmt) {
115
116   const clang::Expr *increment_expr = for_stmt->getInc();
117
118   if(strcmp(increment_expr->getStmtClassName(), "UnaryOperator") == 0) {
119     const clang::UnaryOperator *unary_op =  static_cast<const clang::
      UnaryOperator *>(increment_expr);
120     increment_op_ = unary_op->getOpcodeStr(unary_op->getOpcode());
121
122   }else if(strcmp(increment_expr->getStmtClassName(), "
      CompoundAssignOperator") == 0) {
123     const clang::CompoundAssignOperator *compound_op = static_cast<const
      clang::CompoundAssignOperator *>(increment_expr);
124     increment_op_ = compound_op->getOpcodeStr();
125     const clang::Expr *right_expr = compound_op->getRHS();
126
127     if(strcmp(right_expr->getStmtClassName(), "IntegerLiteral") == 0) {
128       const clang::IntegerLiteral *int_literal = static_cast<const clang::
      IntegerLiteral *>(right_expr);
129       increment_val_ = int_literal->getValue().getZExtValue();
130       increment_val_set_ = true;
131
132     }else if(strcmp(right_expr->getStmtClassName(), "ImplicitCastExpr") ==
      0) {
133       const clang::DeclRefExpr *decl_ref_expr = static_cast<const clang::
      DeclRefExpr *>(*(right_expr->child_begin()));
134       const clang::NamedDecl *named_decl = decl_ref_expr->getFoundDecl();
135       increment_var_ = named_decl->getNameAsString();
136     }
137   }
138
139 }
140
141 void ForNode::CreateXMLPragmaFor(tinyxml2::XMLDocument *xml_doc, tinyxml2::
      XMLElement *for_element) {
142
143 /*
144  * ----- DECLARATION -----
145  */
146   tinyxml2::XMLElement *declaration_element = xml_doc->NewElement("
      Declaration");
147   for_element->InsertEndChild(declaration_element);
148
149   tinyxml2::XMLElement *type_element = xml_doc->NewElement("Type");
150   declaration_element->InsertEndChild(type_element);
151   tinyxml2::XMLText* type_text = xml_doc->NewText(loop_var_type_.c_str());
152   type_element->InsertEndChild(type_text);
153
154   tinyxml2::XMLElement *loop_var_element = xml_doc->NewElement("LoopVariable
      ");
```

```
155   declaration_element->InsertEndChild(loop_var_element);
156   tinyxml2::XMLText* loop_var_text = xml_doc->NewText(loop_var_.c_str());
157   loop_var_element->InsertEndChild(loop_var_text);
158
159   if(loop_var_init_val_set_ == true) {
160     tinyxml2::XMLElement *init_val_element = xml_doc->NewElement("InitValue"
      );
161     declaration_element->InsertEndChild(init_val_element);
162     char loop_var_init_val[100];
163     sprintf(loop_var_init_val, "%d", loop_var_init_val_);
164     tinyxml2::XMLText* init_val_text = xml_doc->NewText(loop_var_init_val);
165     init_val_element->InsertEndChild(init_val_text);
166   }else {
167     tinyxml2::XMLElement *init_var_element = xml_doc->NewElement("
     InitVariable");
168     declaration_element->InsertEndChild(init_var_element);
169     tinyxml2::XMLText* init_var_text = xml_doc->NewText(loop_var_init_var_.
     c_str());
170     init_var_element->InsertEndChild(init_var_text);
171   }
172
173 /*
174  * ---- CONDITION -----
175  */
176   tinyxml2::XMLElement *condition_element = xml_doc->NewElement("Condition")
      ;
177   for_element->InsertAfterChild(declaration_element, condition_element);
178
179   tinyxml2::XMLElement *condition_op_element = xml_doc->NewElement("Op");
180   condition_element->InsertEndChild(condition_op_element);
181   tinyxml2::XMLText* condition_op_text = xml_doc->NewText(condition_op_.
      c_str());
182   condition_op_element->InsertEndChild(condition_op_text);
183
184   if(condition_val_set_ == true) {
185     tinyxml2::XMLElement *condition_val_element = xml_doc->NewElement("
     ConditionValue");
186     condition_element->InsertEndChild(condition_val_element);
187     char condition_val[100];
188     sprintf(condition_val, "%d", condition_val_);
189     tinyxml2::XMLText* condition_val_text = xml_doc->NewText(condition_val);
190     condition_val_element->InsertEndChild(condition_val_text);
191
192   }else {
193     tinyxml2::XMLElement *condition_var_element = xml_doc->NewElement("
     ConditionVariable");
194     condition_element->InsertEndChild(condition_var_element);
195     tinyxml2::XMLText* condition_var_text = xml_doc->NewText(condition_var_.
     c_str());
196     condition_var_element->InsertEndChild(condition_var_text);
197   }
198
199 /*
200  * ---- INCREMENT ----
201  */
202   tinyxml2::XMLElement *increment_element = xml_doc->NewElement("Increment")
      ;
203   for_element->InsertAfterChild(condition_element, increment_element);
204
```

```
205   tinyxml2::XMLElement *increment_op_element = xml_doc->NewElement("Op");
206   increment_element->InsertEndChild(increment_op_element);
207   tinyxml2::XMLText* increment_op_text = xml_doc->NewText(increment_op_.
       c_str());
208   increment_op_element->InsertEndChild(increment_op_text);
209
210   if(increment_val_set_ == true) {
211     tinyxml2::XMLElement *increment_val_element = xml_doc->NewElement("
      IncrementValue");
212     increment_element->InsertEndChild(increment_val_element);
213     char increment_val[100];
214     sprintf(increment_val, "%d", increment_val_);
215     tinyxml2::XMLText* increment_val_text = xml_doc->NewText(increment_val);
216     increment_val_element->InsertEndChild(increment_val_text);
217
218   }else if(increment_var_.compare("") != 0) {
219     tinyxml2::XMLElement *increment_var_element = xml_doc->NewElement("
      IncrementVariable");
220     increment_element->InsertEndChild(increment_var_element);
221     tinyxml2::XMLText* increment_var_text = xml_doc->NewText(increment_var_.
      c_str());
222     increment_var_element->InsertEndChild(increment_var_text);
223   }
224
225
226 }
```

Code 1.12: pragma_handler/Root.cpp

```
1  #include "pragma_handler/Root.h"
2
3  Root::Root(Node *n, FunctionInfo funct_info) {
4
5    children_vect_ = new std::vector<Node *>();
6    children_vect_->push_back(n);
7
8    last_node_ = n;
9    function_info_ = funct_info;
10 }
11
12 void Root::CreateXMLFunction(tinyxml2::XMLDocument *xml_doc) {
13
14   tinyxml2::XMLElement *function_element = xml_doc->NewElement("Function");
15   xml_doc->LastChild()->InsertEndChild(function_element);
16
17   tinyxml2::XMLElement *name_element = xml_doc->NewElement("Name");
18   function_element->InsertEndChild(name_element);
19   tinyxml2::XMLText* name_text = xml_doc->NewText(function_info_.
      function_name_.c_str());
20   name_element->InsertEndChild(name_text);
21
22   if(function_info_.function_class_name_.compare("") != 0){
23     tinyxml2::XMLElement *class_name_element = xml_doc->NewElement("
      ClassName");
24     function_element->InsertEndChild(class_name_element);
25     tinyxml2::XMLText* class_name_text = xml_doc->NewText(function_info_.
      function_class_name_.c_str());
26     class_name_element->InsertEndChild(class_name_text);
27   }
28
```

```
29    tinyxml2::XMLElement *return_type_element = xml_doc->NewElement("
       ReturnType");
30    function_element->InsertEndChild(return_type_element);
31    tinyxml2::XMLText* return_type_text = xml_doc->NewText(function_info_.
       function_return_type_.c_str());
32    return_type_element->InsertEndChild(return_type_text);
33
34    if(function_info_.num_params_ > 0) {
35      tinyxml2::XMLElement *parameters_element = xml_doc->NewElement("
       Parameters");
36      function_element->InsertEndChild(parameters_element);
37
38      for(int i = 0; i < function_info_.num_params_; i ++) {
39        tinyxml2::XMLElement *parameter_element = xml_doc->NewElement("
       Parameter");
40        parameters_element->InsertEndChild(parameter_element);
41
42        tinyxml2::XMLElement *type_element = xml_doc->NewElement("Type");
43        parameter_element->InsertEndChild(type_element);
44        tinyxml2::XMLText* param_type_text = xml_doc->NewText(function_info_.
       function_parameters_[i][0].c_str());
45        type_element->InsertEndChild(param_type_text);
46
47        tinyxml2::XMLElement *param_name_element = xml_doc->NewElement("Name")
       ;
48        parameter_element->InsertEndChild(param_name_element);
49        tinyxml2::XMLText* param_name_text = xml_doc->NewText(function_info_.
       function_parameters_[i][1].c_str());
50        param_name_element->InsertEndChild(param_name_text);
51
52      }
53    }
54
55    tinyxml2::XMLElement *line_element = xml_doc->NewElement("Line");
56    function_element->InsertEndChild(line_element);
57    char line[100];
58    sprintf(line, "%d", function_info_.function_start_line_);
59    tinyxml2::XMLText* line_text = xml_doc->NewText(line);
60    line_element->InsertEndChild(line_text);
61
62
63    tinyxml2::XMLElement *pragmas_element = xml_doc->NewElement("Pragmas");
64    function_element->InsertEndChild(pragmas_element);
65
66    for(std::vector<Node *>::iterator node_itr = children_vect_->begin();
       node_itr != children_vect_->end(); ++ node_itr) {
67      (*node_itr)->CreateXMLPragmaNode(xml_doc, pragmas_element);
68    }
69  }
```

Code 1.13: pragma_handler/create_tree.cpp

```
1  #include "pragma_handler/create_tree.h"
2
3  std::vector<Root *> *CreateTree(std::vector<clang::OMPExecutableDirective *>
       *pragma_list,
4                                  std::vector<clang::FunctionDecl *> *function_list, clang::
       SourceManager &sm) {
5
6    clang::FunctionDecl *function_decl = NULL;
```

```
  clang::FunctionDecl *function_decl_tmp = NULL;
  std::vector<Root *> *root_vect = new std::vector<Root *>();

  std::vector<clang::OMPExecutableDirective *>::iterator omp_itr;

  for(omp_itr = pragma_list->begin(); omp_itr != pragma_list->end(); ++
   omp_itr) {

    function_decl_tmp = GetFunctionForPragma(*omp_itr, function_list, sm);
    Node * n = new Node(*omp_itr, function_decl_tmp, sm);

    /* In case of parallel for skip one stmt.
       Parallel for is represented with two OMPExecutableDirective,
       (OMPParallel + OMPFor) so we have to skip one stmt */
    if((*omp_itr)->getAssociatedStmt()) {
      if(strcmp((*omp_itr)->getStmtClassName(), "OMPParallelDirective") == 0
          && utils::Line((*omp_itr)->getAssociatedStmt()->getLocStart(), sm)
             == utils::Line((*omp_itr)->getAssociatedStmt()->getLocEnd(), sm
   )) {
        n->pragma_type_ = "OMPParallelForDirective";
        omp_itr++;
      }
    }

    if(function_decl_tmp != function_decl) {
      function_decl = function_decl_tmp;
      Root *root = new Root(n, n->getParentFunctionInfo());
      n->setParentNode(NULL);
      root->setLastNode(n);
      root_vect->push_back(root);

    }else {
      BuildTree(root_vect->back(), n);
      root_vect->back()->setLastNode(n);
    }
  }
  return root_vect;
}


clang::FunctionDecl *GetFunctionForPragma(clang::OMPExecutableDirective *
   pragma_stmt,
                     std::vector<clang::FunctionDecl *> *function_list,
                     clang::SourceManager &sm) {

  unsigned funct_start_line, funct_end_line;
  unsigned pragma_start_line = utils::Line(pragma_stmt->getLocStart(), sm);
  std::vector<clang::FunctionDecl *>::iterator funct_itr;

  for(funct_itr = function_list->begin(); funct_itr != function_list->end();
    ++ funct_itr) {
    funct_start_line = utils::Line((*funct_itr)->getSourceRange().getBegin()
   , sm);
    funct_end_line = utils::Line((*funct_itr)->getSourceRange().getEnd(), sm
   );
    if(pragma_start_line < funct_end_line && pragma_start_line >
   funct_start_line)
      return (*funct_itr);
  }
```

```
  return NULL;
}


/*
 * ---- Attach the node to the correct parent (if the node is node annidated
     attach it to root) ----
 * THEOREM: A node can be annidated only in its previous node or in the
    father of the previous node or in the father
 *          of the father ..... of the previous node. (This is due to the
    fact that the list of pragmas is ordered based
 *          on starting line of the associated stmt).
 */
void BuildTree(Root *root, Node *n) {

  Node *last_node = root->getLastNode();
  bool annidation;

  while(last_node != NULL) {
    annidation = CheckAnnidation(last_node, n);

    if(annidation == true) {
      last_node->AddChildNode(n);
      n->setParentNode(last_node);
      return;

    }else
      last_node = last_node->getParentNode();
  }

  root->AddChildNode(n);
  n->setParentNode(NULL);
}

/*
 * ---- Check if n is annidated inside parent: to be annidated it is enough
    that n->endLine < parent->endLine
 * (for sure n->startLine < parent->startLine because pragmas are ordered
    based on their starting line)
 */
bool CheckAnnidation(Node *parent, Node *n) {

  if(n->getEndLine() < parent->getEndLine())
    return true;
  else
    return false;

}
```

Code 1.14: utilis/source_locations.cpp

```
#include "utils/source_locations.h"

using namespace std;
using namespace clang;

namespace utils {

string FileName(SourceLocation const& l, SourceManager const& sm) {
  PresumedLoc pl = sm.getPresumedLoc(l);
```

```
  return string(pl.getFilename());
}

string FileId(SourceLocation const& l, SourceManager const& sm) {
  string fn = FileName(l, sm);
  for(size_t i=0; i<fn.length(); ++i)
    switch(fn[i]) {
      case '/':
      case '\\':
      case '>':
      case '.':
        fn[i] = '_';
    }
  return fn;
}

unsigned Line(SourceLocation const& l, SourceManager const& sm) {
  PresumedLoc pl = sm.getPresumedLoc(l);
  return pl.getLine();
}

std::pair<unsigned, unsigned> Line(clang::SourceRange const& r,
    SourceManager const& sm) {
  return std::make_pair(Line(r.getBegin(), sm), Line(r.getEnd(), sm));
}

unsigned Column(SourceLocation const& l, SourceManager const& sm) {
  PresumedLoc pl = sm.getPresumedLoc(l);
  return pl.getColumn();
}

std::pair<unsigned, unsigned> Column(clang::SourceRange const& r,
    SourceManager const& sm) {
  return std::make_pair(Column(r.getBegin(), sm), Column(r.getEnd(), sm));
}

std::string location(clang::SourceLocation const& l, clang::SourceManager
    const& sm) {
  std::string str;
  llvm::raw_string_ostream ss(str);
  l.print(ss,sm);
  return ss.str();
}
}
```

Code 1.15: xml_creator/xml_creator.cpp

```
#include "xml_creator/XMLcreator.h"

void CreateXML(std::vector<Root *> *root_vect, char *file_name) {

  tinyxml2::XMLDocument *xml_doc = new tinyxml2::XMLDocument();
  tinyxml2::XMLElement *file_element = xml_doc->NewElement("File");
  xml_doc->InsertEndChild(file_element);

  tinyxml2::XMLElement *name_element = xml_doc->NewElement("Name");
  tinyxml2::XMLText* name_text = xml_doc->NewText(file_name);
  name_element->InsertEndChild(name_text);
  file_element->InsertEndChild(name_element);

```

```
14
15    for(std::vector<Root *>::iterator root_itr = root_vect->begin(); root_itr
       != root_vect->end(); ++ root_itr)
16      (*root_itr)->CreateXMLFunction(xml_doc);
17
18
19    std::string out_xml_file (file_name);
20    size_t ext = out_xml_file.find_last_of(".");
21    if (ext == std::string::npos)
22      ext = out_xml_file.length();
23    out_xml_file = out_xml_file.substr(0, ext);
24    std::cout << out_xml_file << std::endl;
25
26    out_xml_file.insert(ext, "_pragmas.xml");
27    std::cout << out_xml_file << std::endl;
28
29    xml_doc->SaveFile(out_xml_file.c_str());
30  }
```

1.3 Run-time

1.3.1 Profiler

Code 1.16: profile_tracker.h

```
1  #include <fstream>
2  #include <time.h>
3  #include <iostream>
4  #include <unistd.h>
5
6  #define log_file "log_file.xml"
7
8  struct ProfileTrackParams {
9
10   ProfileTrackParams(int funct_id, int pragma_line)
11     : funct_id_(funct_id) ,pragma_line_(pragma_line), num_for_iteration_set_
      (false) {}
12   /* Costructor for parallel for */
13   ProfileTrackParams(int funct_id, int pragma_line, int n)
14     : funct_id_(funct_id), pragma_line_(pragma_line), num_for_iteration_(n),
       num_for_iteration_set_(true) {}
15
16   int funct_id_;
17   int pragma_line_;
18   /* In the case of a parallel for this variable saves the number of the
      iteration of the for */
19   int num_for_iteration_;
20   bool num_for_iteration_set_;
21 };
22
23 /*
24  * ----- Class that keep track of the children time and the father of the
      current pragma in execution ----
25  */
26 class ProfileTracker {
27
28   clock_t start_time_;
29   clock_t end_time_;
```

```
30
31    int num_for_iteration_;
32    bool num_for_iteration_set_;
33
34    /* These functions print the result of the profiling in a log file */
35    void PrintPragma();
36    void PrintFunction();
37
38  public:
39    int pragma_line_;
40    int funct_id_;
41
42    double elapsed_time_;
43    /* Time spent by the children of the current pragma or function */
44    double children_elapsed_time_;
45
46    /* Keeps track of which function/pragma has invoked the current function/
       pragma */
47    ProfileTracker *previous_pragma_executed_;
48
49    /* In the costructor a timer is started */
50    ProfileTracker(const ProfileTrackParams & p);
51    /* In the destructor the timer is stopped and the elapsed time is written
       in the log file */
52    ~ProfileTracker();
53
54    /* This is necessary to allow to create an object inside the declaration
       of an if stmt */
55    operator bool() const  { return true; }
56  };
57
58  /*
59   * ---- Singleton class that open and close the log file -----
60   */
61  class ProfileTrackerLog {
62
63    /* Keeps track of which function/pragma has invoked the current function/
       pragma */
64    ProfileTracker *current_pragma_executing_;
65
66    /* Create the log file and write in it the hardware spec */
67    ProfileTrackerLog ();
68
69    void WriteArchitecturesSpec();
70    size_t getTotalSystemMemory();
71
72  public:
73    /* File where the log is written */
74    std::ofstream log_file_;
75
76    static ProfileTrackerLog* getInstance();
77    /* Substitute the pointer of the current pragma in execution and return
       the previous value */
78    ProfileTracker *ReplaceCurrentPragma(ProfileTracker *
       current_pragma_executing_);
79
80    /* Save and close the log file */
81    ~ProfileTrackerLog();
82
```

```
};
```

Code 1.17: profile_tracker.cpp

```
#include "profile_tracker/profile_tracker.h"


/*
 * ---- PROFILE TRACKER LOG ----
 */
ProfileTrackerLog::ProfileTrackerLog () {
  current_pragma_executing_ = NULL;
  log_file_.open(log_file);
  log_file_ << "<LogFile>" << std::endl;
  WriteArchitecturesSpec();
}

void ProfileTrackerLog::WriteArchitecturesSpec() {
  log_file_ << "␣␣<Hardware␣";
  log_file_ << "NumberofCores=\"" << std::thread::hardware_concurrency() <<
   "\"␣";
    log_file_ << "MemorySize=\"" << getTotalSystemMemory() << "\"/>" << std
   ::endl;
}

size_t ProfileTrackerLog::getTotalSystemMemory() {
    /*long pages = sysconf(_SC_PHYS_PAGES);
    long page_size = sysconf(_SC_PAGE_SIZE);
    return (pages * page_size)/1024/1024;*/
    return 2000;
}

ProfileTrackerLog* ProfileTrackerLog::getInstance() {
    static ProfileTrackerLog log;
    return &log;
}

ProfileTrackerLog::~ProfileTrackerLog() {
  log_file_ << "</LogFile>" << std::endl;
  log_file_.close();
}

ProfileTracker *ProfileTrackerLog::ReplaceCurrentPragma(ProfileTracker *
   current_pragma_executing) {
  ProfileTracker *tmp = current_pragma_executing_;
  current_pragma_executing_ = current_pragma_executing;
  return tmp;
}


/*
 * ---- PROFILE TRACKER ----
 */
ProfileTracker::ProfileTracker(const ProfileTrackParams & p) {
  previous_pragma_executed_ = ProfileTrackerLog::getInstance()->
   ReplaceCurrentPragma(this);

  children_elapsed_time_ = 0;

  pragma_line_ = p.pragma_line_;
```

```
53    funct_id_ = p.funct_id_;
54    num_for_iteration_set_ = p.num_for_iteration_;
55
56    if(num_for_iteration_set_)
57      num_for_iteration_ = p.num_for_iteration_;
58
59    start_time_ = clock();
60  }
61
62  ProfileTracker::~ProfileTracker() {
63    end_time_ = clock();
64    elapsed_time_ = ((double)(end_time_ - start_time_))/CLOCKS_PER_SEC;
65    if(previous_pragma_executed_) {
66      previous_pragma_executed_->children_elapsed_time_ += elapsed_time_;
67    }
68    ProfileTrackerLog::getInstance()->ReplaceCurrentPragma(
       previous_pragma_executed_);
69
70    if(pragma_line_ == 0)
71      PrintFunction();
72    else
73      PrintPragma();
74
75  }
76
77  void ProfileTracker::PrintPragma() {
78    ProfileTrackerLog::getInstance()->log_file_ << "␣␣<Pragma" \
79                          << "␣fid=\"" << funct_id_ << "\"␣pid=\"" <<
       pragma_line_ << "\"␣";
80    if(previous_pragma_executed_) {
81      if(previous_pragma_executed_->pragma_line_ != 0)
82        ProfileTrackerLog::getInstance()->log_file_ << "callerid=\"" <<
       previous_pragma_executed_->pragma_line_ << "\"␣";
83      else
84        ProfileTrackerLog::getInstance()->log_file_ << "callerid=\"" <<
       previous_pragma_executed_->funct_id_ << "\"␣";
85    }
86    ProfileTrackerLog::getInstance()->log_file_ << "elapsedTime=\"" <<
       elapsed_time_ << "\"␣" \
87                          << "childrenTime=\"" << children_elapsed_time_ << "
       \"";
88    if(num_for_iteration_set_)
89      ProfileTrackerLog::getInstance()->log_file_ << "␣loops=\"" <<
       num_for_iteration_ << "\"";
90    ProfileTrackerLog::getInstance()->log_file_ << "/>" << std::endl;
91
92  }
93
94  void ProfileTracker::PrintFunction() {
95    ProfileTrackerLog::getInstance()->log_file_ << "␣␣<Function" \
96                          << "␣fid=\"" << funct_id_ << "\"␣";
97    if(previous_pragma_executed_) {
98      if(previous_pragma_executed_->pragma_line_ != 0)
99        ProfileTrackerLog::getInstance()->log_file_ << "callerid=\"" <<
       previous_pragma_executed_->pragma_line_ << "\"␣";
100     else
101       ProfileTrackerLog::getInstance()->log_file_ << "callerid=\"" <<
       previous_pragma_executed_->funct_id_ << "\"␣";
102   }
```

```
103   ProfileTrackerLog::getInstance()->log_file_ << "elapsedTime=\"" <<
      elapsed_time_ <<  "\"␣" \
104                         << "childrenTime=\"" << children_elapsed_time_ << "
      \"/>" << std::endl;
105
106
107 }
```

1.3.2 Final exectution

Code 1.18: thread_pool.h

```
1  #include <string>
2  #include <thread>
3  #include <vector>
4  #include <mutex>
5  #include <map>
6  #include <math.h>
7  #include <iostream>
8  #include <condition_variable>
9  #include <queue>
10 #include <exception>
11 #include <sys/time.h>
12
13 #include "xml_creator/tinyxml2.h"
14
15 int chartoint(const char *cc);
16 int chartoint(char *cc);
17
18 class ForParameter {
19 public:
20     const int thread_id_;
21     const int num_threads_;
22     ForParameter(int thread_id, int num_threads) : thread_id_(thread_id),
       num_threads_(num_threads) {}
23 };
24
25 class NestedBase {
26 public:
27
28     NestedBase(int pragma_id) : pragma_id_(pragma_id) {}
29
30     int pragma_id_;
31     std::queue<std::shared_ptr<NestedBase>> todo_job_;
32
33     void launch_todo_job() {
34         while(todo_job_.size() != 0) {
35             todo_job_.front()->callme(ForParameter(0, 1));
36             todo_job_.pop();
37         }
38     }
39
40     virtual void callme(ForParameter for_param) = 0;
41     virtual std::shared_ptr<NestedBase> clone() const = 0;
42 };
43
44 class ThreadPool {
45 public:
46     typedef int Jobid_t;
```

```
47
48      struct Job
49      {
50          std::shared_ptr<NestedBase> nested_base_;
51          ForParameter for_param_;
52          Job(std::shared_ptr<NestedBase> nested_base, ForParameter for_param)
53              : nested_base_(nested_base), for_param_(for_param) {}
54      };
55
56      /* Launches the threads */
57      void init(int pool_size);
58
59      /* Called by the task to be put in the job queue */
60      bool call(std::shared_ptr<NestedBase> nested_base);
61      void call_sections(std::shared_ptr<NestedBase> nested_b);
62      void call_parallel(std::shared_ptr<NestedBase> nested_b);
63      void call_for(std::shared_ptr<NestedBase> nested_b);
64      void call_barrier(std::shared_ptr<NestedBase> nested_b);
65
66      /* Push a job in the job queue */
67      void push(std::shared_ptr<NestedBase> nested_base, ForParameter
        for_param, int thread_id);
68      void push_completed_job(std::shared_ptr<NestedBase> nested_base,
        ForParameter for_param);
69      void push_termination_job(int thread_id);
70
71      /* Pause a thread till the job[job_id] complete */
72      void join(Jobid_t job_id);
73
74      void joinall();
75
76      static ThreadPool* getInstance(std::string file_name);
77
78      /* Map the thread::id to an integer going from 0 to num_thread - 1 */
79      std::map<std::thread::thread::id, int> thread_id_to_int_;
80
81      ~ThreadPool() { joinall(); }
82
83  private:
84      struct ScheduleOptions {
85          int pragma_id_;
86          int caller_id_;
87          /* In case of a parallel for, specify to the job which part of the
        for to execute */
88          int thread_id_;
89          /* Idicates the pragma type: parallel, task, ... */
90          std::string pragma_type_;
91          /* Indicates the threads that have to run the task */
92          std::vector<int> threads_;
93          /* List of pragma_id_ to wait before completing the task */
94          std::vector<int> barriers_;
95      };
96
97      struct JobIn {
98          Job job_;
99          /* ID of the job = pragma line number */
100         Jobid_t job_id_;
101         Jobid_t pragma_id_;
102         /* Pragma type, e.g. OMPParallelDirective, OMPTaskDirective, ... */
```

```
103         std::string job_type_;
104         /* Fix the bug where a thread waits for another thread which already
     nofied to have compleated */
105         bool job_completed_ = false;
106 
107         bool terminated_with_exceptions_ = false;
108 
109         std::unique_ptr<std::condition_variable> done_cond_var_;
110 
111         std::vector<int> barriers_;
112 
113         JobIn(std::shared_ptr<NestedBase> nested_base, ForParameter
    for_param)
114                 : job_(nested_base, for_param), job_completed_(false) {}
115 
116     };
117 
118     struct JobQueue {
119         Jobid_t j_id_;
120         int thread_id_;
121         JobQueue(Jobid_t j_id, int thread_id) : j_id_(j_id), thread_id_(
    thread_id) {}
122     };
123 
124     ThreadPool(std::string file_name);
125 
126     void run(int id);
127 
128     std::map<int, ScheduleOptions> sched_opt_;
129 
130     std::vector<std::thread> threads_pool_; // not thread safe
131 
132     /* Job queue for each thread */
133     std::map<int, std::queue<JobQueue>> work_queue_;
134 
135     /* For each pragma the list of jobs executing that pragma, e.g. in case
    of parallel for */
136     //typedef std::pair<Jobid_t, std::thread::id> JobKey;
137     std::map<int, std::vector<JobIn>> known_jobs_;
138     //std::map<int, std::map<int, JobIn>> known_jobs_;
139     /* Mutex used by std::condition_variable to synchronize jobs execution
    */
140     //std::mutex cond_var_mtx;
141     std::map<std::thread::id, std::mutex> cond_var_mtx;
142     std::mutex job_pop_mtx;
143     std::mutex job_end;
144 };
```

Code 1.19: thread_pool.cpp

```
1 /*
2 * In case of a parallel pragma is known that each pragma present in the
     parallel's barrier list has been
3 * invoked by the thread that runs the parallel pragma.
4 *
5 * In case of a barrier pragma is known that each pragma present in the
     barrier's barrier list has been invoked
6 * by the same thread that invoked the barrieri pragma.
7 */
8 
```

```
9
10 #include "threads_pool.h"
11
12
13 std::mutex singleton_mtx;
14
15
16 ThreadPool* ThreadPool::getInstance(std::string file_name) {
17     singleton_mtx.lock();
18     static ThreadPool thread_pool(file_name);
19     singleton_mtx.unlock();
20     return &thread_pool;
21 }
22
23
24 ThreadPool::ThreadPool(std::string file_name) {
25     /* Create schdule xml file name from source code file name, e.g. test.
    cpp -> test_schedule.xml*/
26     std::string in_xml_file (file_name);
27     size_t ext = in_xml_file.find_last_of(".");
28     if (ext == std::string::npos)
29         ext = in_xml_file.length();
30     in_xml_file = in_xml_file.substr(0, ext);
31     in_xml_file.insert(ext, "_schedule.xml");
32
33     tinyxml2::XMLDocument xml_doc;
34     //xml_doc.LoadFile(in_xml_file.c_str());
35     xml_doc.LoadFile("schedule.xml");
36
37     tinyxml2::XMLElement *threads_num_element = xml_doc.FirstChildElement("
    Schedule")->FirstChildElement("Cores");
38
39     const char* threads_num = threads_num_element->GetText();
40     /* Set the number of thread as the number of cores plus one thread wich
    is used to run parallel and sections job */
41     init(chartoint(threads_num));
42
43     tinyxml2::XMLElement *pragma_element = xml_doc.FirstChildElement("
    Schedule")->FirstChildElement("Pragma");
44     while(pragma_element != NULL) {
45         ScheduleOptions sched_opt;
46
47         const char* pragma_id = pragma_element->FirstChildElement("id")->
    GetText();
48         int id = chartoint(pragma_id);
49         sched_opt.pragma_id_ = id;
50
51
52         tinyxml2::XMLElement *pragma_type_element = pragma_element->
    FirstChildElement("Type");
53         const char* pragma_type = pragma_type_element->GetText();
54         sched_opt.pragma_type_ = pragma_type;
55
56         tinyxml2::XMLElement *thread_element = pragma_element->
    FirstChildElement("Threads");
57         if(thread_element != NULL)
58             thread_element = thread_element->FirstChildElement("Thread");
59
60         while(thread_element != NULL){
```

```
61              const char *thread_id = thread_element->GetText();
62              sched_opt.threads_.push_back(chartoint(thread_id));
63
64              thread_element = thread_element->NextSiblingElement("Thread");
65          }
66
67          tinyxml2::XMLElement *barriers_element = pragma_element->
    FirstChildElement("Barrier");
68          if(barriers_element != NULL)
69              barriers_element = barriers_element->FirstChildElement("id");
70          while(barriers_element != NULL){
71              const char *thread_id = barriers_element->GetText();
72              sched_opt.barriers_.push_back(chartoint(thread_id));
73
74              barriers_element = barriers_element->NextSiblingElement("id");
75          }
76
77          sched_opt_[id] = sched_opt;
78          pragma_element = pragma_element->NextSiblingElement("Pragma");
79      }
80          //for(std::map<int, ScheduleOptions>::iterator itr = sched_opt_.
    begin(); itr != sched_opt_.end(); ++ itr)
81          //std::cout << "Pragma id: " << (*itr).second.pragma_id_ << ", type:
     " << (*itr).second.pragma_type_ << std::endl;
82
83  }
84
85
86  void ThreadPool::init(int pool_size)
87  {
88      /* This is needed cause otherwise the main process would be considered
    as thread num 0*/
89      thread_id_to_int_[std::this_thread::get_id()] = -1;
90      //std::cout << std::this_thread::get_id() << " = -1 " << std::endl;
91
92      threads_pool_.reserve(pool_size);
93      for(int i = 0; i < pool_size; i++) {
94          threads_pool_.push_back(std::thread(&ThreadPool::run,this, i));
95      }
96
97  }
98
99  /* If a job has to allocate a job on its own thread, it first allocates all
    other job and then execute directly that job */
100 /* This solve the problem of a parallel for. */
101 bool ThreadPool::call(std::shared_ptr<NestedBase> nested_b) {
102     int thread_number = sched_opt_[nested_b->pragma_id_].threads_.size();
103     int thread_id;
104     /* Get the integer id of the running thread */
105     int my_id = thread_id_to_int_[std::this_thread::get_id()];
106     /* In case of a parallel for */
107
108     if(sched_opt_[nested_b->pragma_id_].pragma_type_.compare("
    OMPForDirective") == 0
109         || sched_opt_[nested_b->pragma_id_].pragma_type_.compare("
    OMPParallelForDirective") == 0) {
110
111         call_for(nested_b);
112
```

```
113     }else {
114         thread_id = sched_opt_[nested_b->pragma_id_].threads_[0];
115         if(thread_id != my_id) {
116             push(nested_b->clone(), ForParameter(0, 1), thread_id);
117 
118             if(sched_opt_[nested_b->pragma_id_].pragma_type_.compare("
    OMPParallelDirective") == 0) {
119                 int barriers_id = sched_opt_[nested_b->pragma_id_].barriers_
    [0];
120 
121                 join(barriers_id);
122 
123                 int thread_num = sched_opt_[nested_b->pragma_id_].threads_
    [0];
124                 std::thread::id t_id = threads_pool_[thread_num].get_id();
125                 int barriers_number = sched_opt_[nested_b->pragma_id_].
    barriers_.size();
126 
127                 for (int i = 1; i < barriers_number; i ++) {
128                     barriers_id = sched_opt_[nested_b->pragma_id_].barriers_
    [i];
129                     join(barriers_id);
130                 }
131             }
132         }else {
133             if(sched_opt_[nested_b->pragma_id_].pragma_type_.compare("
    OMPParallelDirective") == 0)
134                 call_parallel(nested_b);
135             else if (sched_opt_[nested_b->pragma_id_].pragma_type_.compare("
    OMPSectionsDirective") == 0
136                 || sched_opt_[nested_b->pragma_id_].pragma_type_.compare("
    OMPSingleDirective") == 0)
137                 call_sections(nested_b);
138             else if(sched_opt_[nested_b->pragma_id_].pragma_type_.compare("
    OMPBarrierDirective") == 0)
139                 call_barrier(nested_b);
140             else {
141                 push_completed_job(nested_b, ForParameter(0, 1));
142                 return true;
143             }
144         }
145 
146     }
147     return false;
148 }
149 
150 void ThreadPool::call_sections(std::shared_ptr<NestedBase> nested_b){
151     nested_b->callme(ForParameter(0, 1));
152 
153     int barriers_number = sched_opt_[nested_b->pragma_id_].barriers_.size();
154     int barrier_id;
155     for(int i = 0; i < barriers_number; i ++) {
156         barrier_id = sched_opt_[nested_b->pragma_id_].barriers_[i];
157         join(barrier_id);
158     }
159     push_completed_job(nested_b, ForParameter(0, 1));
160 }
161 
162 void ThreadPool::call_parallel(std::shared_ptr<NestedBase> nested_b) {
```

```
163     nested_b->callme(ForParameter(0, 1));
164
165     if(sched_opt_[nested_b->pragma_id_].pragma_type_.compare("
        OMPParallelDirective") == 0) {
166         int barriers_id = sched_opt_[nested_b->pragma_id_].barriers_[0];
167         join(barriers_id);
168
169         int barriers_number = sched_opt_[nested_b->pragma_id_].barriers_.
        size();
170         for (int i = 1; i < barriers_number; i ++) {
171             barriers_id = sched_opt_[nested_b->pragma_id_].barriers_[i];
172             int thread_num = sched_opt_[barriers_id].threads_[0];
173             std::thread::id t_id = threads_pool_[thread_num].get_id();
174             join(barriers_id);
175         }
176     }
177 }
178
179 void ThreadPool::call_for(std::shared_ptr<NestedBase> nested_b) {
180     int thread_number = sched_opt_[nested_b->pragma_id_].threads_.size();
181     int thread_id;
182     /* Get the integer id of the running thread */
183     int my_id = thread_id_to_int_[std::this_thread::get_id()];
184     for(int i = 0; i < thread_number; i ++) {
185         thread_id = sched_opt_[nested_b->pragma_id_].threads_[i];
186         if(thread_id != my_id) {
187             push(nested_b->clone(), ForParameter(i, thread_number),
        thread_id);
188         }
189     }
190     /* If a son and a father are on the same thread!!! */
191     for(int i = 0; i < thread_number; i ++) {
192         thread_id = sched_opt_[nested_b->pragma_id_].threads_[i];
193         if(thread_id == my_id) {
194             push_completed_job(nested_b->clone(), ForParameter(i,
        thread_number));
195             nested_b->callme(ForParameter(i, thread_number));
196         }
197     }
198
199     //if(sched_opt_[nested_b->pragma_id_].barriers_.size() > 0) {
200         int barriers_id = sched_opt_[nested_b->pragma_id_].barriers_[0];
201         join(barriers_id);
202     //}
203 }
204
205 void ThreadPool::call_barrier(std::shared_ptr<NestedBase> nested_b) {
206     int barriers_number = sched_opt_[nested_b->pragma_id_].barriers_.size();
207     int barriers_id, threads_num;
208     for (int i = 0; i < barriers_number; i ++) {
209         barriers_id = sched_opt_[nested_b->pragma_id_].barriers_[i];
210         join(barriers_id);
211     }
212 }
213
214 /* Insert a job wich has the flag completed already setted. This is
        necessary in case a thread executes more
215 job consecutively */
216 void ThreadPool::push_completed_job(std::shared_ptr<NestedBase> nested_base,
```

```
                                        ForParameter for_param) {

    Jobid_t id = nested_base->pragma_id_;

    JobIn job_in(nested_base, for_param);
    job_in.job_id_ = id;
    job_in.job_completed_ = true;

    job_pop_mtx.lock();
    if(known_jobs_[id].size() == 0)
        known_jobs_[id].reserve(for_param.num_threads_);
    known_jobs_[id].push_back(std::move(job_in));
    job_pop_mtx.unlock();
}


void ThreadPool::push(std::shared_ptr<NestedBase> nested_base,
                                        ForParameter for_param, int thread_id)
   {

    Jobid_t id = nested_base->pragma_id_;

    JobIn job_in(nested_base, for_param);
    job_in.job_id_ = id;
    job_in.job_type_ = sched_opt_[nested_base->pragma_id_].pragma_type_;
    job_in.done_cond_var_ =
            std::unique_ptr<std::condition_variable>(new std::
   condition_variable());

    job_pop_mtx.lock();
    if(known_jobs_[id].size() == 0)
        known_jobs_[id].reserve(for_param.num_threads_);
    known_jobs_[id].push_back(std::move(job_in));

    JobQueue j_q(id, for_param.thread_id_);
    work_queue_[thread_id].push(j_q);

    job_pop_mtx.unlock();

}


void ThreadPool::push_termination_job(int thread_id) {

    JobQueue j_q(-1, 0);
    work_queue_[thread_id].push(j_q);
}


void ThreadPool::run(int me) {
    thread_id_to_int_[std::this_thread::get_id()] = me;
    while(true) {

        job_pop_mtx.lock();
        if(work_queue_[me].size() != 0) {

            JobQueue j_q = work_queue_[me].front();
            work_queue_[me].pop();
            job_pop_mtx.unlock();
```

```
274
275             int pragma_id = j_q.j_id_;
276             int thread_id = j_q.thread_id_;
277
278             if(pragma_id != 0) {
279                 if(pragma_id == -1)
280                     break;
281
282                 job_pop_mtx.lock();
283                 std::vector<JobIn>::iterator j_itr;
284                 for(j_itr = known_jobs_[pragma_id].begin(); j_itr !=
    known_jobs_[pragma_id].end(); ++ j_itr) {
285                     if(j_itr->job_.for_param_.thread_id_ == thread_id)
286                         break;
287                 }
288
289                 job_pop_mtx.unlock();
290                 ForParameter for_param = j_itr->job_.for_param_;
291
292                 try {
293                     j_itr->job_.nested_base_->callme(for_param);
294                 }catch(std::exception& e){
295                     //known_jobs_[pragma_id][thread_id].
    terminated_with_exceptions_ = true;
296                     std::cerr << "Pragma␣" << pragma_id << "␣terminated␣with
    ␣exception:␣" << e.what() << std::endl;
297                 }
298
299
300                 if(j_itr->job_type_.compare("OMPTaskDirective") == 0
301                     || j_itr->job_type_.compare("OMPSingleDirective") == 0
302                     || j_itr->job_type_.compare("OMPSectionsDirective") ==
    0)
303                 {
304                     int barriers_number = sched_opt_[pragma_id].barriers_.
    size();
305                     int barrier_id;
306                     for(int i = 0; i < barriers_number; i ++) {
307                         barrier_id = sched_opt_[pragma_id].barriers_[i];
308                         join(barrier_id);
309                     }
310                 }
311
312                 job_end.lock();
313                 j_itr->job_completed_ = true;
314                 j_itr->done_cond_var_->notify_one();
315                 job_end.unlock();
316             }
317         }else {
318             job_pop_mtx.unlock();
319         }
320     }
321 }
322
323
324 void ThreadPool::join(Jobid_t job_id) {
325
326     /*for(int i = 0; i < known_jobs_[job_id].size(); i ++) {
327         if(known_jobs_[job_id][i].job_completed_ != true) {
```

```
328             std::unique_lock<std::mutex> lk(cond_var_mtx);
329             known_jobs_[job_id][i].done_cond_var_->wait(lk);
330         }
331     }*/
332     //std::mutex cond_var_mtx;
333 
334     std::vector<JobIn>::iterator j_itr;
335     for(j_itr = known_jobs_[job_id].begin(); j_itr != known_jobs_[job_id].
    end(); ++ j_itr) {
336         job_end.lock();
337         if((*j_itr).job_completed_ != true){
338             job_end.unlock();
339             std::unique_lock<std::mutex> lk(cond_var_mtx[std::this_thread::
    get_id()]);
340             j_itr->done_cond_var_->wait(lk);
341         }else{
342             job_end.unlock();
343         }
344     }
345     job_pop_mtx.lock();
346     known_jobs_.erase(job_id);
347     job_pop_mtx.unlock();
348 }
349 
350 
351 void ThreadPool::joinall() {
352     /* Push termination job in the working queue */
353     std::cout << "Joinall" << std::endl;
354     for (int i = 0; i < threads_pool_.size(); i ++)
355         push_termination_job(i);
356 
357     /* Joining on all the threads in the thread pool */
358     for(int i = 0; i < threads_pool_.size(); i++)
359         threads_pool_[i].join();
360 
361 }
362 
363 
364 int chartoint(const char *cc){
365     std::string s(cc);
366     char c;
367     int n = 0;
368     int tmp;
369     int i = s.size();
370     for(std::string::iterator sitr = s.begin(); sitr != s.end(); ++ sitr){
371         c = *sitr;
372         tmp = c - 48;
373         tmp = tmp*pow(10, i-1);
374         n += tmp;
375         i --;
376     }
377     return n;
378 }
379 
380 int chartoint(char *cc){
381     const char *c = cc;
382     return chartoint(c);
383 }
```

Chapter 2

Python

2.1 Main

Code 2.1: appsched.py

```
iimport sys
import pargraph as par
import copy
import schedule as sched
import profiler as pro
import time
import multiprocessing
import itertools
import random
import threading
import argparse

if __name__ == "__main__":
  import argparse

  parser = argparse.ArgumentParser(description='Code Flow Toolchain')
  parser.add_argument('--profile',help='profile description file',required=
   True)
  parser.add_argument('--pragmaxml',help='profile description file',required
   =True)
  parser.add_argument('--cores',help='number of cores. If 0 or missing use
   the host cores',type=int)
  parser.add_argument('--maxflows',help='number of flows. Estimated from
   profiling or specified',type=int)
  parser.add_argument('--deadline',help='Deadline in seconds',type=float)
  parser.add_argument('--executiontime',help='Execution time in seconds',
   type=float)
  parser.add_argument('--parallelize',help='Execute the Algorithm in
   parallel mode')
  parser.add_argument('--output',help='output schedule')

  args = parser.parse_args()

  pragma_xml = args.pargmaxml
  profile_xml = args.profile
  cores = args.cores
  output = args.output
  deadline = float(args.deadline)
  execution_time = float(args.executiontime)
  parallelized = args.parallelize
```

```
35
36  #return the nested dot graphs in code style (one for each function)
37  visual_nested_graphs = par.getNesGraph(pragma_xml, profile_xml)
38
39  #returns the graphs to be visualized and the object graphs in flow style (
     one for each function)
40  (visual_flow_graphs, flow_graphs) = par.getParalGraph(pragma_xml,
     profile_xml)
41
42  i = 0
43
44  os.mkdir("graphs")
45
46  for g in visual_nested_graphs:
47    g.write_pdf('graphs/%s_code.pdf'%flow_graphs[i].type)
48    g.write_dot('graphs/%s_code.dot'%flow_graphs[i].type)
49    i += 1
50
51  i = 0
52  for g in visual_flow_graphs:
53    g.write_pdf('graphs/%s_flow.pdf'%flow_graphs[i].type)
54    g.write_dot('graphs/%s_flow.dot'%flow_graphs[i].type)
55    i += 1
56
57  #creates the flow type graph --> flow.xml
58  par.dump_graphs(flow_graphs)
59  #adding to the original xml the profiling informations --> code.xml
60  pro.add_profile_xml(profile_xml, pragma_xml)
61  #creating the total graph with the call-tree
62  func_graph = par.create_complete_graph(visual_flow_graphs, profile_xml)
63  #creating the graphs with the function calls
64  func_graph.write_pdf('graphs/function_graphs.pdf')
65  func_graph.write_dot('graphs/function_graphs.dot')
66
67  #creating the expanded graph where the functions are inserted in the flow
     graph
68  exp_flows = copy.deepcopy(flow_graphs)
69  par.explode_graph(exp_flows)
70  main_flow = sched.get_main(exp_flows)
71
72  #creating a generator for the expanded graph
73  gen = sched.generate_task(main_flow)
74
75  #creating a new generator for the expanded graph
76  sched.make_white(main_flow)
77
78  #getting the number of physical cores of the machine profiled
79  if args.maxflows == 0:
80    max_flows = sched.get_core_num(profile_xml)
81  else:
82    max_flows = args.maxflows
83  if cores == 0:
84    cores = multiprocessing.cpu_count()
85
86  #initializing all the lists for the parallel scheduling algorithm
87  tasks_list = []
88  task_list = []
89  flows_list = []
90  optimal_flow_list = []
```

```
91    p_list = []
92    queue_list = []
93    results = []
94    num_tasks = 0
95
96    #getting the number of tasks in the expanded graph and creating a list of
       task
97    for task in gen:
98      task_list.append(task)
99      num_tasks += 1
100
101   if output != "":
102     sched.make_white(main_flow)
103     par.scanGraph(main_flow)
104
105   #starting the parallel or sequential search of the best solution with a
       timing constrain
106   if parallelized:
107     for core in range(cores):
108       tmp = []
109       optimal_flow_list.append(tmp)
110       tmp_2 = []
111       flows_list.append(tmp_2)
112       random.shuffle(task_list)
113       tasks_list.append(copy.deepcopy(task_list))
114       q = sched.Queue()
115       queue_list.append(q)
116       p_list.append(multiprocessing.Process(target = sched.get_optimal_flow,
        args = (flows_list[core], tasks_list[core], 0, optimal_flow_list[core],
      num_tasks, max_flows, execution_time, queue_list[core],  )))
117       print "starting␣core:␣",core
118       p_list[core].start()
119     #getting the results from the processes
120     for queue in queue_list:
121       t = queue.q.get()
122       results.append(t)
123     #joining all the processes
124     i = 0
125     for p in p_list:
126       p.join()
127       print "core␣", i, "␣joined"
128       i += 1
129     #getting the best result
130     optimal_flow = results[0]
131     best = 0
132     for i in range(len(results)):
133       print "result:"
134       for flow in results[i]:
135         flow.dump()
136         if sched.get_cost(results[i]) < sched.get_cost(optimal_flow):
137           best = i
138     optimal_flow = results[best]
139   else:
140       optimal_flow = []
141       flow_list = []
142       execution_time += time.clock()
143       print "searching␣best␣schedule"
144       sched.get_optimal_flow_single(flow_list, task_list, 0, optimal_flow,
      num_tasks, max_flows, execution_time )
```

```
145
146
147
148   #printing the best result
149   print "solution:"
150   for flow in optimal_flow:
151     flow.dump("\t")
152     print "\ttime:",flow.time
153
154
155
156   #substitutes "for␣tasks" with splitted versions if present in the optimal
       flows
157   par.add_new_tasks(optimal_flow, main_flow)
158   sched.make_white(main_flow)
159   gen_ = sched.generate_task(main_flow)
160
161
162   t_list = []
163   for t in gen_:
164     t_list.append(t)
165     """
166 ␣␣␣␣print␣t.type,␣" @ ",␣t.start_line,␣" has parents:"
167 ␣␣␣␣for␣p␣in␣t.parent:
168 ␣␣␣␣␣␣print␣"\t ",p.type,␣" @ ",␣p.start_line
169 ␣␣␣␣print␣"and children:"
170 ␣␣␣␣for␣c␣in␣t.children:
171 ␣␣␣␣␣␣print␣"\t ",c.type,␣" @ ",␣c.start_line
172 ␣␣␣␣print
173 ␣␣␣␣"""
174
175   #adds id's to all the tasks to retrive the flow to which they belong
176   par.add_flow_id(optimal_flow, t_list)
177
178   #sets arrival times and deadlines using a modified version of the chetto
       algorithm
179   sched.chetto(main_flow, deadline, optimal_flow)
180
181   #checks if the schedule is feasible and in case creates the schedule file
182   if sched.check_schedule(main_flow):
183     sched.create_schedule(main_flow, len(optimal_flow))
184     sched.make_white(main_flow)
185     #sched.print_schedule(main_flow)
186   else:
187     print "tasks␣not␣schedulable,␣try␣with␣more␣search␣time"
188
189   #prints extended info of the entire pragma graph
```

2.2 Graph Creator

Code 2.2: pargraph.py

```
1 import pydot as p
2 import profiler as pro
3 import xml.etree.cElementTree as ET
4 from random import randrange
5 import copy
6 import schedule as sched
```

```
import re
import math

colors = ( "beige", "bisque3", "bisque4", "blanchedalmond", "blue",
"blue1", "blue2", "blue3", "blue4", "blueviolet",
"brown", "brown1", "brown2", "brown3", "brown4",
"burlywood", "burlywood1", "burlywood2", "burlywood3", "burlywood4",
"cadetblue", "cadetblue1", "cadetblue2", "cadetblue3", "cadetblue4",
"chartreuse", "chartreuse1", "chartreuse2", "chartreuse3", "chartreuse4",
"chocolate", "chocolate1", "chocolate2", "chocolate3", "chocolate4",
"coral", "coral1", "coral2", "coral3", "coral4",
"cornflowerblue", "crimson", "cyan", "cyan1", "cyan2",
"cyan3", "cyan4", "darkgoldenrod", "darkgoldenrod1", "darkgoldenrod2",
"darkgoldenrod3", "darkgoldenrod4", "darkgreen", "darkkhaki", "
    darkolivegreen",
"darkolivegreen1", "darkolivegreen2", "darkolivegreen3", "darkolivegreen4
    ", "darkorange",
"darkorange1", "darkorange2", "darkorange3", "darkorange4", "darkorchid"
    ,
"darkorchid1", "darkorchid2", "darkorchid3", "darkorchid4", "darksalmon"
    ,
"darkseagreen", "darkseagreen1", "darkseagreen2", "darkseagreen3", "
    darkseagreen4",
"darkslateblue", "darkslategray", "darkslategray1", "darkslategray2", "
    darkslategray3",
"darkslategray4", "darkslategrey", "darkturquoise", "darkviolet", "
    deeppink",
"deeppink1", "deeppink2", "deeppink3", "deeppink4", "deepskyblue",
"deepskyblue1", "deepskyblue2", "deepskyblue3", "deepskyblue4", "dimgray",
"dimgrey", "dodgerblue", "dodgerblue1", "dodgerblue2", "dodgerblue3",
"dodgerblue4", "firebrick", "firebrick1", "firebrick2", "firebrick3",
"firebrick4", "forestgreen", "gold", "gold1", "gold2",
"gold3", "gold4", "goldenrod", "goldenrod1", "goldenrod2", "goldenrod3"
    , "goldenrod4")

class Node(object):
    def __init__(self, Ptype, s_line, time, variance):
        self.type = Ptype
        self.start_line = s_line
        self.children = []
        self.parent = []
        self.options = []
        self.time = float(time)
        self.variance = variance
        self.end_line = 0
        self.callerid = []
        self.deadline = None
        self.arrival = None
        self.d = None
        self.children_time = 0
        self.in_time = 0
        self.color = 'white'
        self.id = None
    def add(self, x):
      x.parent.append(self)
      self.children.append(x)
    def myself(self):
    if self.type != 'BARRIER':
```

```
57        print "pragma␣node:␣", self.type, "\n␣␣␣␣start_line:␣", self.
      start_line, "\n␣␣␣␣endl_line", self.end_line
58        if self.type.find("_end") == -1:
59          if self.time != 0:
60            print "␣␣␣␣time:␣", self.time
61            print "␣␣␣␣variance:␣", self.variance
62            print "␣␣␣␣children␣time:␣", self.children_time
63            print "␣␣␣␣self␣time:␣", self.in_time
64          else:
65            print "␣␣␣␣not␣executed"
66          if(len(self.options) != 0):
67            print "␣␣␣␣Options:"
68            for i in self.options:
69              print "␣␣␣␣␣␣␣␣",i[0],"␣",i[1]
70        print "␣␣␣␣␣chetto␣deadline␣:", self.d
71        print "␣␣␣␣␣chetto␣arrival␣:", self.arrival
72      else:
73        print "pragma␣node:␣", self.type, "\n␣␣␣␣start_line:␣", self.
      start_line
74      print
75
76  class For_Node(Node):
77      def __init__(self, Ptype, s_line, init_type, init_var, init_value,
      init_cond, init_cond_value, init_increment, init_increment_value, time,
      variance, mean_loops):
78          Node.__init__(self, Ptype, s_line, time, variance)
79          self.init_type = init_type
80          self.init_var = init_var
81          self.init_value = init_value
82          self.init_cond = init_cond
83          self.init_cond_value = init_cond_value
84          self.init_increment = init_increment
85          self.init_increment_value = init_increment_value
86          self.mean_loops = mean_loops
87      def myself(self):
88      print "for␣node:␣", self.type, "\n␣␣␣␣start_line:␣", self.start_line, "\
      n␣␣␣␣endl_line:␣", self.end_line, "\n␣␣␣␣init_type:", self.init_type, "\n
      ␣␣␣␣init_var:␣", self.init_var, "\n␣␣␣␣init_value:␣", self.init_value,"\n
      ␣␣␣␣init_condition:␣", self.init_cond, "\n␣␣␣␣init_condition_value:␣",
      self.init_cond_value, "\n␣␣␣␣init_increment_type:␣", self.init_increment,
      "\n␣␣␣␣init_increment:␣", self.init_increment_value, "\n␣␣␣␣mean_loops:",
       self.mean_loops
89      print "␣␣␣␣␣chetto␣deadline␣:", self.d
90      print "␣␣␣␣␣chetto␣arrival␣:", self.arrival
91      if(len(self.options) != 0):
92        print "␣␣␣␣Options:"
93        for i in self.options:
94          print "␣␣␣␣␣␣␣␣", i[0], "␣", i[1]
95      if self.time != 0:
96        print "␣␣␣␣time:␣", self.time
97        print "␣␣␣␣variance:␣", self.variance
98        print "␣␣␣␣children␣time:␣", self.children_time,"\n"
99        print "␣␣␣␣self␣time:␣", self.in_time,"\n"
100     else:
101       print "␣␣␣␣not␣executed\n"
102
103 class Fx_Node(Node):
104   def __init__(self, Ptype, line, returnType, time, variance, file_name):
105     Node.__init__(self, Ptype, line, time, variance)
```

```
106     self.arguments = []
107     self.returnType = returnType
108     self.time = float(time)
109     self.file_name = file_name
110   def add_arg(self, type_):
111     self.arguments.append(type_)
112   def myself(self):
113     print "function␣node:␣", self.type, "()␣{\n␣␣␣␣line:␣", self.start_line,
      "\n␣␣␣␣return␣type:␣", self.returnType
114     print "␣␣␣␣␣chetto␣deadline␣:", self.d
115     print "␣␣␣␣␣chetto␣arrival␣:", self.arrival
116     if(len(self.arguments) != 0):
117       print "␣␣␣␣Parameters:␣"
118       i = 0
119       for par in self.arguments:
120         print "␣␣␣", i, ")␣", par[0], "␣", par[1]
121         i = i + 1
122     else:
123       print "␣␣␣␣No␣input␣parameters"
124     if self.time != 0:
125       print "␣␣␣␣time:␣", self.in_time
126       print "␣␣␣␣variance:␣", self.variance
127       print "␣␣␣␣children␣time:␣", self.children_time,"\n}\n"
128     else:
129       print "␣␣␣␣not␣executed\n}\n"
130
131 class Function():
132   def __init__(self, time, variance, children_time):
133     self.time = float(time)
134     self.variance = variance
135     self.pragmas = {}
136     self.children_time = float(children_time)
137     self.in_time = float(self.time) - float(self.children_time)
138   def add_pragma(self, pragma):
139     self.pragmas[pragma[0]] = (pragma[1], pragma[2], pragma[3], pragma[4],
     pragma[5])
140
141 class Architecture():
142   def __init__(self, num_cores, tot_memory):
143     self.num_cores = num_cores
144     self.tot_memory = tot_memory
145
146 class Time_Node():
147   def __init__(self, func_line, pragma_line ):
148     self.times = []
149     self.func_line = func_line
150     self.pragma_line = pragma_line
151     self.variance = 0
152     self.loops = []
153     self.caller_list = []
154     self.children_time = []
155
156 class Flow():
157   def __init__(self):
158     self.tasks = []
159     self.bandwidth = 0
160     self.time = 0
161   def add_task(self, task):
162     self.tasks.append(task)
```

```
163     self.update(task)
164   def update(self, task):
165     self.time += task.in_time #float(task.time) - float(task.children_time)
166   def dump(self,prefix=""):
167     print prefix,"flow:"
168     for task in self.tasks:
169       print prefix, "\t", task.type, "␣", task.start_line, "␣", task.in_time
      , "␣id␣", task.id
170   def remove_task(self, task):
171     self.tasks.remove(task)
172     self.time -= task.in_time #float(task.time) - float(task.children_time)
173
174 class Task():
175   def __init__(self, count, id):
176     self.count = count
177     self.id = []
178     self.id.append(id)
179
180
181 def scanGraph(node):
182   #print pre, node.type
183   if node.color != 'black':
184     node.color = 'black'
185     node.myself()
186     print "␣␣␣␣␣has␣children:"
187     for c in node.children:
188       print "␣␣␣␣␣␣␣␣␣␣",c.type,"@",c.start_line
189     print "␣␣␣␣␣has␣parent:"
190     for p in node.parent :
191       print "␣␣␣␣␣␣␣␣␣␣",p.type,"@",p.start_line
192     for n in node.children:
193       scanGraph(n)
194
195 def indent(elem, level=0):
196     i = "\n" + level * "␣␣"
197     if len(elem):
198         if not elem.text or not elem.text.strip():
199             elem.text = i + "␣␣"
200         if not elem.tail or not elem.tail.strip():
201             elem.tail = i
202         for elem in elem:
203             indent(elem, level + 1)
204         if not elem.tail or not elem.tail.strip():
205             elem.tail = i
206     else:
207         if level and (not elem.tail or not elem.tail.strip()):
208             elem.tail = i
209
210 def getParalGraph(pragma_xml, profile_xml):
211   pragma_graph_root = ET.ElementTree(file = pragma_xml).getroot()
212   profile_graph_root = ET.ElementTree(file = profile_xml).getroot()
213
214   functions = pro.getProfilesMap(profile_xml)
215   objGraph = []
216   graphs = []
217   count = 0
218   arch = Architecture(profile_graph_root.find('Hardware/NumberofCores').text
      , profile_graph_root.find('Hardware/MemorySize').text)
219
```

```
220   file_name = pragma_graph_root.find('Name').text
221
222   for n in pragma_graph_root.findall('Function'):
223     graphs.append(p.Dot(graph_type = 'digraph'))
224     name = n.find('Name').text
225     time = float(functions[n.find('Line').text].time)
226     callerid = functions[n.find('Line').text].callerid
227     children_time = float(functions[n.find('Line').text].children_time)
228     root = n.find('Line').text
229     if (time == 0):
230       pragma_graph_root = p.Node(n.find('Line').text, label = name + "()\
    nnot executed", root = root)
231     else:
232       pragma_graph_root = p.Node(n.find('Line').text, label = name + "()\
    nexecution time %g" % time, root = root)
233     pragma_graph_root.callerid = callerid
234     graphs[count].add_node(pragma_graph_root)
235     Objroot = Fx_Node(name, n.find('Line').text,n.find('ReturnType').text,
    float(functions[n.find('Line').text].time), functions[n.find('Line').text
    ].variance, file_name)
236     for par in n.findall('Parameters/Parameter'):
237       Objroot.add_arg( ( par.find('Type').text,par.find('Name').text ) )
238     Objroot.children_time = children_time
239     Objroot.in_time = Objroot.time - children_time
240     for caller in functions[n.find('Line').text].callerid:
241       Objroot.callerid.append(caller)
242     objGraph.append(Objroot)
243     scan(n, graphs[count], pragma_graph_root, objGraph[count], functions[n.
    find('Line').text].pragmas, root)
244     count = count + 1
245   return (graphs, objGraph)
246
247 def scan(xml_tree, pragma_graph, node, treeNode, func_pragmas, root):
248   for d in xml_tree.find('Pragmas').findall('Pragma'):
249     end_line = d.find('Position/EndLine').text
250     key = d.find('Position/StartLine').text
251
252     if key not in func_pragmas:
253       time = 0
254       variance = None
255       loops = 0
256       callerid = None
257       children_time = 0
258     else:
259       time = float(func_pragmas[key][0])
260       variance = func_pragmas[key][1]
261       loops = func_pragmas[key][2]
262       callerid = func_pragmas[key][3]
263       children_time =  float(func_pragmas[key][4])
264
265     tmp_name = d.find('Name').text.replace("::", " ")
266     visual_name = tmp_name+"@%s"%key
267
268     if ("For" in tmp_name ):
269       if (d.find('For/Declaration/InitValue') != None):
270         init_value = d.find('For/Declaration/InitValue').text
271       else:
272         init_value = d.find('For/Declaration/InitVariable').text
273       if (d.find('For/Condition/ConditionValue') != None):
```

```
274         init_var = d.find('For/Condition/ConditionValue').text
275       else:
276         init_var = d.find('For/Condition/ConditionVariable').text
277       if(d.find('For/Increment/IncrementValue') != None):
278         inc = d.find('For/Increment/IncrementValue').text
279       else:
280         inc = ""
281       Objchild = For_Node(tmp_name, d.find('Position/StartLine').text, d.
    find('For/Declaration/Type').text, d.find('For/Declaration/LoopVariable')
    .text, init_value, d.find('For/Condition/Op').text, init_var, d.find('For
    /Increment/Op').text, inc, time, variance, loops )
282       visual_name = visual_name + "\nfor( " + Objchild.init_var + " = " +
    Objchild.init_value + "; " + Objchild.init_var + " " + Objchild.init_cond
     + " " + Objchild.init_cond_value + "; " + Objchild.init_var + " " +
    Objchild.init_increment + " " + Objchild.init_increment_value + ")"
283     else:
284       Objchild = Node(tmp_name, key, time, variance )
285
286     deadline = None
287     if(d.find('Options')):
288       for op in d.findall('Options/Option'):
289         Objchild.options.append( (op.find('Name').text,[get_parameter(i) for
     i in op.findall('Parameter')]) )
290         if op.find('Name').text == 'deadline':
291           deadline = op.find('Parameter').text
292     Objchild.end_line = end_line
293     Objchild.callerid.append(callerid)
294     Objchild.deadline = deadline
295     Objchild.children_time = children_time
296     Objchild.in_time = Objchild.time - children_time
297     if (time == 0):
298       child = p.Node(key, label = visual_name + "\nnot executed", root =
    root)
299     else:
300       child = p.Node(key, label = visual_name + "\nexecution time: " + str(
    time) + "\nvariance: " + str(variance), root = root)
301     pragma_graph.add_node(node)
302     pragma_graph.add_node(child)
303     pragma_graph.add_edge(p.Edge(node, child))
304     treeNode.add(Objchild)
305     #print Objchild.type,"@",Objchild.start_line," is attached to ",treeNode
    .type,"@",treeNode.start_line
306
307     if(d.find('Children')):
308       node_ = create_diamond(d.find('Children'), pragma_graph, child,
    Objchild, func_pragmas, root)
309       tmp_name = (node_.start_line)
310       if tmp_name not in func_pragmas:
311         time = 0
312       else:
313         time = func_pragmas[tmp_name][0]
314       #treeNode = Node('BARRIER_end', tmp_name, 0, 0)
315       #Objchild.add(treeNode)
316       treeNode = node_
317       node = p.Node(tmp_name + "_end", label = "BARRIER", root = root)
318     else:
319       node = child
320       treeNode = Objchild
321
```

```
322 def create_diamond(tree, graph, node, treeNode, func_pragmas, root):
323   special_node = p.Node(node.get_name().replace("\"", "") + "_end", label =
       'BARRIER', root = root)
324   Objspecial_node = Node( 'BARRIER_end' , node.get_name() , 0, 0 )
325   color = colors[randrange(len(colors) - 1)]
326   for d in tree.find('Pragmas').findall('Pragma'):
327
328     end_line = d.find('Position/EndLine').text
329     key = d.find('Position/StartLine').text
330
331     if key not in func_pragmas:
332       time = 0
333       variance = None
334       loops = 0
335       callerid = None
336       children_time = 0
337     else:
338       time = float(func_pragmas[key][0])
339       variance = func_pragmas[key][1]
340       loops = func_pragmas[key][2]
341       callerid = func_pragmas[key][3]
342       children_time = float(func_pragmas[key][4])
343
344     tmp_name = d.find('Name').text.replace("::", "␣")
345     visual_name = tmp_name + "@%s" % key
346
347     if ("For" in tmp_name ):
348       loops = func_pragmas[key][2]
349       if (d.find('For/Declaration/InitValue') != None):
350         init_value = d.find('For/Declaration/InitValue').text
351       else:
352         init_value = d.find('For/Declaration/InitVariable').text
353       if (d.find('For/Condition/ConditionValue') != None):
354         init_var = d.find('For/Condition/ConditionValue').text
355       else:
356         init_var = d.find('For/Condition/ConditionVariable').text
357       if(d.find('For/Increment/IncrementValue') != None):
358         inc = d.find('For/Increment/IncrementValue').text
359       else:
360         inc = ""
361       Objchild = For_Node(tmp_name, key, d.find('For/Declaration/Type').text
    , d.find('For/Declaration/LoopVariable').text, init_value, d.find('For/
    Condition/Op').text, init_var, d.find('For/Increment/Op').text, inc ,
    time, variance, loops)
362       visual_name = visual_name + "\nfor(␣" + Objchild.init_var + "␣=␣" +
    Objchild.init_value + ";␣"+Objchild.init_var + "␣" + Objchild.init_cond +
     "␣" + Objchild.init_cond_value + ";␣" + Objchild.init_var + "␣" +
    Objchild.init_increment + "␣" + Objchild.init_increment_value + ")"
363     else:
364       Objchild = Node(tmp_name, key, time, variance)
365
366     deadline = None
367     if(d.find('Options')):
368       for op in d.find('Options').findall('Option'):
369         Objchild.options.append( (op.find('Name').text,[get_parameter(i) for
    i in op.findall('Parameter')]) )
370         if op.find('Name').text == 'deadline':
371           deadline = op.find('Parameter').text
372
```

```
373     Objchild.end_line = end_line
374     Objchild.callerid.append(callerid)
375     Objchild.deadline = deadline
376     Objchild.children_time = children_time
377     Objchild.in_time = Objchild.time - children_time
378 
379     child = p.Node(key, label = visual_name + "\nexecution time: " + str(
    time) + "\nvariance: " + str(variance), root = root)
380     graph.add_node(node)
381     graph.add_node(child)
382     graph.add_edge(p.Edge(node, child, color = color))
383     treeNode.add(Objchild)
384 
385     if(d.find('Children')):
386       #get the real returned label as name
387       tmp_node = create_diamond(d.find('Children'), graph, child, Objchild,
    func_pragmas, root)
388       g_node = p.Node(tmp_node.start_line+ "_end", label = 'BARRIER', root =
     root)
389       graph.add_node(g_node)
390       graph.add_node(special_node)
391       graph.add_edge(p.Edge(g_node, special_node, color = color))
392       #tmp_name = tmp.get_name().replace("\"", "")
393       #ObjTmp = Node(tmp_name, tmp_name, 0, 0)
394       tmp_node.add(Objspecial_node)
395     else:
396       graph.add_node(child)
397       graph.add_node(special_node)
398       graph.add_edge(p.Edge(child, special_node, color = color))
399       Objchild.add(Objspecial_node)
400   return Objspecial_node
401 
402 def find_nesting(tree, graph, node, func_pragmas, pre = ""):
403   color = colors[randrange(len(colors) - 1)]
404   for d in tree.find('Pragmas').findall('Pragma'):
405     key = d.find('Position/StartLine').text
406     if(key in func_pragmas):
407       time = "\n execution time: " + str(func_pragmas[key][0])
408       variance = "\nvariance: " + str(func_pragmas[key][1])
409     else:
410       time = "\nnot executed"
411       variance = ""
412     name = d.find('Name').text.replace("::"," ") + "@%s" % key
413     child = p.Node(name, label = name + time + variance)
414     graph.add_node(node)
415     graph.add_node(child)
416     graph.add_edge(p.Edge(node, child, color = color ))
417     #print pre+name
418     if(d.find('Children')):
419       find_nesting(d.find('Children'), graph, child, func_pragmas, pre + " "
    )
420 
421 def getNesGraph(xml, profile_xml):
422   tree = ET.ElementTree(file = xml)
423   profile_graph_root = ET.ElementTree(file = profile_xml).getroot()
424   functions = pro.getProfilesMap(profile_xml)
425 
426   root = tree.getroot()
427   graphs = []
```

```
428   count = 0
429
430   for n in root.iter('Function'):
431     key = n.find('Line').text
432     time = float(functions[key].time)
433     variance = functions[key].variance
434     graphs.append(p.Dot(graph_type = 'digraph'))
435     name = n.find('Name').text
436     if (time == 0):
437       root = p.Node(name, label = name + "()" + "\nnot executed")
438     else:
439       root = p.Node(name, label = name + "()" + "\n execution time: %f" %
      time + "\nvariance: " + str(variance))
440     graphs[count].add_node(root)
441     find_nesting(n, graphs[count], root, functions[key].pragmas)
442     count += 1
443
444   return graphs
445
446 def create_complete_graph(visual_flow_graphs, profile_xml):
447   func_graph = p.Dot(graph_type = 'digraph', compound = 'true')
448   clusters = []
449
450   i = 0
451
452   for func in visual_flow_graphs:
453     clusters.append(p.Cluster(str(i)))
454     for node in func.get_nodes():
455       clusters[i].add_node(node)
456     for edge in func.get_edge_list():
457       clusters[i].add_edge(edge)
458     func_graph.add_subgraph(clusters[i])
459     i +=  1
460
461   functions_callers = pro.get_table(profile_xml)
462
463   for func in visual_flow_graphs:
464     root = func.get_nodes()[0].obj_dict['attributes']['root']
465     if len(functions_callers[root]) > 0 :
466       for caller in functions_callers[root]:
467         func_graph.add_edge(p.Edge(caller, root))
468
469   return func_graph
470
471 def dump_graphs(flow_graphs):
472   root = ET.Element('File')
473   name = ET.SubElement(root, 'Name')
474   name.text = flow_graphs[0].file_name
475   graph_type = ET.SubElement(root, 'GraphType')
476   graph_type.text = "flow"
477   for func in flow_graphs:
478     function = ET.SubElement(root, 'Function')
479     function.attrib['id'] = str(func.start_line) + str(func.end_line)
480     func_name = ET.SubElement(function, 'Name')
481     func_name.text = func.type
482     returnType = ET.SubElement(function, 'ReturnType')
483     returnType.text = func.returnType
484     if len(func.arguments) != 0:
485       parameters = ET.SubElement(function, 'Parameters')
```

```
      for par in func.arguments:
        parameter = ET.SubElement( parameters, 'Parameter')
        type_ = ET.SubElement( parameter, 'Type')
        type_.text = par[0]
        name_ = ET.SubElement( parameter, 'Name')
        name_.text = par[1]
    line = ET.SubElement(function, 'Line')
    line.text = func.start_line
    time = ET.SubElement(function, 'Time')
    time.text = str(func.time)
    variance = ET.SubElement(function, 'Variance')
    variance.text = str(func.variance)
    func.xml_parent = None
    if ( func.callerid != None ):
      callerids = ET.SubElement(function, 'Callerids')
      for id_ in func.callerid:
        callerid = ET.SubElement(callerids, 'Callerid')
        callerid.text = id_
    if len(func.children) != 0:
      pragma_list = []
      edge_list = []
      pragmas = ET.SubElement(function, 'Nodes')
      dump_pragmas(func, pragmas, pragma_list)
      edges = ET.SubElement(function, 'Edges')
      dump_edges(func, edges, edge_list)

  tree = ET.ElementTree(root)
  indent(tree.getroot())
  tree.write('flow.xml')

def dump_pragmas(pragma_node, pragmas_element, pragma_list):
  for pragma in pragma_node.children:
    if str(pragma.start_line) + str(pragma.end_line) not in pragma_list:
      pragma_list.append(str(pragma.start_line) + str(pragma.end_line))
      pragma_ = ET.SubElement(pragmas_element, 'Pragma')
      pragma_.attrib['id'] = str(pragma.start_line) + str(pragma.end_line)
      name = ET.SubElement(pragma_, 'Name')
      if not "_end" in pragma.type:
        name.text = pragma.type
      else:
        name.text = "BARRIER"
      if(len(pragma.options) != 0):
        options = ET.SubElement(pragma_, 'Options')
        for op in pragma.options:
          option = ET.SubElement(options, 'Option')
          op_name = ET.SubElement(option, 'Name')
          op_name.text = op[0]
          for par in op[1]:
            op_parameter = ET.SubElement(option, 'Parameter')
            op_var = ET.SubElement(op_parameter, 'Var')
            op_var.text = par[1]
            op_type = ET.SubElement(op_parameter, 'Type')
            op_type.text = par[0]
      position = ET.SubElement(pragma_, 'Position')
      start = ET.SubElement(position, 'StartLine')
      start.text = pragma.start_line
      if(name.text != "BARRIER"):
        end = ET.SubElement(position, 'EndLine')
        end.text = pragma.end_line
```

```
545       if (pragma.callerid != None ):
546         callerids = ET.SubElement(pragma_, 'Callerids')
547         for id_ in pragma.callerid:
548           callerid = ET.SubElement(callerids, 'Callerid')
549           callerid.text = id_
550       if(pragma.time != 0):
551         time = ET.SubElement(pragma_, 'Time')
552         time.text = str(pragma.time)
553       if(pragma.variance != None):
554         variance = ET.SubElement(pragma_, 'Variance')
555         variance.text = str(pragma.variance)
556 
557     dump_pragmas(pragma, pragmas_element, pragma_list)
558 
559 def dump_edges(pragma_node, edges_element, pragma_list):
560   for pragma in pragma_node.children:
561     if pragma_node.start_line + pragma.start_line not in pragma_list:
562       pragma_list.append(pragma_node.start_line+pragma.start_line)
563       edge = ET.SubElement(edges_element, 'Edge')
564       source = ET.SubElement(edge, 'Source')
565       source.text = str(pragma_node.start_line) + str(pragma_node.end_line)
566       dest = ET.SubElement(edge, 'Dest')
567       dest.text = str(pragma.start_line) + str(pragma.end_line)
568     dump_edges(pragma,edges_element, pragma_list)
569 
570 def find_node(node, flow_graphs):
571   for function in flow_graphs:
572     tmp_node = find_sub_node(node, function)
573     if tmp_node != None :
574       return tmp_node
575 
576 def find_node2(key_start, key_parent, flow_graphs):
577   tmp_node = find_sub_node2(key_start, key_parent, flow_graphs)
578   if tmp_node != None :
579     return tmp_node
580 
581 def find_sub_node2(key_start, key_parent, function):
582   if (function.start_line) == key_start and ('BARRIER' not in function.type)
     :
583       return function
584   for child in function.children:
585     if (child.start_line) == key_start and ('BARRIER' not in child.type) and
      child.parent[0].start_line == key_parent:
586       return child
587     else:
588       tmp_node = find_sub_node2(key_start, key_parent, child)
589     if tmp_node != None:
590       return tmp_node
591   return None
592 
593 def find_sub_node(node, function):
594   if (function.start_line) == node and ('BARRIER' not in function.type):
595       return function
596   for child in function.children:
597     if (child.start_line) == node and ('BARRIER' not in child.type):
598       return child
599     else:
600       tmp_node = find_sub_node(node, child)
601     if tmp_node != None:
```

```
602       return tmp_node
603   return None
604 
605 class Caller():
606   def __init__(self, original_caller, used_caller):
607     self.original_caller = original_caller
608     self.used_caller = used_caller
609     self.old_children = []
610 
611 #adding to the main graph all the function which are called taking care of multiple connections between pragma and caller
612 def explode_graph(flow_graphs):
613   setted_callers = {}
614   for function in flow_graphs:
615     count = 0
616     caller_list = function.callerid
617     if caller_list != None:
618       for caller in caller_list:
619         function_copy = copy.deepcopy(function)
620         count += 1
621         caller_node = find_node(caller, flow_graphs)
622         if caller_node.start_line not in setted_callers:
623           setted_callers[caller_node.start_line] = Caller(copy.copy(caller_node), caller_node)
624           function_copy.parent.append(caller_node)
625           children_list = []
626           for child in caller_node.children:
627             children_list.append(child)
628             child.parent.remove(caller_node)
629             setted_callers[caller_node.start_line].old_children.append(child)
630           caller_node.children = []
631           caller_node.children.append(function_copy)
632           last_node = sched.get_last(function_copy)
633           last_node.children = children_list
634           for child in children_list:
635             child.parent.append(last_node)
636         else:
637           children_list = []
638           for child in setted_callers[caller_node.start_line].old_children:
639             children_list.append(child)
640           function_copy.parent.append(setted_callers[caller_node.start_line].used_caller)
641           setted_callers[caller_node.start_line].used_caller.children.append(function_copy)
642           last_node = sched.get_last(function_copy)
643           last_node.children = children_list
644           for child in children_list:
645             child.parent.append(last_node)
646 
647 
648 def get_parameter(parameter):
649   if parameter.find('Type') != None:
650     type_ = parameter.find('Type').text
651   else:
652     type_ = 'None'
653   return (type_, parameter.find('Var').text)
654 
655 def create_map(optimal_flow):
```

```
  for_map = {}
  for flow in optimal_flow:
    for task in flow.tasks:
      if "splitted" in task.type:
        l = re.findall(r'\d+',task.type)
        id = str(l[0]) + "_" + str(l[2])
        if id in for_map:
          for_map[id].count += 1
          for_map[id].id.append(task.id)
        else:
          for_map[id] = Task(1, task.id)
  return for_map

def add_new_tasks(optimal_flow, main_flow):
  for_map = create_map(optimal_flow)
  for key in for_map:
    l = re.findall(r'\d+',key)
    node_to_replace = find_node2(l[0], l[1], main_flow)
    nodes_to_add = []

    for i in range(for_map[key].count):
      nodes_to_add.append(For_Node("splitted_" + node_to_replace.start_line
   + "." + str(i), node_to_replace.start_line, node_to_replace.init_type,
    node_to_replace.init_var, node_to_replace.init_value, node_to_replace.
    init_cond, node_to_replace.init_cond_value, node_to_replace.
    init_increment, node_to_replace.init_increment_value, node_to_replace.
    time, node_to_replace.variance, math.floor(float(node_to_replace.
    mean_loops) / (i + 1))))

    for parent in node_to_replace.parent:
      parent.children.remove(node_to_replace)
      for n in nodes_to_add:
        parent.add(n)
        n.id = for_map[key].id.pop(0)
        n.color = 'white'
        n.from_type = node_to_replace.type

    for child in node_to_replace.children:
      child.parent.remove(node_to_replace)
      for n in nodes_to_add:
        n.add(child)


def add_flow_id(optimal_flow, task_list):
  id_map = {}
  for flow in optimal_flow:
    for task in flow.tasks:
      if "splitted" not in task.type:
        if task.start_line not in id_map:
          id_map[task.start_line] = task.id
        else:
          id_map[task.start_line + str(1)] = task.id
  for task in task_list:
    if "splitted" not in task.type:
      if task.start_line in id_map:
        task.id = id_map[task.start_line]
        id_map.pop(task.start_line, None)
      else:
        task.id = id_map[task.start_line + str(1)]
```

2.3 Profiler

Code 2.3: profiler.py

```
from __future__ import with_statement
import os
import pargraph as par
import xml.etree.cElementTree as ET
import numpy
import re

def profileCreator(cycle, executable):
  pragma_times = {}
  function_times = {}
  j = 0
  param_string = ''

  if os.path.exists("./parameters.txt"):
    with open("./parameters.txt","r") as f:
      parameters = f.readlines()
    for s in parameters:
      param_string +=  s.strip()

  for i in range(cycle):
    print "profiling iteration: " + str((j + 1))
    os.system("./" + executable + " " + param_string + " >/dev/null")
    os.system("mv log_file.xml " + "./logfile%s.xml" % j)
    root = ET.ElementTree(file = "./logfile%s.xml" % j).getroot()

    for pragma in root.iter('Pragma'):
      key = pragma.attrib['fid'] + pragma.attrib['pid']
      if (key not in pragma_times):
        pragma_times[key] = par.Time_Node(int(pragma.attrib['fid']), int(
   pragma.attrib['pid']))
      if ('callerid' in pragma.attrib):
        if pragma.attrib['callerid'] not in pragma_times[key].caller_list:
          pragma_times[key].caller_list.append(pragma.attrib['callerid'])
      if ('loops' in pragma.attrib):
        pragma_times[key].loops.append(int(pragma.attrib['loops']))
      if ('time' in pragma.attrib):
        pragma_times[key].time = pragma.attrib['time']
      if ('childrenTime' in pragma.attrib):
        pragma_times[key].children_time.append(float(pragma.attrib['
   childrenTime']))
      pragma_times[key].times.append(float(pragma.attrib['elapsedTime']))

    for func in root.iter('Function'):
      key = func.attrib['fid']
      if (key in function_times):
        function_times[key].times.append(float(func.attrib['elapsedTime']))
      else:
        function_times[key] = par.Time_Node(int(func.attrib['fid']), 0)
        function_times[key].times.append(float(func.attrib['elapsedTime']))
      if ('callerid' in func.attrib):
        if int(func.attrib['callerid']) not in function_times[key].
   caller_list:
          function_times[key].caller_list.append(int(func.attrib['callerid
   ']))
      if ('time' in func.attrib):
        function_times[key].time = func.attrib['time']
```

```
53        if ('childrenTime' in func.attrib):
54          function_times[key].children_time.append(float(func.attrib['
      childrenTime']))
55
56       j += 1
57
58    num_cores = ET.ElementTree(file = "logfile0.xml").getroot().find('Hardware
      ').attrib['NumberofCores']
59    tot_memory = ET.ElementTree(file = "logfile0.xml").getroot().find('
      Hardware').attrib['MemorySize']
60
61    root = ET.Element('Log_file')
62    h = ET.SubElement(root, 'Hardware')
63    h1 = ET.SubElement(h, 'NumberofCores')
64    h2 = ET.SubElement(h, 'MemorySize')
65    h1.text = num_cores
66    h2.text = tot_memory
67
68    for key in function_times:
69      s = ET.SubElement(root, 'Function')
70      line = ET.SubElement(s, 'FunctionLine')
71      time = ET.SubElement(s, 'Time')
72      var = ET.SubElement(s, 'Variance')
73      if (len(function_times[key].caller_list) != 0 ):
74        callerid = ET.SubElement(s, 'CallerId')
75        callerid.text = str(function_times[key].caller_list)
76      if (len(function_times[key].children_time) != 0):
77        children_time = ET.SubElement(s, 'ChildrenTime')
78        children_time.text = str(numpy.mean(function_times[key].children_time)
      )
79      time.text = str(numpy.mean(function_times[key].times))
80      line.text = str(function_times[key].func_line)
81      var.text = str(numpy.std(function_times[key].times))
82
83    for key in pragma_times:
84      s = ET.SubElement(root, 'Pragma')
85      f_line = ET.SubElement(s, 'FunctionLine')
86      p_line = ET.SubElement(s, 'PragmaLine')
87      time = ET.SubElement(s, 'Time')
88      var = ET.SubElement(s, 'Variance')
89      if (len(pragma_times[key].loops) != 0):
90        loops = ET.SubElement(s, 'Loops')
91        loops.text = str(numpy.mean(pragma_times[key].loops))
92      if (len(pragma_times[key].caller_list) != 0 ):
93        callerid = ET.SubElement(s, 'CallerId')
94        callerid.text = str(pragma_times[key].caller_list)
95      if (len(pragma_times[key].children_time) != 0):
96        children_time = ET.SubElement(s, 'ChildrenTime')
97        children_time.text = str(numpy.mean(pragma_times[key].children_time))
98      time.text = str(numpy.mean(pragma_times[key].times))
99      f_line.text = str(pragma_times[key].func_line)
100     p_line.text = str(pragma_times[key].pragma_line)
101     var.text = str(numpy.std(pragma_times[key].times))
102
103   tree = ET.ElementTree(root)
104   par.indent(tree.getroot())
105   tree.write(executable + "_profile.xml")
106
107   return executable + "_profile.xml"
```

```
108
109 def add_profile_xml(profile_xml, xml_tree):
110   functions = getProfilesMap(profile_xml)
111   tree = ET.ElementTree(file = xml_tree)
112   root = tree.getroot()
113   type_ = ET.SubElement(root, 'GraphType')
114   type_.text = 'Code'
115
116   for func in root.findall('Function'):
117     key = func.find('Line').text
118     func_time =  ET.SubElement(func, 'Time')
119     func_time.text = str(functions[key].time)
120     func_variance = ET.SubElement(func, 'Variance')
121     func_variance.text = str(functions[key].variance)
122     if len(functions[key].callerid) > 0:
123       func_caller_ids = ET.SubElement(func, 'Callerids')
124       tmp_list = set(functions[key].callerid)
125       for id in tmp_list:
126         func_caller_id = ET.SubElement(func_caller_ids,'Callerid')
127         func_caller_id.text = id
128     for pragma in func.iter('Pragma'):
129       pragma_key = pragma.find('Position/StartLine').text
130       if pragma_key in functions[key].pragmas:
131         pragma_time = ET.SubElement(pragma, 'Time')
132         pragma_time.text = functions[key].pragmas[pragma_key][0]
133         pragma_variance = ET.SubElement(pragma, 'Variance')
134         pragma_variance.text = functions[key].pragmas[pragma_key][1]
135         if (functions[key].pragmas[pragma_key][2] != 0):
136           pragma_loops = ET.SubElement(pragma, 'Loops')
137           pragma_loops.text = functions[key].pragmas[pragma_key][2]
138         if  (functions[key].pragmas[pragma_key][3] != None):
139           pragma_callerid = ET.SubElement(pragma, 'Callerid')
140           pragma_callerid.text = functions[key].pragmas[pragma_key][3].
    replace('[','').replace(']','').replace('\'','')
141
142   par.indent(tree.getroot())
143   tree.write('code.xml')
144
145 def get_table(profile_xml):
146   tree = ET.ElementTree(file = profile_xml)
147   root = tree.getroot()
148   table = {}
149
150   for func in root.iter('Function'):
151     table[func.find('FunctionLine').text] = []
152     if func.find('CallerId') != None:
153       l = re.findall(r'\d+',func.find('CallerId').text)
154       for j in l:
155         table[func.find('FunctionLine').text].append(j)
156
157   return table
158
159 def getProfilesMap(profile_xml):
160   profile_graph_root = ET.ElementTree(file = profile_xml).getroot()
161
162   functions = {}
163   l = []
164
165   for func in profile_graph_root.findall('Function'):
```

```
    f = par.Function(func.find('Time').text, func.find('Variance').text,
    func.find('ChildrenTime').text)
    f.callerid = []
    if (func.find('CallerId') != None):
      l = re.findall(r'\d+',func.find('CallerId').text.replace("[","").
    replace("]",""))
      for id_ in l:
        f.callerid.append(id_)
    functions[func.find('FunctionLine').text] = f

  for pragma in profile_graph_root.findall('Pragma'):
    if pragma.find('CallerId') != None:
      callerid = pragma.find('CallerId').text.replace("[\'","").replace("\']
    ","")
    else:
      callerid = None
    if (pragma.find('Loops') != None):
      loops = pragma.find('Loops').text
    else :
      loops = 0
    functions[pragma.find('FunctionLine').text].add_pragma( (pragma.find('
    PragmaLine').text, pragma.find('Time').text, pragma.find('Variance').text
    , loops, callerid, pragma.find('ChildrenTime').text ))

  return functions
```

Code 2.4: appprofile.py

```
import sys
import pargraph as par
import copy
import schedule as sched
import profiler as pro
import time
import multiprocessing
import itertools
import random
import threading
import argparse

if __name__ == "__main__":
  import argparse

  parser = argparse.ArgumentParser(description='Profiler')
  parser.add_argument('--profile',help='Output profile description file')
  parser.add_argument('--executable',help='Executable to be used',required=
    True)
  parser.add_argument('--parameters',help='Parameters File defaults to
    parameters.txt',required=False,default="parameters.txt")
  parser.add_argument('--count',help='Repetitions',type=int,default=10)

  args = parser.parse_args()

  if args.profile == "":
    args.profile = app.executable + "_profile.xml"

  profile_xmldata = pro.profileCreator(args.count, args.executable,args.
    parameters,args.profile)
```

2.4 Scheduler

Code 2.5: schedule.py

```
import pargraph as par
import xml.etree.cElementTree as ET
import math
import copy
import time
import multiprocessing


class Queue():
  def __init__(self):
    self.q = multiprocessing.Queue()
    self.set = False

#returns the optimal flows
#if time is to big for the number of possible solutions it does not work.
#parallel version
def get_optimal_flow(flow_list, task_list, level, optimal_flow, NUM_TASKS,
    MAX_FLOWS, execution_time, q):
  if time.clock() < execution_time :
    curopt = get_cost(optimal_flow)
    cur = get_cost(flow_list)
    if len(flow_list) < MAX_FLOWS and len(task_list) != level and cur <=
    curopt :
      task_i = task_list[level]
      # test integrating the single task in each
      for flow in flow_list :
        flow.add_task(task_i)
        get_optimal_flow(flow_list, task_list, level + 1, optimal_flow,
    NUM_TASKS, MAX_FLOWS, execution_time, q)
        flow.remove_task(task_i)
      new_flow = par.Flow()
      new_flow.add_task(task_i)
      flow_list.append(new_flow)
      get_optimal_flow(flow_list, task_list, level + 1, optimal_flow,
    NUM_TASKS, MAX_FLOWS, execution_time, q)
      flow_list.remove(new_flow)

      if 'For' in task_i.type :
        #checks the possible splittings of the for node
        for i in range(2, MAX_FLOWS + 1):
          tmp_task_list = []
          #splits the for node in j nodes
          for j in range(0, i):
            task = par.For_Node("splitted_" + task_i.start_line + "." + str(
    j) + "_" + task_i.parent[0].start_line, task_i.start_line, task_i.
    init_type, task_i.init_var, task_i.init_value, task_i.init_cond, task_i.
    init_cond_value, task_i.init_increment, task_i.init_increment_value,
    task_i.time, task_i.variance, math.floor(float(task_i.mean_loops) / i))
            task.in_time = float(task_i.time) / i
            task_list.append(task)
            tmp_task_list.append(task)
          get_optimal_flow(flow_list, task_list, level + 1, optimal_flow,
    NUM_TASKS + i - 1, MAX_FLOWS, execution_time, q)
          for tmp_task in tmp_task_list:
            task_list.remove(tmp_task)
    else:
```

```
48        if len(task_list) == level and len(flow_list) == MAX_FLOWS and cur <=
    curopt:
49          if cur < curopt or (get_num_splitted(flow_list) > get_num_splitted(
    optimal_flow)):
50            #print "acutal cost: ", get_cost(flow_list), "optimal cost: ",
    get_cost(optimal_flow)
51            del optimal_flow[:]
52            id = 0
53            #print "newflowset:"
54            for flow in flow_list:
55              for task in flow.tasks:
56                task.id = id
57              id += 1
58              optimal_flow.append(copy.deepcopy(flow))
59            while( not q.q.empty() ):
60              q.q.get()
61            q.q.put(optimal_flow)
62
63  #sequential version
64  def get_optimal_flow_single(flow_list, task_list, level, optimal_flow,
    NUM_TASKS, MAX_FLOWS, execution_time):
65    if time.clock() < execution_time :
66      curopt = get_cost(optimal_flow)
67      cur = get_cost(flow_list)
68      if len(flow_list) < MAX_FLOWS and len(task_list) != level and cur <=
    curopt :
69        task_i = task_list[level]
70        # test integrating the single task in each
71        for flow in flow_list :
72          flow.add_task(task_i)
73          get_optimal_flow_single(flow_list, task_list, level + 1,
    optimal_flow, NUM_TASKS, MAX_FLOWS, execution_time)
74          flow.remove_task(task_i)
75        new_flow = par.Flow()
76        new_flow.add_task(task_i)
77        flow_list.append(new_flow)
78        get_optimal_flow_single(flow_list, task_list, level + 1, optimal_flow,
     NUM_TASKS, MAX_FLOWS, execution_time)
79        flow_list.remove(new_flow)
80
81        if 'For' in task_i.type :
82          #checks the possible splittings of the for node
83          for i in range(2, MAX_FLOWS + 1):
84            tmp_task_list = []
85            #splits the for node in j nodes
86            for j in range(0, i):
87              task = par.For_Node("splitted_" + task_i.start_line + "." + str(
    j) + "_" + task_i.parent[0].start_line, task_i.start_line, task_i.
    init_type, task_i.init_var, task_i.init_value, task_i.init_cond, task_i.
    init_cond_value, task_i.init_increment, task_i.init_increment_value,
    task_i.time, task_i.variance, math.floor(float(task_i.mean_loops) / i))
88              task.in_time = float(task_i.time) / i
89              task_list.append(task)
90              tmp_task_list.append(task)
91            get_optimal_flow_single(flow_list, task_list, level + 1,
    optimal_flow, NUM_TASKS + i - 1, MAX_FLOWS, execution_time)
92            for tmp_task in tmp_task_list:
93              task_list.remove(tmp_task)
94      else:
```

```
95        if len(task_list) == level and len(flow_list) == MAX_FLOWS and cur <=
      curopt:
96          if cur < curopt or (get_num_splitted(flow_list) > get_num_splitted(
      optimal_flow)) :
97            #print "acutal cost: ", get_cost(flow_list), "optimal cost: ",
      get_cost(optimal_flow)
98            del optimal_flow[:]
99            id = 0
100           #print "newflowset:"
101           for flow in flow_list:
102             for task in flow.tasks:
103               task.id = id
104             id += 1
105             optimal_flow.append(copy.deepcopy(flow))
106
107 def get_num_splitted(flow_list):
108   num = 0
109   for flow in flow_list:
110     for task in flow.tasks:
111       if 'splitted' in task.type:
112         num += 1
113   return num
114
115
116 #generator for the tasks of the graph
117 def generate_task(node):
118   if node.color == 'white':
119     node.color = 'black'
120     yield node
121     for n in node.children:
122       for node in generate_task(n):
123         yield node
124
125 def generate_list(l, node):
126   if node.color == 'white':
127     node.color = 'black'
128     l.append(node)
129     for n in node.children:
130       generate_list(l, n)
131
132 #returns the number or physical cores
133 def get_core_num(profile):
134   root = ET.ElementTree(file = profile).getroot()
135   return int(root.find('Hardware/NumberofCores').text)
136
137 #sets the color of each node to white
138 def make_white(node):
139   if node.color == 'black':
140     node.color = 'white'
141   for child in node.children:
142     make_white(child)
143
144 #returns the graph which contains the 'main' function
145 def get_main(exp_flows):
146   for i in range(len(exp_flows)):
147     if exp_flows[i].type == 'main':
148       return exp_flows[i]
149
150 #returns the last node of the input graph
```

```
151 def get_last(node):
152   if not node.children:
153     return node
154   else:
155      return get_last(node.children[0])
156 
157 #returns the children with the least deadline - computation_time
158 def get_min(node):
159   minimum = float("inf")
160   found = False
161   for child in node.children:
162     if child.d == None:
163       found = True
164   if found == False:
165     #print "setting: ",child.type,"@",child.start_line
166     for child in node.children:
167       min_tmp = child.d - float(child.in_time)
168       if min_tmp < minimum:
169         minimum = min_tmp
170     return minimum
171 
172 
173 #sets the deadline for each task
174 def chetto_deadlines(node):
175   if node.parent :
176     for p in node.parent:
177       p.d = get_min(p)
178     for p in node.parent:
179       chetto_deadlines(p)
180 
181 #applys the chetto algorithm to obtain the deadline and arrival time for
      each task
182 def chetto(flow_graph, deadline, optimal_flow):
183   node = get_last(flow_graph)
184   node.d = deadline
185   chetto_deadlines(node)
186   flow_graph.arrival = 0
187   chetto_arrival(flow_graph, optimal_flow)
188 
189 #gets the cost of the worst flow
190 def get_cost(flow_list):
191   if len(flow_list) == 0:
192     return float("inf")
193   else:
194     return max([flow.time for flow in flow_list])
195 
196 def chetto_arrival(node, optimal_flow):
197   if node.children :
198     for child in node.children:
199       if child.arrival == None and all_set(child) == True:
200         (a, d) = get_max(child, optimal_flow)
201         child.arrival = max(a, d)
202       chetto_arrival(child, optimal_flow)
203 
204 
205 def get_max(node, optimal_flow):
206   maximum_a = 0
207   maximum_d = 0
208   for p in node.parent:
```

```
209     if p.arrival > maximum_a and p.id == node.id:
210       maximum_a = p.arrival
211     if p.d > maximum_d and p.id != node.id:
212       maximum_d = p.d
213   return (maximum_a, maximum_d)
214 
215 #checks if all the parent nodes have the arrival times set
216 def all_set(node):
217   found = True
218   for p in node.parent:
219     if p.arrival == None:
220       found = False
221   return found
222 
223 def get_id(node, optimal_flow):
224   for flow in optimal_flow:
225     for task in flow.tasks:
226       if node.type == task.type:
227         return flow.id
228 
229 def print_schedule(node):
230   if node.color == 'white':
231     node.color = 'black'
232     print node.type,"␣@␣", node.start_line
233     print "\t␣start:␣", node.arrival
234     print "\t␣deadline:␣", node.d
235     print "\t␣flow:␣", node.id
236   for n in node.children:
237     print_schedule(n)
238 
239 def create_schedule(graph, num_cores):
240   mapped = []
241   schedule = ET.Element('Schedule')
242   cores = ET.SubElement(schedule, 'Cores')
243   cores.text = str(num_cores)
244   make_white(graph)
245   task_list = generate_task(graph)
246   tree = ET.ElementTree(schedule)
247   for task in task_list:
248     if 'splitted' in task.type:
249       serialize_splitted(task, schedule, mapped)
250     elif 'BARRIER' not in task.type:
251       pragma = ET.SubElement(schedule, 'Pragma')
252       id = ET.SubElement(pragma, 'id')
253       id.text = str(task.start_line)
254       caller_id = ET.SubElement(pragma, 'Caller_id')
255       if(len(task.parent) > 0):
256         caller_id.text = str(task.parent[0].start_line)
257       else:
258         caller_id.text = str(0)
259       pragma_type = ET.SubElement(pragma, 'Type')
260       pragma_type.text = str(task.type)
261       threads = ET.SubElement(pragma, 'Threads')
262       thread = ET.SubElement(threads, 'Thread')
263       thread.text = str(task.id)
264       start = ET.SubElement(pragma, 'Start_time')
265       start.text = str(task.arrival)
266       end = ET.SubElement(pragma, 'Deadline')
267       end.text = str(task.d)
```

```
268       created = False
269       if 'BARRIER' not in task.children[0].type :
270         l = []
271         if 'Parallel' in task.type:
272           barrier = ET.SubElement(pragma, 'Barrier')
273           created = True
274           first = ET.SubElement(barrier, 'id')
275           first.text = str(task.start_line)
276         if not ('OMPParallelForDirective' in task.type and 'Parallel' in
    task.children[0].type) and not isinstance(task.children[0], par.Fx_Node):
277           if created == False:
278             barrier = ET.SubElement(pragma, 'Barrier')
279             created = True
280           for c in task.children:
281             if c.start_line not in l:
282               tmp_id = ET.SubElement(barrier, 'id')
283               tmp_id.text = str(c.start_line)
284               l.append(c.start_line)
285       elif ('OMPParallelForDirective' in task.type and 'BARRIER' in task.
    children[0].type):
286         if created == False:
287             barrier = ET.SubElement(pragma, 'Barrier')
288             created = True
289         first = ET.SubElement(barrier, 'id')
290         first.text = str(task.start_line)
291   par.indent(tree.getroot())
292   tree.write('schedule.xml')
293
294 def serialize_splitted(task, schedule, mapped):
295   if task.start_line not in mapped:
296     pragma = ET.SubElement(schedule, 'Pragma')
297     id = ET.SubElement(pragma, 'id')
298     id.text = str(task.start_line)
299     caller_id = ET.SubElement(pragma, 'Caller_id')
300     if(len(task.parent) > 0):
301       caller_id.text = str(task.parent[0].start_line)
302     else:
303       caller_id.text = str(0)
304     pragma_type = ET.SubElement(pragma, 'Type')
305     pragma_type.text = str(task.from_type)
306     threads = ET.SubElement(pragma, 'Threads')
307     thread = ET.SubElement(threads, 'Thread')
308     thread.text = str(task.id)
309     start = ET.SubElement(pragma, 'Start_time')
310     start.text = str(task.arrival)
311     end = ET.SubElement(pragma, 'Deadline')
312     end.text = str(task.d)
313     mapped.append(task.start_line)
314     if 'BARRIER' not in task.children[0].type :
315       l = []
316       barrier = ET.SubElement(pragma, 'Barrier')
317       if 'Parallel' in task.from_type:
318         first = ET.SubElement(barrier, 'id')
319         first.text = str(task.start_line)
320       if not ('OMPParallelForDirective' in task.from_type and 'Parallel' in
    task.children[0].type):
321         for c in task.children:
322           if c.start_line not in l:
323             tmp_id = ET.SubElement(barrier, 'id')
```

```
324             tmp_id.text = str(c.start_line)
325             l.append(c.start_line)
326     elif ('OMPParallelForDirective' in task.from_type and 'BARRIER' in task.
      children[0].type):
327       barrier = ET.SubElement(pragma, 'Barrier')
328       first = ET.SubElement(barrier, 'id')
329       first.text = str(task.start_line)
330   else:
331     for p in schedule.findall("Pragma"):
332       if p.find('id').text == task.start_line:
333         threads_ = p.find('Threads')
334         thread = ET.SubElement(threads_, 'Thread')
335         thread.text = str(task.id)
336
337 def check_schedule(main_flow):
338   make_white(main_flow)
339   gen = generate_task(main_flow)
340   for node in gen:
341     if node.d < 0:
342       return False
343   return True
```