

Contents

1	C++	3
1.1	Include files	3
1.2	Source files	10
1.3	Run-time	39
1.3.1	Profiler	39
1.3.2	Final exectution	43
2	Python	53

Chapter 1

C++

1.1 Include files

Code 1.1: driver/compiler.h

```
1 #include "llvm/Support/Host.h"
2 #include "llvm/ADT/IntrusiveRefCntPtr.h"
3
4 #include "clang/Frontend/CompilerInstance.h"
5 #include "clang/Basic/TargetOptions.h"
6 #include "clang/Basic/TargetInfo.h"
7 #include "clang/Basic/FileManager.h"
8 #include "clang/Basic/SourceManager.h"
9 #include "clang/Lex/Preprocessor.h"
10 #include "clang/Lex/Lexer.h"
11 #include "clang/Basic/Diagnostic.h"
12 #include "clang/AST/ASTContext.h"
13
14 /*
15  * ---- Custom class to instantiate an object of clang::CompilerInstance
16  *      with the options and the file
17  *      passed with argv.
18  */
19
20 class ClangCompiler {
21 private:
22     clang::CompilerInstance compiler_;
23
24 public:
25     ClangCompiler(int argc, char **argv);
26
27     clang::SourceManager &getSourceManager() { return compiler_.
28         getSourceManager(); }
29     clang::DiagnosticConsumer getDiagnosticClient() { return compiler_.
30         getDiagnosticClient(); }
31     clang::LangOptions getLangOpts() { return compiler_.getLangOpts(); }
32     clang::Preprocessor &getPreprocessor() { return compiler_.getPreprocessor
33         (); }
34     clang::ASTContext &getASTContext() { return compiler_.getASTContext(); }
35     clang::FileManager &getFileManager() { return compiler_.getFileManager(); }
36 };
```

Code 1.2: driver/program.h

```

1 #include "driver/compiler.h"
2 #include "utils/source_locations.h"
3 #include "pragma_handler/Root.h"
4
5 #include "clang/StaticAnalyzer/Core/PathSensitive/CheckerContext.h"
6 #include "clang/Basic/DiagnosticOptions.h"
7 #include "clang/Frontend/TextDiagnosticPrinter.h"
8 #include "clang/AST/ASTConsumer.h"
9 #include "clang/Parse/Parser.h"
10 #include "clang/Parse/ParseAST.h"
11 #include "clang/Rewrite/Core/Rewriter.h"
12 #include "llvm/Support/raw_ostream.h"
13 #include <string>
14 #include <iostream>
15
16 /*
17  * ---- Instantiate a compiler object and start the parser.
18  */
19 class Program {
20 /* Contains the list of all the pragmas in the source code */
21 std::vector<clang::OMPExecutableDirective *> *pragma_list_;
22 /* Contains the list of all the functions defined in the source code (for
   profiling purpose) */
23 std::vector<clang::FunctionDecl *> *function_list_;
24
25 /* To create the profiling code and the list of pragmas */
26 void ParseSourceCode(std::string fileName);
27 /* To create the final source code to be used with the scheduler */
28 void ParseSourceCode(std::string fileName, std::vector<Root *> *root_vect);
29
30 public:
31 /* To create the profiling code and the list of pragmas */
32 Program(int argc, char **argv) : ccompiler_(argc, argv), pragma_list_(NULL
   ), function_list_(NULL) {
33     ParseSourceCode(argv[argc - 1]);
34 }
35
36 /* To create the final source code to be used with the scheduler */
37 Program(int argc, char **argv, std::vector<Root *> *root_vect) : ccompiler_
   (argc, argv), pragma_list_(NULL), function_list_(NULL) {
38     ParseSourceCode(argv[argc - 1], root_vect);
39 }
40
41 std::vector<clang::OMPExecutableDirective *> *getPragmaList() { return
   pragma_list_; }
42 std::vector<clang::FunctionDecl *> *getFunctionList() { return
   function_list_; }
43
44 ClangCompiler ccompiler_;
45 };
46
47 /*
48  * ---- Recursively visit the AST of the source code to extract the pragmas
   and rewrite it
49  *
50  * adding profile call.
51  */
52 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
65    how many cycles will
66    do the for */
67 std::string ForConditionVarValue(const clang::Stmt *s);
68 /* For a given stmt retrieve the line of the function where it is defined
69    */
70 unsigned GetFunctionLineForPragma(clang::SourceLocation sl);
71
72 public:
73 ProfilingRecursiveASTVisitor(clang::Rewriter &r_profiling, const clang::
74    SourceManager& sm) :
75     rewrite_profiling_(r_profiling), sm(sm), include_inserted_(false),
76     previous_stmt_(NULL) { }
77
78 /* This function is called for each stmt in the AST */
79 bool VisitStmt(clang::Stmt *s);
80 /* This function is called for each function in the AST */
81 bool VisitFunctionDecl(clang::FunctionDecl *f);
82 bool VisitDecl(clang::Decl *decl);
83 std::vector<clang::OMPExecutableDirective *> pragma_list_;
84 std::vector<clang::FunctionDecl *> function_list_;
85
86 };
87
88 /*
89 * ---- Is responsible to call ProfilingRecurseASTVisitor.
90 */
91 class ProfilingASTConsumer : public clang::ASTConsumer {
92 public:
93 ProfilingASTConsumer(clang::Rewriter &r_profiling, const clang::
94    SourceManager& sm) :
95     recursive_visitor_(r_profiling, sm) { }
96
97 /* Traverse the AST invoking the RecursiveASTVisitor functions */
98 virtual bool HandleTopLevelDecl(clang::DeclGroupRef d) {
99     typedef clang::DeclGroupRef::iterator iter;
100     for (iter b = d.begin(), e = d.end(); b != e; ++b) {
101         recursive_visitor_.TraverseDecl(*b);
102     }
103     return true;
104 }
105
106 ProfilingRecursiveASTVisitor recursive_visitor_;
107 std::vector<clang::OMPExecutableDirective *> pragma_list_;
108 std::vector<clang::FunctionDecl *> function_list_;
109 };
110
111

```

```

107  /*
108  /*
109  * ---- Recursively visit the AST and replace each pragma with a function
110  call.
111  */
112 class TransformRecursiveASTVisitor: public clang::RecursiveASTVisitor<
113     TransformRecursiveASTVisitor> {
114
115     clang::Rewriter &rewrite_pragma_;
116
117     const clang::SourceManager& sm;
118
119     /* Needed because the parse retrieve twice each pragma stmt */
120     clang::Stmt *previous_stmt_;
121     /* Check if the include command has been already inserted*/
122     bool include_inserted_;
123
124     std::vector<Root *> *root_vect_;
125
126     void RewriteOMPPPragma(clang::Stmt *associated_stmt, std::string
127         pragma_name);
128     void RewriteOMPBarrier(clang::OMPExecutableDirective *omp_stmt);
129     std::string RewriteOMPFor(Node *n);
130
131     /* Given a pragma stmt retrieve the Node object that contains all its info
132     */
133     Node *GetNodeObjForPragma(clang::Stmt *s);
134     /* Called by GetNodeObjForPragma is used because the Node objs are saved
135     in a tree */
136     Node *RecursiveGetNodeObjforPragma(Node *n, unsigned stmt_start_line);
137
138 public:
139     TransformRecursiveASTVisitor(clang::Rewriter &r_pragma_, std::vector<Root
140         *> *root_vect, const clang::SourceManager& sm) :
141         rewrite_pragma_(r_pragma_), root_vect_(root_vect), sm(sm),
142         include_inserted_(false), previous_stmt_(NULL) { }
143
144     bool VisitStmt(clang::Stmt *s);
145     bool VisitFunctionDecl(clang::FunctionDecl *f);
146     bool VisitDecl(clang::Decl *decl);
147 };
148
149 /*
150 * ---- Responsible to invoke TransformRecursiveASTVisitor.
151 */
152 class TransformASTConsumer : public clang::ASTConsumer {
153 public:
154
155     TransformASTConsumer(clang::Rewriter &RPragma, std::vector<Root *> *
156         rootVect, const clang::SourceManager& sm) :
157         recursive_visitor_(RPragma, rootVect, sm) { }
158
159     virtual bool HandleTopLevelDecl(clang::DeclGroupRef d) {
160         typedef clang::DeclGroupRef::iterator iter;
161         for (iter b = d.begin(), e = d.end(); b != e; ++b) {
162             recursive_visitor_.TraverseDecl(*b);
163         }
164         return true;
165     }
166 }

```

```

158 }
159
160 TransformRecursiveASTVisitor recursive_visitor_;
161 };

```

Code 1.3: pragma_handler/Node.h

```

1 #include "pragma_handler/ForNode.h"
2
3 /* Contains info about function */
4 struct FunctionInfo {
5     clang::FunctionDecl *function_decl_;
6
7     unsigned function_start_line_;
8     unsigned function_end_line_;
9     std::string function_name_;
10    std::string function_return_type_;
11    int num_params_;
12    /* Matrix Nx2. Contains the list of the parameter of the functions: type
13       name */
14    std::string **function_parameters_;
15    std::string function_class_name_;
16 };
17
18 /*
19  * ---- Contains all the relevant information of a given pragma.
20  */
21 class Node {
22
23 private:
24
25     clang::OMPExecutableDirective *pragma_stmt_;
26
27     /* Stmt start and end line in the source file */
28     std::string file_name_;
29     int start_line_, start_column_;
30     int end_line_, end_column_;
31
32     /*Line number of the function that contains this pragma */
33     FunctionInfo parent_func_info_;
34
35     /* Variables to construct the tree */
36     Node *parent_node_;
37
38     /*Pragma name with all the parameters */
39     //std::string pragma_type_;
40
41     /* Function to extract all the parameters of the pragma */
42     void setPragmaClauses(clang::SourceManager& sm);
43
44 public:
45     /*Pragma name with all the parameters */
46     std::string pragma_type_;
47
48     bool profiled_ = false;
49
50     ForNode *for_node_;
51
52     std::vector<Node *> *children_vect_;

```

```

53 typedef std::map<std::string, std::string> VarList_;
54 std::map<std::string, VarList_> *option_vect_;
55
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_func_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_;

```



```

27 bool loop_var_init_val_set_;
28 std::string loop_var_init_var_;
29
30 /* Loop condition */
31 std::string condition_op_;
32 int condition_val_;
33 bool condition_val_set_;
34 std::string condition_var_;
35
36 /* Loop increment */
37 std::string increment_op_;
38 int increment_val_;
39 bool increment_val_set_;
40 std::string increment_var_;
41
42 void ExtractForParameters(clang::ForStmt *for_stmt);
43
44 void ExtractForInitialization(clang::ForStmt *for_stmt);
45 void ExtractForCondition(clang::ForStmt *for_stmt);
46 void ExtractForIncrement(clang::ForStmt *for_stmt);
47
48
49 ForNode(clang::ForStmt *for_stmt);
50 void CreateXMLPragmaFor(tinyxml2::XMLDocument *xml_doc, tinyxml2::
    XMLElement *for_element);
51
52 };

```

Code 1.5: pragma_handler/Root.h

```

1 #include "pragma_handler/Node.h"
2
3 /*
4  * ---- It's the root node of the annidation tree of the pragmas in a
5  *      specific function
6  *      and contains the first level pragmas.
7  */
8 class Root {
9 private:
10     FunctionInfo function_info_;
11
12     Node *last_node_;
13 public:
14     Root(Node *n, FunctionInfo funct_info);
15
16     std::vector<Node *> *children_vect_;
17
18     void setLastNode(Node *n) {last_node_ = n; };
19     Node* getLastNode() { return last_node_; };
20
21     void AddChildNode(Node *n) { children_vect_->push_back(n); };
22
23     void CreateXMLFunction(tinyxml2::XMLDocument *xml_doc);
24
25     unsigned getFunctionLineStart(){ return function_info_.
        function_start_line_; }
26     unsigned getFunctionLineEnd() {return function_info_.function_end_line_; }
27
28 };

```

Code 1.6: utils/source_locations.h

```

1 #include <string>
2 #include <clang/Basic/SourceLocation.h>
3 #include <clang/Basic/SourceManager.h>
4 #include <sstream>
5
6 #include <llvm/Support/raw_ostream.h>
7
8 namespace clang {
9 class SourceLocation;
10 class SourceRange;
11 class SourceManager;
12 }
13
14 namespace utils {
15
16 std::string FileName(clang::SourceLocation const& l, clang::SourceManager
    const& sm);
17
18 std::string FileId(clang::SourceLocation const& l, clang::SourceManager
    const& sm);
19
20 unsigned Line(clang::SourceLocation const& l, clang::SourceManager const& sm
    );
21
22 std::pair<unsigned, unsigned> Line(clang::SourceRange const& r, clang::
    SourceManager const& sm);
23
24 unsigned Column(clang::SourceLocation const& l, clang::SourceManager const&
    sm);
25
26 std::pair<unsigned, unsigned> Column(clang::SourceRange const& r, clang::
    SourceManager const& sm);
27
28 std::string location(clang::SourceLocation const& l, clang::SourceManager
    const& sm);
29
30 }

```

1.2 Source files

Code 1.7: main.cpp

```

1 int main(int argc, char **argv) {
2
3     if(argc < 2) {
4         llvm::errs() << "Usage: □Source_extractor□[<options>]□<filename>\n";
5         return 1;
6     }
7     /*
8     * ---- Create a clang::compiler object and launch the parser saving the
9     *      pragma stmt.
10    *      Rewrite the sourcecode adding profiling call.
11    */
12    Program p_parser(argc, argv);
13    /*

```

```

14  * ---- With the information extracted by the parser create a linked list
      tree of objects containing
15  *      all the necessary information of the pragmas.
16  */
17  std::vector<Root *> *root_vect = CreateTree(program.getPragmaList(),
      program.getFunctionList(), program.ccompiler_.getSourceManager());
18  /*
19  * ---- Using the tree above create an xml file containing the pragma info.
      This file is used to produce the scheduler.
20  */
21  CreateXML(root_vect, argv[argc - 1]);
22
23  for(std::vector<Root *>::iterator itr = root_vect->begin(); itr !=
      root_vect->end(); ++itr)
24      (*itr)->VisitTree();
25
26  /*
27  * ---- Parse the sourcecode and rewrite it substituting pragmas with
      function calls. This new file
28  *      will be used with the scheduler to produce the final output.
29  */
30  Program p_rewriter(argc, argv, root_vect);
31
32  return 0;
33  }

```

Code 1.8: driver/compiler.cpp

```

1  #include "driver/compiler.h"
2
3  using namespace clang;
4
5  ClangCompiler::ClangCompiler(int argc, char **argv) {
6
7      DiagnosticOptions diagnosticOptions;
8      compiler_.createDiagnostics();
9
10     /* Create an invocation that passes any flags to preprocessor */
11     CompilerInvocation *Invocation = new CompilerInvocation;
12     CompilerInvocation::CreateFromArgs(*Invocation, argv + 1, argv + argc,
13                                         compiler_.getDiagnostics());
14     compiler_.setInvocation(Invocation);
15
16     /* Set default target triple */
17     llvm::IntrusiveRefCntPtr<TargetOptions> pto( new TargetOptions());
18     pto->Triple = llvm::sys::getDefaultTargetTriple();
19     llvm::IntrusiveRefCntPtr<TargetInfo> pti(TargetInfo::CreateTargetInfo(
20         compiler_.getDiagnostics(), pto.getPtr()));
21     compiler_.setTarget(pti.getPtr());
22
23     compiler_.createFileManager();
24     compiler_.createSourceManager(compiler_.getFileManager());
25
26     /* Add default search path for the compiler */
27     HeaderSearchOptions &headerSearchOptions = compiler_.getHeaderSearchOpts()
28     ;
29     headerSearchOptions.AddPath("/usr/local/include",
30                                clang::frontend::Angled,
31                                false,

```

```

31         false);
32
33     headerSearchOptions.AddPath("/usr/include",
34         clang::frontend::Angled,
35         false,
36         false);
37
38     headerSearchOptions.AddPath("/usr/lib/gcc/x86_64-linux-gnu/4.8/include",
39         clang::frontend::Angled,
40         false,
41         false);
42
43     headerSearchOptions.AddPath("/usr/include/x86_64-linux-gnu",
44         clang::frontend::Angled,
45         false,
46         false);
47     headerSearchOptions.AddPath("/usr/include/c++/4.8/",
48         clang::frontend::Angled,
49         false,
50         false);
51
52     headerSearchOptions.AddPath("/usr/include/x86_64-linux-gnu/c++/4.8/",
53         clang::frontend::Angled,
54         false,
55         false);
56
57
58     /* Allow C++ code to get rewritten */
59     clang::LangOptions langOpts;
60     langOpts.GNUMode = 1;
61     langOpts.CXXExceptions = 1;
62     langOpts.RTTI = 1;
63     langOpts.Bool = 1;
64     langOpts.CPlusPlus = 1;
65     Invocation->setLangDefaults(langOpts,
66                                 clang::IK_CXX,
67                                 clang::LangStandard::lang_cxx0x);
68
69     compiler_.createPreprocessor();
70     compiler_.getPreprocessorOpts().UsePredefines = false;
71
72     compiler_.createASTContext();
73
74     /* Initialize the compiler and the source manager with a file to process
75      */
76     std::string fileName(argv[argc - 1]);
77     const FileEntry *pFile = compiler_.getFileManager().getFile(fileName);
78     compiler_.getSourceManager().createMainFileID(pFile);
79     compiler_.getDiagnosticClient().BeginSourceFile(compiler_.getLangOpts(), &
80     compiler_.getPreprocessor());
81 }

```

Code 1.9: driver/program.cpp

```

1 #include "driver/program.h"
2
3 void Program::ParseSourceCode(std::string file_name) {
4
5     /* 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(),
15      out_error_info, 0);
16
17  /* Create the rewriter object to create the profiling file */
18  clang::Rewriter rewrite_profiling;
19  rewrite_profiling.setSourceMgr(ccompiler_.getSourceManager(), ccompiler_.
20      getLangOpts());
21
22  ProfilingASTConsumer ast_consumer(rewrite_profiling, ccompiler_.
23      getSourceManager());
24  /* Parse the AST with the custom ASTConsumer */
25  clang::ParseAST(ccompiler_.getPreprocessor(), &ast_consumer, ccompiler_.
26      getASTContext());
27  ccompiler_.getDiagnosticClient().EndSourceFile();
28
29  /* Save the pragma and function list */
30  pragma_list_ = new std::vector<clang::OMPExecutableDirective *>(
31      ast_consumer.recursive_visitor_.pragma_list_);
32  function_list_ = new std::vector<clang::FunctionDecl *>(ast_consumer.
33      recursive_visitor_.function_list_);
34
35  /*Output rewritten source code into a new file */
36  const clang::RewriteBuffer *rewrite_buf_profiling =
37      rewrite_profiling.getRewriteBufferFor(ccompiler_.getSourceManager().
38          getMainFileID());
39
40  out_file_profile << std::string(rewrite_buf_profiling->begin(),
41      rewrite_buf_profiling->end());
42  out_file_profile.close();
43
44  }
45
46  bool ProfilingRecursiveASTVisitor::VisitDecl(clang::Decl *decl) {
47
48      clang::SourceLocation cxx_start_src_loc = decl->getLocStart();
49      if(sm.getFileID(cxx_start_src_loc) == sm.getMainFileID()
50          && clang::isa<clang::CXXRecordDecl>(decl)
51          && include_inserted_ == false) {
52          include_inserted_ = true;
53          std::string text_include =
54              "#include \"profile_tracker/profile_tracker.h\"\n";
55          rewrite_profiling_.InsertText(cxx_start_src_loc, text_include, true,
56              false);
57      }
58
59      return true;
60  }
61
62  /*
63   * ---- Insert the call to the profilefunction tracker to track the
64   * execution time of each function.

```

```

55  */
56  bool ProfilingRecursiveASTVisitor::VisitFunctionDecl(clang::FunctionDecl *f)
57  {
58      clang::SourceLocation start_src_loc = f->getLocStart();
59      unsigned funct_start_line = utils::Line(start_src_loc, sm);
60
61
62      /* Skip function belonging to external include file and not defined
        function */
63      if(sm.getFileID(start_src_loc) == sm.getMainFileID() && f->hasBody() ==
        true) {
64
65          function_list_.push_back(f);
66
67          /* Include the path to ProfileTracker.h */
68          if(include_inserted_ == false) {
69              std::string text_include =
70                  "#include <\"profile_tracker/profile_tracker.h\">\n";
71
72              rewrite_profiling_.InsertText(start_src_loc, text_include, true, false
        );
73              include_inserted_ = true;
74          }
75
76          start_src_loc = f->getBody()->getLocStart();
77          unsigned start_line = utils::Line(start_src_loc, sm);
78          clang::SourceLocation new_start_src_loc = sm.translateLineCol(sm.
        getMainFileID(), start_line + 1, 1);
79          std::stringstream text_profiling;
80          text_profiling << "if(<ProfileTracker<_x=<ProfileTrackParams(" <<
        funct_start_line << ",<0))<{\n";
81
82          /* Insert the if in the first line of the function definition */
83          rewrite_profiling_.InsertText(new_start_src_loc, text_profiling.str(),
        true, false);
84
85          clang::SourceLocation end_src_loc = f->getLocEnd();
86          std::stringstream text_end_bracket;
87          text_end_bracket << "}<\n";
88          /* Close the if bracket at the end of the function */
89          rewrite_profiling_.InsertText(end_src_loc, text_end_bracket.str(), true,
        false);
90      }
91
92      return true;
93  }
94
95  bool ProfilingRecursiveASTVisitor::VisitStmt(clang::Stmt *s) {
96
97      clang::SourceLocation start_src_loc = s->getLocStart();
98      if(sm.getFileID(start_src_loc) == sm.getMainFileID()) {
99          /* We want just the OpenMP stmt and no duplicate */
100         if (clang::isa<clang::OMPExecutableDirective>(s) && s !=
        previous_stmt_) {
101             previous_stmt_ = s;
102             clang::OMPExecutableDirective *omp_stmt = static_cast<clang::
        OMPExecutableDirective *>(s);
103             pragma_list_.push_back(omp_stmt);

```

```

104         clang::Stmt *associated_stmt = omp_stmt->getAssociatedStmt();
105         if(associated_stmt) {
106             clang::Stmt *captured_stmt = static_cast<clang::CapturedStmt *>(
107                 associated_stmt->getCapturedStmt());
108             /* In the case of #omp parallel for we have to go down two level
109                before finding the ForStmt */
110             if(strcmp(captured_stmt->getStmtClassName(), "OMPForDirective") !=
111                0)
112                 RewriteProfiling(captured_stmt);
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
125     std::stringstream text_profiling;
126     if(clang::isa<clang::ForStmt>(s)) {
127         std::string condition_var_value = ForConditionVarValue(s);
128         //std::string conditionVar = "";
129         text_profiling << "if(␣ProfileTracker␣x␣=␣ProfileTrackParams("
130             << function_start_line << ",␣" << pragma_start_line << ",␣" <<
131             condition_var_value << "))\n";
132         rewrite_profiling_.InsertText(start_src_loc, text_profiling.str(),
133             true, true);
134
135     } else {
136         text_profiling << "if(␣ProfileTracker␣x␣=␣ProfileTrackParams("
137             << function_start_line << ",␣" << pragma_start_line << "))\n";
138         rewrite_profiling_.InsertText(start_src_loc, text_profiling.str(),
139             true, true);
140     }
141
142     /* Comment the pragma in the profiling file */
143     clang::SourceLocation pragma_start_src_loc =
144         sm.translateLineCol(sm.getMainFileID(), pragma_start_line - 1, 1);
145     rewrite_profiling_.InsertText(pragma_start_src_loc, "//", true, false);
146 }
147
148 std::string ProfilingRecursiveASTVisitor::ForConditionVarValue(const clang::
149     Stmt *s) {
150
151     const clang::ForStmt *for_stmt = static_cast<const clang::ForStmt *>(s);
152     const clang::Expr *condition_expr = for_stmt->getCond();
153     const clang::BinaryOperator *binary_op = static_cast<const clang::
154         BinaryOperator *>(condition_expr);
155
156     std::string start_cond_var_value, end_cond_var_value;
157
158     /*

```

```

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::
160     IntegerLiteral *>(right_expr);
161     std::stringstream text_end_value;
162     text_end_value << int_literal->getValue().getZExtValue();
163     //return text.str();
164     end_cond_var_value = text_end_value.str();
165
166 } else if(strcmp(right_expr->getStmtClassName(), "ImplicitCastExpr") == 0)
167 {
168     const clang::DeclRefExpr *decl_ref_expr =
169         static_cast<const clang::DeclRefExpr *>(*(right_expr->child_begin())
170 );
171
172     const clang::NamedDecl *named_decl = decl_ref_expr->getFoundDecl();
173     //return nD->getNameAsString();
174     end_cond_var_value = named_decl->getNameAsString();
175 }
176
177 /*
178 * Condition start value
179 */
180
181 /*
182 * for (int i = ...)
183 */
184 if(strcmp(for_stmt->child_begin()->getStmtClassName(), "DeclStmt") == 0) {
185     const clang::DeclStmt *decl_stmt = static_cast<const clang::DeclStmt
186     *>(*(for_stmt->child_begin()));
187     const clang::Decl *decl = decl_stmt->getSingleDecl();
188
189 /*
190 * for (... = 0)
191 */
192 if(strcmp(decl_stmt->child_begin()->getStmtClassName(), "IntegerLiteral"
193 ) == 0) {
194     const clang::IntegerLiteral *int_literal =
195         static_cast<const clang::IntegerLiteral *>(*(decl_stmt->
196 child_begin()));
197
198     std::stringstream text_star_value;
199     text_star_value << int_literal->getValue().getZExtValue();
200     start_cond_var_value = text_star_value.str();
201
202 /*
203 * for (... = a)
204 */
205 }else if (strcmp(decl_stmt->child_begin()->getStmtClassName(), "
206 ImplicitCastExpr") == 0) {
207     const clang::DeclRefExpr *decl_ref_expr =
208         static_cast<const clang::DeclRefExpr *>(*(decl_stmt->child_begin()
209 ->child_begin()));
210
211     const clang::NamedDecl *named_decl = decl_ref_expr->getFoundDecl();
212     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("\u-");
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 void Program::ParseSourceCode(std::string fileName, std::vector<Root *> *
256     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(),
268         out_error_info, 0);
269
270     clang::Rewriter rewrite_pragma;
271     rewrite_pragma.setSourceMgr(ccompiler_.getSourceManager(), ccompiler_.
272         getLangOpts());
273
274     TransformASTConsumer t_ast_consumer(rewrite_pragma, root_vect, ccompiler_.
275         getSourceManager());
276
277     /* Parse the AST */
278     clang::ParseAST(ccompiler_.getPreprocessor(), &t_ast_consumer, ccompiler_.
279         getASTContext());
280     ccompiler_.getDiagnosticClient().EndSourceFile();
281
282     const clang::RewriteBuffer *rewrite_buff_pragma =
283         rewrite_pragma.getRewriteBufferFor(ccompiler_.getSourceManager().
284             getMainFileID());
285     out_file_pragma << std::string(rewrite_buff_pragma->begin(),
286         rewrite_buff_pragma->end());
287     out_file_pragma.close();
288 }
289
290 bool TransformRecursiveASTVisitor::VisitDecl(clang::Decl *decl) {
291
292     clang::SourceLocation cxx_start_src_loc = decl->getLocStart();
293     if(sm.getFileID(cxx_start_src_loc) == sm.getMainFileID()
294         && clang::isa<clang::CXXRecordDecl>(decl)
295         && include_inserted_ == false) {
296         include_inserted_ = true;
297         std::string text_include = "#include \"thread_pool/threads_pool.h\"";
298         rewrite_pragma.InsertText(cxx_start_src_loc, text_include, true, false)
299         ;
300     }
301     return true;
302 }
303
304 bool TransformRecursiveASTVisitor::VisitFunctionDecl(clang::FunctionDecl *f)
305     {
306     clang::SourceLocation f_start_src_loc = f->getLocStart();
307
308     if(sm.getFileID(f_start_src_loc) == sm.getMainFileID() && !clang::isa<
309         clang::CXXMethodDecl>(f)) {
310         if(include_inserted_ == false) {
311             include_inserted_ = true;

```

```

304     std::string text_include = "#include_\\"thread_pool/threads_pool.h\\"\\n"
305     ;
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
318        pragma stmt */
319     if(sm.getFileID(s_start_stc_loc) == sm.getMainFileID()
320         && clang::isa<clang::OMPExecutableDirective>(s)
321         && s != previous_stmt_) {
322
323         previous_stmt_ = s;
324         clang::OMPExecutableDirective *omp_stmt = static_cast<clang::
325 OMPExecutableDirective *>(s);
326         clang::Stmt *associated_stmt = omp_stmt->getAssociatedStmt();
327         if(associated_stmt) {
328             clang::Stmt *captured_stmt = static_cast<clang::CapturedStmt *>(
329 associated_stmt->getCapturedStmt());
330             if(strcmp(captured_stmt->getStmtClassName(), "OMPForDirective") != 0)
331                 RewriteOMPPragma(associated_stmt, omp_stmt->getStmtClassName());
332
333             }else if(strcmp(omp_stmt->getStmtClassName(), "OMPBarrierDirective") ==
334 0
335                 || strcmp(omp_stmt->getStmtClassName(), "OMPTaskwaitDirective")
336 == 0){
337                 RewriteOMPBarrier(omp_stmt);
338             }
339         }
340     }
341     return true;
342 }
343
344 void TransformRecursiveASTVisitor::RewriteOMPBarrier(clang::
345 OMPExecutableDirective *omp_stmt) {
346     unsigned stmt_start_line = utils::Line(omp_stmt->getLocStart(), sm);
347
348     std::stringstream text_barrier;
349     text_barrier <<
350 "{\\n\\
351 \\class\\Nested_:\\public\\NestedBase_\\{\\n\\
352 \\public:\\_\\n\\
353 \\virtual\\_std::shared_ptr<NestedBase>\\clone()\\_const_\\_return_\\std::
354     make_shared<Nested>(*this);_\\_\\n\\
355 \\Nested(int_\\pragma_id)__:\\NestedBase(pragma_id)__{_}\\n\\
356 \\void\\_callme(ForParameter_\\for_param){_}\\n\\
357 _};\\n\\
358 \\ThreadPool::getInstance(\\\" << utils::FileName(omp_stmt->getLocStart(), sm
359 )
360 << "\\")->call(std::make_shared<Nested>(\\\" << stmt_start_line << \");\\n\\
361 }";

```

```

353 clang::SourceLocation pragma_start_src_loc = sm.translateLineCol(sm.
354   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 void TransformRecursiveASTVisitor::RewriteOMPPPragma(clang::Stmt *
   associated_stmt, std::string pragma_name) {
362
363   clang::Stmt *s = static_cast<clang::CapturedStmt *>(associated_stmt)->
   getCapturedStmt();
364
365   clang::SourceLocation stmt_start_src_loc = s->getLocStart();
366   unsigned pragma_start_line = utils::Line(stmt_start_src_loc, sm);
367
368   Node *n = GetNodeObjForPragma(s);
369
370   std::stringstream text;
371   std::stringstream text_constructor_params;
372   std::stringstream text_class_var;
373   std::stringstream text_fx_var;
374   std::stringstream text_constructor_var;
375   std::stringstream text_constructor;
376
377   /* Insert before pragma */
378   text <<
379   "{\n\
380   \_\_class\_\_Nested\_\_: \_\_public\_\_NestedBase\_\_{\n\
381   \_\_public:\_\_\n\
382   \_\_\_\_\_\_virtual\_\_std::shared_ptr<NestedBase>\_\_clone()\_\_const\_\_{\_\_return\_\_std::
   make_shared<Nested>(*this);\_\_}\_\_\n\
383   \_\_\_\_\_\_Nested(int\_\_pragma_id";
384
385   text_constructor << "\_\_: \_\_NestedBase(pragma_id)";
386
387   clang::CapturedStmt *captured_stmt = static_cast<clang::CapturedStmt *>(
   associated_stmt);
388   /* Iterate over all the variable used inside a pragma but defined outside.
   These variable have to be passed to
389   the newly created function */
390   for(clang::CapturedStmt::capture_iterator capture_var_itr = captured_stmt
   ->capture_begin();
391       capture_var_itr != captured_stmt->capture_end();
392       ++capture_var_itr){
393
394     clang::VarDecl *var_decl = capture_var_itr->getCapturedVar();
395     std::string var_type = var_decl->getType().getAsString();
396
397     if(capture_var_itr != captured_stmt->capture_begin()){
398       text_fx_var << ",\_\_";
399       text_constructor_var << ",\_\_";
400       text_constructor_params << ",\_\_";
401     }else
402

```

```

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_undersand = var_type.find("&");
412     if(pos_undersand != std::string::npos)
413         var_type.erase(pos_undersand - 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 unsigned stmt_end_line = utils::Line(s->getLocEnd(), sm);
452 if(n->for_node_ != NULL) {
453
454     std::string text_for;
455     text_for = RewriteOMPFor(n);
456
457     text << "\n" << text_for;
458     clang::SourceLocation for_src_loc = sm.translateLineCol(sm.getMainFileID
459     (), 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.
464     getMainFileID(), stmt_end_line + 1, 1);
465     rewrite_pragma_.InsertText(for_end_src_loc, "launch_todo_job();\n",
466     true, false);
467 }else {
468     rewrite_pragma_.InsertText(stmt_start_src_loc, text.str(), true, true);
469     //clang::SourceLocation stmt_end_src_loc = sm.translateLineCol(sm.
470     getMainFileID(), stmt_end_line - 1, 1);
471
472     rewrite_pragma_.InsertText(s->getLocEnd(), "launch_todo_job();\n", true
473     , false);
474 }
475
476 /* Comment the pragma */
477 clang::SourceLocation pragma_src_loc = sm.translateLineCol(sm.
478 getMainFileID(), stmt_start_line - 1, 1);
479 rewrite_pragma_.InsertText(pragma_src_loc, "//", true, false);
480
481 /*
482 * ----- Insert after pragma -----
483 */
484
485 std::stringstream text_after_pragma;
486 text_after_pragma << "\n
487 void callme(ForParameter for_param) {\n";
488
489 if(text_fx_var.str().compare("") != 0)
490     text_after_pragma << "    fx(for_param);\n";
491 else
492     text_after_pragma << "    fx(for_param, \"<<text_fx_var.str()<<\" );\n";
493
494 text_after_pragma <<
495 "}\n\
496 };\n\
497 std::shared_ptr<NestedBase> nested_b = std::make_shared<Nested>(\"<<n->
498     getStartLine();
499 if(text_constructor_var.str().compare("") != 0)
500     text_after_pragma << "    , \"
501     text_after_pragma << text_constructor_var.str()<<\" );\n";
502 text_after_pragma <<
503 "if(ThreadPool::getInstance(\"\" << utils::FileName(s->getLocStart(), sm) <<
504     \"\")->call(nested_b))\n";
505
506 std::cout << "classname_\" << pragma_name << std::endl;
507
508 if(pragma_name.compare("OMPParallelDirective") == 0 || pragma_name.compare

```

```

    ("OMPForDirective") == 0) {
502
503 text_after_pragma << "  nested_b->callme(ForParameter(0,1));\n";
504 }else {
505 text_after_pragma << "  todo_job_.push(nested_b); \n";
506 }
507 text_after_pragma << "}\n";
508
509 /* If ForDirective no need to add the if, cause everything is solved inside
   */
510 stmt_end_line = utils::Line(s->getLocEnd(), sm);
511 clang::SourceLocation pragma_end_src_loc = sm.translateLineCol(sm.
    getMainFileID(), stmt_end_line + 1, 1);
512
513 rewrite_pragma_.InsertText(pragma_end_src_loc, text_after_pragma.str(),
    true, false);
514
515 }
516
517 Node *TransformRecursiveASTVisitor::GetNodeObjForPragma(clang::Stmt *s){
518
519 clang::SourceLocation stmt_start_src_loc = s->getLocStart();
520 unsigned stmt_start_line = utils::Line(stmt_start_src_loc, sm);
521
522 std::vector<Root *>::iterator root_itr;
523 for(root_itr = root_vect_->begin(); root_itr != root_vect_->end();
    root_itr++) {
524     if((*root_itr)->getFunctionLineStart() < utils::Line(stmt_start_src_loc,
        sm)
525         && (*root_itr)->getFunctionLineEnd() > utils::Line(
            stmt_start_src_loc, sm))
526
527         break;
528 }
529
530 std::vector<Node *>::iterator node_itr;
531 Node * n;
532 for(node_itr = (*root_itr)->children_vect_->begin();
    node_itr != (*root_itr)->children_vect_->end();
    node_itr++) {
533
534     n = RecursiveGetNodeObjforPragma(*node_itr, stmt_start_line);
535     if(n != NULL)
536         return n;
537 }
538 return NULL;
539 }
540
541 }
542
543 Node *TransformRecursiveASTVisitor::RecursiveGetNodeObjforPragma(Node *n,
    unsigned stmt_start_line) {
544 Node *nn;
545 if(n->getStartLine() == stmt_start_line){
546     return n;
547 }else if(n->children_vect_ != NULL) {
548     for(std::vector<Node *>::iterator node_itr = n->children_vect_->begin();
        node_itr != n->children_vect_->end(); ++ node_itr) {
549
550
551         nn = RecursiveGetNodeObjforPragma(*node_itr, stmt_start_line);
552         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->
568         loop_var_ << "=_" ;
569     if(for_node->loop_var_init_val_set_)
570         text_for << for_node->loop_var_init_val_;
571     else
572         text_for << for_node->loop_var_init_var_;
573
574     text_for << "_+_"for_param.thread_id_*(" ;
575     if(for_node->condition_val_set_)
576         text_for << for_node->condition_val_ << "_-_" ;
577     else
578         text_for << for_node->condition_var_ << "_-_" ;
579
580     if(for_node->loop_var_init_val_set_)
581         text_for << for_node->loop_var_init_val_;
582     else
583         text_for << for_node->loop_var_init_var_;
584
585     text_for << ")/for_param.num_threads_;_" ;
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 << ")\u{000a}\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()
32     )->getCapturedStmt();
33
34 file_name_ = utils::FileName(pragma_stmt_ ->getLocStart(), sm);
35 if(s != NULL) {
36     start_line_ = utils::Line(s->getLocStart(), sm);
37     start_column_ = utils::Column(s->getLocStart(), sm);
38
39     end_line_ = utils::Line(s->getLocEnd(), sm);
40     end_column_ = utils::Column(s->getLocEnd(), sm);
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::
52     SourceManager& sm) {
53
54     parent_func_info_.function_decl_ = funct_decl;
55     parent_func_info_.function_start_line_ = utils::Line(funct_decl->
56     getLocStart(), sm);
57     parent_func_info_.function_end_line_ = utils::Line(funct_decl->getLocEnd
58     (), sm);
59
60     /* Name of the function containing the pragma */
61     parent_func_info_.function_name_ = funct_decl->getNameInfo().getAsStrin
62     g();
63
64     /* Return type of the function containing the pragma */
65     parent_func_info_.function_return_type_ = funct_decl->getResultType().
66     getAsString();
67
68     /* Parameters of the function containing the pragma */
69     parent_func_info_.num_params_ = funct_decl->getNumParams();
70     parent_func_info_.function_parameters_ = new std::string*[
71     parent_func_info_.num_params_];
72
73     for(int i = 0; i < parent_func_info_.num_params_; i++) {
74         parent_func_info_.function_parameters_[i] = new std::string[2];
75
76         const clang::ValueDecl *value_decl = static_cast<const clang::ValueDecl
77         *>(funct_decl->getParamDecl(i));
78         parent_func_info_.function_parameters_[i][0] = value_decl->getType().
79         getAsString();
80
81         const clang::NamedDecl *named_decl = static_cast<const clang::NamedDecl
82         *>(funct_decl->getParamDecl(i));
83         parent_func_info_.function_parameters_[i][1] = named_decl->
84         getNameAsString();
85     }
86
87     /* If the parent function is declared in a class return the name of the

```

```

class */
78 /* if (clang::CXXMethodDecl *cxxMethodD = dynamic_cast<clang::CXXMethodDecl
    *>(funct_decl)){
79     const clang::NamedDecl *nD = static_cast<const clang::NamedDecl *>(
        cxxMethodD->getParent());
80     parent_funct_info_.parentFunctionClassName = nD->
        getQualifiedNameAsString();
81 }
82 */
83     parent_funct_info_.function_class_name_ = "";
84 }
85
86 void Node::setPragmaClauses(clang::SourceManager& sm) {
87
88     pragma_type_ = pragma_stmt_->getStmtClassName();
89     /*
90     * ---- Extract pragma options ----
91     */
92     clang::OMPClause *omp_clause = NULL;
93     const char * clause_name;
94     unsigned num_clauses = pragma_stmt_->getNumClauses();
95
96     for(unsigned i = 0; i < num_clauses; i ++) {
97         omp_clause = pragma_stmt_->getClause(i);
98         clause_name = getOpenMPClauseName(omp_clause->getClauseKind());
99         VarList_ *var_list = new VarList_;
100
101         if(strcmp(clause_name, "shared") == 0 || strcmp(clause_name, "private")
            == 0 || strcmp(clause_name, "firstprivate") == 0) {
102
103             for(clang::StmtRange stmt_range = omp_clause->children(); stmt_range;
                ++ stmt_range) {
104                 const clang::DeclRefExpr *decl_ref_expr = static_cast<const clang::
                    DeclRefExpr *>(*stmt_range);
105                 if(decl_ref_expr) {
106                     const clang::NamedDecl *named_decl = decl_ref_expr->getFoundDecl()
                    ;
107                     const clang::ValueDecl *value_decl = decl_ref_expr->getDecl();
108                     var_list->insert(std::pair<std::string, std::string>(named_decl->
                        getNameAsString(), value_decl->getType().getAsString()));
109                 }
110             }
111         }else if(strcmp(clause_name, "period") == 0) {
112
113             clang::OMPPeriodClause *omp_peroid_clause = static_cast<clang::
                OMPPeriodClause *>(omp_clause);
114             const clang::IntegerLiteral *int_literal = static_cast<const clang::
                IntegerLiteral *>(omp_peroid_clause->getPeriodValue());
115             char period_val[100];
116             sprintf(period_val, "%lu", int_literal->getValue().getZExtValue());
117             var_list->insert(std::pair<std::string, std::string>(period_val, ""));
118         }else {
119             var_list->insert(std::pair<std::string, std::string>("", ""));
120         }
121
122         option_vect_->insert(std::pair<std::string, VarList_>(clause_name, *
            var_list));
123     }
124 }

```

```

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
    ");

```

```

181     nesting_element->InsertEndChild(new_pragmas_element);
182     for(std::vector<Node *>::iterator node_itr = children_vect_->begin();
node_itr != children_vect_->end(); ++node_itr) {
183         (*node_itr)->CreateXMLPragmaNode(xml_doc, new_pragmas_element);
184     }
185 }
186 }
187
188 void Node::CreateXMLPragmaOptions(tinyxml2::XMLDocument *xml_doc, tinyxml2::
XMLElement *options_element) {
189     if(option_vect_->size() != 0) {
190
191         for(std::map<std::string, VarList_>::iterator options_itr = option_vect_
->begin(); options_itr != option_vect_->end(); ++ options_itr) {
192
193             tinyxml2::XMLElement *option_element = xml_doc->NewElement("Option");
194             options_element->InsertEndChild(option_element);
195
196             tinyxml2::XMLElement *option_name_element = xml_doc->NewElement("Name"
);
197             option_element->InsertEndChild(option_name_element);
198             tinyxml2::XMLText* name_opt_text = xml_doc->NewText((*options_itr).
first.c_str());
199             option_name_element->InsertEndChild(name_opt_text);
200
201             if((*options_itr).second.size() != 0) {
202                 for(std::map<std::string, std::string>::iterator var_itr = (*
options_itr).second.begin(); var_itr != (*options_itr).second.end(); ++
var_itr) {
203                     tinyxml2::XMLElement *parameter_element = xml_doc->NewElement("
Parameter");
204                     option_element->InsertEndChild(parameter_element);
205
206                     if(strcmp((*var_itr).first.c_str(), "") != 0) {
207                         tinyxml2::XMLElement *type_element = xml_doc->NewElement("Type"
);
208
209                         tinyxml2::XMLText* type_text = xml_doc->NewText((*var_itr).
second.c_str());
210                         type_element->InsertEndChild(type_text);
211                         parameter_element->InsertEndChild(type_element);
212                     }
213                     tinyxml2::XMLElement *name_element = xml_doc->NewElement("Var");
214                     tinyxml2::XMLText* name_text = xml_doc->NewText((*var_itr).first.
c_str());
215                     name_element->InsertEndChild(name_text);
216                     parameter_element->InsertEndChild(name_element);
217                 }
218             }
219         }
220     }
221 }

```

Code 1.11: pragma_handler/ForNode.cpp

```

1 #include "pragma_handler/ForNode.h"
2
3
4 ForNode::ForNode(clang::ForStmt *for_stmt) {
5     loop_var_type_ = "";

```

```

6   loop_var_init_val_set_ = false;
7
8   loop_var_init_var_ = "";
9
10  condition_val_set_ = false;
11  condition_var_ = "";
12
13  increment_val_set_ = false;
14  increment_var_ = "";
15
16  for_stmt_ = for_stmt;
17  ExtractForParameters(for_stmt);
18 }
19
20 void ForNode::ExtractForParameters(clang::ForStmt *for_stmt) {
21
22     ExtractForInitialization(for_stmt);
23     ExtractForCondition(for_stmt);
24     ExtractForIncrement(for_stmt);
25
26 }
27
28 void ForNode::ExtractForInitialization(clang::ForStmt *for_stmt) {
29     /*
30     * Initialization of the loop variable
31     */
32
33     /* for(int i = ...) */
34     if(strcmp(for_stmt->child_begin()->getStmtClassName(), "DeclStmt") == 0) {
35         const clang::DeclStmt *decl_stmt = static_cast<const clang::DeclStmt
36         *>(*(for_stmt->child_begin()));
37         const clang::Decl *decl = decl_stmt->getSingleDecl();
38
39         /* Return the name of the variable */
40         const clang::NamedDecl *named_decl = static_cast<const clang::NamedDecl
41         *>(decl);
42         loop_var_ = named_decl->getNameAsString();
43
44         /* Return the type of the variable */
45         const clang::ValueDecl *vale_decl = static_cast<const clang::ValueDecl
46         *>(named_decl);
47         loop_var_type_ = vale_decl->getType().getAsString();
48
49         /* for (... = 0) */
50         if(strcmp(decl_stmt->child_begin()->getStmtClassName(), "IntegerLiteral"
51         ) == 0) {
52             const clang::IntegerLiteral *int_literal = static_cast<const clang::
53             IntegerLiteral *>(*(decl_stmt->child_begin()));
54             loop_var_init_val_ = int_literal->getValue().getZExtValue();
55             loop_var_init_val_set_ = true;
56
57             /* for (... = a) */
58             }else if (strcmp(decl_stmt->child_begin()->getStmtClassName(), "
59             ImplicitCastExpr") == 0) {
60                 const clang::DeclRefExpr *decl_ref_expr = static_cast<const clang::
61                 DeclRefExpr *>(*(decl_stmt->child_begin()->child_begin()));
62                 const clang::NamedDecl *named_decl = decl_ref_expr->getFoundDecl();
63                 loop_var_init_var_ = named_decl->getNameAsString();
64             }
65         }

```

```

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 extract 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::
120         UnaryOperator *>(increment_expr);
121         increment_op_ = unary_op->getOpcodeStr(unary_op->getOpcode());
122     } else if(strcmp(increment_expr->getStmtClassName(), "
123     CompoundAssignOperator") == 0) {
124         const clang::CompoundAssignOperator *compound_op = static_cast<const
125         clang::CompoundAssignOperator *>(increment_expr);
126         increment_op_ = compound_op->getOpcodeStr();
127         const clang::Expr *right_expr = compound_op->getRHS();
128
129         if(strcmp(right_expr->getStmtClassName(), "IntegerLiteral") == 0) {
130             const clang::IntegerLiteral *int_literal = static_cast<const clang::
131             IntegerLiteral *>(right_expr);
132             increment_val_ = int_literal->getValue().getZExtValue();
133             increment_val_set_ = true;
134
135         } else if(strcmp(right_expr->getStmtClassName(), "ImplicitCastExpr") ==
136         0) {
137             const clang::DeclRefExpr *decl_ref_expr = static_cast<const clang::
138             DeclRefExpr *>(*(right_expr->child_begin()));
139             const clang::NamedDecl *named_decl = decl_ref_expr->getFoundDecl();
140             increment_var_ = named_decl->getNameAsString();
141         }
142     }
143 }
144
145 void ForNode::CreateXMLPragmaFor(tinyxml2::XMLDocument *xml_doc, tinyxml2::
146     XMLElement *for_element) {
147
148     /*
149     * ----- DECLARATION -----
150     */
151     tinyxml2::XMLElement *declaration_element = xml_doc->NewElement("
152     Declaration");
153     for_element->InsertEndChild(declaration_element);
154
155     tinyxml2::XMLElement *type_element = xml_doc->NewElement("Type");
156     declaration_element->InsertEndChild(type_element);
157     tinyxml2::XMLText* type_text = xml_doc->NewText(loop_var_type_.c_str());
158     type_element->InsertEndChild(type_text);
159
160     tinyxml2::XMLElement *loop_var_element = xml_doc->NewElement("LoopVariable
161     ");

```



```

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 );
162     declaration_element->InsertEndChild(init_val_element);
163     char loop_var_init_val[100];
164     sprintf(loop_var_init_val, "%d", loop_var_init_val_);
165     tinyxml2::XMLText* init_val_text = xml_doc->NewText(loop_var_init_val);
166     init_val_element->InsertEndChild(init_val_text);
167 }else {
168     tinyxml2::XMLElement *init_var_element = xml_doc->NewElement("
169 InitVariable");
170     declaration_element->InsertEndChild(init_var_element);
171     tinyxml2::XMLText* init_var_text = xml_doc->NewText(loop_var_init_var_.
172 c_str());
173     init_var_element->InsertEndChild(init_var_text);
174 }
175
176 /*
177 * ---- CONDITION ----
178 */
179 tinyxml2::XMLElement *condition_element = xml_doc->NewElement("Condition"
180 );
181 for_element->InsertAfterChild(declaration_element, condition_element);
182
183 tinyxml2::XMLElement *condition_op_element = xml_doc->NewElement("Op");
184 condition_element->InsertEndChild(condition_op_element);
185 tinyxml2::XMLText* condition_op_text = xml_doc->NewText(condition_op_.
186 c_str());
187 condition_op_element->InsertEndChild(condition_op_text);
188
189 if(condition_val_set_ == true) {
190     tinyxml2::XMLElement *condition_val_element = xml_doc->NewElement("
191 ConditionValue");
192     condition_element->InsertEndChild(condition_val_element);
193     char condition_val[100];
194     sprintf(condition_val, "%d", condition_val_);
195     tinyxml2::XMLText* condition_val_text = xml_doc->NewText(condition_val);
196     condition_val_element->InsertEndChild(condition_val_text);
197 }else {
198     tinyxml2::XMLElement *condition_var_element = xml_doc->NewElement("
199 ConditionVariable");
200     condition_element->InsertEndChild(condition_var_element);
201     tinyxml2::XMLText* condition_var_text = xml_doc->NewText(condition_var_.
202 c_str());
203     condition_var_element->InsertEndChild(condition_var_text);
204 }
205
206 /*
207 * ---- INCREMENT ----
208 */
209 tinyxml2::XMLElement *increment_element = xml_doc->NewElement("Increment"
210 );
211 for_element->InsertAfterChild(condition_element, increment_element);

```

```

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;

```

```

7 clang::FunctionDecl *function_decl_tmp = NULL;
8 std::vector<Root *> *root_vect = new std::vector<Root *>();
9
10 std::vector<clang::OMPExecutableDirective *>::iterator omp_itr;
11
12 for(omp_itr = pragma_list->begin(); omp_itr != pragma_list->end(); ++
    omp_itr) {
13
14     function_decl_tmp = GetFunctionForPragma(*omp_itr, function_list, sm);
15     Node * n = new Node(*omp_itr, function_decl_tmp, sm);
16
17     /* In case of parallel for skip one stmt.
18        Parallel for is represented with two OMPExecutableDirective,
19        (OMPParallel + OMPFor) so we have to skip one stmt */
20     if((*omp_itr)->getAssociatedStmt()) {
21         if(strcmp((*omp_itr)->getStmtClassName(), "OMPParallelDirective") == 0
22             && utils::Line((*omp_itr)->getAssociatedStmt()->getLocStart(), sm)
23                 == utils::Line((*omp_itr)->getAssociatedStmt()->getLocEnd(), sm
24         )) {
25             n->pragma_type_ = "OMPParallelForDirective";
26             omp_itr++;
27         }
28     }
29
30     if(function_decl_tmp != function_decl) {
31         function_decl = function_decl_tmp;
32         Root *root = new Root(n, n->getParentFunctionInfo());
33         n->setParentNode(NULL);
34         root->setLastNode(n);
35         root_vect->push_back(root);
36     }else {
37         BuildTree(root_vect->back(), n);
38         root_vect->back()->setLastNode(n);
39     }
40 }
41 return root_vect;
42 }
43
44
45 clang::FunctionDecl *GetFunctionForPragma(clang::OMPExecutableDirective *
    pragma_stmt,
46
47         std::vector<clang::FunctionDecl *> *function_list,
48         clang::SourceManager &sm) {
49
50     unsigned funct_start_line, funct_end_line;
51     unsigned pragma_start_line = utils::Line(pragma_stmt->getLocStart(), sm);
52     std::vector<clang::FunctionDecl *>::iterator funct_itr;
53
54     for(funct_itr = function_list->begin(); funct_itr != function_list->end();
55         ++ funct_itr) {
56         funct_start_line = utils::Line((*funct_itr)->getSourceRange().getBegin()
57             , sm);
58         funct_end_line = utils::Line((*funct_itr)->getSourceRange().getEnd(), sm
59             );
60         if(pragma_start_line < funct_end_line && pragma_start_line >
61             funct_start_line)
62             return (*funct_itr);
63     }

```

```

59     return NULL;
60 }
61
62
63 /*
64 * ---- Attach the node to the correct parent (if the node is node annidated
        attach it to root) ----
65 * THEOREM: A node can be annidated only in its previous node or in the
        father of the previous node or in the father
66 *           of the father ..... of the previous node. (This is due to the
        fact that the list of pragmas is ordered based
67             on starting line of the associated stmt).
68 */
69 void BuildTree(Root *root, Node *n) {
70
71     Node *last_node = root->getLastNode();
72     bool annidation;
73
74     while(last_node != NULL) {
75         annidation = CheckAnnidation(last_node, n);
76
77         if(annidation == true) {
78             last_node->AddChildNode(n);
79             n->setParentNode(last_node);
80             return;
81
82         }else
83             last_node = last_node->getParentNode();
84     }
85
86     root->AddChildNode(n);
87     n->setParentNode(NULL);
88 }
89
90 /*
91 * ---- Check if n is annidated inside parent: to be annidated it is enough
        that n->endLine < parent->endLine
92 * (for sure n->startLine < parent->startLine because pragmas are ordered
        based on their starting line)
93 */
94 bool CheckAnnidation(Node *parent, Node *n) {
95
96     if(n->getEndLine() < parent->getEndLine())
97         return true;
98     else
99         return false;
100 }
101 }

```

Code 1.14: utils/source_locations.cpp

```

1 #include "utils/source_locations.h"
2
3 using namespace std;
4 using namespace clang;
5
6 namespace utils {
7
8 string FileName(SourceLocation const& l, SourceManager const& sm) {
9     PresumedLoc pl = sm.getPresumedLoc(l);

```

```

10     return string(pl.getFilename());
11 }
12
13 string FileId(SourceLocation const& l, SourceManager const& sm) {
14     string fn = FileName(l, sm);
15     for(size_t i=0; i<fn.length(); ++i)
16         switch(fn[i]) {
17             case '/':
18             case '\\':
19             case '>':
20             case '.':
21                 fn[i] = '_';
22         }
23     return fn;
24 }
25
26 unsigned Line(SourceLocation const& l, SourceManager const& sm) {
27     PresumedLoc pl = sm.getPresumedLoc(l);
28     return pl.getLine();
29 }
30
31 std::pair<unsigned, unsigned> Line(clang::SourceRange const& r,
32     SourceManager const& sm) {
33     return std::make_pair(Line(r.getBegin(), sm), Line(r.getEnd(), sm));
34 }
35
36 unsigned Column(SourceLocation const& l, SourceManager const& sm) {
37     PresumedLoc pl = sm.getPresumedLoc(l);
38     return pl.getColumn();
39 }
40
41 std::pair<unsigned, unsigned> Column(clang::SourceRange const& r,
42     SourceManager const& sm) {
43     return std::make_pair(Column(r.getBegin(), sm), Column(r.getEnd(), sm));
44 }
45
46 std::string location(clang::SourceLocation const& l, clang::SourceManager
47     const& sm) {
48     std::string str;
49     llvm::raw_string_ostream ss(str);
50     l.print(ss, sm);
51     return ss.str();
52 }

```

Code 1.15: xml_creator/xml_creator.cpp

```

1 #include "xml_creator/XMLcreator.h"
2
3 void CreateXML(std::vector<Root *> *root_vect, char *file_name) {
4
5     tinyxml2::XMLDocument *xml_doc = new tinyxml2::XMLDocument();
6     tinyxml2::XMLElement *file_element = xml_doc->NewElement("File");
7     xml_doc->InsertEndChild(file_element);
8
9     tinyxml2::XMLElement *name_element = xml_doc->NewElement("Name");
10    tinyxml2::XMLText* name_text = xml_doc->NewText(file_name);
11    name_element->InsertEndChild(name_text);
12    file_element->InsertEndChild(name_element);
13

```

```

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_
12         (false) {}
13     /* Costructor for parallel for */
14     ProfileTrackParams(int funct_id, int pragma_line, int n)
15         : funct_id_(funct_id), pragma_line_(pragma_line), num_for_iteration_(n),
16         num_for_iteration_set_(true) {}
17
18     int funct_id_;
19     int pragma_line_;
20     /* In the case of a parallel for this variable saves the number of the
21        iteration of the for */
22     int num_for_iteration_;
23     bool num_for_iteration_set_;
24 };
25
26 /*
27 * ----- Class that keep track of the children time and the father of the
28 * current pragma in execution -----
29 */
30 class ProfileTracker {
31
32     clock_t start_time_;
33     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/
47        pragma */
48     ProfileTracker *previous_pragma_executed_;
49
50     /* In the constructor a timer is started */
51     ProfileTracker(const ProfileTrackParams & p);
52     /* In the destructor the timer is stopped and the elapsed time is written
53        in the log file */
54     ~ProfileTracker();
55
56     /* This is necessary to allow to create an object inside the declaration
57        of an if stmt */
58     operator bool() const { return true; }
59 };
60
61 /*
62  * ---- Singleton class that open and close the log file ----
63  */
64 class ProfileTrackerLog {
65
66     /* Keeps track of which function/pragma has invoked the current function/
67        pragma */
68     ProfileTracker *current_pragma_executing_;
69
70     /* Create the log file and write in it the hardware spec */
71     ProfileTrackerLog ();
72
73     void WriteArchitecturesSpec();
74     size_t getTotalSystemMemory();
75
76 public:
77     /* File where the log is written */
78     std::ofstream log_file_;
79
80     static ProfileTrackerLog* getInstance();
81     /* Substitute the pointer of the current pragma in execution and return
82        the previous value */
83     ProfileTracker *ReplaceCurrentPragma(ProfileTracker *
84        current_pragma_executing_);
85
86     /* Save and close the log file */
87     ~ProfileTrackerLog();
88

```


Code 1.17: profile_tracker.cpp

```

1  #include "profile_tracker/profile_tracker.h"
2
3
4  /*
5   * ---- PROFILE TRACKER LOG ----
6   */
7  ProfileTrackerLog::ProfileTrackerLog () {
8      current_pragma_executing_ = NULL;
9      log_file_.open(log_file);
10     log_file_ << "<LogFile>" << std::endl;
11     WriteArchitecturesSpec();
12 }
13
14 void ProfileTrackerLog::WriteArchitecturesSpec() {
15     log_file_ << "␣␣<Hardware␣";
16     log_file_ << "NumberOfCores=\" " << std::thread::hardware_concurrency() <<
17         "\"␣";
18     log_file_ << "MemorySize=\" " << getTotalSystemMemory() << "\"/>" << std::endl;
19 }
20
21 size_t ProfileTrackerLog::getTotalSystemMemory() {
22     /*long pages = sysconf(_SC_PHYS_PAGES);
23     long page_size = sysconf(_SC_PAGE_SIZE);
24     return (pages * page_size)/1024/1024;*/
25     return 2000;
26 }
27
28 ProfileTrackerLog* ProfileTrackerLog::getInstance() {
29     static ProfileTrackerLog log;
30     return &log;
31 }
32
33 ProfileTrackerLog::~ProfileTrackerLog() {
34     log_file_ << "</LogFile>" << std::endl;
35     log_file_.close();
36 }
37
38 ProfileTracker *ProfileTrackerLog::ReplaceCurrentPragma(ProfileTracker *
39     current_pragma_executing) {
40     ProfileTracker *tmp = current_pragma_executing_;
41     current_pragma_executing_ = current_pragma_executing;
42     return tmp;
43 }
44
45 /*
46 * ---- PROFILE TRACKER ----
47 */
48 ProfileTracker::ProfileTracker(const ProfileTrackParams & p) {
49     previous_pragma_executed_ = ProfileTrackerLog::getInstance()->
50         ReplaceCurrentPragma(this);
51
52     children_elapsed_time_ = 0;
53
54     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_ << "\"\n" \
104                                     << "childrenTime=\"" << children_elapsed_time_ << "
    \"/>" << std::endl;
105
106
107 }

```

1.3.2 Final execution

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         /* Indicates 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
nified to have completed */
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.
26     cpp -> test_schedule.xml*/
27     std::string in_xml_file (file_name);
28     size_t ext = in_xml_file.find_last_of(".");
29     if (ext == std::string::npos)
30         ext = in_xml_file.length();
31     in_xml_file = in_xml_file.substr(0, ext);
32     in_xml_file.insert(ext, "_schedule.xml");
33
34     tinyxml2::XMLDocument xml_doc;
35     //xml_doc.LoadFile(in_xml_file.c_str());
36     xml_doc.LoadFile("schedule.xml");
37
38     tinyxml2::XMLElement *threads_num_element = xml_doc.FirstChildElement("
39     Schedule")->FirstChildElement("Cores");
40
41     const char* threads_num = threads_num_element->GetText();
42     /* Set the number of thread as the number of cores plus one thread wich
43     is used to run parallel and sections job */
44     init(chartoint(threads_num));
45
46
47     tinyxml2::XMLElement *pragma_element = xml_doc.FirstChildElement("
48     Schedule")->FirstChildElement("Pragma");
49     while(pragma_element != NULL) {
50         ScheduleOptions sched_opt;
51
52         const char* pragma_id = pragma_element->FirstChildElement("id")->
53         GetText();
54         int id = chartoint(pragma_id);
55         sched_opt.pragma_id_ = id;
56
57
58         tinyxml2::XMLElement *pragma_type_element = pragma_element->
59         FirstChildElement("Type");
60         const char* pragma_type = pragma_type_element->GetText();
61         sched_opt.pragma_type_ = pragma_type;
62
63
64         tinyxml2::XMLElement *thread_element = pragma_element->
65         FirstChildElement("Threads");
66         if(thread_element != NULL)
67             thread_element = thread_element->FirstChildElement("Thread");
68
69         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,

```

```

217         ForParameter for_param) {
218
219     Jobid_t id = nested_base->pragma_id_;
220
221     JobIn job_in(nested_base, for_param);
222     job_in.job_id_ = id;
223     job_in.job_completed_ = true;
224
225     job_pop_mtx.lock();
226     if(known_jobs_[id].size() == 0)
227         known_jobs_[id].reserve(for_param.num_threads_);
228     known_jobs_[id].push_back(std::move(job_in));
229     job_pop_mtx.unlock();
230 }
231
232
233 void ThreadPool::push(std::shared_ptr<NestedBase> nested_base,
234                      ForParameter for_param, int thread_id)
235 {
236     Jobid_t id = nested_base->pragma_id_;
237
238     JobIn job_in(nested_base, for_param);
239     job_in.job_id_ = id;
240     job_in.job_type_ = sched_opt_[nested_base->pragma_id_].pragma_type_;
241     job_in.done_cond_var_ =
242         std::unique_ptr<std::condition_variable>(new std::
condition_variable());
243
244     job_pop_mtx.lock();
245     if(known_jobs_[id].size() == 0)
246         known_jobs_[id].reserve(for_param.num_threads_);
247     known_jobs_[id].push_back(std::move(job_in));
248
249     JobQueue j_q(id, for_param.thread_id_);
250     work_queue_[thread_id].push(j_q);
251
252     job_pop_mtx.unlock();
253 }
254
255
256 void ThreadPool::push_termination_job(int thread_id) {
257
258     JobQueue j_q(-1, 0);
259     work_queue_[thread_id].push(j_q);
260 }
261
262
263 void ThreadPool::run(int me) {
264     thread_id_to_int_[std::this_thread::get_id()] = me;
265     while(true) {
266
267         job_pop_mtx.lock();
268         if(work_queue_[me].size() != 0) {
269
270             JobQueue j_q = work_queue_[me].front();
271             work_queue_[me].pop();
272             job_pop_mtx.unlock();
273

```

```

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 void ThreadPool::join(Jobid_t job_id) {
324
325     /*for(int i = 0; i < known_jobs_[job_id].size(); i ++) {
326         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].
336 end(); ++ j_itr) {
337         job_end.lock();
338         if((*j_itr).job_completed_ != true){
339             job_end.unlock();
340             std::unique_lock<std::mutex> lk(cond_var_mtx[std::this_thread::
341 get_id()]);
342             j_itr->done_cond_var_>wait(lk);
343         }else{
344             job_end.unlock();
345         }
346     }
347     job_pop_mtx.lock();
348     known_jobs_.erase(job_id);
349     job_pop_mtx.unlock();
350 }
351
352 void ThreadPool::joinall() {
353     /* Push termination job in the working queue */
354     std::cout << "Joinall" << std::endl;
355     for (int i = 0; i < threads_pool_.size(); i++)
356         push_termination_job(i);
357
358     /* Joining on all the threads in the thread pool */
359     for(int i = 0; i < threads_pool_.size(); i++)
360         threads_pool_[i].join();
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

Code 2.1: graphCreator.py

```
1 import sys
2 import paragraph as par
3 import copy
4 import schedule as sched
5 import profiler as pro
6 import time
7 import multiprocessing
8 import itertools
9 import random
10 import threading
11
12 """_Usage:_call_with_<filename>_<pragma_xml_file>_<executable_name>_<
    profiling_interations>_True/False_(for_output)_"""
13
14 if __name__ == "__main__":
15
16     pragma_xml = sys.argv[1]
17     executable = sys.argv[2]
18     count = int(sys.argv[3])
19     output = sys.argv[4]
20     execution_time = float(sys.argv[5])
21     deadline = float(sys.argv[6])
22     multi = sys.argv[7]
23
24     #runs count time the executable and aggregates the informations in
25     #executable_profile.xml. The single profile outputs are saved as profile+
26     #iter.xml
27     profile_xml = pro.profileCreator(count, executable)
28
29     #return the nested dot graphs in code style (one for each function)
30     visual_nested_graphs = par.getNesGraph(pragma_xml, profile_xml)
31
32     #returns the graphs to be visualized and the object graphs in flow style (
33     #one for each function)
34     (visual_flow_graphs, flow_graphs) = par.getParalGraph(pragma_xml,
35     profile_xml)
36
37     i = 0
38
39     for g in visual_nested_graphs:
40         g.write_pdf('graphs/%s_code.pdf'%flow_graphs[i].type)
41         g.write_dot('graphs/%s_code.dot'%flow_graphs[i].type)
42         i += 1
```

```

39
40 i = 0
41 for g in visual_flow_graphs:
42     g.write_pdf('graphs/%s_flow.pdf'%flow_graphs[i].type)
43     g.write_dot('graphs/%s_flow.dot'%flow_graphs[i].type)
44     i += 1
45
46 #creates the flow type graph --> flow.xml
47 par.dump_graphs(flow_graphs)
48 #adding to the original xml the profiling informations --> code.xml
49 pro.add_profile_xml(profile_xml, pragma_xml)
50 #creating the total graph with the call-tree
51 func_graph = par.create_complete_graph(visual_flow_graphs, profile_xml)
52 #creating the graphs with the function calls
53 func_graph.write_pdf('graphs/function_graphs.pdf')
54 func_graph.write_dot('graphs/function_graphs.dot')
55
56 #creating the expanded graph where the functions are inserted in the flow
57   graph
58 exp_flows = copy.deepcopy(flow_graphs)
59 par.explode_graph(exp_flows)
60 main_flow = sched.get_main(exp_flows)
61
62 #creating a generator for the expanded graph
63 gen = sched.generate_task(main_flow)
64
65 #creating a new generator for the expanded graph
66 sched.make_white(main_flow)
67
68 #getting the number of physical cores of the machine profiled
69 max_flows = sched.get_core_num(profile_xml)
70 max_flows = 4
71 #getting cores of the actual machine, but the problem is multithreading
72 cores = multiprocessing.cpu_count()
73 if cores == 1:
74     cores = 2
75
76 #initializing all the lists for the parallel scheduling algorithm
77 tasks_list = []
78 task_list = []
79 flows_list = []
80 optimal_flow_list = []
81 p_list = []
82 queue_list = []
83 results = []
84 num_tasks = 0
85
86 #getting the number of tasks in the expanded graph and creating a list of
87   task
88 for task in gen:
89     task_list.append(task)
90     num_tasks += 1
91
92 if output == 'True':
93     sched.make_white(main_flow)
94     par.scanGraph(main_flow)
95
96 #starting the parallel or sequential search of the best solution with a
97   timing constrain

```

```

95 if multi == 'parallel':
96     for core in range(cores):
97         tmp = []
98         optimal_flow_list.append(tmp)
99         tmp_2 = []
100         flows_list.append(tmp_2)
101         random.shuffle(task_list)
102         tasks_list.append(copy.deepcopy(task_list))
103         q = sched.Queue()
104         queue_list.append(q)
105         p_list.append(multiprocessing.Process(target = sched.get_optimal_flow,
106         args = (flows_list[core], tasks_list[core], 0, optimal_flow_list[core],
107         num_tasks, max_flows, execution_time, queue_list[core], )))
108         print "starting core:", core
109         p_list[core].start()
110     #getting the results from the processes
111     for queue in queue_list:
112         t = queue.q.get()
113         results.append(t)
114     #joining all the processes
115     i = 0
116     for p in p_list:
117         p.join()
118         print "core", i, "joined"
119         i += 1
120     #getting the best result
121     optimal_flow = results[0]
122     best = 0
123     for i in range(len(results)):
124         print "result:"
125         for flow in results[i]:
126             flow.dump()
127             if sched.get_cost(results[i]) < sched.get_cost(optimal_flow):
128                 best = i
129     optimal_flow = results[best]
130 else:
131     optimal_flow = []
132     flow_list = []
133     execution_time += time.clock()
134     print "searching best schedule"
135     sched.get_optimal_flow_single(flow_list, task_list, 0, optimal_flow,
136     num_tasks, max_flows, execution_time )
137
138 #printing the best result
139 print "solution:"
140 for flow in optimal_flow:
141     flow.dump("\t")
142     print "\tttime:", flow.time
143
144 #substitutes "for tasks" with splitted versions if present in the optimal
145 flows
146 par.add_new_tasks(optimal_flow, main_flow)
147 sched.make_white(main_flow)
148 gen_ = sched.generate_task(main_flow)
149
150 t_list = []
151 for t in gen_:

```

```

150     t_list.append(t)
151     """
152     print t.type, " @ ", t.start_line, " has parents:"
153     for p in t.parent:
154         print "\t ", p.type, " @ ", p.start_line
155     print "and children:"
156     for c in t.children:
157         print "\t ", c.type, " @ ", c.start_line
158     print
159     """
160
161     #adds id's to all the tasks to retrieve the flow to which they belong
162     par.add_flow_id(optimal_flow, t_list)
163
164     #sets arrival times and deadlines using a modified version of the chetto
165     #algorithm
166     sched.chetto(main_flow, deadline, optimal_flow)
167
168     #checks if the schedule is feasible and in case creates the schedule file
169     if sched.check_schedule(main_flow):
170         sched.create_schedule(main_flow, len(optimal_flow))
171         sched.make_white(main_flow)
172         #sched.print_schedule(main_flow)
173     else:
174         print "tasks not schedulable, try with more search time"
175
176     #prints extended info of the entire pragma graph

```

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
7 import re
8 import math
9
10 colors = ( "beige", "bisque3", "bisque4", "blanchedalmond", "blue",
11 "blue1", "blue2", "blue3", "blue4", "blueviolet",
12 "brown", "brown1", "brown2", "brown3", "brown4",
13 "burlywood", "burlywood1", "burlywood2", "burlywood3", "burlywood4",
14 "cadetblue", "cadetblue1", "cadetblue2", "cadetblue3", "cadetblue4",
15 "chartreuse", "chartreuse1", "chartreuse2", "chartreuse3", "chartreuse4",
16 "chocolate", "chocolate1", "chocolate2", "chocolate3", "chocolate4",
17 "coral", "coral1", "coral2", "coral3", "coral4",
18 "cornflowerblue", "crimson", "cyan", "cyan1", "cyan2",
19 "cyan3", "cyan4", "darkgoldenrod", "darkgoldenrod1", "darkgoldenrod2",
20 "darkgoldenrod3", "darkgoldenrod4", "darkgreen", "darkkhaki", "
    darkolivegreen",
21 "darkolivegreen1", "darkolivegreen2", "darkolivegreen3", "darkolivegreen4",
    "darkorange",
22 "darkorange1", "darkorange2", "darkorange3", "darkorange4", "darkorchid",
    "darkorchid1", "darkorchid2", "darkorchid3", "darkorchid4", "darksalmon",
23 "darkseagreen", "darkseagreen1", "darkseagreen2", "darkseagreen3", "
    darkseagreen4",
24 "darkslateblue", "darkslategray", "darkslategray1", "darkslategray2", "

```



```

    darkslategray3",
26 "darkslategray4", "darkslategrey", "darkturquoise", "darkviolet", "
    deeppink",
27 "deeppink1", "deeppink2", "deeppink3", "deeppink4", "deepskyblue",
28 "deepskyblue1", "deepskyblue2", "deepskyblue3", "deepskyblue4", "dimgray",
29 "dimgray", "dodgerblue", "dodgerblue1", "dodgerblue2", "dodgerblue3",
30 "dodgerblue4", "firebrick", "firebrick1", "firebrick2", "firebrick3",
31 "firebrick4", "forestgreen", "gold", "gold1", "gold2",
32 "gold3", "gold4", "goldenrod", "goldenrod1", "goldenrod2", "goldenrod3"
    , "goldenrod4")
33
34 class Node(object):
35     def __init__(self, Ptype, s_line, time, variance):
36         self.type = Ptype
37         self.start_line = s_line
38         self.children = []
39         self.parent = []
40         self.options = []
41         self.time = float(time)
42         self.variance = variance
43         self.end_line = 0
44         self.callerid = []
45         self.deadline = None
46         self.arrival = None
47         self.d = None
48         self.children_time = 0
49         self.in_time = 0
50         self.color = 'white'
51         self.id = None
52     def add(self, x):
53         x.parent.append(self)
54         self.children.append(x)
55     def myself(self):
56         if self.type != 'BARRIER':
57             print "pragma_node: ", self.type, "\n\n\n\nstart_line: ", self.
start_line, "\n\n\n\nendl_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\n\n\nstart_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\n_endl_line:_", self.end_line, "\n\n_init_type:_", self.init_type, "\n\n_init_var:_", self.init_var, "\n\n_init_value:_", self.init_value, "\n\n_init_condition:_", self.init_cond, "\n\n_init_condition_value:_", self.init_cond_value, "\n\n_init_increment_type:_", self.init_increment, "\n\n_init_increment_value:_", self.init_increment_value, "\n\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\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],
140                                     pragma[5])
141
142 class Architecture():
143     def __init__(self, num_cores, tot_memory):
144         self.num_cores = num_cores
145         self.tot_memory = tot_memory
146
147 class Time_Node():
148     def __init__(self, func_line, pragma_line ):
149         self.times = []
150         self.func_line = func_line
151         self.pragma_line = pragma_line
152         self.variance = 0
153         self.loops = []
154         self.caller_list = []
155         self.children_time = []
156
157 class Flow():
158     def __init__(self):
159         self.tasks = []
160         self.bandwidth = 0
161         self.time = 0
162     def add_task(self, task):
163         self.tasks.append(task)
164         self.update(task)
165     def update(self, task):
166         self.time += task.in_time #float(task.time) - float(task.children_time)
167     def dump(self, prefix=""):
168         print prefix, "flow:"
169         for task in self.tasks:
170             print prefix, "\t", task.type, "\u", task.start_line, "\u", task.in_time
171             , "\u id\u", task.id
172     def remove_task(self, task):
173         self.tasks.remove(task)
174         self.time -= task.in_time #float(task.time) - float(task.children_time)
175
176 class Task():
177     def __init__(self, count, id):
178         self.count = count
179         self.id = []
180         self.id.append(id)
181
182 def scanGraph(node):
183     #print pre, node.type
184     if node.color != 'black':
185         node.color = 'black'
186         node.myself()
187         print "\u\u\u\u\u\u\u\uhas\u children:"

```

```

187     for c in node.children:
188         print "uuuuuuuuuu",c.type,"@",c.start_line
189     print "uuuuuuhasparent:"
190     for p in node.parent :
191         print "uuuuuuuuuu",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 * "uu"
197     if len(elem):
198         if not elem.text or not elem.text.strip():
199             elem.text = i + "uu"
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
219         , profile_graph_root.find('Hardware/MemorySize').text)
220
221     file_name = pragma_graph_root.find('Name').text
222
223     for n in pragma_graph_root.findall('Function'):
224         graphs.append(p.Dot(graph_type = 'digraph'))
225         name = n.find('Name').text
226         time = float(functions[n.find('Line').text].time)
227         callerid = functions[n.find('Line').text].callerid
228         children_time = float(functions[n.find('Line').text].children_time)
229         root = n.find('Line').text
230         if (time == 0):
231             pragma_graph_root = p.Node(n.find('Line').text, label = name + "()\n
232             nnot_executed", root = root)
233         else:
234             pragma_graph_root = p.Node(n.find('Line').text, label = name + "()\n
235             nexecution_time_%g" % time, root = root)
236             pragma_graph_root.callerid = callerid
237             graphs[count].add_node(pragma_graph_root)
238             Objroot = Fx_Node(name, n.find('Line').text,n.find('ReturnType').text,
239             float(functions[n.find('Line').text].time), functions[n.find('Line').text
240             ].variance, file_name)
241             for par in n.findall('Parameters/Parameter'):
242                 Objroot.add_arg( ( par.find('Type').text,par.find('Name').text ) )
243             Objroot.children_time = children_time
244             Objroot.in_time = Objroot.time - children_time
245             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 =
127 root)
128     else:
129         child = p.Node(key, label = visual_name + "\nexecution_time:" + str(
130 time) + "\nvariance:" + str(variance), root = root)
131     pragma_graph.add_node(node)
132     pragma_graph.add_node(child)
133     pragma_graph.add_edge(p.Edge(node, child))
134     treeNode.add(Objchild)
135     #print Objchild.type,"@",Objchild.start_line,"\nis_attached_to",treeNode
136     .type,"@",treeNode.start_line
137
138     if(d.find('Children')):
139         node_ = create_diamond(d.find('Children'), pragma_graph, child,
140 Objchild, func_pragmas, root)
141         tmp_name = (node_.start_line)
142         if tmp_name not in func_pragmas:
143             time = 0
144         else:
145             time = func_pragmas[tmp_name][0]
146             #treeNode = Node('BARRIER_end', tmp_name, 0, 0)
147             #Objchild.add(treeNode)
148             treeNode = node_
149             node = p.Node(tmp_name + "_end", label = "BARRIER", root = root)
150         else:
151             node = child
152             treeNode = Objchild
153
154 def create_diamond(tree, graph, node, treeNode, func_pragmas, root):
155     special_node = p.Node(node.get_name().replace("\n", "") + "_end", label =
156 'BARRIER', root = root)
157     Objsspecial_node = Node( 'BARRIER_end' , node.get_name() , 0, 0 )
158     color = colors[randrange(len(colors) - 1)]
159     for d in tree.find('Pragmas').findall('Pragma'):
160
161         end_line = d.find('Position/EndLine').text
162         key = d.find('Position/StartLine').text
163
164         if key not in func_pragmas:
165             time = 0
166             variance = None
167             loops = 0
168             callerid = None
169             children_time = 0
170         else:
171             time = float(func_pragmas[key][0])
172             variance = func_pragmas[key][1]
173             loops = func_pragmas[key][2]
174             callerid = func_pragmas[key][3]
175             children_time = float(func_pragmas[key][4])
176
177         tmp_name = d.find('Name').text.replace("::", "\n")
178         visual_name = tmp_name + "@%s" % key

```

```

346 if ("For" in tmp_name ):
347     loops = func_pragmas[key][2]
348     if (d.find('For/Declaration/InitValue') != None):
349         init_value = d.find('For/Declaration/InitValue').text
350     else:
351         init_value = d.find('For/Declaration/InitVariable').text
352     if (d.find('For/Condition/ConditionValue') != None):
353         init_var = d.find('For/Condition/ConditionValue').text
354     else:
355         init_var = d.find('For/Condition/ConditionVariable').text
356     if(d.find('For/Increment/IncrementValue') != None):
357         inc = d.find('For/Increment/IncrementValue').text
358     else:
359         inc = ""
360     Objchild = For_Node(tmp_name, key, d.find('For/Declaration/Type').text
361 , d.find('For/Declaration/LoopVariable').text, init_value, d.find('For/
362 Condition/Op').text, init_var, d.find('For/Increment/Op').text, inc ,
363 time, variance, loops)
364     visual_name = visual_name + "\nfor(□" + Objchild.init_var + "□=□" +
365 Objchild.init_value + ";□"+Objchild.init_var + "□" + Objchild.init_cond +
366 "□" + Objchild.init_cond_value + "□;□" + Objchild.init_var + "□" +
367 Objchild.init_increment + "□" + Objchild.init_increment_value + ")□"
368 else:
369     Objchild = Node(tmp_name, key, time, variance)
370
371 deadline = None
372 if(d.find('Options')):
373     for op in d.find('Options').findall('Option'):
374         Objchild.options.append( (op.find('Name').text,[get_parameter(i) for
375 i in op.findall('Parameter')]) )
376         if op.find('Name').text == 'deadline':
377             deadline = op.find('Parameter').text
378
379 Objchild.end_line = end_line
380 Objchild.callerid.append(callerid)
381 Objchild.deadline = deadline
382 Objchild.children_time = children_time
383 Objchild.in_time = Objchild.time - children_time
384
385 child = p.Node(key, label = visual_name + "\nexexecution□time:□" + str(
386 time) + "\nvariance:□" + str(variance), root = root)
387 graph.add_node(node)
388 graph.add_node(child)
389 graph.add_edge(p.Edge(node, child, color = color))
390 treeNode.add(Objchild)
391
392 if(d.find('Children')):
393     #get the real returned label as name
394     tmp_node = create_diamond(d.find('Children'), graph, child, Objchild,
395 func_pragmas, root)
396     g_node = p.Node(tmp_node.start_line+ "_end", label = 'BARRIER', root =
397 root)
398     graph.add_node(g_node)
399     graph.add_node(special_node)
400     graph.add_edge(p.Edge(g_node, special_node, color = color))
401     #tmp_name = tmp.get_name().replace("\n", "")
402     #ObjTmp = Node(tmp_name, tmp_name, 0, 0)
403     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 = "\nexecution_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 + "()" + "\nexecution_time:%f" %
440 time + "\nvariance:" + str(variance))
441         graphs[count].add_node(root)
442         find_nesting(n, graphs[count], root, functions[key].pragmas)
443         count += 1
444
445     return graphs
446
447 def create_complete_graph(visual_flow_graphs, profile_xml):
448     func_graph = p.Dot(graph_type = 'digraph', compound = 'true')
449     clusters = []
450
451     i = 0

```



```

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')
486             for par in func.arguments:
487                 parameter = ET.SubElement( parameters, 'Parameter')
488                 type_ = ET.SubElement( parameter, 'Type')
489                 type_.text = par[0]
490                 name_ = ET.SubElement( parameter, 'Name')
491                 name_.text = par[1]
492             line = ET.SubElement(function, 'Line')
493             line.text = func.start_line
494             time = ET.SubElement(function, 'Time')
495             time.text = str(func.time)
496             variance = ET.SubElement(function, 'Variance')
497             variance.text = str(func.variance)
498             func.xml_parent = None
499             if ( func.callerid != None ):
500                 callerids = ET.SubElement(function, 'Callerids')
501                 for id_ in func.callerid:
502                     callerid = ET.SubElement(callerids, 'Callerid')
503                     callerid.text = id_
504             if len(func.children) != 0:
505                 pragma_list = []
506                 edge_list = []
507                 pragmas = ET.SubElement(function, 'Nodes')
508                 dump_pragmas(func, pragmas, pragma_list)
509                 edges = ET.SubElement(function, 'Edges')
510                 dump_edges(func, edges, edge_list)

```

```

511 tree = ET.ElementTree(root)
512 indent(tree.getroot())
513 tree.write('flow.xml')
514
515
516 def dump_pragmas(pragma_node, pragmas_element, pragma_list):
517     for pragma in pragma_node.children:
518         if str(pragma.start_line) + str(pragma.end_line) not in pragma_list:
519             pragma_list.append(str(pragma.start_line) + str(pragma.end_line))
520             pragma_ = ET.SubElement(pragmas_element, 'Pragma')
521             pragma_.attrib['id'] = str(pragma.start_line) + str(pragma.end_line)
522             name = ET.SubElement(pragma_, 'Name')
523             if not "_end" in pragma.type:
524                 name.text = pragma.type
525             else:
526                 name.text = "BARRIER"
527             if(len(pragma.options) != 0):
528                 options = ET.SubElement(pragma_, 'Options')
529                 for op in pragma.options:
530                     option = ET.SubElement(options, 'Option')
531                     op_name = ET.SubElement(option, 'Name')
532                     op_name.text = op[0]
533                     for par in op[1]:
534                         op_parameter = ET.SubElement(option, 'Parameter')
535                         op_var = ET.SubElement(op_parameter, 'Var')
536                         op_var.text = par[1]
537                         op_type = ET.SubElement(op_parameter, 'Type')
538                         op_type.text = par[0]
539             position = ET.SubElement(pragma_, 'Position')
540             start = ET.SubElement(position, 'StartLine')
541             start.text = pragma.start_line
542             if(name.text != "BARRIER"):
543                 end = ET.SubElement(position, 'EndLine')
544                 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.start_line + pragma.end_line not in pragma_list:
562             pragma_list.append(pragma.start_line+pragma.end_line)
563             edge = ET.SubElement(edges_element, 'Edge')
564             source = ET.SubElement(edge, 'Source')
565             source.text = str(pragma.start_line) + str(pragma.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     :
584         return function
585     for child in function.children:
586         if (child.start_line) == key_start and ('BARRIER' not in child.type) and
587         child.parent[0].start_line == key_parent:
588             return child
589         else:
590             tmp_node = find_sub_node2(key_start, key_parent, child)
591             if tmp_node != None:
592                 return tmp_node
593     return None
594
595 def find_sub_node(node, function):
596     if (function.start_line) == node and ('BARRIER' not in function.type):
597         return function
598     for child in function.children:
599         if (child.start_line) == node and ('BARRIER' not in child.type):
600             return child
601         else:
602             tmp_node = find_sub_node(node, child)
603             if tmp_node != None:
604                 return tmp_node
605     return None
606
607 class Caller():
608     def __init__(self, original_caller, used_caller):
609         self.original_caller = original_caller
610         self.used_caller = used_caller
611         self.old_children = []
612
613 #adding to the main graph all the function which are called taking care of
614 #multiple connections between pragma and caller
615 def explode_graph(flow_graphs):
616     setted_callers = {}
617     for function in flow_graphs:
618         count = 0
619         caller_list = function.callerid
620         if caller_list != None:
621             for caller in caller_list:
622                 function_copy = copy.deepcopy(function)
623                 count += 1
624                 caller_node = find_node(caller, flow_graphs)
625                 if caller_node.start_line not in setted_callers:
626                     setted_callers[caller_node.start_line] = Caller(copy.copy(
627                     caller_node), caller_node)
628                 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     )
631     caller_node.children = []
632     caller_node.children.append(function_copy)
633     last_node = sched.get_last(function_copy)
634     last_node.children = children_list
635     for child in children_list:
636         child.parent.append(last_node)
637     else:
638         children_list = []
639         for child in setted_callers[caller_node.start_line].old_children:
640             children_list.append(child)
641             function_copy.parent.append(setted_callers[caller_node.start_line
642 ].used_caller)
643             setted_callers[caller_node.start_line].used_caller.children.append
644 (function_copy)
645             last_node = sched.get_last(function_copy)
646             last_node.children = children_list
647             for child in children_list:
648                 child.parent.append(last_node)
649
650 def get_parameter(parameter):
651     if parameter.find('Type') != None:
652         type_ = parameter.find('Type').text
653     else:
654         type_ = 'None'
655     return (type_, parameter.find('Var').text)
656
657 def create_map(optimal_flow):
658     for_map = {}
659     for flow in optimal_flow:
660         for task in flow.tasks:
661             if "splitted" in task.type:
662                 l = re.findall(r'\d+', task.type)
663                 id = str(l[0]) + "_" + str(l[2])
664                 if id in for_map:
665                     for_map[id].count += 1
666                     for_map[id].id.append(task.id)
667                 else:
668                     for_map[id] = Task(1, task.id)
669     return for_map
670
671 def add_new_tasks(optimal_flow, main_flow):
672     for_map = create_map(optimal_flow)
673     for key in for_map:
674         l = re.findall(r'\d+', key)
675         node_to_replace = find_node2(l[0], l[1], main_flow)
676         nodes_to_add = []
677
678         for i in range(for_map[key].count):
679             nodes_to_add.append(For_Node("splitted_" + node_to_replace.start_line
680 + "." + str(i), node_to_replace.start_line, node_to_replace.init_type,
681 node_to_replace.init_var, node_to_replace.init_value, node_to_replace.
682 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))))
678
679     for parent in node_to_replace.parent:
680         parent.children.remove(node_to_replace)
681     for n in nodes_to_add:
682         parent.add(n)
683         n.id = for_map[key].id.pop(0)
684         n.color = 'white'
685         n.from_type = node_to_replace.type
686
687     for child in node_to_replace.children:
688         child.parent.remove(node_to_replace)
689     for n in nodes_to_add:
690         n.add(child)
691
692
693 def add_flow_id(optimal_flow, task_list):
694     id_map = {}
695     for flow in optimal_flow:
696         for task in flow.tasks:
697             if "splitted" not in task.type:
698                 if task.start_line not in id_map:
699                     id_map[task.start_line] = task.id
700                 else:
701                     id_map[task.start_line + str(1)] = task.id
702     for task in task_list:
703         if "splitted" not in task.type:
704             if task.start_line in id_map:
705                 task.id = id_map[task.start_line]
706                 id_map.pop(task.start_line, None)
707             else:
708                 task.id = id_map[task.start_line + str(1)]

```

Code 2.3: profiler.py

```

1 from __future__ import with_statement
2 import os
3 import paragraph as par
4 import xml.etree.cElementTree as ET
5 import numpy
6 import re
7
8 def profileCreator(cycle, executable):
9     pragma_times = {}
10    function_times = {}
11    j = 0
12    param_string = ''
13
14    if os.path.exists("./parameters.txt"):
15        with open("./parameters.txt", "r") as f:
16            parameters = f.readlines()
17        for s in parameters:
18            param_string += s.strip()
19
20    for i in range(cycle):
21        print "profiling iteration: " + str((j + 1))
22        os.system("./" + executable + " " + param_string + ">/dev/null")
23        os.system("mv log_file.xml " + "./logfile%s.xml" % j)

```

```

24 root = ET.ElementTree(file = "./logfile%s.xml" % j).getroot()
25
26 for pragma in root.iter('Pragma'):
27     key = pragma.attrib['fid'] + pragma.attrib['pid']
28     if (key not in pragma_times):
29         pragma_times[key] = par.Time_Node(int(pragma.attrib['fid']), int(
pragma.attrib['pid']))
30     if ('callerid' in pragma.attrib):
31         if pragma.attrib['callerid'] not in pragma_times[key].caller_list:
32             pragma_times[key].caller_list.append(pragma.attrib['callerid'])
33     if ('loops' in pragma.attrib):
34         pragma_times[key].loops.append(int(pragma.attrib['loops']))
35     if ('time' in pragma.attrib):
36         pragma_times[key].time = pragma.attrib['time']
37     if ('childrenTime' in pragma.attrib):
38         pragma_times[key].children_time.append(float(pragma.attrib['
childrenTime']))
39         pragma_times[key].times.append(float(pragma.attrib['elapsedTime']))
40
41 for func in root.iter('Function'):
42     key = func.attrib['fid']
43     if (key in function_times):
44         function_times[key].times.append(float(func.attrib['elapsedTime']))
45     else:
46         function_times[key] = par.Time_Node(int(func.attrib['fid']), 0)
47         function_times[key].times.append(float(func.attrib['elapsedTime']))
48     if ('callerid' in func.attrib):
49         if int(func.attrib['callerid']) not in function_times[key].
caller_list:
50             function_times[key].caller_list.append(int(func.attrib['callerid
']))
51     if ('time' in func.attrib):
52         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     )
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].
141         replace("'",'').replace(' ','').replace('\','')
142
143     par.indent(tree.getroot())
144     tree.write('code.xml')
145
146 def get_table(profile_xml):
147     tree = ET.ElementTree(file = profile_xml)
148     root = tree.getroot()
149     table = {}
150
151     for func in root.iter('Function'):
152         table[func.find('FunctionLine').text] = []
153         if func.find('CallerId') != None:
154             l = re.findall(r'\d+',func.find('CallerId').text)
155             for j in l:
156                 table[func.find('FunctionLine').text].append(j)
157
158     return table
159
160 def getProfilesMap(profile_xml):
161     profile_graph_root = ET.ElementTree(file = profile_xml).getroot()
162
163     functions = {}
164     l = []
165
166     for func in profile_graph_root.findall('Function'):
167         f = par.Function(func.find('Time').text, func.find('Variance').text,
168             func.find('ChildrenTime').text)
169         f.callerid = []
170         if (func.find('CallerId') != None):
171             l = re.findall(r'\d+',func.find('CallerId').text.replace("[",",").
172             replace("]",","))
173             for id_ in l:
174                 f.callerid.append(id_)
175             functions[func.find('FunctionLine').text] = f
176
177     for pragma in profile_graph_root.findall('Pragma'):
178         if pragma.find('CallerId') != None:
179             callerid = pragma.find('CallerId').text.replace("[\',"").replace("\',"")
180             callerid = None
181         if (pragma.find('Loops') != None):
182             loops = pragma.find('Loops').text
183         else :
184             loops = 0
185         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 ))
186
187     return functions

```



```

1 import pargraph as par
2 import xml.etree.cElementTree as ET
3 import math
4 import copy
5 import time
6 import multiprocessing
7
8 class Queue():
9     def __init__(self):
10         self.q = multiprocessing.Queue()
11         self.set = False
12
13 #returns the optimal flows
14 #if time is to big for the number of possible solutions it does not work.
15
16 def get_optimal_flow(flow_list, task_list, level, optimal_flow, NUM_TASKS,
17                     MAX_FLOWS, execution_time, q):
18     if time.clock() < execution_time :
19         curopt = get_cost(optimal_flow)
20         cur = get_cost(flow_list)
21         if len(flow_list) < MAX_FLOWS and len(task_list) != level and cur <=
22             curopt :
23             task_i = task_list[level]
24             # test integrating the single task in each
25             for flow in flow_list :
26                 flow.add_task(task_i)
27                 get_optimal_flow(flow_list, task_list, level + 1, optimal_flow,
28                                 NUM_TASKS, MAX_FLOWS, execution_time, q)
29                 flow.remove_task(task_i)
30                 new_flow = par.Flow()
31                 new_flow.add_task(task_i)
32                 flow_list.append(new_flow)
33                 get_optimal_flow(flow_list, task_list, level + 1, optimal_flow,
34                                 NUM_TASKS, MAX_FLOWS, execution_time, q)
35                 flow_list.remove(new_flow)
36
37             if 'For' in task_i.type :
38                 #checks the possible splittings of the for node
39                 for i in range(2, MAX_FLOWS + 1):
40                     tmp_task_list = []
41                     #splits the for node in j nodes
42                     for j in range(0, i):
43                         task = par.For_Node("splitted_" + task_i.start_line + "." + str(
44                             j) + "_" + task_i.parent[0].start_line, task_i.start_line, task_i.
45                             init_type, task_i.init_var, task_i.init_value, task_i.init_cond, task_i.
46                             init_cond_value, task_i.init_increment, task_i.init_increment_value,
47                             task_i.time, task_i.variance, math.floor(float(task_i.mean_loops) / i))
48                         task.in_time = float(task_i.time) / i
49                         task_list.append(task)
50                         tmp_task_list.append(task)
51                     get_optimal_flow(flow_list, task_list, level + 1, optimal_flow,
52                                     NUM_TASKS + i - 1, MAX_FLOWS, execution_time, q)
53                     for tmp_task in tmp_task_list:
54                         task_list.remove(tmp_task)
55             else:
56                 if len(task_list) == level and len(flow_list) == MAX_FLOWS and cur <=
57                     curopt:
58                     if cur < curopt or (get_num_splitted(flow_list) > get_num_splitted(

```

```

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

```

```

106     get_num_splitted(flow_list) > get_num_splitted(optimal_flow) and
107     get_num_splitted(flow_list) < MAX_FLOWS/2) :
108         #print "actual_cost:", get_cost(flow_list), "optimal_cost:",
109         get_cost(optimal_flow)
110         del optimal_flow[:]
111         id = 0
112         #print "newflowset:"
113         for flow in flow_list:
114             for task in flow.tasks:
115                 task.id = id
116                 id += 1
117                 optimal_flow.append(copy.deepcopy(flow))
118
119 def get_num_splitted(flow_list):
120     num = 0
121     for flow in flow_list:
122         for task in flow.tasks:
123             if 'splitted' in task.type:
124                 num += 1
125     return num
126
127 #generator for the tasks of the graph
128 def generate_task(node):
129     if node.color == 'white':
130         node.color = 'black'
131         yield node
132     for n in node.children:
133         for node in generate_task(n):
134             yield node
135
136 def generate_list(l, node):
137     if node.color == 'white':
138         node.color = 'black'
139         l.append(node)
140     for n in node.children:
141         generate_list(l, n)
142
143 #returns the number of physical cores
144 def get_core_num(profile):
145     root = ET.ElementTree(file = profile).getroot()
146     return int(root.find('Hardware/NumberofCores').text)
147
148 #sets the color of each node to white
149 def make_white(node):
150     if node.color == 'black':
151         node.color = 'white'
152     for child in node.children:
153         make_white(child)
154
155 #returns the graph which contains the 'main' function
156 def get_main(exp_flows):
157     for i in range(len(exp_flows)):
158         if exp_flows[i].type == 'main':
159             return exp_flows[i]
160
161 #returns the last node of the input graph
162 def get_last(node):
163     if not node.children:
164         return node

```

```

152     else:
153         return get_last(node.children[0])
154
155 #returns the children with the least deadline - computation_time
156 def get_min(node):
157     minimum = float("inf")
158     found = False
159     for child in node.children:
160         if child.d == None:
161             found = True
162     if found == False:
163         #print "setting:␣",child.type,"@",child.start_line
164         for child in node.children:
165             min_tmp = child.d - float(child.in_time)
166             if min_tmp < minimum:
167                 minimum = min_tmp
168     return minimum
169
170
171 #sets the deadline for each task
172 def chetto_deadlines(node):
173     if node.parent :
174         for p in node.parent:
175             p.d = get_min(p)
176         for p in node.parent:
177             chetto_deadlines(p)
178
179 #applys the chetto algorithm to obtain the deadline and arrival time for
180     each task
181 def chetto(flow_graph, deadline, optimal_flow):
182     node = get_last(flow_graph)
183     node.d = deadline
184     chetto_deadlines(node)
185     flow_graph.arrival = 0
186     chetto_arrival(flow_graph, optimal_flow)
187
188 #gets the cost of the worst flow
189 def get_cost(flow_list):
190     if len(flow_list) == 0:
191         return float("inf")
192     else:
193         return max([flow.time for flow in flow_list])
194
195 def chetto_arrival(node, optimal_flow):
196     if node.children :
197         for child in node.children:
198             if child.arrival == None and all_set(child) == True:
199                 (a, d) = get_max(child, optimal_flow)
200                 child.arrival = max(a, d)
201                 chetto_arrival(child, optimal_flow)
202
203 def get_max(node, optimal_flow):
204     maximum_a = 0
205     maximum_d = 0
206     for p in node.parent:
207         if p.arrival > maximum_a and p.id == node.id:
208             maximum_a = p.arrival
209         if p.d > maximum_d and p.id != node.id:

```

```

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

```

```

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

```

```

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

```