Compiler Analysis and its application to OmpSs

Sara Royuela Alcázar

Master's Thesis

January 2012

Advisors:

Alejandro Duran González
Parallel Programming Models
Barcelona Supercomputer Center

Xavier Martorell Bofill
Computer Architecture Department
Technical University of Catalonia

A dissertation submitted in partial fulfillment of the requirements for the degree of: Master in Computer Architecture, Networks and Systems

Aknowledgments

I dedicate this work to the two persons who really encourage me to think that my work worth to be done. Thank you Dario and Alex, because without your help and conversations, I'm sure that I will not be here right now.

I thanks my family because they always think that I'm doing right, and this is helps many times to keep trying.

I thanks Jordi for the weekly evenings in the bar, that helped me to break my routine and enjoy some fresh air. I thanks Diego because he is always bearing my silly thoughts and makes me laugh.

A special acknowledgement to my Monday kids, without your smiles, your stories and our good times skating, all this process had been harder.

Abstract

Nowadays, productivity is the buzzword in any computer science area. Several metrics have been defined in order to measure the productivity in any type of system. Some of the most important are the performance, the programmability, the cost or the power usage. From architects to programmers, the improvement of the productivity has became an important aspect of any development. Programming models play an important role in this topic. Thanks to the expressiveness of any high level representation not specified for any particular architecture, and the extra level of abstraction they contribute against specific programming languages, programming models aim to be a cornerstone in the enhancement of the productivity.

OmpSs is a programming model developed at the Barcelona Supercomputing Center, built on the top of the Mercurium compiler and the Nanos++ runtime library, which aims to exploit task level parallelism and heterogeneous architectures. This model covers many productivity aspects such as the programmability, defining easy directives that can be integrated in sequential codes avoiding the need of restructuring the originals to get parallelism, and the performance, allowing the use of these directives to give support to multiple architectures and support for asynchronous parallelism.

Nonetheless, not only the convenient design of a programming model and the use of a powerful architecture can help in the achievement of good productivity. Compilers are crucial in the communication between these two components in computers. They are meant to exploit both the underlying architectures and the programmers codes. In order to do that, analysis and optimizations are the techniques that can procure better transformations.

Therefore, we have focused our work in the enhancement of the productivity of OmpSs by means of implementing a set of high level analysis and optimizations in the Mercurium compiler. They address two directions: obtain better performance by improving the code generation and improve the programmability of the programming model relieving the programmer of some tedious and error-prone tasks. Since Mercurium is a source-to-source compiler, we have applied these analyses in a high level representation and they are important because they are architecture independent and, thereupon, they can be useful for any target device in the back-end transformations.

Contents

List of Figures

List of Algorithms

Chapter 1. *Introduction*

Embedded and high performance computing systems research is leaded by the need to obtain more **productivity**. Each area of the computing sciences has its own fields of study in order to achieve this common objective. Between the avalanche of new architectures with faster components and new memory hierarchies, and the huge amount of languages that try to meet better the specific requirements of each application, **parallel programming models** are one of the most important topics. This interest comes from the fact that they can interact in different levels of deepness with both the architectures and the programming languages. Parallel programming languages allow the programmer to balance the competing goals of performance and programmability by implicitly or explicitly specifying different program properties such as the computational tasks, the mapping between these tasks and the processing elements, the communication network and the synchronization. **OmpSs** is a parallel programming model with implicit task identification and synchronization defined by high level directives. It extends OpenMP API to support asynchronous task parallelism and integrates different features of StarSs to support heterogeneous devices. The importance of this model relies on the easiness of using directives, its independence of the architecture and its expressiveness when defining both synchronous and asynchronous parallelism, as well as the scalability it contributes to allow the definition of different target architectures such as GPUs.

Hand by hand to the programming models are the compilers. We can distinguish between **source-to-source compilers** and back-end compilers. Even though back-end compilers are indispensable and they can provide many benefits by the knowledge they can have from the underlaying architectures, source-to-source compilers are a great vehicle to support research. They provide a wide range of activity for the development of high level analysis and optimizations that can exploit the characteristics of the codes without loss of **portability**. In this context, Mercurium is a source-to-source compiler developed for fast prototyping. This kind of infrastructure is a breeding ground for the research and testing of new proposals. In the world of the source-to-source compilers, many groups have based their efforts in the analysis and optimization of both programming languages and programming models. For example, the ROSE compiler group have built a platform for complex program transformations and domain-specific optimizations; more recently the have developed techniques of auto-parallelization and auto-tuning. Other example is LLVM, a project allowing transformation among different high level programming languages as C, Ada, FORTRAN or Java as well as many different levels of optimizations, from source- and target- independent optimizations to run-time optimizations. In the Mercurium group we do not want to offer a platform for aggressive optimizations or back-end dependent transformations. Instead of that, we want to provide a set of tools that can help in the improvement of productivity offering support for OmpSs.

Compiler analysis and optimizations are very valuable to achieve our goals because of the beneficial impact they can have in the processing of programming models and thus, the enhancement in the productivity in specific algorithms. In order to tackle some lacks in the Mercurium compiler, we have defined a set of analysis in the middle-end phase that allow us to improve both the performance of the generated code and the programmability of OmpSs. Our challenge is to adapt the classical analyses such as control flow, use-definition chains, liveness analysis and reaching definitions, into the parallel and heterogeneous behavior of OmpSs. We have defined a set of analyses that gather enough information to implement a few optimizations demonstrating the value of implementing architecture independent analysis in Mercurium and, by extension, to any other compiler. These optimizations have been directed to improve the generated code by analyzing the impact of using shared or private variables and to improve the programmability of OmpSs by analyzing the scope of variables in parallel codes to release the programmer of some tedious work while using tasks. We have tested these optimizations in different common algorithms and we will show the obtained results.

Most of the project has been developed within the Programming Models group of the Computer Sciences department at the Barcelona Computer Center. The main goal of this group is the research of new programming paradigms and the runtime system support for high performance of parallel applications. The group works on both multi-core and SMP processors with either shared- or distributed-memory systems and for both homogeneous and heterogeneous architectures using accelerators like GPGPUs. The exploration is supported with the development of the Mercurium compiler and the Nanos++ runtime library for fast prototyping. The usability of programming models is tested in different scenarios with OmpSs, which proposes extensions to standards like OpenMP.

The group focus its efforts in different projects approaching different as is composed by many divided in three different projects which are: the OmpSs programming model environment, the Mercurium source-to-source compiler and the Nanos++ runtime library. This project is framed in the context of the OmpSs project and the Mercurium project and aims to improve the code generated by Mercurium within the frame of OmpSs programming model.

The current project is the final dissertation of the Masterś degree in Computer Architecture, Networks and Systems (CANS), at the Computer Sciences Faculty of Barcelona (FBI), part of the Technical University of Catalonia (UPC). The project has been funded by the Barcelona Supercomputing Center (BSC), the European Commission through the ENCORE project (FP7-248647) and the ROSE group at Lawrence Livermore National Lab.

The rest of the document is organized as follows. Chapter 2 describes the motivation and goals of this thesis. Chapter 3 defines the methodology followed by along the project in order to achieve our goals. Chapter 4 describes the environment of the project and the main components used in its execution. Chapter 5 contains the different analyses we have implemented in the Mercurium compiler. Chapter 6 explains different OmpSs optimizations implemented in the Mercurium compiler based on the previous analyses and their evaluation. Chapter 8 concludes this dissertation and outlines the future work.

Chapter 2. *Motivation and Goals*

Parallel programming models as OmpSs and Runtime Libraries as Nanos++ play an important part in increasing the **productivity** of high-performance systems. Research compilers as Mercurium can snappily prototype new features to determine their effect. Mercurium generates intermediate code to exploit the Nanos++ runtime library and OmpSs is built on top these two components. The research nature of these projects leads us to implement analysis in Mercurium compiler that can help in the commitment of productivity. We do not try to implement aggressive optimizations such as auto-parallelization or loop transformations. It takes too much time and effort, and other compilers already focus in that area of research. Instead of that, we are in pursuit of the investigation about asynchronous parallelism and multiple devices execution.

Keeping in mind the previous arguments, in this project we want to focus essentially in two points: on one hand the compiler analysis to improve Mercurium code generation and get a better **performance** and on the other hand the enhancement of the **programmability** of OmpSs programming model. In order to achieve these goals, we require the implementation of a set of basic analysis in the middle-end phase of the compiler. We find at this point our *first challenge*: classical analyses for sequential and/or synchronous parallel programs have lacks of information to analyze the asynchronous parallelism introduced by tasks; some of these classical analysis have to be extended. As the basis of most of the data-flow analysis, we need to break down program control flow behavior for sequential and, synchronous and asynchronous parallelism. With this baseline analysis we can then implement a reasonable set of analysis that will be used to achieve our goals.

Based on the analysis performed in the compiler, we have defined two improvements of the productivity to be applied in Mercurium: one is the auto-definition of **data-dependencies** in **asynchronous tasks** to free the programmer from the tedious mission of defining the data dependencies for all the variables included in the task code; the other is the improvement of the performance of the generated code by **privatizing variables** that conservatively have been scoped as global. Here appears our *second challenge*: the automatic computation of data-dependencies requires the previous computation of the data-sharing for the involved variables; although some rules for automatic data-sharing have been defined until now, they are not for asynchronous tasks.

Thus, the major contributions of this thesis are:

1. We developed a new control flow representation containing information for sequential and synchronous and asynchronous parallelism by defining the key synchronization points that can guarantee correctness.

2. We implemented a set of basic data-flow analysis in the Mercurium infrastructure that includes: use-define chains, liveness analysis and reaching definitions.

3. We improved the programmability of OmpSs by automatically computing data-dependencies among tasks. In order to do that, we developed an algorithm to extend auto-scoping rules defined for OpenMP parallel constructs [LTaMC04] and analyze data-sharing in asynchronous tasks.

4. We developed a memory flush analysis. Along with liveness analysis, this analysis help us to privatize variables that had been conservatively scoped as shared.

Chapter 3. *Methodology*

As we explained in the previous chapter, we aim to improve the Mercurium compiler infrastructure to help us to enhance the productivity of OmpSs. The groundwork consists in developing a set of classical analyses adapted to synchronous and asynchronous parallel programs. We will reach our goal of productivity by implementing optimizations based on the previous analyses following two directions: the enhancement of the generated code to obtain a better performance and the improvement of the programmability of our programming model.

We have followed the time-line defined in the Gantt chart bellow. Find in pink color the initial planning of the work once it was carried out, and in blue color the work we had to redefine.

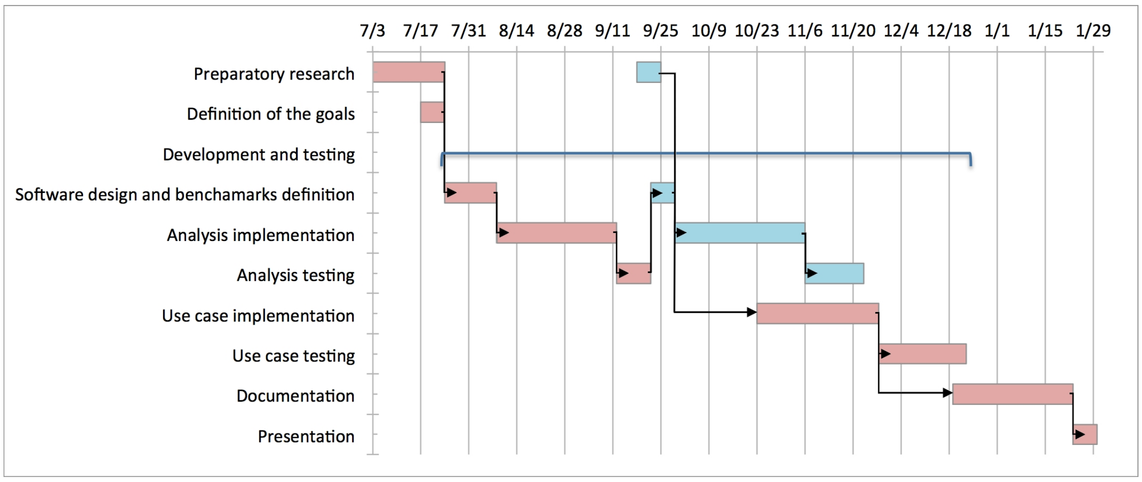


Figure 3.1: Gantt chart of the project

The following is an account of the methodology used for this project. We have organized the next paragraphs as the steps defined in Figure 3.1.

3.1 Preparatory research

The first step was the evaluation of different ideas within our area of interest that could be profitable for the two parts involved in the project: myself, as the developer of this thesis, and Barcelona Supercomputing Center, as the funder of the project. Once we defined in broad outline the main aspects we wanted to focus on, we started the analysis of the related work. We studied the state of the art of classical analysis for parallel programming models and we investigated some compilers implementing this kind of features such as ROSE or OpenUH. We read many publications about control flow analysis and data flow analysis to know the strengths and weaknesses of the current implementations.

3.2 Definition of our goals

Based on the previous study, we accurately defined the goals to be reached in this project. With the objective of developing some useful work in the frame of the BSC projects and highlighting that no analyses were implemented in Mercurium, we defined a set of classical analysis and different use cases to prove the benefits we can obtain with these analyses in terms of productivity.

3.3 Development and testing

To achieve our objectives, we used a spiral approach. With this technique we revisited the same concepts a few times while increasing the level of complexity in each pass. The advantage of this technique is that we never reached a position of no progress.

We first defined a minimum of requirements to fulfill and a set of benchmarks to test the results of every use case. Since the analyses defined in the previous step are dependents ones from the others, we developed sequentially a first approach of each. With this first release, we tested the results in our use cases. This work revealed some weaknesses in the implementation and some lacks in the process we had to solve in order to obtain profitable results. We redefined our analyses from a coarse-grained design into a fine-grained design to keep details that we had not took into account in the first sketch. Then, we tested again our benchmarks and we used these feed-back to iterate in this flow until we got the desired results. At the end we had an implementation that works for most of the C++ and OmpSs cases we have tested.

Because of the research nature of Mercurium, we found a remarkable difficulty during the development. Half way across our initial scheduling, our work team made the decision of changing the internal representation of the compiler. Since we have to deal directly with this representation, that modification affected substantially our work. At that point we had to go backwards and adapt our analysis to the new representation. Time constraints and work restrictions influenced the set of use cases we present in the project. We selected a set of analyses and optimizations that is representative enough to prove the benefits of our implementation.

3.4 Documentation and Presentation

Finally, we wrote the current dissertation as both part of the requirements of the Master degree and to serve as technical support for the features implemented in the Mercurium compiler.

Chapter 4. *Environment*

In this section we introduce the environment where the project has been developed. We have designed a set of compile-time analyses in the the context of three related projects: the OmpSs programming model, the Mercurium compiler and the Nanos++ run-time system. We will briefly describe OpenMP as it is the base of OmpSs. Finally, we will introduce the compiler where we have developed our thesis: Mercurium.

4.1 OpenMP

OpenMP is an interface that covers user-directed parallelization. The API provides a set of directives that allow the programmer to specify a structured block of code to be executed by multiple threads and to describe how the data will be shared between the threads. It uses the fork-join model of parallel execution. Parallel regions are defined by the constructs `parallel` and `task`. The directives to express worksharing are `for`, `sections`, `single` and `master`. Synchronization directives are used to protect data and order execution among threads. These directives are `critical`, `barrier`, `atomic`, `flush` and `ordered`.

OpenMP provides a **relaxed-consistency, shared-memory model**. This means that there are two kinds of memory: the *main memory*, accessed by all threads in any point of the execution, and the *threadprivate memory*, which is a private memory for each thread. The **flush operation** provides a guarantee of consistency between the *threadprivate memory* and the *main memory*. This operation can be done explicitly by the user or implicitly by the programming model (the `parallel` directive, worksharing directives or any combined worksharing directive imply a memory flush at the end of the execution of their associated block of code). The flush operation restricts some optimizations like reordering memory operations but allows some others like shared variables temporary privatization.

Some directives accept **data-sharing** attribute clauses. These clauses determine the kind of access (shared or private) of the variables inside the structured block associated with the directive's structured block. The different data-sharing clauses accepted are `private`, `shared`, `firstprivate` and `lastprivate` and their availability depends on each directive (for example, `lastprivate` clause is not allowed in `task` directives). A **data race** occurs when multiple threads write without synchronization to the same memory unit. Due to the laxity of the programming model this situation can appear frequently. To avoid this data hazards and maintain sequential consistency, OpenMP offers different methods: the definition of the proper data-sharing for every variable in a conflictive block of code and the synchronization directives to avoid simultaneous access to the same memory space.

All the rules defined by OpenMP model can be found in the Official OpenMP Specifications [Boa11]. For this project we have worked with the release 3.0 for C++.

4.2 OmpSs

OmpSs [DAB+11] is a parallel programming model which extends the OpenMP model to support asynchronous task parallelism. OmpSs manage to express the parallelism in such a way that is able to deal with both homogeneous and heterogeneous architectures. This programming model has been developed at the Barcelona Supercomputing Center (BSC) based on the StarSs [1] [PBAL09] and OpenMP.

The programming model is used in the simple form of introducing a few directives in the original code. In the next sections these directives are explained exhaustively with their features and showing different use cases.

4.2.1 The `task` directive

OmpSs extends the OpenMP task directive to suppport asynchronous parallelism by means of data-dependencies. The model ensures the correctness of the asynchronous execution by defining data-dependencies between the different tasks of a program. The syntax of the directive used to create a task is as follows:

```
#pragma omp task [clauses]
function_or_code_block
```

where:

– clauses is a list of new clauses that allows specifying restrictions about the dependencies. The allowed clauses are:

[1] StarSs is a task-based programming model developed at the Barcelona Supercomputing Center with two main objectives: to enable the automatic exploitation of the functional (task-level) parallelism and to keep applications unaware of the target execution platform.

  * input(list_of_expressions): evaluating an *lvalue* as an input dependence implies the related task cannot run until all previously defined tasks with an output dependence on the same expression have finished its execution.
  * output(list_of_expressions): evaluating an *lvalue* as an output dependence implies the related task cannot run until all previously defined tasks with an input or an output dependence on the same expression have finished its execution.
  * inout(list_of_expressions): evaluating an *lvalue* as an inout dependence means that it may behave as an input and as an output dependence.
  * concurrent(list_of_vars): this is a relaxed version of the inout clause. The task is scheduled taking into account input, output and inout previous clauses, but not concurrent clauses.
  * The rest of clauses allowed in OpeMP for the `task` construct, which are: if(scalar_logical_expression), final(scalar_logical_expression), untied, default(private | fisrtprivate | shared | none), mergeable, private(list_of_variables), firstprivate(list_of_variables) and shared(list_of_variables)

– function_or_code_block specifies the block of code that will be executed asynchronously in parallel.

It is important to note that the user assumes the liability on the correctness of the dependencies' definition. For the concurrent clause, as it relaxes the synchronization between tasks, the programmer must ensure that either the task can be executed concurrently or that additional synchronization is used (like atomic OpenMP directive).

4.2.1.1 Expression extensions

OmpSs allows two C/C++ extensions in the expressions that can appear in the data-dependence clauses. These extensions are:

– **Array sections**: allow to refer to multiple elements of an array or data addressed by a pointer. They can be specified as a range of accesses by the doublet [lower_bound : upper_bound].

– **Shaping expressions**: allow to recover the dimensions of an array that has been degraded to pointer. It is used by adding one or more [size] expressions before a pointer.

4.2.1.2 Execution model

As the tasks are created, they are inserted in the *graph of execution* that determines the dependences between tasks. This graph ensure the dependence satisfaction of every task. So, each time a task is created, its dependences are checked against those of the previous tasks and the new task is scheduled as soon as possible (i.e., when all its predecessors in the graph have already been completed).

4.2.1.3 Examples

An example of task creation with different clauses is shown in Listing 4.1. The task execution graph created for this graph is the one shown in Figure 4.1.

```
void compute ( int* A, int* NB ) {
  for ( int i = 1; i < N; ++i ) {
    #pragma omp task input(A[i-1]) inout(A[i]) output(B[i])
    foo( A[i-1], A[i], B[i] );

    #pragma omp task input(B[i-1]) inout(B[i])
    bar( B[i-1], B[i] )
  }
}

void foo ( int a, int& b, int& c ) {
  b = b + a;
  c = b;
}

void bar (int a, int& b) {
  b = b * a;
}
```

LISTING 4.1: OmpSs task code example

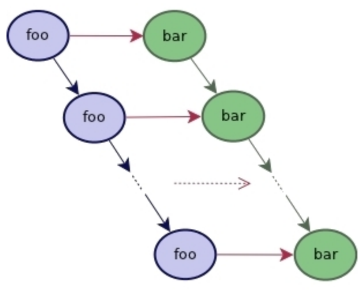


FIGURE 4.1: OmpSs dependency graph for code in Listing 4.1

Not just structured blocks, but also function definitions can be annotated with the task construct. In this case, each invocation of the function becomes the generation of an asynchronous parallel point. In Listing 4.2 we show an example of this kind of task definition.

```
#pragma omp task
void foo ( int i );

void bar ( )
{
  for ( int i = 0; i < 10; i ++ ) {
    foo(i);
  }
}
```

LISTING 4.2: OmpSs task code example at declaration level

The example in Listing 4.3 shows a merge sort code using tasks and the extended expressions allowed by OmpSs. Shaping expressions are used to transform pointer variable *a* to an array in the call to *merge* function. Array section regions are used to specify the region that will be used in each level of the recursion of the method *sort*.

```
void sort ( int n, int *a )
{
  if ( n < small ) seq_sort ( n, a );

  #pragma omp task inout( a[ 0 : n/2 ] )
  sort ( n/2, a );

  #pragma omp task inout( a[ n/2+1 : n ] )
  sort ( n/2, a[n/2+1] );

  #pragma omp task inout( [n] a )
  merge ( n/2 , a, a, a[n/2+1] );
}
```

Listing 4.3: OmpSs extensions example code: array sections and shaping expressions

4.2.2 The `taskwait` directive

The `taskwait` directive allows to enforce synchronization among tasks regardless of data-dependencies clauses. It is useful when there is no need for synchronous data output but a synchronization is required. Its syntax is the following:

```
#pragma omp taskwait [clauses]
```

where clauses can be:

- on (list_of_expressions): it allows waiting only to those previous tasks having some output dependence on the defined expressions.
- noflush: OpenMP enforces a memory flush immediately before and immediately after every task scheduling point. The use of this directive avoids the execution of these flushes.

4.2.2.1 Example

In the example shown in Listing 4.4 a code with tasks for the N-Queens problem is presented using a taskwait directive to wait the computation of the queens disposition in each recursion level. When all tasks in a given level have finished, then the number of possible solutions for that level is stored.

```
void nqueens(int n, int j, char *a, int *solutions, int depth)
{
  int *csols; int i;

  if (n == j) {
    *solutions = 1;
    return;
  }

  *solutions = 0;
  csols = alloca(n*sizeof(int));

  for (i = 0; i < n; i++) {
    #pragma omp task untied
    {
      char * b = alloca(n * sizeof(char));
      memcpy(b, a, j * sizeof(char));
      b[j] = (char) i;
      if (no_conflict(j + 1, b))
        nqueens(n, j + 1, b,&csols[i],depth+1);
    }
  }

  #pragma omp taskwait

  for ( i = 0; i < n; i++)
    *solutions += csols[i];
}
```

Listing 4.4: N-queens code with OmpSs taskwait directive

4.2.3 The `target` directive

As explained at the beginning of this section, the OmpSs programming model not only allows the creation of asynchronous parallelism, but also supports multiple platforms. To support heterogeneity, a new construct is introduced with the following syntax:

```
#pragma omp target [clauses]
task_construct | function_definition | function_header
```

where clauses can be:

- device(device_name): it specifies the device where the construct should be targeted. If no device clause is specified, then SMP device is assumed. The other currently supported target is CUDA for GPGPUs.
- copy_in(list_of_vars): it specifies the set of shared data that must be transferred to the device before the execution of the code associated to the construct.

– copy_out(list_of_vars): it specifies the set of shared data that must be transferred from the device after the execution of the code associated to the construct.

– copy_inout(list_of_vars): it specifies the set of shared data that must be transferred to and from the device, before and after the execution of the associated code.

– copy_deps: this clause specifies that the dependence clauses of the attached construct (if there exists) will have also copy semantics; it means that input dependencies will be considered as copy_in variables, output dependencies as copy_out variables and inout as copy_inout. If the attached construct has a `concurrent` clause, then all the dependencies are considered as `inout`.

– implements: this clause specifies that the code is an alternate implementation for the target device and it could be used by the target instead of the original if the implementation considers it appropriately.

4.2.3.1 Example

In the code shown in Listing 4.5 a new task is created for function *scale_task* and its target is a CUDA device. With the clause copy_deps in the target directive, we say that all the dependencies specified in the following task directive will be copied to/from the device. In this case, the whole *c* array will be copied to the device at the beginning of the execution and the whole *b* array will be copied from the device at the end of the execution.

```
#pragma omp target device (cuda) copy_deps implements (scale_task)
#pragma omp task input ([size] c) output ([size] b)
void scale_task_cuda(double *b, double *c, double scalar, int size)
{
  const int threadsPerBlock = 128;
  dim3 dimBlock;

  dimBlock.x = threadsPerBlock;
  dimBlock.y = dimBlock.z = 1;

  dim3 dimGrid;
  dimGrid.x = size/threadsPerBlock+1;

  scale_kernel<<<dimGrid,dimBlock>>>(size, 1, b, c, scalar);
}
```

Listing 4.5: OmpSs target directive example code

4.3 The Mercurium compiler

Mercurium is an agile source-to-source compiler supporting C, C++ and Fortran that aims at easy **prototyping** of parallel programing models. The goal of Mercurium is to rewrite, translate and mix the input source code into another source code that is fed into a object-code generating compiler. In this process, different constructs are recognized and transformed to calls to the runtime system enabling parallel execution. Mercurium does not build architecture dependent back-ends, instead, it supports the invocation of many native compilers as gcc, icc or nvcc. Mercurium is useful transforming high level directives into a parallelized version of the application, as well as profiling, instrumenting and synthesizing information at compile time. It is not useful for performing hard optimizations in the code; this area of research is develop in other compilers like ROSE, LLVM or Open64.

There are different parts in the compilation process. In the next paragraphs we explain the specifics of each step. Figure 4.2 outlines an schema of the whole process.

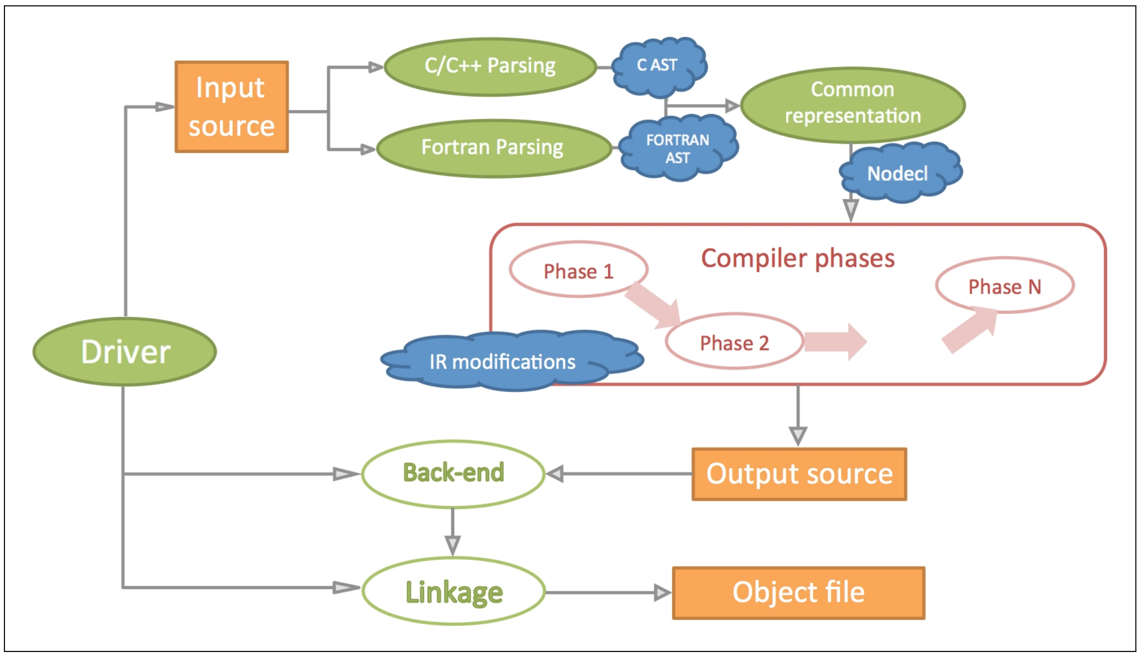


FIGURE 4.2: Mercurium compilation stages

4.3.1 Parsing

The compiler parses each input file by creating the **Abstract Syntax Tree (AST)** that contains the input code. Once the tree is built, a classical type-checking is performed creating the symbol table for each scope, removing ambiguities and synthesizing all expressions types. This non-ambiguous tree is used to the costruction of the Internal Representation, called ***Nodecl***, which will be used in the next compiler phases. *Nodecl* is also an AST but it differs from the previous one in some aspects:

– Nodecl does not contain declarations. Instead of that, it includes a new node called **CONTEXT** for every block of code creating a new scope. The **CONTEXT** node stores information about the different scopes that apply for the given context (global, namespace, function, block and current).

– Nodecl is aimed to represent with the same structure both C/C++ and Fortran. That means that similar constructs in the two languages are represented by the same type of nodes in Nodecl. This step is very useful for the next phases in the compiler since in most of the cases, the phases will not need to have specific implementations for each language.

```
double foo(int n)
{
  int i, res;

  #pragma omp parallel for
  for (i = 0; i < n; i++)
  {
    res += i;
  }
  return res;
}
```

Listing 4.6: Code snippet with OmpenMP parallel for construct

In Figure 4.3 we show the *Nodecl* for the code in Listing 4.6. It is the very essential structure, containing just the kind of the nodes an their relations. The structure starts with the function *foo* in the top level. Function code node is the root of a compound statement containing the function code. This compound statement has two children, the *pragma parallel for* (pink frame) and the *return* statement. Notice here that no definitions appear in the tree while the code declares the variables *i* and *res* at this level; information about declarations is attached to the tree but not as a node. Finally, hanging from the pragma appears the loop statement (green frame). Notice also that symbols (blue boxes) appear always as a leaf of the tree. Other kind of nodes are always leafs, like literals. Other significant aspect to realize in the tree are the context nodes inserted for each new context created in the input code (yellow boxes).

For more details, we show in Figure 4.4 the information about the context. Other nodes have been removed to aid to comprehension of the tree. Specifically, we display the contexts generated by the function, the **pragma omp parallel** and the *for-loop*. In that case, the global, the namespace and the function scopes are the same for the three contexts. The block scope is different for each one because each one is creating a new scope. The relations of ownership are shown through the dotted edges labeled as *contained_in*.

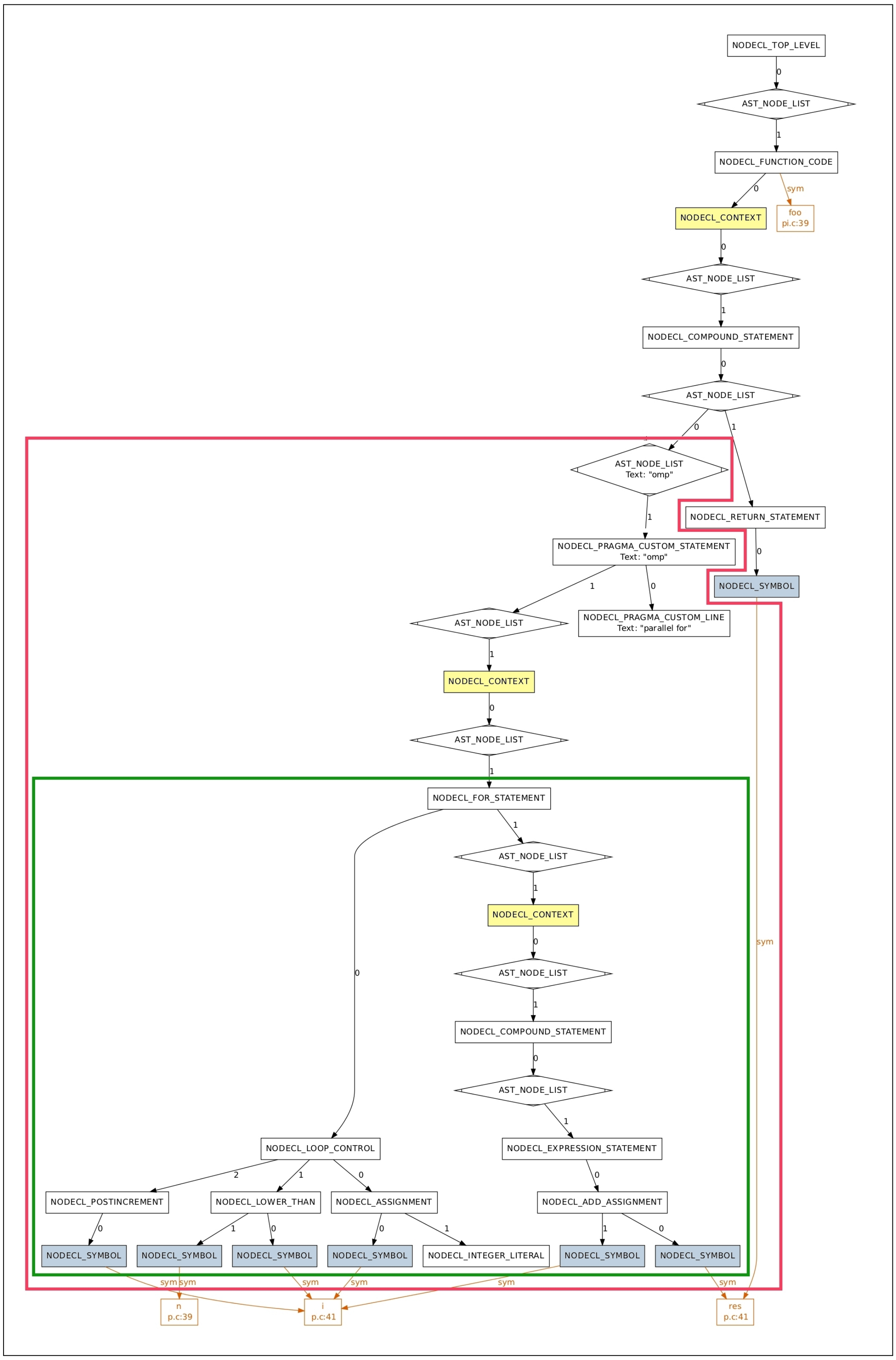


FIGURE 4.3: *Nodecl* generated from code in Listing 4.6

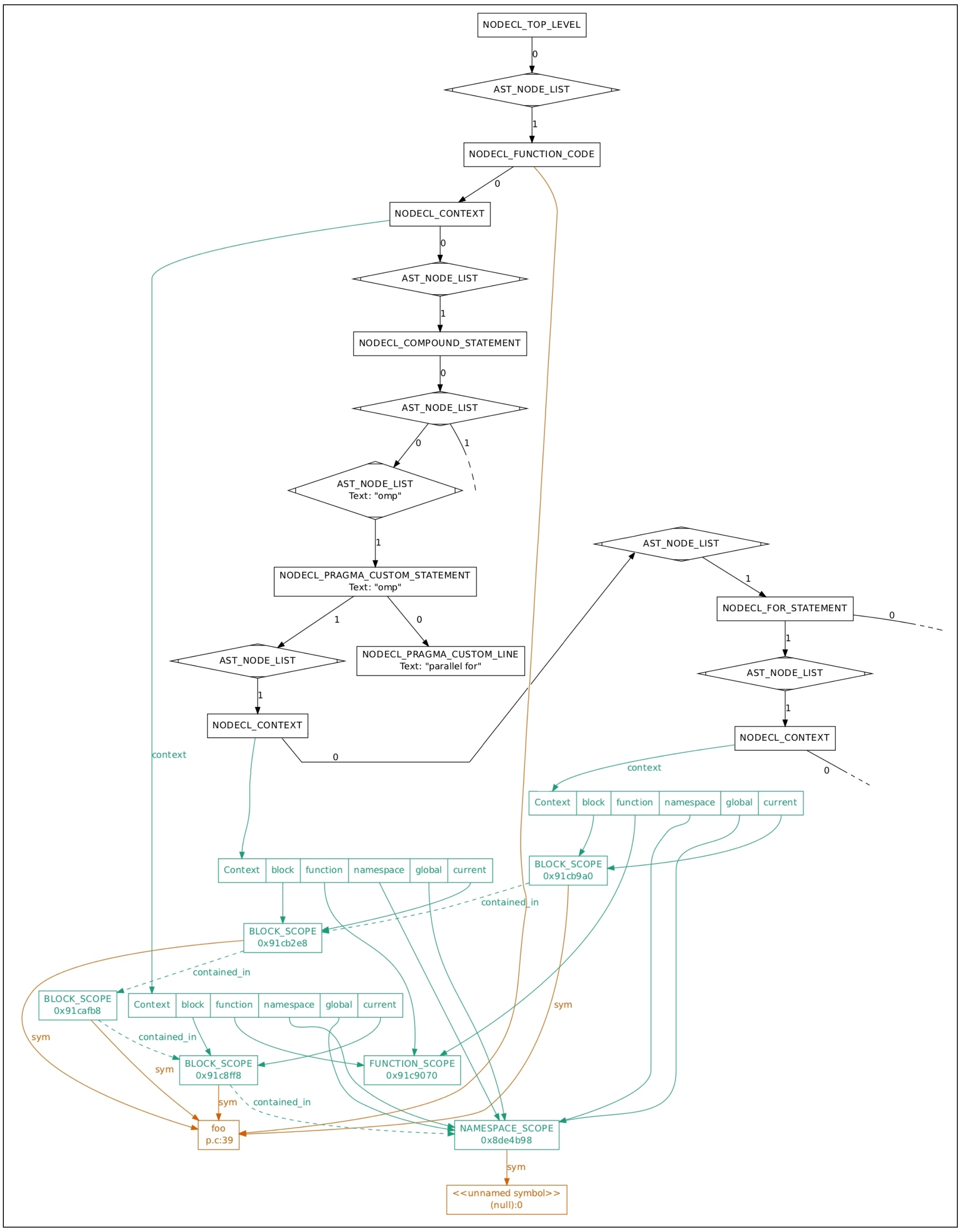


Figure 4.4: *Nodecl* snippet with context information from code in Listing 4.6

4.3.2 Compiler phases

The compiler phases are a set of dynamic libraries that work as a pipeline. These phases are written in C++ and they are enabled or disabled depending on the profile set in the compiler command line. The unambiguous AST *Nodecl* arrives to the first phase and a common internal representation (IR) is used among the phases. Nonetheless, each phase can create a new IR that will be used in the later phases. The Data Transfer Object (DTO) pattern is used to transfer data between the phases. The DTO is just a dictionary containing a string as the key and an Object as the value. In any point of the compilation process we can find available the *translation_unit* IR with the processed code. A powerful way to deal with trees has been implemented just recently. Following the Visitor Pattern, traversals through the *Nodecl* can be performed completely separated from the operation to be performed during this traversal. The compiler provides exhaustive and base visitors and they can be easily extended for particular purposes.

For this thesis, we have added a new phase to the pipeline that can be activated to enable the different analysis. The analysis methods can be called anywhere in the pipeline as well, without being necessary to execute the entire phase. The difference is that the phase will analyze all the translation unit while by calling the methods, the programmer will use the analysis on demand, analyzing just the codes he is interested in. Since *Nodecl* is a common IR for different input languages, the analysis we will implement here will be always **language independent**.

4.3.3 Code generation

The synthesis part generates an output code which is the conclusion of all transformations performed in the previous steps. Since the intermediate representation is the same for the different accepted languages by the compiler, information about the input must pass through the previous stages until this point.

4.3.4 Object code generation

Finally, a back-end compiler and a linker are invoked to generate object code. This will depend on the profile set to the compiler at the compiler command line.

Chapter 5. *Analysis*

Traditional compiler analysis play an important role in generating efficient code. The classical analysis are quite mature and routinely employed in compilers. Among the most common methods in compiler analysis for optimizing code, flow analysis is a technique for determining useful information about a program at compile time. This is the root of a set of analysis that permit us both the analysis and the optimization of OmpSs codes. The handicap of analyzing parallel codes is that we have to adapt the classical analysis to keep information about parallel execution.

We built a graph for control flow analysis. This graph represents all OpenMP 3.0 constructs and OmpSs specifics. The graph also stores additional information about the clauses associated to the constructs, if applicable. With this data structure, we can calculate **data flow analyses** such as use-definition chains, liveness information and reaching definitions. We have implemented an specific loop analysis to determine accessed ranges in arrays with restricted loop definition conditions. In the next sections we explain the details of each one of these analysis.

We have created and **API** providing different the analysis. Compiler developers can ask to analyze any piece of code represented by the compiler intermediate representation (IR). Since the different compiler phases can change this representation, the application of the analyses at different points of the compiler phase pipeline can return different results. While analyzing, developers must remember the dependencies existing between some of the analysis. This means that asking for reaching definitions without previously having computed liveness analysis will cause a *null* result. For testing purposes we have added a new phase in the compiler which analyzes the whole translation unit. Finally, we have created two new **debug options**: a verbose mode to show the result of the different analyses at compilation time and a printing mode that creates a file in DOT language with the control flow graph and all the information computed during the analyses embedded in the nodes of the graph.

5.1 Parallel Control Flow Graph (PCFG)

Flow analysis techniques allows determining path invariant facts in a given program. This is a key tool in compiler's analysis due to the huge list of optimizations that can be addressed with flow analysis (constant subexpression elimination, constant propagation, dead code elimination, loop invariant detection, induction variable elimination, range analysis, and a long etcetera).

The problem of the flow analysis is solved by the construction of a graph commonly known as **Control Flow Graph** (CFG). Building this graph for sequential codes does not introduce many challenges but in our case, we aim to implement a graph that must be able to correctly represent the semantics of OmpSs parallel codes. And not only that, but we also bear in mind that we are implementing this analysis in a research compiler such as Mercurium, and that led us to think in an extensible and scalable implementation. Assuming these premises, we have built a Parallel Control Flow Graph (PCFG) called *Extensible Graph* (EG) that allows both intra-procedural and inter-procedural data-flow analysis, and both intra-thread and inter-thread. We have created an API that allows the construction of the EG from a portion or the whole IR. In Figure 5.1 we show the basic class diagram of the components of and Extensible Graph. Basically, a graph is formed by one node; one node can contain other nodes inside, and the nodes are interconnected by edges. To traverse the graph we have specified a class which implements the visitor pattern among the nodes in the AST.

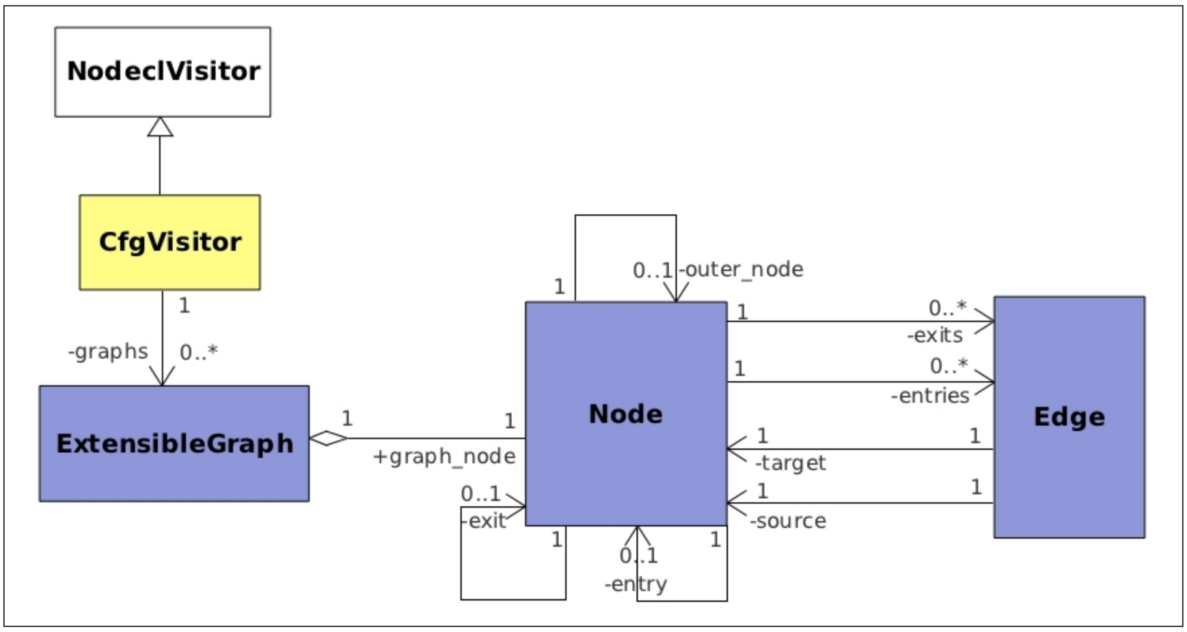


FIGURE 5.1: Basic class diagram for the PCFG

5.1.1 The Extensible Graph

The *Extensible Graph* is a directed graph formed by a 2-tuple $< id, N >$ where *id* is the identifier of the graph and *N* is the node containing the flow graph. This structure models the control flow of a section of code being that a whole function code or just a statement. The data structure contains only structural information, this is nodes and the directed edges connecting these nodes. We have created different kinds of nodes and edges to represent C++ statements and OmpSs specifics. All the semantics are linked to the structure as a pair of $< Name, Object >$. Each kind of element implies a series of additional attributes that will be linked to it. It is important to note that this way of attaching information to one object has some advantages and disadvantages. As a disadvantage, the implementation of this object leaves to the programmer the responsibility of maintaining the correctness of the data structure but, as an advantage, we obtain a structure that is clean and agile, free of specific attributes for every case. In the next sections we explain the details of the two elements, nodes and edges, and the particularities of OmpSs nodes.

5.1.1.1 Node

A node is a 3-tuple of $< Id, Entries, Exits >$ where *Id* is the unique identifier of a node within a given graph, *Entries* is the set of edges coming from the nodes of which the current node depends on and *Exits* is the set of edges to nodes that depend on the current node. Moreover, as we said before and depending on the data represented, each node will have additional linked attributes. We have defined the following node types:

– Basic nodes (They contain a expression or a set of expressions):

  ∗ **BB**: this node contains a *Basic Block* [1] .

  ∗ **Labeled**: it is a special kind of **BB** node that can be a jump target.

  ∗ **Function Call**: it is a special kind of **BB** containing a function call. We keep it separated because we need some analyses to determine the flow behavior of this kind of expression.

– OmpSs nodes (They refer to OmpSs instances in the original code):

  ∗ **Pragma Directive**: it contains a pragma directive.

  ∗ **Flush**: it contains a flush directive.

  ∗ **Barrier**: it contains a barrier directive.

  ∗ **Taskwait**: it contains a taskwait directive.

– Structural nodes (They aid the composition and comprehension of the graph)

[1] A Basic Block is a portion of code that has one entry point, meaning no code within it is the destination of a jump instruction anywhere in the program, and one exit point, meaning only the last instruction can cause the program to begin execution code in a different Basic Block.

    * **Entry**: it is added at the very beginning of a **Graph** node. Any flow that traverses the graph goes in through its **Entry** node.
    * **Exit**: it is added at the very end of a **Graph**. Any flow that traverses the graph goes out through its **Exit** node.
    * **Graph**: node containing a set of nodes structured as an EG. The first node in the graph is always an **Entry**, which is the dominator of all the nodes inside the graph (except itself), and the last node is always an **Exit**, which is the post-dominator of all nodes inside the graph (except itself).
  - Temporary nodes (They represent simple control structures and they are used only during the construction of the graph. Afterwards, they are removed as nodes and we only maintain in the EG their flow information):
    * **Break**: it represents a break statement.
    * **Continue**: it represents a continue statement.
    * **Goto**: it represents a goto statement node.

The linked data available and/or mandatory for each node is listed bellow:

- **Node Type**: this is the type of the node and is one of the values listed above. This attribute is mandatory for every node.
- **Outer Node**: this is a pointer to the *Graph* node containing the current node. All nodes but the outer most one have an **Outer Node**. For the outer most node (N from the 2-tuple conforming an EG), the **Outer Node** is *null*.
- **Statements**: this is the list of statements contained in the node. Only *Basic* nodes have this attribute.
- **Label**: this attribute has different meanings depending on the node it is applied to. For *Labeled* and *Goto* nodes, it contains the symbol representing the label or the jump target, respectively. For *Graph* nodes representing a block of code, the label contains the statement that creates the block of code; for example, in OpenMP nodes, the label contains the pragma line of the construct and for for-loop nodes, the label contains the control of the loop.
- **Graph Type**: this attribute only applies for *Graph* nodes and it contains the type of the graph node. It can be one from the list below:
  * **Extensible Graph**: this is the most outer node of a set of nodes. There is one and only one node of this kind in every EG and it is the N value of the 2-tuple representing the EG.

  * **Split Expression**: it is the result of a statement that has been split in the CFG due to its flow semantics. It can be, for example, a expression containing inside a function call: in that case a node containing the function call is created first, and then follows the node with the whole expression; both nodes will be included in a *Graph* node.
  * **Function Call**: all *Function Call* nodes are embedded in a *Graph* node for analysis purposes.
  * **Conditional Expression**: conditional expressions are special statements that contain an implicit flow. The different nodes created from this kind of expression are embedded in a *Graph* node.
  * **Loop**: it contains the structure of nodes created from the statements inside a loop.
  * **Omp Pragma**: it contains the structure of nodes created from the block code related to a pragma directive.
  * **Task**: it contains the structure of nodes created from the block code related to a task.

The attributes defined above are those that are created during the construction of the graph. Posterior analyses will add more attributes to the different nodes. The specific attributes added by each analysis are specified in the section related to the specific analysis.

5.1.1.2 Edge

An edge is a 2-tuple of $< Entry, Exit >$ where *Entry* is a pointer to the node source of the edge and *Exit* is a pointer to the node target of the edge. It links two nodes unidirectionally. We have defined different kind of edges:

– **Always**: this is an edge that connects two nodes accomplishing that, once the source node has been executed, the target will always be the very next to be executed.
– **True**: this is an edge that connects a source node containing a condition and a target node containing the very next node to be executed when the condition is fulfilled.
– **False**: this is an edge that connects a source node containing a condition and a target node containing the very next node to be executed when the condition is not fulfilled.
– **Case**: this is an edge connecting the control expression of a switch statement with the first node created by a given case of this switch.
– **Catch**: this is an edge connecting any expression that might be an exception with the first node created by the handler related to this exception.

This kind of edge does not imply that the target node will be executed every time the source node is executed, because some analyses are needed to determine that.

– **Goto**: this is an edge connecting a *Goto* node with a *Labeled* node.

The linked data available and/or mandatory for each edge is listed below:

– **Edge Type**: this is the type of the edge and must be a value from the list above. This attribute is mandatory for every edge.
– **Is Task**: it marks the edge as a non flow edge. This edge mark the point where an OpenMP task is declared and the point where a task code is synchronized with the main memory. It entails a different analysis than the other edges.
– **Is Back Edge**: it marks an edge as a backward edge encountered in a loop iteration.

5.1.1.3 Example

In Figure 5.2 we show the EG corresponding to the matrix multiply code of Listing 5.1. Among the different elements shown in the figure, we want to emphasize the loop constructions and the different edges (*True* and *False*; the edges remaining without a label are *Always* edges) generated by the conditions. Note that for the loop graph node, the initialization expression remains outside. That is because this statement do not belong to the set of statements repeated within the loop ranges.

5.1.2 Specifics of OpenMP

Classical analysis must be adapted to capture the parallelism expressed by OpenMP programs as well as the asynchronism expressed by OmpSs. Some parallel representation of the CFG have been already presented [Sar97, HEHC09]. We define an alternative representation of the Parallel Control Flow Graph (PCFG) for OmpSs. The PCFG expressed with the Extensible Graph is built as follows:

– A *Graph* node is built for every OpenMP constructs like `parallel`, `task` and the worksharings.
– All implicit memory flush operation introduced by the OpenMP directives are made explicit in the graph.
– For every OpenMP worksharing without a *nowait* clause we add a *Barrier* node at the end of the *Graph* node containing the pragma construct.

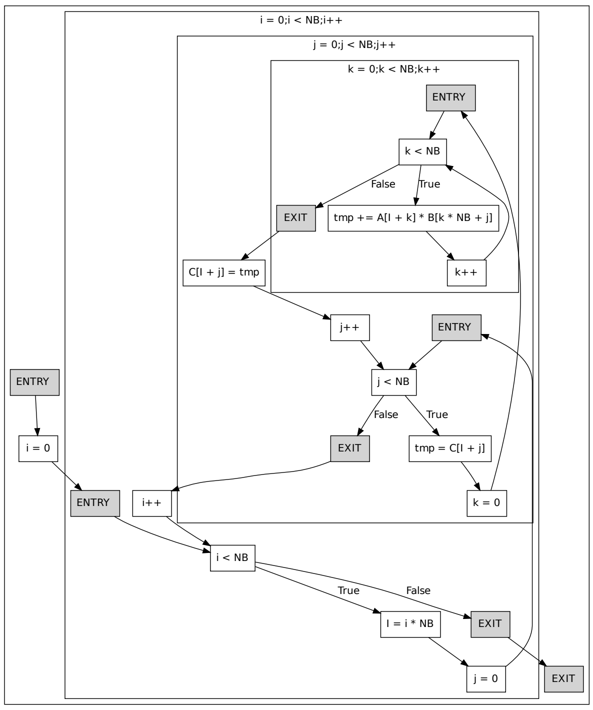


FIGURE 5.2: EG for code in Listing 5.1

```
void matmul(double *A,
            double *B, double *C,
            unsigned long NB)
{
  int i, j, k, l;
  float tmp;
  for (i = 0; i < NB; i++)
  {
    l = i * NB;
    for (j = 0; j < NB; j++)
    {
      tmp = C[l+j];
      for (k = 0; k < NB; k++)
      {
        tmp += A[l+k] * B[k*NB+j];
      }
      C[l+j] = tmp;
    }
  }
}
```

LISTING 5.1: Block partitioned Matrix Multiply

– A barrier operation implies a flush during its execution. We represent this action by adding to every barrier node *b* one flush node as dominator of *b* and another flush node as post-dominator of *b*.
– We add marks at the beginning and the end of every function graph and in the entry and exit point of every function call, where we assume memory flushes are done to ensure the correctness of the memory model.
– OmpSs tasks are analyzed in a specific way taking accounting for either their parallelism and the uncertainty they introduce in the parallel flow.

In the following paragraphs we show different examples of codes and the PCFG we generate. We have chosen a set of codes containing different remarkable C++ structures as well as OpenMP and OmpSs directives.

We define in Listing 5.2 a simple example of OpenMP sections. The EG generated is the one shown in Figure 5.3. A **GRAPH** node is created for every `section`. All the edges exiting from the dominator node of the *sections* node are **ALWAYS** edges. This means that those codes can be executed in parallel depending on the availability of threads. All sections are embedded in a **GRAPH** node that contains the `sections` directive. The OpenMP specification says that there is an implicit barrier at the end of a `sections` construct. We add this barrier with its respective surrounding **FLUSH** nodes before the **EXIT** node.

In Listing 5.3 we show an example with a combined worksharing (`parallel + for`) with and without the presence of a `nowait` clause. In Figure 5.4 there is the EG resultant of this code. One can see the difference between the loop with a `nowait` clause, which finalizes its execution with no synchronization node, and the loop without the `nowait` clause, that adds a **BARRIER** node with its implicit **FLUSH** nodes before and after the barrier. At the end of the parallel region, as specified by the OpenMP model, another **BARRIER** is inserted before the **EXIT** node.

```
void sect_example()
{
#pragma omp parallel sections
  {
#pragma omp section
     XAXIS();
#pragma omp section
     YAXIS();
#pragma omp section
     ZAXIS();
  }
}
```

Listing 5.2: OpenMP sections example

```
void parallel_for_nowait_example(int n, int m,
        float *a, float *b, float *y, float *z)
{
  int i;
#pragma omp parallel
  {
#pragma omp for nowait
     for (i=1; i<n; i++)
       b[i] = (a[i] + a[i-1]) / 2;

#pragma omp for
     for (i=0; i<m; i++)
       y[i] = sqrt(z[i]);
  }
}
```

Listing 5.3: OpenMP worksharing example

In Listing 5.4 we show a code for calculating the pi number using OpenMP tasks. One `task` is generated for each iteration of a loop contained in a `parallel` region. We show in Figure 5.5 the EG built for this code. The `critical` construct is embedded in a **GRAPH** node surrounded by two *Flush nodes*. For the `single` construct no additional synchronization node is added because of the existence of a `nowait` clause. The `parallel` construct adds a **BARRIER** with its surrounding **FLUSH** nodes. Note the different nature of the edges connecting the task with its dominator and post-dominator. The first corresponds with the scheduling point of the task (the first moment where the task can be executed) while the second corresponds to the synchronization point of the task (the last moment when the task can be executed).

5.2 Use-definition chains

The first step in liveness analysis is to compute, for every node in the graph, which variables are used and/or defined. We follow an algorithm that computes this information in two ways: from top to bottom, regarding the flow control, and from inside to outside regarding the topology of the graph (a given **GRAPH** node will compute recursively the use-definition information of its inner nodes and then will propagate the information to itself). This analysis will add three

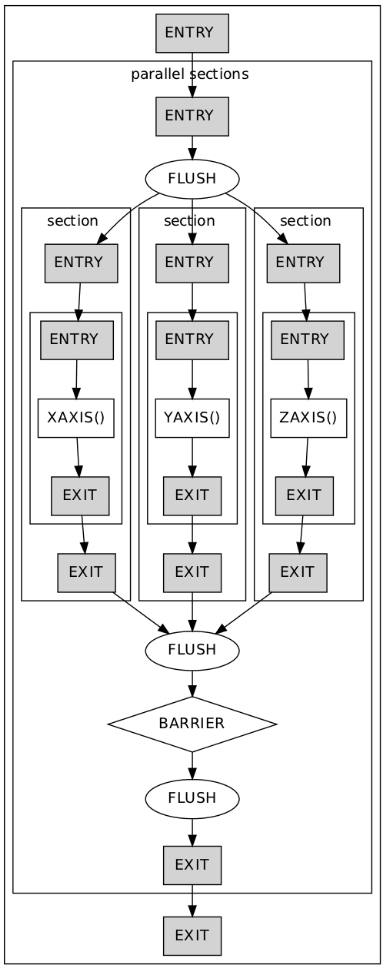


Figure 5.3: EG for code in Listing 5.2

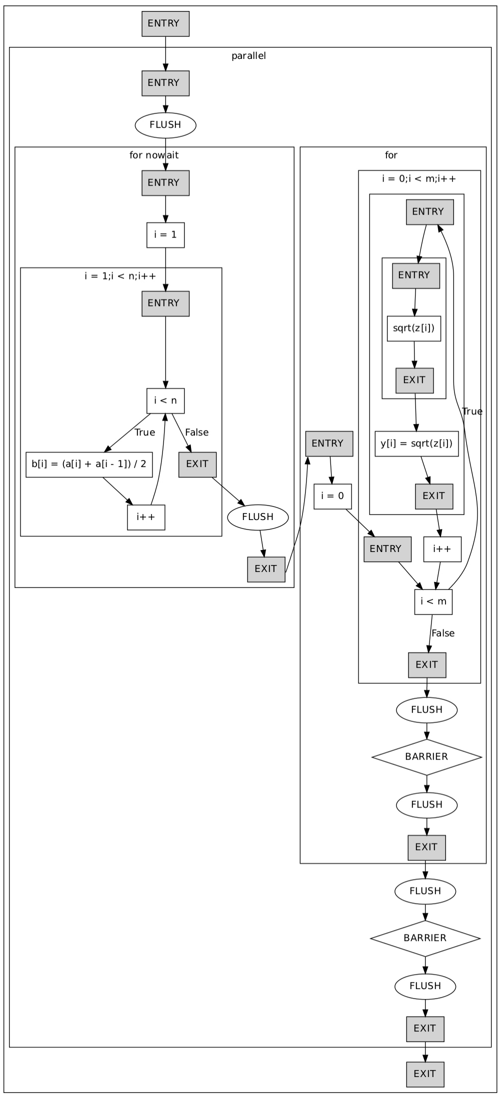


Figure 5.4: EG for code in Listing 5.3

```
double pi(int n) {
  const double fH = 1.0 / (double) n;
  double fSum = 0.0, fX;
  int i;
#pragma omp parallel
#pragma omp single private(i) nowait
  for (i = 0; i < n; i += 1) {
#pragma omp task private(fX) firstprivate(i)
    {
      fX = f(fH * ((double)i + 0.5));
#pragma omp critical
        fSum += fX;
    }
  }
  return fH * fSum;
}
```

Listing 5.4: Pi computation with OpenMP tasks

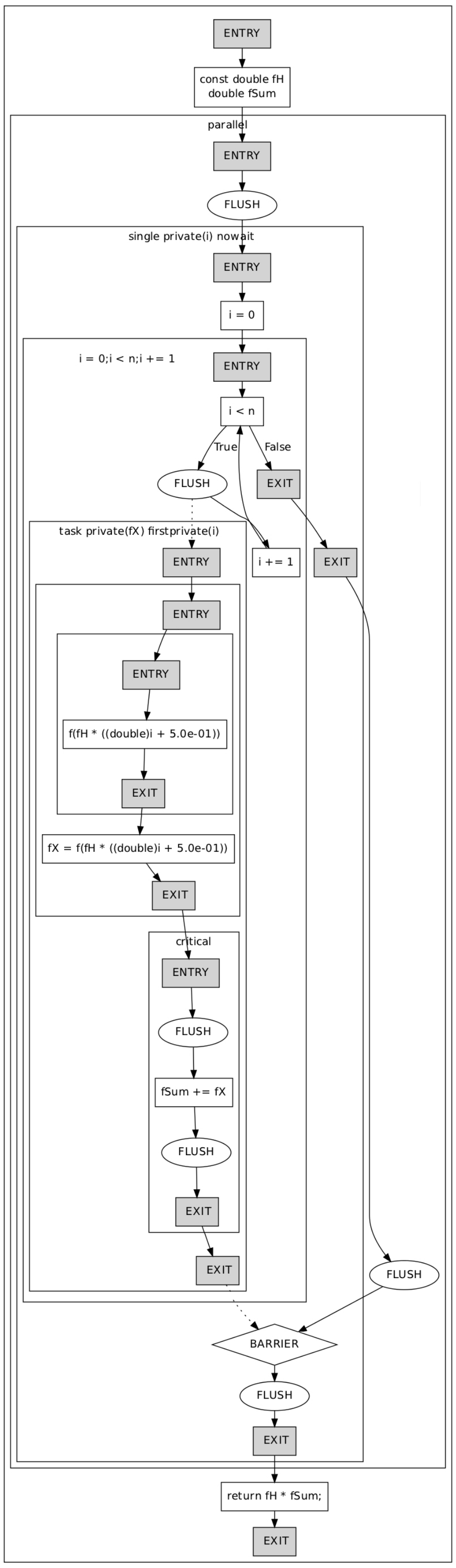


Figure 5.5: EG for code in Listing 5.4

new attributes to every node in the graph:

– Upper Exposed (UE): set of variables that are used before being defined within the current node.

- Killed (KILL): set of variables that are defined within the current node.
- Undefined behavior (UNDEF): set of variables which we are not able to define their usage.

When the information is being propagated from inner nodes to its outer **GRAPH** node, we follow a recursive depth traversal from the **ENTRY** node of the graph until the **EXIT** node of the graph. The post-condition of the traversal is that the current UE, KILL and UNDEF sets contains the information of the current node combined with the concatenated information of all its children. Given a specific step of this computation we consider the current node s and the set the target nodes $t_1, ..., t_n$ that are reachable from s. The algorithm works as follows:

1. The current sets (UE, KILL and UNDEF) are initialized with the info of s.
2. Three auxiliary sets (UE_AUX, KILL_AUX and UNDEF_AUX) are created concatenating the info from every node in $t_1, ..., t_n$. This concatenation takes into account including expression such us arrays and classes. That means for example that, if we find the use of an access of a determined position in an array but the whole array has been already used, the access of the specific position will not be added into the list because it will be redundant. If it happens the contrary, we find the use of a whole array and it already existed in the list the use of a specific position, then the access to the specific position will be deleted from the list and the access to the whole array will be added. In addition, during this step, if some variable is added in the UNDEF_AUX list, this variable or any form of the variable is deleted from the other two lists (UE_AUX and KILLED_AUX).
3. Finally we complete the current sets info with the info computed in the children. During this process, any variable appearing in the UNDEF set will not be propagated from any of the AUX sets to its corresponding current sets. If some variable appears in the KILL set, then it will not be propagated from UNDEF_AUX to UNDEF or from UE_AUX to UE. The rest of variables will be propagated from the AUX sets to the current sets by following the same rules for arrays and classes that have been described in the previous step.

Since we perform inter-procedural analysis, use-definition chains are computed recursively in functions calls. When we find a function call during the analysis of a graph, we stop analyzing the current graph to analyze the function called. In this situation, one and only one of the following cases must be applied:

– We have access to the code of the function call and there is no recursion If the graph of the called function is not yet built or the use-definition chains are not yet computed, then, since we have access to the code that will be executed in the function call, we do this analysis immediately. Once we have the PCFG and the use-definition information of the called graph, we propagate this information to the node containing the function call. During this propagation we must transform the usage computed in the called graph by usage meaningful for the current graph. This means that:

  I. The usage of the local variables of the called function is not propagated to the current graph.
  II. The usage of the parameters is renamed to the usage of the arguments.
  III. The usage of global variables is directly propagated to the current graph.

– We have access to the declaration of a function call and there is recursion In this case we cannot proceed in the same way as before because we would enter in an infinite loop analyzing the same function over and over again. To detect recursive calls we store in every graph the list of functions called in the graph. At the point were a recursive call appears we launch a new analysis that only deals with the variables which are relevant in the current analysis, which are pointed parameters (parameters with pointer type or parameters passed by reference) and global variables. We traverse the recursive function by computing the usage of these specific variables and then we propagate the information to the current graph as explained in the previous case (here we do not have local variables because no info is computed for them).

– We don't have access to the code of the called function In the case we cannot analyze the code of a function call we cannot define the usage over the global variables but still can define the usage of the arguments regarding the types of the declaration of the function. Only parameters passed by reference or parameters with pointer type will have an undefined behavior.

We introduce in Listing 5.5 a sample code that we will use as example for all the analysis. This code comes from the *floorplan* benchmark and it lays down a cell represented by an identifier (*id*) into a section of a board delimited by four points (*top*, *bot*, *lhs*, *rhs*). In Figure 5.6 we show the resultant PCFG with the information computed during the phase of Use-Definition chains.

```c
static int lay_down(int id, ibrd board, struct cell *cells) {
  int  i, j, top, bot, lhs, rhs;

  top = cells[id].top;
  bot = cells[id].bot;
  lhs = cells[id].lhs;
  rhs = cells[id].rhs;

  for (i = top; i <= bot; i++) {
    for (j = lhs; j <= rhs; j++) {
      if (board[i][j] == 0) board[i][j] = (char)id;
      else                  return(0);
    }
  }

  return (1);
}
```

Listing 5.5: Lay down method from Floorplan benchmark

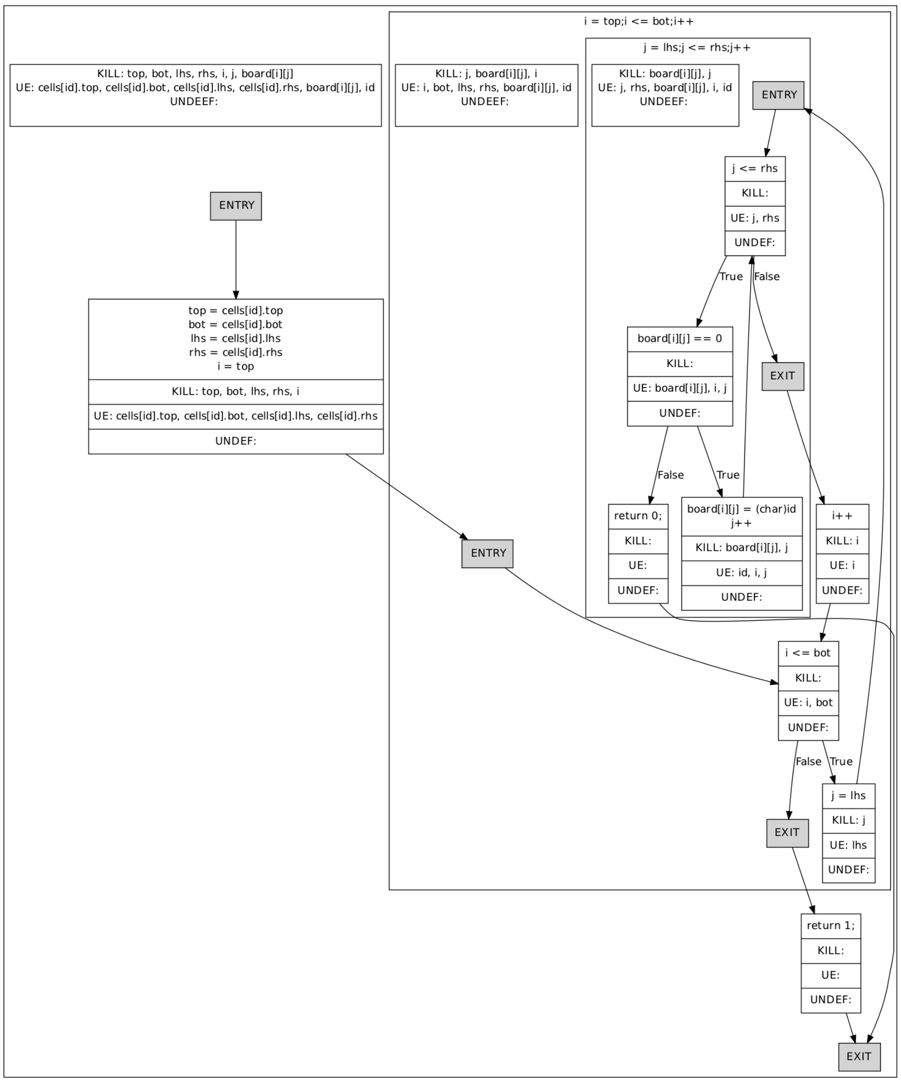


Figure 5.6: EG with Use-Define information for code in Listing 5.5

5.3 Loop analysis

The major data manipulated in scientific programs is the array. The use-define analysis on a whole array must take into account the existence of loops surrounding the access to an array. The analysis of arrays is costly in terms of computation and space storage so the methods used in every situation require a compromise between accuracy and complexity. Since we do not aim to implement aggressive optimizations such as auto-parallelization, we have defined a set of constraints that represent the frame we will use to apply our loop analysis.

What we want with this analysis is to determine which regions or elements of a given array are accessed in every code segment (where a code segment can be a basic block, a loop or a whole procedure). In order to do that we analyze the loops existing in the code. Specifically, we are interested in the analysis of **induction variables**. That is because they are frequently used as array subscripts and, in that case, we can define the rage of accesses to the array by defining the range of accesses to the induction variable. When arrays are accessed by constant values, the range access computation is trivial; in the case the arrays are accessed by non-induction variables, then we are not able to determine the range of accesses by analyzing the loop bounds and it becomes more difficult to discover which positions of the array are being accessed across the iteration space.

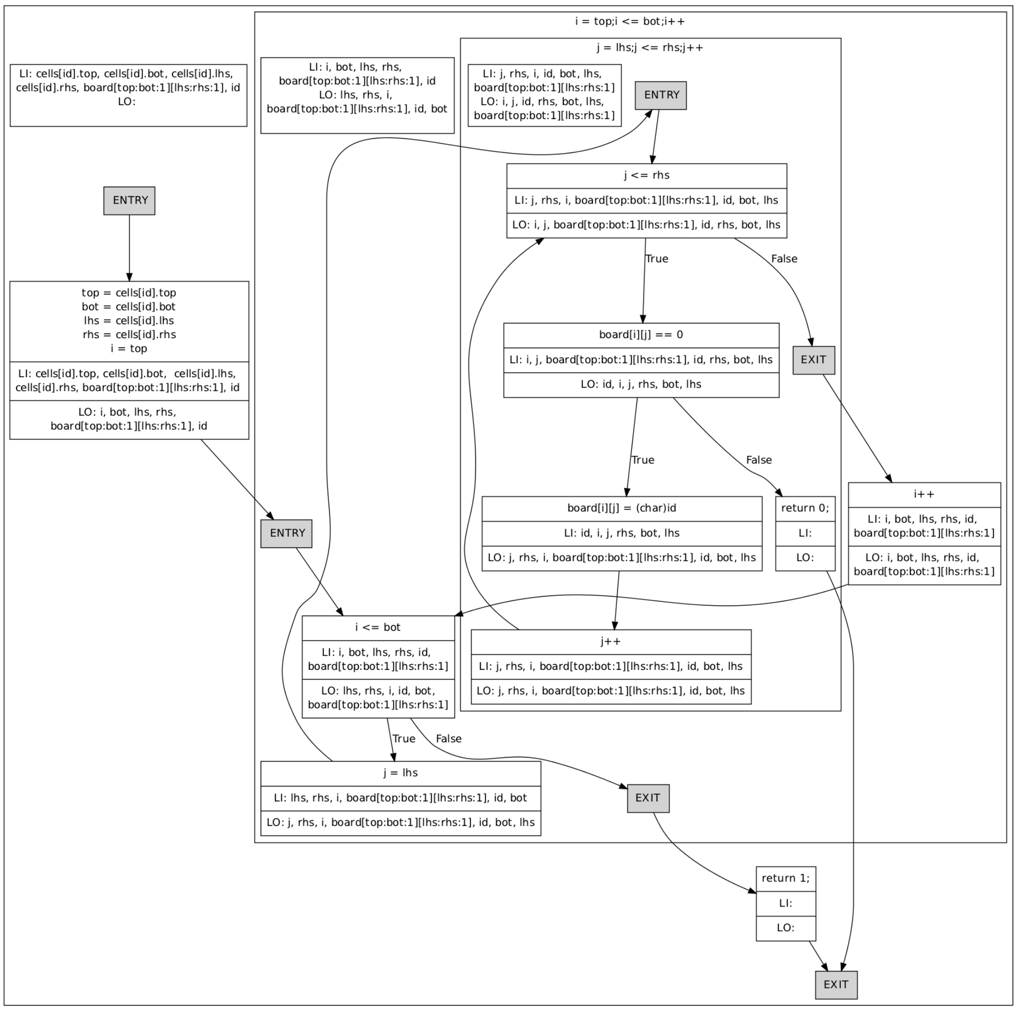
With our PCFG it is easy to determine when we are into a loop construct because we have represented the loop with specific nodes. The work here is to determine whether a variable is an induction variable or not. That requires the analysis of all the statements within the loop by searching variables that are increased or decreased by a fixed amount in every iteration or variables that are a linear function of another induction variable. We have decided to simplify this step and work only with *for-loop* constructs and induction variables that can be found in the control loop. We are missing many other cases like *while-loops* or *goto* control statements, but most of the codes appearing in our benchmarks fulfill our conditions.

In order to compute the induction variables and their ranges, we traverse the PCFG looking for *for-loop* nodes. The information about the induction variables is stored in a map structure as a 2-tuple of $< id, induc\_vars >$, where id is the identifier of the loop and $induc\_vars$ is the list of induction variables that fall within the scope of the loop id. Every variable is represented with the triplet notation $< LB, UB, S >$, where LB is the lower bound accessed by iv, UB is the upper bound accessed by iv and S is the stride used to increment or decrement iv in the loop. For every loop we found, we apply the following algorithm:

- If we are in a nested loop, we propagate the induction variables computed for our outer loop to the current loop storing this information in $induc\_var$.
- We traverse the nodes representing the loop control (initialization, condition and next) and we store in $induc\_var$ all the variables that are defined there. In this traverse we compute and we store in the structure the upper and lower bounds, and the stride of the defined variables. Not always we can compute these limits.
- All variables introduced in $induc\_var$ that have incomplete information are deleted from the structure.
- We traverse the inner statements of the loop by searching possibles redefinitions of the variables remaining in the structure. If some statement makes a variable to violate the induction variable conditions, then it is deleted from the structure.

Code in Listing 5.5 contains a 2 nested loops. If we apply loop analysis in this function we will compute two induction variables. In the outer loop we detect the induction variable i, represented by the triplet $[top; bot; 1]$. In the inner loop, we detect the induction variable j, represented by the triplet $[lhs; rhs; 1]$. In our records, we store the validity of i for the inner loop, and we do that by creating a virtual induction variable i' in the scope of the inner loop with the same attributes (symbol, i, and triplet, $[top; bot; 1]$) as the original induction variable of the outer loop.

Once we know the ranges of the induction variables, we modify the information computed during Use-Definition analysis to adapt the element information into range information. This work consists in traversing the graph looking for for-loop nodes. In a given loop, for every use of an induction variable as a subscript, we substitute the single values of the induction variable by the range values computed during the loop analysis. In Figure 5.7 we show the result of applying this transformation to the *lay down* example of Listing 5.5. The accesses to the matrix *board* that previously where single values, now have been transformed to ranges represented with the same triplets as the induction variables. Note here that the occurrences of the induction variable in situations different from an array subscript are not substituted. That is because they do not represent a set of memory units, but the change of value in a unique memory unit.



i = top;i <= bot;i++
j = lhs;j <= rhs;j++
LI: cells[id].top, cells[id].bot, cells[id].lhs, cells[id].rhs, board[top:bot:1][lhs:rhs:1], id
LO:
LI: i, bot, lhs, rhs, board[top:bot:1][lhs:rhs:1], id
LO: lhs, rhs, i, board[top:bot:1][lhs:rhs:1], id, bot
LI: j, rhs, i, id, bot, lhs, board[top:bot:1][lhs:rhs:1]
LO: i, j, id, rhs, bot, lhs, board[top:bot:1][lhs:rhs:1]
ENTRY
j <= rhs
LI: j, rhs, i, board[top:bot:1][lhs:rhs:1], id, bot, lhs
LO: i, j, board[top:bot:1][lhs:rhs:1], id, rhs, bot, lhs
True
False
ENTRY
top = cells[id].top
bot = cells[id].bot
lhs = cells[id].lhs
rhs = cells[id].rhs
i = top
LI: cells[id].top, cells[id].bot, cells[id].lhs, cells[id].rhs, board[top:bot:1][lhs:rhs:1], id
LO: i, bot, lhs, rhs, board[top:bot:1][lhs:rhs:1], id
board[i][j] == 0
LI: i, j, board[top:bot:1][lhs:rhs:1], id, rhs, bot, lhs
LO: id, i, j, rhs, bot, lhs
EXIT
True
False
board[i][j] = (char)id
LI: id, i, j, rhs, bot, lhs
LO: j, rhs, i, board[top:bot:1][lhs:rhs:1], id, bot, lhs
return 0;
LI:
LO:
i++
LI: i, bot, lhs, rhs, id, board[top:bot:1][lhs:rhs:1]
LO: i, bot, lhs, rhs, id, board[top:bot:1][lhs:rhs:1]
ENTRY
i <= bot
LI: i, bot, lhs, rhs, id, board[top:bot:1][lhs:rhs:1]
LO: lhs, rhs, i, id, bot, board[top:bot:1][lhs:rhs:1]
j++
LI: j, rhs, i, board[top:bot:1][lhs:rhs:1], id, bot, lhs
LO: j, rhs, i, board[top:bot:1][lhs:rhs:1], id, bot, lhs
True
False
j = lhs
LI: lhs, rhs, i, board[top:bot:1][lhs:rhs:1], id, bot
LO: j, rhs, i, board[top:bot:1][lhs:rhs:1], id, bot, lhs
EXIT
return 1;
LI:
LO:
EXIT

Figure 5.7: EG with Loop Analysis for code in Listing 5.5

5.4 Reaching definitions

We have implemented the reaching definitions data flow analysis in our PCFG. With the information computed during the Use-Definition analysis and the Loop Analysis, we are able to determine which definitions potentially reach any node in our graph. We need this information in order to analyze more accurately the values of some specific variables after a given iterative construct such as the induction variables and variables depending on induction variables like arrays.

We define the reaching definition set of a given node as the set of variables that reach the exit of the node. We will call this set *Reach_out*. The computation of this set is done traversing forward the graph; for **GRAPH** nodes we compute recursively the reaching definitions of the inner nodes and then we propagate this information to the outer node, which will have the same reaching definitions as its **EXIT** node. Once we have finished this computation, every node will have a new attribute containing the *Reach_out* set. When we are not capable to determine the value of a variable at a given node, then this variable will have an *UNKNOWN VALUE*.

For codes without any iterative construct nor with back edges, this analysis is as trivial as propagate forward the values defined at some point of the graph until the next definition of the same variable. Nonetheless, the existence of loops increases the difficulty of the analysis because it requires some arithmetic computation with the limits of the loops. Our purpose with this analysis is not to implement all cases supported by the C/C++ language, but mainly being able to determine the values of the induction variables when their bounds are simple expressions such as constants values, symbols or arithmetic functions of constants and symbols. We have implemented a calculator for constant expressions and a set of rules for algebraic simplification. These rules will help us to normalize the arithmetic expressions in the loop boundaries and simplify them in most cases. In Figure 5.8 we show the set of rules that

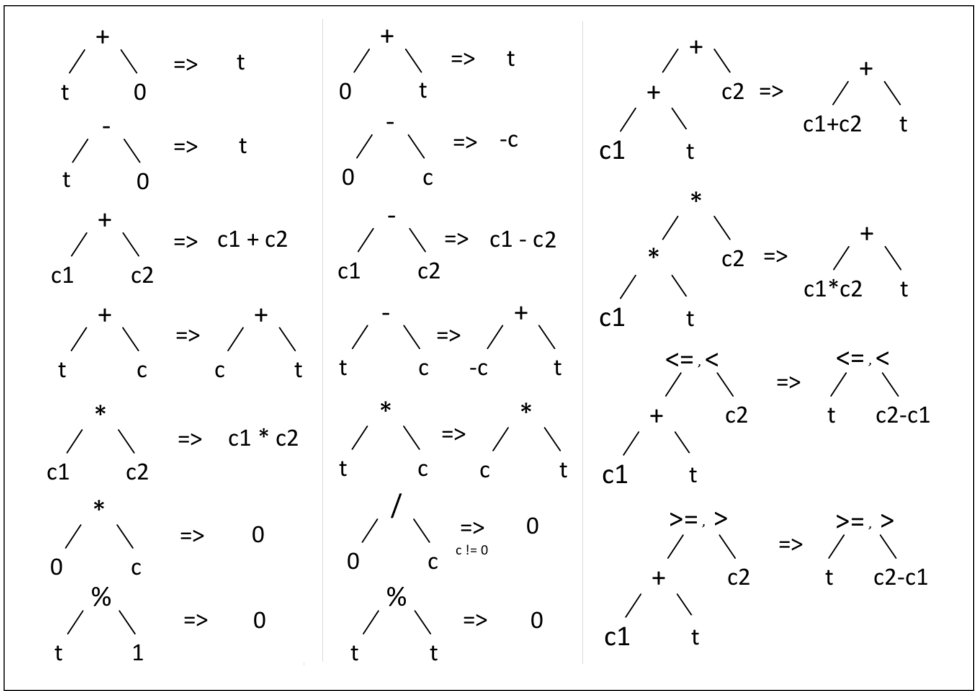


Figure 5.8: Arithmetic simplifications

we have implemented to simplify arithmetic expressions. Understand c, $c1$ and $c2$ as constant values, and t as the tree of a expression (a *Nodecl*). The left part of the implication is the input expression and the right and is the output expression. For example, for the first rule (top left of the figure), whenever we find a *Nodecl* of type *Addition* where the left hand side of the addition is an unrestricted variable and the right hand side of the addition is a constant value equal to zero, then we can substitute this expression by an expression equal to the tree on the left hand side of the assignment.

Using once again the example introduced in Listing 5.5, we show in Figure 5.9 the values of the *Reach_out* set for every node. Note the results of applying the rules introduced in the previous paragraph. For example, focusing in the outer loop, the values of the induction variable i are different depending on the node we look at:

- In the node containing the condition of the loop, i takes values in the range $top : 1 + bot : 1$.
- In the node containing the stride, i takes values in the range $1 + top : 1 + bot : 1$.
- In any other node within the loop, i takes values int he range $top : bot : 1$.
- The value of i after the execution of the loop is $1 + bot$.

Focusing now in the inner loop, we compute the value of j as unknown at the exit of the **GRAPH** node because not all branches inside the loop have the same value of j. However, since j is a range of values inside the loop, we can compute the ranges in every node using the same technique as used for the variable i.

5.5 Liveness analysis

Liveness analysis is a data flow analysis that computes for each program point the variables that may be potentially read before their next write. It is that a variable is live if it holds a value that may be needed in the future.

For this analysis we need the information computed in the previous analysis. We use the commonly used data-flow equations for defining the variables that are live at the entry (*Live_in*) and at the exit (*Live_out*) of every node in the graph. So, given a node X, the set of upper exposed variables in the node, $UE(X)$, and the set of killed variables in the node, $KILL(X)$, the equations are the following:

$$Live\_out(X) = \bigcup_{Y \in Succ(X)} Live\_in(Y) \quad \text{(}Succ(X)\text{ are all nodes reachable from }X\text{)}$$

$$Live\_in(X) = UE(X) + (Live\_out(X) - KILL(X))$$

These equations are applied backwards from the **EXIT** node up to the **ENTRY** node of the PCFG. This traversal is embedded in a loop iteration that stops

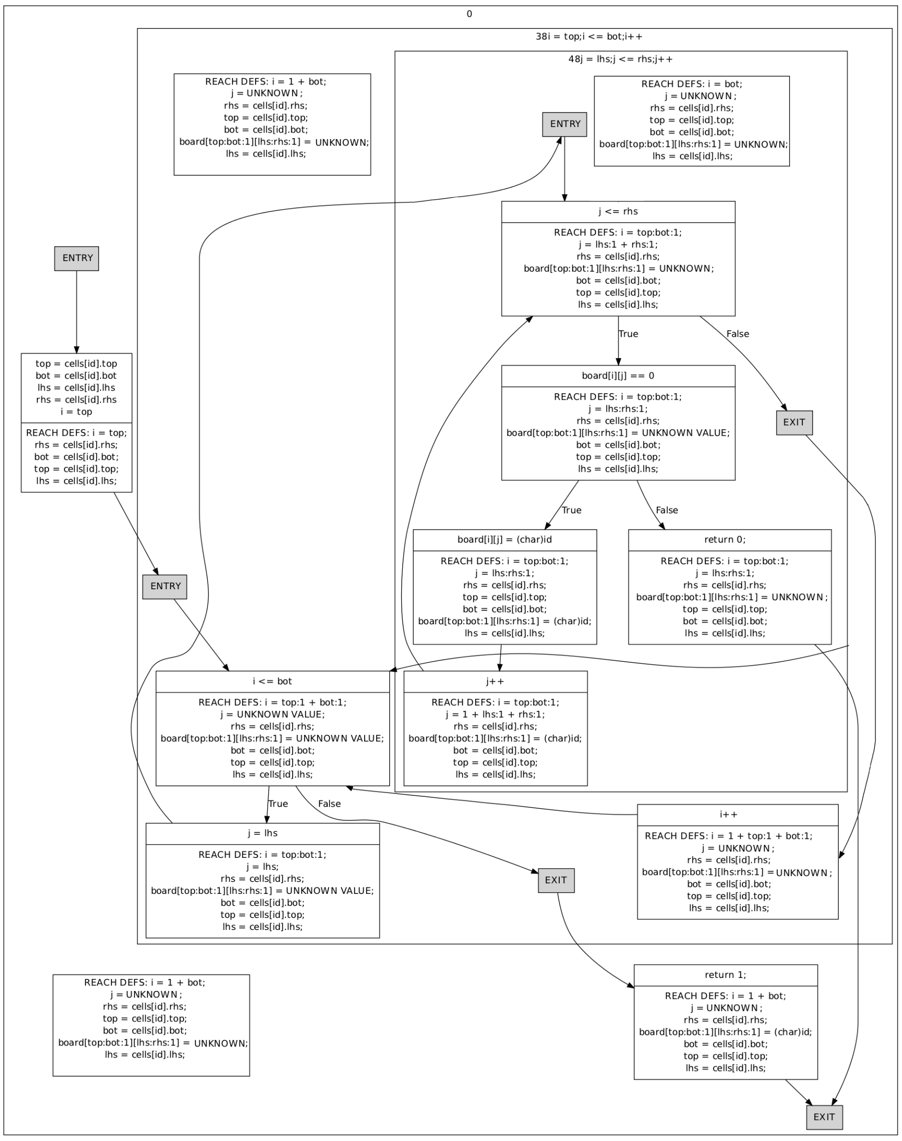

0
38i = top;i <= bot;i++
48j = lhs;j <= rhs;j++
REACH DEFS: i = 1 + bot; j = UNKNOWN ; rhs = cells[id].rhs; top = cells[id].top; bot = cells[id].bot; board[top:bot:1][lhs:rhs:1] = UNKNOWN; lhs = cells[id].lhs;
ENTRY
REACH DEFS: i = bot; j = UNKNOWN ; rhs = cells[id].rhs; top = cells[id].top; bot = cells[id].bot; board[top:bot:1][lhs:rhs:1] = UNKNOWN; lhs = cells[id].lhs;
j <= rhs
REACH DEFS: i = top:bot:1; j = lhs:1 + rhs:1; rhs = cells[id].rhs; board[top:bot:1][lhs:rhs:1] = UNKNOWN; bot = cells[id].bot; top = cells[id].top; lhs = cells[id].lhs;
ENTRY
True
False
top = cells[id].top bot = cells[id].bot lhs = cells[id].lhs rhs = cells[id].rhs i = top
REACH DEFS: i = top; rhs = cells[id].rhs; bot = cells[id].bot; top = cells[id].top; lhs = cells[id].lhs;
board[i][j] == 0
REACH DEFS: i = top:bot:1; j = lhs:rhs:1; rhs = cells[id].rhs; board[top:bot:1][lhs:rhs:1] = UNKNOWN VALUE; bot = cells[id].bot; top = cells[id].top; lhs = cells[id].lhs;
EXIT
True
False
board[i][j] = (char)id
REACH DEFS: i = top:bot:1; j = lhs:rhs:1; rhs = cells[id].rhs; top = cells[id].top; bot = cells[id].bot; board[top:bot:1][lhs:rhs:1] = (char)id; lhs = cells[id].lhs;
return 0;
REACH DEFS: i = top:bot:1; j = lhs:rhs:1; rhs = cells[id].rhs; board[top:bot:1][lhs:rhs:1] = UNKNOWN ; top = cells[id].top; bot = cells[id].bot; lhs = cells[id].lhs;
ENTRY
i <= bot
REACH DEFS: i = top:1 + bot:1; j = UNKNOWN VALUE; rhs = cells[id].rhs; board[top:bot:1][lhs:rhs:1] = UNKNOWN VALUE; bot = cells[id].bot; top = cells[id].top; lhs = cells[id].lhs;
j++
REACH DEFS: i = top:bot:1; j = 1 + lhs:1 + rhs:1; rhs = cells[id].rhs; board[top:bot:1][lhs:rhs:1] = (char)id; bot = cells[id].bot; top = cells[id].top; lhs = cells[id].lhs;
True
False
j = lhs
REACH DEFS: i = top:bot:1; j = lhs; rhs = cells[id].rhs; board[top:bot:1][lhs:rhs:1] = UNKNOWN VALUE; bot = cells[id].bot; top = cells[id].top; lhs = cells[id].lhs;
EXIT
i++
REACH DEFS: i = 1 + top:1 + bot:1; j = UNKNOWN ; rhs = cells[id].rhs; board[top:bot:1][lhs:rhs:1] = UNKNOWN ; bot = cells[id].bot; top = cells[id].top; lhs = cells[id].lhs;
REACH DEFS: i = 1 + bot; j = UNKNOWN ; rhs = cells[id].rhs; top = cells[id].top; bot = cells[id].bot; board[top:bot:1][lhs:rhs:1] = UNKNOWN; lhs = cells[id].lhs;
return 1;
REACH DEFS: i = 1 + bot; j = UNKNOWN ; rhs = cells[id].rhs; board[top:bot:1][lhs:rhs:1] = (char)id; bot = cells[id].bot; top = cells[id].top; lhs = cells[id].lhs;
EXIT

Figure 5.9: EG with Reaching Definitions for code in Listing 5.5

when, after the last two iterations, the liveness information has not changed in any node. In the case of graph nodes, the backward traversal is applied from its **exit** node until its **entry** node; then, its *Live_in* information is obtained from its **entry** node and its *Live_out* information is obtained from its **exit** node.

In order to properly keep OpenMP tasks liveness information, we have to do some extra work. For tasks appearing within loop iterations, computing *Live_out* as we do for the rest of nodes is not enough because variables within the tasks can be used in the task instance of the following iteration. To compute these special variables, we virtually add the task as a child of itself. With virtually we mean that no physical edge is added to the task, but we add to the task *Live_out* set all these variables that are in the task *Live_in* set.

We add these new analysis results to every node in the form of two new attributes:

- Live in: set of variables computed as live at the entry of the node.
- Live out: set of variables computed as live at the exit of the node.

Following with the example introduced in the previous section, the *lay down* method from *floorplan* benchmark (Listing 5.5), we show in Figure 5.10 the same graph but now with the information computed during the liveness analysis.

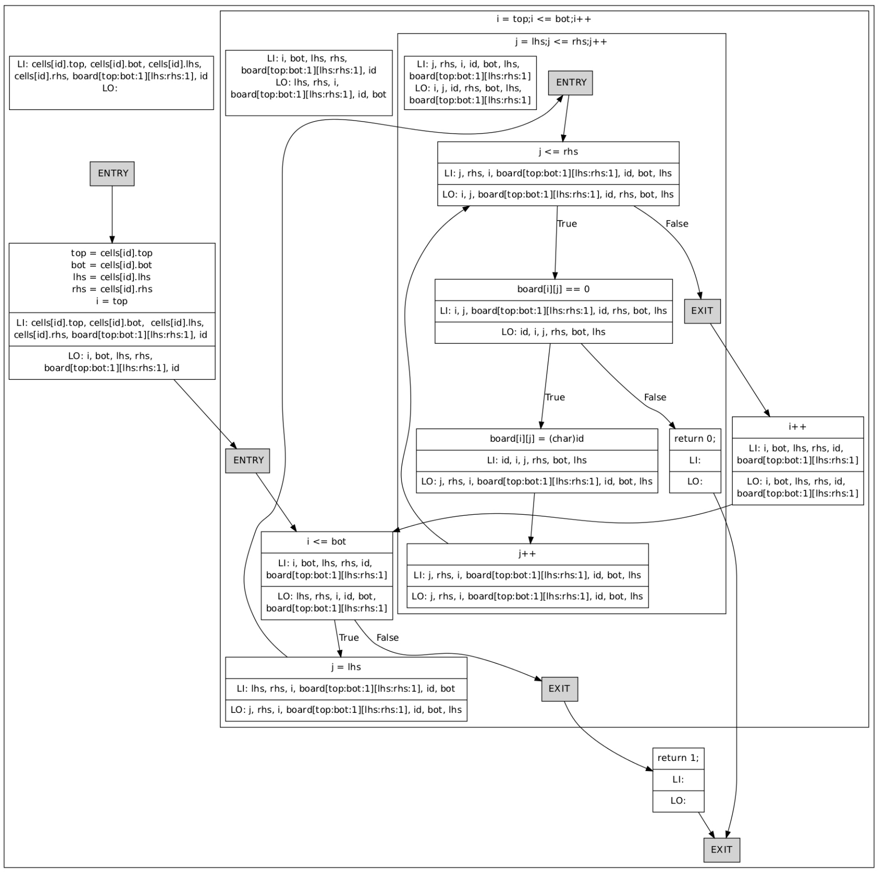

LI: cells[id].top, cells[id].bot, cells[id].lhs, cells[id].rhs, board[top:bot:1][lhs:rhs:1], id
LO:
ENTRY
top = cells[id].top
bot = cells[id].bot
lhs = cells[id].lhs
rhs = cells[id].rhs
i = top
LI: cells[id].top, cells[id].bot, cells[id].lhs, cells[id].rhs, board[top:bot:1][lhs:rhs:1], id
LO: i, bot, lhs, rhs, board[top:bot:1][lhs:rhs:1], id
i = top;i <= bot;i++
LI: i, bot, lhs, rhs, board[top:bot:1][lhs:rhs:1], id
LO: lhs, rhs, i, board[top:bot:1][lhs:rhs:1], id, bot
ENTRY
i <= bot
LI: i, bot, lhs, rhs, id, board[top:bot:1][lhs:rhs:1]
LO: lhs, rhs, i, id, bot, board[top:bot:1][lhs:rhs:1]
True
False
j = lhs
LI: lhs, rhs, i, board[top:bot:1][lhs:rhs:1], id, bot
LO: j, rhs, i, board[top:bot:1][lhs:rhs:1], id, bot, lhs
EXIT
j = lhs;j <= rhs;j++
LI: j, rhs, i, id, bot, lhs, board[top:bot:1][lhs:rhs:1]
LO: i, j, id, rhs, bot, lhs, board[top:bot:1][lhs:rhs:1]
ENTRY
j <= rhs
LI: j, rhs, i, board[top:bot:1][lhs:rhs:1], id, bot, lhs
LO: i, j, board[top:bot:1][lhs:rhs:1], id, rhs, bot, lhs
True
False
board[i][j] == 0
LI: i, j, board[top:bot:1][lhs:rhs:1], id, rhs, bot, lhs
LO: id, i, j, rhs, bot, lhs
EXIT
True
False
board[i][j] = (char)id
LI: id, i, j, rhs, bot, lhs
LO: j, rhs, i, board[top:bot:1][lhs:rhs:1], id, bot, lhs
return 0;
LI:
LO:
j++
LI: j, rhs, i, board[top:bot:1][lhs:rhs:1], id, bot, lhs
LO: j, rhs, i, board[top:bot:1][lhs:rhs:1], id, bot, lhs
i++
LI: i, bot, lhs, rhs, id, board[top:bot:1][lhs:rhs:1]
LO: i, bot, lhs, rhs, id, board[top:bot:1][lhs:rhs:1]
return 1;
LI:
LO:
EXIT

FIGURE 5.10: EG with liveness information for code in Listing 5.5

Chapter 6. *OmpSs optimizations*

This section presents three optimizations that we have undertaken to exploit the benefits of the analyses described in Chapter 5. As we introduced in the early chapters of this dissertation, our goal when we decided to implement analysis in Mercurium compiler was to improve the productivity of OmpSs. Each one of the optimizations we have implemented takes one direction in order to achieve this objective. In the first case, privatizing shared variables we can achieve better performance in codes that have many accesses to these variables. In the second case, automatically discovering task dependencies does not produce a better performance but it enhances the programmability of OmpSs releasing the user from the task of doing this job manually. We present the details of each approach and the results we have obtained for a set of benchmarks.

6.1 Privatization: optimizing shared variables

6.1.1 Scope of the optimization

The OpenMP model defines the existence of two different contexts for variables living in parallel environments: private and shared. Variables in a **private context** are hidden from other threads; this means that each thread has its own private copy of the variable and modifications made by a thread are not visible to other threads. On the contrary, variables in a **shared context** are visible to all threads running in associated work teams.

In OpenMP each directive is associated to a structured block that defines a new scope (in the case of directives in declaration level, this block will be the code executed in a given call to the declared function). Each variable referenced in the structured block has an original variable existing in the program immediately outside the construct. Each access to a shared variable in the structured block becomes a reference to the original variable. For each private variable referenced in the structured block, a new version of the original variable (of the same type and size) is created in memory for each task that contains code associated with the directive.

As it is explained in Section 4.1, OpenMP defines for shared variables a relaxed memory model where threads may have regions where they define their own temporary view of the memory. Threads temporary view is not required to be consistent with main memory at every moment of execution. OpenMP guarantees consistency across the local memories and the main memory by the flush operation. The completion of a flush executed by a thread is defined as the point at which all the variables involve are synchronized with main memory. A memory flush operation can be performed in two different ways:

– OpenMP provides a flush directive with the following syntax:

  ```
  #pragma omp flush [(list)]
  ```

  where list specifies the set of items to which the flush is applied on.
– OpenMP implicitly performs memory flushes in the following situations:

  ∗ During a barrier region.
  ∗ At entry to and at exit from `parallel`, `critical` and `ordered` regions.
  ∗ At exit from worksharing regions unless a `nowait` clause is present.
  ∗ A entry to and at exit from combined parallel worksharing regions.
  ∗ During `omp_set_lock` and `omp_unset_lock` regions.
  ∗ During `omp_test_lock`, `omp_set_nest_lock`, `omp_unset_nest_lock` and `omp_test_nest_lock` regions, if the region causes the lock to be set or unset.
  ∗ Immediately before and immediately after every task scheduling point.
  ∗ A flush region with a list is implied at the entry to and at the exit from `atomic` regions, where the list contains only the variable updated in the `atomic` construct.

Regarding to a shared variable, a **flush region** ensures the following statements:

– At the beginning of the region, a flush enforces the value of the variable to be consistent in main memory and all the local views.
– At the end of the region, a flush enforces the value of the variable to be synchronized across the memories.

Shared variables are represented in the Mercurium compiler as pointers to the original memory locations. In Listing 6.1 we show a matrix multiplication code parallelized with the OpenMP `parallel` construct. In Listing 6.2 we show a snippet of code generated by Mercurium. This code contains the outlined function called *_smp__ol_matmul_0* corresponding to the block of code embedded in a parallel region in the original code. The original method *matmul* has been transformed into a set of instructions allowing the communication with the Nanos++ runtime library. We show only the lines that are useful for our purpose, which are the creation of the data structure with *_nx_data_env_0_t_tag* called *ol_args* that contains the parameters needed for the execution of the outlined function and the different calls to the Nanos++ specific functions allowing the creation of parallelism. Conservatively, the compiler uses shared pointers to the matrices. While in the original code, the matrix multiplication is done over the matrices *A*, *B* and *C*, in the outlined parallel version, every access to the matrix is done by referencing the access through a shared pointer. The **overhead** paid for using shared variables is proportional to the number of accesses to this variable. In this case, it supposes doing three extra references for each iteration of the three loops.

```
int MATSIZE = 0;
void matmul ( double *A , double *B, double *C) {
  int i , j , k;

#pragma omp parallel for private (i, j, k)
  for ( i = 0 ; i < MATSIZE; i++)
    for ( j = 0 ; j < MATSIZE; j++)
      for ( k = 0 ; k < MATSIZE; k++)
        C [ i ][ j ] += A [ i ][ k ] * B[ k ][ j ] ;
}
```

Listing 6.1: Matrix multiply with OpenMP parallel

The motivation of privatizing shared variables comes from the fact, confirmed in the previous example, that using shared variables when it is not indispensable to do so, introduces unnecessary overheads without bringing any benefit. However, incorrectly privatizing a variable may result in an undefined value for the variable outside the construct. The key is to determine when a shared variable can be privatized. We can take advantage of the characteristics of the OpenMP memory model and the flush operations described previously to state the following **privatization criterion**:

> "If we are capable of guaranteeing that there is a region where no flushes are performed, then, each thread can privatize a shared variable in that region."

By the methodology we use during the construction of the PCFG, all flush operations are made explicit in the graph. In addition, to ensure the correctness of the OpenMP memory model, we suppose flush operations at the entry and at the exit of every function call and we add marks in the code indicating this assumption. With this information we can define which are the regions where no flush are performed besides the flush at the entry to the region and the flush at the exit from the region. We call these regions **no-flush regions**. Now we are able to use the results of the **liveness analysis** to know which variables are live at any point of the flow and combine this data to decide which privatized shared variables within a no-flush region need to be initialized with the content of the main memory. The liveness analysis also give us information about the privatized variables that need to be flushed at the end of the no-flush region. It is important to remark that we are applying this optimization in a high-level representation of the program, when the code is not yet specialized for a particular architecture.

```c
int MATSIZE = 1000;
typedef struct _nx_data_env_0_t_tag {
  nanos_loop_info_t loop_info;
  int *MATSIZE_0;
  double ***A_0;
  double ***B_0;
  double ***C_0;
} _nx_data_env_0_t;

static void _smp__ol_matmul_0(_nx_data_env_0_t *const __restrict__ _args) {
  int i, j, k;
  int *MATSIZE_0 = (int *) (_args->MATSIZE_0);
  double ***A_0 = (double ***) (_args->A_0);
  double ***B_0 = (double ***) (_args->B_0);
  double ***C_0 = (double ***) (_args->C_0);
  int _nth_lower = _args->loop_info.lower;
  int _nth_upper = _args->loop_info.upper;
  int _nth_step = _args->loop_info.step;

  for (i = _nth_lower; i <= _nth_upper; i += _nth_step)
    for (j = 0; j < (*MATSIZE_0); j++)
      for (k = 0; k < (*MATSIZE_0); k++)
        (*C_0)[i][j] += (*A_0)[i][k] * (*B_0)[k][j];
}

void matmul(double **A, double **B, double **C) {
  ...
  _nx_data_env_0_t *ol_args = (_nx_data_env_0_t *) 0;
  ...
  err = nanos_create_sliced_wd(&wd, 1, _ol_matmul_0_devices, sizeof(_nx_data_env_0_t),
      __alignof__(_nx_data_env_0_t), (void **) &ol_args, nanos_current_wd(), static_for,
      sizeof(nanos_slicer_data_for_t), __alignof__(nanos_slicer_data_for_t),
      (nanos_slicer_t *) &slicer_data_for, &props, 0, (nanos_copy_data_t **) 0);
  ...
  ol_args->MATSIZE_0 = &(MATSIZE);
  ol_args->A_0 = &(A);
  ol_args->B_0 = &(B);
  ol_args->C_0 = &(C);
  slicer_data_for->_lower = 0;
  slicer_data_for->_upper = (MATSIZE) - 1;
  slicer_data_for->_step = 1;
  slicer_data_for->_chunk = 0;
  err = nanos_submit(wd, 0, (nanos_dependence_t *) 0, (nanos_team_t) 0);
  ...
  }
  nanos_omp_barrier();
}
```

Listing 6.2: Mercurium outline for code in Listing 6.1

We have implemented the analysis for these cases where the shared variables are not used beyond the point of the privatization. A possible extension of this optimization consists on the research of portions of code were it is possible the privatization of a variable that must be flushed afterwards into the shared memory. In this case, we should define a trade-off between the cost of privatizing and flushing a variable and the cost of the shared access to the variable, which will depend on the number of accesses performed to the shared value.

6.1.2 The results

For this test we have used a machine with 24 Intel Xeon E7450 x86-64 processors of 2.50GHz machine with SUSE Linux. In order to test the opportunities of the shared variables privatization we have chosen a set of different benchmarks containing a great number of shared variables accesses. We have used the Mercurium compiler and GCC as the back-end compiler and for all the executions, both the original and the optimized versions, with have used -O3 level of optimization.

6.1.2.1 Matrix multiplication

In Listings 6.1 and 6.2 we have introduced respectively the code of a matrix multiplication algorithm parallelized with the OpenMP `parallel` construct and the output originally generated by Mercurium for this input code. In Listing 6.3 we present the meaningful parts of the optimized version of the translation generated by Mercurium once the privatization has been implemented. During the analysis, the compiler detects that the access to the three matrices A, B and C can be privatized, as well as the access to the global variable $MATSIZE$. It makes that decision because all these variables are not live after the call to the *_smp__ol_matmul_0* method. Therefor, in the optimized version, the data structure *_nx_data_env_0_t_tag* is created with the private versions of the variables. Thus, we have avoided three extra references for each iteration of the three loops.

Matrix multiply performance is highly dependent from the micro-architecture because of the repeated number of accesses to the same of contiguous memory positions. For our optimization in particular, as big is the matrix, as much benefit we expect to obtain from the privatization. But the size consequences are sensible to other aspects like cache conflicts, access to slower levels than L1 in the cache hierarchy, etcetera. In order to avoid this kind of interferences in our results, we have chosen a matrix size, 2KB x 2KB, which perfectly fits in the L1 cache, 12MB. We have tested the original translation and the optimized translation for both serial and parallel codes from 1 thread up to 16 threads.

```
typedef struct _nx_data_env_0_t_tag {
  nanos_loop_info_t loop_info;
  int MATSIZE_0;
  double **A_0;
  double **B_0;
  double **C_0;
} _nx_data_env_0_t;

static void _smp__ol_matmul_0(_nx_data_env_0_t *const __restrict__ _args) {
  int MATSIZE_0 = (int ) (_args->MATSIZE_0);
  double **A_0 = (double **) (_args->A_0);
  double **B_0 = (double **) (_args->B_0);
  double **C_0 = (double **) (_args->C_0);
  ...
  for (i = _nth_lower; i <= _nth_upper; i += _nth_step)
    for (j = 0; j < MATSIZE_0; j++)
      for (k = 0; k < MATSIZE_0; k++)
        C_0[i][j] += A_0[i][k] * B_0[k][j];
}

void matmul(double **A, double **B, double **C) {
  ...
  ol_args->MATSIZE_0 = MATSIZE;
  ol_args->A_0 = A;
  ol_args->B_0 = B;
  ol_args->C_0 = C;
  ...
}
```

Listing 6.3: Mercurium optimized outline for code in Listing 6.1

In Figure 6.1 we show the execution time comparison among the different executions. As we expected, the optimized version reduces the execution time against the base version; that is because of the reduction of memory access due to the use of non-shared variables. In Figure 6.2 we can observe the perfect scalability we achieve with the optimized version, although the speed-up obtained with the unoptimized version is close to being perfect. In this chart we show the gain with the optimized version in relation to the original version as well. It is close to the 1% an it is stable while we scale the application.

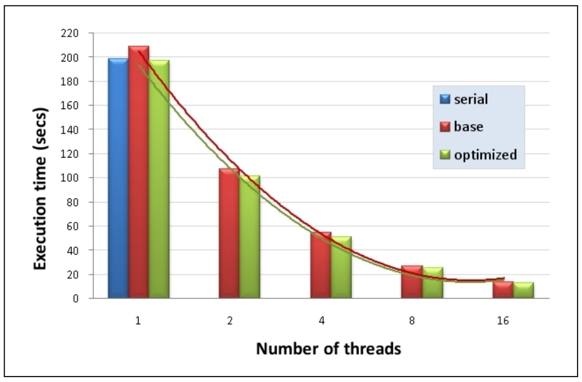


Figure 6.1: Matrix multiply execution time

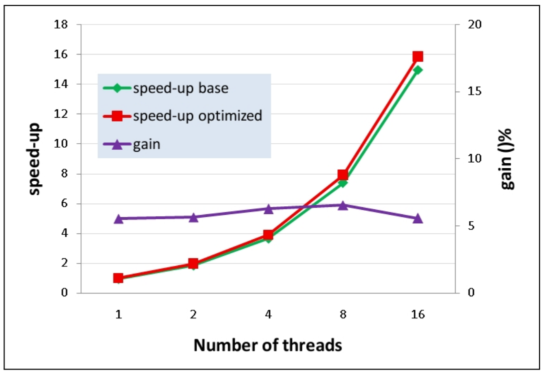


Figure 6.2: Matrix multiply speed-up & gain

6.1.2.2 Jacobi

The 2D Jacobi iteration is an stencil algorithm that computes the arithmetic mean of a cell's four neighbors, as it is showed in Figure 6.3. We have used the parallel version of this algorithm implemented by using the OpenMP `parallel` construct that is showed in Listing 6.4. The most outer loop repeats the computation of the Jacobi iteration in a 2D matrix and it is parallelized among the threads in the current team. The two nested inner loops implement the Jacobi iteration are private for each one of the threads.

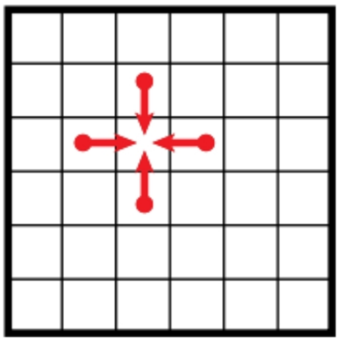


FIGURE 6.3: Jacobi dependencies

```
static void jacobi(float ** a, float ** b) {
#pragma omp parallel
{
    for (int iter = 0; iter < ITERS; iter++) {
      int i, j;
#pragma omp for private(i,j)
      for (i = 1; i < N - 1; i++)
        for (j = 1; j < N - 1; j++)
          b[i][j] = 0.25 * (a[i-1][j] + a[i+1][j] + a[i][j-1] + a[i][j+1]);
      swap(a, b);
    }
  }
}
```

LISTING 6.4: Jacobi iteration with OpenMP parallel

In the same line as with the matrix multiply, we have obtained an improvement in the performance of the optimized version. In Figure 6.4 we show the differences in the execution time for different executions of the two versions both serial and parallel from 1 up to 24 threads and as it happened in the previous example, we reduce the execution time with the optimization. But the trend is to equalize the time of the non-optimized version while we increase the number of threads. Regarding on Figure 6.5, we can observe that the application is far from obtaining a perfect speed-up as we scale the number of threads. That fact occurs most specially from 16 threads forth. The reason of this impasse is that the application reaches the limits of the memory bandwidth of the machine.

6.1.2.3 Vector scan

Vector scan is an approach to parallelize a computation in a vector which a priori is not parallel. The input of the algorithm is a vector of size N and the output is a vector of size N where each position $p_{i,i\ \in[0..N-1]}$ is the summation

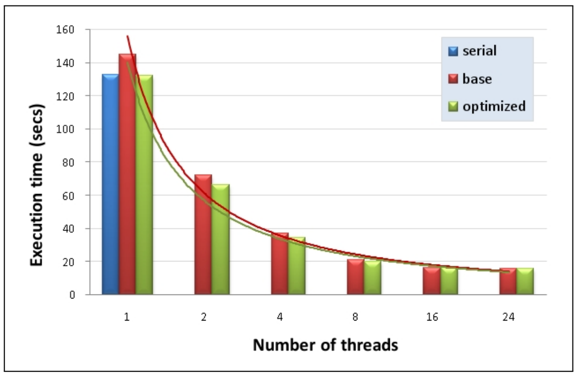


Figure 6.4: Jacobi execution time

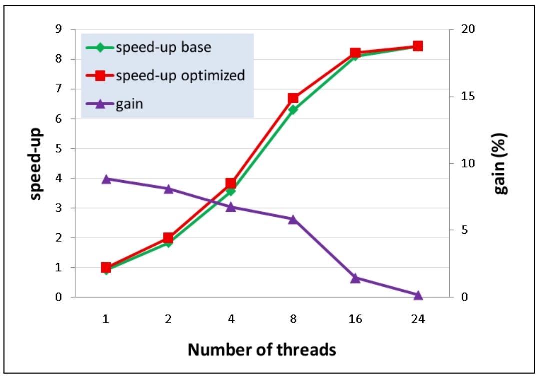


Figure 6.5: Jacobi speed-up & gain

of all elements $p_{j,j} \in [0..i-1]$. The code of the parallel version is shown in Listing 6.5.

```
void scan ( elem_t* output , elem_t* input , int n)
{
  int log2n = log(n) / log(2);
  int d, k;

#pragma omp for
  for ( d = 0; d < n; d++ )
     output[d] = input[d];

  for ( d = 0 ; d < log2n ; d++ ) {
    int s = 1 << (d+1);
    int s2 = 1 << (d);
#pragma omp for firstprivate(s,s2)
    for ( k = 0; k < n; k += s )
       output[k+s-1] += output[k+s2-1];
  }

  output[n - 1] = 0;
  for ( d = log2n - 1 ; d >= 0; d-- ) {
    int s = 1 << (d+1);
    int s2 = 1 << (d);
#pragma omp for firstprivate(s,s2)
    for ( k = 0; k < n; k += s ) {
       elem_t t = output[k+s2-1];
       output[k+s2-1] = output[k+s-1];
       output[k+s-1] = output[k+s-1] + t;
    }
  }

  #pragma omp taskwait
}
```

Listing 6.5: Vector scan computation with OpenMP parallel

We have tested the performance and scalability of the base and the optimized version as we did with the other examples. The performance results are shown in Figure 6.6. We can observe the reduction of the execution time obtained by the optimized version against the base version. In Figure 6.7 we show the scalability and the gain. The values bellow one in the speed-up obtained before the barrier of the 8 threads are due to the fact that the parallelization of this code requires the magnitude in the array and the number of threads executing in parallel to be large in order to perceive the profit of the parallelization. It is not the case of the executions between 1 and 4 threads of the two parallel versions, which work out to be slower than the serial version. Regarding on the gain, we can see that, when the computation time is dominated by the memory accesses (between 1 and 4 threads), the optimization results in almost a gain of the ten percent. From 8 threads and forth, the benefits of the optimization are hide by the benefits of the parallelization.

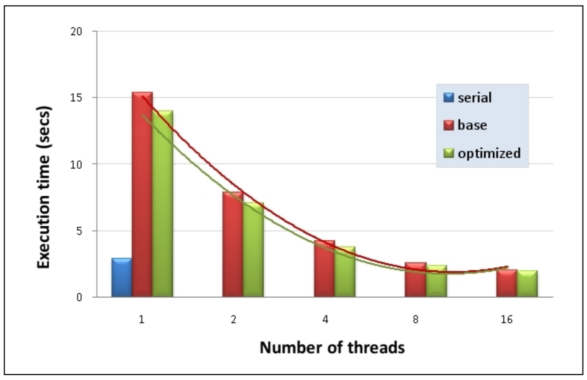


FIGURE 6.6: Scan execution time comparison

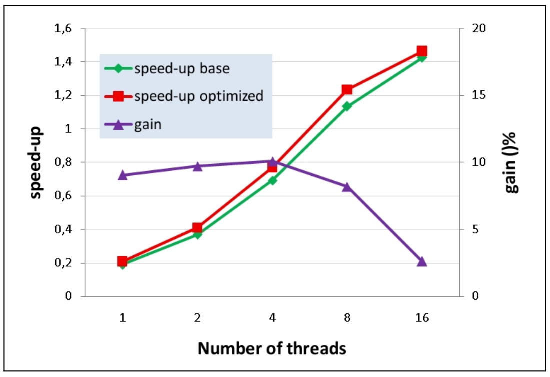


FIGURE 6.7: Scan speed-up comparison

6.2 Automatic scoping of variables in tasks

6.2.1 Scope of the optimization

The process of automatically defining the scope of the variables in a parallel region is tedious and error-prone. We can substantially improve the productivity of our programming model leaving this responsibility to the compiler. Lin et al. [LTaMC04] proposed a solution for the auto-scoping problem within synchronous parallel regions in OpenMP. They defined a set of rules that, applied to the variables appearing in the parallel region, allow determining the proper scope of the variables. They define four possible values, which are `PRIVATE`, `LASTPRIVATE`, `REDUCTION` and `SHARED`. When the compiler is not able to decide the scope of a variable, then the variable is not auto-scoped.

Based on this work, we have defined an algorithm that solves the scoping problem in presence of asynchronous parallelism (i.e. OpenMP tasks). The uncertainty and the semantics introduced by OpenMP tasks requires different rules that the ones applied when the parallelism is synchronous. The basic differences we have taken into account to develop our method are:

- The uncertainty about the exact moment when the task will be executed against the determinism of the synchronous parallel regions forces us to define the correct region of code where a data race can appear, while for parallel regions this region is perfectly defined by the `parallel` construct. This code should be not sequential, since tasks scheduled in different and non-contiguous points of the code can be executed in parallel.
- OpenMP task construct does not admit `lastprivate` and `reduction` clauses.
- `PRIVATE` variables can be specialized into `FIRSTPRIVATE` variables when the input value is used.

Taking into consideration these facts, we have defined the following methodology for determining the auto-scoping of the variables within a `task` construct:

1. Traverse the PCFG looking for task nodes. Given a task t and its *scheduling* point:
2. Determine the different regions that interfere in the analysis of t:
    - One region is the one defined by the code in the encountered thread that can potentially be executed in parallel with the task. This region is defined by two points:
        * *Scheduling*: is the point where the task is scheduled. Any previous access by the encountering thread to a variable appearing in the task is irrelevant when analyzing the task because it is already executed.
        * *Next_sync*: is the point where the task is synchronized with the rest of the threads in execution. This point can only be a `barrier` or a `taskwait`. Here we take into account that `taskwait` constructs only enforces the synchronization of tasks that are children of the current task region.
    - Other regions are the ones enclosed in tasks that can be executed in parallel with t. We will call these tasks $t_{i,\ i \in [0..T]}$ and the region of code where we can find tasks in this condition is defined by:
        * *Last_sync*: is the immediately previous point to the *scheduling* point where a synchronization enforces all previous executions to be synchronized. We can only assure this point with a `barrier` and in specific cases with a `taskwait`. We only can trust the `taskwait` if we know all the code executed previously and we can assure that the current task region has not generated grandchild tasks.

∗ *Next_sync*: is the same point as explained for the analysis of the encountered thread.

In order to simplify the reading of the algorithm bellow, from now on we will talk about the region defined between the *scheduling* point and the *next_sync* point and the different regions defined by the tasks $t_{i,\ i \in [0..T]}$ as one unique region defined by the points:

– *init*, referencing both *scheduling* and any *entry* point to the tasks $t_{i,\ i \in [0..T]}$.

– *end*, referencing both *next_sync* and any *exit* point to the tasks $t_{i,\ i \in [0..T]}$.

3. For each v scalar variable appearing within the task t:

   (a) If we cannot determine the type of access (read or write) performed over v either within the task or between *init* and *end* because the variable appears as a parameter in a function call that we do not have access to, then v is scoped as `UNDEFINED`.

   (b) If v is not used between *init* and *end*, then:

      i. If v is only read within the task, then v is scoped as `FIRSTPRIVATE`.

      ii. If v is written within the task, then:

         A. If v is live after *end*, then v is scoped as `SHARED`.

         B. If v is dead after *end*, then:

            – If the first action performed in v is a write, then v is scoped as `PRIVATE`.

            – If the first action performed in v is a read, then v is scoped as `FIRSTPRIVATE`.

   (c) If v is used between *init* and *end*, then:

      i. If v is only read in both between *init* and *end* and within the task, then the v is scoped as `FIRSTPRIVATE`.

      ii. If v is written in either between *init* and *end* or within the task, then we look for *data race* conditions (see 6.2.1.1 for the details about data race analysis), thus:

         A. If we can assure that no data race can occur, then v is scoped as `SHARED`.

         B. If it can occur a data race condition, then we tag v as a `RACE`. At the end of the analysis we will decide how do we deal with these variables.

4. For each use $a_{i,\ i \in [0..N]}$ (where N is the number of uses) of an array variable a appearing within the task t.

   (a) We apply the methodology used for the scalars.
   (b) Since OpenMP does not allow different scopes for the subparts of a variable, then we have to mix all the results we have get in the previous step. In order to do that we will follow the rules bellow:
      i. If the whole array a or all the parts a_i have the same scope sc, then a is scoped as sc.
      ii. If there are different regions of the array with different scopes, then:
         A. If some a_i has been scoped as `UNDEFINED` then a is scoped as `UNDEFINED`.
         B. If at least one a_i is `FIRSTPRIVATE` and all $a_{j,\ j \in [0..N]}$ where $j! = i$ are `PRIVATE`, then a is scoped as `FIRSTPRIVATE`.
         C. If at least one a_i is `SHARED` and all $a_{j,\ j \in [0..N]}$ where $j! = i$ are `PRIVATE` or `FIRSTPRIVATE`, then, fulfilling the sequential consistency rules, a is scoped as `SHARED`.

5. NOTE: If we cannot determine the `init` point, then we cannot analyze the task because we do not know which regions of code can be executed in parallel with t.
6. NOTE: If we cannot determine the `end` point, then we can only scope those variables that are local to the function containing t.
7. NOTE: This algorithm is not dealing with aggregates.

When we are executing auto-scoping analysis, variables must be classified into *PRIVATE*, *FIRSTPRIVATE* or *SHARED*. Variables which has been determined as *RACE* have to be classified. Since OpenMP standard says that the occurrence of a data race implies the result of the program to be unspecified and this is not the behavior we expect for a program, then auto-scoping analysis will privatize all the variables classified as *RACE*. So, given a variable v classified as *RACE*, then:

– If the first action performed in v within the task is a write, then v is scoped as `PRIVATE`.
– If the first action performed in v within the task is a read, then v is scoped as `FIRSTPRIVATE`.

6.2.1.1 Data race conditions

Data race conditions can appear when two threads can access to the same memory unit at the same time and at least one of these accesses is a write.

In order to analyze data race conditions in the process of auto-scoping the variables of a task we have to analyze the code appearing in all regions defined between the *init* and *end* points described in the previous section. Any variable *v* appearing in two different regions where at least one of the accesses is a write and none of the two accesses is blocked by either and `atomic` construct, a `critical` construct or a lock routine (`omp_init_lock` / `omp_destroy_lock`, `omp_set_lock` / `omp_unset_lock`), can trigger a data race situation.

6.2.2 The results

We have tested the auto-scoping algorithm in a set of benchmarks. We explain the results in the sections bellow.

6.2.2.1 Fibonacci

Fibonacci is a recursive algorithm that computes a sequence of integers called Fibonacci numbers. Auto-scoping analysis applied to the algorithm shown in Listing 6.6 results as follows:

– Firstprivate: *n*.

– Shared: *x, y*.

Variable *n* is firstprivatized because it is only read inside the task and no simultaneous code to the task writes to this variable. Instead, variables *x* and *y* are shared as there is no data race condition with them (a `taskwait` in every level of recursion avoids multiple accesses to the same variable at the same time) and their values are live out of the task. This is the result we expected.

```
int fib (int n) {
  int x, y;
  if (n < 2) return n;

#pragma omp task untied default(AUTO)
  x = fib(n - 1);
#pragma omp task untied default(AUTO)
  y = fib(n - 2);
#pragma omp taskwait

  return x + y;
}

void fib_par (int n) {
#pragma omp parallel
#pragma omp single
  par_res = fib(n);
}
```

Listing 6.6: Fibonacci code from BOTS benchmarks

6.2.2.2 Floorplan

Floorplan is an optimizing code that computes the optimal placement of cells in a floorplan. In Listing 6.7 we show the code of the main function. When we run the auto-scoping analysis in the *floorplan* code we obtain the following classification:

– Private: $area, footprint$.

– Firstprivate: $i, j, NWS, id, FOOTPRINT, \&footprint, CELLS, BOARD$.

– Shared: nnc.

– Undefined: $N, MIN\_AREA, MIN\_FOOTPRINT, BEST\_BOARD, board$.

– Race: $area$.

First of all, we explain the `UNDEFINED` results. $N, MIN\_AREA, MIN\_FOOTPRINT$ and $BEST\_BOARD$ are global variables; since the method $memcpy$, not accessible to us, is called in the function, and these global variables are not defined before the call to $memcpy$, then they have an undefined behavior. $board$ is a parameter passed by reference to the method $memcpy$; as it happened with the global variables, we do not know which is the behavior of this variable, so we tag it as `UNDEFINED`.

The variable nnc is tagged as `SHARED` because its use is protected with an `atomic` construct, so it can not produce data race, and the value is live out of the task.

We detect a possible data race in the access to the variable $area$. This variable is tagged as `RACE`. At the end of the analysis, the variable is privatized to avoid the data race. This privatization causes the variable to be `PRIVATE` because the first use of the variable within every thread is a write. Variable $footprint$ is `PRIVATE` because the task kills the two first positions of the array and the rest is never used.

Finally, regarding to the `FIRSTPRIVATE` set, we see variables that, in this case, are only used within the task. These are NWS, which appears with the exact range of accesses performed regarding to the most significant dimension (the access to the less significant in constant). Variables $CELLS$ and $BOARD$ are passed by value to the method $memcpy$, so they are just used within the task. The address of the variable $footprint$ is passed by value to the recursive call to add_cell, so this address is `FIRSTPRIVATE`.

This were the results we expected.

```
void compute_floorplan (void) {
  coor footprint;
  footprint[0] = 0;
  footprint[1] = 0;
  bots_number_of_tasks = add_cell(1, footprint, board, gcells);
}

static int add_cell(int id, coor FOOTPRINT, ibrd BOARD, struct cell *CELLS) {
  int  i, j, nn, area, nnc,nnl;
  ibrd board;
  coor footprint, NWS[DMAX];
  nnc = nnl = 0;

  for (i = 0; i < CELLS[id].n; i++) {
    nn = starts(id, i, NWS, CELLS);
    nnl += nn;
    for (j = 0; j < nn; j++)
#pragma omp task untied autodeps
{
      struct cell cells[N+1];
      memcpy(cells,CELLS,sizeof(struct cell)*(N+1));
      cells[id].top = NWS[j][0];
      cells[id].bot = cells[id].top + cells[id].alt[i][0] - 1;
      cells[id].lhs = NWS[j][1];
      cells[id].rhs = cells[id].lhs + cells[id].alt[i][1] - 1;
      memcpy(board, BOARD, sizeof(ibrd));

      if (!lay_down(id, board, cells))
        goto _end;

      footprint[0] = max(FOOTPRINT[0], cells[id].bot+1);
      footprint[1] = max(FOOTPRINT[1], cells[id].rhs+1);
      area         = footprint[0] * footprint[1];

      if (cells[id].next == 0) {
        if (area < MIN_AREA) {
#pragma omp critical
          if (area < MIN_AREA) {
            MIN_AREA         = area;
            MIN_FOOTPRINT[0] = footprint[0];
            MIN_FOOTPRINT[1] = footprint[1];
            memcpy(BEST_BOARD, board, sizeof(ibrd));
          }
        }
      } else if (area < MIN_AREA) {
#pragma omp atomic
          nnc += add_cell(cells[id].next, footprint, board, cells);
      }
See also the specialization of the variable
_end:;
}
  }

#pragma omp taskwait
  bots_number_of_tasks = nnc + nnl;
}
```

Listing 6.7: Floorplan code from BOTS benchmarks

6.2.2.3 Nqueens

Nqueens is a search algorithm that finds a solution of the N Queens problem. In Listing 6.8 we present the algorithm. The auto-scoping analysis performed in this benchmark computes, as it was expected, the following information:

– Firstprivate: *i, j, mycount, n, a.*

Variables i, j, n are only used within the task and also outside the task; since the values are not modified, then we make them PRIVATE. $mycount$ is privatized because the use of this variable can occur concurrently in two different tasks (recursive call to $nqueens$) and the access is not restricted, then there can be a data race condition and we privatize the variable; since the value is always used before being defined, then we define $mycount$ as FIRSTPRIVATE. Variable a is is passed by value to the function call to $memcpy$, and this value is never used in a concurrent section of code that executes at the same time as the task, then we make the variable FIRSTPRIVATE.

```
void nqueens(int n, int j, char *a, int *solutions) {
  int i;

  if (n == j) {
    mycount++;
    return;
  }

  for (i = 0; i < n; i++) {
#pragma omp task untied default(auto)
    {
      char * b = alloca(n * sizeof(char));
      memcpy(b, a, j * sizeof(char));
      b[j] = (char) i;
      if (ok(j + 1, b))
        nqueens(n, j + 1, b);
    }
  }
}
```

Listing 6.8: Nqueens code from BOTS benchmarks

6.2.2.4 Cholesky

Cholesky decomposition is an algorithm for linear algebra programming. We present in Listing 6.9 an implementation of the *cholesky* solution with different tasks. Applying the auto-scoping algorithm to this code returns the following classification:

– First, second, fifth and sixth tasks:
  * Firstprivate: j, jj
  * Race: a

– Third and fourth tasks:
  * Firstprivate: j
  * Race: a

Variables j and jj are only used within the tasks and no other statement, inside or outside the tasks can create a data race condition, so this variables are scoped as FIRSTPRIVATE. After the computation, variable a is defined as RACE because multiple accesses to the same array can be done at the same time, being some of them writes. Because of that, we must privatize the variable. Since some of the values of a are first read and some others are first write, we scope the variable as FIRSTPRIVATE. We obtain the result we expected.

```
void cholesky(float a[NUM_ELEMS][NUM_ELEMS])
{
  for (int jj = 0; jj < NUM_ELEMS; jj += BLOCK_SIZE*) {
    for (int j = jj; j < MIN(NUM_ELEMS, jj + BLOCK_SIZE); j++) {
#pragma omp task            // 1
      for (int i = j + 1; i < (jj + BLOCK_SIZE); i++)
        for (int k = 0; k < j; k++)
          a[i][j] = a[i][j] - a[i][k] * a[j][k];

#pragma omp task            // 2
      for (int i = (jj + BLOCK_SIZE); i < NUM_ELEMS; i++)
          for (int k = 0; k < j; k++)
              a[i][j] = a[i][j] - a[i][k] * a[j][k];

#pragma omp task            // 3
      for (int k = 0; k < j; k++)
          a[j][j] = a[j][j] - a[j][k] * a[j][k];

#pragma omp task            // 4
      a[j][j] = sqrt(a[j][j]);

#pragma omp task            // 5
      for (int i = j + 1; i < (jj + BLOCK_SIZE); i++)
          a[i][j] = a[i][j] / a[j][j];

#pragma omp task            // 6
      for (int i = (jj + BLOCK_SIZE); i < NUM_ELEMS; i++)
          a[i][j] = a[i][j] / a[j][j];
    }
  }

#pragma omp taskwait
}
```

Listing 6.9: Cholesky code

6.2.2.5 Stencil

The code presented in Listing 6.10 is an stencil algorithm using and defining different regions in a matrix. Applying the auto-scoping algorithm we obtain, as we expected, the following result:

– Firstprivate: $I, J, iter, A$.

Variables $I, J, iter$ are FIRSTPRIVATE because they are only used within the task and there is not a statement that can cause a data race in the code outside the task. Variable A is tagged as RACE because, even if many of the accesses to the array are reads, there is one access that can cause different tasks to write to the same memory location at the same time. Afterwards, variable A is privatized to avoid the data race and, since some of the access are first read, then the variable is FIRSTPRIVATE.

```
int main(int argc, char **argv) {
  long (*A)[(NB+2)*B];
  alloc_and_genmat(&A);
  int iters, z = 0;
  long i, j, k, l;
  double diff;

  for (iter=0; iter<1; iter++) {
    for (i=B; i < (NB+1)*B; i+=B) {
      for (j=B; j < (NB+1)*B; j+=B) {
        long I = i-1, J=j-1;
#pragma omp task
        {
          if (I+1L == 1*B)
            for (k=1; k <= B; k++)
              if (A[0][k] != INITIAL_VALUE(I, 0, J, k))
                abort();
          else
            for (k=1; k <= B; k++)
              if (A[0][k] != INITIAL_VALUE(I, 0, J, k)
                  + ITERATION_INCREMENT*(iter+1L))
                abort();

          if (I+1L == (1+NB-1)*B)
            for (k=1; k <= B; k++)
              if (A[B+1][k] != INITIAL_VALUE(I, B+1, J, k))
                abort();
          else
            for (k=1; k <= B; k++)
              if (A[B+1][k] != INITIAL_VALUE(I, B+1, J, k)
                  + ITERATION_INCREMENT*iter)
                abort();

          if (J+1L == 1*B)
            for (k = 1; k <= B; i++)
              if (A[k][0] != INITIAL_VALUE(I, k, J, 0))
                abort();
          else
            for (k = 1; k <= B; k++)
              if (A[k][0] != INITIAL_VALUE(I, k, J, 0)
                  + ITERATION_INCREMENT*(iter+1L))
                abort();

          if (J+1L == (1+NB-1)*B)
            for (k = 1; k <= B; i++)
              if (A[k][B+1] != INITIAL_VALUE(I, k, J, B+1))
                abort();
          else
            for (k = 1; k <= B; k++)
              if (A[k][B+1] != INITIAL_VALUE(I, k, J, B+1)
                  + ITERATION_INCREMENT*iter)
                abort();

          for (k = 1; k <= B; i++)
            for (l=1; l <= B; l++)
              A[k][l] += ITERATION_INCREMENT;
        }
        z++;
      }
    }
  }

#pragma omp barrier
  return 0;
}
```

Listing 6.10: Stencil code

6.3 Automatic dependencies discovery in tasks

6.3.1 Scope of the optimization

As we explained in Section 4.2, OpenMP defines the `task` directive which allows asynchronous parallelism. The construct can be followed by a series of clauses describing the scope of the variables inside the task. OmpSs extends this directive allowing the definition of dependencies in the tasks. The runtime decision about which tasks have to be executed before the execution of the current task and which tasks have to be executed after the execution of the current task is made in terms of these dependencies. Four clauses allow the specification of data dependencies in tasks: `input`, `output`, `inout` and `concurrent`.

Using the optimization described in Section 6.2 we can substantially improve the programmability of OmpSs releasing the programmer from the arduous task of defining the dependencies of a task. In the previous analysis we classified the variables within a task in four different groups: `PRIVATE`, `FIRSTPRIVATE`, `SHARED` and `UNDEF`. Any dependency from a given task to another can occur only for shared variables or for some of the variables that during the auto-scoping have been detected as `RACE`. The algorithm to determine the dependencies among the tasks is almost the same as defined for the auto-scoping. The algorithm is defined as follows:

1. For a given task *t*, we run the algorithm defined for auto-scoping until we have classified the variables as `PRIVATE`, `FIRSTPRIVATE`, `SHARED`, `UNDEF` and `RACE`. We do not specify the variables scoped as `RACE` directly as `PRIVATE` or `FIRSTPRIVATE`; instead of that, for each variable *v* classified as `RACE`, we distinguish two cases:
   (a) If the race condition occurs between one statement in *t* and some statement executed within the sequential code executed concurrently with the task by the encountered thread that created the task, then *v* has to be privatized to avoid the race condition. In that case, we use the same methodology used in the algorithm of auto-scoping:
      i. If the first action performed in *v* is a write, then *v* is scoped as `PRIVATE`.
      ii. If the first action performed in *v* is a read, then *v* is scoped as `FIRSTPRIVATE`.
   (b) If the race condition occurs between *t* and some other task that can be executed concurrently, then we can consider *v* as `SHARED` and we

can compute the dependencies between the different tasks involved in the race condition in order to avoid this situation. We classify these variables into three groups: `INPUT`, `OUTPUT` and `INOUT`. Using the same nomenclature as we used for the auto-scoping algorithm, we define the classification as follows:

i. If v is live at the `entry` and the `exit` points of the task, then it is scoped as `INOUT`.

ii. If v is live only at the `entry` of the task, then it is scoped as `INPUT`.

iii. If v is live only at the `exit` of the task, then it is scoped as `OUTPUT`.

iv. If none of the previous cases apply, then v remains as `SHARED`. This means that the variable is accessed by the task and the code executed by the encountering thread that creates the task without a data race condition, and no other statement after the synchronization of the task will use the value it produces, so it cannot be a dependence.

2. NOTE: At that moment we are not able to distinguish variables that are tagged as `INOUT` from variables that can be `CONCURRENT`. This is an specialization of the clause `INOUT`, so the dependence computed is not correct, but is more restrictive than needed.

6.3.2 The results

We have tested the auto-dependencies a set of algorithms. Since the variables which are interesting in this analysis are these that have been classified as `RACE` or `SHARED` in the auto-scoping analysis, we present the results for the examples introduced in Section 6.2 that have matched this kind of variables.

6.3.2.1 Fibonacci

We take the example introduced in Listing 6.6. In the auto-scoping analysis we found two ompSHARED variables: x and y. When we run auto-dependencies analysis, we obtain that both variables are computed as `OUTPUT` dependencies, each one for its respective task. This is because there is no task using the value written in a previous task, but the value is read after the synchronization point. Variable n classified as `FIRSTPRIVATE` remains as such. This is the result we expected.

6.3.2.2 Floorplan

In the *floorplan* benchmark introduced in Listing 6.7 we found *nnc* as `SHARED`. This variable is an input value coming from an output value of a previous task. With the auto-dependencies analysis we obtain *nnc* as an *inout* dependence in the task. Variable *area* is classified as `RACE`; since the value of the variable within a task is always written before being read, then the variables is classified as an `OUTPUT` dependence. The rest of variables remain as they were in the auto-scoping analysis (variables that are `FIRSTPRIVATE` are not ranged because at that moment `firstprivate` clause does not accept array regions). This is the result we expected.

6.3.2.3 Cholesky

For Cholesky code introduced in Listing 6.9, the auto-scoping algorithm found shared variables for all the tasks in the code. Each task defined as shared the variable *A*. As we expected, the auto-dependencies computation for the arrays appearing in each one of the tasks is the following:

– First task:
    * Input: $a[j+1:-1+(128+jj):1][0:-1+j:1], a[j][0:-1+j:1]$
    * Inout: $a[j+1:-1+(128+jj):1][j]$

– Second task:
    * Input: $a[jj+128:4:1][0:-1+j:1], a[j][0:-1+j:1]$
    * Inout: $a[jj+128:4:1][j]$

– Third task:
    * Input: $a[j][0:j-1:1]$
    * Inout: $a[j][j]$

– Forth task:
    * Inout: $a[j][j]$

– Fifth:
    * Input: $a[j][j]$
    * Inout: $(a[j+1:-1+(128+jj):1][j]$

– Sixth:
    * Input: $a[j][j]$
    * Inout: $a[jj+128:4:1][j]$

6.3.2.4 Stencil

The stencil code in Listing 6.10 computes the array A as a variable that can produce a race condition. Since it is tagged as `RACE`, then the specific ranges of the variable accessed within the task are analyzed in order to find out the dependencies and we obtain the following results:

– Firstprivate: $A[0][1:256L:1], A[256L+1][1:256L:1], A[1:256L:1][0], A[1:256L:1][256L+1]$

– Inout: $A[1:256L:1][1:256L:1]$

The rest of variables remain as they were classified in the auto-scoping analysis. This is the result we expected.

Chapter 7. *State of the Art*

A huge number of researchers have focused in concurrency and data flow analysis for shared memory programming models. Regarding to the State of the Art we focus in two different aspects: on one hand the Control Flow Graph and on the other hand the analysis we have defined based on these graph and the classical analysis for sequential or for synchronous parallel codes to extend them to asynchronous parallelism.

Common Control Flow Graphs for sequential codes cannot express the behavior of parallel codes. In order to symbolize the parallel information and the relaxed memory consistency model of OpenMP this representation must be extended and enriched. Different approaches have appeared in the 20 last years with the goal of analyzing and optimizing explicitly parallel codes. But OpenMP is a programming model in continuous development, and that gives us the opportunity of improve the existing representations adapting them to the new semantics.

Wolfe and Srinivasan [WS91] presented new data structures as a Parallel Control Flow Graph and the Parallel Precedence Graph for programs with parallel constructs. Based on the fact that precedence relation is not the same as dominance relation in parallel programs, they used these new structures to develop algorithms for optimizing parallel programs. Although theirs is a powerful representation, their optimizations are based in Parallel Static Single Assignment and we aim to remain in a highest representation of the program. Grunwald and Srinivasan [GS93] presented a Parallel Flow Graph and a set of data-flow equations for computing reaching definitions in explicitly parallel programs with event synchronization. However, their work is focused only in `parallel sections` and event synchronizations.

Satoh, Kusano and Sato [SKS01] proposed a Parallel Control Flow Graph modeling both flow control and synchronization between threads. They approach the reaching definitions problem based on the synchronization nodes in the graph for both intra-procedural and inter-procedural analysis. They present different optimizations for explicitly parallel programs such as reduction of coherence overhead, redundant barrier removal and privatization of dynamically-allocated objects. Nonetheless, they do not cover all the OpenMP constructs and they do not consider the impact of flush operations, preventing this of any violation of the memory consistency rules of OpenMP when applying optimizations.

Huang et al. [HEHC09] developed a compiler framework for OpenMP program analysis and optimization. Based on the OpenMP relaxed memory semantics they are able to remove the conservative restrictions on optimizing in the presence of shared data. Based on the previous work of Huang, Sethuraman and Chapman [HSC07], they make explicit barriers and define liveness equations for the parallel nodes in the Parallel Control Flow Graph. They applied these studies in the OpenUH compiler to optimize OpenMP constructs before they are lowered to threaded code and get encouraging results in terms of performance improvement after code analysis and optimizations.

However, none of the previous works have presented any approach for asynchronous parallel executions such as OpenMP `task` constructs. Weng and Chapman [WC03] defined a task execution graph representing precedence among tasks based on their dependencies. However, it can be expensive to apply their testing between two call statements or two larger regions of code because they try to make decisions on scheduling. Instead of that, we represent the semantics on the asynchronous task execution in our Parallel Control Flow Graph. The control flow in the occurrence of the `task` directive cannot be processed in the same way as the rest of OpenMP directives in the sense that its control flow is more relaxed than the others. We add tasks in the graph where the scheduling point of the task is defined inside the code. Since the exact point of execution of the tasks will be decided at run-time and not at compile time, our analysis referred to this directive must define the range of code where this task can be executed in parallel. This range will start always in the task scheduling point and will end when a synchronization point ensures the task has been executed.

Based on the OmpSs programming model, tasks can be defined together with a set of clauses specifying the dependences of the task. With the aim of improve the programmability of our parallel programming model, we have developed a new task analysis to determine automatically the correct dependencies of a given task untying the user from the job of defining each single dependence for every task. We have added to the `task` directive the clause `auto-dependence` which lets the compiler the responsibility of computing the correct dependencies of the tasks. We are not being conservative in this analysis, so, in the case that the compiler cannot assure the data-sharing of a given variable, then the result of this analysis for that variable will be that the compiler returns to the user the responsibility of defining the correct dependencies.

Defining the dependencies of a task requires the previous analysis of the scoping of variables. Auto-scoping rules for `parallel` have been defined previously by Lin et al. [LTaMC04]. They defined the clause `default(auto)` for

OpenMP constructs and they established and algorithm for auto-scoping rules in synchronous parallel executions for both scalar and array variables. These rules do not work properly with the asynchronous execution of tasks. We have defined a new algorithm that determines the scoping of the variables inside a task depending on the use of this variable between the scheduling point of the task and the synchronization points that follow the scheduling point. In the case the compiler is not able to determine a data-sharing, then it warns to user to do that work by hand.

Chapter 8. *Conclusions and Future Work*

8.1 Conclusions

In this thesis we have presented a set of compiler analyses and optimizations in the context of the Mercurium compiler and the OmpSs programming model. We have shown an adaptation of some of the most common classic compiler analysis to the asynchronous parallelism expressed with OmpSs tasks such as a new Parallel Control Flow Graph for control flow analysis and new rules for liveness analysis. We have defined as well a set of use cases using the previous analysis that demonstrates the benefits of compiler analyses to increase the productivity of systems. These use cases have induced us to develop new techniques for asynchronous parallel codes that were developed only for synchronous parallelism, like the automatic discovery of the scope of variables in OpenMP tasks. We have enhanced our system by improving Mercurium performance with the privatization of shared variables in the code generation and by improving OmpSs programmability with the automatic detection of dependencies between tasks, releasing the user from this work in many cases.

The results obtained in the different tests applied to our optimizations demonstrate the profit we can obtain due to the use of compiler analysis. With the privatization of shared variables we have improved the performance in codes where there is a high number of accesses to shared memory. With the automatic computation of the scope of variables and tasks dependencies we have enhanced the programmability of OmpSs making easier for programmers to use the parallel model.

We are highly motivated at the end of this project because we have transcended the expectations of the use cases we were be able to implement. The definition of the automatic detection of the dependencies in tasks as a goal for this project burst upon the problem as of the automatic scoping in tasks. We have defined and tested a new algorithm to solve that problem and finally, we have found a solution for the automatic dependencies computation use case. This result has a great importance because of its contribution to the compiler analyses area; the analysis is not only applicable to our scope, but also to any other compiler implementing analysis for asynchronous parallel applications.

8.2 Future Work

Because of the complexity and amplitude of C/C++ languages, different interesting cases did not fit in our time limitations. For example, we are only able to analyze *for-loop* constructs and we cannot deal with complex loop boundaries in induction variables or induction variables not appearing in the loop control statements. At the beginning of the project we were ambitious in the sense of exploiting the Mercurium intermediate representation to deal with Fortran examples but we did not have the time to test our analyses with these samples. Nonetheless, since the representation is the same in many constructs, we only have to extend our implementation for the AST nodes that are specific for FORTRAN codes.

We are now thinking about the directions we want to take in mid- and long-term to extend our analyses and to use them in order to better improve Mercurium and OmpSs productivity. We show bellow a list of the most important issues we have in mind:

– To apply the analyses in Fortran codes.
– To extend the analysis of loops to deal with other iterative constructs rather than *for-loops*, such as *while-loops* or *goto-statements*
– To implement the dependencies analysis in the OpenMP `taskwait` clause. This may allow us to determine which tasks should a given `taskwait` wait for, sic, automatically determine the clause `on` in the `taskwait`.
– To extend our algorithms of auto-scoping and auto-dependencies to deal with aggregates.
– To extend our auto-scoping algorithm to distinguish when a variable that can be shared and private, is useful to be privatized. A priori, scalars will always be privatized because the cost of making a copy is the same as accessing to a shared variable in the worst of the cases (the variable is only accessed once) and it is always better to do the copy in the rest of the cases. For the arrays is not clear that privatizing is the best option because of the cost of this process.
– To tag functions that are common like *memcpy* or *alloca*. Since the compiler can know which is the behavior of this functions, then, it can annotate the usage of the parameters. Thus, during the analyses (Use-Definition and so on), the global variables will not be classified as `UNDEF` because of the appearance of these functions.

Besides getting deeper into this subject, we are currently working on a workshop paper with the results of our automatic scoping in OpenMP tasks and we will present it in the next International Workshop on OpenMP (IWOMP).

References

[Boa11] OpenMP Architecture Review Board. The openmp® api specification for parallel programming, September 2011.

[DAB+11] Alejandro Duran, Eduard Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. Ompss: a proposal for programming heterogeneous multi-core architectures. Parallel Processing Letters, 21(2):173–193, 2011.

[GS93] Dirk Grunwald and Harini Srinivasan. Data flow equations for explicitly parallel programs. In PPOPP, pages 159–168, 1993.

[HEHC09] Lei Huang, Deepak Eachempati, Marcus W. Hervey, and Barbara M. Chapman. Exploiting global optimizations for openmp programs in the openuh compiler. In Daniel A. Reed and Vivek Sarkar, editors, PPOPP, pages 289–290. ACM, 2009.

[HSC07] Lei Huang, Girija Sethuraman, and Barbara M. Chapman. Parallel data flow analysis for openmp programs. In Barbara M. Chapman, Weimin Zheng, Guang R. Gao, Mitsuhisa Sato, Eduard Ayguadé, and Dongsheng Wang, editors, IWOMP, volume 4935 of Lecture Notes in Computer Science, pages 138–142. Springer, 2007.

[LTaMC04] Yuan Lin, Christian Terboven, Dieter an Mey, and Nawal Copty. Automatic scoping of variables in parallel regions of an openmp program. In Barbara M. Chapman, editor, WOMPAT, volume 3349 of Lecture Notes in Computer Science, pages 83–97. Springer, 2004.

[PBAL09] Judit Planas, Rosa M. Badia, Eduard Ayguadé, and Jesús Labarta. Hierarchical task-based programming with starss. IJHPCA, 23(3):284–299, 2009.

[Sar97] Vivek Sarkar. Analysis and optimization of explicitly parallel programs using the parallel program graph representation. In Zhiyuan Li, Pen-Chung Yew, Siddhartha Chatterjee, Chua-Huang Huang, P. Sadayappan, and David C. Sehr, editors, LCPC, volume 1366 of Lecture Notes in Computer Science, pages 94–113. Springer, 1997.

[SKS01] Shigehisa Satoh, Kazuhiro Kusano, and Mitsuhisa Sato. Compiler optimization techniques for openmp programs. Scientific Programming, 9(2-3):131–142, 2001.

[WC03] Tien-Hsiung Weng and Barbara M. Chapman. Asynchronous execution of openmp code. In Peter M. A. Sloot, David Abramson, Alexander V. Bogdanov, Jack Dongarra, Albert Y. Zomaya, and Yuri E. Gorbachev, editors, International Conference on Computational Science, volume 2660 of Lecture Notes in Computer Science, pages 667–678. Springer, 2003.

[WS91] Michael Wolfe and Harini Srinivasan. Data structures for optimizing programs with explicit parallelism. In Hans P. Zima, editor, ACPC, volume 591 of Lecture Notes in Computer Science, pages 139–156. Springer, 1991.