On the Interoperability of Programming Languages based on the Fork-Join Parallelism Model

(Spine title: On the Interoperability of Programming Languages based on the Fork-Join Parallelism Model)

(Thesis format: Monograph)

by

Sushek Shekar

Graduate Program
in
Computer Science

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science

The School of Graduate and Postdoctoral Studies
The University of Western Ontario
London, Ontario, Canada

THE UNIVERSITY OF WESTERN ONTARIO
THE SCHOOL OF GRADUATE AND POSTDOCTORAL STUDIES

CERTIFICATE OF EXAMINATION

Supervisor:

Dr. Marc Moreno Maza

Examination committee:

Dr.Allan MacIsaac

Dr.Hanan Lutfiyya

Dr.Kaizhong Zhang

The thesis by

Sushek Shekar

entitled:

On the Interoperability of Programming Languages based on the Fork-Join Parallelism Model

is accepted in partial fulfillment of the
requirements for the degree of
Master of Science

Date _____

Chair of the Thesis Examination Board

Abstract

This thesis describes the implementation of MetaFork, a meta-language for concurrency platforms targeting multicore architectures. First of all, MetaFork is a multithreaded language based on the fork-join model of concurrency: it allows the programmer to express parallel algorithms assuming that tasks are dynamically scheduled at run-time. While MetaFork makes no assumption about the run-time system, it formally defines the serial C-elision of a MetaFork program.

In addition, MetaFork is a suite of source-to-source compilers permitting the automatic translation of multithreaded programs between programming languages based on the fork-join model. Currently, this compilation framework supports the OpenMP and CilkPlus concurrency platforms. The implementation of those compilers explicitly manages parallelism according to the directives specified in MetaFork, OpenMP and CilkPlus.

We evaluate experimentally the benefits of MetaFork. First, we show that this framework can be used to perform comparative implementation of a given multithreaded algorithm so as to narrow performance bottlenecks in one implementation of this algorithm. Secondly, we show that the translation of hand written and highly optimized code within MetaFork generally produces code with similar performance as the original.

Keywords. Multicore architectures, concurrency platforms, fork-join model, CilkPlus, OpenMP, source-to-source compilation, performance.

Acknowledgments

Firstly, I would like to thank my thesis supervisor Dr. Marc Moreno Maza in the Department of Computer Science at The University of Western Ontario. His helping hands toward the completion of this research work were always extended for me. He consistently helped me on the way of this thesis and guided me in the right direction whenever he thought I needed it. I am grateful to him for his excellent support to me in all the steps of successful completion of this research.

Secondly, I would like to thank Xiaohui Chen in the Department of Computer Science at The University of Western Ontario for working along with me and helping me successfully complete this research work. I am also grateful to our colleagues Priya Unnikrishnan and Abdoul-Kader Keita (IBM Toronto Labs) for their advice and technical support.

Thirdly, all my sincere thanks and appreciation go to all the members from our Ontario Research Centre for Computer Algebra (ORCCA) lab, Computer Science Department for their invaluable teaching support as well as all kinds of other assistance, and all the members of my thesis examination committee.

Finally, I would like to thank all of my friends around me for their consistent encouragement and would like to show my heartiest gratefulness to my family members for their continued support.

Contents

Chapter 1

Introduction

In the past decade the pervasive ubiquitous of multicore processors has stimulated a constantly increasing effort in the development of concurrency platforms (such as CilkPlus, OpenMP, Intel TBB) targeting those architectures. While those programming languages are all based on the fork-join parallelism model, they largely differ on their way of expressing parallel algorithms and scheduling the corresponding tasks. Therefore, developing programs involving libraries written with several of those languages is a challenge.

Nevertheless there is a real need for facilitating interoperability between concurrency platforms. Consider for instance the field of symbolic computation. The DMPMC library (from the TRIP project `www.imcce.fr/trip` developed at the *Observatoire de Paris*) provides sparse polynomial arithmetic and is entirely written in OpenMP meanwhile the BPAS library (from the *Basic Polynomial Algebra Subprograms* `www.bpaslib.org` developed at the University of Western Ontario) provides dense polynomial arithmetic and entirely written in CilkPlus. Polynomial system solvers require both sparse and dense polynomial arithmetic and thus could take advantage of a combination of the DMPMC and BPAS libraries. Unfortunately, the run-time systems of CilkPlus and OpenMP cannot cooperate since their schedulers are based on different strategies, namely *work stealing* and *work sharing*, respectively. Therefore, an automatic translation mechanism is needed between CilkPlus and OpenMP.

Another motivation for such software tools is *comparative implementation*, with the objective of narrowing performance bottlenecks. The underlying observation is that the same multithreaded algorithm, based on the fork-join parallelism model, implemented with two different concurrency platforms, say CilkPlus and OpenMP, could result in very different performance and this performance is often very hard to

analyze and compare. If one code scales well while the other does not, one may suspect an efficient implementation of the latter as well as other possible causes such as higher parallelism overheads. Translating the inefficient code to the other language can help narrowing the problem. Indeed, if the translated code still does not scale one can suspect an implementation issue (say the programmer missed to parallelize one portion of the algorithm) whereas if it does scale, then one can suspect a parallelism overhead issue in the original code (say the grain-size of the parallel for-loop is too small).

In this thesis, we propose MetaFork, a metalanguage for multithreaded algorithms based on the fork-join parallelism model [18] and targeting multicore architectures. By its parallel programming constructs, this language is currently a super-set of CilkPlus [20, 34, 38] and OpenMP Tasks [13]. However, this language does not compromise itself in any scheduling strategies (work stealing [19], work sharing, etc.) Thus, it does not make any assumptions about the run-time system.

Based on the motivations stated above, a driving application of MetaFork is to facilitate automatic translations of programs between the above mentioned concurrency platforms. To date, our experimental framework includes translators between CilkPlus and MetaFork (both ways) and, between OpenMP and MetaFork (both ways). Hence, through MetaFork, we are able to perform program translations between CilkPlus and OpenMP (both ways). Adding Intel TBB to this framework is work in progress. As a consequence, the MetaFork program examples given in this thesis were obtained either from original CilkPlus programs or from original OpenMP programs.

The original contributions of this work are presented in Chapter 3, 4 and 5. They are a joint work with Xiaohui Chen and Dr.Marc Moreno Maza.

In Chapter 3, we specify the syntax and semantics of the parallel constructs of MetaFork. Indeed, MetaFork differs only from the C language by its parallel constructs. Hence, it is sufficient to define MetaFork's parallel constructs in order to define the entire language. As we shall see, the main difficulty is to define the serial C-elision of a given MetaFork program $\mathcal{M}$, that is the program $\mathcal{C}$ written in the C language and which computes exactly the same thing as $\mathcal{M}$. As a byproduct of this definition, we obtain a strategy for translating MetaFork code to CilkPlus. Since CilkPlus is a subset of MetaFork, translating in the other direction is easy.

In Chapter 4, we propose algorithms for translating programs between OpenMP Tasks and MetaFork. Since data attribute rules are not the same in both languages, great care is required. These algorithms have been implemented to-

gether with those presented in Chapter 3. They form the a suite of four language translators: MetaFork to CilkPlus, CilkPlus to MetaFork, MetaFork to OpenMP Tasks, OpenMP Tasks to MetaFork. As a byproduct one can translate CilkPlus code to OpenMP Tasks and vice-versa.

In Chapter 5, we report on our experimentation with our language translators: CilkPlus to OpenMP Tasks and OpenMP Tasks to CilkPlus. This actually includes two sets of experiences corresponding respectively to the above motivations For this experimentation to be meaningful, it is essential to ensure that each translated program computes the same thing as its original counterpart. To do this, we proceed as follows with most test cases. The original program, say $\mathcal{P}$, contains both a parallel code and its serial elision (manually written). When program $\mathcal{P}$ is executed, both codes run and compare their results. Let us call $\mathcal{Q}$ the translated version of $\mathcal{P}$. Since serial elisions are unchanged by translation, then $\mathcal{Q}$ can be verified by the same process used for program $\mathcal{P}$.

Our experimentation suggest that MetaFork can, indeed, be used to narrow performance bottlenecks. More precisely, for each of our test cases, the overall trend of the performance results can be used to narrow causes for low performance. In addition, MetaFork can be used to translate large projects written with one concurrency platform to another, as we did with the entire BOTS test suite [26]. This application was the initial motivation of this work. Finally, this translation process seems not to add any overheads on the tested examples.

Chapter 2

Background

This section gathers background materials which are used throughout this thesis. Section 2.1 describes the parallelism model used in MetaFork which is the *fork-join model*. Section 2.2 is a brief introduction to the history of the OpenMP standard; it also describes its API and runtime support. Section 2.3 is a brief introduction to the history of CilkPlus, its API and runtime support. Section 2.4 gives a brief overview to compilation theory, steps involved in compilation and source-to-source compilers. In particular, Sections 2.4.3 and Section 2.4.5 present the Lex and YACC software tools for lexical and semantic analyzes.

2.1 Fork-join model

The fork-join execution model is a model of computations where a parent task gives birth to child tasks, all tasks (parent and children) execute code paths concurrently and synchronize at the point where the child tasks terminate. On a single core, a child task preempts its parent which resumes its execution when the child terminates.

In particular, the *fork-join parallelism model* is a simple theoretical model of multithreading for parallel computation. This model represents the execution of a multithreaded program as a set of non-blocking threads denoted by the vertices of a dag (*direct acyclic graph*), where the dag edges indicate dependencies between instructions. See Figure 2.1.

A correct execution of a program which uses *fork-join parallelism model* must meet all the dependencies in the dag, that is, a thread cannot be executed until all the depending treads have completed. The order in which these dependent threads will be executed on the processors is determined by the *scheduler*.

From a theoretical viewpoint, there are two natural measures that allow us to

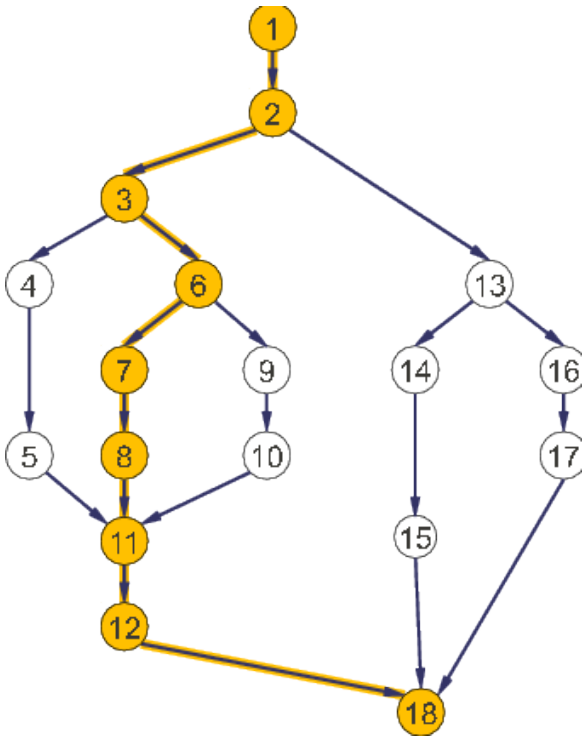


Figure 2.1: A directed acyclic graph (dag) representing the execution of a multithreaded program. Each vertex represents an instruction while each edge represents a dependency between instructions.

define parallelism precisely, as well as to provide important bounds on performance and speedup which are discussed in the following subsections.

2.1.1 The work law

The first important measure is the *work* which is defined as the total amount of time required to execute all the instructions of a given program. For instance, if each instruction requires a unit amount of time to execute, then the work for the example dag shown in Figure 2.1 is 18.

Let T_P be the fastest possible execution time of the application on P processors. Therefore, we denote the work by T_1 as it corresponds to the execution time on 1 processor. Moreover, we have the following relation

$$T_p \geq T_1/P \tag{2.1}$$

which is referred as the *work law*. In our simple theoretical model, the justification of this relation is easy: each processor executes at most 1 instruction per unit time and

therefore P processors can execute at most P instructions per unit time. Therefore, the *speedup* on P processors is at most P since we have

$$T_1/T_P \leq P. \tag{2.2}$$

2.1.2 The span law

The second important measure is based on the program's *critical-path length* denoted by T_∞. This is actually the execution time of the application on an infinite number of processors or, equivalently, the time needed to execute threads along the longest path of dependency. As a result, we have the following relation, called the *span law*:

$$T_P \geq T_\infty. \tag{2.3}$$

2.1.3 Parallelism

In the fork-join parallelism model, *parallelism* is defined as the ratio of work to span, or T_1/T_∞. Thus, it can be considered as the average amount of work along each point of the critical path. Specifically, the speedup for any number of processors cannot be greater than T_1/T_∞. Indeed, Equations 2.2 and 2.3 imply that speedup satisfies

$$T_1/T_P \leq T_1/T_\infty \leq P.$$

As an example, the parallelism of the dag shown in Figure 2.1 is $18/9 = 2$. This means that there is little chance for improving the parallelism on more than 2 processors, since additional processors will often starve for work and remain idle.

2.1.4 Performance bounds

For an application running on a parallel machine with P processors with work T_1 and span T_∞, the CilkPlus *work-stealing scheduler* achieves an expected running time as follows:

$$T_P = T_1/P + O(T_\infty), \tag{2.4}$$

under the following three hypotheses:

- each strand executes in unit time,
- for almost all parallel steps there are at least p strands to run,
- each processor is either working or stealing.

See [30] for details.

If the parallelism T_1/T_∞ is so large that it sufficiently exceeds P, that is $T_1/T_\infty \gg P$, or equivalently $T_1/P \gg T_\infty$, then from Equation (2.4) we have $T_P \approx T_1/P$. From this, we easily observe that the work-stealing scheduler achieves a nearly perfect linear speedup of $T_1/T_P \approx P$.

2.1.5 Work, span and parallelism of classical algorithms

The work, span and parallelism of some of the classical algorithms [22] in the fork-join parallelism model is shown in Table 2.1.

Algorithm	Work	Span	parallelism
Merge sort	$\Theta(n \log_2(n))$	$\Theta(\log_2(n)^3)$	$\Theta(\frac{n}{\log_2(n)^2})$
Matrix multiplication	$\Theta(n^3)$	$\Theta(\log_2(n))$	$\Theta(\frac{n^3}{\log_2(n)})$
Strassen	$\Theta(n^{\log_2(7)})$	$\Theta(\log_2(n)^2)$	$\Theta(\frac{n^{\log_2(7)}}{\log_2(n)^2})$
LU-decomposition	$\Theta(n^3)$	$\Theta(n \log_2(n))$	$\Theta(\frac{n^2}{\log_2(n)})$
Tableau construction	$\Theta(n^2)$	$\Omega(n^{\log_2(3)})$	$\Theta(n^{0.415})$
FFT	$\Theta(n \log_2(n))$	$\Theta(\log_2(n)^2)$	$\Theta(\frac{n}{\log_2(n)})$

Table 2.1: Work, span and parallelism of classical algorithms.

2.2 OpenMP

OpenMP [4] (Open Multi-Processing) is an API that supports shared memory multiprocessing programming in `C`, `C++` and `FORTRAN`, on most processor architectures and operating systems. It consists of a set of compiler directives, library routines and environment variables that influence run-time behavior. It supports both task parallelism and data parallelism.

It is an implementation of multithreading, a method of parallelizing whereby a master thread forks a specified number of slave threads and a task is divided among them. The threads then run concurrently, with the runtime environment allocating threads to different processors.

The section of code that is meant to run in parallel is marked accordingly, with a preprocessor directive that will cause the threads to form before the section is

executed. After the execution of the parallelized code, the threads join back into the master thread, which continues onward to the end of the program.

The OpenMP Architecture Review Board (ARB) published its first API specifications, OpenMP for `FORTRAN` 1.0 [10], in October 1997. The following year they released the `C/C++` standard [5]. 2000 saw version 2.0 [11] of the `FORTRAN` specifications with version 2.0 [6] of the `C/C++` specifications being released in 2002. Version 2.5 [12] is a combined `C/C++/FORTRAN` specification that was released in 2005. OpenMP continues to evolve, with new constructs and features being added over time. In May 2008, version 3.0 [7] was published with the new concept of *tasks*. Version 3.1 [8] was released in July 2011. The latest version 4.0 [9] which was released in July 2013 improving some of its compiler directives along with SIMD support and `FORTRAN` 2003 support.

In order to illustrate the OpenMP standard, a sample C program is used to show the parallelization methodology of the OpenMP directives in Example 1.

The following sections describe in detail the syntax and functionality of the OpenMP directives, runtime library routines and environment variables. The `#pragma` directive is the method specified by the C standard for providing additional information to the compiler, beyond what is conveyed in the language itself. To be recognized as an OpenMP directive, the `#pragma` directive is to be followed by the following sentinel : `omp`. After this sentinel is the type of the directive that is encountered (e.g `parallel,tasks,for,sections`).

2.2.1 OpenMP directives

The OpenMP directives in `C/C++` has the following format.

```
#pragma omp directive_name [clause,...]
```

The following sections describe the type and functionality of the OpenMP directives.

2.2.1.1 The `parallel` directive

A parallel region is a block of code that will be executed by multiple threads. This is the fundamental OpenMP parallel construct that starts parallel execution. These directives have the following format:

```
#pragma omp parallel [clause ...]  newline
```

When a thread reaches a `parallel` directive, it creates a team of threads and becomes the master of the team. The master thread has the thread number 0. The code is duplicated and all threads will execute that code defined in the parallel region. There is an implicit barrier at the end of a parallel section provided by the OpenMP compiler. Only the master thread continues execution past this point. If any thread terminates within a parallel region, all threads in the team will terminate, and the work done up until that point is undefined.

The `parallel` directive has eight optional clauses that take one or more arguments: `private`, `shared`, `default`, `firstprivate`, `reduction`, `if`, `copyin` and `num_threads`. These clauses are also called data-sharing attribute clauses. These clauses are used to explicitly define how variables should be scoped. They provide the ability to control the data environment during execution of parallel constructs. They define how and which data variables in the serial section of the program are transferred to the parallel sections of the program. They define which variables will be visible to all threads in the parallel sections and which variables will be privately allocated to all threads. for format and description of these clauses, see Section 2.2.5.

2.2.1.2 The `for` directive

The `for` directive specifies that the iterations of the loop immediately following it must be executed in parallel by the team. This assumes a parallel region has already been initiated, otherwise it executes serially. The `for` directive has the following format:

```
#pragma omp for [clause ...] newline
```

The `for` directive has nine optional clauses that take one or more arguments: `private`, `shared`, `firstprivate`, `reduction`, `schedule`, `ordered`, `collapse`, `nowait`. The format and description of these clauses, see Section 2.2.5.

The way iterations of the loop are divided among the threads in the team are defined by `schedule` clause. The default schedule is implementation dependent. The `schedule` format is shown below:

```
#pragma omp for schedule (type [,chunk])
```

There are five types of scheduling options in the schedule clause: `static`, `dynamic`, `guided`, `runtime` and `auto`. In `static`, loop iterations are divided into pieces of size chunk and then statically assigned to threads. If chunk is not specified, the iterations

are evenly (if possible) divided contiguously among the threads. In `dynamic`, loop iterations are divided into pieces of size chunk, and dynamically scheduled among the threads; when a thread finishes one chunk, it is dynamically assigned another. The default chunk size is 1. In `guided`, loop iterations are dynamically assigned to threads in blocks as threads request them until no blocks remain to be assigned. Similar to dynamic except that the block size decreases each time a parcel of work is given to a thread. The size of the initial block is proportional to:

```
number_of_iterations / number_of_threads
```

Subsequent blocks are proportional to:

```
number_of_iterations_remaining / number_of_threads
```

The chunk parameter defines the minimum block size. The default chunk size is 1. In `runtime`, the scheduling decision is deferred until runtime by the environment variable omp_schedule. It is illegal to specify a chunk size for this clause. In `auto`, the scheduling decision is delegated to the compiler and/or runtime system.

If `nowait` clause is specified, then threads do not synchronize at the end of the parallel loop. The `ordered` clause specifies that the iterations of the loop must be executed as they would be in a serial program. The `collapse` clause specifies how many loops in a nested loop should be collapsed into one large iteration space and divided according to the schedule clause. The sequential execution of the iterations in all associated loops determines the order of the iterations in the collapsed iteration space. Example 1 gives an idea of OpenMP for loops.

Example 1.
```
main ()  {
  int i, chunk, CHUNKSIZE = 10, N = 100;
  float a[N], b[N], c[N];

  for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;
  chunk = CHUNKSIZE;

  #pragma omp for private(i) schedule(static,chunk)
  {
    for (i=0; i < n; i++)
    c[i] = a[i] + b[i];
  }
}
```

2.2.1.3 The `sections` directive

The `sections` directive is a non-iterative work-sharing construct. It specifies that the enclosed section/sections of code are to be divided among the threads in the team. Independent `section` directives are nested within a `sections` directive. Each `section` is executed once by a thread in the team. Different sections may be executed by different threads. It is possible for a thread to execute more than one section if it is quick enough and the implementation permits such. The format of this directive is as follows:

```
#pragma omp sections [clause ...]  newline
{
#pragma omp section   newline

      structured_block

   #pragma omp section   newline

      structured_block
}
```

There are five clauses: `private`, `firstprivate`, `lastprivate`, `reduction`, `nowait`. There is an implied barrier at the end of a `sections` directive, unless the `nowait` clause is used. for format and description of these clauses, see Section 2.2.5. Example 2 gives an example of OpenMP sections.

Example 2.

```
main () {
int i,N = 100;
float a[N], b[N], c[N], d[N];

for (i=0; i < N; i++) {
  a[i] = i * 1.5;
  b[i] = i + 22.35;
  }

#pragma omp parallel
{
```

```
 #pragma omp sections
 {
  #pragma omp section
    for (i=0; i < N; i++)
      c[i] = a[i] + b[i];

   #pragma omp section
    for (i=0; i < N; i++)
      d[i] = a[i] * b[i];
     }  /* end of sections */
   }  /* end of parallel section */
}
```

2.2.1.4 The `single` directive

The `single` directive specifies that the enclosed code is to be executed by only one thread in the team. This directive is useful when dealing with sections of code that are not thread safe. Threads in the team that do not execute the `single` directive, wait at the end of the enclosed code block, unless a `nowait` clause is specified. The format of this directive is as follows:

```
#pragma omp single [clause ...]  newline
```

This directive has two clauses: `private` and `firstprivate`. For the format and description of these clauses, see Section 2.2.5.

2.2.2 The `task` directive

This is a new construct with OpenMP version 3.0. The `task` construct defines an explicit task, which may be executed by the encountering thread, or deferred for execution by any other thread in the team. The data environment of the task is determined by the data sharing attribute clauses. The execution of task is subject to task scheduling. The format of this directive is as follows:

```
#pragma omp task [clause[[,] clause] ...] newline
```

`task` directive has six clauses: `if`, `untied`, `default`, `private`, `firstprivate` and `shared`. For the format and description of these clauses, see Section 2.2.5.

When a thread encounters a task construct, a task is generated from the code for the associated structured block. The data environment of the task is created according to the data-sharing attribute clauses on the task construct and any defaults that apply. The encountering thread may immediately execute the task, or defer its execution. In the latter case, any thread in the team may be assigned the task. Completion of the task can be guaranteed using task synchronization constructs. A task construct may be nested inside an outer task, but the task region of the inner task is not a part of the task region of the outer task. A thread that encounters a task scheduling point within the task region may temporarily suspend the task region. By default, a task is tied and its suspended task region can only be resumed by the thread that started its execution. If the `untied` clause is present on a task construct, any thread in the team can resume the task region after a suspension. For the format and description of these clauses, see Section 2.2.5. Example 3 gives an idea on OpenMP `task`s.

Example 3.
```
main () {
  int x;
  #pragma omp parallel
  #pragma omp task
 {
    x = some_func();
 }
 #pragma omp taskwait
}
```

2.2.3 Task scheduling

Whenever a thread reaches a task scheduling point, the implementation may cause it to perform a task switch, beginning or resuming execution of a different task bound to the current team. *Task scheduling* points are implied at a point immediately following the generation of an explicit task or after the last instruction of a task region or in taskwait 2.2.4.4 regions or in implicit and explicit barrier 2.2.4.3 regions.

When a thread encounters a task scheduling point it might begin execution of a tied task bound to the current team or resume any suspended task region, bound to the current team, to which it is tied or begin execution of an untied task bound to the current team or resume any suspended untied task region bound to the current team.

2.2.4 Synchronization constructs

2.2.4.1 The `master` Directive

The `master` directive specifies a region that is to be executed only by the master thread of the team. All other threads on the team skip this section of code. There is no implied barrier associated with this directive. The format of this diretive is as follows:

```
#pragma omp master  newline

    structured_block
```

2.2.4.2 The `critical` directive

The `critical` directive specifies a region of code that must be executed by only one thread at a time. If a thread is currently executing inside a `critical` region and another thread reaches that `critical` region and attempts to execute it, it will block until the first thread exits that `critical` region. The format of this directive is as follows:

```
#pragma omp critical [ name ]  newline

    structured_block
```

The optional name enables multiple different `critical` regions to exist.

2.2.4.3 The `barrier` directive

The `barrier` directive synchronizes all threads in the team. When a `barrier` directive is reached, a thread will wait at that point until all other threads have reached that barrier. All threads then resume executing in parallel the code that follows the barrier. All threads in a team (or none) must execute the `barrier` region. The format of this directive is as follows:

```
#pragma omp barrier  newline
```

2.2.4.4 The `taskwait` directive

This directive is new with OpenMP version 3.0. The `taskwait` construct specifies a wait on the completion of child tasks generated since the beginning of the current task. The format of this directive is as follows:

```
#pragma omp taskwait  newline
```

2.2.4.5 The `atomic` directive

The `atomic` directive specifies that a specific memory location must be updated atomically, rather than letting multiple threads attempt to write to it.The directive applies only to a single, immediately following statement. The format of this directive is as follows:

```
#pragma omp taskwait  newline

statement_expression
```

2.2.4.6 The `flush` directive

The `flush` directive identifies a synchronization point at which the implementation must provide a consistent view of memory. Thread-visible variables are written back to memory at this point.

```
#pragma omp flush (list) newline
```

2.2.4.7 The `ordered` directive

The `ordered` directive specifies that iterations of the enclosed loop will be executed in the same order as if they were executed on a serial processor. Threads will need to wait before executing their chunk of iterations if previous iterations haven't completed yet. It is used within a `for` loop with an `ordered` clause. Only one thread is allowed in an ordered section at any time. An iteration of a loop must not execute the same `ordered` directive more than once, and it must not execute more than one `ordered` directive. The format of this directive is as follows:

```
#pragma omp for ordered [clauses...]
    (loop region)
```

```
#pragma omp ordered  newline

    structured_block

    (endo of loop region)
```

2.2.4.8 The `threadprivate` directive

The `threadprivate` directive is used to make global file scope variables (C/C++) or common blocks (fortran) local and persistent to a thread through the execution of multiple parallel regions. The format of this directive is as follows:

```
#pragma omp threadprivate (list)
```

2.2.5 Data-Sharing attribute clauses

Some directives accept clauses that follow a user to control the scope attributes of variables for the duration of the construct. Not all of the following clauses are allowed on all directives, but the clauses that are valid on a particular directive are included with the description of the directive. The following sections describe the data scope attribute clauses.

2.2.5.1 The `private` clause

The `private` clause declares variables in its list to be private to each thread. This clause has the following format:

```
private (list)
```

2.2.5.2 The `shared` clause

The `shared` clause declares variables in its list to be shared among all threads in the team. This clause has the following format:

```
shared (list)
```

2.2.5.3 The `default` clause

The `default` clause allows the user to specify a default scope for all variables in the lexical extent of any parallel region. This clause has the following format:

```
default (shared | none)
```

2.2.5.4 The `firstprivate` clause

The `firstprivate` clause combines the behavior of the private clause with automatic initialization of the variables in its list. This clause has the following format:

```
firstprivate (list)
```

2.2.5.5 The `lastprivate` clause

The `lastprivate` clause combines the behavior of the `private` clause with a copy from the last loop iteration or section to the original variable object. This clause has the following format:

```
lastprivate (list)
```

2.2.5.6 The `copyin` clause

The `copyin` clause provides a means for assigning the same value to `threadprivate` variables for all threads in the team. This clause has the following format:

```
copyin (list)
```

2.2.5.7 The `copyprivate` clause

The `copyprivate` clause can be used to broadcast values acquired by a single thread directly to all instances of the private variables in the other threads. This clause has the following format:

```
copyprivate (list)
```

2.2.5.8 The `reduction` clause

The `reduction` clause performs a reduction on the variables that appear in its list. A private copy for each list variable is created for each thread. At the end of the reduction, the reduction variable is applied to all private copies of the shared variable, and the final result is written to the global shared variable. This clause has the following format:

```
reduction (operator : list)
```

For the OpenMP runtime routines and environment variables, see references [7, 8, 9] .

2.3 CilkPlus

Cilk [23] has been developed since 1994 at the MIT Laboratory for Computer Science by Prof. Charles E. Leiserson and his group, in particular by Matteo Frigo. Over the years, the implementations of Cilk have run on computers ranging from networks of Linux laptops to an 1824-nodes Intel Paragon. From 2007 to 2009 Cilk has lead to `Cilk++`, developed by Cilk Arts, an MIT spin-off, which was acquired by Intel in July 2009 and became CilkPlus.

CilkPlus [24, 17, 30, 33, 28] is a small set of linguistic extensions to C and C++ supporting fork-join parallelism. It has a provably efficient work-stealing scheduler. It provides a hyperobject library for parallelizing code with global variables and performing reduction for data aggregation. CilkPlus includes the Cilkscreen race detector and the Cilkview performance analyzer.

2.3.1 CilkPlus scheduling

The CilkPlus randomized work-stealing scheduler [30] load-balances the computation at run-time. Each processor maintains a ready deque. A ready deque is a double ended queue, where each entry is a procedure instance that is ready to execute. Adding a procedure instance to the bottom of the deque represents a procedure call being spawned. A procedure instance being deleted from the bottom of the deque represents the processor beginning/resuming execution on that procedure. Deletion from the top of the deque corresponds to that procedure instance being stolen.

2.3.2 CilkPlus keywords

The following are the CilkPlus keywords.

1. cilk_for
2. cilk_spawn
3. cilk_sync

2.3.2.1 cilk_for

A `cilk_for` loop is a replacement for the normal C/C++ for loop that permits loop iterations to run in parallel. The general `cilk_for` syntax is:

```
cilk_for (declaration  and initialization;
        conditional expression;
        increment expression)
 body
```

Since the loop body is executed in parallel, it must not modify the control variable nor should it modify a non-local variable, as that would cause a data race. A reducer can often be used to prevent a race. The `cilk_for` statement divides the loop into chunks containing one or more loop iterations. Each chunk is executed serially, and is spawned as a chunk during the execution of the loop. The maximum number of iterations in each chunk is the grain size. The pragma used to specify the grain size for one `cilk_for` loop is:

```
#pragma cilk grainsize = expression
```

Example 4 gives an example for CilkPlus for loops.

Example 4.
```
void function() {
   int i = 0, j = 0;
   cilk_for ( int j = 0; j < ny; j++ )
   {
      int i;
      for ( i = 0; i < nx; i++ )
      {
```

```
            u[i][j] = unew[i][j];
        }
    }
}
```

2.3.2.2 cilk_spawn

The `cilk_spawn` keyword modifies a function call statement to tell the runtime system that the function may (but is not required to) run in parallel with the caller. This can take one of the following forms:

```
1. var = cilk_spawn func(args); // func() returns a value
2. cilk_spawn func(args); // func() may return void
```

"func" is the name of a function which may run in parallel with the current strand. A strand is a serial sequence of instructions without any parallel control. The code following the `cilk_spawn` in the current routine can execute in parallel with func. "var" is a variable that can be initialized or assigned from the return value of func. args are the arguments to the function being spawned. These arguments are evaluated before the spawn takes effect. Example 5 gives an idea on spawning a function in CilkPlus.

Example 5.
```
void test() {
   int x, y;
   x = cilk_spawn test1();
   y = cilk_spawn test2();
   cilk_sync;
}
```

2.3.2.3 cilk_sync

This specifies that all spawned calls in a function must complete before execution continues. There is an implied `cilk_sync` at the end of every function that contains a cilk_spawn. In other words, it indicates that the current function cannot continue in parallel with its spawned children. After the children all complete, the current function can continue. The syntax is as follows:

```
cilk_sync;
```

`cilk_sync` only syncs with children spawned by this function. Children of other functions are not affected.

2.4 Compilation theory

A *compiler* [2, 35] is a program that can read a program in one language which is called the *source language* and translate it into an equivalent program in another language called the *target language*. For example, a C compiler translates a program written in C language into a program in assembly language.

The compilation process [2, 35] can be divided in two parts: *analysis part* and *synthesis part*. In the *analysis part*, the source program is analyzed for its syntactical and semantic correctness and transformed into an *intermediate representation* (IR). In the *synthesis part*, the IR is used to do optimizations and generate the *target program*. The *analysis part* is also called the *front end* of the compiler, while the *synthesis part* is called the *back end* of the compiler.

To give more details, the compilation process [2, 35] can also be further divided into several phases. The input of each phase is the output of the previous phase. The initial input is the text of a program which is to be compiled. The internal output of each phase is a form of *Intermediate Representation* (IR) such as *Abstract Syntax Tree* (AST). The final output is the translated program that is written in another language.

Usually, the compilation process is divided into the following phases. A brief introduction about the functionality of each phase is given below. Refer [2, 35] for more detailed information.

- **Lexical analysis :** The input of the lexical analysis is the stream of characters of a program written in the source language. The output is a stream of tokens.
- **Syntax analysis :** The input of syntax analysis is the stream of tokens. The output is the IR of the program in the form of an AST.
- **Semantic analysis :** The semantic analysis phase checks the semantic correctness regarding to the language definition by analyzing the IR.
- **Intermediate code generation :** After semantic analysis, a compiler usually generates a lower level IR of the program from the higher level IR such as AST. There could be several layers of IR, such as high-level IR, middle-level IR and low-level IR.
- **Code optimization :** The optimization phase performs different kinds of optimizations at the suitable layer of IR.

- **Code generation :** In the code generation phase, the input is the IR. The output is the code that is generated from the IR using the target language.

2.4.1 Source-to-source compiler

A *source-to-source* compiler refers to translating a program written in one programming language to another programming language that are both at approximately the same abstraction level. For example, unoptimized C++ to optimized C++ [25], C++ to CUDA [1, 31] (Compute Unified Device Architecture) etc. The traditional compiler usually translates from a high level programming language to a low level programming language that are at different abstraction levels, for example, from C/C++ to assembly language.

A main difference between a traditional compiler and a source to source compiler is the IR. In a traditional compiler, the higher level IR is transformed down to several lower level IRs until it is suitable to generate the target code. In a source to source compiler, since both the source language and target language are at the same abstraction level, the difficulty is to find the best level of IR that could be used to translate from source language to target language. Using the lowest level IR or the assembly language style IR, is possible for source to source translation. However, the generated target program is likely to look very different from the source program and to be difficult to understand.

2.4.2 Basic block

A basic block is a portion of the code within a program with only one entry point and only one exit point. This makes a basic block highly amenable to analysis. Compilers usually decompose programs into their basic blocks as a first step in the analysis process. Thus the code in a basic block has:

- one entry point, meaning no code within it is the destination of a jump instruction anywhere in the program,

- one exit point, meaning only the last instruction can cause the program to begin executing code in a different basic block.

2.4.3 Lex

Lex [2, 35] is a program that generates lexical analyzers and is written by Eric Schmidt and Mike Lesk. Lex is commonly used with the YACC (`Yet Another Compiler`

`Compiler`) [32] parser generator. It reads an input stream specifying the lexical analyzer and outputs source code implementing the lexer in the C programming language. The structure of a Lex file is divided into three sections, separated by lines that contain only two percent signs, as follows:

```
Definition section
%%
Rules section
%%
C code section
```

- The definition section defines macros and imports header files written in C. It is also possible to write any C code here, which will be copied verbatim into the generated source file.
- The rules section associates regular expression patterns with C statements. When the lexer sees text in the input matching a given pattern, it will execute the associated C code.
- The C code section contains C statements and functions that are copied verbatim to the generated source file. These statements presumably contain code called by the rules in the rules section. In large programs it is more convenient to place this code in a separate file linked in at compile time.

2.4.4 Lex **theory**

During the first phase, the compiler reads the input and converts strings in the source to tokens. With regular expressions we can specify patterns to Lex so it can generate code that will allow it to scan and match strings in the input. Each pattern in the input to Lex has an associated action. Typically an action returns a token that represents the matched string for subsequent use by the parser. Initially we will simply print the matched string rather than return a token value.

The following represents a simple pattern, composed of a regular expression, that scans for identifiers. Lex will read this pattern and produce C code for a lexical analyzer that scans for identifiers.

```
letter(letter|digit)*
```

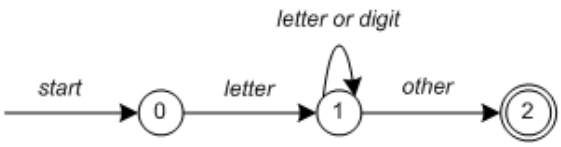


Figure 2.2: Finite state automaton represented using states.

This pattern matches a string of characters that begins with a single letter followed by zero or more letters or digits. This example nicely illustrates operations allowed in regular expressions:

- Repetition, expressed by the "*" operator
- Alternation, expressed by the "—" operator
- Concatenation

Any regular expression expressions may be expressed as a `finite state automaton` (FSA). We can represent FSA using states, and transitions between states. There is one start state and one or more final or accepting states.

In the above figure, state 0 is the start state and state 2 is the accepting state. As characters are read we make a transition from one state to another. When the first letter is read we transition to state 1. We remain in state 1 as more letters or digits are read. When we read a character other than a letter or digit we transition to accepting state 2. Any FSA may be expressed as a computer program. for example, our 3-state machine is easily programmed:

```
start:  goto state0

state0: read c
        if c = letter goto state1
        goto state0

state1: read c
        if c = letter goto state1
        if c = digit goto state1
        goto state2

state2: accept string
```

This is the technique used by Lex. Regular expressions are translated by Lex to a computer program that mimics an FSA. Using the next input character and current state the next state is easily determined by indexing into a computer-generated state table.

2.4.5 YACC

YACC is an acronym for `Yet Another Compiler Compiler`. YACC [2, 35, 32] reads the grammar and generate C code for a parser . It's grammars written in `Backus Naur form` (BNF). BNF grammar are used to express context-free languages. The structure of a YACC file is similar to Lex and is divided into three sections, separated by lines that contain only two percent signs, as follows:

```
Definition section
%%
Rules section
%%
C code section
```

2.4.6 YACC theory

Grammars for YACC are described using a variant of Backus Naur form (BNF). A BNF grammar can be used to express context-free languages. Most constructs in modern programming languages can be represented in BNF. For example, the grammar for an expression that multiplies and adds numbers is:

```
1    E -> E + E
2    E -> E * E
3    E -> id
```

Three productions have been specified. Terms that appear on the left-hand side (lhs) of a production, such as E, are nonterminals. Terms such as id (identifier) are terminals (tokens returned by Lex) and only appear on the right-hand side (rhs) of a production. This grammar specifies that an expression may be the sum of two expressions, the product of two expressions, or an identifier. We can use this grammar to generate expressions:

```
E -> E * E            (r2)
  -> E * z            (r3)
```

```
-> E + E * z          (r1)
-> E + y * z          (r3)
-> x + y * z          (r3)
```

At each step we expanded a term and replace the lhs of a production with the corresponding rhs. The numbers on the right indicate which rule applied. To parse an expression we a need to do the reverse operation. Instead of starting with a single nonterminal (start symbol) and generating an expression from a grammar we need to reduce an expression to a single nonterminal. This is known as bottom-up or shift-reduce parsing and uses a stack for storing terms. Here is the same derivation but in reverse order:

```
 1   . x + y * z     shift
 2   x . + y * z     reduce(r3)
 3   E . + y * z     shift
 4   E + . y * z     shift
 5   E + y . * z     reduce(r3)
 6   E + E . * z     shift
 7   E + E * . z     shift
 8   E + E * z .     reduce(r3)
 9   E + E * E .     reduce(r2)   emit multiply
10   E + E .         reduce(r1)   emit add
11   E .             accept
```

Terms to the left of the dot are on the stack while remaining input is to the right of the dot. We start by shifting tokens onto the stack. When the top of the stack matches the rhs of a production we replace the matched tokens on the stack with the lhs of the production.The matched tokens are known as a handle and we are reducing the handle to the lhs of the production. This process continues until we have shifted all input to the stack and only the starting nonterminal remains on the stack.

In step 1 we shift the x to the stack. Step 2 applies rule r3 to the stack to change x to E. We continue shifting and reducing until a single nonterminal, the start symbol, remains in the stack. In step 9, when we reduce rule r2, we emit the multiply instruction. Similarly the add instruction is emitted in step 10. Consequently multiply has a higher precedence than addition.

Consider the shift at step 6. Instead of shifting we could have reduced and applied rule r1. This would result in addition having a higher precedence than multiplication.

This is known as a shift-reduce conflict. Our grammar is ambiguous because there is more than one possible derivation that will yield the expression. In this case operator precedence is affected. As another example, associativity in the rule:

```
E -> E + E
```

is ambiguous, for we may recurse on the left or the right. To remedy the situation, we could rewrite the grammar or supply YACC with directives that indicate which operator has precedence.

The following grammar has a reduce-reduce conflict. With an id on the stack we may reduce to T or E.

```
E -> T
E -> id
T -> id
```

YACC takes a default action when there is a conflict. For shift-reduce conflicts YACC will shift. For reduce-reduce conflicts it will use the first rule in the listing. It also issues a warning message whenever a conflict exists. The warnings may be suppressed by making the grammar unambiguous.

2.4.7 Lvalues and Rvalues

An *object* is a region of storage that can be examined and stored into. An *lvalue* is an expression that refers to such an object. An lvalue does not necessarily permit modification of the object it designates. For example, a const object is an lvalue that cannot be modified. The term modifiable lvalue is used to emphasize that the lvalue allows the designated object to be changed as well as examined.

The term *rvalue* refers to a data value that is stored at some address in memory. Typical rvalues are function calls or arithmetic expressions. An rvalue is an expression that cannot have a value assigned to it. Both a literal constant and a variable can serve as an rvalue. When an lvalue appears in a context that requires an rvalue, the lvalue is implicitly converted to an rvalue. The reverse, however, is not true: an rvalue cannot be converted to an lvalue. Rvalues always have complete types or the void type.

Chapter 3

MetaFork: A metalanguage for concurrency platforms targeting multicores

This chapter focuses on specifying the syntax and semantics of the parallel constructs of MetaFork. Indeed, MetaFork differs only from the C language by its parallel constructs. Specifying the semantics of the parallel constructs of MetaFork is equivalent to define the serial C elision of a MetaFork program. That is, given a program $\mathcal{M}$ written in MetaFork define a program $\mathcal{C}$ written in the C language and such that $\mathcal{M}$ and $\mathcal{C}$ compute exactly the same thing.

While defining the serial C elision of a CilkPlus program is easy, this is much harder in the case of a MetaFork program. This difficulty is caused by the spawning of parallel regions that are not function calls. As far as we know, and after verifying with OpenMP developers, no formal definition of the serial C elision of OpenMP was ever given in the literature.

The algorithms that we state in Section 3.4 define the serial C elision of a MetaFork program. Consequently, they define the serial C elision of an OpenMP Tasks program, since MetaFork is a super-set of Tasks flavor in OpenMP.

To help the reader understanding those algorithms and the related notions, Sections 3.5, 3.6, 4.1.1 and 3.8 provide various examples of code translation between CilkPlus, OpenMP and MetaFork.

This chapter is a joint work with Xiaohui Chen and Dr.Marc Moreno Maza.

3.1 Basic principles and execution model

We summarize in this section a few principles that guided the design of MetaFork. First of all, MetaFork is an extension of the C language and a multithreaded language based on the fork-join parallelism model. Thus, concurrent execution is obtained by a parent thread creating and launching one or more child threads so that, the parent and its children execute a so-called *parallel region*. An important example of parallel regions are for-loop bodies. MetaFork has the following natural requirement regarding parallel regions: control flow cannot branche into or out of a *parallel region*.

Similarly to CilkPlus, the parallel constructs of MetaFork grant permission for concurrent execution but do not command it. Hence, a MetaFork program can execute on a single core machine.

As mentioned above, MetaFork does not make any assumptions about the runtime system, in particular about task scheduling. Another design intention is to encourage a programming style limiting thread communication to a minimum so as to

- prevent data-races while preserving a satisfactory level of expressiveness and,
- minimize parallelism overheads.

This principle is similar to one of CUDA's principles [37] which states that the execution of a given kernel should be independent of the order in which its thread blocks are executed.

To understand the implication of that idea in MetaFork, let us return to our concurrency platforms targeting multicore architectures: OpenMP offers several clauses which can be used to exchange information between threads (like `threadprivate`, `copyin` and `copyprivate`) while no such mechanism exists in CilkPlus. Of course, this difference follows from the fact that, in CilkPlus, one can only fork a function call while OpenMP allows other code regions to be executed concurrently. MetaFork has both types of parallel constructs. But, for the latter, MetaFork does not offer counterparts to the above OpenMP data attribute clauses. Data attributes in MetaFork are discussed in Section 3.3.

3.2 Parallel constructs of MetaFork

MetaFork has four parallel constructs: function call spawn, block spawn, parallel for-loop and synchronization barrier. The first two use the keyword `meta_fork` while

the other two use respectively the keywords `meta_for` and `meta_join`. We stress the fact that `meta_fork` allows the programmer to spawn a function call (like in CilkPlus) as well as a block (like in OpenMP).

3.2.1 Spawning a function call or a block with `meta_fork`

As mentioned above, the `meta_fork` keyword is used to express the fact that a function call or a block is executed by a child thread, concurrently to the execution of the parent thread. If the program is run by a single processor, the parent thread is suspended during the execution of the child thread; when this latter terminates, the parent thread resumes its execution after the function call (or block) spawn.

If the program is run by multiple processors, the parent thread may continue its execution after the function call (or block) spawn, without being suspended, meanwhile the child thread is executing the function call (or block) spawn. In this latter scenario, the parent thread waits for the completion of the execution of the child thread, as soon as the parent thread reaches a synchronization point.

Spawning a function call, for the function `f` called on the argument sequence `args` is done by

```
meta_fork f(args)
```

The semantics is similar to those of the CilkPlus counterpart

```
cilk_spawn f(args)
```

In particular, all the arguments in the argument sequence `args` are evaluated before spawning the function call `f(args)`. However, the execution of `meta_fork f(args)` differs from that of `cilk_spawn f(args)` on one aspect: none of the `return` statements of the function `f` assumes an implicit barrier. This design decision is motivated by the fact that, in addition to fork-join parallelism, the MetaFork language may be extended to other forms of parallelism such as *parallel futures* [39, 16].

Figure 3.1 illustrates how `meta_fork` can be used to define a function spawn. The underlying algorithm in this example is the classical divide and conquer *quicksort* procedure.

The other usage of the `meta_fork` construct is for spawning a basic block `B`, which is done as follows:

```
meta_fork { B }
```

```
#include <algorithm>
#include <iostream>
using namespace std;
void parallel_qsort(int * begin, int * end)
{
        if (begin != end) {
        --end;  // Exclude last element (pivot) from partition
        int * middle = std::partition(begin, end,
                            std::bind2nd(std::less<int>(), *end));
        using std::swap;
        swap(*end, *middle);    // move pivot to middle
        meta_fork parallel_qsort(begin, middle);
        parallel_qsort(++middle, ++end); // Exclude pivot and restore end
        meta_join;
    }
}
int main(int argc, char* argv[])
{
        int n = 10;
        int *a = (int *)malloc(sizeof(int)*n);
        srand( (unsigned)time( NULL ) );
        for (int i = 0; i < n; ++i)
              a[i] = rand();
        parallel_qsort(a, a + n);
        return 0;
}
```

Figure 3.1: Example of a MetaFork program with a function spawn.

Note that if `B` consists of a single instruction, then the surrounding curly braces can be omitted.

We also refer to the above as a *parallel region*. There is no equivalent in CilkPlus while it is offered by OpenMP. Similarly to a function call spawn, no implicit barrier is assumed at the end of the parallel region. Hence synchronization points have to be added explicitly, using `meta_join`; see the examples of Figure 3.2.

A variable `v` which is not local to `B` may be either shared by both the parent and child threads; alternatively, the child thread may be granted a private copy of `v`. Precise rules about data attributes, for both parallel regions and parallel for-loops, are stated in Section 3.3.

Figure 3.2 below illustrates how `meta_fork` can be used to define a parallel region.

```
long int parallel_scanup (long int x [], long int t [], int i, int j)
{
        if (i == j) {
                return x[i];
        }
        else{
                int k = (i + j)/2;
                int right;
                meta_fork {
                        t[k] = parallel_scanup(x,t,i,k);
                }
                right = parallel_scanup (x,t, k+1, j);
                meta_join;
                return t[k] + right;}
}
```

Figure 3.2: Example of a MetaFork program with a parallel region.

The underlying algorithm is one of the two subroutines in the work-efficient parallel prefix sum due to Guy Blelloch [15].

3.2.2 Parallel for-loops with `meta_for`

Parallel for-loops in MetaFork have the following format

```
meta_for (I, C, S) { B }
```

where `I` is the *initialization expression* of the loop, `C` is the *condition expression* of the loop, `S` is the *stride* of the loop and `B` is the loop body. In addition:

- the initialization expression initializes a variable, called the *control variable* which can be of type integer or pointer,
- the condition expression compares the control variable with a compatible expression, using one of the relational operators `<, <=, >, >=, !=`,
- the stride uses one the unary operators `++, --, +=, -+` (or a statement of the form `cv = cv + incr` where `incr` evaluates to a compatible expression) in order to increase or decrease the value of the control variable `cv`,
- if `B` consists of a single instruction, then the surrounding curly braces can be omitted.

```
template<typename T> void multiply_iter_par(int ii, int jj, int kk, T* A, T* B,
    T* C)
{
    meta_for(int i = 0; i < ii; ++i)
        for (int k = 0; k < kk; ++k)
            for(int j = 0; j < jj; ++j)
                C[i * jj + j] += A[i * kk + k] + B[k * jj + j];
}
```

Figure 3.3: Example of `meta_for` loop.

An implicit synchronization point is assumed after the loop body, That is, the execution of the parent thread is suspended when it reaches `meta_for` and resumes when all children threads (executing the loop body iterations) have completed their execution.

As one can expect, the iterations of the parallel loop `meta_for (I, C, S) B` must execute independently from each other in order to guarantee that this parallel loop is semantically equivalent to its serial version `for (I, C, S) B` .

A variable `v` which is not local to `B` may be either shared by both the parent and the children threads, or each child may be granted a private copy of `v`. Precise rules about data attributes, for both parallel regions and parallel for-loops, are stated in Section 3.3.

Figure 3.3 displays an example of `meta_for` loop, where the underlying algorithm is the naive (and cache-inefficient) matrix multiplication procedure.

3.2.3 Synchronization point with `meta_join`

The construct `meta_join` indicates a *synchronization point* (or *barrier*) for a parent thread and its children tasks. More precisely, a parent thread reaching this point must wait for the completion of the its children tasks but not for those of the subsequent descendant tasks.

3.3 Data attribute rules

Let B be the block of a parallel region or the body of a parallel for-loop. We specify hereafter the attribute rules for the variables that are non-local to B. We start by reviewing a few standard definitions.

3.3.1 Terminology

3.3.1.1 Notions of shared and private variables

Consider a parallel region with block Y (resp. a parallel for-loop with loop body Y). We denote by X the immediate outer scope of Y. We say that X is the *parent region* of Y and that Y is a *child region* of X.

A variable v which is defined in Y is said to be *local* to Y; otherwise we call v a *non-local* variable for Y. Let v be a non-local variable for Y. Assume v gives access to a block of storage before reaching Y. (Thus, v cannot be a non-initialized pointer.) We say that v is *shared* by X and Y if its name gives access to the same block of storage in both X and Y; otherwise we say that v is *private* to Y.

If Y is a parallel for-loop, we say that a local variable w is *shared* within Y whenever the name of w gives access to the same block of storage in any loop iteration of Y; otherwise we say that w is *private* within Y.

3.3.1.2 Notions of value-type and reference-type variables

In the C programming language, a *value-type variable* contains its data directly as opposed to a *reference-type variable*, which contains a reference to its data. Value-type variables are either of primitive types (`char`, `float`, `int`, `double` and `void`) or user-defined types (`enum`, `struct` and `union`). Reference-type variables are pointers, arrays and functions.

3.3.1.3 Notions of `static` and `const` type variables

In the C programming language, a *static* variable is a variable that has been allocated statically and whose lifetime extends across the entire run of the program. This is in contrast to

- *automatic* variables (*local* variables are generally *automatic*), whose storage is allocated and deallocated on the call stack and,
- other variables (such as objects) whose storage is dynamically allocated in heap memory.

When a variable is declared with the qualifier *const*, the value of that variable cannot typically be altered by the program during its execution.

3.3.2 Data attribute rules of meta_fork

A non-local variable v which gives access to a block of storage before reaching Y is shared between the parent X and the child Y whenever v is

(1) a global variable,
(2) a file scope variable,
(3) a reference-type variable,
(4) declared `static` or `const`,
(5) qualified `shared`.

In all other cases, the variable v is private to the child. In particular, value-type variables (that are not declared `static` or `const`, or qualified `shared` and, that are not global variables or file scope variables) are private to the child.

3.3.3 Data attribute rules of meta_for

A non-local variable which gives access to a block of storage before reaching Y is *shared between parent and child*.

A variable local to Y is *shared within* Y whenever it is declared `static`, otherwise it is private within Y. In particular, loop control variables are private within Y.

3.3.4 Examples

In the example of Figure 3.4, the variables `array`, `basecase`, `var` and `var1`, are shared by all threads while the variables `i` and `j` are private.

In the example of Figure 3.5, the variable `n` is private to `fib_parallel(n-1)`. In Figure 3.6, we specify the variable `x` as shared and the variable `n` is still private. Notice that the programs in Figures 3.5 and 3.6 are equivalent, in the sense that they compute the same thing.

In Figure 3.7, the variables `a`, `c`, `d`, `f` and `g` are shared, meanwhile the variable `b` and `e` are still private.

```
/* To illustrate file scope
variable, 3 files
(incl. a headerfile) are used.
This file is a.cpp  */
#include<stdio.h>
extern int var;
void test(int *array)
{
  int basecase = 100;
  meta_for(int j = 0; j < 10; j++)
  {
          static int var1=0;
          int i = array[j];
          if( i < basecase )
                  array[j]+=var;
  }
}
```

```
/* This file is b.cpp*/
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include"a.h"
int var = 100;
int main(int argc, char **argv)
{
  int *a=(int*)malloc(sizeof(int)*10);
  srand((unsigned)time(NULL));
  for(int i=0;i<10;i++)
    a[i]=rand();
  test(a);
  return 0;
}
/* This file is a.h*/
void test(int *a);
```

Figure 3.4: Example of shared and private variables with `meta_for`.

```
long fib_parallel(long n)
{
        long x, y;
        if (n < 2)
                return n;
        else{
                 x = meta_fork fib_parallel(n-1);
                 y = fib_parallel(n-2);
                 meta_join;
                 return (x+y);}
}
```

Figure 3.5: Example of private variables in a function spawn.

```
long fib_parallel(long n)
{
        long x, y;
        if (n < 2)
                return n;
        else{
                 meta_fork shared(x)
                 {
                        x = fib_parallel(n-1);
                 }
                 y = fib_parallel(n-2);
                 meta_join;
                 return (x+y);}
}
```

Figure 3.6: Example of a shared variable attribute in a parallel region.

```
#include<stdio.h>
#include<time.h>
#include<stdlib.h>
int a;
void subcall(int *a, int *b){
        for(int i=0;i<10;i++)
            printf("%d %d\n",a[i],b[i]);
}
long par_region(long n){
        int b;
        int *c = (int *)malloc(sizeof(int)*10);
        int d[10];
        const int f=0;
        static int g=0;
        meta_fork{
           int e = b;
           subcall(c,d);
        }
}
int main(int argc, char **argv){
        long n=10;
        par_region(n);
        return 0;
}
```

Figure 3.7: Example of various variable attributes in a parallel region.

3.4 Semantics of the parallel constructs in MetaFork

The goal of this section is to formally define the semantics of each of the parallel constructs in MetaFork. To do so, we introduce the *serial C-elision* of a MetaFork program $\mathcal{M}$ as a C program $\mathcal{C}$ whose semantics define those of $\mathcal{M}$. The program $\mathcal{C}$ is obtained from the program $\mathcal{M}$ by a set of rewriting rules stated through Algorithm 1 to Algorithm 4.

As mentioned before, spawning a function call in MetaFork has the same semantics as spawning a function call in CilkPlus. More precisely: `meta_fork f(args)` and `cilk_spawn f(args)` are semantically equivalent.

Next, we specify the semantics of the spawning of a block in MetaFork. To this end, we use Algorithms 2 and 3 which reduce the spawning of a block to that of a function call:

- Algorithm 2 takes care of the case where the spawned block consists of a single instruction, where a variable is assigned to the result of a function call,
- Algorithm 3 takes care of all other cases.

Note that, in the pseudo-code of those algorithms, the **generate** keyword is used to indicate that a sequence of string literals and string variables is written to the medium (file, screen, etc.) where the output program is being emitted.

A `meta_for` loop allows iterations of the loop body to be executed in parallel. By default, each iteration of the loop body is executed by a separate thread. However, using the `grainsize` compilation directive, one can specify the number of loop iterations executed per thread

```
#pragma meta grainsize = expression
```

In order to obtain the *serial C-elision* a MetaFork for loops, it is sufficient to replace `meta_for` by `for`.

3.5 Translating code from CilkPlus to MetaFork

Translating code from CilkPlus to MetaFork is easy in principles since CilkPlus is a subset of MetaFork. However, implicit CilkPlus barriers need to be explicitly inserted in the target MetaFork code, see Figure 3.9. This implies that, during translation, it is necessary to trace the structure of the CilkPlus parallel regions in order to properly insert barriers in the generated MetaFork code.

Algorithm 1: Fork_region(V, R)

Input: R is a statement of the form:

meta_fork [shared(Z)] B

where Z is a sequence of variables, B is basic block of code, V is a list of all variables occuring in the parent region of R and which are not local to B.

Output: The serial C-elision of the above MetaFork statement.

```
1 L ← [Z]  /* L is a list consisting of the items of the sequence Z */
2 if B consists of a single statement which is a function call without left-value then
3     generate(B);
4 else if B consists of a single statement which is a function call with left-value then
5     Function_with_lvalue(V, L, B);
6 else
7     Block_call(V, L, B);
```

```
void function()
{
   int i = 0, j = 0;
   cilk_for ( int j = 0; j < ny; j++ )
   {
     int i;
     for ( i = 0; i < nx; i++ )
     {
        u[i][j] = unew[i][j];
     }
   }
   ...
}
```

```
void function()
{
   int i = 0, j = 0;
meta_for ( int j = 0; j < ny; j++ )
   {
     int i;
     for ( i = 0; i < nx; i++ )
     {
        u[i][j] = unew[i][j];
     }
   }
   ...
}
```

Figure 3.8: Example of translating parallel for loop from CilkPlus to MetaFork

Algorithm 2: Function_with_lvalue(V, L, B)

Input: B is a statement of the form:

$$\text{G} = \text{F(A)};$$

where G is a left-value, A is an argument sequence, F is a function name and, V, L are as in Algorithm 1.

Output: The serial C-elision of the MetaFork statement below:

$$\texttt{meta\_fork}\ [\texttt{shared}(\text{Z})]\ \text{G} = \text{F(A)}$$

where L is [Z].

```
1 if (G not in L) and (G is a value-type variable) and (G is not declared static
  or const) then
2     (E, H, P) = Outlining_region(V, L, B);
3     generate(E);
4     generate(H," ( ", P, " ) ");
5 else
6     generate(B);
```

Algorithm 3: Block_call(V, L, B)

Input: V, L and B are as in Algorithm 1.

Output: The serial C function call of the above input.

```
1 (E, H, P) = Outlining_region(V, L, B);
2 generate(E);
3 generate(H," ( ", P, " ) ");
```

```
void test()
{
        int x;
        x = cilk_spawn test1();
}
```

```
void test()
{
        int x;
        x = meta_fork test1();
        meta_join;
}
```

Figure 3.9: Example of inserting barrier from CilkPlus to MetaFork.

Algorithm 4: Outlining_region(V, L, B)

Input: V, L and B are as in Algorithm 1.

Output: (E, H, P) where E is a function definition of the form

`void` H(T) {D}

H is a function name, T is a sequence of formal parameters, D is a function body and P is a sequence of actual parameters such that $H(P)$ is a valid function call.

Note that the output is obtained in 2 passes: H(T) in Pass 1 and D in Pass 2.

```
1  let M be the subset of all variables in V that are value-type and declared static;
2  Initialize each of D, P, T to the empty string /* PASS 1 */
3  foreach variable v in V do
4      if v is redeclared in B and thus local to B then
5          do nothing
6      else if v is in L or in M then
7          append ("&" + v.name) to P
8          append (v.type + " * " + v.name) to T
9      else
10         append v.name to P
11         append (v.type + v.name) to T
12 Translate B verbatim except /* PASS 2 */
13 foreach right-value R within B do
14     if R is a function call G(A) then
15         Initialize A_new to the empty list
16         foreach variable v in A do
17             if v in L or in M then
18                 create a new variable name tmp
19                 insert v.type tmp = v.value at the beginning of D
20                 append tmp to A_new
21             else
22                 append v to A_new
23         append G(A_new) to D
24     else if R is in L or in M then
25         append ("*" + R.name) to D
26     else
27         append R to D
```

3.6 Translating code from MetaFork to CilkPlus

Since CilkPlus has no constructs for spawning a parallel region (which is not a function call) we use the *outlining technique* (widely used in OpenMP) to wrap the parallel region as a function, and then call that function concurrently. In fact, we follow the algorithms of Section 3.4, that is Algorithms 1, 2, 3 and 4. Indeed, the problem of translating code from MetaFork to CilkPlus is equivalent to that of defining the serial elision of a MetaFork program.

```
void function()
{
    int i, j;
    meta_fork shared(i)
    {
        i++;
    }

    meta_fork shared(j)
    {
        j++;
    }

    meta_join;
}
```

```
void fork_func0(int* i)
{
    (*i)++;
}
void fork_func1(int* j)
{
    (*j)++;
}

void function()
{
    int i, j;
    cilk_spawn fork_func0(&i);
    cilk_spawn fork_func1(&j);
    cilk_sync;
}
```

Figure 3.10: Example of translating parallel region from MetaFork to CilkPlus.

```
void function()
{
     int x;
     meta_fork shared(x)
     {
            func1(x);
     }
}
```

```
void function()
{
      int x;
      cilk_spawn func1(x);

}
```

Figure 3.11: Example of translating function spawn from MetaFork to CilkPlus.

```
void function()
{
   int i = 0, j = 0;
   meta_for ( int j = 0; j < ny; j++ )
   {
     int i;
     for ( i = 0; i < nx; i++ )
     {
        u[i][j] = unew[i][j];
     }
   }
   ...
}
```

```
void function()
{
   int i = 0, j = 0;
cilk_for ( int j = 0; j < ny; j++ )
   {
     int i;
     for ( i = 0; i < nx; i++ )
     {
        u[i][j] = unew[i][j];
     }
   }
   ...
}
```

Figure 3.12: Example of translating parallel for loop from MetaFork to CilkPlus

3.7 Translation from OpenMP to MetaFork: examples

Algorithms translating OpenMP code to MetaFork will be presented in Chapter 4. This section simply collects various translation examples.

3.7.1 Examples covering different cases of OpenMP Tasks

The following examples cover different scenarios of OpenMP Tasks when translating OpenMP code to MetaFork code.

- There are no shared variables (see Figure 3.13).
- Array is a local variable and used inside the task region (see Figure 3.14).
- Variable x stores the value of the function func1 (see Figure 3.15).
- Variable x is passed as an argument to func1 (see Figure 3.16).
- Translating a block of a task (see Figure 3.17).

3.7.2 Translation from OpenMP Parallel for loop to MetaFork

The scheduling strategies that are in OpenMP parallel for loops are ignored in MetaFork.

```
void function()
{
        int x;
        #pragma omp task
        {
         x = func1();
        }
        #pragma omp task:
        {
         func2();
        }
        #pragma omp taskwait
}
```

```
void function()
{
        int x;
        meta_fork
        {
         x = func1();
        }

        meta_fork func2();

        meta_join;
}
```

Figure 3.13: Example of translating task directive from OpenMP to MetaFork without shared variable

```
void function()
{
        int x[10];
        #pragma omp task
        {
         func1(x);
        }
        ...
        #pragma omp taskwait
}
```

```
void function()
{
        int x[10];
        int temp[10];
        memcpy(temp,x,sizeof(x));
        meta_fork func1(temp);
        ...
        meta_join;
}
```

Figure 3.14: Example of translating task directive from OpenMP to MetaFork with private array

```
void function()
{
        int x;
        #pragma omp task shared(x)
        {
         x = func1();
        }
       ...
       #pragma omp taskwait
}
```

```
void function()
{
        int x;
        x = meta_fork func1();
        ...
        meta_join;
}
```

Figure 3.15: Example of translating task directive from OpenMP tp MetaFork with shared lvalue

```
void function()
{
        int x;
        #pragma omp task shared(x)
        {
         func1(x);
        }
        ...
        #pragma omp taskwait
}
```

```
void function()
{
        int x;
        meta_fork func1(x);
        ...
        meta_join;
}
```

Figure 3.16: Example of translating task directive from OpenMP to MetaFork with shared variable as parameter

```
void function()
{
        int x;
        char y;
        #pragma omp task
        {
                x = 10;
                y = 'c';
                func1(x);
        }
        ...
        #pragma omp taskwait
}
```

```
void function()
{
        int x;
        char y;
        meta_fork
        {
                x = 10;
                y = 'c';
                func1(x);
        }
        ...
        meta_join;
}
```

Figure 3.17: Example of translating task directive from OpenMP to MetaFork with shared variable as parameter

The loop control variable is initialized inside the loop. Those variables that are declared as private are reinitialized inside the `parallel for loop` region in MetaFork (see Figure 3.18).

3.7.3 Translation from OpenMP sections to MetaFork

OpenMP allows us to spawn a particular region. This is also supported in MetaFork. Since there is an implied barrier at the end of sections (unless there is a nowait clause specified), a barrier at the end of the translated MetaFork code is provided (see Figure 3.19).

```
void function()
{
   int i = 0, j = 0;
   #pragma omp parallel
   {
     #pragma omp for private(i, j)
      for ( j = 0; j < ny; j++ )
      {
        for ( i = 0; i < nx; i++ )
        {
          u[i][j] = unew[i][j];
        }
      }
    }
    ...
}
```

```
void function()
{
   int i = 0, j = 0;
   meta_for ( int j = 0; j < ny; j++ )
   {
     int i;
     for ( i = 0; i < nx; i++ )
     {
        u[i][j] = unew[i][j];
     }
   }
   ...
}
```

Figure 3.18: Example of translating parallel for loop from OpenMP to MetaFork

```
void function()
{
        int i = 0, j = 0;
        #pragma omp parallel
        #pragma omp sections
        {
           #pragma omp section
               {
                  i++;
               }
           #pragma omp section
               {
                  j++;
               }
         }
}
```

```
void function()
{
        int i = 0, j = 0;
        meta_fork shared(i)
        {
           i++;
        }
        meta_fork shared(j)
        {
           j++;
        }
        meta_join;
}
```

Figure 3.19: Example of translating sections from OpenMP to MetaFork

3.8 Translation from MetaFork to OpenMP: examples

Algorithms translating MetaFork code to OpenMP will be presented in Chapter 4. This section simply collects various translation examples.

3.8.1 Translation from MetaFork to OpenMP Task

The `meta_fork` statement can have the following forms:

- `var = meta_fork function();`
- `meta_fork function();`
- `meta_fork [shared(var)]`
  `block`

In Figure 3.20 there is an example which shows how the above forms are translated to OpenMP tasks.

```
void function()
{
        int x = 0, i = 0;

        x = meta_fork y();

        meta_fork z();

        meta_fork shared(i)
        {
             i++;
        }
        meta_join;
}
```

```
void function()
{
        int x = 0, i = 0;

        #pragma omp task shared(x)
        {
              x = y();
        }

        #pragma omp task
              z();

        #pragma omp task shared(i)
        {
              i++;
        }
        #pragma omp taskwait
}
```

Figure 3.20: Example of translating meta_fork from MetaFork to OpenMP.

3.8.2 Translation from MetaFork parallel for loops to OpenMP

This translation is straightforward and simple as shown in Figure 3.21.

```
void main()
{
       int n = 10, j = 0;
       meta_for(int i = 0; i < n; i++)
       {
           j = j+1;
       }
}
```

```
void main()
{
        int n = 10, j = 0;
        #pragma omp parallel
        {
          #pragma omp for
          for(int i = 0; i < n; i++)
          {
              j = j+1;
          }
         }
}
```

Figure 3.21: Example of translating parallel for loop from MetaFork to OpenMP.

Chapter 4

Implementation of the translators between MetaFork and OpenMP

This chapter provides implementation details of our MetaFork source-to-source compiler. More precisely this chapter deals with the translation from OpenMP to MetaFork and vice-versa. Section 4.1 describes the implementation of the compiler while translating from OpenMP to MetaFork. Section 4.2 describes the implementation of the compiler while translating from MetaFork to OpenMP. As stated before in Section 3.2, MetaFork allows the programmer to spawn a function call (like in Cilk) as well as a block (like in OpenMP). Recall that Section 3.4 defines algorithms for translating code from Cilk to MetaFork and vice versa.

4.1 Translation strategy from OpenMP to MetaFork

To translate parallel constructs from OpenMP to MetaFork, the translator's scanner rules have been implemented so as to accept the OpenMP directives that we support (see Section 4.1.1). For the OpenMP directives that are not supported in MetaFork, the translator reports an error. Most OpenMP directives that we support allow for optional clauses (e.g: `private,shared etc.`) that specify the functionality of the directives in a detailed manner. When the scanner encounters such an optional clause, associated operations with respect to those clauses will be executed accordingly. Token definitions that represent different OpenMP directives along with optional clauses have been added in the scanner as rules. Depending on the type of

the directive, it is matched by its rule and associated operations take place. So, while translating, the following steps are perfomed:

- The OpenMP parallel directives and their associated bodies are translated as per the specifications of MetaFork and the translation takes place in two passes.
- Header files necessary to OpenMP are removed.
- Everything else is copied verbatim.

As stated above, the translation from OpenMP to MetaFork takes place in two passes:

1. In the first pass: (1) the source file(s) are scanned, (2) necessary information (e.g. local variables, global variables etc ..) required for translation is stored and (3) an intermediate file is generated. This is shown in Algorithm 5
2. In the second pass: the generated intermediate file becomes the parsed input file and the target file is generated. In this pass, we only take care of the parallel directives. Algorithms 6 to 14 describe this second pass.

4.1.1 OpenMP directives supported by the translators

OpenMP directives that we support and translate are given below:

1. Task directive as shown below:

   ```
   #pragma omp task
   ```

   The optional `task` directive clauses supported are `shared`, `private` and `firstprivate`.

2. Parallel for directive as shown below:

   ```
   #pragma omp for
   ```

   The optional `for` directive clause supported is `private`

3. Sections directive as shown below:

   ```
   #pragma omp sections
   #pragma omp section
   ```

We only support the default variable attribute in `sections`.

4. Single directive as shown below:

```
#pragma omp single
```

   None of the optional clauses are supported in `single` directive.

5. The synchronization constructs that we support are `master`, `barrier` and `taskwait` directives which is shown below as well.

```
1. #pragma omp master
2. #pragma omp barrier
3. #pragma omp taskwait
```

4.1.2 Translating OpenMP's `omp tasks` to MetaFork

In OpenMP, the `task` construct defines an explicit task, which may be executed by the encountering thread, or deferred for execution by any other thread in the team. When a thread encounters a `task` construct, a task is generated from the code for the associated structured block. The data environment of the task is created according to the data-sharing attribute clauses on the `task` construct and any defaults that apply. The encountering thread may immediately execute the task, or defer its execution. A task construct may be nested inside an outer `task`, but the task region of the inner task is not a part of the task region of the outer `task`.

There are different cases listed below that have to be considered while translating from OpenMP's `omp tasks` to MetaFork's meta_fork. This case distinction is about the block of code associated with the `task` directive:

1. the associated block contains only a statement which is a function call and does not have the `shared` clause.
2. the associated block contains only a statement which is a function call and has the `shared` clause.
3. the associated block contains only a statement which (1) is not a function call or, (2) contains more than one statements (and thus might contain function calls and other statements). In addition, this `task` might contain a `shared` clause.
4. Tasks nested inside a task.

Below are the notations used in the algorithms listed in this chapter.

Notation 1. TASK: This denotes OpenMP `task` directive which is given below:

```
#pragma omp task
```

OPENMP FOR: This denotes OpenMP `for` directive which is given below:

```
#pragma omp for
```

OPENMP SECTIONS: This denotes OpenMP `for` directive which is given below:

```
#pragma omp sections
#pragma omp section
```

As in Section 3, our algorithms use the keyword **generate** to indicate that a sequence of string literals and string variables is written to the medium (file, screen, etc.) where the output program is being emitted.

Below are the list of algorithms which form the translation scheme from OpenMP Tasks to MetaFork.

1. Algorithm 5 is the first pass in the translation. Example 6 illustrates Algorithm 5.
2. Algorithm 6 is a high level algorithm used to convert OpenMP `task` to MetaFork. Example 7 illustrates Algorithm 6.
3. Algorithms 7 and 8 state translation from OpenMP Tasks to MetaFork with `task`'s body containing only a statement which is function call and without `shared` clause. Examples 8 and 9 illustrate Algorithm 7 and Algorithm 8 respectively.
4. Algorithms 9 and 10 state translations from OpenMP Tasks to MetaFork with `task`'s body containing only a statement which is function call with a `shared` clause. Examples 10 and 11 illustrate Algorithm 9 and Algorithm 10 respectively.
5. Algorithm 11 states the translations from OpenMP Tasks to MetaFork with `task`'s body containing either only a statement which is not a function call or more than one statements (that might contain function and non-function calls). This `task` might also have a `shared` clause. Example 12 illustrates Algorithm 11.

Algorithm 5: first_pass(F)

Input: F where F is a function definition.

Output: (A, W, E) where

- A is a list of the variables declared in F but not within a parallel construct,
- W is a list of the variables of A that: (1) are array variables used within an OpenMP task region and, (2) not declared `static` and not qualified `shared`,
- E is a list of the variables of A that are present in the `private` clause of an OpenMP `for` loop.

```
1  let M be a list of the local variables to F that are (1) array variables and, (2)
   not declared static and not qualified shared.
2  Initialize A to empty list
3  Initialize W to empty list
4  Initialize E to empty list
5  foreach statement S in the body of F do
6      if S is not a parallel construct then
7          append to A every variable declared in R
8      if S is an OpenMP task then
9          foreach variable v in M used inside the task do
10             append v to W
11     if S is an OpenMP for directive then
12         if the private clause is present then
13             append all the variables present in the private clause to E
14     if S is an OpenMP sections then
15         do nothing
16     if S is any other OpenMP directive, thus not currently supported then
17         error ("Not supported yet")
```

Algorithm 6: task_translate(I)

Input: I is the OpenMP `task` statement of the form:

TASK [`shared(`Z`)`]
B

where

- TASK is the OpenMP task directive,
- Z is a sequence of variables,
- W as in Algorithm 5 and
- B is a basic block of code that contains only a single statement.

Output: A MetaFork statement semantically equivalent to the input OpenMP statement.

```
1  Initialize N to empty list
2  foreach occurrence of variable v in list W do
3      create a tmp variable of type v
4      copy the blocks of storage accessible by v to tmp
5      append tmp to N
6      generate (N)
7  if there is no shared variables list Z in the TASK directive then
8      if there exists a function F in B and F has no left-value then
9          no_shared_no_lvalue(I)
10     else if there exists a function F in B and F has left-value then
11         no_shared_with_lvalue(I)
12 else if there is shared variables list Z in the TASK directive then
13     if there exists a function F in B and F has no left-value then
14         with_shared_no_lvalue(I)
15     else if there exists a function F in B and F has left-value then
16         with_shared_and_lvalue(I)
17 else
18     block_call(I)
```

Algorithm 7: no_shared_no_lvalue(I)

Input: I is the OpenMP `task` statement of the form:

TASK
F(P)

where

- TASK, N as in Algorithm 6,
- W as in Algorithm 5,
- F(P) is a function call where F is a function name and P is the argument sequence.

Output: A MetaFork statement semantically equivalent to the input OpenMP statement

1 **foreach** *variable v common to list W and argument sequence* P **do**
2 overwrite v in P by it's corresponding variable `tmp` from N
3 **generate** ("meta_fork", F, "(", P, ")")

6. Nested tasks (i.e. tasks inside another task) In the case of nested tasks, the inner-most task is processed first using Algorithm 6. The bodies of the outer tasks must then be considered as block spawns.

Below are examples which explain all the above algorithms from Algorithm 5 to Algorithm 11.

Example 6. Input is the below program:

```
void test() {
   int a = 5;
   int b[10];
   static int c[10];
   float d = 0;
   #pragma omp task
   b = test1();
   #pragma omp task
   c = test2();
   #pragma omp parallel
   #pragma omp for
   for(int i = 0; i < a; i++)
```

Algorithm 8: no_shared_with_lvalue(I)

Input: I is the OpenMP `task` statement of the form:

TASK
Y = F(P)

Output: A MetaFork statement of one the following forms:

Y = meta_fork F(P) or meta_fork { Y = F(P) }

where

- TASK, meta_fork, N as in Algorithm 6,
- W as in Algorithm 5,
- Y = F(P) is a function call where Y is the left-value F is a function name and P is the argument sequence.

```
1 foreach variable v common to list W and argument sequence P do
2     overwrite v in P by it's corresponding variable tmp from N
3 if left-value Y is in list W then
4     overwrite left-value Y by its corresponding variable tmp from N
5     generate (Y, "= meta_fork", F,"(", P ")")
6 else if left-value Y is in list A which are value-type and not declared static
  then
7     generate ("meta_fork {", Y,"=", F, "(", P, ")"})
8 else
9     generate (Y, "= meta_fork", F,"(", P ")")
```

Algorithm 9: with_shared_no_lvalue(I)

Input: I is the OpenMP `task` statement of the form:

TASK `shared`(Z)
F(P)

Output: A MetaFork statement of one of the following forms:

meta_fork F(P) or meta_fork shared(Z) F(P)

where

- TASK, meta_fork, N as in Algorithm 6,
- Z is the sequence of `shared` variables.
- W as in Algorithm 5,
- F(P) is a function call where F is a function name and P is the argument sequence.

```
1 if all the array variables in P are either qualified static or present in shared variables Z then
2     generate ("meta_fork", F, "(", P, ")")
3 else
4     foreach array variable v in argument sequence P do
5         if variable v is either not qualified static or not present in shared variables Z then
6             overwrite v in P by it's corresponding variable tmp from N
7     generate ("meta_fork shared", "(", Z, ") {", F, "(", P, ") }")
```

Algorithm 10: with_shared_and_lvalue(I)

Input: I is the OpenMP `task` statement of the form:

TASK `shared`(Z)
Y = F(P)

Output: A MetaFork statement of one of the following forms:

Y = meta_fork F(P) or meta_fork {Y = F(P)}

- TASK, meta_fork, N as in Algorithm 6,
- Z is the sequence of `shared` variables.
- W as in Algorithm 5,
- Y = F(P) is a function call where Y is the left-value F is a function name and P is the argument sequence.

```
1  foreach array variable v in argument sequence P do
2      if array variable v is either not qualified static or not present in shared
       variables Z then
3          overwrite v in P by it's corresponding variable tmp from N
4  if left-value Y is in list W then
5      overwrite left-value Y by corresponding variable from N
6      generate (Y, "= meta_fork", F,"(", P ")")
7  else if left-value Y is in list A which are value-type and not declared static
   then
8      generate ("meta_fork {", Y,"=", F, "(", P, ")"})
9  else
10     generate (Y, "= meta_fork", F,"(", P ")")
```

Algorithm 11: block_call(I)

Input: I is the OpenMP `task` statement of the form:

TASK [`shared`](Z)
B

where

- TASK, N, B as in Algorithm 6,
- Z is the sequence of `shared` variables.
- W as in Algorithm 5.

Output: A MetaFork statement semantically equivalent to the input OpenMP statement

```
1 foreach variable v in B do
2     if variable v is in list W then
3         overwrite v by corresponding variable tmp from N
4 generate ("meta_fork shared(", Z ") {",B}")
```

```
  { a = d + i; }
  #pragma omp parallel
  #pragma omp for private(d)
  for(int i = 0; i < a; i++)
  {  a = a + i; }
  #pragma omp parallel
  #pragma omp sections
  {
     #pragma omp section
     { i++; }
  }
}
```

Output:

- List A will have variables a,b,c,d,
- List W will have variable b,
- List E will have variable d.

Example 7. Input is the below program:

```
void test() {
   int a = 5;
   int b[1] = {0,1}
   static int c[10];
   float d = 0;
   #pragma omp task
   test1();
   #pragma omp task
   c = test2();
   #pragma omp task shared(d)
   test1();
   #pragma omp task shared(d)
   c = test2();
}
```

Output:

```
void test() {
   int a = 5;
   int b[1] = {0,1}
   int temp[10];              /* new variable "temp" of type "b" is created
   memcpy(temp,b,sizeof(b));  and the contents of "b" is copied to "temp".*/
   static int c[10];
   float d = 0;
   #pragma omp task           /* Algorithm 7 will be called */
   test1();
   #pragma omp task           /* Algorithm 8 will be called */
   c = test2();
   #pragma omp task shared(d) /* Algorithm 9 will be called */
   test3(d);
   #pragma omp task shared(d)  /* Algorithm 10 will be called */
   c = test4(d);
   #pragma omp task
   b = a;
}
```

Example 8. Input is the below program:

```
void test() {
   int a = 5;
```

```
   int b[1] = {0,1}
   static int c[10];
   float d = 0;
   #pragma omp task
   test1(b)
   #pragma omp task
   test2(a);
}
```

Output:

```
void test() {
   int a = 5;
   int b[1] = {0,1}
   int temp[10];              /* new variable "temp" of type "b" is created
   memcpy(temp,b,sizeof(b));  and the contents of "b" is copied to "temp".*/
   static int c[10];
   float d = 0;
   meta_fork test1(temp);     /* "b" is replaced by "temp" */
   meta_fork test(a);
}
```

Example 9. Input is the below program:

```
int e = 10;
void test() {
   int a = 5;
   int b[1] = {0,1}
   static int c[10];
   float d = 0;
   #pragma omp task
   b = test1(b);
   #pragma omp task
   b = test2(a);
   #pragma omp task
   a = test3(a);
   #pragma omp task
   e = test4();
}
```

Output:

```
int e = 10;

void test() {
   int a = 5;
   int b[1] = {0,1}
   int temp[10];              /* new variable "temp" of type "b" is created
   memcpy(temp,b,sizeof(b));  and the contents of "b" is copied to "temp".*/
   static int c[10];
   float d = 0;
   temp = meta_fork test1(temp);     /* "b" is replaced by "temp" */
   temp = meta_fork test2(a);
   meta_fork { a = test3(a); }
   e = meta_fork test4();            /* "e" is a global variable
}
```

Example 10. Input is the below program:

```
void test() {
   int a = 5;
   int b[1] = {0,1}
   static int c[10];
   float d = 0;
   #pragma omp task shared(b)
   test1(b);
   #pragma omp task shared(a)
   test2(a,b);
}
```

Output:

```
void test() {
   int a = 5;
   int b[1] = {0,1}
   int temp[10];             /* new variable "temp" of type "b" is created
   memcpy(temp,b,sizeof(b));  and the contents of "b" is copied to "temp".*/
   static int c[10];
   float d = 0;
   meta_fork test1(temp);     /* "b" is replaced by "temp" */
   meta_fork shared(temp) { test2(a,temp); }
}
```

Example 11. Input is the below program:

```
int e = 10;
void test() {
   int a = 5;
   int b[1] = {0,1}
   static int c[10];
   float d = 0;
   #pragma omp task shared(b)
   b = test1(a);
   #pragma omp task
   a = test2();
   #pragma omp task
   e = test3();
}
```

Output:

```
int e = 10;
void test() {
   int a = 5;
   int b[1] = {0,1}
   int temp[10];              /* new variable "temp" of type "b" is created
   memcpy(temp,b,sizeof(b));  and the contents of "b" is copied to "temp".*/
   static int c[10];
   float d = 0;
   temp = meta_fork test1(a);     /* "b" is replaced by "temp" */
   meta_fork { a = test2(a); }
   e = meta_fork test4();            /* "e" is a global variable
}
```

Example 12. Input is the below program:

```
void test() {
   int a = 5;
   int b[1] = {0,1}
   static int c[10];
   float d = 0;
   #pragma omp task
   b = a;
}
```

Output:

```
void test() {
   int a = 5;
   int b[1] = {0,1}
   int temp[10];              /* new variable "temp" of type "b" is created
   memcpy(temp,b,sizeof(b));  and the contents of "b" is copied to "temp".*/
   static int c[10];
   float d = 0;
   meta_fork { temp = a; }   /* "b" is replaced by "temp" */
}
```

4.1.3 Translating OpenMP's `omp for` to MetaFork

The OpenMP `for` directive specifies that the iterations of the loop immediately following it must be executed in parallel by the team. This assumes a parallel region has already been initiated, otherwise it executes serially.

The scheduling strategies that are in OpenMP parallel for loops are ignored in MetaFork. The loop initialization variable is initialized inside the loop. Those variables that are declared as private in OpenMP are reinitialized inside the **for** loop body in MetaFork.

There will be two changes that occur while translating from OpenMP for loop to MetaFork for loop.

- The `private` clause variables specified in an OpenMP for loop are initialized inside the body of the MetaFork for loop,
- The stride part in the MetaFork for loop accepts one of the unary operators ++, –, +=, -+ (or a statement of the form cv += incr where incr evaluates to a compatible expression) in order to increase or decrease the value of the control variable cv. But OpenMP accepts stride of the form cv = cv + incr. So, we change the representation of stride part while translating from OpenMP for loop to MetaFork for loop.

Note that we only support `private` clause of an OpenMP `for` directive.

Below are the algorithms that shows the translation of OpenMP for loops to MetaFork for loops.

1. Algorithm 12 shows the translation from OpenMP for loops to MetaFork for loops without `private` clause variables. Example 13 depicts Algorithm 12.

2. Algorithm 13 shows the translation from OpenMP for loops to MetaFork for loops with `private` clause variables. Example 14 depicts Algorithm 13.

Algorithm 12: For loops without `private` clause variables

Input: A OpenMP `for` statement of the form:

OPENMP FOR
`for` (I ; C ; S) { B }

where

- OPENMP FOR is the OpenMP for loops directive,
- **for()** is a parallel for loop,
- I is the initialization expression,
- C is the condition expression,
- S is the stride of the loop,
- B is the body of the loop.

Output: A MetaFork statement semantically equivalent to the input OpenMP statement

1 let N be a pattern of the form $cv = cv + incr$ where cv is the control variable and $incr$ evaluates to a compatible expression
2 let A be a pattern of the form $cv+ = incr$ where cv is the control variable and $incr$ evaluates to a compatible expression
3 **if** *S admits the sub-expression matching the pattern N* **then**
4 replace sub-expression S by pattern A
5 **generate** ("meta_for", "(",I, ";", C, ";", A, ")", "{", B, "}")
6 **else**
7 **generate** ("meta_for", "(",I, ";", C, ";", S, ")", "{", B, "}")

Below are examples which explain the algorithms Algorithm 12 to Algorithm 13.

Example 13. Input is the below program:

```
void test() {
   int a = 5;
   int b[10];
   static int c[10];
```

Algorithm 13: For loops with `private` clause variables

Input: A OpenMP `for` statement of the form:

OPENMP FOR `private`(Z)
`for` (I ; C ; S) { B }

where

- OPENMP FOR, meta_for, **for()**, I, C, S as in Algorithm 12
- Z is a sequence of `private` variables.

Output: A MetaFork statement semantically equivalent to the input OpenMP statement

1 let N be a pattern of the form $cv = cv + incr$ where cv is the control variable and $incr$ evaluates to a compatible expression
2 let A be a pattern of the form $cv+ = incr$ where cv is the control variable and $incr$ evaluates to a compatible expression
3 **if** *S admits the sub-expression matching the pattern N* **then**
4 replace sub-expression S by pattern A
5 **generate** ("meta_for **for**", "(",I, ";", C, ";", A, ")", "{" Z , B, "}")
6 **else**
7 **generate** ("meta_for", "(",I, ";", C, ";", S, ")", "{", Z , B, "}")

```
   float d = 0;
   #pragma omp parallel
   #pragma omp for
   for(int i = 0; i < a; i++)
   { a = a + i; }
}
```

Output:

```
void test() {
   int a = 5;
   int b[10];
   static int c[10];
   float d = 0;
   meta_for(int i = 0; i < a; i++)
   { a = a + i; }
```

```
}
```

Example 14. Input is the below program:

```
void test() {
   int a = 5;
   int b[10];
   static int c[10];
   float d = 0;
   #pragma omp parallel
   #pragma omp for private(d)
   for(int i = 0; i < a; i++)
   { a = d + i; }
}
```

Output:

```
void test() {
   int a = 5;
   int b[10];
   static int c[10];
   float d = 0;
   meta_for(int i = 0; i < a; i++)
   {  float d = 0;
      a = d + i; }
}
```

4.1.4 Translating OpenMP's `omp sections` to MetaFork

The `sections` directive is a non-iterative work-sharing construct. Independent `section` directives are nested within a `sections` directive. OpenMP allows us to spawn a particular region. This is also supported in MetaFork. Since there is an implied barrier at the end of `sections` directive (unless there is a `nowait` clause specified), a barrier at the end of `sections` directive in MetaFork is provided.

Algorithm 14 shows the translation from OpenMP `sections` to MetaFork. Example 15 depicts Algorithm 14.

Example 15. Input is the below program:

```
void test() {
   int a = 5;
```

Algorithm 14: OpenMP `sections` to MetaFork's meta_fork

Input: The OpenMP `sections` statement of the form:

OPENMP SECTIONS { B }

Output: A MetaFork fork statement of the form:

meta_fork { B } or meta_fork shared(A) { B }

where

- OPENMP SECTIONS is the OpenMP `sections` directive,
- meta_fork is the MetaFork keyword for spawning a region,
- B is a basic block of code,
- A is a sequence of shared variables.

```
1 Initialize A to empty list
2 foreach variable v in B do
3     if variable v is a non-local variable to B then
4         if variable v is not declared static in the parent region on B then
5             append variable v to list A
6 if A is non-empty then
7     generate ("meta_fork shared", "(", A, ")", B)
8 else
9     generate ("meta_fork", B)
```

```
  int b[10];
  static int c[10];
  float d = 0;
  int i = 1;
  #pragma omp parallel
  #pragma omp sections
  {
     #pragma omp section
     { i++; }
  }
}
```

Output:

```
void test() {
   int a = 5;
   int b[10];
   static int c[10];
   float d = 0;
   int i = 1;
   meta_fork shared(i)
   { i++; }
}
```

4.2 Translation strategy from MetaFork to OpenMP

MetaFork does not make any assumptions about the run-time system, in particular about task scheduling. It has four parallel constructs: function call spawn, block spawn, parallel for-loop and synchronization barrier. The first two use the keyword meta_fork while the other two use respectively the keywords meta_for and meta_join. We stress the fact that MetaFork allows the programmer to spawn a function call (like in Cilk) as well as a block (like in OpenMP).

4.2.1 Translating from MetaFork's meta_fork to OpenMP

1. Algorithm 15 is a higher level algorithm that tells the translator whether it's a function spawn or a block spawn.
2. Algorithms 16 to 18 show the translations from MetaFork's meta_fork function spawn to OpenMP Tasks. Examples 16 to 18 depicts Algorithm 16 to 18
3. Translating from MetaFork's meta_fork block spawn to OpenMP `task`. This is a straightforward case and translation is simple and elegant. MetaFork keyword with/without shared qualifier is simply replaced by OpenMP directive with/without `shared` clause respectively.

Example 16. Input is the below program:

```
void test() {
   int i = 1;
```

Algorithm 15: meta_high_level(I)

Input: I is the MetaFork fork statement of the form:

meta_fork B

where

- meta_fork is MetaFork keyword,
- B is a basic block of code that contains only a single statement.

Output: An OpenMP statement semantically equivalent to the input MetaFork statement.

```
1 if there exists a function F in B then
2     if function F has no left-value then
3         meta_without_lvalue(I)
4 else if there exists a function F in B then
5     if function F has left-value then
6         meta_with_lvalue(I)
7 else
8     meta_block_call(I)
```

```
   meta_fork test1(i);
   meta_fork shared(i) { test2(i); }
}
```

Output:

```
void test() {
   int i = 1;
   #pragma omp task
   test1(i);
   #pragma omp task shared(i)
   { test2(i); }
}
```

Example 17. Input is the below program:

```
void test() {
```

Algorithm 16: meta_without_lvalue(I)

Input: The MetaFork fork statement of the form:

meta_fork F(P)

where

- F(P) is a function call where F is the function name, P is the argument sequence,
- meta_fork as in Algorithm 15.

Output: An OpenMP statement semantically equivalent to the input MetaFork statement

```
1  let A be a list of the local variables to F that are (1) array variables and, (2)
   not declared static.
2  let S be initialized to empty list
3  foreach variable v in list A do
4      if v is in argument sequence P then
5          append v to list S
6  if list S is non-empty then
7      generate (TASK, "shared(", S, ")\n")
8      generate (F, "(", P, ")")
9  else
10     generate (TASK, "\n")
11     generate (F, "(", P, ")")
```

```
   int i = 1;
   int b[1] = {0,1};
   b = meta_fork test1(b,i);
   i = meta_fork { test2(i); }
   i = meta_fork test1(b);
}
```

Output:

```
void test() {
   int i = 1;
   int b[1] = {0,1};
   #pragma omp task shared(b)
```

Algorithm 17: meta_with_lvalue(I)

Input: A MetaFork fork statement of the form:

Y = meta_fork F(P)

where

- Y = F(P) is a function call where Y is the left-value F is the function name, P is the argument sequence,
- meta_fork and TASK as in Algorithm 15.

Output: An OpenMP statement semantically equivalent to the input MetaFork.

```
1  let A be a list of the local variables to F that are (1) array variables and, (2)
   not declared static.
2  let S be initialized to empty list
3  foreach variable v in list A do
4      if variable v is in argument sequence P then
5          append variable v to list S
6  if left-value Y is in list A then
7      if list S is non-empty then
8          generate (TASK, "shared(", Y, S")\n")
9          generate(Y, "=", F,"(",P")")
10     else
11         generate (TASK, "shared(", Y")\n")
12         generate(Y, "=", F,"(",P")")
13 else if list S is non-empty then
14     generate (TASK, "shared("S")\n")
15     generate(Y, "=", F,"(",P")")
16 else
17     generate (TASK)
18     generate (Y, "=", F,"(",P")")
```

Algorithm 18: meta_block_call(I)

Input: A MetaFork fork statement of the form:

meta_fork B

where

- meta_fork as in Algorithm 15,
- B is the basic block of code.

Output: An OpenMP statement semantically equivalent to the input MetaFork statement

```
1  Initialize S to empty list
2  foreach variable v in B do
3      if v is an array variable then
4          append v to list S
5  if list S is non-empty then
6      generate (TASK, "shared(", S")\n")
7      generate(B)
8  else
9      generate (TASK "\n")
10     generate(B)
```

```
   b = test1(i);
   #pragma omp task shared(i)
   { test2(i); }
   #pragma omp task shared(b)
   i = test1(b);
}
```

Example 18. Input is the below program:

```
void test() {
   int b[0];
   int a = 5;
   meta_fork
   { b[0] = a; }
}
```

Output:

```
void test()
{
   int b[0];
   int a = 5;
   #pragma omp task shared(b)
   { b[0] = a; }
}
```

4.2.2 Translating from MetaFork's meta_for to OpenMP

The MetaFork keyword for the for loop is replaced by OpenMP parallel for loop directive. Example 19 shows how MetaFork for loop is translated to OpenMP for loop.

Example 19. Input is the below program:

```
void test() {
   int a = 5;
   int b[10];
   static int c[10];
   float d = 0;
   meta_for(int i = 0; i < a; i++)
   { float d = 0;
     a = d + i; }
}
```

Output:

```
void test() {
   int a = 5;
   int b[10];
   static int c[10];
   float d = 0;
   #pragma omp parallel
   #pragma omp for
   for(int i = 0; i < a; i++)
   {  float d = 0;
      a = d + i; }
}
```

Chapter 5

Experimental evaluation

The work reported in this chapter evaluates the correctness, performance and usefulness of the four MetaFork translators (MetaFork to CilkPlus, CilkPlus to MetaFork, MetaFork to OpenMP, OpenMP to MetaFork) introduced in Chapter 3. To this end, we run these translates on various input programs written either in CilkPlus or OpenMP, or both. Two series of experiences are conducted: their set up is described in Section 5.1 meanwhile the results appear in Sections 5.2 and 5.3 respectively.

We stress the fact that our purpose is not to compare the performance of the CilkPlus or OpenMP run-time systems and programming environments. The reader should notice that the codes used in this experimental study were written by different persons with different levels of expertise. In addition, the reported experimentation is essentially limited to one architecture (Intel Xeon) and one compiler (GCC). Therefore, it is delicate to draw any clear conclusions that would compare CilkPlus or OpenMP. For this reason, this questions is not addressed in this thesis And, once again, this is not the purpose of this work.

5.1 Experimentation set up

We conducted two experiments. In the first one, we compared the performance of hand-written codes. The motivation, recalled from the introduction, is *comparative implementation*. The underlying observation is that the same multithreaded algorithm, based on the fork-join parallelism model, implemented with two different concurrency platforms, say CilkPlus and OpenMP, could result in very different performance, often very hard to analyze and compare. If one code scales well while the other does not, one may suspect an efficient implementation of the latter as well

as other possible causes such as higher parallelism overheads. Translating the inefficient code to the other language can help narrowing the problem. Indeed, if the translated code still does not scale one can suspect an implementation issue (say the programmer missed to parallelize one portion of the algorithm) whereas if it does scale, then one can suspect a parallelism overhead issue in the original code (say the grain-size of the parallel for-loop is too small).

For this experience, we use a series of test-cases consisting of a pair of programs, one hand-written OpenMP program and one, hand-written CilkPlus program. We observe that one program (written by a student) has a performance bottleneck while its counterpart (written by an expert programmer) does not. We translate the inefficient program to the other language, then check whether the performance bottleneck remains or not, so as to narrow the performance bottleneck in the inefficient program.

Our second experience, also motivated in the introduction, is dedicated to automatic translation of highly optimized code. Now, for each test-case, we have either a hand-written-and-optimized CilkPlus program or a hand-written-and-optimized OpenMP. Our goal is to determine whether or not the translated programs have similar serial and parallel running times as their hand-written-and-optimized counterparts.

For both experiences, apart from student's code, the code that we use comes from the following the sources:

- John Burkardt's Home Page (Florida State University) `http://people.sc.fsu.edu/~%20jburkardt/c_src/openmp/openmp.html`
- Barcelona OpenMP Tasks Suite (BOTS) [26]
- CilkPlus distribution examples `http://sourceforge.net/projects/cilk/`

The source code of those test case was compiled as follows:

- CilkPlus code with GCC 4.8 using `-O2 -g -lcilkrts -fcilkplus`
- CilkPlus code with GCC 4.8 using `-O2 -g -fopenmp`

All our compiled programs were tested on

- AMD Opteron 6168 48core nodes with 256GB RAM and 12MB L3
- Intel Xeon 2.66GHz/6.4GT with 12 physical cores and hyper-threading,

The two main quantities that we measure are:

- *scalability* by running our compiled OpenMP and CilkPlus programs on $p = 1, 2, 4, 6, 8, \ldots$ processors; speedup curves data are shown in Figures 5.4, 5.3, 5.2, 5.5, 5.6, 5.7, 5.8, 5.9, 5.10, 5.11, 5.12, 5.13, 5.14, 5.15, 5.16 and 5.17 is

mainly collected on Intel Xeon's nodes (for convenience) and repeated/verified on AMD Opteron nodes.

- *parallelism overheads* by running our compiled OpenMP and CilkPlus programs on $p = 1$ against their serial elisions. This is shown in Table 5.1

As mentioned in the introduction, validating the correctness of our translators was a major requirement of our work. Depending on the test-case, we could use one or the other following strategy.

- Assume that the original program, say $\mathcal{P}$, contains both a parallel code and its serial elision (manually written). When program $\mathcal{P}$ is executed, both codes run and compare their results. Let us call $\mathcal{Q}$ the translated version of $\mathcal{P}$. Since serial elisions are unchanged by our translation procedures, then $\mathcal{Q}$ can be verified by the same process used for program $\mathcal{P}$. This first strategy applies to the `Cilk++` distribution examples and the BOTS (Barcelona OpenMP Tasks Suite) examples
- If the original program $\mathcal{P}$ does not include a serial elision of the parallel code, then the translated program $\mathcal{Q}$ is verified by comparing the output of $\mathcal{P}$ and $\mathcal{Q}$. This second strategy had to be applied to the FSU (Florida State University) examples.

5.2 Comparing hand-written codes

5.2.1 Matrix transpose

In this example, the two original parallel programs are based on different algorithms for matrix transposition which is a challenging operation on multi-core architectures. Without doing complexity analysis, discovering that the OpenMP code (written by a student) runs in $O(n^2 \log(n))$ bit operations instead of $O(n^2)$ as the CilkPlus (written by Matteo Frigo) is very subtle.

Figure 5.1 shows code snippet for the matrix transpose program for both original CilkPlus and translated OpenMP. Figure 5.2 shows the running time of both the original OpenMP and the translated OpenMP code from the original CilkPlus code and it suggests that the code translated from CilkPlus to OpenMP is more appropriate for fork-join multi-threaded languages targeting multicores because of the algorithm used in CilkPlus code.

```
              /*---------------Original CilkPlus code-------------*/
template <typename T>
void transpose(T *A, int lda, T *B, int ldb,
               int i0, int i1, int j0, int j1)
{
 tail:
     int di = i1 - i0, dj = j1 - j0;
     if (di >= dj && di > THRESHOLD) {
          int im = (i0 + i1) / 2;
          cilk_spawn transpose(A, lda, B, ldb, i0, im, j0, j1);
          i0 = im; goto tail;
     } else if (dj > THRESHOLD) {
          int jm = (j0 + j1) / 2;
          cilk_spawn transpose(A, lda, B, ldb, i0, i1, j0, jm);
          j0 = jm; goto tail;
     } else {
          for (int i = i0; i < i1; ++i)
               for (int j = j0; j < j1; ++j)
                    B[j * ldb + i] = A[i * lda + j];
     }
}

              /*---------------Translated OpenMP code-------------*/
template <typename T>
void transpose(T *A, int lda, T *B, int ldb,
               int i0, int i1, int j0, int j1)
{
 tail:
     int di = i1 - i0, dj = j1 - j0;
     if (di >= dj && di > THRESHOLD) {
          int im =(i0 + i1) / 2;
           #pragma omp task
 transpose(A, lda, B, ldb, i0, im, j0, j1);
          i0 = im; goto tail;
     } else if (dj > THRESHOLD) {
          int jm =(j0 + j1) / 2;
           #pragma omp task
 transpose(A, lda, B, ldb, i0, i1, j0, jm);
          j0 = jm; goto tail;
     } else {
          for (int i = i0; i < i1; ++i)
               for (int j = j0; j < j1; ++j)
                    B[j * ldb + i] = A[i * lda + j];
     }
#pragma omp taskwait
 }
```


Figure 5.1: Code snippet for the matrix transpose program

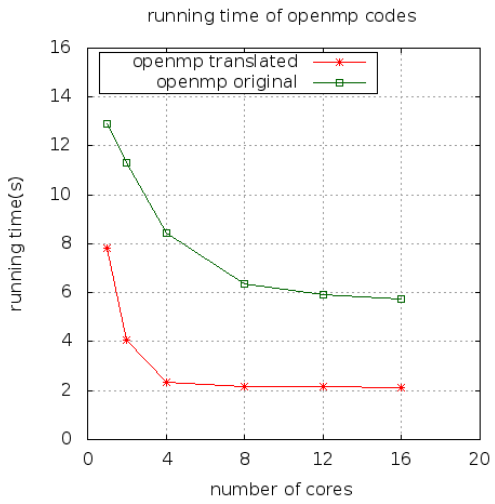


Figure 5.2: Matrix_transpose : $n = 32768$

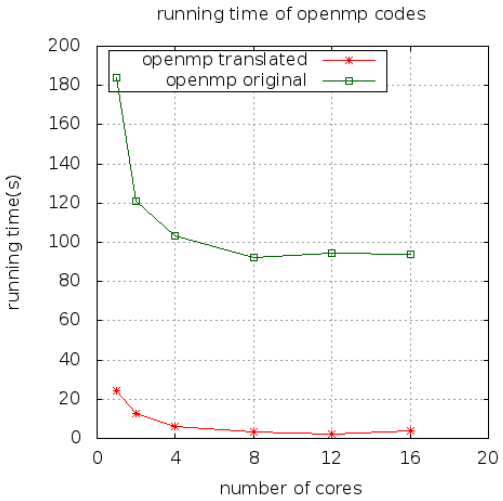


Figure 5.3: Matrix inversion : $n = 4096$

5.2.2 Matrix inversion

In this example, the two original parallel programs are based on different serial algorithms for matrix inversion. The original OpenMP code uses Gauss-Jordan elimination algorithm while the original CilkPlus code uses a divide-and-conquer approach based on Schur's complement. Figure 5.3 shows the running time of both the original OpenMP and the translated OpenMP code from the original CilkPlus code.

As in the previous example, the translation narrows the algorithmic issue in this example as well.

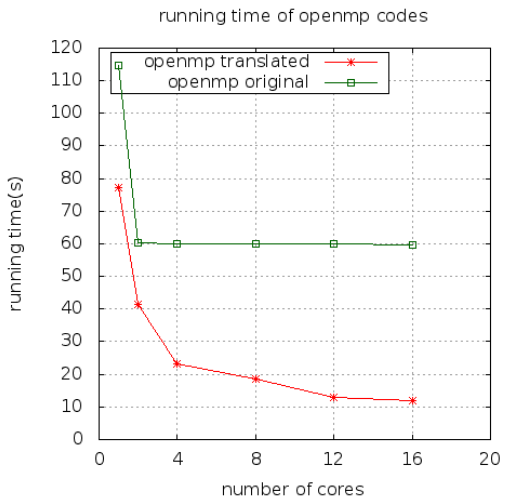


Figure 5.4: Mergesort (Student's code) : $n = 5 \cdot 10^8$

5.2.3 Mergesort

There are two different versions of this example. The original OpenMP code (written by a student) misses to parallelize the merge phase and simply spawns the two recursive calls whereas the original CilkPlus code (written by an expert) does both. Figure 5.4 shows the running time of both the original OpenMP and the translated OpenMP code from the original CilkPlus code.

As you can see in Figure 5.4 the speedup curve of the translated OpenMP code is as theoretically expected while the speedup curve of the original OpenMP code shows a limited scalability.

Hence, the translated OpenMP (and the original CilkPlus program) exposes more parallelism, thus narrowing the performance bottleneck in the original OpenMP code.

5.2.4 Naive matrix multiplication

This is the naive three-nested-loops matrix multiplication algorithm, where two loops have been parallelized. The parallelism is $O(n^2)$ as for DnC MM. However, the ratio work-to-memory-access is essentially equal to 2, which is much less that for DnC MM. This limits the ability to scale and reduces performance overall on multicore processors. Figure 5.5 shows the speed-up curve for naive matrix multiplication.

As you can see from Figure 5.5 both CilkPlus and OpenMP scale poorly because of algorithmic issues.

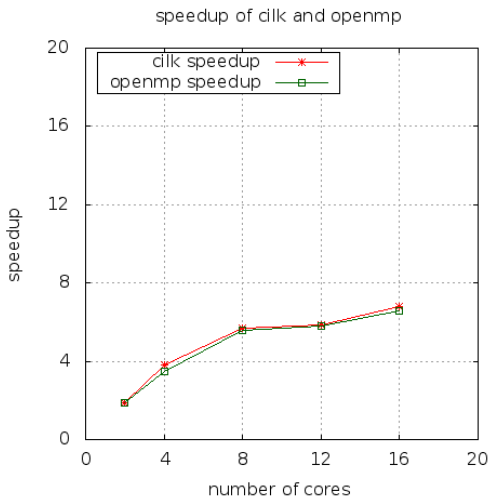


Figure 5.5: Naive Matrix Multiplication : $n = 4096$

5.3 Automatic translation of highly optimized code

5.3.1 Fibonacci number computation

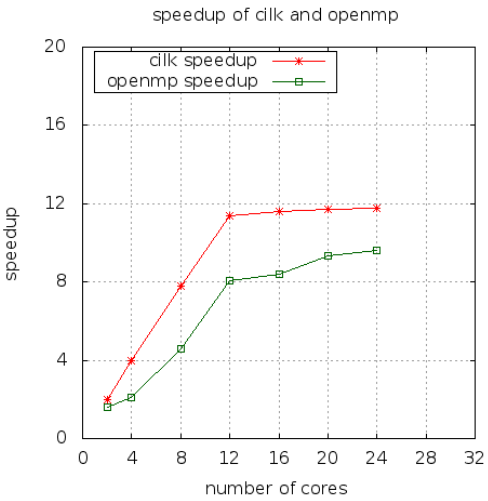
In this example, we have translated the original CilkPlus code to OpenMP. The algorithm used in this example has high parallelism and no data traversal. The speed-up curve for computing Fibonacci with inputs 45 and 50 are shown in Figure 5.6(a) and Figure 5.6(b) respectively.

As you can see from Figure 5.6 CilkPlus (original) and OpenMP (translated) codes scale well.

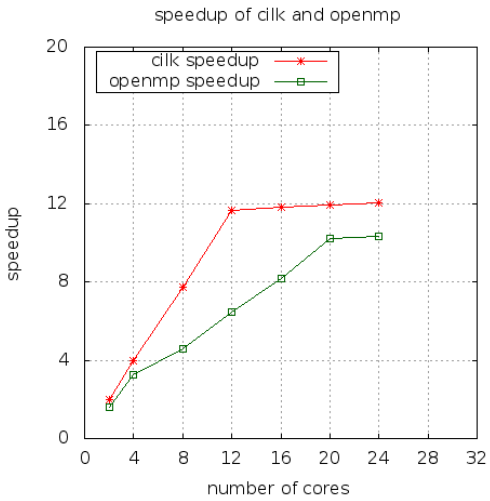
5.3.2 Divide-and-conquer matrix multiplication

In this example, we have translated the original CilkPlus code to OpenMP code. The divide-and-conquer algorithm of [29] is used to compute matrix multiplication. This algorithm has high parallelism and optimal cache complexity. It is also data-and-compute-intensive. The speed-up curve after computing matrix multiplication with inputs 4096 and 8192 are shown in Figure 5.7(a) and Figure 5.7(b) respectively.

As you can see from Figure 5.7 CilkPlus (original) and OpenMP (translated) codes scale well.

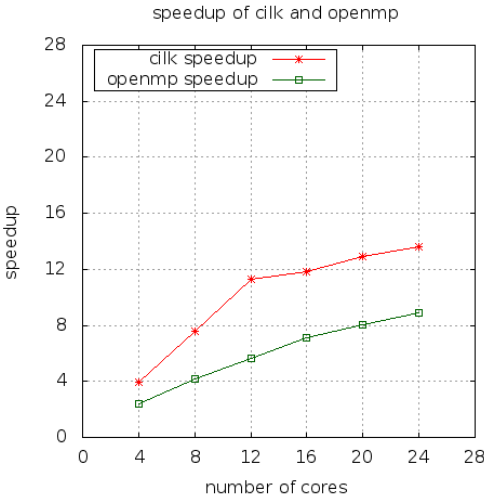


(a) Fibonacci : 45

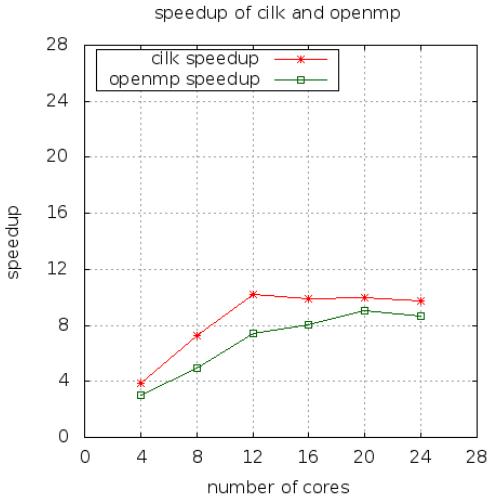


(b) Fibonacci : 50

Figure 5.6: Speedup curve on Intel node

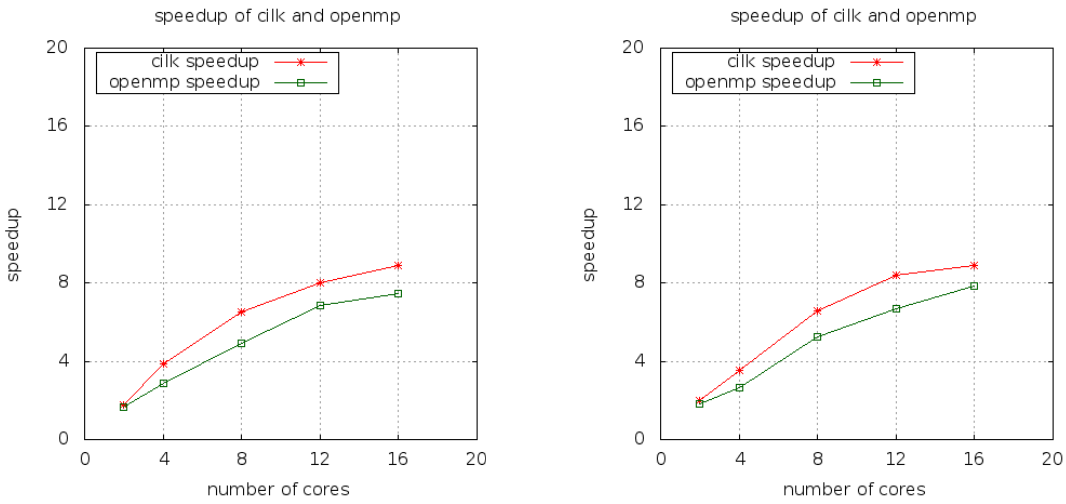


(a) DnC MM : 4096



(b) DnC MM : 8192

Figure 5.7: Speedup curve on intel node



(a) Prefix_sum : $n = 5 \cdot 10^8$ (b) Prefix_sum : $n = 10^9$

Figure 5.8: Speedup curve on Intel node

5.3.3 Parallel prefix sum

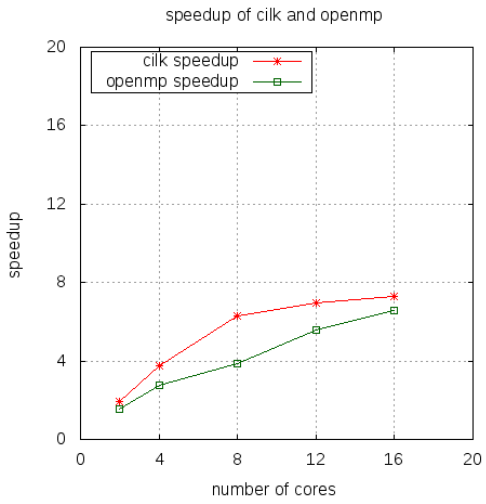
In this example, we have translated the original CilkPlus code to OpenMP code. The algorithm [15] used in this example has high parallelism, low work-to-memory-access ratio which is ($O(log(n))$ traversals for a $O(n)$ work. The speed-up curves with inputs $5 \cdot 10^8$ and 10^9 are shown in Figure 5.8(a) and Figure 5.8(b) respectively.

As you can see from Figure 5.7 CilkPlus (original) and OpenMP (translated) codes scale well at almost the same rate.

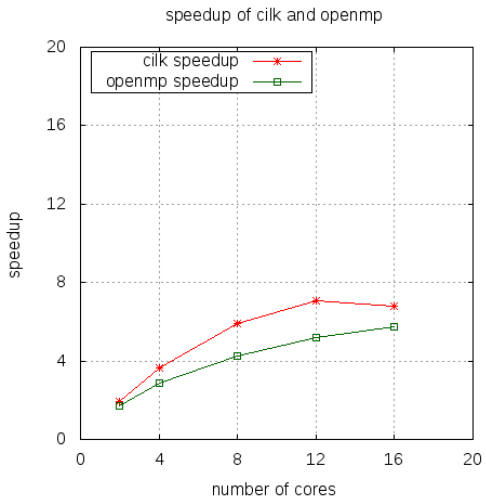
5.3.4 Quick sort

This is the classical quick-sort where the division phase has not been parallelized, on purpose. Consequently, the theoretical parallelism drops to ($log(n)$). The ratio of work-to-memory-access is constant. The speed-up curve for quick sort is shown in Figure 5.9(a) and Figure 5.9(b).

As you can see from Figure 5.9 CilkPlus (original) and OpenMP (translated) codes scale at almost the same rate.



(a) Quick Sort: $n = 2 \cdot 10^8$



(b) Quick Sort: $n = 5 \cdot 10^8$

Figure 5.9: Speedup curve on Intel node

5.3.5 Mandelbrot set

The Mandelbrot set [1] is a mathematical set of points whose boundary is a distinctive and easily recognizable two-dimensional fractal shape.

Mandelbrot set images are made by sampling complex numbers and determining for each whether the result tends towards infinity when a particular mathematical operation is iterated on it. Treating the real and imaginary parts of each number as image coordinates, pixels are colored according to how rapidly the sequence diverges, if at all.

In this example, algorithm is compute-intensive and does not traverse large data. The running time after computing the Mandelbrot set with grid size of 500x500 and 2000 iterations is shown in Figure 5.10.

As you can see from Figure 5.10 the speed-up is close to linear since this application is embarrassingly parallel. Both OpenMP (original) and CilkPlus (translated) codes scale at almost the same rate.

5.3.6 Linear system solving (dense method)

In this example, different methods of solving the linear system A*x=b are compared. In this example there is a standard sequential code and slightly modified sequential

[1] `http://en.wikipedia.org/wiki/Mandelbrot_set`

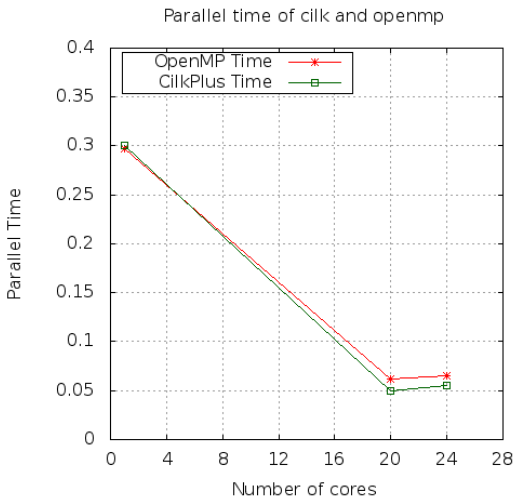


Figure 5.10: Mandelbrot set for a 500×500 grid and 2000 iterations.

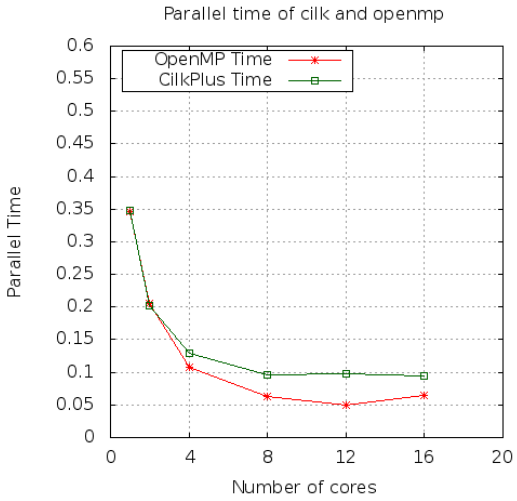


Figure 5.11: Linear system solving (dense method)

code to take advantage of OpenMP. The algorithm in this example uses Gaussian elimination.

This algorithm has lots of parallelism. However, minimizing parallelism overheads and memory traffic is a challenge for this operation. The running time of this example is shown in Figure 5.11.

As you can see from the Figure 5.11 both OpenMP (original) and CilkPlus (translated) codes scale well up to 12 cores. Note that that we are experimenting on a Xeon node with 12 physical cores with hyper-threading turned on.

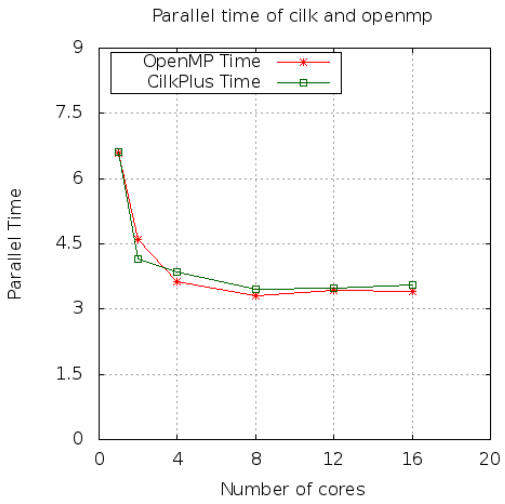


Figure 5.12: FFT (FSU) over the complex in size 2^{25}.

Test	input size	CilkPlus		OpenMP	
		Serial	T_1	serial	T_1
FFT (BOTS)	33554432	7.50	8.12	7.54	7.82
MergeSort (BOTS)	33554432	3.55	3.56	3.57	3.54
Strassen	4096	17.08	17.18	16.94	17.11
SparseLU	128	568.07	566.10	568.79	568.16

Table 5.1: Running times on Intel Xeon 12-physical-core nodes (with hyperthreading turned on).

5.3.7 FFT (FSU version)

This example demonstrates the computation of a Fast Fourier Transform in parallel. The algorithm used in this example has low work-to-memory-access ratio which is challenging to implement efficiently on multi-cores. The running time of this example is shown in Figure 5.12.

As you can see from the Figure 5.12 both OpenMP (original) and CilkPlus (translated) codes scale well up to 8 cores.

Table 5.1 shows the running time of the serial version v/s single core for some of the examples.

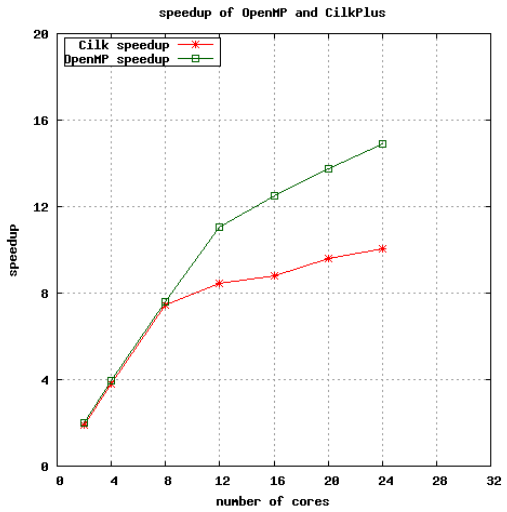


Figure 5.13: Protein alignment - 100 Proteins.

5.3.8 Barcelona OpenMP Tasks Suite

Barcelona OpenMP Tasks Suite (BOTS) project [26] is a set of applications exploiting regular and irregular parallelism, based on OpenMP tasks.

5.3.9 Protein alignment

Alignment application aligns all protein sequences from an input file against every other sequence using the Myers and Miller [36] algorithm.

The alignments are scored and the best score for each pair is provided as a result. The scoring method is a full dynamic programming algorithm. It uses a weight matrix to score mismatches, and assigns penalties for opening and extending gaps. The output is the best score for each pair of them. The speed-up curve for this example is shown in Figure 5.13.

This algorithm is compute-intensive and has few coarse grained tasks which is a challenge for the implementation to avoid load balance situations. As you can see from the Figure 5.13 both OpenMP (original) and CilkPlus (translated) codes scale almost same up to 8 cores.

5.3.10 FFT (BOTS version)

In this example, FFT computes the one-dimensional Fast Fourier Transform of a vector of n complex values using the Cooley-Tukey [21] algorithm. It's a divide and

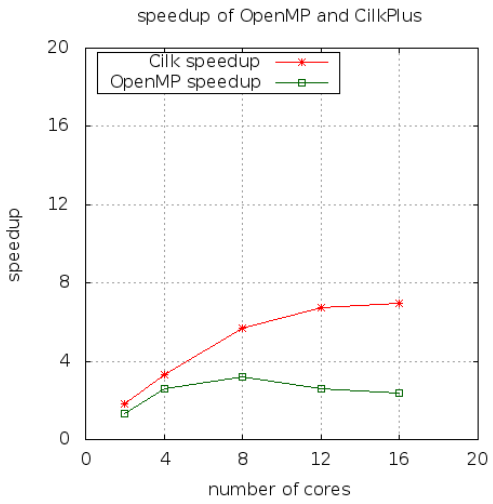


Figure 5.14: FFT (BOTS).

conquer algorithm that recursively breaks down a Discrete Fourier Transform (DFT) into many smaller DFTs. The speed-up curve for this example is shown in Figure 5.14.

5.3.11 Merge sort (BOTS version)

Sort example sorts a random permutation of n 32-bit numbers with a fast parallel sorting variation [3] of the ordinary merge sort. First, it divides an array of elements in two halves, sorting each half recursively, and then merging the sorted halves with a parallel divide-and-conquer method rather than the conventional serial merge. The speed-up curve for this example is shown in Figure 5.15.

Hence, the translated CilkPlus (and the original OpenMP program) exposes more parallelism, thus narrowing the performance bottleneck in the original OpenMP code.

5.3.12 Sparse LU matrix factorization

Sparse LU computes an LU matrix factorization over sparse matrices. A first level matrix is composed by pointers to small submatrices that may not be allocated. Due to the sparseness of the matrix, a lot of imbalance exists.

Matrix size and submatrix size can be set at execution time which can reduce the imbalance, a solution with tasks parallelism seems to obtain better results [14]. The speed-up curve for this example is shown in Figure 5.16

This algorithm is compute-intensive and has few coarse grained tasks which is a

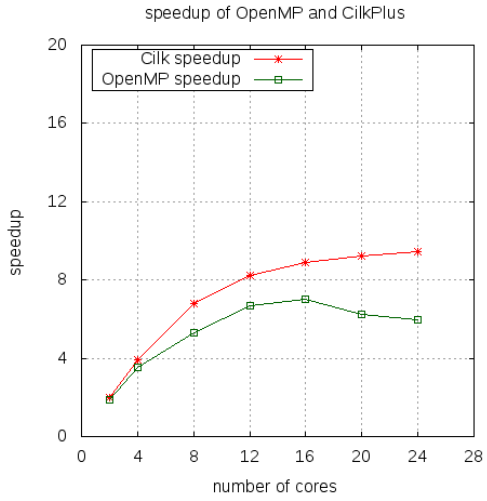


Figure 5.15: Mergesort (BOTS).

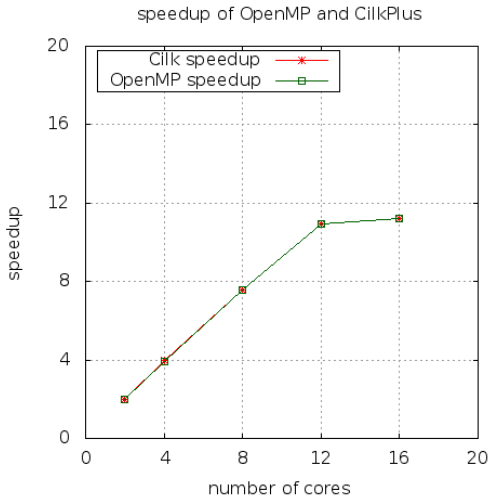


Figure 5.16: Sparse LU matrix factorization.

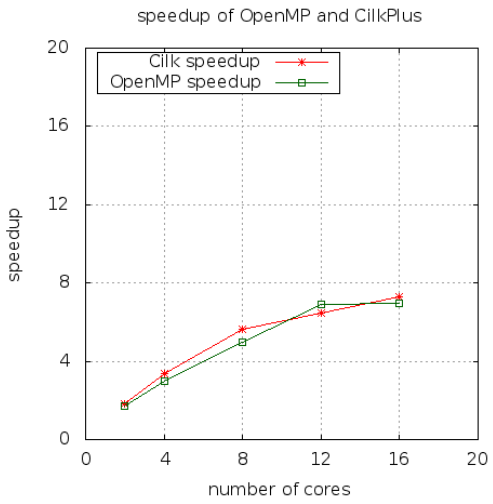


Figure 5.17: Strassen matrix multiplication.

challenge for the implementation to avoid load balance situations. As you can see from the Figure 5.16 both OpenMP (original) and CilkPlus (translated) codes scale almost same. Note that that we are experimenting on a Xeon node with 12 physical cores with hyperthreading turned on.

5.3.13 Strassen matrix multiplication

Strassen algorithm [2] uses hierarchical decomposition of a matrix for multiplication of large dense matrices [27]. Decomposition is done by dividing each dimension of the matrix into two sections of equal size. The speed-up curve for this example is shown in Figure 5.17

As you can see from the Figure 5.17 both OpenMP (original) and CilkPlus (translated) codes scale almost same.

5.4 Concluding remarks

Examples in Section 5.2 suggest that our translators can be used to narrow performance bottlenecks. By translating a parallel program with low performance, we could suspect the cause of inefficiency whether this cause was a poor implementation (in the case of mergesort, where not enough parallelism was exposed) or an algorithm

[2] `http://en.wikipedia.org/wiki/Strassen_algorithm`

inefficient in terms of data locality (in the case of matrix inversion) or an algorithm inefficient in terms of work (in the case of matrix transposition).

For Section 5.3, we observe that, in most cases, the speed-up curves of the original and translated codes either match or have similar shape. Nevertheless in some cases, either the original or the translated program outperforms its counterpart. For instance the original CilkPlus programs for Fibonacci and the divide-and-conquer matrix multiplication perform better than their translated OpenMP counterparts, see Figure 5.6 and Figure 5.7 respectively.

For the original OpenMP programs from FSU, (Mandelbrot Set, Linear solving system and FFT (FSU version) the speed-up curves of the original programs are close to those of the translated CilkPlus programs. This is shown by Figures 5.10, 5.11 and 5.12 respectively.

The original OpenMP programs from BOTS offer different scenarios. First, for the sparse LU and Strassen matrix multiplication examples, original and translated programs scale in a similar way, see Figure 5.16 and Figure 5.17 respectively. Secondly, for the mergesort (BOTS) and FFT (BOTS) examples, translated programs outperform their original counterparts., see Figure 5.15 and Figure 5.14 respectively. For the protein alignment example, the scalability of the translated CilkPlus program and the original OpenMP program are the same from 2 to 8 cores while after 8 cores the latter outperform the former, as shown in Figure 5.13.

Bibliography

[1] CUDA. http://www.nvidia.com/object/cuda_home_new.html.

[2] Ravi Sethi Aho, Alfred V. and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, Massachusetts, 2006.

[3] S. G. Akl and N. Santoro. Optimal parallel merging and sorting without memory conflicts. *IEEE Trans. Comput.*, 36(11):1367–1369, November 1987.

[4] OpenMP ARB. OpenMP. http://openmp.org/wp/.

[5] OpenMP ARB. OpenMP C/C++ 1.0. http://www.openmp.org/mp-documents/cspec10.pdf.

[6] OpenMP ARB. OpenMP C/C++ 2.0. http://www.openmp.org/mp-documents/cspec20.pdf.

[7] OpenMP ARB. OpenMP C/C++ 3.0. http://www.openmp.org/mp-documents/cspec30.pdf.

[8] OpenMP ARB. OpenMP C/C++ 3.1. http://www.openmp.org/mp-documents/OpenMP3.1.pd

[9] OpenMP ARB. OpenMP C/C++ 4.0. http://www.openmp.org/mp-documents/OpenMP4.0.0

[10] OpenMP ARB. OpenMP Fortran 1.0. http://www.openmp.org/mp-documents/fspec10.pdf.

[11] OpenMP ARB. OpenMP Fortran 2.0. http://www.openmp.org/mp-documents/fspec20.pdf.

[12] OpenMP ARB. OpenMP Fortran 2.5. http://www.openmp.org/mp-documents/fspec25.pdf.

[13] Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. The design of openmp tasks. *IEEE Trans. Parallel Distrib. Syst.*, 20(3):404–418, 2009.

[14] Eduard Ayguadé, Alejandro Duran, Jay Hoeflinger, Federico Massaioli, and Xavier Teruel. Languages and compilers for parallel computing. chapter An Experimental Evaluation of the New OpenMP Tasking Model, pages 63–77. Springer-Verlag, Berlin, Heidelberg, 2008.

[15] Guy E. Blelloch and James J. Little. Parallel solutions to geometric problems in the scan model of computation. *J. Comput. Syst. Sci.*, 48(1):90–115, 1994.

[16] Guy E. Blelloch and Margaret Reid-Miller. Pipelining with futures. *Theory Comput. Syst.*, 32(3):213–239, 1999.

[17] R. D. Blumofe and C. E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. *Foundations of Computer Science, Annual IEEE Symposium on*, 0:356 – 368, 1994.

[18] R. D. Blumofe and C. E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM J. Comput.*, 27(1):202–229, 1998.

[19] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999.

[20] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 207–216, New York, NY, USA, 1995. ACM.

[21] James Cooley and John Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

[22] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.

[23] Intel corporation. Cilk++. `http://www.cilk.com`.

[24] Intel corporation. Cilk++. `http://www.cilkplus.org`.

[25] B. Philip D. J. Quinlan, M. Schordan and M. Kowarschik. The specification of source-to-source transformations for the compile-time optimization of parallel object-oriented scientific applications. In *Proceedings of the 14th international conference on Languages and compilers for parallel computing*, LCPC'01, pages 383–394, Berlin, Heidelberg, 2003.

[26] Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguade. Barcelona openmp tasks suite: A set of benchmarks targeting the

exploitation of task parallelism in openmp. In *Proceedings of the 2009 International Conference on Parallel Processing*, ICPP '09, pages 124–131, Washington, DC, USA, 2009. IEEE Computer Society.

[27] Patrick C. Fischer and Robert L. Probert. Efficient procedures for using matrix algorithms. In Jacques Loeckx, editor, *ICALP*, volume 14 of *Lecture Notes in Computer Science*, pages 413–427. Springer, 1974.

[28] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin. Reducers and Other Cilk++ Hyperobjects. In *SPAA '09: Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, pages 79–90, New York, NY, USA, 2009. ACM.

[29] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious Algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, FOCS '99, pages 285–297, New York, USA, October 1999.

[30] M. Frigo, C. E. Leiserson, and K. H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *ACM SIGPLAN*, 1998.

[31] T. D. Han and T. S. Abdelrahman. ThiCUDA: a high-level directivebased language for GPU programming. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units,*, GPGPU-2, pages 52–61, New York, NY, USA, 2009.

[32] Stephen C. Johnson. *Yacc: Yet Another Compiler Compiler*. Computing Science Technical Report No. 32, Bell Laboratories, Murray hill, New Jersey., 1975.

[33] C. E. Leiserson. The Cilk++ Concurrency Platform. In *DAC '09: Proceedings of the 46th Annual Design Automation Conference*, pages 522–527, New York, NY, USA, 2009. ACM.

[34] Charles E. Leiserson. The cilk++ concurrency platform. *The Journal of Supercomputing*, 51(3):244–257, 2010.

[35] Tony Mason Levine, John R. and Doug Brown. *Lex and YACC*. OReilly & Associates, Inc. Sebastopol, California., 1992.

[36] Gene Myers, Sanford Selznick, Zheng Zhang, and Webb Miller. Progressive multiple alignment with constraints. In *Proceedings of the First Annual International*

Conference on Computational Molecular Biology, RECOMB '97, pages 220–225, New York, NY, USA, 1997. ACM.

[37] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 2008.

[38] Arch D Robison. Composable parallel patterns with Intel Cilk Plus. *Computing in Science & Engineering*, 15(2):0066–71, 2013.

[39] Daniel Spoonhower, Guy E. Blelloch, Phillip B. Gibbons, and Robert Harper. Beyond nested parallelism: tight bounds on work-stealing overheads for parallel futures. In Friedhelm Meyer auf der Heide and Michael A. Bender, editors, *SPAA*, pages 91–100. ACM, 2009.