

# Integrating modern parallel techniques in the Tiger compiler

Warren Seine

Technical Report *n°0911*, January 2010  
revision 2158

**English:** Tiger is a language designed as a reference for pedagogical compiler writing. Our C++ implementation of a Tiger compiler takes advantage of well-established practices in program transformation tools.

The multi-core era has made parallelization a requirement in any computer science curriculum. As a support for teaching, our compiler has to evolve and make use of modern parallel techniques.

This report introduces a solution to distribute work in a task-based concurrency model using Intel Threading Building Blocks (TBB) to decouple the programming from hardware specificities.

**French:** Tiger est un langage utilisé à des fins pédagogiques dans l'étude des compilateurs. Écrite en C++, notre implémentation d'un compilateur Tiger profite de techniques éprouvées dans la transformation de programmes.

L'ère du multi-cœur a rendu la parallélisation indispensable dans le cursus d'un étudiant en informatique. Utilisé comme support de cours, notre compilateur doit évoluer et tirer profit des nouvelles techniques de parallélisme.

Ce rapport présente une solution pour distribuer le travail au sein d'un modèle de programmation concurrente par tâche. Nous utiliserons Intel Threading Building Blocks pour nous détacher des problématiques matérielles.

## Keywords

parallelism, Tiger, compiler, program transformation, TBB



Laboratoire de Recherche et Développement de l'Epita  
14-16, rue Voltaire – F-94276 Le Kremlin-Bicêtre cedex – France  
Tél. +33 1 53 14 59 47 – Fax. +33 1 53 14 59 22

[soow@lrde.epita.fr](mailto:soow@lrde.epita.fr) – <http://publis.lrde.epita.fr/2010-01-Seminar-Seine>

## Copying this document

Copyright © 2009 LRDE.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with the Invariant Sections being just “Copying this document”, no Front-Cover Texts, and no Back-Cover Texts.

A copy of the license is provided in the file COPYING.DOC.

# Contents

<b>Introduction</b>	<b>4</b>
<b>1 Parallelism techniques</b>	<b>5</b>
1.1 Origin and breakthrough	5
1.1.1 Software	5
1.1.2 Hardware	6
1.2 Approaches to parallelism	8
1.2.1 Data parallelism	8
1.2.2 Task parallelism	8
1.3 Modern techniques	9
1.3.1 Parallel Extensions	9
1.3.2 Grand Central Dispatch	9
1.3.3 Threading Building Blocks	10
1.3.4 Hierarchical Tiled Arrays	10
<b>2 Within the Tiger compiler</b>	<b>11</b>
2.1 Design considerations	11
2.2 Fitting in the functional architecture	12
2.3 Implementation choices	12
2.3.1 Vertical scaling	12
2.3.2 Horizontal scaling	13
<b>3 Current state and future work</b>	<b>14</b>
3.1 Current state	14
3.2 Benchmarking	15
3.3 Next steps	15
<b>Conclusion</b>	<b>17</b>
<b>Bibliography</b>	<b>19</b>

# Introduction

The compilation process is the center of most software development. As a major tool for the computer scientist, its theory is taught in most C.S. classes and its improvement is still ongoing research in many directions. Recent work in the field of program transformation focus on safety, genericity and performance, and contributions come from many area of computer science as the need for efficient compilers is essential to the industry. The core of compiler design, however, has not changed for a few decades now as pointed out by [Morrisett \(2009\)](#).

The usual construction of a compiler is quite linear. From the *front-end* analyzing the input program to the *back-end* generating the output program, the workflow can easily be seen as a serie of filters with inputs and outputs. State-of-the-art compilers enhance this approach by giving more freedom for extension development ([Lattner, 2002](#)), leading the way to a more open architecture. For instance, static analyzers can be built on top of these extensions. Without going into details, it is a key for better optimizations in the late stages. Ideally, we hope to see auto-parallelizing compilers in a reachable future ([Kulkarni et al., 2008](#)).

Our focus here will not be to improve the generated code, but the internals of the compiler. We want to adapt the historical design of compilers to make maximum use of the threading capacity of modern computers. Parallelizing a compiler is not direct due to the intrinsic linearity of the pipeline and the literature in that field is almost non-existent. Taking an educational Tiger compiler as target, we will try to find which stages can be parallelized and if there is a resulting performance gain. The goal is to eventually fix the bottlenecks of a usual pass.

This report describes a solution to add parallelism within a compiler. Using a performance-oriented threading library, we will suggest and implement different solutions to speed up the program in a safe and functional way.

## Acknowledgments

- Roland Levillain for his pointers to important documents.
- Vincent Ordy for debugging help and proofreading.
- Guillaume Sadegh for typographical error detection.

# Chapter 1

## Parallelism techniques

Traditionally, computer programs have been written for serial computation, copying the linear processing of instructions in chips. This model is still used in numerous end-user applications, but scientific ones tend to follow new approaches in order to distribute work. Parallel computing is a tremendous solution to improve the performance of a program by breaking the problem into multiple smaller problems.

### 1.1 Origin and breakthrough

Parallel programming is a large field of research which was started by [Dijkstra \(1965\)](#). We will specifically focus on the multi-core class of parallel computers — distributed computing is away from the point. As of today, it is gaining more and more attention with the arrival of cheap processors with multiple execution units.

#### 1.1.1 Software

First, let us introduce a couple of domain-specific terms to avoid any ambiguous usage later. In parallel computing, *process* might designate several objects. The most common usage represents a running program in memory. It might also be a user-maintained stream of instructions.

**Heavy processes** are instances of a computer program, consisting of one or more threads, that is being sequentially executed. Their creation is the basics of any modern operating system. Inter-process communications are used to share data between processes.

**Light-weight processes** are commonly called *threads*: they are forks of a program designed to run asynchronously ([Figure 1.1](#)). This solution is a widespread model for concurrent programming. In most implementations, they are manually started by the developer and monitored by

a system-dependent scheduler. Dialog between threads happens in shared memory: access is fast but requires control policies to avoid typical problems such as race conditions.

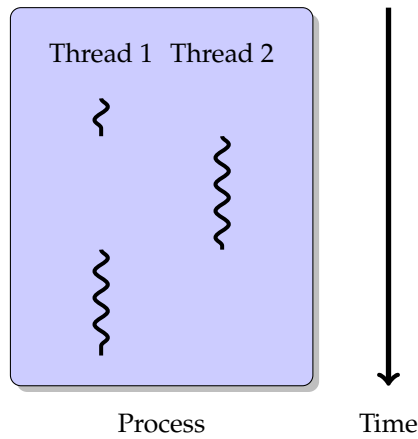


Figure 1.1: Multithreaded process

Both heavy and light-weight processes can be manipulated with libraries to spawn more processes or to change priorities.

### 1.1.2 Hardware

The hardware part of parallel programming is essentially how processing units are organized; it is a usual 1-on-2 or 2-on-1 dilemma. We will not address time-sliced scheduling or virtual processors such as Intel's Hyper-threading because they mostly clone physical designs.

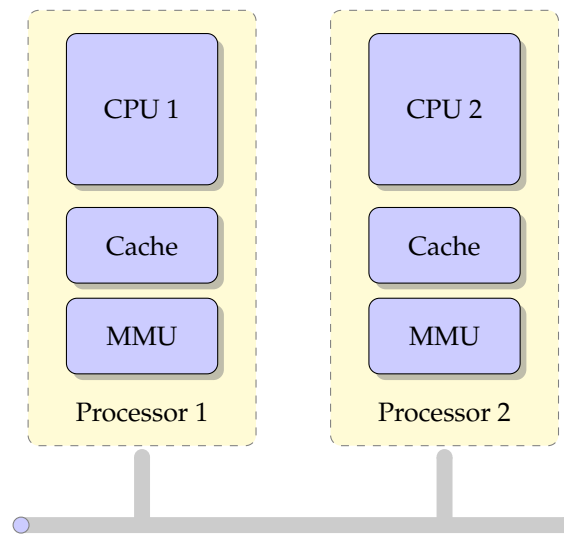


Figure 1.2: Single-core multiprocessors

Multiprocessor systems (Figure 1.2) were the first kind of parallelizable hardware. They contain multiple CPUs that are not on the same chip. Whether they are found on different boards or on the same one, they are connected through a communication interface. They are less complex but less efficient than tightly-coupled multi-core systems, because they are essentially single chip CPUs connected together. Due to their incapacity to operate fast communication between each other, they are better suited for running heavy processes simultaneously than for threading a single program.

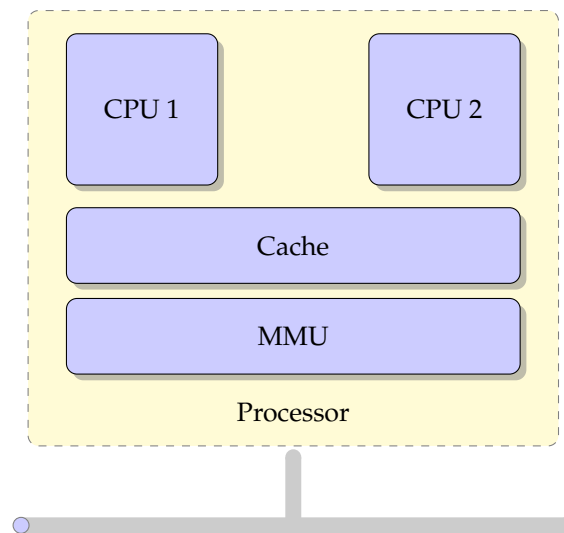


Figure 1.3: Multi-core processor

Multi-core processors ([Figure 1.3](#)) are a family of processors that contain any number of multiple CPUs on a single chip (usually 2, 4, or 8). The challenge with multi-core processors is in the area of software development. Performance speed-up is directly related to the way the application was written or optimized. Indeed, the operating system will distribute the light-weight forks of the process between cores. Theoretically, the more multithreaded the code, the better performance.

New hardware designs in the horizontal layer have a promising future as we have reached physical limits of frequency scaling. They have already proven they are able to catch-up with Moore's law. The bad side of changing the architecture is that the developer has to deal with various targets that may need a completely different design for the program. Thus, patterns may come handy. The holy grail is automatic and scalable parallelization. Some ideas ([Kulkarni et al., 2007](#)) are attractive but do not exhibit a generic solution to the problem.

## 1.2 Approaches to parallelism

### 1.2.1 Data parallelism

Data parallelism is the most usual and intuitive sort of parallelism. It focuses on distributing the data across different parallel computing nodes. The definition of *node* scales: it may refer to bits (for a SIMD device), to items in a list (for loop-parallelization), to cores (in multi-core systems) or to actual computer (in a cluster environment).

Long, repetitive, functional computations are well-suited for this kind of parallelism. A scholar, but actually used, example is the matrix sum: in a divide-and-conquer manner, the matrix is splitted across nodes, each one computing a local result, then the results are *reduced* with a `join` primitive.

MapReduce architectures such as Google's ([Dean and Ghemawat, 2004](#)) are based on a functional data-parallelization model to compute large data sets over a cluster of machines. At a lower level, the architecture fits in the domain of GPGPU computing.

Data-parallelism contrasts with a more abstract vision of concurrency, task-based parallelism.

### 1.2.2 Task parallelism

Task parallelism is a possible approach for improving the performance of long routine pipeline. It is quite easy to understand and abstracts the underlying architecture. Instead of assigning a piece of data to each node, one would address a function. Therefore, nodes become task processors for each element of the input data. Nodes are often referred to as *workers*. They are dedicated and may be tuned for a special job.

A consequence is that data will pass through many threads to get the final result whereas a data parallel algorithm would allocate one thread to complete the work. It is then harder to scale up when the number of processors increase. A solution implemented in most libraries is the dynamic allocation of tasks, using a customized Thread Pool design pattern. However, some tasks may be orders of magnitude more time-consuming than others, which cannot easily

be guessed, even at run-time, and may still be a bottleneck in a pipeline of tasks.

Most real programs fall somewhere on a continuum between task parallelism and data parallelism (Andrade et al., 2009). We will see in Section 2.2 how we can make use of both approaches in our compiler.

## 1.3 Modern techniques

The POSIX standard defines a threading library. The API states functions and data types that a compliant operating system must provide. Implementations of this specification exist in many environments. Moreover, the upcoming C++, known as C++1X, introduces parallelism primitives as part of the standard library. Compiler vendors are free to implement those specifications and to possibly back-port this library.

In any case, the developer wants sugar over a thread API, to make it simple and to abstract the notion of parallel process. In this section, we will go over such libraries and see how they fit in our problem.

### 1.3.1 Parallel Extensions

*Parallel Extensions* is a Microsoft/Microsoft Research project (Leijen et al., 2009) developed for the .NET environment. It provides two libraries, for data-parallelism and task-based parallelism. The data-parallelism library, *Parallel LINQ*, focuses on database queries. The *Task Parallel Library* (TPL) is more of our concern: modeling tasks as functions, it provides a convenient solution for abstracting thread-level parallelism. The C#-specific *delegates* are a form of  $\lambda$ -functions which can be passed to any TPL algorithm taking a task as argument. Synchronization primitives and classes for *future* values are also provided.

An important design choice is that no concurrency guarantee is made: the run-time system will choose whether to start the task asynchronously or not, depending on the architecture of the machine and the load balancing. Even if a C++ version exists, the show-stopper is the requirement for Microsoft's CLR type system, our compiler being written in portable C++.

### 1.3.2 Grand Central Dispatch

*Grand Central Dispatch* (GCD) is an Apple technology used to optimize application support for systems with multicore processors and other symmetric multiprocessing systems. Like TPL, it provides a form of task parallelism, with the major difference of coordination being handled at a system level, giving the operating system a global view for load balancing. Function objects or *blocks* (Apple's language extension for closures) may be given as tasks.

A specificity of GCD is the use of user-maintained queues for storing tasks. All the items in these queues are processed serially, resolving dependencies on other tasks first. An open-source version is available, and ports to non-Darwin systems are on their way.

### 1.3.3 Threading Building Blocks

Intel's attempt to task parallelism, *Threading Building Blocks* (TBB), targets C++ developers who want a modern approach to speed up their programs. By design, it is similar to TPL and uses standard C++ function objects to describe tasks. TBB has routines for common task parallelism (`parallel_for`, etc.) but mainly focus on pipeline parallelism. Typical applications include real-time stream processing (Navarro et al., 2009).

TBB implements *task stealing* to balance a parallel workload across available processing cores. When a processor becomes idle, TBB looks for a data chunk that isn't currently being worked on, and "steals" that task assignment from the processor it had originally been assigned to. In practice, it is an elegant solution to the problem mentioned in Section 1.2.2.

Also, TBB uses templates thereby relying on compile-time polymorphism that can be more efficient than traditional run-time polymorphism since modern C++ compilers are tuned to minimize any abstraction penalty arising from heavy use of templates of the C++ standard library and TBB. Recent improvements were made to support the upcoming C++1x language, in particular  $\lambda$ -functions.

The library has the advantage of being an open-source, well-supported, widely available and portable solution to task-parallelism. Consequently, it fits in our problem's environment and scope.

### 1.3.4 Hierarchical Tiled Arrays

*Hierarchical Tiled Arrays* (HTA) are a proposed parallel technique specifically designed for sparse matrix multiplications (Fraguela et al., 2004). The idea is to recursively partition arrays and then map a function to the HTA to be distributed between the computing nodes. Originated from Matlab's *cell* objects, the paradigm is implemented in C++ through language extensions.

HTA's extension is a convenient solution for describing a data type that scales. It is not incompatible with the previous libraries (in theory) and might give extra performance boost.

Quoting Farnham (2007):

HTAs appear to be an ideal data type for parallel processing, and the structure of HTAs and TBB's methods of processing tasks in parallel appear to be a perfect match.

Indeed, TBB's task stealing model is especially efficient on recursive data types: algorithms descending in a branch guarantee they will not use another branch of the tree.

## Chapter 2

# Within the Tiger compiler

### 2.1 Design considerations

Tiger is our reference language for teaching compiler construction. It is often described as a simple, well-designed, easy-to-learn programming language and is based on reliable programming language origins. Used as a support for a compilation class, it is made of functional modules that follow the usual steps of a compiler from the front-end to the code generation. Our implementation of a Tiger compiler (TC) takes advantage of tried and tested practices (Abstract Syntax Trees, intermediate representation, etc.). It is written in C++ and is a nifty way to introduce to advanced programming solutions such as design patterns and template metaprogramming. Recent work in the compiler includes new ideas: using concrete syntax to make AST generation and rewriting more natural ([Sigoure, 2008](#)).

The technique allows the compiler developer to write less code to declare desugaring rules. This breaks the usual design of compilers, as the syntax analysis may be repeated after a desugaring pass.

The data structure that represents the compilation workflow is a graph with possible cycles. We can directly conclude that a simple pipeline will not suffice to express all the dependencies between tasks.

Another important consideration is that the compiler is a reference for the compilation class: the compiler can be stopped at any step to see the current representation of the program. This simple idea makes complex the implementation of a simple `make`-like dependency checker, because `make` makes all possible attempts to generate the requested file. Our system needs to simulate a complete pass and stop at a specific step.

Futhermore, `make` has the advantage of being able to compute the whole dependency graph at startup, but in the current model, dependencies are on tasks, not on objects. And since the number and nature of tasks in the compiler are dynamically computed, it is hard to have an entirely parallel pipeline. For instance, we cannot know how many canonizations will happen until we know how many functions are declared. We will detail this problem in [Section 2.3](#).

As we do not intend to make use of the compiler for developing actual software, performance

is not a top priority in the project. We are more likely to focus on feature additions and genericity improvements. Adding parallelism inside the program is an addition that might actually improve the performance, but the aim is to teach students safe parallel programming.

## 2.2 Fitting in the functional architecture

The Tiger compiler was designed with the idea of being a model for software design. Students who follow the compilation class have to write their implementation of the compiler, step by step, from the lexical analyzer to the register allocator. The compiler is thereby divided into slices, and each slice is a module. Each module correspond to a task, which can be called in the compiler command line. The modular architecture helps to keep a clean functional model, letting no room for undesirable side effects.

At the program start, the target (defaults to a full pass) is added to a list of “to-do” and a topological sort is done on dependencies. This approach is safe and would easily serve a task concurrency model.

In the front-end, many traversals are implemented as visitors. The VISITOR design pattern (Gamma et al., 1995) decouples the algorithm (the traversal) from the data (the AST). Since they all inherit from a `Visitor` abstraction, it is easy to apply parallelism to any traversal. In practice, side effects reveal themselves, sometimes breaking the compiler. Algorithms may require a parallel rewrite.

The goal here is to add performance-oriented parallelism using light-weight threads. TBB is the library of choice and can be integrated into our task system without breaking the design.

## 2.3 Implementation choices

The task system needs not be changed right now. Keeping in mind that parallelism is an advantage – not a need – is essential. A toy implementation of a the parallel task system has been started, under the name of `tc-prototype`. Reusing the principles of design of the Tiger compiler, it is a taking advantage of TBB to map tasks to parallel `filters` (TBB’s word for pipeline stages).

### 2.3.1 Vertical scaling

Using TBB’s `pipelines` is simple: you define your tasks as classes inheriting a `filter` class. Then you declare your `pipeline` and add every task to it. A task is static: its arguments have to be given on construction. The only thing left is to run the process with the maximum number of tokens that can be in flight. Conceptually, *tokens* flow through the pipeline. In a serial stage, each token must be processed serially in order. In a parallel stage, multiple tokens can be processed in parallel by the stage. They are an indicator for the level of parallelism the application needs. Theoretically, it could be guessed at run-time, but in the current implementation of TBB, it is required to specify this number.

As said earlier, there are special cases where the developer does not know in advance how many output tokens will flow out of a filter. Indeed, in the compiler, we use a list to handle function declarations, and for each function, multiple transformations will happen. This maps perfectly with the concept of pipelines: we can produce as many tokens as functions and let the run-time deal with the low-level thread issues. The problem is that the number of functions obviously depends on the source file being processed, and yet TBB needs the number of tokens entering/leaving a stage to be constant. It cannot be bypassed in a static way. [Navarro et al. \(2009\)](#) propose a solution based on a hybrid TBB/POSIX Threads combination, associated with a queue to handle the excess of tokens produced. Another solution is to dynamically instantiate pipelines when the program knows how many tokens there will be. I chose this proposition, and had success implementing it in the prototype. No port has been made to the `tc-tbb` branch yet.

The dynamic approach for pipeline parallelism is not usual since TBB `pipelines` are typically designed for stream processing. We cannot know the cost of dynamic allocation of those objects until a working implementation is made in the branch.

### 2.3.2 Horizontal scaling

The horizontal scaling is a synonym for data parallelism: at our level, that is loop-parallelization, or `parallel_for`-like algorithms. In TBB, this is a little different: the function mapped is implemented as a task, and therefore uses the task stealing paradigm previously described in [Section 1.3.3](#). An inescapable effect is the performance will only improve if the number of items is high enough; that is, if the input file contains many definitions instead of a single giant function.

## Chapter 3

# Current state and future work

### 3.1 Current state

The current state of the `tc-tbb` branch of the `tc` compiler mainly consists of the addition of a `parallel_foreach` function (Figure 3.1) that is being used in a generic `tree::Visitor` abstraction. The `Visitor` is inherited in multiple places. With `const Visitor`s, the transposition is immediate: no need to protect access to shared resources. Some others require more changes.

---

```
// Function object used by parallel_foreach
template <typename Container, typename Functor>
struct ParallelApply
{
    ParallelApply(Container& c, Functor& f) :
        container_(c),
        functor_(f)
    {
    }
    void operator()(const tbb::blocked_range< size_t >& r) const
    {
        for (size_t i = r.begin(); i < r.end(); ++i)
            functor_(container_[i]);
    }

    Container& container_;
    Functor& functor_;
};

// Apply 'f' to all the members of 'c', and return it.
template <typename Container, typename Functor>
inline Functor&
parallel_for_each (Container& c, Functor& f)
{
    tbb::parallel_for(tbb::blocked_range< size_t >(0, c.size(), 100),
        ParallelApply<Container, Functor>(c, f));
    return f;
}
```

---

Figure 3.1: A `parallel_foreach` implementation.

The code is here simple to be introduced in any loop and does not require a lot of changes in the existing code.

## 3.2 Benchmarking

The interesting part of the report is to see how much we gain by using a single but intensively used optimization. Figure 3.2 shows a comparison <sup>1</sup> of the “master” and “tc-tbb” branches of the compiler.

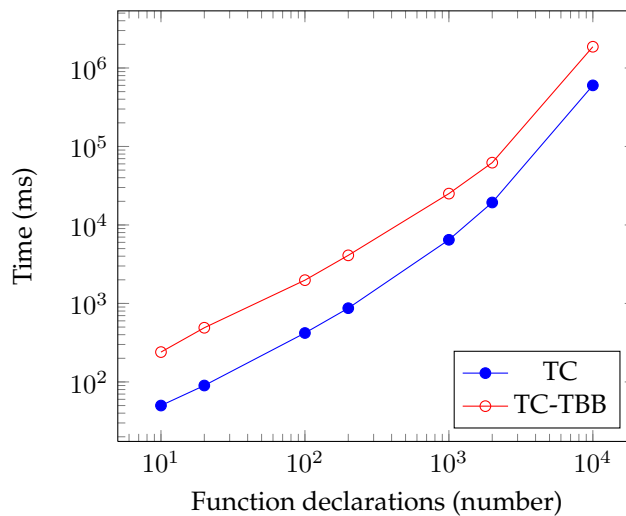


Figure 3.2: Compilation times with big input files

The benchmark axis are shown in a log scale. Curves are near linear, which means that the compilation time is exponentially proportional to the number of function definitions. Functions are 10 lines of standard Tiger code.

Surprisingly, and sadly, there is a considerable loss of performance (put simply, three times slower). With small files, we could have expected a loss caused by the overhead of the library. With big files however (> 100000 lines), the loss is quite worrying. As a side note, I expect some kind of bug within the algorithm that would cause this regression.

## 3.3 Next steps

The integration of TBB in Tiger is dirty. In order to protect ourselves from the side effects in a particular module, no parallel call is made in this module, requiring an ugly check in the core

<sup>1</sup>Machine: 3.06 Ghz Intel Core 2 Duo

of the `Visitor`. Solutions are to lock the data structure causing the race condition or, better, to rewrite the algorithm in a thread-safe way.

There is plenty of room for improvement. We want the parallel code to be easy to disable so it can be integrated as an option in the project cycle. We also need to fix constness at some point, which is actually harder than thought.

The task system is still untouched from the original branch, but a revamping based on the prototype described in [Section 2.3](#) is an interesting start. New ideas to parallelize other stages of the compiler are welcome. We already have been suggested some optimizations:

- Concurrent parsing (`imports`).
- Concurrent tree manipulation and rewriting.

# Conclusion

Parallelization is a solution for boosting performance and scaling up easily, but until recently, concurrent development was painful. The recent techniques have proven to give excellent results while abstracting the underlying system calls. TBB is one of the most promising libraries in the field of multicore parallelism.

We have seen, in this report, different techniques to make the most of such a tool and have tried to apply them to a pedagogical C++ program, a Tiger compiler. Unfortunately, the full integration of TBB is not complete, and the temporary results did not show any encouraging performance gains. (Fixing the implementation is our short-term objective.) From a design point of view, modular and pure-functional programming is the key to efficient parallelism. In fact, we have experienced trouble caused by side effects.

The long-awaited auto-parallelizing compiler is not expected in a near future, but this is one active field of the research led by a strong demand from the industry. Until then, concurrency is still part of the basics of Computer Science teaching.

# Bibliography

- Andrade, D., Fraguela, B. B., Brodman, J., and Padua, D. (2009). Task-parallel versus data-parallel library-based programming in multicore systems. *Parallel, Distributed, and Network-Based Processing, Euromicro Conference on*, 0:101–110.
- Dean, J. and Ghemawat, S. (2004). MapReduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*, pages 10–10, Berkeley, CA, USA. USENIX Association.
- Dijkstra, E. W. (1965). Solution of a problem in concurrent programming control. *Commun. ACM*, 8(9):569.
- Farnham, K. (2007). Hierarchically Tiled Arrays and Threading Building Blocks. <http://software.intel.com/en-us/blogs/2007/09/28/hierarchically-tiled-arrays-and-threading-building-blocks/>.
- Fraguela, B. B., Guo, J., Bikshandi, G., Garzarán, M. J., Almási, G., Moreira, J., and Padua, D. (2004). The Hierarchically Tiled Arrays programming approach. In *LCR '04: Proceedings of the 7th workshop on Workshop on languages, compilers, and run-time support for scalable systems*, pages 1–12, New York, NY, USA. ACM.
- Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
- Kulkarni, M., Carribault, P., Pingali, K., Ramanarayanan, G., Walter, B., Bala, K., and Chew, L. P. (2008). Scheduling strategies for optimistic parallel execution of irregular programs. In *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 217–228, New York, NY, USA. ACM.
- Kulkarni, M., Pingali, K., Walter, B., Ramanarayanan, G., Bala, K., and Chew, L. P. (2007). Optimistic parallelism requires abstractions. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 211–222, New York, NY, USA. ACM.
- Lattner, C. (2002). LLVM: An infrastructure for multi-stage optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL. See <http://llvm.cs.uiuc.edu>.
- Leijen, D., Schulte, W., and Burckhardt, S. (2009). The design of a Task Parallel Library. *SIGPLAN Not.*, 44(10):227–242.
- Morrisett, G. (2009). Technical perspective: A compiler's story. *Commun. ACM*, 52(7):106–106.

Navarro, A., Asenjo, R., Tabik, S., and Cascaval, C. (2009). Analytical modeling of pipeline parallelism. In *PACT '09: Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 281–290, Washington, DC, USA. IEEE Computer Society.

Sigoure, B. (2008). Run-time concrete-syntax program-transformation in general purpose languages. Technical report, EPITA Research and Development Laboratory (LRDE).

# List of figures

1.1	Multithreaded process	6
1.2	Single-core multiprocessors	7
1.3	Multi-core processor	7
3.1	A <code>parallel_foreach</code> implementation.	14
3.2	Compilation times with big input files	15