First Year PhD Report
2010/2011

Alessandro Pellegrini
DIS - Dipartimento di Informatica e Sistemistica
Sapienza, University of Rome
Via Ariosto 25 - 00185 - Rome - Italy

October 24, 2011

In this document I present the research activities I have followed through during the first year of PhD in Computer Science, along with a description of the research area and fields I am focusing on and the key problems I am addressing. A State of the Art as comprehensive as possible is provided, trying to present the various disciplines I am touching. In addition, I am presenting my current activities and some results achieved so far.

1 Research Area, Problem Overview and Current Technological Trend

My Research focuses on Code Parallelization. My interest is both in framework design/development and methodologies definition, with a special attention to runtime execution layers as a support to automatic parallelization. A special attention is devoted to trasparency, which is the degree of freedom the user is given to fully exploit all the semantic constructs offered by programming languages, without the need to adhere to a particular programming model or to use ad-hoc markers for supporting the code parallelization.

From a methodological point of view, the interest in parallelization techniques comes out from the fact that they can be regarded as a cross-disciplinary and standard way to achieve high performance execution of important applications. Of course, this research is strongly connected with the analysis of the effects generated from the usage of multicore vs large scale computers, which are the current technological trends becoming the de-facto standard of the future.

Over the past 45 years, the total number of transistors available on a microchip has doubled every 18–24 months, a trend which is known as Moore's law [38]. This yielded a propotional increase in a single processor's clock speed which, in the past decades, was bringing an enhancement in the computation speed which researchers, developers and users were receiving for free, whenever they were upgrading their hardware. Improvements in algorithms and/or code optimizations were not strict requirements to be pursued, because — as the time was passing by — the software was working more and more efficiently.

In Figure 1 we see, with reference to Intel processing units, how this trend has been almost the same until year 2003. Later on, although the number of transistors is presenting the same trend, clock speed increase has stalled. This is connected with the other curves in the plot: In fact, 130 W of power consumption in a processor is considered an upper bound [49], and generates a *clock-frequency wall*. This is connected to physical constraints, as the power consuption is proportionally related to the frequency [28]:

$$P = CV^2 f$$

therefore, an increase in the clock frequency would create an unacceptable power consuption.

The industry has therefore approached physical limits in the computational power of a single processing unit, altough the number of transistors per area unit is still increasing. Nevertheless, the demand for continued improvements

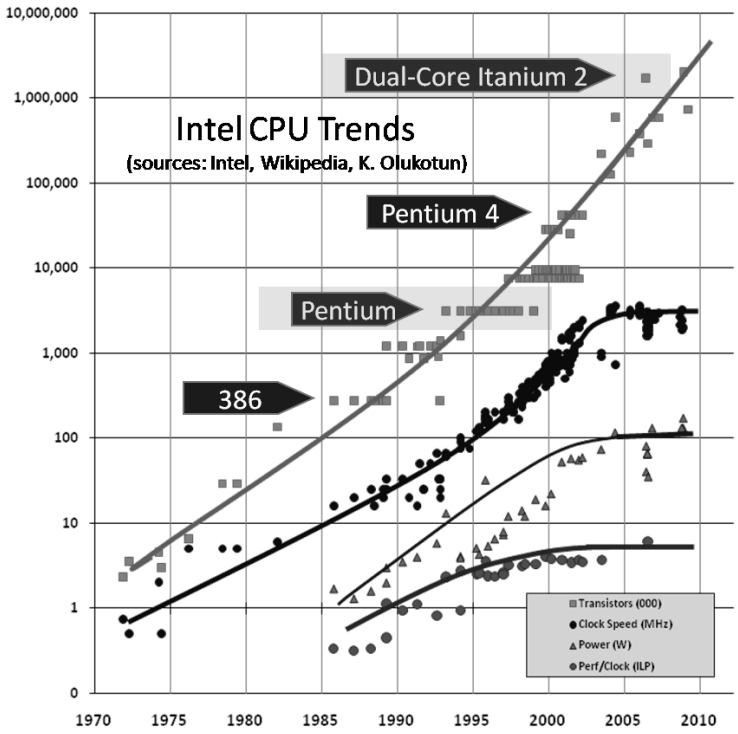


Figure 1: Moore's Law

in computation speed is still there, and this is driving hardware manufacturers to switch to the multicore technology, where chips with multiple processing units are being produced.

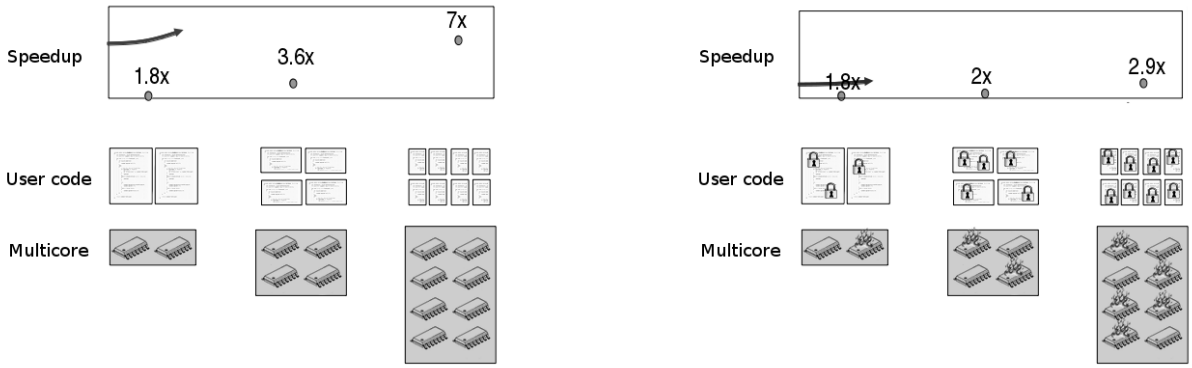
This opens the way to a heavy usage of concurrent programming, where the user is expected to partition its code and make it run on different processing unit istances. What the user would expected is that, as in the past years, doubling the computational power would consequently produce a proportional increase in the speed. This scenario is shown in Figure 2(a).

Unfortunately, this is not the real scale up: In fact, to ensure correctness of the result, a program cannot execute any instruction in parallel. In particular, every concurrent access on objects not serialized by any underlying layer (e.g., memory accesses) might produce unexpected (inconsistent) results. To avoid this, several techniques have been proposed, the simplest of which is the *locking* primitive, which enables one instance of the parallel program to read/modify data only if no-one else is performing the same action at the same time. As presented in Figure 2(b), this produces a performance increase which is no longer described by Moore's Law. In particular, we can see that having eight times the computational power of a single processor, produces a real speedup which is even lower that a factor 3, because many computational resources are burnt in locking operations to produce consistent results.

In particular, there is a maximum speedup that a program being parallelized can reach. This is stated by Amdahl's Law [3]: Every program has some fraction of its code (namely P) which can be run in parallel, since in that portion there is no conflict on data accesses. In the optimal case, given that we have S processing units which can run the program in parallel, the maximum achievable speedup for the entire program is:

$$S_{max} = \frac{1}{\Big(1 - P\Big) + \dfrac{P}{S}}$$

Usually, the factor S is lower than the number of available processing units which can run concurrently the program, because of secondary effects and intercommunication. As shown in Figure 3, if 80% of the program is paral-



(a) Expected Speedup

(b) Real Speedup

Figure 2: Parallel Speedup

lelizable and S is equal to the number of processors, then the maximum achievable speedup is:

$$\lim_{S \to +\infty} S_{max} = \frac{1}{1 - 0.80} = 5$$

Therefore, no matter how many processors are used, this program can never run more than five times faster.

The maximum theoretical speedup is anyway extremely difficult to be reached: In fact, a programmer should find a well-balanced trade-off between two main factors. First, data structures and/or tasks in a serial program must be decomposed — or mapped — into parallel sections. Second, synchronization and coordination primitives must be used in the code in order to produce a parallel program which runs correctly. Mixing these two aspects is a difficult procedure, which requires a careful balance between a program's communication and computation parts. When programming for a single-processor architecture, a common efficiency metric is the number of operations the program will execute: Minimizing them should make the program run faster.

But if counting and trying to reduce the number of operations in a serial program is generally straightforward, doing so when a program is moved to a parallel machine does not guarantee an optimal performance. In addition, the problem is compounded with the presence of multi-level memory hierarchies. In fact, a serial program which is moved to a parallel processing architecture can significantly suffer from secondary effects due to enhanced contention on memory devices. This means that beyond the theoretical speedup limit dictated by the longest serial path — as espressed by Amdahl's Law — the real speedup is constrained by the architectural-dependent interprocessor communication. This poses an extreme importance on how the application and/or its distributed parts are mapped on the multiple processing elements, thus showing how the performance bounds of a parallel algorithm strongly depend on how data and computation units are distributed; this dependance makes it even more difficult to estimate and optimize the performance of a serial program being transformed into a parallel one. In fact, this changes deeply the algorithmic details of a program: Often, the most efficient serial algorithm is not the most efficient parallel algorithm, and yet the efficient parallel one is usually less comprehensible by humans.

In addition to this, given that the parallelization is performed in the best way, Amdahl's Law emphasizes that the serial fractions of an applications are the ones that induce the biggest brake in the execution speed. So, if we want to break this law and exploit all the beneifts provided by the new architectural trends (such as multi-core and hyper-threading computing), we have to develop new methodologies and techniques which are able to produce a speedup even in the serial fractions of the code.

To achieve all these goals in the most effective way, a methodology which is accessible to the massess must be provided. In fact, parallel programming can no longer be the privilege of a restrict set of experts: The slowdown of

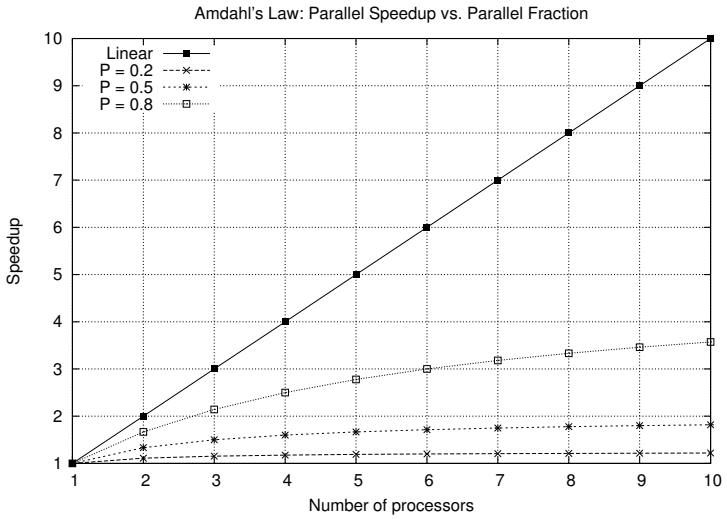


Figure 3: Amdahl's Law

Moore's law and the need to process increasingly large data sets are driving the computer hardware community to develop multicore chips in addition to the already prevalent commodity cluster systems, for scientific purposes. In addition, these architectures are becoming dominant in all areas of computing, from personal computers, to realtime signal processing. Therefore, nowadays, a larger set of programmers and algorithm developers must take advantage of these computer architectures. As parallel processors become ubiquitous, the key question is: "*how do we convert the applications we care about into parallel programs?*"

The need for parallel processing is evident in real-time computing, scientific and non-scientific areas. Scientific computing in high-level languages requires high-fidelity simulations and computations that can be achieved by increasing the number of parameters and the size of the datasets. Similarly, in the field of embedded real-time computing the replacement of analog receiver technology (in sensor arrays) by digital technology requires faster digital processing capabilities at the sensor front end. Additionally, the migration of more and more post-processing, such as tracking and target recognition, to the sensor front end necessitates increasingly powerful processing architectures. Finally, multicore processing units are now a consolidated reality on out-of-the-shelf machines, giving normal users a relevant computational power which is most of the times wasted or not fully exploited.

Transparency is the answer to this need of wide-spread need for parallelization. In order to provide the user with full transparency, we have to develop tools which are able to understand at their best what the programmer is really asking the machine to do, that is tools which are able to understand both the program and the machine they will be making the program run onto. In addition, to make these tools general purpose, a low-level approach should be used: Every programs talks with the underlying hardware with machine code, therefore producing tools and methodologies which are able to directly handle final executable files and and their code will produce a benefit for programmers of multiple languages. This means that if we want to produce a general-purpose tool, we cannot move from a language of a level higher than machine code, or its 1:1 companion, the assembly.

Whether multicore computing is a choice or a need, it is important to reduce the complexity of parallel programming and develop frameworks for supporting the task. This is the time for us to develop techniques which will prevent parallel programming sub-optimizations from becoming a habit, or before parallel programming becomes irreparably the privilege of a small and restricted portion of programmers. Furthermore, it is important to provide the programmers with transparency as complete as possible, providing efficient frameworks and tools which are able to interpret the code's and the machine's needs, in order to give the highest possible performance. Only in this way, the doors of high performance computing will be opened to everyone.

2 State of the Art

Transforming a sequential program into a parallel one is a well-known problem. Designing parallelism facilities or creating parallelism into programs involves a number of related decisions. Is parallelism expressed explicitly or implicitly? Is the degree of parallelism static or dynamic? How do the individual processes interact? How do they communicate data and synchronize with each other? Answers to these questions have been pursued for decades in literature, proposing solutions which address the problem from different points of view. In this section I present a survey on the most significant of them.

2.1 Models of Sharing Communication

As stated in Section 1, interprocess communication is one of the main bottlenecks of parallel programming. The two basic communication means between processes are *accessing shared variables* and *sending messages*. Shared memory is generally considered easier to program, because communication is one-sided: processes can access shared data at any time without interrupting other processes, and shared data structures can be directly represented in memory.

Message passing is more cumbersome, requiring both a two-sided protocol and marshalling of non-trivial data structures, but it is also more popular on large-scale machines because it makes data movement explicit on both sides of the communication. MPI [15] provides primitives for both point-to-point communication and broadcast (reductions, global operations, ...). Systems like this have popularized the *Single Program Multiple Data* (SPMD) programming paradigm, in which a single program executes in each of a fixed number of processes that are created at startup and remain throughout the execution. The parallelism is *explicit* in the parallel system semantics, in the sense that a serial, deterministic abstract machine cannot descrive all possible behaviours in any straightforward way. The SPMD model offers more flexibility than an implicit model based on data parallelism or automatic parallelization of serial code, yet leaves to the user the full responsibility of performance.

2.2 Synchronization Algorithms

During the last decades, attempts to find methodologies for supporting parallel execution of code has lead to a strong interest in *non-blocking* synchronization algorithms, which allow asynchronous and concurrent access to data objects, yet guaranteeing consistency in the updates, through the exploitation of atomic operations like *compare and swap* (`CAS`) and *load-linked/store-conditional* (`LL/SC`) [8, 30, 26, 23] which avoid costly primitives like *spinlocks* and are safe from drawbacks like *deadlocks*, *priority inversions*, and *convoying*.

Depending on the progress guarantee they give, non-blocking synchronization algorithms can be grouped into three main categories:

- *Wait freedom* [23] – in a set of processes all contending on the same set of data objects, they *all* progress in finite execution time steps. This is the strongest property for a non-blocking algorithm.
- *Lock freedom* – in a set of processes all contending on the same set of data objects, *at least one* progresses in finite execution time steps. This property ensures no deadlock will be present, but does not guarantee the absence of *starvation*.
- *Obstruction freedom* [22] – in a set of processes all contending on the same set of data objects, one progresses in finite execution time steps only if it executes by itself for long enough[1]. This property ensures no deadlock will be present, but *livelocks* may occur if a set of processes keep preempting or aborting each other's atomic operations.

[1] Another definition of this property is that a process progresses if it encounters no contention on data. This definition is more constraining, as if a process is suspended when holding a resource, there is data contention. Obstruction freedom can be guaranteed as well by removing the ownership on a resource if a process is suspended.

Transactional Memories [24, 47, 37, 12] are a generic non-blocking synchronization construct, which have been studied for over a decade and offer — in their software version — both obstruction-free [25, 20] and lock-free [47, 14] implementations. They allow a correct sequential object to be mapped automatically into a correct concurrent object, using the definition of a *transaction*, which is a sequence of instructions which atomically modifies a set of data objects. To fully exploit transactions, the user must explicitly mark in the code which regions have to be executed atomically. Later on, at runtime, the underlying framework uses a system-wide declaration of a transaction's update intention, to notify any other concurrently-running transaction that an update to a particular object is about to be performed. If the programmers were not compelled to use markers to define which is the beginning and the end of each transaction, the runtime library should have been burden with the task of considering the whole application as a transactional program, with an execution cost which wouldn't have been neither negligible nor sustainable.

Event-Driven Programming [36] is a programming paradigm in which the flow of the program is determined by events (generated from the users or from other software modules) or messages from other programs or threads. In event-driven programming the application has a main loop in which events are selected (according to some scheduling policy) and then handled by specified routines. In embedded systems the same is achieved using interrupts instead of a constantly running main loop. Event-driven programs can be written in any language, although the task is easier in languages that provide high-level abstractions. In simulation environments, event-driven programming is used to implement simulation models through a reduced-size set of APIs offered by simulation platforms. Events scheduling can be based on the *safety* property of events (i.e., an event will not cause any time-causality inconsistency), or using an optimistic approach [29], where events are executed regardless of their safety, and if an inconsistency is later found, a rollback operation (supported by state log/restore operations) is performed, bringing the state back to consistency. The synchronization guarantees offered by event-driven programming depend on the scheduling algorithm chosen, moving from wait freedom in case of a Round Robin scheduler, to a lock freedom when relying on a Lowest Timestamp First one.

2.3 Frameworks and Tools

Parallelizing Compilers [50] and Optimization Tools [17, 2] try to identify within the application which are the istructions (or instruction blocks) which bring an inborn parallelism [21], and therefore try to rearrange the execution patterns of such parts so that they can be concurrently executed onto several processing units. The techniques which are used by these compilers are well-suited for vector machines, yet they strongly rely on application sources [44] or Abstract Syntax Trees [51], and commonly forget to take into account architecture-dependent aspects in an attempt to be as general-purpose as possible. In addition, a common metric to decide whether a parallelization is convenient or not is the comparison between the theoretical execution time of the code after its parallelization with respect to its sequential execution time. As stated before, such a prevision is not easy to do programmatically. This makes these tools really efficient only in specific scenarios, and — somewhat counterintuitively — the code does not always benefit from parallel execution, as secondary effects associated with using multiple processors can burn all the expected speedup.

A lot of effort has been put into the development of analythical theories aimed at transforming loops [33, 5], in order to allow concurrent execution of their iterations on several processors. Although these techniques are very polished and have a strong mathematical basis, the degree of parallelism they can offer is very reduced, as they support concurrency only in nested loops, affecting in the least the overall execution time.

For several decades, compilers have tried to implement symbolic analysis [18] for analyzing and capturing essential properties needed for detecting and exploiting parallelism in applications [13]. In this context, *dependancy analysis* [42] has received a great attention: In fact, it defines the necessary constrains on the program components' execution order and provides sufficient information for the exploitation of the available parallelism. Static discovery and management of the dependance structure of programs save a tremendous amount of execution time, and dynamic utilization of dependance information results in a significant performance gain on parallel computers.

DynamoRIO [7] is a runtime code manipulation system that supports code transformations on any part of a program, while it executes. It exports an interface for building dynamic tools for a wide variety of uses: program analysis and understanding, profiling, instrumentation, optimization, translation, etc. Unlike many dynamic tool systems,

DynamoRIO is not limited to insertion of callouts/trampolines and allows arbitrary modifications to application instructions via a powerful IA-32/AMD64 instruction manipulation library. DynamoRIO provides efficient, transparent, and comprehensive manipulation of unmodified applications running on stock operating systems (Windows or Linux) and commodity IA-32 and AMD64 hardware. This framework relies on runtime executable disassembly, which is very good for critical situations (i.e., mutagen code, as the execution time is the only moment in which the actual instruction to be executed are found), but produce a significant performance drop.

DTHREADS [35] is a library the purpose of which is replacing the common `pthread` one, giving the user the possibility to run parallel programs in a deterministic fashion, through the use of barriers and memory partitions. The performance of this library is as good as the `pthread`'s one, and in some circumstances it provides the user with an even greater throughput because of good caching architecture use and avoidance of cache-line false sharing. At the same time, the parallelizing activity is completely left to the programmer, as this library requires him to programmatically define what is the task of a particular thread, and what is the synchronization mechanisms thhe overall program should adopt.

Worker Threads The Worker Thread paradigm tries to exploit the parallelism of a machine by having a pool of threads equals to the number of available processing units. The work is then divided into chunks (or tasks), and each thread is able to complete a work unit whenever asked to do so. In [10], a proposal for outlining code sections, organizing them in chunk queues and exploiting worker threads to perform parallel computations has been proposed. Nevertheless, this solution is not a general-purpose one, as it is focused on Fortran code and performs transformation only on that language's sources.

2.4 Parallel Languages and Specifications

In the attempt to provide programmers with technologies aimed at simply supporting the process of creating parallel programs, in literature several *parallel languages* made their pace. A parallel language is a programming language the constructs of which are specifically designed for expressing interactions and memory accesses peculiar to concurrent code executions. Their purpose is to provide a tool for the high-level description of interactions, communications, and synchronizations between a collection of independent agents or processes. They also provide algebraic laws that allow process descriptions to be manipulated and analyzed, and permit formal reasoning about equivalences between processes (e.g., using bisimulation).

Communicating Sequential Processes (CSP) Presented in [27], it is a formal language which is able to express pattern of interactions in concurrent systems. It is one of the most important member of *process algebrae*, a branch of mathematical theories consisting of approaches to formally model concurrent systems. Since its first proposal, CSP has received a warm welcome from the community and has been successfully applied to several complex tasks in the industry, like specifying and verifying the concurrent aspects of systems such as the T9000 Transputer [4] or a secure e-commerce system [19].

As its name suggests, CSP allows the description of systems in terms of component processes that operate independently, and interact with each other solely through message-passing communication. However, the "Sequential" part of the CSP name is now something of a misnomer, since modern CSP allows component processes to be defined both as sequential processes, and as the parallel composition of more primitive processes. The relationships between different processes, and the way each process communicates with its environment, are described using various process algebraic operators. Using this algebraic approach, quite complex process descriptions can be easily constructed from a few primitive elements.

CSP provides two classes of primitives in its process algebra: i) events, which represent communications or interactions: They are assumed to be indivisible and instantaneous; ii) primitive processes, which represent fundamental behaviors: Examples include STOP (the process that communicates nothing, also called deadlock), and SKIP (which represents successful termination).

An extension of CSP, *Occam* [34], has been developed as the native programming language for transputer microprocessors, but implementations for other platforms are available.

OpenMP [46] OpenMP cannot be correctly categorized neither as a parallel language, nor as a framework. Is is a collection of compiler directives, library routines, and environment variables which together define a specification which provides a model for parallel programming that is portable across shared memory architectures from different vendors and is supported by compilers from numerous vendors.

To exploit OpenMP, the programmer has to insert different annotations within the sources in order to instruct the compiler to generate ad-hoc code and link to a specific library in order to support the parallel execution. Several implementations of the runtime layer exploit the Worker-Thread paradigm in order to take care of the computations, although the exact number of threads which will be spawned at startup must be manually specified by the application user.

Parallel Object-Oriented Languages Languages like Charm++ [31] and CC++ [9] present a form of task parallelism in which method invocation logically result in the creation of a separate process to run the method body. Therefore the parallelism unfolds at runtime, so it is normally the responsibility of the runtime system to control the mapping of processes to processors.

Data Parallel Languages Data parallel languages such as ZPL, NESL, HPF, HPJava, and pC++ are popular as a research languages because of their semantic simplicity: The degree of parallelism is determined by the data structures in the program, and need not be expresses directly by the user [51].

The semantic simplicity of data-parallel languages is attractive, yet these languages are not widely used in practice today, as they are not expressive enough for some of the most irregular parallel algorithms, and they rely on fairly sophisticated compiler and runtime support that takes control away from application programmers.

The purely data-parallel model is fundamentally limited to performing identical operations in parallel, which makes computations like divide-and-conquer parallelism or adaptivity challenging at best. Complex dependence patterns such as those arising in parallel discrete-event simulation or sparse matrix factorizations algorithms are still difficult to express.

The second challenge for data-parallel languages is that the logical level of parallelism in the application is likely many times larger than the physical parallelism available on the machine. This places an enormous burden on compiler and runtime system to handle resources management. On massively parallel SIMD machines of the past, the mapping of data parallelism to processors was straightforward, but on modern machines built from general-purpose microprocessors or vector processors, the compiler and runtime system must map the fine-grained parallilsm onto the coarse-grained machines. This approach is being applied in GPU parallel programmin, using frameworks like CUDA [39].

Extensions to Existing Languages In the attempt to simplify the process of writing parallel code, proposals for extending existing languages have been made. Their main purpose is to prevent the programmers from being forced to learn new programming languages to exploit the full power of multicore architectures. For example, in [1] a specification to extend the C++ language with constructs and keywords aimed at specifying transactional sections of code has been presented. The purpose of such a specification would be to allow compilers to automatically produce supports for software transactional memories within standard C++ code. In [16], an extension to the C++ language is proposed as well, in order to provide programmers with STM and mutual exclusion locks primitives.

2.5 Speculative Processing (SP)

SP can be regarded as a means to improve parallel-program performance with respect to serial fractions. In particular, SP tries to guess which is going to be the outcome of parallel fractions of programs and tries to execute the serial ones concurrently. By using a retry-until-commit approach, SP does some work, the result of which may be incorrect, but if it not, a significant increase in performance is obtained.

This technique has been successfully applied in a number of different fields such as pipelined computing architectures [32, 45] and high performance computing systems and applications [11, 48, 43, 41]. Using SP, parallel parts of the application are not subject to locks, therefore the processing units are fully exploited, and with some probability

the partial results will be committed, with some other probability they will be undone. Overall, the global performance of the application will benefit from this approach.

The work in [6] has targeted non-replicated real-time databases and shows the benefits, in terms of transaction timeliness, by speculatively forking, upon detection of a conflict, a copy of the current transaction that remains idle and serves as a save-point to reduce the rollback cost.

2.6 A final summary

Several approaches to face the inborn limit on the maximum speedup a parallel program can reach have been presented in literature. Nevertheless, none of them appears to be a general-purpose one. In addition, there exists a strong trade-off between the degree of transparency — and therefore the possibility of fully exploiting all the facilities given by programming languages — and the actual efficiency these parallelization techniques are offering to the end users.

The most evident issue with the current proposals is the significant trade-off between transparency and efficiency: The programmer is commonly required to be aware either of the machine the code is going to be run onto, or of the specific part of the application which can be explicitly parallelized. In particular, proposed solutions appear to be efficient only when the programmer is able to correctly identify what are the parallel tasks and which are the data portions which must necesarily be accessed concurrently. As mentioned before, an efficient parallel algorithm usually shares little with sequential algorithms, yet the proposed solutions are thought to be used in scenarios where the programmer is still thinking sequentially. This usually makes most parallel application sub-optimized.

On the other hand, if the programmer leaves the underlying frameworks the job of taking care of parallelization, the efficiency of such approaches suffers from the high burden of performing internal operations which are to support parallel execution even when there is no parallelism at all. This comes from the fact that these framework have no aggressive strategies to fully understand or foresee what is going to be the actual workflow of the application, as they have no semantical interpretation capabilities of the code.

Furthermore, many proposals in literature are not language-independent, as they try to produce optimizations at source-code level. In conclusion, proposals presented so far are not fully transparent, they do not exploit the full potential of multiprocessing, and they are not general-purpose.

3 Methodologies to be Investigated

One of the main goals is to define what is a convenient trade-off between transparency seen by the user and the programming model which he is allowed to use, trying to define a paradigm and to develop a framework which will be able to offer a transarency as higher as possible, yet allowing an efficient mapping on parallel objects onto the available computational resources, and without preventing the user from relying on the complete semantic power of a programming language.

I will face this topic with the help of highly advanced techniques, such as code instrumentation, co-existence of different versions of the same code within the same executable, and self-modifying code. They all need methodological contributes to be achieved, entailing the possibility of exploiting compile & linking time modifications to the application level code, which I intend to provide during my research.

Instrumentation allows to transparently change the user's application code, in order to add functionalities (such as monitoring routines, tracking utilities, ...) in a transparent way to the user itself. The goal I have is to study possible systematic approaches to perform such modifications without affecting the original program's semantics.

Software multiversioning is an effective technique which allows to have portions of code having the same semantics on the one hand, but performing different control functionalities on the other hand. This is extremely useful when taking into account the time-criticality of parallel applications, as this allows the global system to make decisions by switching to operating modes with a cost as low as changing some function pointers. In my research, I intend to assess the actual viability of this technique.

Mutagen code is another viable solution to the problem of self tuning of parallel programs. This technique raises security and consistency problems, since a writable executable code might produce unexpected errors, due to undesired changes in semantics. An objective I have is to deeply study this technique in order to produce methodologies and tools suited at preserving consistency and avoiding security issues.

The main obstacle to cope with is the high freedom given to the user by several programming languages, as data objects could be modified in esotheric ways, making it hard to interpret the actual code meaning. For example, nothing prevents the user to implement applications that modifies themselves. In this scenario, a deep study should be performed to tell what is code and what is data, allowing the execution framework to handle the two objects consistently, even in the case an overlapping between the two is present.

Multiprocessing environments are unable to deliver the desired performance over a wide range of real applications, mainly due to the lack of precision of their dependance information. This calls for an effective compilation scheme capable of understanding the dependence structure of complex application programs. During my research I want to investigate the viability of performing complex analysis of dependance graphs, in order to determine what might be the best parallelization pattern/technique to be applied for the specific software which is under automatic parallelization.

Semantic interpretation will be a key factor in my research: I want to define clearly what is the limit of semantic interpretation at low-level code, exploiting both static and dynamic techniques. This would lead to some possibility/impossibility results, which should clarify the way to those willing to research in the automatic code parallelization field.

Speculativity will be a strongly-exploited methodology, in various forms: i) execution flow prediction (a technique similar to the branch prediction in super-pipelined processing units) will be used to statically analyze the application code, in order to help determining which is the best parallelization pattern to be used; ii) interprocedural/symbolic/semantic analysis will enforce the static part of the speculative analysis; iii) process cloning and execution traces on decision points will be techniques viable for extracting parallelism from applications whenever the static analysis would not lead to a deterministic decision; iv) work units to be exploited by worker threads will be speculatively determined in a dynamic fashion, thus allowing a refinement of the static analysis performed at compile-time; v) log/restore facilities will guarantee a consistent result, whenever the parallelization procedure produces an incorrect intermediate result, due to a wrong foresight.

From my research I expect to develop both innovative methodologies and innovative techniques. This entails developing new formal instruments which, in the future, could be considered as a base for an even more advanced research work. At the same time, I will design and develop a set of real innovative open-source software modules and platforms, targeted at end users who feel the need to parallelize application, although they might not have the advanced skills needed to achieve this goal.

Both those aspects are determinant: On the one hand, the new innovative methodologies and techniques that I will propose during my research will contribute to the advance in the automatic parallelization state of the art, producing an enlargement in the current knowledge base of the topic. On the other hand, the development of high performance software tools with bleeding edge components' implementation will be useful for the masses. In fact, the possibility to use a programming model as complete as possible, together with the full transparency provided at the user, will allow even the least experienced programmer to perform parallelizing activities, be they complex or not, relying on an advanced and performing tool. This will be particularly useful in domains such as medicine, biology, pharmacy, etc., where there is a strong interest in the development of parallel applications (e.g., for simulation or analysis purposes) and yet there is a strong lack of skills to efficiently develop them.

4 Achieved Results and Current Activity

During the first PhD year I have focused on a problem narrower than general-purpose parallelization. In particular, I have been focusing on Parallel Discrete Event Simulation and Optimistic Synchronization. In this scenario, I have been working on an Instrumentation tool targeted at ELF executable format and x86 instruction set, in both its 32- and 64-bits flavours, up to the SSE2 extensions. This tool has been successfully used in an Optimistic Simulation platform to intercept changes in the memory map of simulation objects, in order to allow a fast incremental restore of the simulation state whenever a rollback operation is to be executed [IC1]. Its main capabilities are to perform several types of tranformation on the assembly code which entail the injection of single instructions, code fragments or solid modules and routines which alter the execution of the code as it was intended by the developer, but without changing the semantics of the application and therefore not skewing the final result of the original program.

This allows the user to write applications using standar ANSI-C, and transparently adding code aimed at performing housekeeping operations on memory, which in turn allow an efficient parallelization of the code itself, according

to the optimistic synchronization protocol presented in [29].

Later, the Instrumentation tool has been augmented in order to allow the coexistence of differently-intrumented versions of the same application-level code [IC3, IC4]. This has allowed the optimistic simulation platform to perform an autonomic self-tuning on memory log/restore operations, which — by the exploitation of runtime measures of efficiency — decides which is the best operating mode to be placed in action in order to maximize the overall throughput of the system.

Finally, our optimistic simulation platform, namely the ROme OpTimistic Simulator (ROOT-Sim) [40], has been released along with the first version of the Istrumentation tools. This has been done in the hope that it will become a useful tool for both researchers and developers since its early stages.

My current activity focuses on a further enhancement of the Intrumentation tool: In particular, so far the code parsing layer has been extended in order to categorize assembly instructions into families, depending on their actual behaviour and on the effects they produce on data and/or internal CPU registers. The next step is to extend the code generation layer in order to make it rule-programmable. In this way, the tool will be fully exploitable in different scenarios, having the possibility to exploit an instrumentation phase driven by the actual program/machine needs to obtain a performance as high as possible.

Concurrently, I am studying the viability to develop a programmatic optimistic stack frame analyzer. Such a tool would be a powerful runtime module which, once injected into an application executable, would allow tracking execution patterns, in order to self-tune the speculative execution of concurrent functions.

At the same time, I am studying the viability of designing a new memory allocator, based on the standard `malloc` library, which would force any application to use a partitioned memory manager, even in the case of languages which have a flat memory model, like C or Java. This would allow an enhancement in the data separation, therefore simplifying the task of finding parallel patterns on data, even at runtime.

5 Publications

IC5. Alessandro Pellegrini, Roberto Vitali and Francesco Quaglia
The ROme OpTimistic Simulator: Core Internals and Programming Model
In Proc. of the 4-th International ICST Conference of Simualtion Tools and Techniques (SIMUTOOLS) - Demo Paper - 2011

IC4. Alessandro Pellegrini, Roberto Vitali and Francesco Quaglia
An Evolutionary Algorithm to Optimize Log/Restore Operations within Optimistic Simulation Platforms
In Proc. of the 4-th International ICST Conference of Simualtion Tools and Techniques (SIMUTOOLS) - 2011

IC3. Roberto Vitali, Alessandro Pellegrini and Francesco Quaglia
Autonomic Log/Restore for Advanced Optimistic Simulation Systems
In Proc. of the 18-th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication System (MASCOTS) - 2010

IC2. Roberto Vitali, Alessandro Pellegrini and Francesco Quaglia
Benchmarking Memory Management Capabilities within ROOT-Sim
In Proc. of the 13-th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications (DS-RT) - 2009

IC1. Alessandro Pellegrini, Roberto Vitali and Francesco Quaglia.
Di-DyMeLoR: Logging only Dirty Chunks for Efficient Management of Dynamic Memory Based Optimistic Simulation Objects
In Proc. of the 23-rd ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation (PADS) - 2009

6 Submitted Papers

S1. Roberto Vitali, Alessandro Pellegrini, and Francesco Quaglia
A Load-sharing Architecture for High Performance Optimistic Simulations on Multi-core Machines
In Proc. of the 26th IEEE International Parallel & Distributed Processing Symposium (IPDPS)

References

[1] Ali-Reza Adl-Tabatabai et al. *Draft Specification of Transactional Language Constructs for C++*. Aug. 2009.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. 2nd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986. Chap. 11. ISBN: 0-201-10088-6.

[3] Gene M. Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities". In: *Proceedings of the April 18-20, 1967, spring joint computer conference*. AFIPS '67 (Spring). Atlantic City, New Jersey: ACM, 1967, pp. 483–485.

[4] Geoff Barrett. "Model Checking in Practice - The T9000 Virtual Channel Processor". In: *Proceedings of the First International Symposium of Formal Methods Europe on Industrial-Strength Formal Methods*. FME '93. London, UK: Springer-Verlag, 1993, pp. 129–147. ISBN: 3-540-56662-7.

[5] Cédric Bastoul. "Efficient Code Generation for Automatic Parallelization and Optimization". In: *Parallel and Distributed Computing, International Symposium on* 0 (2003), p. 23.

[6] Azer Bestavros and Spyridon Braoudakis. "Value-cognizant Speculative Concurrency Control". In: *Proceedings of the 21th International Conference on Very Large Data Bases*. VLDB '95. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1995, pp. 122–133. ISBN: 1-55860-379-4.

[7] Derek Bruening, Evelyn Duesterwald, and Saman Amarasinghe. "Design and Implementation of a Dynamic Optimization Framework for Windows". In: *In 4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4*. 2000.

[8] Richard P. Case and Andris Padegs. "Architecture of the IBM system/370". In: *Commun. ACM* 21 (Jan. 1978), pp. 73–96. ISSN: 0001-0782.

[9] K. Mani Chandy and Carl Kesselman. *CC++: A Declarative Concurrent Object Oriented Programming Notation*. 1992.

[10] Jyh-Herng Chow, Leonard E. Lyon, and Vivek Sarkar. "Automatic parallelization for symmetric shared-memory multiprocessors". In: *Proceedings of the 1996 conference of the Centre for Advanced Studies on Collaborative research*. CASCON '96. Toronto, Ontario, Canada: IBM Press, 1996, pp. 5–.

[11] J.D. Collins et al. "Speculative precomputation: long-range prefetching of delinquent loads". In: *Computer Architecture, 2001. Proceedings. 28th Annual International Symposium on*. 2001, pp. 14 –25.

[12] D. Dice, O. Shalev, and Nir Shavit. "Transactional Locking II". In: *Proc. of the 20th International Symposium on Distributed Computing (DISC 2006)*. 2006, pp. 194–208.

[13] D.A. Fisher. *Program analysis for multiprocessing*. Graduate School of Arts and Sciences, University of Pennsylvania, 1967.

[14] Keir Fraser. "Practical lock freedom". Also available as Technical Report UCAM-CL-TR-579. PhD thesis. Cambridge University Computer Laboratory, 2003.

[15] Al Geist et al. "MPI-2: Extending the Message-Passing Interface". In: *Proceedings of the Second International Euro-Par Conference on Parallel Processing - Volume I*. Euro-Par '96. London, UK: Springer-Verlag, 1996, pp. 128–135. ISBN: 3-540-61626-8.

[16] Justin E. Gottschlich et al. "Toward Simplified Parallel Support in C++". In: *Proceedings of the Fourth International Conference on Boost Libraries (BoostCon)*. May 2009.

[17] R. Gupta, E. Mehofer, and Y. Zhang. *The Compiler Design Handbook: Optimizations and Machine Code Generation*. Ed. by Y. N. Srikant and Priti Shankar. CRC Press, 2002. Chap. 4. ISBN: 0-8493-1240-X.

[18] Mohammad R. Haghighat and Constantine D. Polychronopoulos. "Symbolic analysis for parallelizing compilers". In: *ACM Trans. Program. Lang. Syst.* 18 (4 July 1996), pp. 477–518. ISSN: 0164-0925.

[19] Anthony Hall and Roderick Chapman. "Correctness by Construction: Developing a Commercial Secure System". In: *IEEE Softw.* 19 (1 2002), pp. 18–25. ISSN: 0740-7459.

[20] Tim Harris and Keir Fraser. "Language support for lightweight transactions". In: *SIGPLAN Not.* 38 (11 2003), pp. 388–402. ISSN: 0362-1340.

[21] John Hennessy and David Patterson. *Computer Architecture - A Quantitative Approach*. Ed. by D. Penrose. Morgan Kaufmann, 2003.

[22] M. Herlihy, V. Luchangco, and M. Moir. "Obstruction-free synchronization: double-ended queues as an example". In: *Distributed Computing Systems, 2003. Proceedings. 23rd International Conference on*. 2003, pp. 522 –529.

[23] Maurice Herlihy. "Wait-free synchronization". In: *ACM Trans. Program. Lang. Syst.* 13 (1 1991), pp. 124–149. ISSN: 0164-0925.

[24] Maurice Herlihy and J. Eliot B. Moss. "Transactional memory: architectural support for lock-free data structures". In: *Proceedings of the 20th annual international symposium on computer architecture*. ISCA '93. San Diego, California, United States: ACM, 1993, pp. 289–300. ISBN: 0-8186-3810-9.

[25] Maurice Herlihy et al. "Software Transactional Memory for Dynamic-Sized Data Structures". In: *PODC '03: Proc. 22nd ACM Symposium on Principles of Distributed Computing*. 2003, pp. 92–101.

[26] Maurice P. Herlihy and Jeannette M. Wing. "Linearizability: a correctness condition for concurrent objects". In: *ACM Trans. Program. Lang. Syst.* 12 (3 1990), pp. 463–492. ISSN: 0164-0925.

[27] C. A. R. Hoare. "Communicating sequential processes". In: *Commun. ACM* 21 (8 1978), pp. 666–677. ISSN: 0001-0782.

[28] Intel Corporation. *Enhanced Intel SpeedStep Technology for the Intel Pentium M Processor*. White Paper. Available online. 2004.

[29] David R. Jefferson. "Virtual Time". In: *ACM Transactions on Programming Languages and System* 7.3 (July 1985), pp. 404–425.

[30] Eric H. Jensen, Gary W. Hagensen, and Jeffrey M. Broughton. "A New Approach to Exclusive Data Access in Shared Memory Multiprocessors". In: Technical Report UCRL-97663 (1987).

[31] Laxmikant V. Kale and Sanjeev Krishnan. "CHARM++: a portable concurrent object oriented system based on C++". In: *SIGPLAN Not.* 28 (10 1993), pp. 91–108. ISSN: 0362-1340.

[32] V. Krishnan and J. Torrellas. "A chip-multiprocessor architecture with speculative multithreading". In: *Computers, IEEE Transactions on* 48.9 (1999), pp. 866 –880. ISSN: 0018-9340.

[33] Leslie Lamport. "The parallel execution of DO loops". In: *Commun. ACM* 17 (2 1974), pp. 83–93. ISSN: 0001-0782.

[34] SGS-THOMSON Microelectronics Limited. *Occam 2.1 Reference Manual*. 1995.

[35] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. "DTHREADS: Efficient Deterministic Multithread". In: *Proc. of the 23rd Symposium on Operating Systems Principles (SOSP 2011)*. ACM, 2011, pp. 327–336.

[36] David C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001. ISBN: 0201727897.

[37] Virendra J. Marathe and Michael L. Scott. *A Qualitative Survey of Modern Software Transactional Memory Systems*. Tech. rep. 2004.

[38] Gordon E. Moore. "Cramming More Components onto Integrated Circuits". In: *Electronics* 38.8 (Apr. 1965), pp. 114–117. ISSN: 0018-9219.

[39] NVIDIA Corporation. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*. NVIDIA Corporation, 2007.

[40] Francesco Quaglia et al. *ROOT-Sim: The ROme OpTimistic Simulator*. Oct. 2011. URL: http://www.dis.uniroma1.it/~hpdcs/ROOT-Sim/.

[41] T. Ragunathan and P. Krishna Reddy. "Improving the performance of read-only transactions through speculation". In: *Proceedings of the 5th international conference on Databases in networked information systems*. DNIS'07. Aizu-Wakamatsu, Japan: Springer-Verlag, 2007, pp. 203–221. ISBN: 3-540-75511-X, 978-3-540-75511-1.

[42] C. V. Ramamoorthy and M. J. Gonzalez. "A survey of techniques for recognizing parallel processable streams in computer programs". In: *Proceedings of the November 18-20, 1969, fall joint computer conference*. AFIPS '69 (Fall). Las Vegas, Nevada: ACM, 1969, pp. 1–15.

[43] A. Roth and G.S. Sohi. "Speculative data-driven multithreading". In: *High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on*. 2001, pp. 37 –48.

[44] Roland Rühl and Marco Annaratone. "Parallelization of FORTRAN code on distributed-memory parallel processors". In: *SIGARCH Comput. Archit. News* 18 (3b 1990), pp. 342–353. ISSN: 0163-5964.

[45] Peter G. Sassone and D. Scott Wills. "Dynamic Strands: Collapsing Speculative Dependence Chains for Reducing Pipeline Communication". In: *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 37. Portland, Oregon: IEEE Computer Society, 2004, pp. 7–17. ISBN: 0-7695-2126-6.

[46] Sanjiv Shah and Mark Bull. "OpenMP". In: *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. SC '06. Tampa, Florida: ACM, 2006. ISBN: 0-7695-2700-0.

[47] Nir Shavit and Dan Touitou. "Software transactional memory". In: *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*. PODC '95. Ottowa, Ontario, Canada: ACM, 1995, pp. 204–213. ISBN: 0-89791-710-3.

[48] G.S. Sohi and A. Roth. "Speculative multithreaded processors". In: *Computer* 34.4 (2001), pp. 66 –73. ISSN: 0018-9162.

[49] Herb Sutter. "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software". In: *Dr. Dobbs Journal* 30.3 (2005), pp. 202–210.

[50] Michael Joseph Wolfe. *High Performance Compilers for Parallel Computing*. Ed. by Carter Shanklin and Leda Ortega. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0805327304.

[51] K. Yelick et al. "Parallel Languages and Compilers: Perspective From the Titanium Experience". In: *Int. J. High Perform. Comput. Appl.* 21 (3 2007), pp. 266–290. ISSN: 1094-3420.