FACULTAD DE INFORMÁTICA
UNIVERSIDAD POLITÉCNICA DE MADRID

TESIS DE MÁSTER

MÁSTER EUROPEO EN COMPUTACIÓN LÓGICA

FUZZY GRANULARITY CONTROL IN PARALLEL/DISTRIBUTED COMPUTING

AUTOR: Mª TERESA TRIGO DE LA VEGA
TUTOR: PEDRO LÓPEZ GARCÍA

SEPTIEMBRE, 2010

Index

List of Figures

List of Tables

Abstract

Automatic parallelization has become a mainstream research topic for different reasons. For example, multicore architectures, which are now present even in laptops, have awakened an interest in software tools that can exploit the computing power of parallel processors. Distributed and (multi)agent systems also benefit from techniques and tools for deciding in which locations should processes be run to make a better use of the available resources. Any decision on whether to execute some processes in parallel or sequentially must ensure correctness (i.e., the parallel execution obtains the same results as the sequential), but also has to take into account a number of practical overheads, such as those associated with tasks creation, possible migration of tasks to remote processors, the associated communication overheads, etc. Due to these overheads and if the *granularity* of parallel tasks, i.e., the "work available" underneath them, is too small, it may happen that the costs are larger than the benefits in their parallel execution. Thus, the aim of granularity control is to change parallel execution to sequential execution or vice-versa based on some conditions related to grain size and overheads. In this work, we have applied fuzzy logic to automatic granularity control in parallel/distributed computing and proposed fuzzy conditions for deciding whether to execute some given tasks in parallel or sequentially. We have compared our proposed fuzzy conditions with existing sufficient (conservative) conditions. Our experiments showed that the proposed fuzzy conditions result in more efficient executions on average than the conservative conditions. Finally, we have developed a profiler for estimating the granularity (i.e. execution time) of tasks, which is also useful for other applications such as performance verification and debugging.

Keywords: Fuzzy Logic Application, Parallel Computing, Automatic Parallelization, Granularity Control, Scheduling, Complexity Analysis.

This work will be published at the International Conference on Fuzzy Computation (ICFC) in 2010 [112].

Resumen

La paralelización automática se ha convertido en un tema de investigación fundamental por diferentes razones. Entre ellas, las arquitecturas *multicore*, que actualmente se encuentran incluso en ordenadores portátiles, han despertado el interés en herramientas *software* capaces de explotar el poder computacional de los procesadores paralelos. Los sistemas distribuidos y multiagente también obtienen beneficio de las técnicas y herramientas para la decisión de en qué ubicación debe ejecutarse cada proceso con el fin de hacer el mejor uso de los recursos disponibles. Cualquier decisión acerca de si ejecutar algunos procesos en paralelo o secuencialmente debe garantizar correción (es decir, la ejecución paralela obtiene los mismos resultados que la secuencial) pero además debe tener en cuenta un conjunto de *overheads*, como los asociados a la creación de tareas, posible migración de tareas a procesadores remotos, comunicación, etc. Debido a esos overheads si la *granularidad* de las tareas paralelas, es decir, el trabajo que suponen, es muy pequeña, puede ocurrir que los costes sean mayores que los beneficios de la ejecución paralela. Por ello el objetivo del control de granularidad es cambiar la ejecución de secuencial a paralela (o viceversa) basándose en condiciones relacionadas con el tamaño del grano de las tareas y los overheads del sistema. En este trabajo hemos aplicado lógica borrosa al control de granularidad automático en computación paralela/distribuida y hemos propuesto condiciones para decidir si un grupo de tareas debe ser ejecutado secuencialmente o en paralelo. Asimismo, hemos comparado nuestro enfoque borroso propuesto en este trabajo con condiciones suficientes (conservadoras) existentes. Los resultados experimentales demuestran que las nuevas condiciones basadas en lógica borrosa seleccionan ejecuciones más eficientes (en media) que las conservadoras. Finalmente, hemos desarrollado una herramienta de perfilado (*profiler*) para estimar la granularidad (el tiempo de ejecución) de las tareas, que es a su vez de utilidad en otras aplicaciones como verificación de rendimiento y depuración.

Palabras clave: Aplicación de la lógica borrosa, computación paralela, paralelización automática, planificación, análisis de complejidad.

Este trabajo será publicado en la conferencia internacional “International Conference on Fuzzy Computation (ICFC)” de 2010 [112].

Chapter 1

Introduction

Automatic parallelization is nowadays of great interest since highly parallel processors, which were previously only considered in high performance computing, have steadily made their way into mainstream computing. Currently, even standard desktop and laptop machines include multicore chips with up to twelve cores and the tendency is that these figures will consistently grow in the foreseeable future. Thus, there is an opportunity to build much faster and eventually much better software by producing parallel programs or parallelizing existing ones, and to exploit these new multicore architectures. Performing this by hand will inevitably lead to a decrease in productivity. An ideal alternative is automatic parallelization. There are however some important theoretical and practical issues to be addressed in automatic parallelization. Two of them are: (i) preserving correctness (i.e., ensuring that the parallel execution obtains the same results as the sequential one) and (ii) (theoretical) efficiency (i.e., ensuring that the amount of work performed by executing some tasks in parallel is not greater than the one obtained by executing the tasks sequentially, or at least, there is no slowdown). Solutions to these problems have already been proposed, such as [28, 62]. However, these solutions assume an idealized execution environment in which a number of practical overheads such as those associated with task creation, possible migration of tasks to remote processors, the associated communication overheads, etc, are ignored. Due to these overheads and if the *granularity* of parallel tasks, i.e., the "work available" underneath them, is too small, it may happen that the costs of parallel execution are larger than its benefits.

In order to take these practical issues into account, some methods have been proposed whereby the granularity of parallel tasks and their number are controlled. The aim of *granularity control* is to change parallel execution to sequential execution or vice-versa based on some conditions related to grain size and overheads. Granularity control has been studied in the context of traditional [72, 84], functional [65, 66] and logic programming [69, 33, 123, 79].

Taking all these theoretical and practical issues into account, an interesting goal in automatic parallelization is thus to ensure that the parallel execution will not take more time than the sequential one. In general, this condition cannot be determined before executing the task involved, while granularity control should intuitively be carried out ahead of time. Thus, we are forced to use approximations. One clear alternative is to evaluate a (simple) sufficient condition

to ensure that the parallel execution will not take more time than the sequential one. This was the approach developed in [79]. It has the advantage of ensuring that whenever a given group of tasks are executed in parallel, there will be no slowdown with respect to their sequential execution.

However, the sufficient conditions can be very conservative in some situations and lead to some tasks being executed sequentially even when their parallel execution would take less time. Although not producing slowdown, this causes a loss in parallelization opportunities, and thus, no speedup is obtained. An alternative is to give up strictly ensuring the no slowdown condition in all parallel executions and to use some conditions that have a good average case behavior. It is in this point where fuzzy logic can be successfully applied to evaluate "fuzzy" conditions that, although can entail eventual slowdowns in some executions, speedup the whole computation on average (always preserving correctness).

It is remarkable the originality of this approach that is betting for the expressiveness of fuzzy logic to improve the decision making in the field of program optimization and, in particular, in automatic program parallelization, including granularity control.

1.1 Fuzzy Logic Programming

Fuzzy logic has been a very fertile area during the last years. Specially in the theoretical side, but also from the practical point of view, with the development of many fuzzy approaches. The ones developed in logic programming are specially interesting by their simplicity. The fuzzy logic programming systems replace their inference mechanism, SLD-resolution, with a fuzzy variant that is able to handle partial truth. Most of these systems implement the fuzzy resolution introduced by Lee in [74]: the Prolog-Elf system [68], the FRIL Prolog system [14] and the F-Prolog language [76].

One of the most promising fuzzy tools for Prolog was the "Fuzzy Prolog" system [45]. Fuzzy Prolog adds fuzziness to a Prolog compiler using CLP($\mathcal{R}$) instead of implementing a new fuzzy resolution method, as other former fuzzy Prologs do. It represents intervals as constraints over real numbers and *aggregation operators* as operations with these constraints, so it uses the Prolog built-in inference mechanism to handle the concept of partial truth.

RFuzzy

Besides the advantages of Fuzzy Prolog [113, 45], its truth value representation based on constraints is too general, which makes it complex to be interpreted by regular users. That was the reason for implementing a simpler variant that was called RFuzzy [96, 90, 97, 109]. In RFuzzy, the truth value is represented by a simple real number.

RFuzzy is implemented as a Ciao Prolog [63] package because Ciao Prolog offers the possibility of dealing with a higher order compilation through the implementation of Ciao packages.

The compilation process of a RFuzzy program has two pre-compilation steps: (1) the

RFuzzy program is translated into CLP($\mathcal{R}$) constraints by means of the RFuzzy package and (2) the program with constraints is translated into ISO Prolog by using the CLP($\mathcal{R}$) package.

As the motivation of RFuzzy was providing a tool for practical application, it was loaded with many nice features that represent an advantage with respect to previous fuzzy tools to model real problems. That is why we have chosen RFuzzy for the implementation of our prototype in this work.

Chapter 2

State of the Art

2.1 Multicores

2.1.1 Parallel Computing and Multicores. Main Concepts

Parallel computers are classified in two main groups: those that use only one machine (e.g., multicore and multiprocessor computers) and those that use more than one machine (e.g., distributed computers, clusters, massive parallel processing (MPPs) and grids). A multicore processor is the one that contains more than one execution unit in only one chip while a multiprocessor has more than one processor (which, at the same time, can be both single or multicore). In a distributed memory computer the processing elements are connected by a network. A cluster is a group of coupled computers that work together closely, a MPP is a single computer with many networked processors and grid computing makes use of computers communicating over the Internet.

Executing a task in parallel involves splitting it into subtasks, executing these subtasks in different cores and obtaining the outcome of the main task by combining the outcomes of the subtasks. In a multicore system, one of the cores is in charge of assembling the final result [40]. Sometimes, the operating system acts as the scheduler which is in charge of the task assignment. The main problems in parallel execution are the existence of new sources of bugs (like race conditions) due to the execution of more than one operation at the same time (concurrency)[1] and the associated overheads (due to communication and synchronization operations) that limit performance. Cache memories have an important impact in these overheads. They can be shared by all the cores or independent (each core its cache). Shared cache memories are faster but they require a method for controlling concurrent accesses while the independent ones only need a synchronization protocol.

There are several levels of parallel computing. *Bit-level parallelism*, which is based on increasing the word size, reduces the number of instructions that the processor must execute to

[1] In this field concurrent and parallel are not interchangeable. Concurrent refers to the execution of the threads interleaved onto a single hardware resource while in a parallel execution we find more than one thread running simultaneously on different hardware resources [7].

complete an operation. Longer instructions can specify the status of more Arithmetic and Logic Units (ALUs) and thus more operations can be completed in each clock cycle. Very-Large-Scale-Integration (VLSI) chips can take advantage of this type of parallelism. *Instruction level parallelism* consists on re-ordering the instructions, that are going to be executed by the processor, without changing the result of their execution. Some instructions depend on the result of previous operations. Without instruction level parallelism no instruction is executed until these dependencies have been solved. Nevertheless, further instructions that do not depend on previous results could be executed in the waiting time. The performance is improved by putting them first (i.e., re-ordering the instructions flow). Pipelined processors (RISC) increase its performance by re-ordering the instruction flow. Superescalar processors combine pipelining with the ability of issuing more than one instruction at the same time. Instruction parallelism inherent in program loops is called *data parallelism* (loop iterations over different data), i.e., the same instruction with different data is executed at the same time. On the other hand *task-level parallelism* refers to executing in parallel different operations with different data. Multiprocessors (and multicores) are able to exploit this type of parallelism.

Traditionally computers architects have focused their efforts in increasing the performance by using parallelization techniques. Hardware structures exploited bit, instruction and data parallelism. This optimizations were made at hardware level without any software modification. As software has become more complex, applications have become capable of running multiple tasks at the same time. In order to take advantage of this parallelism (at thread-level) both, hardware and software, must be adapted [7].

The evolution of the architectures has been characterized by multiplying by two the number of transistors that can be placed on an integrated circuit, approximately every two years (Moore's Law). Furthermore the clock frequency of these components has been multiplied by two every two years. So every two years we have the double of components working twice faster in the same space. The power consumption and the generated heat are the two main problems that have limited the development of more powerful processors. On one hand, the gate that switches the electricity on and off gets thinner as a transistor gets smaller and the flow of electrons through this element could not be blocked. Thus the energy consumption becomes unboundable. On the other hand, transistors switch faster as the clock frequency is increased and thus also consume more power and also generate more heat [40].

Multicore systems are a good solution to this problem. Having more than one core working at low frequencies we can continue improving the productivity while the heat dissipation and the energy consumption can be controlled.

The evolution steps from single cores to multicores can be found in [7]. They can be summarized as follows: the natural next approach seemed to be multiprocessor systems. Nevertheless the cost of adding more processors was unaffordable. So the initial solution was to use additional logical processors: *symultaneous multithreading* or SMT (Hyper-Threading Technology is Intel's implementation). With this technology, both operating system and applications schedule multiple threads as if there were several physical processors. Then they are executed in

parallel over logical processors (but in the end sequentially over the only physical one). Multicore processors (two or more cores in a single processor, chip multiprocessing or CMP) are the next logical transition. The main difference between SMT and CMP is that in the later, threads are executed in a real parallel way, i.e., at the same time over different harware elements. A multicore is a single processor in which each core is perceived as a logical processor with all the associated resources [117] (while in SMT, resources are shared). From the programmer point of view, multicores (and also multiprocessors) are the same than SMT, i.e., programs use the physical cores as they use the logical processors of the SMT technology.

2.1.2 Automatic Parallelization Tools

CMPs are composed by small processors whose ability of finding instruction-level parallelism is also reduced. Thus, these processors depend on thread-level parallelism [53]. Operating systems are already designed to take advantage of these new architectures [117], nevertheless traditional applications are not enough to take advantage of them, because they have only one execution thread. So it is necessary to develop programs with more than one execution thread or to parallelize the existing ones. Both tasks are complex and prone to errors and increase the complexity inherent to the software development process. Automatic parallelization (see Section 2.2) seems to be a good solution.

2.1.3 Performance

In order to obtain the best performance, resources must be used as much as possible. Nevertheless, in any case for N processors a speedup [2] of N is never going to be achieved, i.e., the parallel execution will never be N times faster than the sequential one.

There is an upper bound on the usefulness of adding more parallel execution units that depends on the portion of the program that can be parallelized. This fact is gathered in Amhdal's law [7]. This bound is set up as $\frac{1}{S}$ where S is the non-parallelizable fraction of the program ($0 \leq S \leq 100$). In a system with n processing units the speedup will be equal to $\frac{1}{S+(1-S)/n}$ or more precisely $\frac{1}{S+(1-S)/n+H(n)}$ where *H(n)* is the overhead associated to the parallel execution itself (Operating System overhead due to threads creation plus the overhead due to communication and synchronization between threads).

Gustafson's law reinforces Amdahl's law by taking into account two facts: that the problem size is not fixed and that the size of the sequential section depends on the number of processors. In this case the speedup is equal to $P - n(P-1)$ where P is the parallelizable fraction of the program and n is the number of processing units.

Previous laws reveal one of the main drawbacks of multicores, which is their lower serial performance. Single-thread applications are only executed in one core and thus they can not take advantage of the multicore architecture. As cores in a multicore architecture are slower

[2] $Speedup = \frac{Sequential\ \ Execution\ \ Time}{Parallel\ \ Execution\ \ Time}$

than the ones in single core machines the execution of a single-thread application will take more in this new architectures.
Up to now we have assumed that every core in a multicore processor provides the same performance but this is true only in symmetric multicore processors. In an asymmetric multicore processor, each core can provide a different performance. We could solve the main disadvantage of multicores having a faster core for executing single-thread applications, while at the same time, with slower cores we can keep the energy consumption and the heat dissipation under control. In practice, without considering asymmetry in the design of the applications its benefits can not be exploited [12]. That means that, in general, without adapting the design of the applications, symmetric multicores are expected to behave better. It has been shown [12] that the more asymmetry the more negative effects. In general the impacts on performance are that: it becomes less predictable (although unpredictability could be eliminated with the operating system kernel and applications asymmetry-aware) and under certain conditions they can increase the performance of the serial fragments of code.

Performance Factors

Parallel execution performance can be affected by the garbage collector (it can interfere in the execution of the program), scheduling, locking, synchronization, cache thrashing (allocation overhead), Operating System (identified as the primary source of instability), work imbalancing among threads buffer spaces (that are warmed up in the first iterations so they must be discarded) and the number of processors in the system [12].
Work imbalancing among threads is an issue that must be taken into account. The scheduler must keep the cores busy as much as possible and with similar work loads. In order to avoid the effects of having big tasks they can be divided into smaller tasks [114].
In any case performance scalability is limited. Lower clock frequencies combined with pipelining result in higher performance. Overheads due to the execution of sequential portions of code can be compensated by executing the workloads long enough [98].

The hierarchy of memory of multicores is one of the elements in which they present the bigger number of particularities. For the best performance the following properties are important [98]: fast cache-to-cache communication, large L2 or shared capacity, fast L2 to core latency and fair cache resource sharing.
In multicores, the higher clock frequency the higher memory demand (on and off-chip) and the higher on-chip cache size the longer average memory access delay.
A shared L2 cache is able to eliminate data replication (so it provides a larger cache capacity) but, as a main drawback, it suffers longer hit latency and competitions of its resources are possible.

Memory latency and bandwidth (among processor, memory, network, file system and disk) can be examined using benchmarks. The cache-to-cache latency (highly relevant in workload performance) and the speedup can be determined using the same procedure. In this last case it is needed to run single thread and multithread benchmarks.

2.2 Automatic Parallelization Tools and Techniques

2.2.1 From Manual to Automatic Parallelization

As said before, traditional software is not enough for taking advantage of parallel architectures because it only has one execution thread. Thus it is necessary to develop programs with more than one execution thread working concurrently (or to parallelize the existing ones). Both tasks are complex and prone to errors and increase the own complexity of the software development. One of the main problems of the programmers is that it is not trivial to understand how parallel programs behave in multicore systems. VisAndOr, a tool for visualizing parallel execution of logic programs was presented in [24]. This tool is useful for users in order to realize about the behavior of the programs in a graphical way. Many automatic parallelization tools also allow to visualize parallel executions [75, 105, 116, 6, 36, 1]. These tools solve some of the main problems so automatic parallelization seems to be a good solution.

2.2.2 Automatic Parallelization Basis

A parallel execution must provide the same results than the sequential one and also reduce (or at least not increase) the amount of work performed, i.e., must be correct and efficient [79]. There are two types of parallelism in logic programming. In And-Parallelism some goals of a given body clause are executed at the same time. In Or-Parallelism, different clauses, i.e., branches of the derivation tree, are explored simultaneously. We are focused on And-Parallelism, in concrete on Independent And-Parallelism (IAP). In this kind of parallelism the goals are executed in parallel when they are strictly independent, i.e., the execution of a goal does not affect to the others. This feature can be determined before the execution (at compile-time). Or-Parallelism is out of the scope of this work.

In automatic parallelization it is necessary to take into account the granularity of the parallel tasks [33, 79] (the work available under them). Overheads due to the work involved in task creation, scheduling, communication, synchronization, etc. appear in parallel execution scenarios. These overheads can cause that parallel execution takes more time than the sequential one (which is known as a *slowdown*). The extra amount of work depends directly on the number of processing units whereas one program execution can only take advantage of the same number of processing units that its number of threads (Amdahl's law). In general terms every program has two parts: the one that is parallelizable and the one that is not. The code of the non-parallelizable part contains dependent calculations. The longest chain of these operations is known as the *critical path*. The execution time of the critical path is a lower bound on the parallel execution time.

Methods for estimating the granularity of a goal at compile time can be found in [33] and in [79]. The distributed random-access model (DRAM) presented in [83] considers costs related to tasks communication and tries to reduce the overhead related to executing in parallel. In this model there is an interprocessor communication (besides the accesses to the RAM memory)

with an associated cost. The main goal is to minimize the execution time instead of maximizing the usage of processing resources.

2.2.3 Automatic Parallelization Techniques

One of the main problems of automatic parallelization is that we need complex program analysis. Program proofs are helpful in order to validate sequential programs and also in order to parallelize them. An algorithm that given a proven program transforms its proof obtaining a proven parallelized and optimized program is presented in [67].
Abstract Interpretation can be also used to validate programs (in general it can be used to compute properties at compile-time). It can be applied in cases of program specialization in which the input values are unknown as, for example, program parallelization. The &-Prolog [101] compiler uses abstract multiple specialization to perform automatic parallelization. CiaoPP, the preprocessor of the Ciao multiparadigm programming system also uses modular, incremental abstract interpretation to perform high-level program transformations (including automatic parallelization) [60]. Ciao is the successor of &-Prolog [57].

It is usual to translate the program into an intermediate representation as, for example, Directed Acyclic Graph (DAG), in order to detect parallelism [118, 78, 121, 105, 6, 119, 108, 60]. First the program is translated into its DAG and then program transformations are performed over it. The method presented in [107] selects the scheduling algorithm that best assign the DAG to the target parallel machine. It is based on five decision levels taking into account the characteristics of the DAG of a C program. The levels are: communication cost, execution time, level to task ratio, granularity and number of processors. Each level suggests a subset of algorithms according to the characteristics of the DAG. Then the intersection of these subsets is evaluated and the optimal scheduling algorithm is selected.

Other approaches infer parallelism by detecting pieces of programs that access disjoint parts of the heap, i.e., that are independent [41, 48, 55].

2.2.4 Automatic Parallelization of Logic Programs

The parallelism present in the execution of logic programs can be of two classes: explicit (message passing, threads, ...) or implicit (Or and And-parallelism) [31]. In both cases it can be exploited in a simple way [46]. In fact, work in parallel logic programming (LP) began at the same time than the work in LP [47]. Its high level nature, the presence of non-determinism, its referential transparency and other features allow to obtain speedup by executing in parallel. Automatic parallelization can be performed without any user intervention. Formal methods can be relatively easily used to prove correctness and efficiency of the performed transformation (due to the semantics of logic languages). Logic languages have been extended with explicit constructs for concurrency (or by modifying the semantics of the language) in oder to allow manual parallelization. This extensions complements the automatic parallelization process.
Due to its properties, automatic parallelization of logic programs can be considered to paral-

lelize automatically programs written in other languages by translating them into logic programs. In Parafrase-2 [99] the source program is translated into an intermediate (logic) representation which is parallelized instead of the source program. CiaoPP [64] techniques [94, 93, 92] can analyze several languages via an Intermediate Representation (IR) based on blocks that are easier to manipulate. Each block is similar to a Horn clause. In the same way this clausular representation could be scheduled in a set of parallel and sequential tasks. So translating any source language into this clausular form we would be able to analyze any source language in a simple way. The problem is reduced to the translation of the source language into the clausular form.
The &-Prolog system is described in [59]. It is a practical implementation of a parallel execution model for Prolog. It exploits strict and non-strict parallelism and supports both, manual and automatic parallelization. It takes advantage of the generalized version of Independent And-Parallelism (IAP) presented in [61]. A full description of the framework for the automatic parallelization of logic programs for restricted, goal-level IAP can be found in [91]. &-ACE [100, 47] is another system based on the logic programming paradigm (SICStus Prolog) than exploits all sources of parallelism.

2.2.5 Declarative Aspects of Multicore Programming

Declarative (functional, logic, constraint-based, etc.) languages specify what the program does without entering in details of how it is done. Due to this main feature they allow to write simpler parallel programs and their parallelism can be exploited in a easier way (with respect to imperative ones). Since the mid Noughties (2006) most of the results are presented in the Declarative Aspects of Multicore Programming (DAMP).

Some parallel programming languages have been presented. Parallel Haskell [80, 52] (pH) is implicitly parallel and combines the declarative execution model with the (Eager) Haskell syntax and types. Partially computed data are held in the heap. Its main problem is the cost associated with non-strictness. More efforts on implementing nested data parallelism have obtained excellent speedups [25]. Despite of the efforts, no high performance benefits were obtained in all the cases. In order to improve speedups one option is to change the way in which users compose parallel programs by inserting higher-order algorithmic schemas in the program [8]. When Haskell synchronization primitives are used it is necessary to take into account that not all of them have the same efficiency [110].
NESL [16] was developed in the early Nineties and revisited in DAMP'06. It allows to describe parallel algorithms that can be analyzed at runtime. One of the main targets of our work is to do as much analysis tasks as possible at compile time. Despite of the drawback of not being analyzable at compile time it has the following advantages: it can be programmed, analyzed and debugged in a simple way. It has a cost semantics that allow to estimate the amount of work that a program performs without taking into account implementation details as the number of processors. It is true that a model that takes into account details of the system is not going to

be portable but it is going to be more precise. CiaoPP [64] solves this problem in the following way: when a technique depends on a system detail it is obtained during the installation process. As C is a wide used language in the industry, it is analyzed in the new context in which parallel computing is emerging as the most natural way of computation [35]. Declarative languages seems to be more promising (faster on multicores and parallelism expressed easily) but much important code is written in imperative languages as C. The best solution is to perform the easiest software translation to a declarative language. Jekyll is a functional programming language that can be easily translated to/from C. This solution allows to maintain some code in C.
The Hume Programming Language [52] is for concurrent asynchronous multithreading safety-critical systems. It presents a high level of parallelism and minimizes communication and synchronization operations.

Current and future LP programs and systems have been revisited [47, 31, 57]. In general, parallel LP systems exploit parallelism from symbolic applications by keeping the control in LP. Advances in LP have been made separately and they would be combined. At the same time implementation technologies should be simplified in order to simplify further developments. Taking into account the overheads by performing granularity control (using the results of cost analysis) [57, 58] seems to be a very promising approach because of its no slowdown guarantee. Prolog has become more popular since the late Noughties. That is why some Prolog systems have been adapted in order to support principles on working in Parallel Logic Programming [32]. The parallelism of other types of applications (numerical, etc.) can be also exploited.

Solutions to the problems introduced by the new execution model itself have also being presented. They must be solved in a safe way [56]. New primitives have been presented in order to guarantee mutual exclusion in critical sections avoiding the use of locks [43]. With this solution the control continue being declarative and deadlocks are avoided. Of course, the sequential meaning is preserved when new primitives are used. Nested transactions are the ones with worst cost [43, 81, 25].

Automatic parallelization [58] requires automatic detection of independent tasks which, at the same time, requests a definition of independence. In general the term independence is the condition that guarantee correctness and efficiency. Granularity control optimizations (like simplifying comparisons of cost equations and stopping granularity control) have been shown very promising. Using events we can obtain flexible parallel executions but in a more difficult way than if we use threads [104]. In order to have the best of each world the two approaches have been combined by abstracting both elements [122]. The different parts of the programs are written using the more suitable mechanisms and then both are unified.
Threads and events are not the only mechanisms that can be used. In fact the use of threads is prone to errors and bugs. An alternative could be an approach based on using sieves [77]. It has a simpler semantics, so reasoning about that programs is easier. Another advantage is its scalability, suitable for a bigger number of processors.
Transactions are an alternative that makes concurrent programming simpler, improves scalabil-

ity and increases performance. If there is a situation in which the transaction can not continue in a correct way then its effects are revoked and the transaction is aborted and, when possible, it will be re-executed. Nevertheless this can cause a slowdown for long-living transactions. Memorization allow re-executing when is possible (when a procedure is retried with the same argument then it is not evaluated again). So memorization guarantees that communication actions performed in the aborted execution can be satisfied when the transactions is retried [124]. In general the synchronization of parallel computations must be guaranteed. Join patterns were a way for threads and asynchronous distributed computations. They have been improved [49] in order to provide synchronization in more complex contexts (message-passing, token-passing, etc.).

One of the main problems of performing static analysis at compile-time is that input values are unknown. Although performing as much work as possible at compile time reduces the run-time overhead, sometimes it is better to postpone the decision until execution time. This is the case of some self-adjusting-computation techniques [51]. Using these techniques the parallel execution is updated depending on the program data. A change propagation algorithm is used. More semantics [9] have been developed for other programming languages in order to allow formal verification.

As we have pointed out before, concurrent operations require the definition of some memory management rules in order to guarantee correct results. Declarative languages usually deal with simple big structures while functional or object-oriented languages tend to work with small complex elements. New heap managers need to be developed. One that is based on reference counting with impact quantification has been already presented [42].

Speculative execution is also a technique for detecting parallelism in sequential programs. It is well understood when the computation is relatively simple (read and write in known locations) but it becomes more complicated in cases with multithreads that communicate and synchronize a lot. Using (n-way) barriers [125] expressions are guarded (until barriers are satisfied) and speculative execution becomes easier to understand.

More approaches started to appear in the late Noughties for object-oriented languages like Java. A novel approach that uses traces as units of parallelization has been presented [20]. This work shows that the approach is viable and increases the performance (by comparing with parallelizations performed by hand).

2.2.6 Unsolved Problems

Despite of the work that has been developed during the last decades in automatic parallelization, problems with nested loops (frequent in scientific and engineering applications) are still unsolved. Some efforts towards an effective automatic parallelization system use a polyhedral model for data dependencies and program transformation have been presented [17]. This model provides a powerful abstraction on such nested loops and seems to be a solution to the problem of *gcc* and many vendor compilers. Nested data parallelism have been traditionally

implemented by vectorizing program transformations although this approach modifies the data representation. Traditional compilers (like NESL [16]) vectorized the whole program including parts that rely on non-vectorizable features. Subsequent contributions [26] only vectorizes suitable program parts and then vectorized and non-vectorized modules are integrated.

In scientific and engineering applications stencil computations are also usual, but automatic parallelization must deal also with irregular programs. Compilers that are able to transform sequential code from stencil applications into optimized parallel code have been developed previously. Nevertheless, loop parallelization without special techniques originates load imbalancing. The approach presented in [71] is for automatic parallelization of stencil codes that explicitly addresses the issue of load-balanced execution of tiles. Its effectiveness has been shown. On the other hand irregular programs [73] originate different kinds of computations that must be executed in parallel with complex dependencies between iterations. The system Galois [73] (described in more detail below) deals successfully with this problems.

Work in [29] is for automatic parallelization on symmetric shared-memory multiprocessors (SMPs). It supports unbalanced processor loads and nesting of parallel sections. Due to its features the speedup is a real multiplicative over highly optimized uniprocessor execution times. The presented solution is for FORTRAN 90 and High performance FORTRAN. The compiler has been obtained by modifying the back end of the serial compiler obtaining, in the end, a platform-independent solution. The conceptual foundations of other automatic scheme for Fortran programs are detailed in [10]. The main goal is the detection of DO loops suitable to execute in parallel because iterations (and execution times) are similar.

2.2.7 Current Research Topics

In the mid Noughties some of the current research topics were presented in a seminar [11]. The main launched ideas were: (a) to compile one function at the time as *gcc* does. This will allow to exploit task level parallelism available in a single thread with independent tasks due to the possibility of executing hundreds of executions per cycle in machines with unbounded resources. (b) Convergent scheduling. A new framework that simplifies and facilitates the application of constraints and scheduling heuristics. It also produces better schedules. (c) Including tasks granularity in tasks graphs. This can be achieved using a Graph Rewrite System (GRS). It merges tasks into larger ones and ensures that the parallel version of the task graph is not increasing by reducing cost of the communication. (d) Dynamic scheduling for load balancing. Instead of traditionally information about processor workloads obtained via profiling. This approach uses new scheduling strategies that take into account system irregularities that can be predictable. (e) Scheduling in spiral as it is done in digital signal processing (DSP). Spiral takes high-performance implementations for the domain of DSP, transforms and translates them into a search problem for the best implementation of a given transformation on a given computer system. (f) Load balancing strategies for irregular parallel divide and conquer computations as quicksort: re-pivoting, serialization of subproblems, ... (g) Lookahead scheduling for re-

configurable data-parallel applications. (h) Generic software pipelining at the assembly level. Useful in cases of constraints on time and space like in embedded systems. (i) Telescoping languages build compile-time support for user-defined abstractions and for their optimization. (j) Scheduling for heterogeneous systems and grid environments. Extends previous work (for homogeneous systems) so that the same program can be also executed in heterogeneous systems and grid environment (dynamic changing execution environment). It is also needed to take into account the communication networks and the dynamism of the platforms. (k) Data re-distribution selected for multiprocessor tasks (M-tasks) for the scheduling. It plays an important role that must be taken into account. (l) Scheduling hierarchical malleable task graphs, i.e. tasks that can be executed on several processors.

In the last 10 years most of the work in parallel job scheduling has been done in systems with large homogeneous architectures and workloads dominated by both computation and communication-intensive applications. Some of the novel innovations presented on the Job Scheduling Strategies for Parallel Processing (JSSPP) [38] are described below. (a) Scheduling for a mixed workload. The workload has become highly variable and more complex. First of all it is needed to understand the workload in order to satisfy the new conflicting scheduling requirements (for example processes competing over resources suffer from degraded performance when co-scheduled or, on the contrary, collaborating processes suffer it when not co-scheduled). For dealing with mixed workload it is also needed to deal with priorities. (b) Power consumption is lower in multicores than in single core processors and this also limits the performance. Tradeoffs between performance and power consumption must be addressed. This fact will become more important when the number of cores becomes bigger and the granularity of the tasks becomes finer. (c) Asymmetric cores heterogeneity. In some cases having more than one core with the same features is not enough in order to solve the problem. Automatic parallelizers must be take this fact into account. In general every change in the system architecture introduces parallelization constraints. (d) Grids are a better option to execute workloads with little parallelism. This explains in part why they are becoming more popular. Nevertheless job scheduling in grids is more difficult than in single parallel machines because it is needed to perform both the machine allocation and the scheduling itself. When the execution is performed over a grid its properties (heterogeneity, service levels agreements, security, ...) determine the execution. (e) Virtualization or running multiple operating system environments in one or more nodes at the same time. The scheduling must depend on the operating system in which it is going to be executed. Research in this field has started recently.

In summary, the considerations that have bigger influence on parallel scheduling are: workload, heterogeneity, scheduling for power, security, economy and metrics.

2.2.8 Automatic Parallelization Tools

Some of the automatic parallelization tools [30] that have appeared since the early Nineties are described below.

Hypertool [118] (1990) receives a partitioned C program and returns the partitions allocated to the available processors (with the needed synchronization primitives) besides their corresponding explanations. Its algorithm is based on performing optimizations and on the critical path method. If the user is not satisfied with the result the partition strategy can be redefined (using the information and the explanations provided by the tool) in order to improve the results.
Parafrase-2 [99, 3] also appeared in 1990. It takes a sequential program (written in C or FORTRAN, for example) and returns a set of tasks and code for scheduling them. It also shows compiler information to the user (through its GUI). The input language is transformed into an intermediate representation. The compiler performs symbolic dependence and interprocedural analysis followed by auto-scheduling at coarse-grain level. A code generator is used for each language desired as output.
OREGAMI [78] (1991) uses a parallel program in OCCAM, C*, Dino, Par, C or FORTRAN as input. Instead of generating target code this tool generates some mapping directives. Its purpose is to map parallel computations to message-passing parallel architectures by exploiting regularity. The tool allows the user to guide and to evaluate the mapping decisions.
PYRROS [121, 4] was presented in 1992. It takes a task graph and its associated sequential C code and performs a static scheduling plus a parallel C code for message-passing architectures (with synchronization primitives).
Parallax [75] (1993) also needs a task graph as input (apart from an estimation of -or real- task execution times and the target machine as an interconnection topology graph). It returns Gantt chart schedules, speedup graphs and processor and communication use/efficiency charts. It also displays an animation of the simulated running program.
Starting the mid Nineties (1994) three tools were presented: PARSA [105], DF [39] and CompSys HPF & FM [37]. The first one receives a sequential SISAL program and displays the expected runtime behavior of the scheduled program at compile-time. The input program is translated into a DAG with execution delays for the target system specification. The second one makes use of the *Filaments package* which is a library of C code that runs on several distributed-memory machines. It takes a sequential C code and gives the resulting application code written in C plus Filaments calls for distributed shared memory systems to the user. The last one works with High Performance FORTRAN and FORTRAN M procedures obtaining synchronous SPMD code, structured as alternating phases of local computation and global communications. HPF and FM procedures are clearly distinguished and are compiled with the HPF or FM compiler respectively. The communication is detected and generated using pattern matching. Processes do not need to synchronize during local computations.
Paradigm [15, 2] (1995) also works with FORTRAN. In concrete, it takes a sequential F77/HPF program and constructs an explicit message-passing version. The input program is transformed into an intermediate representation and the data partitioning is made automatically. This tool exploits both data and functional parallelism while overlaps computation and communication. It contains a generic library interface.
ProcSimity [116] is also from 1995 but it works with independent user job streams and returns

trace files, the selected algorithm and system (and job) performance metrics. It supports both stochastic independent user job streams and communication patterns for the actual parallel applications, apart from selected allocation and scheduling algorithms on a set of architectures.
In 1996 appeared RIPS [106] and MARS [27]. RIPS takes a set of tasks and returns a scheduling of them. It schedules incoming tasks incrementally and combines the advantages of static and dynamic scheduling. MARS receives a sequential F77 code and returns a machine specific parallel FORTRAN code. It uses loop generation techniques (including Parameter Integer Programming -PIP-). The input program is translated into an annotated syntax tree. The system operates on the linear algebraic representation of the program.
The next tools date back of 1997. Then the activity on developing automatic parallelization tools slowed down until the year 2000. PROMIS [21] works with both C and FORTRAN. The output consists in code for a simulated shared-memory multiprocessor wherein each processor is a pipelined VLIW. This multisource and multitarget parallelizing compiler allows loop level and instruction level parallelization techniques. CASCH [6] takes a sequential program that manipulates, obtaining parallel code generated by including appropriate library procedures from a standard package. Users can compile the resulted parallel program to generate native parallel machine code on the target machine for testing. The input program is transformed into a task graph. The algorithm selects the best scheduling (of the generated ones) in which the communication primitives are inserted automatically. In order to improve the results users can repeat the whole process.
The activity on developing automatic parallelization tools reemerges in the year 2000 with tools like Hypertool/2 and SADS&DBSADS. Hypertool/2 [119] deals with graphs. It takes a CDAG generated at compile-time, expanded incrementally into a DAG and returns its parallel scheduling (in a way that can be incrementally executed at runtime). The execution model is incremental and it uses the HPMCP (Horizontal Parallel Modified Critical-Path) algorithm in which each processor applies the MCP algorithm to its partition. SADS&DBSADS [50] transforms a task tree into a set of scheduled tasks. The process comprises load balancing and memory locality.
ParAgent [70] (2002) requires a F77 serial program that transforms into a parallel F77 program with primitives for message passing that are portable to distributed memory platforms. It has an interactive GUI to help the user. The same interface can be used to see the synchronization/exchange points and the communication patterns of the parallel program.
The next year, 2003, is the one of PETSc, SUIF and Cronus/1. PETSc [13] solves a set of partial differential equations (PDEs) providing its numerical solution. It consists on a variety of libraries. Each of them manipulates a particular family of objects and the operations that can be performed on them. SUIF [44] receives a main program (called a *driver*) and applies a series of transformations on it. Eventually writes out the information. Cronus/1 [111] takes a serial program and the number of processors as input. It returns the resulted parallel code (for the given number of processors) written in C containing run time routines for *SDS* and MPI primitives for data communication.
ASKALON [36, 1] (2005) is a tool set for supporting the development of parallel and distributed

(grid) applications. It is composed by 4 tools, each one composed by remote grid services that are shared. It allows visualization of performance and output data diagrams both online and post-mortem. It supports mostly FORTRAN90 based distributed and parallel programs. In the future, ASKALON tools will also support Java programs with the novel developed Java-based programming paradigm (for performance-oriented parallel and distributed computing).
AspectJ [5] is an extension to the Java programming language that enables clean modularization of aspects that are difficult to implement in a modular way such as error checking and handling, synchronization and performance optimizations. A loop join point model allows AspectJ to intervene directly in loops (direct parallelization of loops without refactoring the code) and thus parallelization was presented in 2006 [54]. The model is based on a control-flow analysis at the bytecode level in order to avoid the ambiguities at the source level. The extension which provides AspectJ with a loop join point is called LoopsAJ.
PLuTo [19]: A Practical and Fully Automatic Polyhedral Program Optimization system was presented in 2007. It performs source-to-source transformations in order to optimize programs by obtaining parallelism and locality at the same time. Thanks to a polyhedral model it deals properly with nested loops. It is done by performing abstraction on them [17]. The transformed code is then reordered in order to improve cache locality and/or parallelized loops. This system improves the result by using realistic cost functions and by combining parallelism and locality. It accepts C, Fortran and any high-level language whose polyhedral domains can be obtained. It also generates OpenMP from C code. More tool information was presented in 2008 [18].
An extensible implementation was presented in MLton in 2008 [108]. It works with a DAG of the sequential program. It deals with one graph for all the possible parallel executions in which the nodes are the units of work and the edges are sequential dependencies. The scheduling (which is a traversal) of a program is determined by a system policy. The system includes three scheduling polices. A cost semantics allows the users to understand the impact of the different schedule policies without taking into account other implementation details.
Galois [73] (2008) is a system that supports parallel execution of irregular (management of pointer-based data structures like trees and graphs instead of arrays and matrices) applications. Its main features are an iterators set for expressing worklist-based data parallelism and a run-time system that performs optimistic parallelization of these iterators. The policy is to assign an iteration to a core when it needs work to do (although it is not optimal in all the cases). It works with *client code* (code with well-understood sequential semantics) in which data parallelism is implicit and returns a set of iterations assigned to a set of processors. It has mechanisms for detecting and solving conflicts (concurrent accesses to a given object for more than one iteration). The assignment is performed balancing the load of each processor.

We have seen how, in general, different tools use the same approach like, for example, management of DAGs [107, 105, 119, 108] or polyhedral model [17, 18]. Furthermore each approach uses more or less the same mechanisms. In real terms this means that the same algorithms are particularly implemented by each tool that makes use of them. The problem is that each tool has its own intermediate representation and the algorithms are designed to be

applied to them. As a result developers are re-implementing algorithms every time that they construct a new optimizing compiler. A new method [34] suggests to separate the algorithms from the intermediate representation in order to be able to reuse their implementation for any platform. This new component technology seems to be very promising (even more due to the popularity that grids are reaching).

In general terms the programming languages that are supported by automatic parallelization tools are C [118, 99, 78, 121, 39, 21, 19], FORTRAN [99, 78, 37, 15, 27, 21, 70, 36, 19] and Java [36] [3] [5].

[3] In the future.

Chapter 3

The Granularity Control Problem

We start by discussing the basic issues to be addressed in our approach to granularity control, in terms of the generic execution model described in [79]. In particular, we discuss how conditions for deciding between parallel and sequential execution can be devised. We consider a generic execution model: let $g = g_1, \ldots, g_n$ be a task such that subtasks $g_1, \ldots, g_n$ are candidates for parallel execution. T_s represents the cost (execution time) of the sequential execution of g and T_i represents the (sequential) cost of the execution of subtask g_i.

There can be many different ways to execute g in parallel, involving different choices of scheduling, load balancing, etc., each having its own cost (execution time). To simplify the discussion, we will assume that T_p represents in some way all of the possible costs. More concretely, $T_p \leq T_s$ should be understood as "T_s is greater or equal than any possible value for T_p."

In a first approximation, we assume that the points of parallelization of g are fixed. We also assume, for simplicity, and without loss of generality, that no tests – such as, perhaps, "independence" tests [28, 62] – other than those related to granularity control are necessary. Thus, the purpose of granularity control will be to determine, based on some conditions, whether the g_i's are going to be executed in parallel or sequentially. In doing this, the objective is to improve the ratio between the parallel and sequential execution times.

Performing an accurate granularity control at compile-time is difficult since most of the information needed, as for example, input data size, is only known at run-time. An useful strategy can be to do as much work as possible at compile-time and postpone some final decisions to run-time. This can be achieved by generating at compile-time cost functions which estimate task costs as a function of input data sizes, which are then evaluated at run-time when such sizes are known. Then, after comparing costs of parallel and sequential executions, it can be determined which of these types of executions must be performed. This scheme was proposed by [33] in the context of logic programs and by [103] in the context of functional programs. An interesting goal is to ensure that $T_p \leq T_s$. In general, this condition cannot be determined before executing g, while granularity control should intuitively be carried out ahead of time. Thus, we are forced to use approximations. The way in which these approximations can be performed, is the subject of the two following sections.

Chapter 4

The Conservative (Safe) Approach

The approach proposed in [79] consists on using safe approximations, i.e., evaluating a (simple) sufficient condition to ensure that the parallel execution will not take more time than the sequential one. Ensuring $T_p \leq T_s$ corresponds to the case where the action taken when the condition holds is to run in parallel, i.e., to a philosophy were tasks are executed sequentially unless parallel execution can be shown to be faster. We call this "parallelizing a sequential program." The converse approach, "sequentializing a parallel program," corresponds to the case where the objective is to detect whether the sufficient condition $T_s \leq T_p$ holds.

Parallelizing a Sequential Program In order to derive a sufficient condition for the inequality $T_p \leq T_s$, we obtain upper bounds for its left-hand-side and lower bounds for its right-hand-side, i.e., a sufficient condition for $T_p \leq T_s$ is $T_p^u \leq T_s^l$, where T_p^u denotes an upper bound on T_p and T_s^l a lower bound on T_s. We will use the superscripts l and u to denote lower and upper bounds respectively throughout the paper. The discussion about how these upper and lower bounds on the sequential and parallel execution times can be estimated are outside the scope of this paper. We refer the reader to [86] and [79] for a full description of compile-time analysis that obtain lower and upper bounds on sequential and parallel execution times respectively as functions of input data sizes.

Sequentializing a Parallel Program Assume now that we want to detect when $T_s \leq T_p$ holds, because we have a parallel program and want to profit from performing some sequentializations. In this case, a sufficient condition for $T_s \leq T_p$ is $T_s^u \leq T_p^l$.

Chapter 5

The Fuzzy Approach

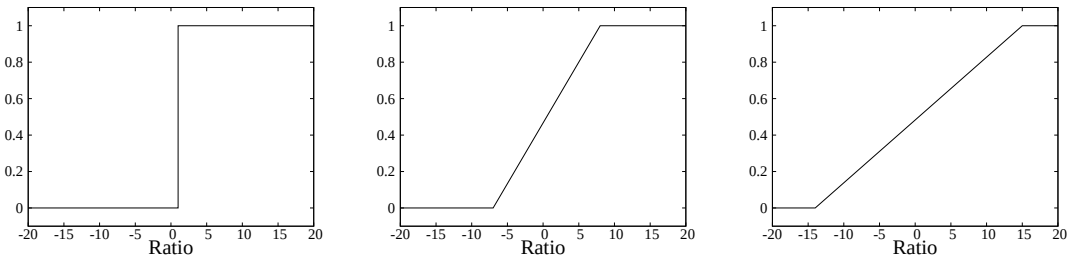
In some scenarios, it is not allowed to perform parallelizations if it does not ensure any speedup. However, in most environments it is justified to sacrifice efficiency in some cases in order to improve the speedup on average or in the majority of the cases. Thus our approach is to give up strictly ensuring that $T_p \leq T_s$ holds and to use some relaxed heuristics using fuzzy logic able to detect favorable cases.

We use as a decision criteria the formula $T_p \leq T_s$. It is easy to transform the formula in $1 \leq T_s/T_p$ or the equivalent $T_s/T_p \geq 1$. We are implicitly using a crisp criteria in the sense that we use an operator whose truth values are defined mathematically.

If we move to classical logic and want to represent the condition of parallelizing or not a set of subtasks using a logic predicate, we could define *greater/2* as a predicate of two arguments that is successful if the first one is greater than the second one and false otherwise. We could check the condition *greater(T_s/T_p,1)* or rename this condition to a logic predicate, *greater1/1*, of arity 1 that compares its argument with 1, succeeds if it is greater than 1 and fails otherwise (i.e., *greater1(1.8)* succeeds, whereas *greater1(0.8)* fails). With the boolean condition represented by the predicate *greater1/1* it is easy to follow the conservative approach presented in Chapter 4.

For a gentle intuition to fuzzy logic, we continue talking about this predicate. We can see that the concept of being "greater than" is very strict in the sense that some cases in which the value is close to 1 are going to be rejected. Let us introduce the concept of truth value. Till now we have been using two truth values *true* and *false*, or 1 and 0. But if we introduce levels of truth we could for example provide for a logic predicate intermediate truth values in between 0 and 1. We have defined other predicates similar to *greater1/1* that are more flexible in their semantics. They are *quite_greater/1* and *rather_greater/1*. Their definition is clearer in Figure 5.1 (and described in Section 7.1). With this set of predicates we are going to define a fuzzy framework for the experimental possibilities of using a fuzzy criteria to take decisions about parallelization of tasks.

Figure 5.1: Fuzzy sets for greater.



Greater Quite greater Rather greater

5.1 Decision Making

Instead of deciding about the goodness of the parallelization depending on a crisp condition as in the conservative approach, in this paper we are going to make the decision attending to a couple of certainty factors: *SEQ*, the certainty factor that is going to represent the preference (its truth value) for executing the sequential variant of a program, and *PAR*, the certainty factor that is going to represent the preference (its truth value) for executing the parallel variant of such program. Both certainty factors are real numbers, $SEQ,\ PAR \in [0,1]$. The way of assigning a value to each certainty factor is not unique. We can define different fuzzy heuristics for their calculation. In Section 7.2 we are going to compare a set of them to choose (in Section 7.3) our selected model.

Once the values of *SEQ* and *PAR* have been already assigned, if $PAR > SEQ$ then our task scheduling prototype executes the parallel variant of the program, otherwise it executes the sequential one.

Chapter 6

Estimating Execution Times

We have seen (Chapter 3) that we need to know execution times for performing granularity control. Since parallel execution times can be derived from the sequential ones [79] (see Chapter 7) in this section we are focused on obtaining sequential execution times at compile and run time (Sections 6.2 and 6.3).

6.1 The problem

Our problem is to estimate safe (upper and lower) bounds on execution time. Note that this problem is a particularization of the one of estimating bounds for any resource consumption. Safe bounds notion means that if R is an amount of resource consumption and R^l (respectively R^u) its lower (resp. upper) bound, then: $R^l \leq R \leq R^u$.

6.2 Execution Time Estimation

Static analysis techniques [87, 88, 95] have been traditionally used for obtaining (upper and lower) bounds on resource usage. *CiaoPP* [22], the advanced program development framework in which our work has been developed, is able to obtain bounds on the usage that a program makes of different resources. As far as we know this analysis is also the most precise one because it is *data sensitive* i.e., bounds are functions or inputs data sizes.

An example of sequential execution time estimation [86] at compile time using CiaoPP is shown in Figure 6.1. In this case, for brevity, we only present the upper bound (*ub*) on the execution time. Lower bounds are written in the same format but replacing *ub* by *lb*. The assertion

```
+ cost(ub,exectime,13101.358*(exp(2,int(N)-1)*int(N))
          -7828.215000000001*exp(2,int(N)-1)+3864.789*exp(2,int(N))-7729.578).
```

must be read as follows: an upper bound on the sequential execution time of *hanoi(N,A,B,C,_)* (where N is the number of disks and A, B and C are the rods) is equal to $13101.358 * N * 2^{N-1} -$

Figure 6.1: Code and CiaoPP executions time estimation for hanoi.

```
hanoi(1,A,_1,C,[mv(A,C)]) :- !.
hanoi(N,A,B,C,M) :-
        N1 is N-1,
        hanoi(N1,A,C,B,M1),
        hanoi(N1,B,A,C,M2),
        concat(M1,[mv(A,C)],T),
        concat(T,M2,M) .

:- true pred hanoi(N,A,B,C,M)
         : ( num(N), elem(A), elem(B), elem(C), var(M) )
        => ( num(N), elem(A), elem(B), elem(C), rt11(M),
             size(ub,N,int(N)), size(ub,A,0), size(ub,B,0),
             size(ub,C,0), size(ub,M,exp(2,int(N))-1.0) )
        + cost(ub,exectime,13101.358*(exp(2,int(N)-1)*int(N))
          -7828.215000000001*exp(2,int(N)-1)+3864.789*exp(2,int(N))-7729.578).
```

Table 6.1: Qsort sequential execution time equations estimated.

Benchmark	Data Size	Approx.	Cost Function
Qsort(L,_)	x=length(L)	T_s^l	$0.013 + 5.46e-5 * x + 0.0034 * x * log(x)$
		T_s^u	$0.24 - 0.016 * x + 0.0014 * x^2$

$7828.215 * 2^{N-1} + 3864.789 * 2^N - 7729.578$. For more information on the assertion language in which the output of the analysis is written we refer the reader to [94].

CiaoPP's sequential execution time estimation at compile time only deals with a subset of Ciao Prolog while real programs can be written using the whole set. So, in order to test our approach, another option is to obtain the equations in a two-phase process. The stages are profiling and linear regression respectively. In the profiling phase, sequential execution times are measured directly over the platform using best and worst input values (if possible) for lower and upper bounds respectively. For example, for *qsort* (see code in Figure 6.2) we have to deal with the degree of order of the elements. The worst case is a list whose elements are already ordered while the best case is a uniformly distributed list of random elements. In the second phase, the linear cost model of the execution time is obtained. For example, for qsort the model for the lower bound is $A * 1 + B * X + C * X * log(X)$ where X is the length of the input list and *A, B and C* are the parameters of the cost model that we want to estimate. Models are obtained with *CiaoPP*, performing execution steps consumption analysis [95]. Then the parameters of the average cost models are estimated using the Ciao costmodel module [85]. Table 6.1 shows both upper and lower bounds on the execution time of qsort.

Furthermore, there are more techniques for estimating sequential execution times at run-time. They can be directly measured or, if we need more information, they can be profiled (see

Figure 6.2: Qsort code.

```
qsort([],[]).
qsort([X|L],R) :-
        partition(L,X,L1,L2),
        qsort(L2,R2),
        qsort(L1,R1),
        append(R1,[X|R2],R).

partition([],_,[],[]).
partition([E|R],C,[E|Left1],Right) :-
        E < C, !,
        partition(R,C,Left1,Right).
partition([E|R],C,Left,[E|Right1]) :-
        E >= C,
        partition(R,C,Left,Right1).

append([],X,X).
append([H|X],Y,[H|Z]) :- append(X,Y,Z).
```

Section 6.3).

6.3 Profiling

We have developed a profiling [89] that can be used for estimating execution times. It is a profiling method for logic programs that owns the following features:

1. The accumulated execution time of program procedures do not overlap, which means that the total resource usage of a program can be computed in a compositional way, by adding the resource usage of all its procedures.

2. It gives separate accumulated resource usage information of a given procedure depending on where it is called from.

3. It is tightly integrated in a program development framework which incorporates in a uniform way run-time checking, static verification, unit-testing, debugging, and optimization. Our profiler is used for run-time checking as well as debugging purposes whitin this framework.

4. It includes a (configurable) automatic process for detecting procedures that are performance bottlenecks following several heuristics.

5. The user can configure the best trade-off between overhead and collected information.

Other features of our profiler are the combination of time profiling with count profiling, which has proved to be non-trivial [82], and the modularity, which allows specifying which modules should be instrumented for profiling. These and other original features of our profiler are possible thanks to the usage of the `Ciao`'s module system and the automatic code transformation through `Ciao`'s semantic packages.

The profiling technique is based on associating (either automatically or manually) cost centers to certain program elements, such as procedures or calls in clause bodies. The concept of *cost center* is inspired in the one defined by Morgan [102] in the context of functional languages. However, we have adapted this concept to deal with the unique features of logic programming. A *cost center* for us is a place in a program (typically a predicate or a literal in a body clause) where data about computational events are accumulated each time the place is reached by the program execution control flow. In our current implementation both predicates and literals can be marked as cost centers. We introduce a special cost center, named *remainder cost center* (denoted *rcc*), used for accumulating data about events not corresponding to any defined cost center.

In order to deal with the control flow of Prolog, we adopt the "box model" of Lawrence Byrd [23], where predicates (procedures) are seen as "black boxes" in the usual way. However, due to backtracking, the simple call/return view of procedures is not enough, and we have a "4-port box view" instead. Thus, given a *goal* (i.e., a unique, run-time call to a predicate), there are four ports (events) in Prolog execution: (1) *call* goal (start to execute goal), (2) *exit* goal (succeed in producing a solution to goal), (3) *redo* goal (attempt to find an alternative solution to goal), and (4) *fail* goal (exit with failure, if no further solutions to goal are found). Thus, there are two ports for "entering" the box (*call* and *redo*), and two ports for "leaving" it (*exit* and *fail*).

6.3.1 Definitions

Lets see some definitions of elements of our profiler:

Definition 1 (Calls relation)**.** *We define the* calls *relation between predicates in a program as follows: p calls q, written $p \rightsquigarrow q$, if and only if a literal with predicate symbol q appears in the body of a clause defining p. Let $\rightsquigarrow^\star$ denote the reflexive transitive closure of $\rightsquigarrow$.*

Definition 2 (Cost center set)**.** *Given a program P to be profiled, the cost center set for P (denoted C_P), is the set $C_P = \{p \mid p \text{ is a predicate of } P \text{ marked as a cost center}\} \cup \{rcc\}$, where rcc is the remainder cost center.*

Definition 3 (Cost center call-graph)**.** *The* cost center call-graph *of a program P (denoted G_P) is the graph defined by the set of nodes C_P and the set of edges V, such that $(p,q) \in V$ iff:*

1. *p is not the remainder cost center (i.e., $p \neq rcc$), $q \neq rcc$, $p \rightsquigarrow^\star q$ and there is no $t \in C_P$ such that $p \rightsquigarrow^\star t$ and $t \rightsquigarrow^\star q$, or*

2. *$p = rcc$, and for all s being a literal of the program P, we have: (1) $q = s$ if $s \in C_P$, (2) $q = r$ if exist $r \in C_P$ such that $s \rightsquigarrow^\star r$ and there is no $t \in C_P$ such that $s \rightsquigarrow^\star t$ and $t \rightsquigarrow^\star r$, or (3) $q = rcc$ otherwise.*

Definition 4 (Edge cumulated resource usage)**.** *Each edge $(c,d) \in G_P$ has a label, R_{cd}, which represents the cumulated resource usage of the computation since cost center d was entered from cost center c, until a new cost center is entered or the computation finishes. This allows to give separate resource usage information of a given procedure depending on where it is called from.*

Figure 6.3: Profiling example. Source code, call-graph and cost centers call-graph.

```
:- cost_center p/0, q/0.
p :-
    q, r.
```

Example 1. *Figure 6.3 illustrates how the resource usage information is stored in the edges of the cost center call-graph during the profiling process. At any time in this process, only one edge is active. When execution enters a predicate which is defined as a cost center, the resource usage monitored so far is stored in the active edge, and other edge is activated. Consider the program `p`, and its call-graph and cost center call-graph. Before starting the program execution, the active edge is `(rcc, rcc)`. Then, `p` is called. Since `p` is defined as a cost center, the resource usage monitored so far is cumulated in the active edge `(rcc, rcc)`, the counters are reset, and the active edge changes to `(rcc, p)`. Then, the execution of the body of `p` starts by executing `q`. Since `q` is not defined as a cost center, the active edge remains the same as before, `(rcc, p)` (and the counters are not reset). When the execution of `q` finishes, `r` is called. Since `r` is defined as a cost center, the resource usage monitored so far is cumulated in the active edge `(rcc, p)`, the counters are reset, and the active edge changes to `(p, r)`. Since `r` is the last call in the definition of `p`, when the execution of `r` finishes, the resource usage monitored far is cumulated in `(p, r)` and the program execution finishes.*

Definition 5 (Cumulated resource usage of a cost center)**.** *The cumulated resource usage in a given cost center d (denoted R_d) is the cumulated resource usage of the computation since cost center d is reached, until a new cost center is entered or the computation finishes.*

Lemma 1. *The cumulated resource usage in a given cost center is the sum of the cumulated resource usages of its incident edges: $R_d = \sum_{(c,d)\in V} R_{cd}$.*

Proof. Trivial, based on the cumulated resource usage of an edge. □

Lemma 2. *The total resource usage of a program P, denoted T_P, is the addition of the cumulated resource usage of all its cost centers: $R_P = \sum_{c\in C_P} R_c$*

Proof. Trivial, by the definition of cumulated resource usage of a cost center. □

Note that our definition of cumulated resource of a cost center is compositional, in the sense that the total resource usage of a set predicates can be computed by adding the cumulated resource usage of each predicate. This doesn't happens in traditional profilers, where the cumulated execution time of different predicates may overlap (and thus, the addition of them may be greater than their actual resource usage).

6.3.2 Using Profiling Techniques in Granularity Control

For understanding how profiling techniques can be succesfully applied to granularity control it is useful to know how they provide they results.

Figure 6.4: An example of profiler output.

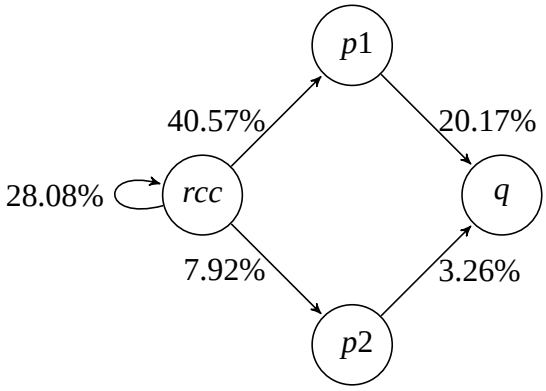
```
:- module(_, _, [profiler]).
:- cost_center p1/0, p2/0, q/1.

p1 :-
    q(a),
    q(b),
    q(c),
    q(d).


p2 :-
    q(a).

main :-
    p1,
    p2.

q(a).
q(b).
q(c).
q(d).
```



Example 2. *Figure 6.4 contains a program with a set of predicates defined as cost centers and the output of the profiler after profiling the goal* main. *The measured resource is execution time. The percentage of execution time that each cost center predicate consumes can be obtained by adding the labels of all its incident edges. Thus* p1, p2 *and* q *consumes 40.57%, 7.92% and 23.43% respectively.*

In this case we have simplified the output. Our profiling tool returns the number of ticks (or units of time) accumulated in each cost edge (according to the procedure descibed in Section 6.3.1). But the same information is clearer with percentages.

Other works for improving automatic parallelization are based on balancing the load of the parallel executions (see Performance Factors in Section 2.1.3). It has been shown that these techniques reduce the execution time of the parallel programs [115, 120]. Load balancing schedulings guarantee the best usage of the execution units.

Lets see a simple example of how we can apply our profiling to load balancing at compile-time. As we have explained in Chapter 3 there are many ways of executing a task in parallel. Let g be a task and $g = g_1, g_2, g_3$ where all the g_i's are candidates for parallel execution. In a system with 3 (or more) processors g can be executed in the following ways (note that the parallel execution operator is represented as &): g_1, g_2, g_3 / $g_1 \& g_2 \& g_3$ / $g_1 \& (g_2, g_3)$ / $g_2 \& (g_1 \& g_3)$ / $g_3 \& (g_1, g_2)$.

The load of executing two tasks x, y sequentially is equal to $cost(x) + cost(y)$ while if they are executed in parallel is equal to $max(cost(x), cost(y))$ where $cost(t)$ is the cost of executing the task t.

Automatic parallelization will transform $g = g_1, g_2, g_3$ into $g = g_1 \& g_2 \& g_3$. This transformation reduces the execution time of g from $T_{g_1} + T_{g_2} + T_{g_3}$ to $max(T_{g_1}, T_{g_2}, T_{g_3})$ and keeps the 3 processors busy this amount of time. Remember that T_i refers to the sequential execution time of the task i.

Assume that each T_{g_i} is executed in the processor i and suppose that $T_{g_1} = T_{g_2} + T_{g_3}$. This means that the processors 2 and 3 will be idle T_{g_3} and T_{g_2} units of time respectively.

The fact $T_{g_1} = T_{g_2} + T_{g_3}$ can be derived from the information provided by our profiler. Then if the automatic parallelization process takes it into account it will transform $g = g_1, g_2, g_3$ into $g = g_1 \& (g_2, g_3)$. This transformation reduces the execution time of g from $T_{g_1} + T_{g_2} + T_{g_3}$ to $max(T_{g_1}, (T_{g_2} + T_{g_3}))$ which in this case is equal to $max(T_{g_1}, T_{g_2}, T_{g_3})$, i.e., it is the same execution time than the one of the transformation performed without profiling information. Nevertheless this scheduling keeps only 2 processors busy this amount of time.

Thus taking into account profiling information in automatic parallelization provides load balancing scheduling that optimizes the usage of the execution units which, at the same time, increases the system performance.

Chapter 7

Experimental Results

We have developed a prototype (Section 7.1) of a fuzzy task scheduler based on the approach described in Chapter 5. We have prepared a common framework to test the behavior of a set of different heuristics (Section 7.2) and we have compared them also with the rules of the conservative approach (Chapter 4) in order to be able to select the best results (Section 7.3). For a better understanding of these experiments, we present the behavior of our prototype for a progression of execution time data (Section 7.4). Finally, we have tested our prototype with real programs (Section 7.5) in order to demonstrate that it can be successfully applied in practice.

7.1 Prototype Implementation

All the selection methods have been implemented in *Ciao Prolog*. The classical logic rules have been implemented using the CLP(Q) package and the fuzzy logic rules using the *Rfuzzy* package.

We have decided to use logic programming for implementing our approach because of its simplicity and for taking the advantage of some useful extensions provided by the *Ciao Prolog* framework. In particular, *Ciao Prolog* has integrated static analysis techniques for obtaining upper and lower bounds on execution times and a fuzzy library for the calculation of certainty factors.

As explained before, in our new approach to granularity control, the decision of how to execute is based on the certainty factors associated to both, sequential and parallel executions. So that, first of all, we have to quantify such certainty and then decide how to execute. The value to the certainty factors is provided by fuzzy rules that are able to combine fuzzy values using aggregation operators. According to RFuzzy syntax:

$$SEQ(P,V_s):\ op\ cond_1(V_1), cond_2(V_2), ...cond_n(V_n).$$
$$PAR(P,V_p):\ op'\ cond_1'(V_1'), cond_2'(V_2'), ...cond_n'(V_n').$$

The truth value V_s represents how much executing the program P in a sequential way is adequate. V_s is obtained by combining the truth values of the partial conditions $V_1,...,V_n$ with the

aggregation operator op. Symmetrically, V_p represents how much adequate is the parallel execution for the program P.
The bigger factor (SEQ or PAR) will point out the selected execution (sequential or parallel).

In order to test the behavior of our method we have developed a set of conditions comparing a group of values of execution times: $\{T_p^l, T_p^m, T_p^u, T_s^l, T_s^m, T_s^u\}$ by pairs. The comparison that makes each condition is calculated with the fuzzy relations *quite_greater* and *rather_greater* (represented in Figure 5.1), whose definitions are:

$$quite\_greater(X) = \begin{cases} 0 & if \quad X \leq -7 \\ \frac{X+7}{15} & if \quad -7 < X < 8 \\ 1 & if \quad X \geq 8 \end{cases}$$

$$rather\_greater(X) = \begin{cases} 0 & if \quad X \leq -14 \\ \frac{X+14}{29} & if \quad -14 < X < 15 \\ 1 & if \quad X \geq 15 \end{cases}$$

We also use the *relative harmonic difference*, an experimental relation described in [86] as follows:
$harmonic\_diff(X,Y) = (X-Y) * (1/X + 1/Y)/2.$
We have selected this relation because it compares two numbers in a relative and symmetric way, i.e.:
$harmonic\_diff(X,Y) = -harmonic\_diff(Y,X).$
The *harmonic difference* only works well for positive numbers, but as we are working with execution times, it is enough for our purposes.

These fuzzy relations can be redefined with different bounds, although in this prototype we have only used the values 0, 7 and 14. These bounds have been selected according to the magnitude of the execution times that we provide for the programs (see Table 7.2) in order to obtain significant results depending on the selected fuzzy relation.

7.2 Heuristic Comparison

In this section we discuss the evaluation of our prototype with different aggregation operators. A suite of benchmarks to test the prototype has been developed. Each benchmark has been defined in terms of its execution times (average, upper and lower bounds on parallel and sequential execution times) in order to see if the new approach provides better results than the conservative one. Obviously, in real cases, these values will need to be estimated at compile-time using a program analyzer like, for example, *CiaoPP* [60, 86]. Table 7.2 contains the description of the benchmarks. Each row shows the information of one program. The first column contains the name of the program and, under it and between brackets, the name of the figure which contains the graphical representation of the benchmark. This figure allows to identify the optimal execution in a graphic way. The following columns show : T_s^l (lower bound on sequential execution time), T_s^m (average sequential execution time), T_s^u (upper bound on sequential execution time), T_p^l (lower bound on parallel execution time), T_p^m (average parallel execution time) and T_p^u (upper bound on parallel execution time). Each execution time is in microseconds.

Table 7.1: Aggregation operators execution time.

Program	Aggregation Operator		
	max	dprod	dluka
p1	1.23	1.11	1.04
p2	0.42	0.51	0.45
p3	0.93	0.88	0.88
p4	0.43	0.51	0.45
p5	0.62	0.76	0.63
p6	0.56	0.62	0.57
average	0.70	0.73	0.67

Figures 7.1, 7.2, 7.3, 7.4, 7.5 and 7.6 describe the benchmarks in a graphic way. In horizontal we find both (parallel and sequential) executions. In vertical we find, for each execution, the interval comprised between its upper and lower bound on execution time.

To make things simpler we have made the following translations in the conditions: the fuzzy set is called *gt* and the *relative harmonic difference* relation is called *hd*.

The rules of *fuzzy logic* for calculating each condition PAR_i or SEQ_i ($i \in [1,2,3]$, see Table 7.3) have been composed using several aggregation operators but the results have shown that only the t-conorms max (*max*), Lukasiewicz (*dluka*) and sum (*dprod*) are correct (i.e. always suggest the optimal execution) so we do not mention the rest[1] in the results. We have seen how the three t-conorms max (*max*), Lukasiewicz (*dluka*) and sum (*dprod*) have the same behavior. Thus, in order to chose one of these aggregation operators, we have followed the criteria of the one more efficiently evaluated. In this sense, we have measured the execution time of evaluating the condition PAR_1 for each program using the three operators. These execution times have been obtained over an Intel platform (Intel Pentium 4 CPU 2.60GHz). They are shown in Table 7.1. The first column shows the name of the program (see Table 7.2) and the three next ones, the aggregation operators. Each row shows the execution time (in microseconds) of the evaluation of the condition PAR_1 (see Table 7.3) for the program using the three mentioned operators. The last row contains, for each operator, an average value on the execution time of evaluating such condition for all the programs. As we can see, the results are very similar for the aggregation operators *max* and *dluka* while for *dprod* are almost always bigger. Although *max* is a little bit less efficient (on average) than *dluka*, *max* seems to be the best option due to its simplicity.

The whole set of proposed certainty factors and the results for each approach are shown in Table 7.3. They correspond to the case of parallelizing a sequential program (i.e., where the action taken by default when there is no evidence towards executing is parallel is to execute sequentially). The first column shows the name of the program. The second column shows what would be the right (optimal) decision about the type of execution that should be performed (either parallel or sequential). The rest of the columns contain the results of evaluating the

[1]The rest of the tested operations are: *min*, *luka* and *prod*.

Table 7.2: Benchmarks -times in microseconds-.

Program \ Time	T_s^l	T_s^m	T_s^u	T_p^l	T_p^m	T_p^u
p1 (Figure 7.1)	400	600	800	100	175	250
p2 (Figure 7.2)	50	175	300	350	550	750
p3 (Figure 7.3)	250	525	800	300	375	450
p4 (Figure 7.4)	50	150	250	100	325	550
p5 (Figure 7.5)	200	400	600	200	325	450
p6 (Figure 7.6)	150	325	500	100	275	450

Figure 7.1: Program p1 execution times.

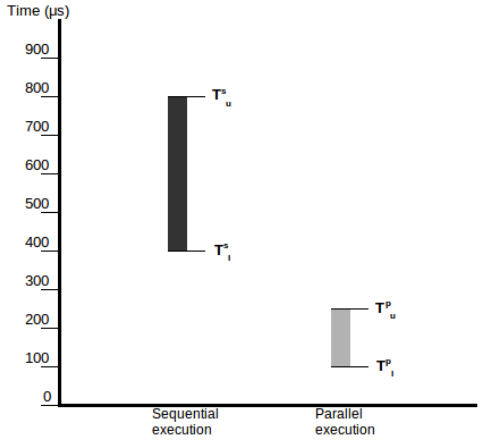


Figure 7.2: Program p2 execution times.

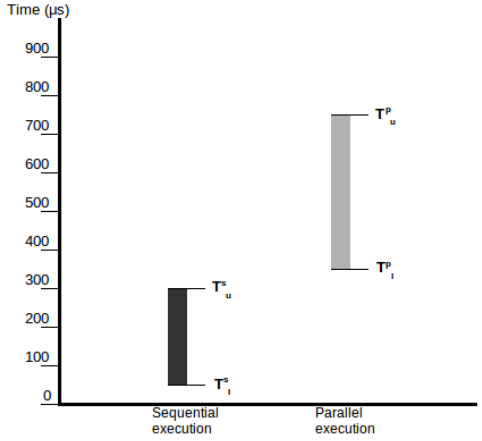


Figure 7.3: Program p3 execution times.

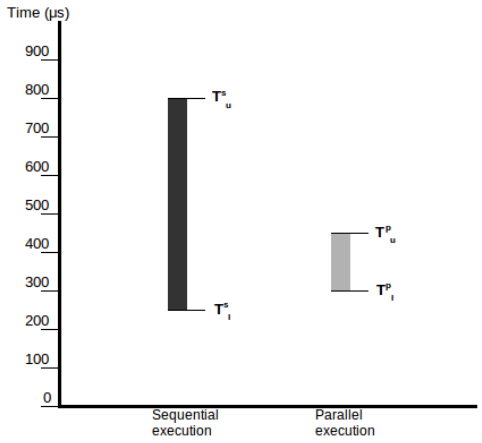


Figure 7.4: Program p4 execution times.

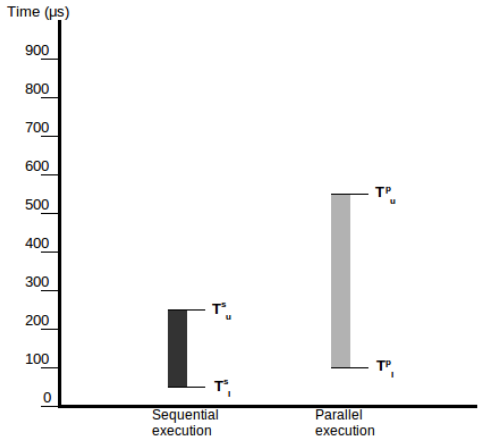


Figure 7.5: Program p5 execution times.

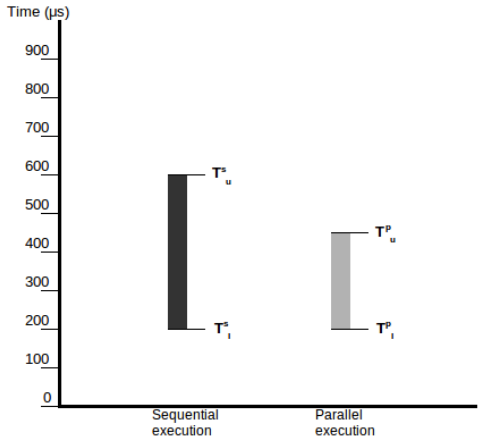
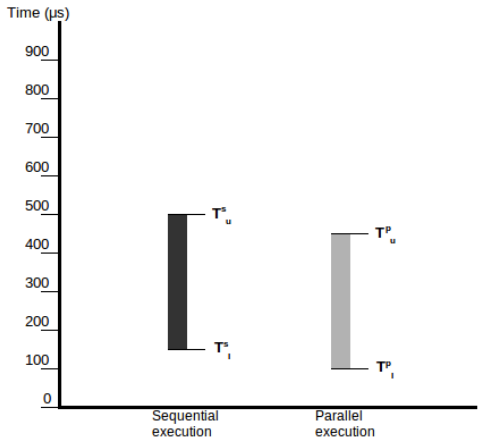


Figure 7.6: Program p6 execution times.



conditions. Columns 3 and 4 contain the results obtained using the conservative approach, while columns 5-18 contain the results obtained using our proposed conditions based on fuzzy logic. Each column in the later group of columns corresponds to a different fuzzy condition. The selected type of execution (using the process explained in Section 5.1) are highlighted. SEQ_i and PAR_i are the truth values obtained for the certainty factors of the sequential and parallel executions of the program p_i. We have performed the experiments for two different levels of flexibility using *quite_greater* and *rather_greater* respectively. The decisions made by using the fuzzy conditions are always the optimal ones for these experiments. However, the conservative approach (*classical logic*) disagrees with the optimal ones in half of the cases. For example, the condition $T_p^u \leq T_s^l$ holds for *p1* (see Figure 7.1). Thus, the parallel execution of *p1* is more efficient than the sequential one. In this case, both the *conservative approach* (*classical logic*) and the *fuzzy logic* approach agree in that the execution of *p1* should be parallel. The converse condition ($T_s^u \leq T_p^l$) holds for program *p2* (see Figure 7.2), and thus, the optimal action is executing it sequentially. In this case, also both approaches agree in that the execution of *p2* should be parallel.

For programs 3-6, the classical logic truth values (PAR_c and SEQ_c) are always zero, which means that the suggested type of execution is *sequential* for all of these programs (i.e., the default type of execution). However, from Figures 7.3, 7.4, 7.5 and 7.6, we can see that in some cases the optimal decision is to execute these programs in parallel.

For example, consider program *p3* (see Figure 7.3). We have that $T_p^u = 450\ \mu s$ and $T_s^l = 250\ \mu s$, and thus $T_p^u \leq T_s^l$ does not hold. The decision of executing *p3 sequentially* made by *classical logic* is safe. However, in this case, since $T_s^u = 800\ \mu s$, assuming that *p3* is run a significant number of times, we have that on average, executing *p3* in parallel would be more efficient than executing it sequentially. In contrast, our proposed fuzzy approach selects the optimal type of execution for *p3*: its two subtasks should be executed in parallel. Program *p4* (see Figure 7.4) represents the opposite case. In this case $T_s^u = 250\ \mu s$ and $T_p^l = 100\ \mu s$ so $T_s^u \leq T_p^l$ does not hold.

Table 7.3: Selected executions using the whole set of rules.

Program	Optimal	Classical Logic (Greater)		Fuzzy Logic (Quite greater)													
		Classical		Fuzzy 1		Fuzzy 2		Fuzzy 3		Fuzzy 4		Fuzzy 5		Fuzzy 6		Fuzzy 7	
		PAR_c	SEQ_c	PAR_1	SEQ_1	PAR_2	SEQ_2	PAR_3	SEQ_3	PAR_4	SEQ_4	PAR_5	SEQ_5	PAR_6	SEQ_6	PAR_7	SEQ_7
p1	Parallel	1	0	0.73	0.48	0.73	0.48	0.73	0.48	0.57	0.35	0.57	0.35	0.57	0.35	0.57	0.36
p2	Sequential	0	1	0.48	0.93	0.49	0.93	0.49	0.93	0.34	0.58	0.33	0.59	0.34	0.58	0.31	0.58
p3	Parallel	0	0	0.56	0.54	0.58	0.54	0.58	0.54	0.48	0.44	0.48	0.44	0.48	0.44	0.47	0.44
p4	Sequential	0	0	0.5	0.61	0.5	0.61	0.5	0.61	0.41	0.52	0.41	0.52	0.41	0.52	0.41	0.52
p5	Parallel	0	0	0.54	0.53	0.55	0.53	0.55	0.53	0.47	0.45	0.47	0.45	0.47	0.45	0.47	0.45
p6	Parallel	0	0	0.56	0.52	0.56	0.52	0.56	0.52	0.48	0.45	0.48	0.45	0.48	0.45	0.48	0.45

Program	Optimal	Classical Logic (Greater)		Fuzzy Logic (Rather greater)													
		Classical		Fuzzy 1		Fuzzy 2		Fuzzy 3		Fuzzy 4		Fuzzy 5		Fuzzy 6		Fuzzy 7	
		PAR_c	SEQ_c	PAR_1	SEQ_1	PAR_2	SEQ_2	PAR_3	SEQ_3	PAR_4	SEQ_4	PAR_5	SEQ_5	PAR_6	SEQ_6	PAR_7	SEQ_7
p1	Parallel	1	0	0.62	0.49	0.62	0.49	0.62	0.49	0.53	0.42	0.53	0.42	0.53	0.42	0.54	0.42
p2	Sequential	0	1	0.49	0.72	0.49	0.72	0.49	0.72	0.41	0.54	0.41	0.55	0.41	0.54	0.4	0.54
p3	Parallel	0	0	0.53	0.52	0.54	0.52	0.54	0.52	0.49	0.47	0.49	0.47	0.49	0.47	0.48	0.47
p4	Sequential	0	0	0.5	0.55	0.5	0.55	0.5	0.55	0.45	0.51	0.45	0.51	0.45	0.51	0.45	0.51
p5	Parallel	0	0	0.52	0.51	0.52	0.51	0.52	0.51	0.48	0.47	0.48	0.47	0.48	0.47	0.48	0.47
p6	Parallel	0	0	0.53	0.51	0.53	0.51	0.53	0.51	0.49	0.47	0.49	0.47	0.49	0.47	0.49	0.47

Conditions:

PAR_c is $T_p^u \leq T_s^l$

SEQ_c is $T_s^u \leq T_p^l$

PAR_1 is $max(gt(T_s^l/T_p^u), gt(T_s^l/T_p^l), gt(T_s^m/T_p^m))$

SEQ_1 is $max(gt(T_p^l/T_s^u), gt(T_p^l/T_s^l), gt(T_p^m/T_s^m))$

PAR_2 is $max(gt(T_s^l/T_p^u), gt(T_s^l/T_p^l), gt(T_s^u/T_p^u))$

SEQ_2 is $max(gt(T_p^l/T_s^u), gt(T_p^l/T_s^l), gt(T_p^u/T_s^u))$

PAR_3 is $max(gt(T_s^l/T_p^u), gt(T_s^l/T_p^l), gt(T_s^m/T_p^m), gt(T_s^u/T_p^u))$

SEQ_3 is $max(gt(T_p^l/T_s^u), gt(T_p^l/T_s^l), gt(T_p^m/T_s^m), gt(T_p^u/T_s^u))$

PAR_4 is $rel\_hd(0.5 * hd(T_s^m, T_p^m) + 0.25 * hd(T_s^u, T_p^u) + 0.25 * hd(T_s^l, T_p^l))$

SEQ_4 is $rel\_hd(0.5 * hd(T_p^m, T_s^m) + 0.25 * hd(T_p^u, T_s^u) + 0.25 * hd(T_p^l, T_s^l))$

PAR_5 is $rel\_hd((hd(T_s^m, T_p^m) + hd(T_s^u, T_p^u) + hd(T_s^l, T_p^l))/3)$

SEQ_5 is $rel\_hd((hd(T_p^m, T_s^m) + hd(T_p^u, T_s^u) + hd(T_p^l, T_s^l))/3)$

PAR_6 is $rel\_hd(0.25 * hd(T_s^m, T_p^m) + 0.5 * hd(T_s^u, T_p^u) + 0.25 * hd(T_s^l, T_p^l))$

SEQ_6 is $rel\_hd(0.25 * hd(T_p^m, T_s^m) + 0.5 * hd(T_p^u, T_s^u) + 0.25 * hd(T_p^l, T_s^l))$

PAR_7 is $rel\_hd(0.25 * hd(T_s^m, T_p^m) + 0.25 * hd(T_s^u, T_p^u) + 0.5 * hd(T_s^l, T_p^l))$

SEQ_7 is $rel\_hd(0.25 * hd(T_p^m, T_s^m) + 0.25 * hd(T_p^u, T_s^u) + 0.5 * hd(T_p^l, T_s^l))$

But in this case $T_p^u = 550\ \mu s$ and $T_s^u = 250\ \mu s$. Thus, the best choice seems to be executing *p4* sequentially. This is the type of execution suggested by our fuzzy conditions. However, using classical logic, the selected execution is sequential (the one selected by default when none of the sufficient conditions PAR_c nor SEQ_c hold). However, our fuzzy logic conditions provide enough evidences that support the decision of executing in parallel.
In the situations illustrated by the last two programs it is not so clear what type of execution should be selected. For program *p5* we have that $T_p^u = 450\ \mu s$ and $T_s^l = 200\ \mu s$. Thus, since the sufficient condition $T_p^u \leq T_s^l$ for executing in parallel does not hold, it seems that the program should be executed sequentially. However, since $T_p^l = 200\ \mu s$ and $T_s^u = 600\ \mu s$, the sufficient condition $T_s^u \leq T_p^l$ for executing in parallel does not hold either. Now, using our fuzzy logic approach, taking the four values T_p^l, T_p^u, T_s^l *and* T_s^u into account, a certainty factor of nearly 0.5 suggests that the best choice is to execute *p5* in parallel.
For program *p6* (see Figure 7.6), none of the sufficient conditions $T_p^u \leq T_s^l$ and $T_s^u \leq T_p^l$ (for selecting parallel and sequential execution respectively) hold. However, since $T_p^u \leq T_s^u$ and $T_p^l \leq T_s^l$ hold, it is clear that the execution time of the sequential execution is going to belong to an interval whose limits are bigger than the limits of the parallel execution. Thus, is it more likely that the execution time of the parallel execution be less than the execution time of the sequential one, so that the right decision seems to execute *p6* in parallel. We can see that our proposed fuzzy conditions also suggests the parallel execution.

Finally, we can see that in those cases in which classical logic suggests a type of execution (with truth value 1), our fuzzy logic approach suggests the same type of execution (sequential or parallel).

7.3 Selected Fuzzy Model

Table 7.3 shows that all the fuzzy conditions (*Fuzzy 1-7*) select the same type of execution, sequential or parallel (independently of the fuzzy set used, either *quite_greater* or *rather_greater*). Our goal is to detect those situations where the parallel execution is faster than the sequential one, such that a conservative (safe) approach is not able to detect it but the fuzzy approach is. Approaches *Fuzzy 4*, *5*, *6* and *7* suggest parallel execution with less evidence than *Fuzzy 1*, *2* and *3* for both fuzzy sets (*quite_greater* and *rather_greater*). As we are interested in suggesting to execute in parallel with evidences as bigger as possible we rule out this subset of conditions and we focus our attention in the first set. Both *Fuzzy 2* and *3* obtain the same values in all cases. Furthermore they provide higher evidences for parallel execution than the condition *Fuzzy 1*. This fact can be seen in programs *p3*, *p5* and *p6*. As *Fuzzy 2* is a subset of *Fuzzy 3*, evaluating the first one is more efficient than the second one (the *Fuzzy 3* condition has one more comparison). Thus, the condition that we have selected is *Fuzzy 2*:

$$\boxed{PAR \text{ is } max(gt(T_s^l/T_p^u), gt(T_s^l/T_p^l), gt(T_s^u/T_p^u))}$$

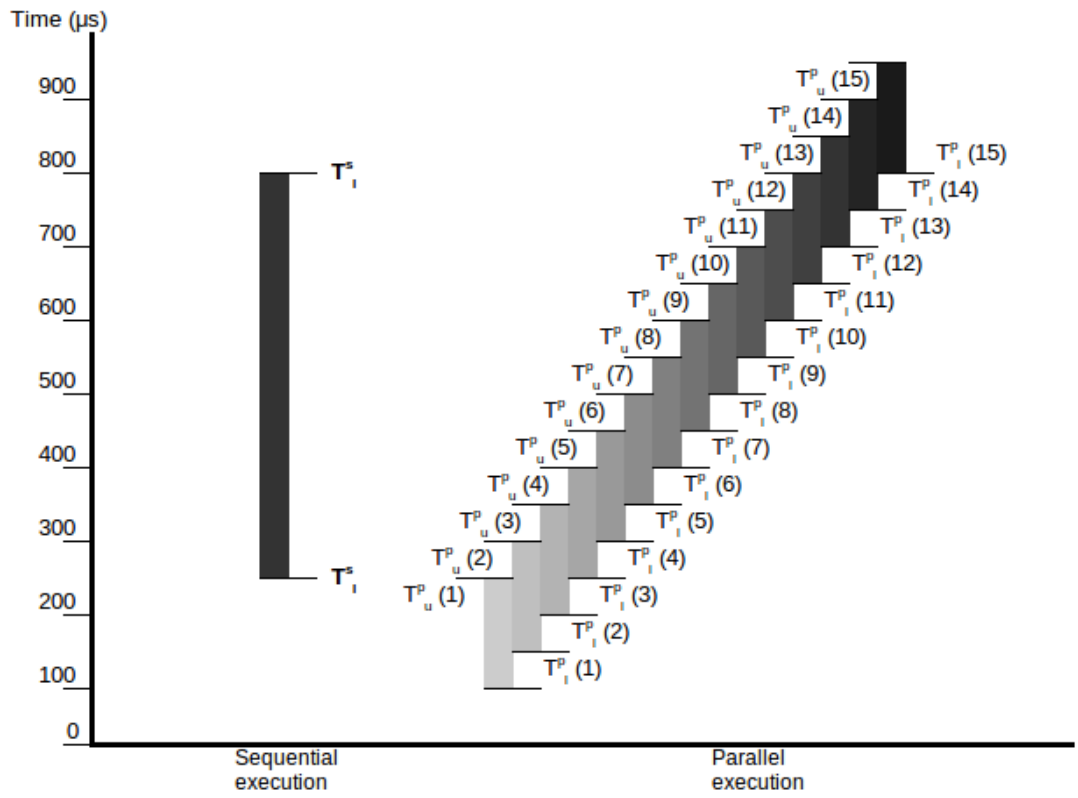


Figure 7.7: Progression of executions of the example program p3.

This condition obtains a better average case behavior by relaxing decision conditions (and losing some precision). There may be cases in which our approach will select the slowest execution, however it will select the fastest one in a bigger number of cases. This tradeoff between safety and efficiency makes this new approach only applicable to non-critical systems, where no constraints about execution times must be met, and a wrong decision will only cause a slowdown which is admissible. In the same way that it happens in the conservative approach, the fuzzy approach for sequentializing a parallel program is also symmetric to the problem of parallelizing a sequential program. The condition that we have selected for sequentializing a parallel program is:

$$\boxed{SEQ \text{ is } max(gt(T_p^l/T_s^u), gt(T_p^l/T_s^l), gt(T_p^u/T_s^u))}$$

7.4 Decisions Progression

Focusing on program *p3* and using the fuzzy set *quite_greater* with the selected fuzzy model (Section 7.3) we have developed an incremental experiment whose results are in Table 7.4. The main goal is to see how with this fuzzy logic approach we can select the optimal execution in those cases in which the conservative approach is not able to give a conclusion, and also, how our fuzzy logic approach detects all situations (safely) detected optimal by the conservative

Table 7.4: Progression of decisions using the fuzzy set quite greater.

Execution	Optimal	Classical Logic (Greater)		Fuzzy Logic (Quite greater)	
		Classical		Fuzzy 2	
		PAR_c	SEQ_c	PAR_2	SEQ_2
p3_execution1	Parallel	1	0	0.68	0.49
p3_execution2	Parallel	0	0	0.64	0.5
p3_execution3	Parallel	0	0	0.61	0.52
p3_execution4	Parallel	0	0	0.6	0.53
p3_execution5	Parallel	0	0	0.58	0.54
p3_execution6	Parallel	0	0	0.57	0.56
p3_execution7	Sequential	0	0	0.56	0.57
p3_execution8	Sequential	0	0	0.55	0.58
p3_execution9	Sequential	0	0	0.54	0.6
p3_execution10	Sequential	0	0	0.54	0.61
p3_execution11	Sequential	0	0	0.53	0.62
p3_execution12	Sequential	0	0	0.53	0.64
p3_execution13	Sequential	0	0	0.52	0.65
p3_execution14	Sequential	0	0	0.52	0.66
p3_execution15	Sequential	0	1	0.52	0.68

Conditions:

PAR_c is $T_p^u \leq T_s^l$

SEQ_c is $T_s^u \leq T_p^l$

PAR_2 is $max(gt(T_s^l/T_p^u), gt(T_s^l/T_p^l), gt(T_s^u/T_p^u))$

SEQ_2 is $max(gt(T_p^l/T_s^u), gt(T_p^l/T_s^l), gt(T_p^u/T_s^u))$

approach. Figure 7.7 shows all the execution scenarios. The sequential execution times are fixed, while the parallel execution ones depend on each scenario. The later are represented by pairs $(T_p^l(i), T_p^u(i))$ where i is the concrete case. The parallel execution times of each scenario are the times of the previous one plus 50 units, in order to appreciate the progression. The times of the first scenario are $T_p^l(1) = 100\ \mu s$ and $T_p^u(1) = 250\ \mu s$. Attending to classical logic we can see how only when $PAR_c = 1$ or $SEQ_c = 1$ we obtain a justified answer (that the program must be executed in parallel or sequentially respectively). In the rest of the cases the selected type of execution is *sequential* by default, since we are following the philosophy of parallelizing a sequential program, and there are no evidences towards either type of execution. On the other hand, fuzzy logic always selects the optimal execution (supported by evidences).

7.5 Experiments with Real Programs

Former experiments (Section 7.2) have shown that our fuzzy granularity control framework is able to capture which is the optimal type of execution on average. Moreover, in order to ensure

Table 7.5: Real benchmarks.

Qsort	qsort(n) sorts a list of n random elements.
Substitute	substitute(n) replaces by 2 the x's that appears in an expresion (x + x ...) composed by n +'s and n+1 x.
Fib	fib(n) obtains the nth Fibonacci number.
Hanoi	hanoi(n) solves Hanoi puzzle with 3 rods and n disks.

that our approach can be applied in practice, we have performed some experiments with real programs (and real execution times). The experimental assessments have been made over an UltraSparc-T1, 8 cores x 1GHz (4 threads per core), 8GB of RAM, SunOS 5.10.

We have tested the *fuzzy model* selected in Section 7.3, so that only upper and lower bounds on (parallel and sequential) execution times were needed. Sequential execution times have been measured directly over the execution platform (executing the worst and best possible cases) while the parallel ones have been estimated.

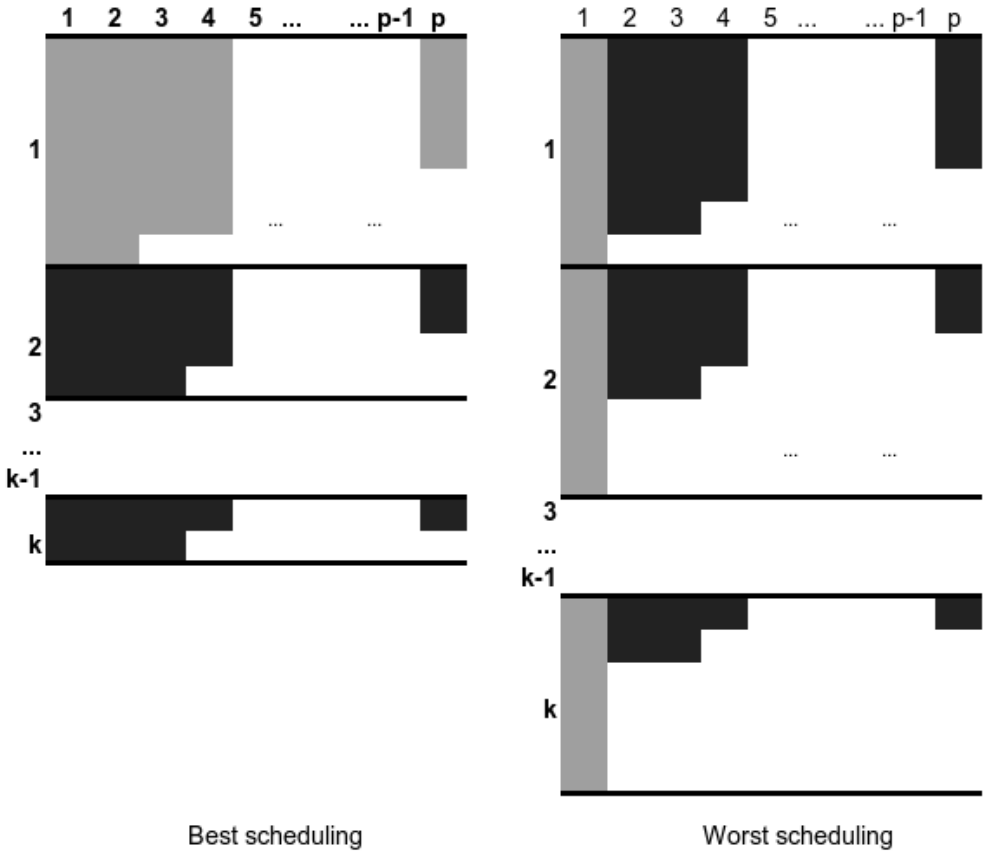
The number of cores of the processor is denoted as p, the number of tasks (candidates for parallel or sequential execution) as n, and the relation $\lceil n/p \rceil$ is denoted as k. We consider two different overheads of parallel execution: (a) the time needed for creating n parallel tasks, called *Create(n)*, and (b) an upper bound on the time taken from the point in which a parallel subtask g_i is created until its execution is started by a processor, denoted as $SysOverhead_i$. Both types of overheads have been experimentally measured for the execution platform. For the first one, we have measured directly the time of creating p threads. The second one has been obtained by using the expression $(S/2) - P$, where S and P are the measured execution times of a program consisting of two perfectly balanced parallel tasks running with one and two threads respectively.

There are different ways of executing a task in parallel depending on the scheduling. The highest parallel execution time will be the one with the worst scheduling (i.e. the one in which the cores are idle as much as possible). Figure 7.8 represents both, the best and the worst possible scheduling scenarios. In each scenario, columns show the number of cores cores (from 1 to p) and rows the execution iterations (from 1 to k). In each iteration the level of occupation of each core is colored while the amount of time in which it is idle is in white. Longest tasks are the lightest ones. It can be seen how the way in which these tasks are assigned to the cores has a direct implication on the efficiency.

Suppose that the execution times of the n subtasks $Ts_1, Ts_2, ..., Ts_n$ of g (Chapter 3) are in descending order. Then we can estimate parallel execution time of both scheduling cases as follows:

$$T_p^{best} = Create(p) + \sum_{i=0}^{k} (SySoverhead_{1+(i*p)} + T^u_{s_{1+(i*p)}}) \tag{7.1}$$

Figure 7.8: Best and worst schedulings in a parallel system.



$$T_p^{worst} = Create(p) + \sum_{i=1}^{k} (SySoverhead_i + T_{s_i}^u) \tag{7.2}$$

Assuming that an ideal parallel execution environment has no overheads, we can estimate T_p^l *and* T_p^u as follows:

$$T_p^l = T_s^l / p \tag{7.3}$$

$$T_p^u = Create(p) + \sum_{i=1}^{k} (SySoverhead_i + T_{s_i}^u) \tag{7.4}$$

Table 7.6: Selected executions for real programs using the fuzzy set Quite Greater.

Execution	Optimal	Classical Logic (Greater)		Fuzzy Logic (Quite greater)		Speedup
		Classical		Fuzzy 2		
		PAR_c	SEQ_c	PAR_2	SEQ_2	
qsort(250)	Parallel	0	0	0.6	0.53	1.66
qsort(500)	Parallel	0	0	0.6	0.53	1.74

Table 7.6: (continued).

Execution	Optimal	Classical Logic (Greater)		Fuzzy Logic (Quite greater)		Speedup
		Classical		Fuzzy 2		
		PAR_c	SEQ_c	PAR_2	SEQ_2	
qsort(750)	Parallel	0	0	0.6	0.53	1.74
qsort(1000)	Parallel	0	0	0.6	0.53	1.75
qsort(1250)	Parallel	0	0	0.6	0.53	1.71
substitute(0)	Sequential	0	0	0.53	0.53	1.0
substitute(10)	Sequential	0	0	0.6	0.65	1.0
substitute(20)	Sequential	0	0	0.6	0.59	0.97
substitute(30)	Parallel	0	0	0.6	0.57	1.09
substitute(40)	Parallel	0	0	0.6	0.56	1.22
substitute(50)	Parallel	0	0	0.6	0.56	1.32
substitute(60)	Parallel	0	0	0.6	0.55	1.39
substitute(70)	Parallel	0	0	0.6	0.55	1.47
substitute(80)	Parallel	0	0	0.6	0.55	1.51
substitute(90)	Parallel	0	0	0.6	0.54	1.55
substitute(100)	Parallel	0	0	0.6	0.54	1.59
substitute(110)	Parallel	0	0	0.6	0.54	1.62
substitute(120)	Parallel	0	0	0.6	0.54	1.64
substitute(130)	Parallel	0	0	0.6	0.54	1.66
substitute(140)	Parallel	0	0	0.6	0.54	1.69
substitute(150)	Parallel	0	0	0.6	0.54	1.70
substitute(160)	Parallel	0	0	0.6	0.54	1.72
substitute(170)	Parallel	0	0	0.6	0.54	1.73
substitute(180)	Parallel	0	0	0.6	0.54	1.74
substitute(190)	Parallel	0	0	0.6	0.54	1.75
substitute(200)	Parallel	0	0	0.6	0.54	1.77
fib(1)	Sequential	0	0	0.53	0.53	1.0
fib(2)	Sequential	0	0	0.6	0.59	0.64
fib(3)	Sequential	0	0	0.6	0.56	0.82
fib(4)	Parallel	0	0	0.6	0.53	1.04
fib(5)	Parallel	1	0	0.6	0.52	1.0
fib(6)	Parallel	1	0	0.6	0.51	1.0
fib(7)	Parallel	1	0	0.6	0.51	1.0
fib(8)	Parallel	1	0	0.6	0.51	1.0
fib(9)	Parallel	1	0	0.6	0.5	1.0
fib(10)	Parallel	1	0	0.6	0.5	1.0
fib(11)	Parallel	1	0	0.6	0.5	1.0
fib(12)	Parallel	1	0	0.6	0.5	1.0
fib(13)	Parallel	1	0	0.6	0.5	1.0
fib(14)	Parallel	1	0	0.6	0.5	1.0
fib(15)	Parallel	1	0	0.6	0.5	1.0

Table 7.6: (continued).

Execution	Optimal	Classical Logic (Greater)		Fuzzy Logic (Quite greater)		Speedup
		Classical		Fuzzy 2		
		PAR_c	SEQ_c	PAR_2	SEQ_2	
fib(16)	Parallel	1	0	0.6	0.5	1.0
fib(17)	Parallel	1	0	0.6	0.5	1.0
fib(18)	Parallel	1	0	0.6	0.5	1.0
hanoi(1)	Sequential	0	0	0.53	0.53	1.0
hanoi(2)	Sequential	0	0	0.6	1	1.0
hanoi(3)	Sequential	0	0	0.6	0.9	1.0
hanoi(4)	Sequential	0	0	0.6	0.68	1.0
hanoi(5)	Sequential	0	0	0.6	0.58	0.94
hanoi(6)	Parallel	0	0	0.6	0.53	1.28
hanoi(7)	Parallel	1	0	0.6	0.51	1.0
hanoi(8)	Parallel	1	0	0.6	0.5	1.0
hanoi(9)	Parallel	1	0	0.6	0.5	1.0
hanoi(10)	Parallel	1	0	0.6	0.5	1.0
hanoi(11)	Parallel	1	0	0.6	0.5	1.0
hanoi(12)	Parallel	1	0	0.6	0.5	1.0
hanoi(13)	Parallel	1	0	0.6	0.5	1.0
hanoi(14)	Parallel	1	0	0.6	0.5	1.0

Table 7.7: Selected executions for real programs using the fuzzy set Rather Greater.

Execution	Optimal	Classical Logic (Greater)		Fuzzy Logic (Rather greater)		Speedup
		Classical		Fuzzy 2		
		PAR_c	SEQ_c	PAR_2	SEQ_2	
qsort(250)	Parallel	0	0	0.55	0.51	1.66
qsort(500)	Parallel	0	0	0.55	0.51	1.74
qsort(750)	Parallel	0	0	0.55	0.51	1.74
qsort(1000)	Parallel	0	0	0.55	0.51	1.75
qsort(1250)	Parallel	0	0	0.55	0.51	1.71
substitute(0)	Sequential	0	0	0.51	0.51	1.0
substitute(10)	Sequential	0	0	0.55	0.58	1.0
substitute(20)	Sequential	0	0	0.55	0.55	1.0
substitute(30)	Parallel	0	0	0.55	0.54	1.09
substitute(40)	Parallel	0	0	0.55	0.53	1.22
substitute(50)	Parallel	0	0	0.55	0.53	1.32
substitute(60)	Parallel	0	0	0.55	0.52	1.39
substitute(70)	Parallel	0	0	0.55	0.52	1.47
substitute(80)	Parallel	0	0	0.55	0.52	1.51
substitute(90)	Parallel	0	0	0.55	0.52	1.55

Table 7.7: (continued).

Execution	Optimal	Classical Logic (Greater)		Fuzzy Logic (Rather greater)		Speedup
		Classical		Fuzzy 2		
		PAR_c	SEQ_c	PAR_2	SEQ_2	
substitute(100)	Parallel	0	0	0.55	0.52	1.59
substitute(110)	Parallel	0	0	0.55	0.52	1.62
substitute(120)	Parallel	0	0	0.55	0.52	1.64
substitute(130)	Parallel	0	0	0.55	0.52	1.66
substitute(140)	Parallel	0	0	0.55	0.52	1.69
substitute(150)	Parallel	0	0	0.55	0.52	1.70
substitute(160)	Parallel	0	0	0.55	0.52	1.72
substitute(170)	Parallel	0	0	0.55	0.52	1.73
substitute(180)	Parallel	0	0	0.55	0.52	1.74
substitute(190)	Parallel	0	0	0.55	0.52	1.75
substitute(200)	Parallel	0	0	0.55	0.52	1.76
fib(1)	Sequential	0	0	0.51	0.51	1.0
fib(2)	Sequential	0	0	0.55	0.54	0.64
fib(3)	Sequential	0	0	0.55	0.53	0.81
fib(4)	Parallel	0	0	0.55	0.51	1.04
fib(5)	Parallel	1	0	0.55	0.51	1.0
fib(6)	Parallel	1	0	0.55	0.5	1.0
fib(7)	Parallel	1	0	0.55	0.5	1.0
fib(8)	Parallel	1	0	0.55	0.5	1.0
fib(9)	Parallel	1	0	0.55	0.5	1.0
fib(10)	Parallel	1	0	0.55	0.5	1.0
fib(11)	Parallel	1	0	0.55	0.5	1.0
fib(12)	Parallel	1	0	0.55	0.5	1.0
fib(13)	Parallel	1	0	0.55	0.5	1.0
fib(14)	Parallel	1	0	0.55	0.5	1.0
fib(15)	Parallel	1	0	0.55	0.5	1.0
fib(16)	Parallel	1	0	0.55	0.5	1.0
fib(17)	Parallel	1	0	0.55	0.5	1.0
fib(18)	Parallel	1	0	0.55	0.5	1.0
hanoi(1)	Sequential	0	0	0.51	0.51	1.0
hanoi(2)	Sequential	0	0	0.55	0.93	1.0
hanoi(3)	Sequential	0	0	0.55	0.7	1.0
hanoi(4)	Sequential	0	0	0.55	0.59	1.0
hanoi(5)	Sequential	0	0	0.55	0.54	0.94
hanoi(6)	Parallel	0	0	0.55	0.51	1.28
hanoi(7)	Parallel	1	0	0.55	0.5	1.0
hanoi(8)	Parallel	1	0	0.55	0.5	1.0
hanoi(9)	Parallel	1	0	0.55	0.5	1.0
hanoi(10)	Parallel	1	0	0.55	0.5	1.0

Table 7.7: (continued).

Execution	Optimal	Classical Logic (Greater)		Fuzzy Logic (Rather greater)		Speedup
		Classical		Fuzzy 2		
		PAR_c	SEQ_c	PAR_2	SEQ_2	
hanoi(11)	Parallel	1	0	0.55	0.5	1.0
hanoi(12)	Parallel	1	0	0.55	0.5	1.0
hanoi(13)	Parallel	1	0	0.55	0.5	1.0
hanoi(14)	Parallel	1	0	0.55	0.5	1.0

Conditions:
PAR_c is $T_p^u \leq T_s^l$
SEQ_c is $T_s^u \leq T_p^l$
PAR_2 is $max(gt(T_s^l/T_p^u), gt(T_s^l/T_p^l), gt(T_s^u/T_p^u))$
SEQ_2 is $max(gt(T_p^l/T_s^u), gt(T_p^l/T_s^l), gt(T_p^u/T_s^u))$

Tables 7.6 and 7.7 show the results. First columns show the same information than in Table 7.4 although in this table *Program* refers to the real benchmarks, which are described in Table 7.5. Note that in this case, in order to determine the optimal execution, both sequential and parallel execution times have been measured directly over the platform. The last row shows the *speedup* of our fuzzy approach with respect to the conservative approach: $Speedup = \frac{T_c}{T_f}$, where T_c is the time of the selected execution using the conservative approach and T_f is the time of the selected execution using our fuzzy approach. A value bigger than one of $speedup$ means that the execution selected with our approach is faster than the one selected by the conservative one.

We can distinguish two main set of cases: on one hand *qsort* and *substitute*, and on the other hand *fib* and *hanoi*. In the first set the *upper bound* on the sequential execution time is different from the *lower bound* whereas in the second set, both are equal. This is understandable, since the execution time for the first set of cases not only depends on the length of the input list, but also on the values of its elements. Thus, for a given list length, there may be different execution times, depending on the actual values of the lists with such length. However, in the second set of cases, the execution time only depends on the size (using the integer value metric) of the input argument, and all executions for the same input data size take the same execution time. Our approach improves provides better average case behaviour than the conservative approach in both cases.

Figures 7.9, 7.10, 7.11 and 7.12 show, in detail, how both approaches work in particular cases in a graphical way. Input has the same meaning that in previous Tables (7.6 and 7.7) and execution times are presented in milliseconds. In all the figures the more conservative approach is called *Classic* and represented with a stars line while our approach is called *Fuzzy* and its symbol is a white square.

Figure 7.9: Qsort selected executions.

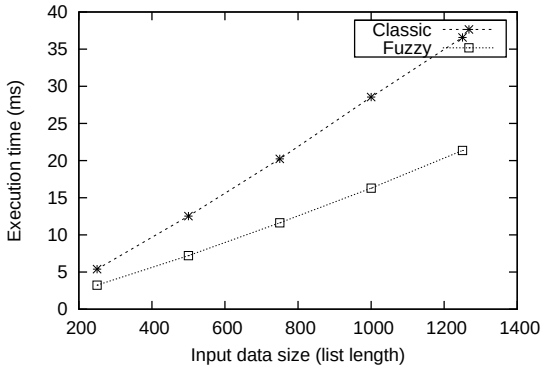
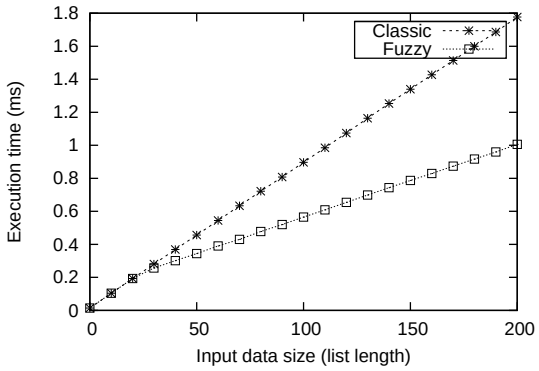


Figure 7.10: Substitute selected executions.



Figures 7.11 and 7.12 show how for *fibonacci* and *hanoi* both approaches have nearly the same behavior for all the tested cases. In fact, at this scale, the scarce cases in which there is a slowdown (see Tables (7.6 and 7.7) can not be appreciated. Figures 7.9 and 7.10 show the behavior for *qsort* and *substitute*. It is clear how the times of the executions selected by our approach are smaller (except in a small number of cases that is insignificant), and how the difference between both approaches becomes bigger when input data sizes increase.

Figure 7.11: Fibonacci selected executions.

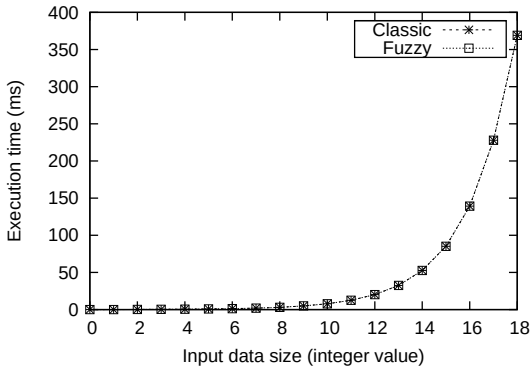
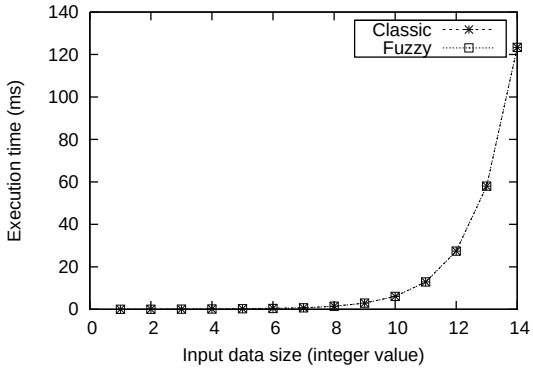


Figure 7.12: Hanoi selected executions.



Chapter 8

Conclusions

We have applied fuzzy logic to the program optimization field, in particular, to automatic granularity control in parallel/distributed computing. We have derived fuzzy conditions for deciding whether to execute some tasks in parallel or sequentially, using information about the cost of tasks and parallel execution overheads. We have developed a profiling tool that can be also applied to the program optimization field.

We have performed an experimental assessment of the fuzzy conditions and identified the ones that have the best average case behavior. We have also compared our proposed fuzzy conditions with existing sufficient (conservative) ones for performing granularity control. Our experiments showed that the proposed fuzzy conditions result in better program optimizations (on average) than the conservative conditions. The conservative approach ensures that execution decisions will never result in a slowdown, but loses some parallelizations opportunities (and thus, no speedup is obtained). In contrast, the fuzzy approach makes a better use of the parallel resources and although fuzzy conditions can produce slowdown for some executions, the whole computation benefits from some speedup on average (always preserving correctness). Of course, the fuzzy approach is applicable in scenarios where the no slowdown property is not needed, as for example video games, text processors, compilers, etc.

Experiments performed with real programs (and real execution times) have demonstrated that our approach can be successfully applied in practice. We intend to perform a more rigorous and broad assessment or our approach, by applying it to large real life programs and using fully automatic tools for estimating execution times.

Although a lot of work still remains to be done, the preliminary results are very encouraging and we believe that it is possible to exploit all the potential offered by multicore systems by applying fuzzy logic to automatic program parallelization techniques.

Bibliography

[1] ASKALON - A Programming Environment and Tool Set for Clusters and Grid Computing. http://www.dps.uibk.ac.at/projects/askalon/parallel/

[2] Paradigm: A Parallelizing Compiler for Distributed Memory Message-Passing Multicomputers. http://www.ece.northwestern.edu/cpdc/Paradigm/Paradigm.html

[3] Parafrase-2 Home Page. http://www.csrd.uiuc.edu/parafrase2/index.html

[4] Pyroos Web Page. http://www.cs.rutgers.edu/pub/gerasoulis/pyrros/

[5] The AspectJ Project. http://www.eclipse.org/aspectj/

[6] Ishfaq Ahmad, Yu-Kwong Kwok, Min-You Wu, and Wei Shu. Automatic Parallelization and Scheduling of Programs on Multiprocessors Using CASCH. In *ICPP '97: Proceedings of the International Conference on Parallel Processing*, pages 288–291, Washington, DC, USA, 1997. IEEE Computer Society.

[7] Shameem Akhter and Jason Roberts. *Multi-Core Programming: Increasing Performance Through Software Multi-threading*. Intel, Santa Clara, CA, 2006.

[8] Abdallah Deeb I. Al Zain, Kevin Hammond, Jost Berthold, Phil Trinder, Greg Michaelson, and Mustafa Aswad. Low-Pain, High-Gain Multicore Programming in Haskell: Coordinating Irregular Symbolic Computations on Multicore Architectures. In *DAMP '09: Proceedings of the 4th Workshop on Declarative Aspects of Multicore Programming*, pages 25–36, New York, NY, USA, 2008. ACM.

[9] Jade Alglave, Anthony Fox, Samin Ishtiaq, Magnus O. Myreen, Susmit Sarkar, Peter Sewell, and Francesco Zappa Nardelli. The Semantics of Power and ARM Multiprocessor Machine Code. In *DAMP '09: Proceedings of the 4th Workshop on Declarative Aspects of Multicore Programming*, pages 13–24, New York, NY, USA, 2008. ACM.

[10] R. Allen, D. Callahan, and K. Kennedy. Automatic Decomposition of Scientific Programs for Parallel Execution. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 63–76, New York, NY, USA, 1987. ACM.

[11] Erik Altman, James Dehnert, Christoph W. Kessler, and Jens Knoop. 05101 Abstracts Collection – Scheduling for Parallel Architectures: Theory, Applications, Challenges. In Erik Altman, James Dehnert, Christoph W. Kessler, and Jens Knoop, editors, *Scheduling for Parallel Architectures: Theory, Applications, Challenges*, number 05101 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2005. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[12] Saisanthosh Balakrishnan, Ravi Rajwar, Mike Upton, and Konrad Lai. The Impact of Performance Asymmetry in Emerging Multicore Architectures. *SIGARCH Comput. Archit. News*, 33(2):506–517, 2005.

[13] Satish Balay, Kris Buschelman, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, and Hong Zhang Barry F. Smith. The PETSc User's Manual.

[14] J. F. Baldwin, T.P. Martin, and B.W. Pilsworth. *Fril: Fuzzy and Evidential Reasoning in Artificial Intelligence*. John Wiley & Sons, 1995.

[15] Prithviraj Banerjee, John A. Chandy, Manish Gupta, Eugene W. Hodges IV, John G. Holm, Antonio Lain, Daniel J. Palermo, Shankar Ramaswamy, and Ernesto Su. The Paradigm Compiler for Distributed-Memory Multicomputers. *Computer*, 28(10):37–47, 1995.

[16] Guy Blelloch. NESL: Revisited.

[17] U. Bondhugula, M. Baskaran, A. Hartono, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Towards Effective Automatic Parallelization for Multicore Systems. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–5, April 2008.

[18] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 101–113, New York, NY, USA, 2008. ACM.

[19] Uday Bondhugula, J. Ramanujam, and P. Sadayappan. PLuTo: A Practical and Fully Automatic Polyhedral Parallelizer and Locality Optimizer. Technical Report OSU-CISRC-10/07-TR70, The Ohio State University, October 2007.

[20] Borys J. Bradel and Tarek S. Abdelrahman. Automatic Trace-Based Parallelization of Java Programs. In *ICPP '07: Proceedings of the 2007 International Conference on Parallel Processing*, page 26, Washington, DC, USA, 2007. IEEE Computer Society.

[21] C.J. Brownhill, A. Nicolau, S. Novack, and C.D. Polychronopoulos. The PROMIS Compiler Prototype. In *Parallel Architectures and Compilation Techniques., 1997. Proceedings., 1997 International Conference on*, pages 116–125, Nov 1997.

[22] F. Bueno, P. López-García, G. Puebla, and M. Hermenegildo. The Ciao Preprocessor. Technical Report CLIP1/06, Technical University of Madrid (UPM), Facultad de Informática, 28660 Boadilla del Monte, Madrid, Spain, January 2006.

[23] L. Byrd. Understanding the Control Flow of Prolog Programs. In S.-A. Tärnlund, editor, *Proceedings of the 1980 Logic Programming Workshop*, pages 127–138, Debrecen, Hungary, July 1980.

[24] M. Carro, L. Gómez, and M. Hermenegildo. Some Paradigms for Visualizing Parallel Execution of Logic Programs. In *1993 International Conference on Logic Programming*, pages 184–201. MIT Press, June 1993.

[25] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon L. Peyton Jones, Gabriele Keller, and Simon Marlow. Data Parallel Haskell: A Status Report. In *DAMP*, pages 10–18, 2007.

[26] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Gabriele Keller. Partial Vectorisation of Haskell Programs, 2008.

[27] Z.S. Chamski and M.F.P. O'Boyle. Practical Loop Generation. In *System Sciences, 1996., Proceedings of the Twenty-Ninth Hawaii International Conference on* ,, volume 1, pages 223–232 vol.1, Jan 1996.

[28] J. Chassin and P. Codognet. Parallel Logic Programming Systems. *Computing Surveys*, 26(3):295–336, September 1994.

[29] Jyh-Herng Chow, Leonard E. Lyon, and Vivek Sarkar. Automatic Parallelization for Symmetric Shared-Memory Multiprocessors. In *CASCON '96: Proceedings of the 1996 Conference of the Centre for Advanced Studies on Collaborative Research*, page 5. IBM Press, 1996.

[30] Florina Monica Ciorba. A Brief Survey of Parallelization and/or Code Generation Software Tools. Available at www.cslab.ntua.gr/ cflorina/research/Brief_survey.doc

[31] Victor Santos Costa. Parallelism in Logic Programs. Available at http://glew.org/damp2006/VitorSantosCosta.pdf

[32] Vitor Santos Costa. On Supporting Parallelism in a Logic Programming System., 2008.

[33] S. K. Debray, N.-W. Lin, and M. Hermenegildo. Task Granularity Analysis in Logic Programs. In *Proc. of the 1990 ACM Conf. on Programming Language Design and Implementation*, pages 174–188. ACM Press, June 1990.

[34] A. Yu. Drozdov. Component Approach for Construction of Optimizing Compilers. *Programming and Computer Software*, 35(5):291–300.

[35] Rob Ennals. Now you C it. Now you don't.

[36] Thomas Fahringer, Alexandru Jugravu, Sabri Pllana, Radu Prodan, Clovis Seragiotto, Jr., and Hong-Linh Truong. ASKALON: A Tool Set for Cluster and Grid Computing: Research Articles. *Concurr. Comput. : Pract. Exper.*, 17(2-4):143–169, 2005.

[37] I. Foster, Ming Xu, and B. Avalani. A Compilation System that Integrates High Performance Fortran and Fortran M. In *Scalable High-Performance Computing Conference, 1994., Proceedings of the*, pages 293–300, May 1994.

[38] Eitan Frachtenberg and Uwe Schwiegelshohn. New Challenges of Parallel Job Scheduling, May 2008.

[39] Vincent W. Freeh, David K. Lowenthal, and Gregory R. Andrews. Distributed Filaments: Efficient Fine-Grain Parallelism on a Cluster of Workstations. In *OSDI '94: Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, page 15, Berkeley, CA, USA, 1994. USENIX Association.

[40] David Geer. Industry Trends: Chip Makers Turn to Multicore Processors. *Computer*, 38(5):11–13, 2005.

[41] Rakesh Ghiya, Laurie J. Hendren, and Yingchun Zhu. Detecting Parallelism in C Programs with Recursive Data Structures. In *CC*, pages 159–173, 1998.

[42] Clemens Grelck and Sven-Bodo Scholz. Efficient Heap Management for Declarative Data Parallel Programming on Multicores, 2008.

[43] Dan Grossma. Design and Implementation Issues for Atomicity and Functional Languages. Available at http://glew.org/damp2006/DanGrossman.pdf

[44] The SUIF Group. The SUIF Compiler System. Available at http://suif.stanford.edu/

[45] S. Guadarrama, S. Muñoz, and C. Vaucheret. Fuzzy Prolog: A new Approach Using Soft Constraints Propagation. *Fuzzy Sets and Systems, FSS*, 144(1):127–150, May 2004. ISSN 0165-0114.

[46] G. Gupta and M. Hermenegildo. ACE: And/Or-parallel Copying-based Execution of Logic Programs. Technical Report without, T.U. of Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, June 1991. also in Proc. ICLP91 Workshop on Parallel Execution of Logic Programs.

[47] G. Gupta, E. Pontelli, K. Ali, M. Carlsson, and M. Hermenegildo. Parallel Execution of Prolog Programs: a Survey. *ACM Transactions on Programming Languages and Systems*, 23(4):472–602, July 2001.

[48] Rajiv Gupta, Santosh Pande, Kleanthis Psarris, and Vivek Sarkar. Compilation Techniques for Parallel Systems. *Parallel Computing*, 25(13-14):1741 – 1783, 1999.

[49] Philipp Haller and Tom Van Cutsem. Implementing Joins using Extensible Pattern Matching, 2008.

[50] B. Hamidzadeh, Lau Ying Kit, and D.J. Lilja. Dynamic Task Scheduling Using Online Optimization. *Parallel and Distributed Systems, IEEE Transactions on*, 11(11):1151–1163, Nov 2000.

[51] Matthew Hammer, Umut A. Acar, Mohan Rajagopalan, and Anwar M. Ghuloum. S Proposal for Parallel Self-Adjusting computation. In *DAMP*, pages 3–9, 2007.

[52] Kevin Hammond. Hume and Multicore Architectures.

[53] Lance Hammond, Basem A. Nayfeh, and Kunle Olukotun. A Single-Chip Multiprocessor. *Computer*, 30(9):79–85, 1997.

[54] Bruno Harbulot and John R. Gurd. A Join Point for Loops in AspectJ. In *AOSD '06: Proceedings of the 5th International Conference on Aspect-Oriented Software Development*, pages 63–74, New York, NY, USA, 2006. ACM.

[55] L. J. Hendren and A. Nicolau. Parallelizing Programs with Recursive Data Structures. *IEEE Trans. Parallel Distrib. Syst.*, 1(1):35–47, 1990.

[56] Stephan Herhut, Sven-Bodo Scholz, and Clemens Grelck. Controlling Chaos: On Safe Side-Effects in Data-Parallel Operations. In *DAMP '09: Proceedings of the 4th Workshop on Declarative Aspects of Multicore Programming*, pages 59–67, New York, NY, USA, 2008. ACM.

[57] M. Hermenegildo, F. Bueno, D. Cabeza, M. Carro, A. Casas, P. López-García, and G. Puebla. Automatic Parallelization of Logic and Constraint Programs. In *DPMC, Intel Workshop on Declarative Programming Languages for Multicore Programming*, January 2006.

[58] M. Hermenegildo, F. Bueno, A. Casas, J. Navas, E. Mera, M. Carro, and P. López-García. Automatic Granularity-Aware Parallelization of Programs with Predicates, Functions, and Constraints. In *DAMP'07, ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming*, January 2007.

[59] M. Hermenegildo and K. Greene. The &-Prolog System: Exploiting Independent And-Parallelism. *New Generation Computing*, 9(3,4):233–257, 1991.

[60] M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2), 2005.

[61] M. Hermenegildo and F. Rossi. On the Correctness and Efficiency of Independent And-Parallelism in Logic Programs. In *Proc. of the 1989 North American Conference on Logic Programming*, pages 369–389. MIT Press.

[62] M. Hermenegildo and F. Rossi. Strict and Non-Strict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions. *Journal of Logic Programming*, 22(1):1–45, 1995.

[63] M. V. Hermenegildo, F. Bueno, M. Carro, P. López, J.F. Morales, and G. Puebla. An Overview of The Ciao Multiparadigm Language and Program Development Environment and its Design Philosophy. In *Festschrift for Ugo Montanari*, number 5065 in LNCS, pages 209–237. Springer-Verlag, June 2008.

[64] M. V. Hermenegildo, F. Bueno, M. Carro, P. López, J.F. Morales, and G. Puebla. An Overview of The Ciao Multiparadigm Language and Program Development Environment and its Design Philosophy. In Pierpaolo Degano, Rocco De Nicola, and Jose Meseguer, editors, *Festschrift for Ugo Montanari*, number 5065 in LNCS, pages 209–237. Springer-Verlag, June 2008.

[65] L. Huelsbergen. Dynamic Language Parallelization. Technical Report 1178, Computer Science Dept. Univ. of Wisconsin, September 1993.

[66] L. Huelsbergen, J. R. Larus, and A. Aiken. Using Run-Time List Sizes to Guide Parallel Thread Creation. In *Proc. ACM Conf. on Lisp and Functional Programming*, June 1994.

[67] Clément Hurlin. Automatic Parallelization and Optimization of Programs by Proof Rewriting. In Jens Palsberg and Zhendong Su, editors, *Static Analysis Symposium 16th International Static Analysis Symposium*, volume 5673 of *LNCS*, pages 52–68, Los Angeles États-Unis d'Amérique, 2009. Springer-Verlag. D.: Software/D.1: PROGRAMMING TECHNIQUES/D.1.3: Concurrent Programming/D.1.3.1: Parallel programming, D.: Software/D.2: SOFTWARE ENGINEERING/D.2.4: Software/Program Verification IST-FET-2005-015905 Mobius project.

[68] Mitsuru Ishizuka and Naoki Kanai. Prolog-ELF incorporating fuzzy logic. In *IJCAI*, pages 701–703, 1985.

[69] S. Kaplan. Algorithmic Complexity of Logic Programs. In *Logic Programming, Proc. Fifth International Conference and Symposium, (Seattle, Washington)*, pages 780–793, 1988.

[70] S.C. Kothari, Jaekyu Cho, Yunbo Deng, S. Mitra, Xindi Bian, R. Leung, S.J. Ghan, and A.J. Bourgeois. Software Tools and Parallel Computing for Numerical Weather Prediction Models. In *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM*, pages 236–243, 2002.

[71] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P Sadayappan. Effective Automatic Parallelization of Stencil Computations. *SIGPLAN Not.*, 42(6):235–244, 2007.

[72] B. Kruatrachue and T. Lewis. Grain Size Determination for Parallel Processing. *IEEE Software*, January 1988.

[73] Milind Kulkarni, Patrick Carribault, Keshav Pingali, Ganesh Ramanarayanan, Bruce Walter, Kavita Bala, and L. Paul Chew. 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, 2008. ACM.

[74] R.C.T. Lee. Fuzzy logic and the resolution principle. *Journal of the Association for Computing Machinery*, 19(1):119–129, 1972.

[75] T. Lewis and H. El-Rewini. Parallax: A Tool for Parallel Program Scheduling. *Parallel & Distributed Technology: Systems & Applications, IEEE*, 1(2):62–72, May 1993.

[76] Deyi Li and Dongbo Liu. *A Fuzzy Prolog Database System*. John Wiley & Sons, New York, 1990.

[77] Sam Lindley. Implementing Deterministic Declarative Concurrency Using Sieves. In *DAMP*, pages 45–49, 2007.

[78] Virginia M. Lo, Sanjay Rajopadhye, Samik Gupta, David Keldsen, Moataz A. Mohamed, Bill Nitzberg, and Xiaoxiong Zhong Jan Arne Telle. OREGAMI: Tools for Mapping Parallel Computations to Parallel Architectures. 20:237–270, 1991. The origional publication is available at springerlink.com (DOI) 10.1007/BF01379319.

[79] P. López-García, M. Hermenegildo, and S. K. Debray. A Methodology for Granularity Based Control of Parallelism in Logic Programs. *Journal of Symbolic Computation, Special Issue on Parallel Symbolic Computation*, 21(4–6):715–734, 1996.

[80] Jan-Willem Maessen. pH: Lessons Learned. Available at http://glew.org/damp2006/pH%20Retrospective.pdf

[81] Gabriele Keller Manuel M. T. Chakravarty. Nested Data Parallelism in Haskell. Available at http://glew.org/damp2006/ndp-mc.pdf

[82] A. B. Matos. A matrix model for the flow of control in prolog programs with applications to profiling. *Software Practice and Experience*, 24(8):729–746, August 1994.

[83] Carolyn McCreary and Helen Gill. Automatic Determination of Grain Size for Efficient Parallel Processing. *Commun. ACM*, 32(9):1073–1078, 1989.

[84] C. McGreary and H. Gill. Automatic Determination of Grain Size for Efficient Parallel Processing. *Communications of the ACM*, 32, 1989.

[85] E. Mera. Estimación de los coeficientes del análisis de complejidad mediante técnicas estadísticas. Technical Report CLIP14/2004.0, Technical University of Madrid, School of Computer Science, UPM, September 2004.

[86] E. Mera, P. López-García, M. Carro, and M. Hermenegildo. Towards Execution Time Estimation in Abstract Machine-Based Languages. In *10th Int'l. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'08)*, pages 174–184. ACM Press, July 2008.

[87] E. Mera, P. López-García, G. Puebla, M. Carro, and M. Hermenegildo. Using Combined Static Analysis and Profiling for Logic Program Execution Time Estimation. In *22nd International Conference on Logic Programming (ICLP'06)*, number 4079 in LNCS, pages 431–432. Springer-Verlag, August 2006.

[88] E. Mera, P. López-García, G. Puebla, M. Carro, and M. Hermenegildo. Combining Static Analysis and Profiling for Estimating Execution Times. In *Ninth International Symposium on Practical Aspects of Declarative Languages (PADL'07)*, number 4354 in LNCS, pages 140–154. Springer-Verlag, January 2007.

[89] E. Mera, T. Trigo, P. López-García, and M. Hermenegildo. An Approach to Profiling for Run-Time Checking of Computational Properties and Performance Debugging. Technical Report CLIP3/2010.0, Technical University of Madrid (UPM), School of Computer Science, UPM, March 2010.

[90] Susana Muñoz-Hernández, Víctor Pablos-Ceruelo, and Hannes Strass. Rfuzzy: An expressive simple fuzzy compiler. In *IWANN (1)*, pages 270–277, 2009.

[91] K. Muthukumar, F. Bueno, M. García de la Banda, and M. Hermenegildo. Automatic Compile-time Parallelization of Logic Programs for Restricted, Goal-level, Independent And-parallelism. *Journal of Logic Programming*, 38(2):165–218, February 1999.

[92] J. Navas, M. Méndez-Lojo, and M. Hermenegildo. Customizable Resource Usage Analysis for Java Bytecode. Technical Report UNM TR-CS-2008-02 - CLIP1/2008.0, University of New Mexico, Department of Computer Science, UNM, January 2008.

[93] J. Navas, M. Méndez-Lojo, and M. Hermenegildo. Safe Upper-bounds Inference of Energy Consumption for Java Bytecode Applications. In *The Sixth NASA Langley Formal Methods Workshop (LFM 08)*, April 2008. Extended Abstract.

[94] J. Navas, M. Méndez-Lojo, and M. Hermenegildo. User-Definable Resource Usage Bounds Analysis for Java Bytecode. In *Proceedings of the Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'09)*, volume 253 of *Electronic Notes in Theoretical Computer Science*, pages 6–86. Elsevier - North Holland, March 2009.

[95] J. Navas, E. Mera, P. López-García, and M. Hermenegildo. User-Definable Resource Bounds Analysis for Logic Programs. In *23rd International Conference on Logic Programming (ICLP'07)*, volume 4670 of *Lecture Notes in Computer Science*. Springer, 2007.

[96] Víctor Pablos-Ceruelo, Susana Muñoz-Hernández, and Hannes Strass. Rfuzzy framework. *Paper presented at the 18th Workshop on Logic-based Methods in Programming Environments (WLPE2008), CoRR*, abs/0903.2188, 2009.

[97] Víctor Pablos-Ceruelo, Hannes Strass, and Susana Muñoz Hernández. Rfuzzy—a framework for multi-adjoint fuzzy logic programming. In *Fuzzy Information Processing Society, 2009. NAFIPS 2009. Annual Meeting of the North American*, pages 1–6, June 2009.

[98] Lu Peng, Jih-Kwon Peir, Tribuvan K. Prakash, Yen-Kuang Chen, and David M. Koppelman. Memory Performance and Scalability of Intel's and AMD's Dual-Core Processors: A Case Study. In *IPCCC*, pages 55–64, 2007.

[99] Constantine D. Polychronopoulos, Miliand B. Gikar, Mohammad R. Haghighat, Chia L. Lee, Bruce P. Leung, and Dale A. Schouten. The Structure of Parafrase-2: An Advanced Parallelizing Compiler for c and fortran. In *Selected Papers of the Second Workshop on Languages and Compilers for Parallel Computing*, pages 423–453, London, UK, UK, 1990. Pitman Publishing.

[100] E. Pontelli, G. Gupta, and M. Hermenegildo. &ACE: A High-Performance Parallel Prolog System. In *International Parallel Processing Symposium*, pages 564–572. IEEE Computer Society Technical Committee on Parallel Processing, IEEE Computer Society, April 1995.

[101] G. Puebla and M. Hermenegildo. Abstract Specialization and its Applications. In *ACM Partial Evaluation and Semantics based Program Manipulation (PEPM'03)*, pages 29–43. ACM Press, June 2003. Invited talk.

[102] S. A. Jarvis R. G. Morgan. Profiling large-scale lazy functional programs. *Journal of Functional Programing*, 8(3):201–237, May 1998.

[103] F. A. Rabhi and G. A. Manson. Using Complexity Functions to Control Parallelism in Functional Programs. Res. Rep. CS-90-1, Dept. of Computer Science, Univ. of Sheffield, England, January 1990.

[104] John Reppy and Yinqi Xiao. Toward a Parallel Implementation of Concurrent ml, 2008.

[105] Behrooz Shirazi, Krishna Kavi, A.R. Hurson, and Prasenjit Biswas. PARSA: A Parallel Program Scheduling and Assessment Environment. In *Parallel Processing, 1993. ICPP 1993. International Conference on*, volume 2, pages 68–72, Aug. 1993.

[106] Wei Shu and Min-You Wu. Runtime Incremental Parallel Scheduling (rips) on Distributed Memory Computers. *Parallel and Distributed Systems, IEEE Transactions on*, 7(6):637–649, June 1996.

[107] Mauricio Solar and Mario Inostroza. An Automatic Scheduler for Parallel Machines (Research Note). In *Euro-Par '02: Proceedings of the 8th International Euro-Par Conference on Parallel Processing*, pages 212–216, London, UK, 2002. Springer-Verlag.

[108] Daniel Spoonhower, Guy E. Blelloch, Robert Harper, and Phillip B. Gibbons. Space Profiling for Parallel Functional Programs. *SIGPLAN Not.*, 43(9):253–264, 2008.

[109] Hannes Strass, Susana Muñoz-Hernández, and Víctor Pablos-Ceruelo. Operational semantics for a fuzzy logic programming system with defaults and constructive answers. In *IFSA/EUSFLAT Conf.*, pages 1827–1832, 2009.

[110] Martin Sulzmann, Edmund S.L. Lam, and Simon Marlow. Comparing the Performance of Concurrent Linked-List Implementations in Haskell. In *DAMP '09: Proceedings of the 4th Workshop on Declarative Aspects of Multicore Programming*, pages 37–46, New York, NY, USA, 2008. ACM.

[111] Florina Ciorba Theodore, Theodore Andronikos, Dimitris Kamenopoulos, Panagiotis Theodoropoulos, and George Papakonstantinou. Simple Code Generation for Special UDLs. In *In 1st Balkan Conference in Informatics (BCI'03*, 2003.

[112] T. Trigo, P. López-García, and S. Mu noz Hernandez. Towards Fuzzy Granularity Control in Parallel/Distributed Computing. In *International Conference on Fuzzy Computation (ICFC 2010)*, pages 43–55. SciTePress, October 2010.

[113] C. Vaucheret, S. Guadarrama, and S. Muñoz. Fuzzy Prolog: A Simple General Implementation using CLP(R). In *9th International Conference on Logic for Programming Artificial Intelligence and Reasoning*, Tbilisi, Georgia, October 2002.

[114] Bo Wang. Task Parallel Scheduling over Multi-core System. In *CloudCom*, pages 423–434, 2009.

[115] M.H. Willebeek-LeMair and A.P. Reeves. Strategies for dynamic load balancing on highly parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 4:979–993, 1993.

[116] K. Windisch, J.V. Miller, and V. Lo. ProcSimity: An Experimental Tool for Processor Allocation and Scheduling in Highly Parallel Systems. In *Frontiers of Massively Parallel Computation, 1995. Proceedings. Frontiers '95., Fifth Symposium on the*, pages 414–421, Feb 1995.

[117] Thomas Wolfgang Burger. Intel Multi-Core Processors: Quick Reference Guide, August 2005.

[118] M. Y. Wu and D. D. Gajski. Hypertool: A Programming Aid for Message-Passing Systems. *IEEE Trans. Parallel Distrib. Syst.*, 1(3):330–343, 1990.

[119] Min-You Wu, Wei Shu, and Yong Chen. Runtime Parallel Incremental Scheduling of DAGs. In *ICPP '00: Proceedings of the Proceedings of the 2000 International Conference on Parallel Processing*, page 541, Washington, DC, USA, 2000. IEEE Computer Society.

[120] Chengzhong Xu and Francis C. Lau. *Load Balancing in Parallel Computers: Theory and Practice*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[121] Tao Yang and Apostolos Gerasoulis. PYRROS: Static Task Scheduling and Code Generation for Message Passing Multiprocessors. In *ICS '92: Proceedings of the 6th International Conference on Supercomputing*, pages 428–437, New York, NY, USA, 1992. ACM.

[122] Steve Zdancewic. Application-Level Concurrency: Combining Events and Treads: invited talk. In *DAMP*, page 2, 2007.

[123] X. Zhong, E. Tick, S. Duvvuru, L. Hansen, A.V.S. Sastry, and R. Sundararajan. Towards an Efficient Compile-Time Granularity Analysis Algorithm. In *Proc. of the 1992 International Conference on Fifth Generation Computer Systems*, pages 809–816. Institute for New Generation Computer Technology (ICOT), June 1992.

[124] Lukasz Ziarek and Suresh Jagannathan. Memoizing Multi-Threaded Transactions, 2008.

[125] Lukasz Ziarek, Suresh Jagannathan, Matthew Fluet, and Umut A. Acar. Speculative n-way barriers. In *DAMP '09: Proceedings of the 4th Workshop on Declarative Aspects of Multicore Programming*, pages 1–12, New York, NY, USA, 2008. ACM.