ÉCOLE
SUPÉRIEURE
D'ÉLECTRICITÉ

UNIVERSITATEA
POLITEHNICA
BUCURESTI

UPB

PARALELLIZATION AND GRID INTEGRATION OF A MOBILE ROBOTICS APPLICATION
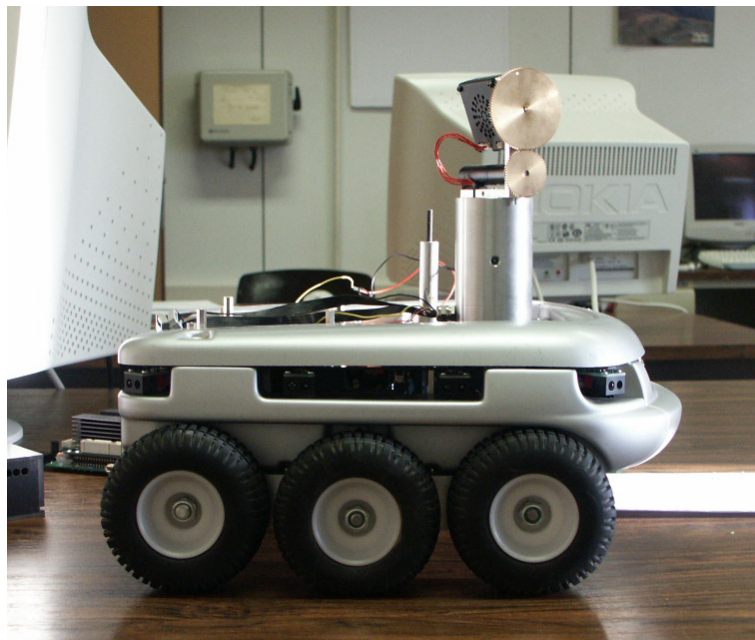
Author:

Alexandru IOSUP

Supervisors:

SUPÉLEC   Ph.D. Prof. Eng. **Stéphane VIALLE**

U.P.B.   Ph.D. Prof. Eng. **Nicolae ȚĂPUȘ**

2003

U.P.B.

Universitatea Politehnica Bucureşti

SUPÉLEC

École Supérieure d'Electricité

Parallelization and Grid Integration of a Mobile Robotics Application

Alexandru Iosup

Supervisors:

Supélec   Stéphane Vialle
Ph.D. Prof. Eng.

U.P.B.   Nicolae Ţăpuş
Ph.D. Prof. Eng.

Abstract

Current trends in mobile robotics involve many small yet surprisingly computing-intensive tasks combined in one complex application. While the use of just one personal computer is insufficient in terms of computing power and reliability, the small dimensionality of each of the application's tasks does not justify supercomputers use. In addition, as more and more robotics laboratories across the world meet the urging need of sharing the robotic resources between them, remote computing becomes increasingly important, with problems such as communication slowdown arising.

This work addresses these challenges in the context of a simple mobile robotics application, with plan-based navigation and landmarks based self-localization.

- We present our solution for parallelizing and migrating a module of this application on local and remote computers, based on target machine's architecture.

- We present and discuss comprehensive experimental tests.

- We discuss the possibility of Grid integration for mobile robotics applications.

The main achievements are the use of parallel approach for a module of our application, the proof that common-off-the-shelf computers of today can be used to speedup mobile robotic applications and the promising theoretical aspects of Grid integration of such an application.

Keywords computerised control; parallel mobile robot control; distributed mobile robot control; optimized mobile robotics algorithms; mobile robotics application; dubins trajectories; psimilar landmarks self-localization.

For Ana Lucia,
with all my love.

Chapter i

Acknowledgements

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this thesis mobile robotics challenges are investigated from the perspective of parallel computing, with some degrees of remote and Grid computing. This chapter discusses the relevance of such an investigation and introduces a problem statement.

1.1 Motivations

Traditionally, robots have been used in simple, static environments, with the sole purpose of performing repetitive tasks which require precision over a large amount of time. As the operating environment and the task challenges became more and more complicated, control requirements increased significantly. Nowadays, robotic computing often involves many small and medium-sized tasks which, added in a fairly complex application, may lead to a large processing power requirement [66]. This amount of processing ability is generally not available on the robot onboard processor, because of energy, volume and weight constraints that these embedded processors must satisfy [47]. In addition, most industrial robot controllers used today are very simple uniprocessor systems [38]. Therefore, it is very difficult for these serial controllers to produce real-time responses, making them impractical for mobile robotics [60, 45]. Solutions lie between having special architectures handle special tasks [41] and having the onboard processor execute just simple control routines in order to produce real-time responses, while other methods, like remote processing, are used to ensure that more computationally expensive tasks are being successfully solved [13, 96, 63].

When the robotic system targets a common use, flexibility should be considered alongside the more industry important efficiency [82]. The results will be in the first phase more important for the research community, but in the long run the industry will most benefit from this approach. Therefore, not focusing on dedicated devices becomes of great importance, as the computational architectures found in the industry often favor just the performance factor and, for time and cost reasons, tend to lack adequate developing environments.

Having local machines directly communicating with the robot may be insufficient in the following situations:

1. Special problems involving complex robot behaviors need a very powerful processing resource and cannot run on a simple PC. This is the case of intensive image-processing for vision-based systems or planning for task completion [12].

2. Because of reliability issues, having just one computer able to solve a particular problem may become a problem [45]. The computer may break, burn, be accidentally plugged-off or suffer many other kinds of mechanical or electrical damage, with the obvious outcome of the system lacking the vital computing element.

3. When the robotic resources are not in direct reach, such as the case of a research laboratory sharing robotic resources with several partners [19, 59], without giving them access to the in-place computing power, or the case of robots performing risky actions, like demining a zone [100], or the case of teleoperated robots [93, 75].

When dealing with complex robot behaviour (point 1), one PC turns out to be not enough. This points out to a distributed or parallel approach to the mobile robotics, with some observations. First, the small dimensionality of the mobile robotic tasks makes the use of supercomputers inefficient, if not impossible [98]. Second, the prices of a common off-the-shelf multiprocessor system are constantly decreasing, which makes more and more users renounce at the old buying only sequential processors policy. Third, with the advantages of cluster computing more at hand, the high performance computing needs more directions to evolve [7], and mobile robotics seems to be very promising into becoming one.

As most robots work in a closed and secure environment, testbeds for assessing the robustness to failures of their control programs are hard to conceive and execute. From a software point of view, simulations are good, but cannot conceivably cover all situations. In situations where one error may be fatal, such as surgery, human operators are called to assist or even perform manual control. Having reliable computing sources is in this case of great interest, but remains to be seen how to do it efficiently. We are interested in the effect of having more than one resource performing the same task, without looking at wasting computing units, as we consider the worldwide computing infrastructure as a generous provider (point 2).

In today's world, with the incredible opportunities of high-speed network connections, having to actually send human observers to a certain place is more of an unnecessary luxury (point 3). We are tempted to try to evaluate the difficulties of running an interactive robotic application across the Internet.

1.2   Project framework and goals

All the considerations in the previous chapter made us consider the general case of a simple mobile robotics application, viewed as a hard real-time control problem. We deal with a reliable navigation task controlled remotely, and try to detect and solve the difficulties. The navigation task has three parts: the path planner, the path executor and the self-localization, each performed after the previous one has ended.

Problem statement   The main problem is to control the mobile robot accurately and as fast as possible, through the use of common off-the-shelf computers, with communication performed over various networks. The architectural impact on application's speed, reliability and cost-effectiveness is to be established.

1.3   Technical difficulties

There are many types of technical difficulties encountered during this project. We present here the ones that have been overcome, grouped by their nature into research and engineering problems.

Research tasks

- Establishing if global speedup for mobile robotics applications is achievable, being given that the tasks involved have a very small dimensionality;

- Proving that optimizing just some more intensively used tasks, called critical tasks, leads to speedup;

- Establishing the best way to exploit the natural parallelism of mobile robotic applications for common off-the-shelf computers with or without parallel computing capabilities, if speedup is achievable;

- Establishing the relationship between machine architecture and the algorithms used to obtain speedup;

- Establishing the parameters for parallel performance evaluation;

- Exploring remote control for mobile robotics benefits and drawback;

- Predicting the way a mobile robotics application may integrate into a Grid environment.

Engineering tasks

- Building a testbed for a mobile robotic application, with profiling and results processing tools for the software part;

- Assembling the existing software modules into a single mobile robotics application;

- Writing and testing the parallel versions of the applications modules;

- Thoroughly testing the mobile robotics application.

- Obtaining all the needed values for parallel performance evaluations;

- Obtaining the base values for (future) remote performance evaluations;

- Establishing the impact of improving the camera mechanical parts to a certain performance factor;

- Establishing the impact of using image compression techniques for acquired images;

- Finding the drawbacks of specific third-party libraries used in this project, such as *MPICH* or *nono*.

1.4 Executive summary

This section presents the reader the contents of this thesis. So, without further ado, this report is structured as follows:

Chapter 1 contains a brief introduction of the project's *raison d'etre*, framework, and goals. it also contains the executive summery of this work, which is currently under the reader's eyes.

Chapters 2 and **3** aim to present some facts required for the complete understanding of this work, such as a brief introduction in mobile robotics and a short course on parallel computing. Projects dealing with similar concepts are also presented and briefly discussed.

Chapter 4 assumes the role of introducing the (now advised) user into the details of this project, by presenting the essential and expanded information on project's general framework, and on application's hardware and software components. A brief history, a discussion upon how the goals may be fulfilled being given the mechanical limits and a short summary are in this chapter, too.

Chapter 5 presents the solutions attempted for establishing the best sequential solution for our problem. The sequential solution at the very beginning of this chapter provided important information about the parameters involved into evaluating system's performance, as well as serving as a base for the overlapped solution presented later on on this chapter. Another version, based on hyper-threading, has been assessed for these architectures. Performance of the two latter versions are discussed thoroughly, while the tables of their complete results are also available at the end of this chapter.

Chapter 6 intends to discuss our robot control algorithms for parallel architectures. An optimized multi-threaded approach for shared-memory machines and an optimized message-passing approach for distributed-memory machines are under investigation.

Chapter 7 presents the results for the algorithms discussed in chapter 6. Their evaluation contains not only the complete results tables, but also predictions about future mechanical improvements impact upon application performance (again, with thorough testing). Issues like speedup and parallelizability of the system are explored. Some (mis-)features of the MPICH Linux implementation and their impact on our application are presented. Finally, costs for similar mobile robotics applications are evaluated.

In **chapter 8** we show the base of remote control for mobile robotics experiments that are going to be performed as this project continues. Thorough results for the LAN and some results for the Internet communication are presented discussed, while expected results for the ATM connection between Metz and Nancy are also used to provide early conclusions on the remote control approach.

Chapter 9 tries to evaluate the Grid technology availability for mobile robotics applications, in order to establish future steps for our project.

The conclusions for this project are drawn in **chapter 10**, while **chapter 11** shows the foreseen directions for this project, split by the expected duration into three categories: short term goals, expected to end in maximum one year, medium term goals, expected to take longer than one year, but not more than three, and long term goals, for tasks that will surely take longer than the previous ones, as they rely heavily upon the short and, especially, medium term goals' results.

Finally, an enumeration of the bibliographic sources is displayed in the **bibliography** chapter, while the **appendices** present other useful information with regard to this project.

Summary   Please note that, while the chapters are basically independent one from each other, chapters 2 and 3 are a must if the reader is a complete novice into the field of mobile robotics or parallel computation, chapter 4 is essential to the understanding of this projects' features and goals, chapters 5 to 9 present all the necessary facts about the fulfillment of the current and short term project goals, chapter 10 evaluates the outcomes of this projects, and chapter 11 provides a glimpse into the future of this project.

Chapter 2

Issues on mobile robotics

This section will try to clarify our view on what is a robot, what is robotics and what is the meaning of mobile robotics. In order to achieve that, definitions from industrial standards associations and academic research groups will be presented and briefly commented. Then a summary of the most common tasks that a mobile robot may execute and, finally, a list of complete robotic application that may be called the *state of the art* in mobile robotics are going to be named.

2.1 Industrial and Academic Definitions

Webster's dictionary defines the robot as *a machine that looks like a human being and performs various complex acts (as walking or talking) of a human being*, or *an automatic device that performs functions normally ascribed to humans or a machine in the form of a human*, or, simply, *a mechanism guided by automatic controls*.

Deciding what a robot is can be pretty difficult, as one would be forced to admit that an automatic camera or an automatic car may easily qualify as robots according to the above definitions. The expansion of robotic systems around the mid-80s forced the industry to take some actions into the direction of uniquely defining what a robot is.

Sosoka, 1985 The Japan definition of a robot may be called at best obscure:

> An *industrial robot* is any device that replaces human labor.

Note that an automatic sawing machine or an automatic gear-shifting box qualify as robots under this definition, which is why at a certain point Japan had the officially registered number of industrial robots higher than the rest of the world together [38].

RIA, 1985 The Robotics Industry Association, USA, issued what is currently the industry standard for industrial robots, outside Europe:

> An *industrial robot* is a multifunctional, reprogrammable manipulator that can move materials, pieces, tools and special devices, following variable programmed trajectories, to realize different tasks.

Multifunctional means here that the robot may be used to perform more than one type of task, while reprogrammable means that a robot may change the functioning program and still be called a robot. The problem with this definition is that it depends on deciding on what a manipulator is or does, and does not draw a precise separation line between industrial robots and other types of manipulators.

AFNOR The French Standard Association made the distinction between robots and manipulators more clear. First, the definition of manipulators states that

> A *manipulator* is formed, in general, from a series of connected elements which can pickup and move objects or tools. It is multifunctional and can be controlled directly or via a program.

Industrial robots are then defined as follows

> A *robot* is a manipulator which is reprogrammable, servocontrolled, and versatile, that can position and orientate pieces and tools and other devices, following variable trajectories.

We can see that in the French definition what differentiates the industrial robot from a simple manipulator is the servocontrol. This term designates a negative feedback control, normally of joint positions or speeds.

Our acception of a robot We consider that robots are best described by the French definition, with the addition of intelligence conditions, as we consider that robots should seamlessly interact with the environment they are placed into by means of sensing and (re)acting systems.

Mobile robots For this work, the conditions for mobile robotics are that the entire robotic system is mobile, i.e. it moves with regard to the environment, is at least semi-autonomous, i.e. it needs at most a limited human interaction, and is intelligent, i.e. it senses and reacts to its environment.

2.2 The beginnings of robotics

This section is my tribute to the guys that made this work come to life. They are the pioneers of robots and robotics, writers and scientists alike.

It all begun with the Mary Shelley's Frankenstein, back in 1818. The myth of creation, this time under human control, came into the peoples' minds and never went away.

It took about 100 years until, in 1920, Karel Capek (figure 2.1, a) coined the term robot in his play, *Rossum Universal Robots*, or, in short, `RUR` (figure 2.1, b). `RUR` was translated in English in 1921 and the term `robot`, which comes from the word *robota*, or work, in Czech, or *robotcik*, the slave work each serf had to perform two or three days a week on the master's land, became a common word in every language. The play presented the story of humans creating workers, called robots, which are then forced to labor as slaves. The robots (which's birth is more closely related to genetics than to what we call today robotics) become conscious

of their situation and the revolution ignites, resulting in all humans except one being killed by the robots. Now it is sure that Capek did not intend his play to be remembered as *about robots* and tried get rid of his creation by claiming that his brother invented the term in 1919, but he will always be remembered as the father of robots.
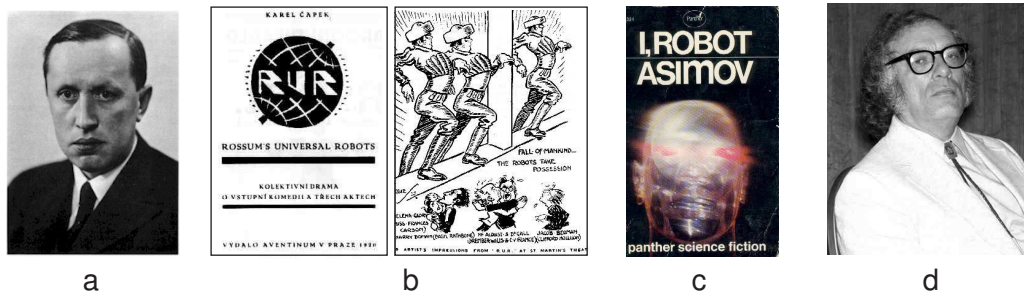
Figure 2.1: *(a)* Karel Capek; *(b)* RUR play posters; *(c) I, the robot* by Isaac Asimov; *(d)* Isaac Asimov.

After 30 years, in 1940, it was another writer's turn to came on to the stage. This writer's name was none other than Isaac Asimov (figure 2.1, d), and his first touch on the robots subject was a short story called *Robbie*, published in 1940. In 1950 he invented the term *robotics*, which defines the science of designing, creating and operating robots, and used it in *I, the robot*, probably the best selling book about robots ever and the birth place of Asimov's famous *laws of robotics* (figure 2.1, c). It was the starting point for a growing industry.

In 1947, the first servo-controlled electric-powered teleoperator was developed. But the robot age trully begun in 1954, with the first manipulator with play-back memory created by George Devol. In 1956, George Devol and an associate, Joseph Engelberger started the first robots company. In 1961, their first industrial robot, `UNIMATE`, was coming into use at General Motors (figure 2.2, b). Running a program from a magnetic play-back memory, this robot sequenced and stacked hot-pieces of die-cast metal for TV picture tubes. This is the end of the pioneering period. Robots in the form of programmed manipulators begun the robot industrial revolution. In 1960, Shakey is created at Standford Researfch Institute (figure 2.2, a). It contained a television camera and was controlled from a computer situated in another room.

Other issues started to interest the researchers, such as humanoid robots walking around specially conceived environments. In 1973, Wabot-1, built by Ichira Kato of the Waseda's Tokyo University, became the first attempt to build such a robot (figure 2.2, c). This robot was able to communicate with a person in Japanese, walk around using his human-like limbs and transport simple objects with its hands. It is estimated that Wabot-1 had the intelligence of a one and a half year-old human being.

Today, almost 200 years later, the myth of creation is (painfully slowly) starting to become reality. Personally, I found and continue to find working in this area fascinating.
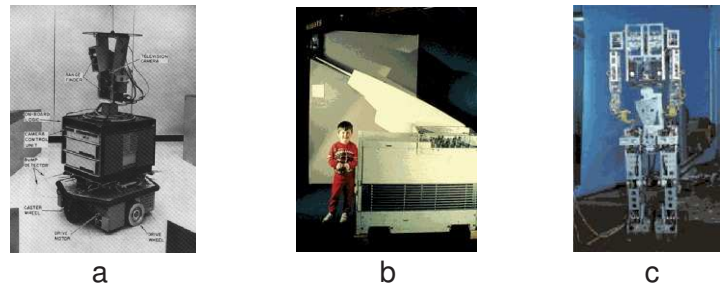
Figure 2.2: *(a)* Shakey, 1960; *(b)* UNIMATE, 1961; *(c)* Wabot-1, 1973.

2.3   Mobile robots tasks

Ideally, robots will be able to handle any kind of objects with incredible precision and with constant performance. However, watching today's results is a bit disappointing [55]. 2D and 3D object recognition, stereoptic mapping of navigation environments and many more failed to pass from laboratory promises to industrial world. This does not mean that robots of today are not capable of performing many tasks, which, combined, may yield into impressive applications, but that robustness of these performances is not their high point. This is the main problem that prevents the industry into accepting the robots [77].

Nowadays, robots are capable of successfully perform small-sized locomotion, use dead-reckoning sensors to estimate position and tactile, proximity, range, sound and smell sensors to detect certain environmental features or commands.

Navigation   is about to be solved through the use of sensors, landmarks, discrete map representation and planning. The computing power required for such tasks seldom takes more than a few computing cycles on any kind of processor and is usually handled by the slowest processing unit available on-board.

Perception   is currently in a curious state. While the information about the systems actually composing the robot tend to be accurate, this is not at all the case for the environment. Establishing relations between the sensors output and the real-world event that enforced the output is for now impossible. The computing power required to receive the input is high enough to make the on-board treatment hardly possible on small robots. Therefore, another processor, placed either on-board or remotely accessed deals with using the available information. Especially in the case of having multiple sensing systems and having to corroborate their results (sensors fusion), a second processor which may act in parallel with the acquisition system is required.

Vision   is is no better shape than perception. 3D object detection is far from being solved, and not only because different sensors detect or not different features of objects. So far some progress has been made into navigating in known and highly stable environments. Interesting exceptions are robots walking at very slow paces, such as museum tour-guides.

When dealing with image treatment, which is pretty much straightforward and thus embarrassingly parallel, having more than one processing unit is a *de facto* standard.

Obstacle avoidance is being performed in outdoor as well as in indoor places, while the available space is not too tight or too crowded. This activity is very simple and requires very few computing power, but heavily relies on the perception and vision components of a robotic system, which means that it will certainly require extra computing power.

Interaction with the environment, in the form of object manipulation, is at its beginnings. Pushing and kicking abilities have been developed for mobile robots, such as the cases of RoboCup football players or sewer pipes cleaners. More complex forms of manipulation involve some sort of advanced planning system - this problem is often called din the literature *the piano mover's problem*. Man-machine interaction is suitable for coordinating a machine into performing the small number of tasks a robot could now possibly accomplish. Manipulation is well developed, allowing for direct or tele- operation. Teaching a robot by first showing the right moves is a well developed feature and is similar to assembly line tuning. The machine-machine interaction is still in its infancy, but some progress has been reported in having teams of robots coordinating actions to achieve a common goal. Interaction is one of the heavy issues in imposing robotics into the industry. It is not a computing power hungry action by itself, but it heavily relies on other systems, like comprehension, natural language processing and some other reasoning, knowledge and planning systems which are very computationally demanding (especially planning in compelx environments for complex actions, that might change the environment).

Summary In conclusion, the robotic tasks problems that are currently robustly solved need more computing power to be fully productive. We consider that productivity is going to be achieved when several task-solving systems are going to be coupled into a complex system, capable of successfully pursuing a certain complex goal, without human intervention except for the issuing of the appropriate goal-defining command. For this, it is obvious that more computing power is required and, as robots on-board space is very expensive, resorting to remote computing power sources is very important.

2.4 Mobile robots control

Control is a useful when we need to be sure that the outcome of a given action is always the same, despite any perturbations or disturbances that affect the action or the subject of the action. With a more formal definition,

> Control is a process that causes a system to remain at or be near of a certain state, which is called reference state. The system's state is usually expressed as a finite set of variables.

A **controller** is, according to Alistair MacFarlane,

> an information-processing device which manipulates the flow of energy, material, or other resource in an associated physical system in such a way as to make the overall system function in some specified manner in the face of arbitrary changes in the physical system itself.

Depending on how the control routine is being executed, there are two types of control: open- and closed- loop control.

Open-loop control   In open loop control, the sequence of commands in the control routine is carried out irrespective of the consequences. No response is expected of the controlled system, nor any observations are being made to ensure that the expected outcome is obtained. This method of control is useful only for very stable systems, which encounter the same type of minor perturbations during their running cycle.

Closed-loop control   In open loop control, the sequence of commands in the control routine is carried out according to the original routine and the immediate consequences, which are observed and sent back into the control system. This process is also called feedback.

We based our research on closed-loop control for mobile robots, as these systems are proven to be highly unstable when mechanically involved in non-stop actions, as it is the case for our testing sessions.

2.5   Applications for today's mobile robotics

The ultimate goal of an autonomous robotic system is, of course, to perform some useful function in place of its human counterpart. But such systems are currently unable to do no more than following paths, maybe some simple manipulating and interaction with the environment, and planning and reasoning for very simple actions. However, these actions are enough to perform some useful tasks. In the following several activity classes are presented, each with industrial examples:
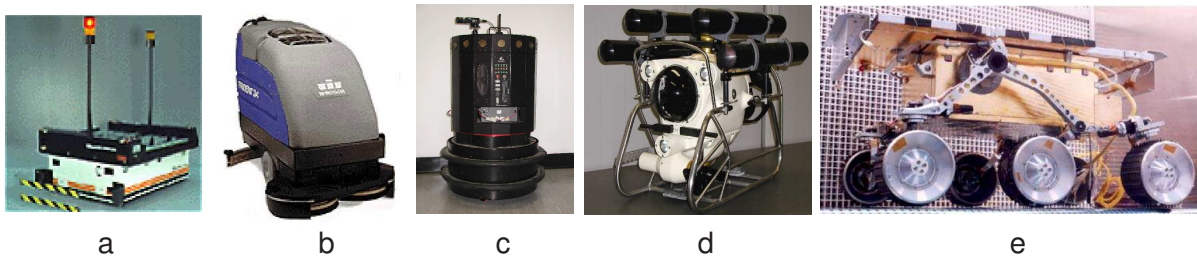
Figure 2.3: *(a)* Rapistan-Demag AGVs; *(b)* Windsor Industries PowerTec 26; *(c)* Nomadic Technologies, Nomad 200; *(d)* Autonomous underwater vehicle; *(e)* NASA Sojourner, Mars conqueror.

Automated Guided Vehicles (AGVs)   This class includes mostly carriers, with the sole purpose of industrial material handling in factories, shipping areas and warehouses. An example is Rapistan built by Demag (figure 2.3, a).
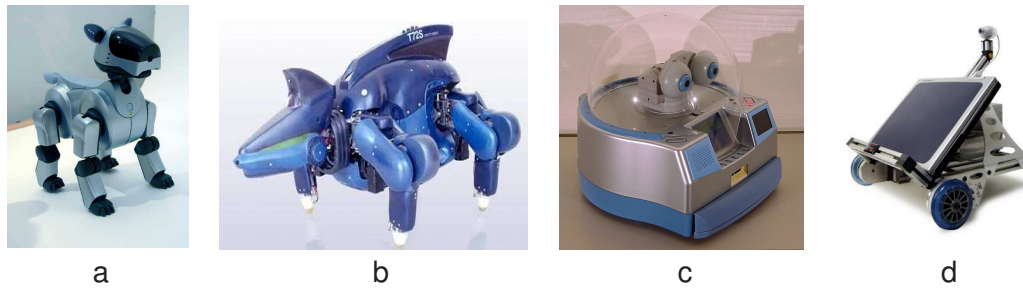
Figure 2.4: *(a)* Sony Aibo; *(b)* Sanyo Banryu; *(c)* Fujitsu Maron; *(d)* Evolution Robotics ER1.

Service robots   This class includes autonomous and semi-autonomous vehicle-like robots, used for simple tasks in human environments. Service tasks include light material handling, like mail delivery, surveillance, and cleaning of large and tight areas. Robots cleaning sewing systems are found in this class, as well as ones cleaning airplanes or warehouses floors, like PowerTec 26 (figure 2.3, b).

Rovers   This class includes robots whose sole purpose is to move across remote or high-risk areas and collect data in the form of pictures, physical samples or some other. Most of these rovers are just in a laboratory phase, but there are some which have been currently been deployed in real missions.

   Land rovers   This subclass includes all forms of planetary roves, demining rovers and land scouts. Probably the most famous robot in the recent years is the Sojourner which was used to explore a small part of Mars' surface in July 1997 (figure 2.3, e). The demining rover project at CMU, SBP2, is also a good example of such systems.

   Aerial rovers   This subclass contains mostly aerial scouts, with results into fire-fighting and landscape mapping and surveillance.

   Aquatic rovers   This subclass is represented by scouting robots in water researchers service (figure 2.3, d). They are used to collect data where the hard working conditions make human intervention too risky and / or expensive.

Research platforms   This class is represented by all kinds of research projects mostly, but not restricted to, on the industrial areas of interest. Nomad (figure 2.4, c)is the best-known representative of such a platform, while the Koala and Kephera systems are also well accepted by the researchers. There is also a multitude of custom robots build by big laboratories such as MIT, or CMU.

Robotoys   This niche in the robot's market is represented by toy robots, of which Sony Aibo, Sanyo Banryu, Fujitsu Maron and Evolution Robotics ER1 (figure 2.4, a-d) are just a few

examples. Robots in this class may be used as simple entertainment or even be programmed to perform some simple tasks.

Military robots This class contains several known and more unknown robots that are used for military missions. From demining robots used during the Gulf War in 1991 to the transmission repeaters mounted on scouts, these robots tend to drive the state of the art further and further away, as they are backed up by real necessities and, more important, real money.

Telesurgery This class is of great importance, as it contains all robots involved in assisting surgeons when operating on patients. Spectacular successes have been achieved, culminating with the operation of a French patient by an US surgeon through the use of a robot operated over the Internet.

Chapter 3

Introduction to parallel computing

This chapter presents a view on parallelism theory and parallel computing for mobile robotics applications. It was not in our intention to cover all possible topics, but those who, in our opinion, introduce the reader to the scope of this report.

3.1 Principles of parallel computing

Parallel programming is the programming paradigm that allows the use of (several machines or a machine with several processors or any combination of these two) to solve a problem. Bigger problems are usually tackled because the multicomputers provide more computing power/resources. The speed at which these problems are solved should be higher than in the purely sequential case, but this remains a result of machine architectures and programming skills only.

3.2 Benefits of parallel computing

Theoretical speed   Parallel computing is required in applications both from research as well as from industry fields. Indeed, numerical computation in maths, complex systems simulation in physics[99], image processing and scene rendering in low-level and 3D graphics, Monte Carlo techniques [65], simulation systems in military applications [76], all have a great need of reducing their computing time [35]. This is done by optimizing algorithms and by taking account of the computer architectures. Parallel computers offer a natural architecture for many such problems.

Natural programming approach   Problems such as application design and implementation can be better tackled though the use of the parallel or distributed programming paradigms. According to Bal et al. [8], the parallel approach is suitable problems displaying inherent parallelism, e.g. a robot waving both his arms at the same time. Using the parallel programming paradigm such applications can be more easily designed, while applications that rely heavily on data communication gain in performance, through the use of a distributed approach, when running on multiple machines connected through a network.

Industrial solutions already tested   Since the area of parallel computing has matured, lots of optimized algorithms have been implemented and studied, allowing for practical off-

the-shelf solutions, both hardware and software, for particular problems [26]. In [36], Fox enumerates five types of problems and their best parallel implementation approach:

1. Synchronous problems, where tightly coupled hardware and software need to exploit the problem structure to obtain good performance, because the data elements are essentially identical;

2. Loosely synchronous problems, like the synchronous problems, but with the data elements are not identical, or their linkage differs;

3. Asynchronous problems, where data parallelism is irregular in space and time, rendering this kind of problems to difficult parallelization;

4. Embarassingly parallel problems, where data elements processing is essentially independent;

5. Metaproblems, where the application consists of collections of synchronous modules, linked asynchronously.

3.3 Parallel computing architectures

When dealing with parallelizing a specific application, one must adapt to the characteristics of the available machine(s), as given by a proper study of the application requirements and project goals. A bad choice may lead to in fact worse performance of the parallel application when compared to its sequential version, both running on the same parallel machine. For our application, performance is the primer factor.

3.3.1 Flynn taxonomy

When choosing a specific computer architecture a head-to-head machines comparison is required. It is also desirable to do this without having to compare detailed data sheets for each computer. For this purpose computers can be classified based on just a few of their characteristics. The process of classification is called 'taxonomy'.

Currently, the most popular nomenclature for the classification of computer architectures is that proposed by Flynn in 1966 [28]. Flynn does not examine the explicit structure of the machines, but instructions and data flows. Specifically, the taxonomy identifies whether there are single or multiple 'streams' for data and for instructions. The term 'stream' refers to a sequence of either instructions or data operated on by the computer. The four possible machines categories, according to the Flynn's taxonomy are:

SISD (Single Instruction Single Data) machines, with *one instruction* executed on *one data* at a certain moment of time (see 3.1, a). All sequential machines belong to this category; however, as today's processors are pipelined or superscalar, the definition was converted to machines in which one operation is executed on one data per state transition [29].

SIMD (Single Instruction Multiple Data) machines, with *one instruction* executed on *several data* at a certain moment of time (see 3.1, b). The typical processors in this class are the vectorial processors.
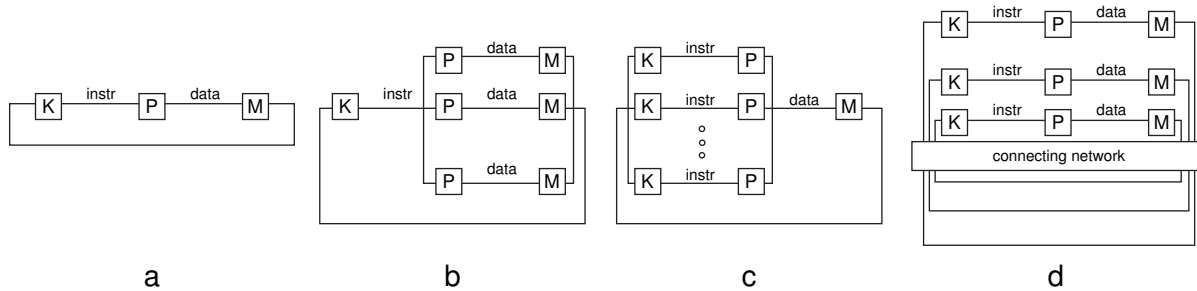
Figure 3.1: *(a)* SIMD: the command unit K fetches one instruction (lock-step) and feeds it to every processing unit P, which uses it on data from the memory unit M; *(b)* SISD: the command unit K fetches one instruction and feeds with it the processing unit P, which uses it on data from the memory unit M.; *(c)* MISD: the command units K fetch one instruction and feed it to their processing units P, which use them on the same data from the memory unit M.; *(d)* MIMD: each command unit K fetches one instruction and feeds it its processing unit P, which uses it on data from its memory unit M. Note the connection network between each subsystem of K, P, M.

MISD (Multiple Instruction Single Data) machines, with *several instruction* executed on *one data* at a certain moment of time (see 3.1, c). This can be done in two ways: a single data stream fed up at the same time to several processing elements, each executing a different sequence of instructions, or having each processing element execute its instruction on the input data, then passing the results to another processing element and restarting its task on a different data, with the obvious analogy to a pipeline architecture. However the machines in this category are often upgraded to become more versatile MIMDs, thus the lack of real MISD machines implementations (one example is the C.mmp built by Carnegie-Mellon University - a reconfigurable machine able to run also as a MISD).

MIMD (Multiple Instruction Multiple Data) machines, with *several instruction* executed on *several data* at a certain moment of time (see 3.1, d). The processors work on their own data with their own instructions. Tasks executed by different processors can start or finish at different times. They are not lock-stepped, as in SIMD computers, but run asynchronously. As will be seen in the following section, most multiprocessor systems and multiple computer systems can be placed in this category.

From the four types described above, MIMD machines are the best suited for our type of application - a metaproblem [36] -. SISD would normally serve as performance reference, while SIMD and MISD are not well-suited, first because the application granularity is too big for lock-stepped implementation, the latter because the data does not require multiple processing (at least not in this stage, although a parallel processing of images for intruder detection as well as for self-localization can be forseen) and because such an architecture cannot be easily found on the market.

3.3.2 MIMD machines

Pros and Cons According to Flynn, MIMD machines have been the computing paradigm of the '90s [29]. Of course, discussions over the pros and cons of such machines may become passionate, as the MIMD machines are still in their infancy and still have lots of problems to solve. One problem that really stands out is the Moore's law outcome for common processors, which is that SISD and SIMD machines increase their processing capacity up to one order of magnitude, i.e. 10 times, every 4-5 years. This leads to the question one might ask: *why not wait* a few years for the next generation of sequential processors, instead of working with MIMD machines. Furthermore, they might argue, by the time a new and novel MIMD architecture has been developed it may well be superseded by a faster sequential machine. The problem with these arguments is that the development of implementation technologies, sequential processor architectures and MIMD architectures complement each other. Thus, faster sequential processors mean faster processing elements within MIMD machines, and higher performance overall. Ideally, MIMD machines with n processors should simply be n times faster than SISD machines constructed from equivalent technology, although in practice this is rarely the case. Also, having the Moore's law applied nowadays seems a bit harsh, since technological limits on manufacturing processors are close (Moore himself declared *the doubling will slow down* in a July 2002 interview, but did not agree that his law is going to a dead-end).

MIMD types Most computer scientists, and users of MIMD machines, draw a distinction between *multiprocessor* and *multicomputer* systems. If one considers a processor as simply a component of a computer system, then the distinction becomes clearer. A multiprocessor is then a system in which there is a simple replication of processors within a framework which does not alter the relationship between the processor(s) and other components (such as memory). Conversely, a multicomputer is a system in which the whole computer (processor and memory together) are replicated, and some form of communication network added, to allow them to exchange information.

   Multiprocessor systems are natively shared-resource systems, with the most important feature of having a common, or shared, memory system. The connection between the processors and memory is done through some form of interconnection network (see 3.2, b). Every processor may use the shared memory system and have access to the same data. However, the fact that each processor has its own cache implies establishing a cache coherency protocol. These protocols are classified into UMA, NUMA, and COMA.

   Multicomputer systems are distributed computer systems, with each processor having it's own I/O system, its own local memory and cache, and its own permanent data storing device. The total system memory is virtual and contains all the local memories, mapped with virtual addresses. Thus, accessing a location of memory which is outside the local memory is being done through a message mechanism: the processor which needs the data sends messages through the interconnection network to the data owner, which in turn sends a message with the data (see 3.2, c). This is why the multicomputers are also called message passing systems. A comparison of multicomputers to multiprocessors reveals that multicomputer systems *scale* better than multiprocessor systems, that is, more processing units may be easily added to the system, while the speed of data access is much higher in the case of multiprocessors.
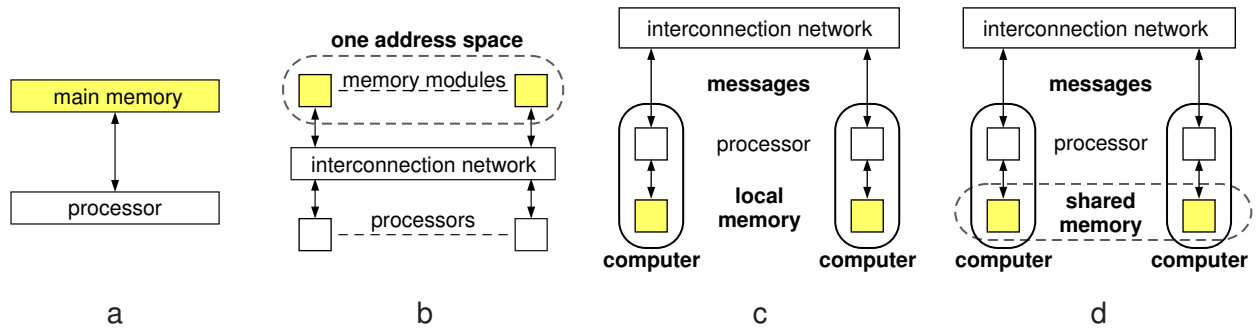
Figure 3.2: *(a)* Conventional computer, with a single processor and memory; *(b)* Multi-processor system, with several processors accessing the shared-memory system through an interconnection network; *(c)* Multicomputer system, with several processors having each a local memory and data transfer through message passing; *(d)* Distributed shared-memory system, with several processors accessing the memory through a *transparent* shared-memory system.

Distributed shared-memory(DSM) systems have emerged from the necessity of implementing shared-memory systems over distributed systems. They combine the ease of use of shared-memory systems with the scalability of, for instance, clusters of workstations (COWs). In figure 3.2, you may see that DSMs (d) are very similar to multicomputers (c), but the local memory system from the latter is replaced by a shared-memory system which is totally controlled by the hardware and, thus, transparent to the user.

3.4 Performance evaluation of parallel systems

Fundamental laws The main problem when dealing with parallel computing is how to utilize effectively the available processing power. It is obvious that, in order to obtain more performance, one has to execute several job streams in parallel, but how to effectively do this depends from machine architecture to machine architecture, and from problem to problem. Due to this cause, comparing a system with another in terms of performance may be difficult and yield into different results as different performance criteria (metrics) are used. In this section several fundamental criteria for measuring the performance of parallel systems are presented. More formal definitions may be found in the papers edited by Lee [61] or Hu [52].

Normal speedup and parallelizability

Normal speedup is defined as the ratio of the execution time of the parallel algorithm on a single processor to the execution time of the (same) parallel algorithm on a n-processor parallel system (see equation 3.1). Due to the fact that it compares the execution times of the parallel program on a varying number of processors, this performance criteria is also called *the parallel programmer's point of view*. As a parallel algorithm usually yields to worse results when run on a single processor than a good purely sequential algorithm run on the

same processor, the speedup measures are not of great interest to the user.

$$SU1(n) = \frac{T_{par}(1)}{T_{par}(n)} \tag{3.1}$$

Parallelizability is defined as the ratio of the execution time of the best sequential algorithm on a single processor to the execution time of the parallel algorithm on a n-processor parallel system (see equation 3.2). Due to the fact that it compares the execution time of the best sequential algorithm (that is, the best version a normal user would get) to the parallel program on a varying number of processors, this performance criteria is also called *the user's point of view*. This criteria is also prone to errors, because the best sequential algorithm is a term that may be hard to decide for each problem.

$$SU2(n) = \frac{T_{seq}(1)}{T_{par}(n)} \tag{3.2}$$

Speedup has been introduced to unify the definitions in the previous two paragraphs. As the terms of *normal speedup* and *parallelizability* have pretty much the same theoretical value (one might argue that it suffices to *adjust* the parallel algorithm to act differently depending on the number of available processors to make the two terms the same), the term speedup defines the best ratio from the two above and is formally written as in the equation 3.3, where n is the number of available processors to run the parallel algorithm.

$$S(n) = \frac{T(1)}{T(n)} \tag{3.3}$$

Efficiency is defined as the ratio of speedup to the number of processors, or

$$E(n) = \frac{S(1)}{n} = \frac{T(1)}{nT(n)} \tag{3.4}$$

Degree of parallelism is the maximum number of processors used to execute a program at a particular moment in time, given an unbounded number of available processors and all the other required resources. Normally DOP is not achievable in real world, due to memory or network latency problems. Note that DOP is usually a statistical figure more than a single run result.

Parallelism profile is the graphical representation of the degrees of parallelism (DOP) over time. The system contains n processors and the maximum DOP in a profile is m. In the figure 3.3, a work pool algorithm starts by allocating 6 processors (one is the producer and the other five are the consumers). This means that DOP when the system begins working is 6. Then, one consumer decides its processing task finished with an unexpected error and quits the system; thus the DOP also drops to 5. The producer has the ability to allocate more consumers as it determines the system has enough resources available, and so it does until it reaches the maximum DOP of 12 at about time tick 15 (note that $m = 12$). Then, as the consumers take more and more processor cycles and another user runs an external program, the DOP drops to about 4, where it stabilizes until the end. This is an example of the fact that DOP often changes at different periods of time during the execution cycle. However, the
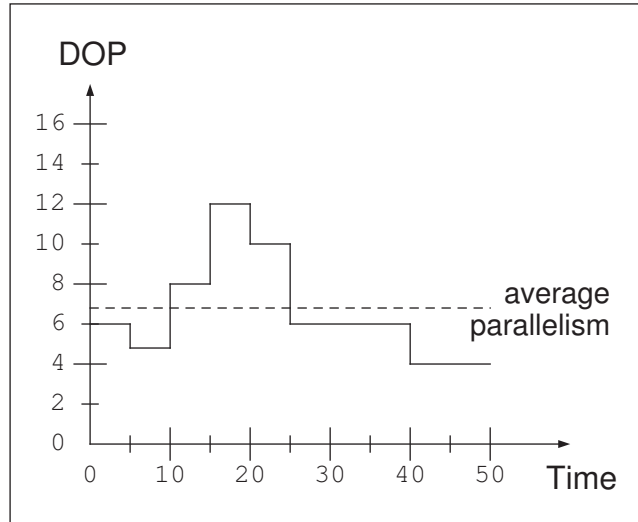
Figure 3.3: The parallelism profile and average parallelism for a work pool algorithm.

parallelism profile is of much help into detecting the various situations that occurred during program execution.

The **average parallelism** is the average number of processors that are busy during the execution time of the software system in question, given an unbounded number of available processors and needed resources, as shown in equation 3.5. This gives an useful metric to evaluate both speedup and efficiency.

$$A(n) = \frac{\left(\sum_{i=1}^{m} i \cdot t_i\right)}{\left(\sum_{i=1}^{m} t_i\right)} \tag{3.5}$$

Eager et al. [27] show that the relation between the speedup, the efficiency and the average parallelism for systems without communication overhead (ideal) is as follows:

$$S(n) \geq \frac{nA}{n + A - 1}$$

and

$$E(n) \geq \frac{A}{n + A - 1}$$

We can see that $S(n)$ is bounded by $nA/(n + A + 1)$ and $E(n)$ is bounded $A/(n + A + 1)$. Also, when $n \ll A$, $S(n) \to n$, and when $n \gg A$, $S(n) \to A$.

Asymptotic speedup is the ratio between the total time for executing a workload W on a single processor to the total time of executing the same workload on an infinity of processors, considering that no overhead occurs. Note that this performance index is the best possible speedup that can be achieved.

Let W be the total amount of work for a given application, W_i the amount of work executed with $DOP = i$, and $\triangle$ the computing capacity of each processor, which can be approximated as the execution rate. Thus, we have

$$W_i = i\triangle t_i$$

and

$$W = \sum_{i=1}^{m} W_i = \triangle \sum_{i=1}^{m} i \cdot t_i$$

By denoting the execution time of W_i (the amount of work executed when i processors work in parallel) on k processors as $t_i(k)$, we can write

$$
\begin{aligned}
t_i(1) &= W_i/\triangle \\
t_i(k) &= W_i/k\triangle \\
t_i(\infty) &= W_i/i\triangle, \quad 1 \le i \le m
\end{aligned}
$$

Note that even if an infinity of processors would be available, the maximum number of processors used to execute W_i is still i.
We have

$$
\begin{aligned}
T(1) &= \sum_{i=1}^{m} t_i(1) = \sum_{i=1}^{m} \frac{W_i}{\triangle} \\
T(\infty) &= \sum_{i=1}^{m} t_i(\infty) = \sum_{i=1}^{m} \frac{W_i}{i\triangle}
\end{aligned}
$$

Thus, the asymptotic speedup is formally defined as in the equation 3.6.

$$S_\infty = \frac{T(1)}{T(\infty)} = \frac{\sum_{i=1}^{m} W_i/\triangle}{\sum_{i=1}^{m} W_i/i\triangle} = \frac{\sum_{i=1}^{m} W_i}{\sum_{i=1}^{m} W_i/i} \tag{3.6}$$

The Amdahl's law governs the maximum speedup that may be achieved for a given algorithm, when performing a fixed amount of work [3].
In the asymptotic speedup model, we assume that the number of available processors is unbounded. If the number of processors n is taken into account, and $n < i$, we have the execution time on n processors, $t_i(n)$, given by the formula

$$t_i(n) = \frac{W_i}{i\triangle} \left\lceil \frac{i}{n} \right\rceil$$

Note that if $n \ge i$, $\lceil i/n \rceil = 1$ and the equation holds. We have the execution time of W on n processors

$$T(n) = \sum_{i=1}^{m} t_i(n) = \sum_{i=1}^{m} \frac{W_i}{i\triangle} \left\lceil \frac{i}{n} \right\rceil$$

and the corresponding speedup given by the formula

$$S(n) = \frac{T(1)}{T(n)} = \frac{\sum_{i=1}^{m} W_i}{\sum_{i=1}^{m} \frac{W_i}{i} \left\lceil \frac{i}{n} \right\rceil}$$

or, considering the effects of overhead denoted by $Q(n)$, the speedup is being given by the formula 3.7:

$$S(n) = \frac{T(1)}{T(n)} = \frac{\sum_{i=1}^{m} W_i}{\sum_{i=1}^{m} \frac{W_i}{i} \left\lceil \frac{i}{n} \right\rceil + Q(n)} \tag{3.7}$$

In the following discussion we consider the problem size, or workload, as fixed, and as being the sum of a completely sequential part, i.e. with $DOP = 1$, and a perfectly parallel part, i.e. with $DOP = n$ and $W_i = 0$ for $1 < i < n$, with a null system overhead, i.e. with $Q(n) = 0$. Hence, the equation 3.7 describes a *fixed-size speedup* and becomes

$$S(n) = \frac{W_1 + W_n}{W_1 + W_n/n} \tag{3.8}$$

Let α be $W_1/(W_1 + W_n)$, representing the percentage of the sequential part of the program from the total workload. If we consider a normalized workload, i.e. $W_1 + W_n = 1$, we have $W_1 = \alpha$ and $W_n = 1 - \alpha$. The speedup formula 3.8 becomes

$$S(n) = \frac{1}{\alpha + (1 - \alpha)/n} = \frac{n}{1 + (n - 1)\alpha} \tag{3.9}$$

which is in fact the Amdahl's law. It states that if α is the sequential fraction of an arbitrary algorithm, the maximum speedup is $1/\alpha$, no matter how many processing units are added. Hence, α is called the *sequential bottleneck* and is *algorithm* and *machine* specific.

Discussing a bit further equation 3.9 we can observe some other obvious speedup values. If a program is purely sequential, i.e. $\alpha = 1$, $S(n) = 1$, which means that the program will not be executed any faster no matter how many processors we add. If a program is purely parallel, i.e. $\alpha = 0$, $S(n) = n$, which means that a program execution cannot increase more than linearly with the increase in processing units number. *Superlinear speedup* is thus impossible to achieve under the Amdahl's law considerations. However, nearly linear speedup has been achieved for large problems run on supercomputers.

The Gustafson's law deals with solving problems in a fixed amount of time, with increase in computing accuracy or problem dimension [43]. The execution time for these problems is not always important (although it needs to be finite and, usually, upper-bounded), so focus is put on increasing the problem size when adding more computing resources to the system. This is opposed to Amdahl's considerations, which considered the workload as fixed and the time as (hopefully) reducing with addition of new computing units.

Let W' be the amount of scaled workload, W'_i be the amount of scaled work with $DOP = i$,

and m' be the maximum DOP of the scaled problem. We have the total workload given by the following equation:

$$W = \sum_{i=1}^{m'} W'$$

Under the Gustafson's law conditions, total computing time must be kept constant, i.e. $T(1) = T'(n)$. This means that

$$\sum_{i=1}^{m} W_i = \sum_{i=1}^{m'} \frac{W_i'}{i} \left\lceil \frac{i}{n} \right\rceil + Q(n) \tag{3.10}$$

and the speedup is given by the formula.

$$S'(n) = \frac{T'(1)}{T'(n)} = \frac{\sum_{i=1}^{m'} W_i'}{\sum_{i=1}^{m'} \frac{W_i'}{i} \left\lceil \frac{i}{n} \right\rceil + Q(n)} = \frac{\sum_{i=1}^{m'} W_i'}{\sum_{i=1}^{m} W_i} \tag{3.11}$$

As with the discussion in the Amdahl law's case, we consider the workload as being the made of a completely sequential part and a perfectly parallel part, with a null system overhead. We also assume that the sequential part is independent of the problem size, i.e. $W_1 = W_1'$. Under these circumstances, form equation 3.10 we obtain $W_1 + W_n = W_1' + W_n'/n$, thus $W_n' = nW_n$, which means that the scaled amount of work performed with n processors is n times the amount of work performed with 1 processor. Hence, the equation 3.7 describes a *fixed-time speedup* and becomes

$$S'(n) = \frac{W_1' + W_n'}{W_1 + W_n} = \frac{W_1 + nW_n}{W_1 + W_n} \tag{3.12}$$

Let the workload total be normalized. We have $\alpha = W_1$ and $1 - \alpha = W_n$, and the speedup becomes:

$$S'(n) = n + (1 - n)\alpha \tag{3.13}$$

which is Gustafson's law. It states that as more processing units are added, the parallel part scales up linearly. Therefore, the sequential part of a program is no longer a bottleneck for the speedup. The speedup may still become linear, if a program is purely parallel, or stay at 1, or *program not speeded up*, if a program is purely sequential.

Scalability is a measure of compatibility between a computer architecture and an application algorithm. A given algorithm on a given platform is perfectly scalable if the performance of a parallel architecture increases linearly with the increase of the processing units number. Hence, scalability helps evaluate an algorithm performance, but it does so while heavily depending on the computer architecture on which the algorithm is being run. On the other hand, scalability may also help evaluating a given architecture for a given algorithm.

Scalability analysis may be useful in the following cases:

- Selecting the best algorithm-architecture combination for a problem, under different constraints on the growth of the problem size (workload) and the number of processors (machine size);

- For a fixed problem size, determining the optimal number of processors to be used and the maximum speedup that may be achieved;

- Predicting the performance of a parallel algorithm / architecture duo for a large number of processing units from the performance on few processors;

In general, a parallel system is considered *scalable* if the speedup curve is linear or near-linear (for instance, if the problem size can grow linearly with the machine size), and *unscalable*, otherwise.

Isoefficiency greatly helps analyzing the scalability of a parallel system.

Let T_1 be the sequential execution time, T_p be the parallel execution time on p processors, and T_o be the total overhead. We can see that

$$T_p = \frac{T_1 + T_o}{p}$$

The speedup is

$$S = \frac{T_1}{T_p} = \frac{pT_1}{T_1 + T_o}$$

The efficiency is then computed as

$$E = \frac{S}{p} = \frac{T_1}{T_1 + T_o} = \frac{1}{1 + \dfrac{T_o}{T_1}}$$

Let W be the initial problem size, measured in number of operations executed by the best sequential algorithm to solve the problem, and t_c be the cost of each operation. Then $T_1 = t_c W$ and the efficiency can be rewritten as

$$E = \frac{1}{1 + \dfrac{T_o}{t_c W}} \tag{3.14}$$

The main goal is to keep the efficiency of the parallel system stable, or increasing, when adding new processing units. For this matter, analyzing equation 3.14 gives some information. First, if the problem size is fixed, i.e. W is constant, the efficiency decreases, as T_o, the overhead time, grows constantly with the addition of new processors. Second, if the number of processors is fixed, the efficiency increases, as the overhead time, T_o is supposedly growing slower than the problem size, W, that the now more numerous processors may tackle. It is clear that varying the problem size and the number of processors it may be possible to maintain a constant efficiency.

By rewriting equation 3.14 we obtain the workload as a function of efficiency, as shown by the formula below:

$$W = \frac{1}{t_c}\left(\frac{E}{1 - E}\right) = kT_o, \quad k = \frac{E}{t_c(1 - E)}$$

If we maintain the efficiency constant, k is constant too, and the workload needs to grow as fast as the overhead time. If this happens, the isoefficiency function is defined as $f_E(p) = kT_o$.

Isoefficiency is a good measure for scalability analysis. A small isoefficiency function shows that small increments in problem size are sufficient for efficiently using an increasing number of processors, which in turn means that the system is highly scalable. On the contrary, a large isoefficiency function shows that problem size needs to be greatly increased even when adding just one more processor to achieve the same efficiency, and such a system is poorly scalable. Moreover, when an isoefficiency function cannot be established or it is too big, the system is considered unscalable.

3.5 Difficulties for parallel programming

Sequential and parallel impediments

Sequential   parts of a program limit, as shown by Amdahl's law, the maximum performance achievable when parallelizing the application. However, Gustafson's law demonstrates that, if we can increase the problem size with the number of new processors added to the system, ie. the problem is scalable, we can obtain an almost linear speedup. See the discussion about this in the previous section.

Parallel   approaches for an arbitrary algorithm may suffer from at least two problems:

- Too little use of parallelism, which means that some parts of the program are left sequential, thus limiting the overall speedup (see the *sequential bottleneck* discussion in the previous chapter, in the Amdahl's law section)

- Too much use of parallelism, a consequence of trying to parallelize everything possible in a given sequential algorithm, which in turn leads to increased overhead, due to communication, concurrent memory accesses and so on.

It is clear that one has to carefully prepare which parts of a given sequential algorithm to parallelize and how to do it to avoid unnecessary overhead, in order to achieve superior speedup and efficiency. Furthermore, dealing with unscalable problems only allows for efficient approaches when using small numbers of processors, such as 2, 4, or even 8.

Programming complexity drawback

Parallel languages and libraries   Programming parallel-aware applications requires special compilers, such as `pC++` [11], `Orca` [9], `High-Performance Fortran` [50], `Linda` [68], `SR` or libraries, such as `MPI` [31], `PVM` [37], `OpenMP` [73]. There is also the third option of designing and implementing a new compiler / library *from scratch*, but this case is very seldom nowadays [92]. The need to learn the compilers and / or libraries features and drawbacks increases the time for developing parallel applications and also increasing the possibility of software bugs.

Synchronization and race conditions   When linking together the parallel parts of an application one must ensure that no module executes its computation on data which has not been yet processed by all other modules, as required by the processing algorithm, or that there are not two separate modules that modify the value of the same variable. Both

are resolved through the use of synchronization primitives: *locks* or *mutexes*, *semaphores*, *monitors*, *barriers*.

Modules utility impediments

Modules frequency of use As appealing as it may seem to parallelize, parts of the applications may be rarely used in practice, thus not requiring a parallel approach. Another type of applications for which parallelization is too time-costly are the systems with a reduced life-term, e.g. modules that need frequent and in-depth modifications.

Modules dimension As parallelism induces increases of development time, it may be not worth to spend time parallelizing a problem whose dimensions would yield into gaining just a few fractions of a second second against a purely sequential version. On the other hand, if the problem is scalable or it is a critical part of the application (as it is the case for the module we have optimized and parallelized in our robotic system), a parallel approach may be desired.

Tackling industry reticence Carver Mead of Caltech introduced the idea of *headroom* - how much better does a new technology to be in order to replace the old in the industry. For parallel computing to become common use it would need a headroom of at least an order of magnitude bigger [35]. That is, in order to replace the *traditional* sequential programming, parallel computing should have an outcome of at least 10 times the performance / cost / efficiency (the criteria may differ from case to case) of the sequential implementation.

3.6 Parallel programming paradigms

As seen in the previous chapters, writing efficient parallel programs is a difficult task, so efforts have been put into providing parallel computing paradigms, some of which deal with communication among processing units:

- Shared-memory paradigm, where data may be accessed directly by every processing unit. This is the natural method of parallel programming for multiprocessors.

- Message-passing paradigm, where all data retrieval and communication between processing units is based on explicit messages. This is the natural method of parallel programming for multicomputers. Two standardized libraries are publicly available: `PVM` [37] and `MPI` [31].

- Mixed shared-memory and message-passing paradigm, combining both message-passing and shared-memory paradigms. An example is `MPI-OpenMP`, which is very efficient on execution time, but much more complex to deal with, because of the two programming models.

- Natively parallel paradigms, such as `ParCeL` model [94, 85, 84, 83] and `Linda` [68].

In this project we used both the shared-memory paradigm and the message-passing paradigm, separately, to compare the outcomes.

3.6.1 Shared-memory parallel programming paradigms

In this model the memory is shared across the system, which means that every processor may access directly any memory location. Having more processors tackling the same problem is being done in one of the following methods:

- Explicit multi-processing. Several processes are being explicitly started by the user in order to handle different parts of a common problem. A process is a code execution entity, with its own context information (program counter, register set, stack), its own variables space, and is treated by the operating system like any other program (for systems like `UNIX` every running program is a process; processes may be created by using a `fork` system call). Therefore, processes are usually called *heavyweight processes.*

- Explicit multi-threading. Several threads are being explicitly started by the user in order to handle different parts of a common problem. A thread is a code execution entity, with its own context information, just like the processes'. A thread, however, does not contain its own address space; instead, it uses the parent process / program address space. This is why threads are also called lightweight processes. The most important advantages of threads, compared to processes, are the creation and switching time and the ability to easily share data between several threads. The disadvantages of threads, compared to processes, include the lack of data security, and, on some operating systems, the fact that all threads in a task block if one of them blocks. Several multi-threading interfaces exist, of which `pthreads` [5, 21, 62], the standard `POSIX` threads interface stands out.

- Implicit multi-threading, where some thread-specific techniques are directly applied by the compiler, based on hints given by the programmer. Data decomposition, for example, is such a technique, but might prove ineffective for problems with irregular data treatment patterns. Examples of implicit multi-threading systems are `OpenMP` [73] and the `Java` threads system.

In this project we used explicit multi-threading, through the `pthreads` library, for the tests involving shared-memory systems.

3.6.2 Distributed memory programming paradigms

In this model the memory is distributed across the system, which means that every processor has to use messages to other processors in order to access any distant memory location. Optimizing the message passing routines are critical for libraries implementing the distributed memory programming paradigms, such as `MPI`, but this has to be performed in different ways depending on the parallel data treatment flows. From this point of view, there are two main directions:

- SPMD - Typically, message passing and distributed shared memory abstractions are used in a Single-Program-Multiple-Data (SPMD) model, where a fixed number of identical programs operate on their local data and communicate with one another at well defined points in time. This is particularly good for multiple operations like `gather`ing or `scatter`ing values to several processors at the same time (broadcasting and multicasting), and is a reason why several hardware architectures have been designed to

favor this kind of behavior. It is no surprise that the `MPI` system optimally handles the SPMD programming model. A typical SPMD problem is an image treatment in which each processing unit handles a chunk of the original image, such as the case of inverting the colors of a given picture.

- MPMD - This model allows for multiple programs to dynamically create concurrently executing tasks that communicate with one another at any point in time. Though this model may not have hardware support available on all machines, it is it is well suited for applications that exhibit irregular or unknown communication patterns, or that can benefit from a *client-server* type of setting. Also this system is more appropriate for meta-computing in a heterogeneous, large-scale environments because it supports more loosely-coupled components than SPMD. A typical MPMD problem is a boss-worker problem, in which a processor, called the `boss`, must control the work and assemble the results of several other processors, called the `workers`. Note that MPMD programs may be generally mimicked by SPMD programs that have some sort of flow control modifications and include all the necessary modules in a tightly-coupled manner.

In this project we used the SPMD paradigm. Mimicking MPMD for the boss-workers model was also used.

3.7 Exploiting the parallel nature of mobile robotics applications, an overview

Fifteen years ago, the traditional robot control architectures involved one vertical, single-threaded control module. Since then, especially with the works of Brooks [17] on layered architectures and Arkin [4] on behaviors, this classical architecture started to be decomposed into smaller, independent units, organized in a hierarchical, if not flat, manner. Especially parallel and distributed approaches were employed to make use of these architectures. Tele-operation was born from the use of remote control, sometimes with the use of distributed computing.

Parallel

Parallel processing in robotic applications were starting to appear as early as mid-80s [41]. Nowadays, many robotic systems use parallel approaches to tackle the soft real-time constraints of mobile robotics [96, 45, 44, 46]. Depending on the level of parallelism, there are two main categories of robotic applications: single and multi- robots.

Single robots Single robots systems make use of just one robot in order to solve a certain problem. This does not mean that the approach is simplified, as a single robot may present several degrees of freedom. There are seven directions in which parallel computing is used: components, kinematics, control, functions, behaviors, abstraction and algorithm. A good survey of the parallel processing approaches in single robots applications can be found in [47].

Components This approach sees each distinct hardware component of the robotic system as a computationally independent module, with its own attached processing unit. For

example, a robotic system with one manipulator mounted on top of a mobile rack, with camera-based orientation, will have at least one processing unit assigned for the manipulator, the mobile rack and the camera.

Kinematics For some robotic systems, the main control loop is just too complicated to be executed by just one processing unit. This approach tries to decompose the controller into several smaller items, each executed in parallel.

Speculative parallel approaches have been tried when data-dependencies occur inside the main control loop [13, 44].

Control The control task for a given system involves high speed input updating, usually through some form of polling, i.e. sampling of the input sensors occurring at fixed rates. The polling process involves setting hardware timers, and handling several synchronous or asynchronous input systems at the same time. This may prove too computationally expensive for just one processing unit. A solution is to partition the task into simpler, smaller, subtasks which are small enough to be performed by a single processing unit, then pipelining the control loop subtasks on several processing units.

Functions Robot control architectures have four main functions [47]: perception, planning, execution and exception handling. Assigning one or more processors for each of these functions, so that high-level parallel processing occurs.

Parallel blackboard based processing is a powerful method for flexibly combining individually developed modules into a single integrated application . The term blackboard suggests the following scenario involving a group of people engaged in a brainstorming session: a group of specialists are in a room, with a large blackboard serving as the workplace for cooperatively developing a solution to the problem being examined. For mobile robotics, each functional module modifies the global application state variables and all other modules may see the changes. An example of such a system is the NAVLAB from CMU [40]. NAVLAB has five modules (global plnning, local planning, perception, mission execution and hardware control), communicating via a parallel blackboard.

Behaviors Behaviors are simple sensor-effector connections, much like the stimulus-response reflex. Thus, the behavior based robotics paradigm advocates for minimal representation of knowledge.

Subsumptive, or bottom-up, architectures assume that multiple or all behaviors may be active at a given moment [17]. Such a system, based on very simple behaviors, called schemas, is presented in [4]. It is also possible that some behaviors, considered more *competent*, subsume the others. An example is the DAMN system [80], where a voting system uses priorities to determine the dominant behavior. The inherent parallelism of the subsumptive systems suffers from a severe problem: the behaviors interaction can become intractable for large problems [24]. This problem is avoided in [24], where each control loop performs in a completely independent fashion, in order to control a Kawasaki UX120 system.

Abstraction The abstraction direction divides the system according to the data abstraction and time response. The approaches types vary from centralized (tree) to layered, with communication taking place between a level and its neighbors only.

In [90], ANIMA (Architecture for Natural Intelligence in Machine Applications), a three-levelled centralized architecture, uses five processors, and can be easily extended with others. In [2], NASA NASREM architecture presents a six-layered system based on timing, with the mission, service bay, task, e-move, primitive and servo tasks being handled by parallel processing units.

Algorithm This direction targets the use of parallel algorithms to solve the real-time constraints of a robotic system. Usually embarrassingly parallel algorithms such as ones used in image processing are employed, but motion planning and dynamics algorithms which need large amounts of computation are also frequent subjects to parallelization.

Image processing techniques have been under study for over three decades now, and parallel algorithms, implemented in both hardware and software, are available. Good overviews on vision algorithms for mobile robotics are given in [18, 25]. Reconfigurable logic arrays have been used in [79] to efficiently implement image processing algorithms.

An efficient system for solving the dynamics of a robotic system is described in [41], alongside guidelines on the use of special hardware architectures in robotics. A survey of parallel approaches for kinematics is given in [48].

Multirobots Multirobots are often used in robotic applications covering large areas, such as exploratory missions, or where it is critical that all the space is safely under surveillance, such as the team house-guards, or when, simply, the task is shaped for several robotic agents, such as the case of robotic soccer or cooperative movement of objects.

Completely decentralized Nearly all the work involving cooperative mobile robotics started immediately after pioneering work described in [4] and [17]. Because of its biological inspiration, many scientist began studying various biological societies, like ants and bees, turning towards a decentralized system with the robots grouped in swarms achieving targets by the sheer fact of numbers. Such systems may use one processing unit per robot, or even none, due to miniaturization, with a global observation system checking for goal achievement.

The number of robots in the swarms has greatly increased in the recent years, as costs for producing such robots was greatly reduced. Nowadays, over a thousand robots may be used in a single swarms application, hence, the emergence of the term `kilorobotics`. Simulation studies with as much as 1500 robots were successfully conducted [89].

Distributed agents One way to increase the performance of a robot swarm is collaboration. The robots are not acting by themselves only, nor are placed under a centralized control; instead, they use communication to coordinate the group towards a certain goal. Such systems may use one processing unit per robot, or even several processors when the tasks involved require such features.

A representative example is the RoboCup robotic soccer game. Teams of robots are engaged in real or simulated soccer matches. Many of the early RoboCup teams used this kind of approach, when teams used communication to avoid unnecessary crowding of players near the ball [70].

Sometimes cooperation means that other robots' data is used to verify one robot's results, especially regarding localization. The Millibots system [69] from CMU used four robots, with each robot moving while the other three are used as means of beacons of a triangulation-based

localization. Another approach, which sets a team of surveillance robots such as to best cover a guarded area is described in [74].

Centralized The centralized approach is used for teams in which a certain member acts as the master (hive), while the others obey the master's orders. This approach may be two-levelled or hierarchical. Usually, each of the robots involved in the centralized approach uses at least one processing unit, with several processors usually embarked on the most important, master, robots.

The master scheduling algorithm sometimes uses slaves feedback (requests) to refine the plan [95].

Another interesting approach is given in [39], where a team of scouting robots that is deployed by a *hive* robot. After searching the area, the robots return to the hive robot and are carried away by it.

Remote control

The remote control for mobile robotics has two major directions. One is the use of human operators as robotic systems assistants, and the second is the use of distant resources to perform some of the necessary computing tasks for the robots. This section focuses on the first approach, while the second is described in the following section.

Teleoperation is the process of direct control of robotic actions by a human operator, through the use of a low-level control functions interface, such as multiple joysticks or keyboards. The *hidden robot* concept [57] sees the robot as a virtual resource, located in a virtual environment. The operator commands the robot through a hi-tech interface that tries to simulate the robot's world and its responses to orders as accurately as possible. Typical problems lie in the communication layer, with delayed transmission and data loss being the main issues [71, 72].

Full-time teleoperated robots This approach is used when a direct operator control is required at all times, during the application run-time. Typical examples are radioactive manipulators at SANDIA or `telesurgery` [1], the human assisted robot-on-human operation. In these cases the robot acts as a tool, not a *smart* machine, performing the exact movements given by the operator. Evolved interfaces with tactile feedback have been conceived for telesurgery applications.

Part-time teleoperated robots Having operators using master/slave manipulator tools for long periods of time is not always possible. Fatigue and boredom are two important causes that can be avoided if the robots could perform autonomously tasks with high success expectations or execute more complicate commands than the low-level articulation state change. Virtual laboratories impose collaboration features and resource control arbitration. Finally, long distance communication renders continuous remote control impossible.

`Teleautonomy` involves a shared control between the robot and the human operator. In [14], the operator issues commands that are executed just to the extent of becoming potentially dangerous for the robotic system. The control is then gradually transferred to the autonomous obstacle avoidance. Thus, errors that result from fatigue or mistake are corrected.

---

[1]robotic surgery may be operator- or image- driven

`Operabotics` are systems that make use of the robotic components in combination with an operator. The robotic low-level components are used by an automated system, while the operator uses just high-level commands, sometimes expressed in a set of natural language orders, to a clear productivity improvement. An example of a team of robots coordinated by a single operator using dialogue is presented in [30]. Use of operabotics greatly reduces the operator's boredom.

Virtual laboratories with on-line robots are also examples of discontinuous remote control. Such laboratories may provide distributed collaboration among multiple sites, dynamic reconfiguration of physical robotic resources and user connections, up to public outreach in media covered experiments [6]. In order to make them available to multiple users, physical resources, such as robots or sensors, must be managed by servers accepting incoming requests and providing control and arbitration over resource allocation.

Finally, distance considerations make the continuous remote control impractical for some applications, such as lunar rovers or other kinds of space missions. A recent example is the Mars Pathfinder mission, in which an intelligent robot was teleoperated with radio signals from Earth [6]. The robot executed the orders and sent back the results. The human operators selected new courses of action based on the transmitted results and uploaded the new plan to the remote robot.

One-time teleoperated robots This category includes robots that learn from human demonstration or robots that follow a demonstration track - every motion performed by the operator is recorded, then reproduced by the robot. This is similar to the way the first robots were built, but adds the ability of remote training.

Distributed

Distributed robotic systems make use of local or distant processing units for their computing needs, in a *proxy-processing* scheme. Thus, in this paradigm, remote machines add computing power to the limited on-board computing capabilities. Common problems include the typical remote control communication problems, i.e. delays in data delivery and information loss, and add dynamic resource allocation and further management, under efficiency and hierarchical considerations.

From a resource awareness point of view, there are two kinds of distributed robotics systems: robots with resource awareness and robots with resource delegates awareness. The first has the advantage of specifically knowing where to locate resources, but also have the disadvantage of having to use application-built schedulers, while the latter systems do not know specifically where the resources are located, but have the ability of using a resource advertise (broker) system, which handles the resource management for them.

Robots with resource awareness There are three different classes of systems that have *apriori* knowledge of the resources locations: resource adapted systems, resource adaptive systems and resource adapting systems [10].

Resource adapted systems have been optimized for a given set of resources prior to the application execution. Thus, their performance is well-known and follows patterns determined in advance.

Resource adaptive systems use a single strategy to deal with variation in resource quality and/or number. Thus, the performance varies greatly with the available resources.

Resource adapting systems use several approaches to deal with variation in resource quality and/or number. Usually, some sort of expert system is involved in the strategy selection. Thus, the performance varies with the available resources, but best solutions may be obtained for more configurations than in the case of *resource adaptive* systems. An example is the REAL system proposed in [10], where the navigation system automatically selects between indoor and outdoor localization strategies, depending on the current user's position.

Robots using Client/Broker/Server architectures A special kind of robotic systems is represented by the class of robots which do not know where the much-needed resources may be located, but have a broker address instead. The resources register themselves to the broker agent, which is known to all the possible client applications (or may be found using some broker agents directory service). Upon request, the broker may find the appropriate resource through some reservation mechanism and facilitate a connection between the robotic system and the resource. Two key technologies involved are inter-process communication and remote code execution.

There are two approaches towards remote code execution: remote procedure call and object-oriented. Under the remote procedure call paradigm, a system may execute a certain chunk of code and return its results, while the caller blocks its execution until the result is being returned, much like a procedure execution in a von Neumann sequential architecture. The object-oriented paradigm advocates an object oriented view upon the world, with objects being serialized and sent to the remote execution point. Due to security reasons, information about the characteristics of the code that is to be remotely executed need to be known on the remote machine *before* the actual execution. The remote procedure call paradigm is essentially similar to distributed object oriented paradigm.

Distributed object oriented This paradigm uses object encapsulation and serialization as key technologies. The process of sending code from a machine to another is called *marshalling*, while the process of decoding the *marshalled* code is called unmarshalling. The marshalling and unmarshalling of the objects is usually being automatically done by an object management system.

Probably the most used such system today is the *Common Object Request Broker Architecture* (`CORBA`), by the *Object Management Group* (`OMG`). CORBA IDL is a universal language-independent notation for software interfaces. Objects are defined in CORBA using the IDL specification language, then implemented in the language of choice, as long as an IDL mapping exists for that language. The code is then compiled in two versions, for the server (called *stub* [2]) and for the client (called skeleton). This approach is also used in Java Remote Method Invocation. Systems like CORBA and Java RMI provide the OSI presentation layer, that is, they allow for data to be transparently transmitted from one system to another, alongside with general purpose execution methods. However, their main problems stem in the overhead introduced to the employing system, combined with information delivery issues. CORBA has recently improved performance dramatically, and is now close to the high-speed low-level software approaches, like sockets [23].

---

[2]a *stub* is the very small part that remains in a tree after it has been cut by a woodsman; in CORBA, a *stub* is a small program with no other use than to perform the code unmarshalling and then the unmarshalled code execution

Object-oriented systems are, however, difficult to master, and not many approaches have been done in this direction.

Hirukawa and Hara [51] propose a framework based on OO programming for robot control and argue that performance for using a *Java*-based approach or a *C++*-based approach on top of *CORBA* communication is the same for LANs and WANs.

A system making use of TAO [87], an advanced communication system built on top of CORBA, which offers hard transmission times guarantees, is described in [16].

Chapter 4

The robotic application

4.1 Introduction and goals

Several years ago we joined robotic researchers of ENSAM (France) and started research in the field of partially autonomous robots. We have designed and built a robotic application including signal processing, robot control and move, and some concurrent and distributed computing to support complex robot behaviors [88]. The main issue is to have an accurate and robust mobile robot control system. The control task is to ensure that the robot is performing as fast as possible for a given situation, through the use of common off-the-shelf computers, for the computing part and of various network types, for the communication part.

The work presented in this paper primarily focuses on severely optimizing the performance of a critical robot control module, the self-localization, in order to verify whether scalable speedup may be obtained for our mobile robotic applications. In addition, we target first attempts at discovering what the remote control paradigm constraints are in the case of mobile robotics, with a brief look upon remote control applicability for our application. For the computing part we prefer using simple PCs, but cheap off-the-shelf multiprocessor systems and small clusters of workstations may also be used. The communication network is ensured by the laboratory `LAN` (under `100 m` - very short distance), with Fast-Ethernet and Gigabit Ethernet technologies, or by an `ATM` connection to a similar laboratory at Nancy (distance under `100 km` - medium distance), or by `Internet` means to a laboratory in Salerno, Italy (distance oven `1000 km` - long distance).

4.2 Project history

1999 This project started back in 1999, when Stephane Vialle, from Supelec, and Ali Siadat, from ENSAM, created the project general framework and established the first milestones.

2000 Vincent Rieger completed a study about PSimilar Landmarks between February and June 2000. This work was done as a his due student practice. During this time, his work was parallelized by C.Rose.

2001 In 2001, Cyrille Roussel worked between February and September on a localization system based on PSimilar landmarks, in a Master's programme.
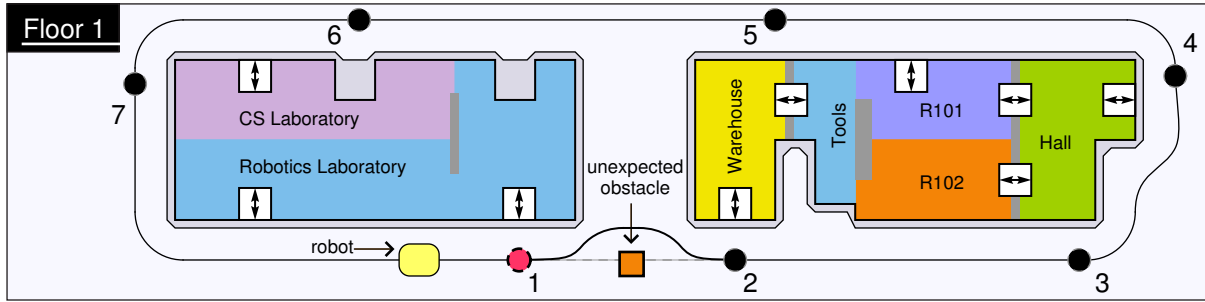
Figure 4.1: The complete mobile robotics application. An unexpected obstacle is observed (point 1). The old trajectory (dotted line between points 1 and 2) is replaced with a new one (curved trajectory between points 1 and 2). New self-localization routines are performed from time to time (points 3-7).

2002 Between January and April 2002 P.E. Mélet and F.Thuault completed their due end of studies project by parallelizing the work of Cyrille Roussel. A new communication library developed by Herve Frezza-Buet (Supelec) and Olivier Rochel (LORIA) was catching shape, starting late 2002.

2003 Matthieu Massonneau worked between January and April 2003 at a trajectory planning system for the robot.

Present general framework We consider robots navigating in industrial and office buildings. Because of the indoor environment, there are no guaranteesfor radio wave reception, so GPS usage is impossible. In addition, robots are being placed in changing environments: dorrs may be left open, chairs and other small objects may be moved, even furniture and decorative plants may have their locations changed. This is why we cannot use a map database, though it could prove beneficial into estimating the robot position. Instead, we use artificial landmarks, placed in the environment at known locations. Robots can detect these landmarks and self-localize with an adapted triangulation algorithm [64]. The robot may then compute a theoretical trajectory to reach a final position and follow that trajectory. Checking the environment while moving is also an option. While following the theoretical trajectories, robots use infra-red sensors and edge detection in camera acquired images to dynamically detect unexpected obstacles (see figure 4.1, point 1). If an obstacle is detected, a new self-localization allows to accurately point out the obstacle and to upgrade the map. Then a new trajectory is computed (see figure 4.1, the curved trajectory between points 1 and 2 replaces the old trajectory, the dotted line between the same points), and the process repeats until the goal is reached (in a surveillance mission the goal would be to stroll around the building for as much time as possible, with some time used to recharge at a battery charging base). Other self-localizations allow to compensate error on final or intermediate positions, when following long trajectories (see figure 4.1, points 5 and 6).

Application environment The physical testbed for our application is an area of $4m \times 2m$, enclosed by $30cm$ tall walls, which allow for PSimilar landmarks to be effectively mounted.
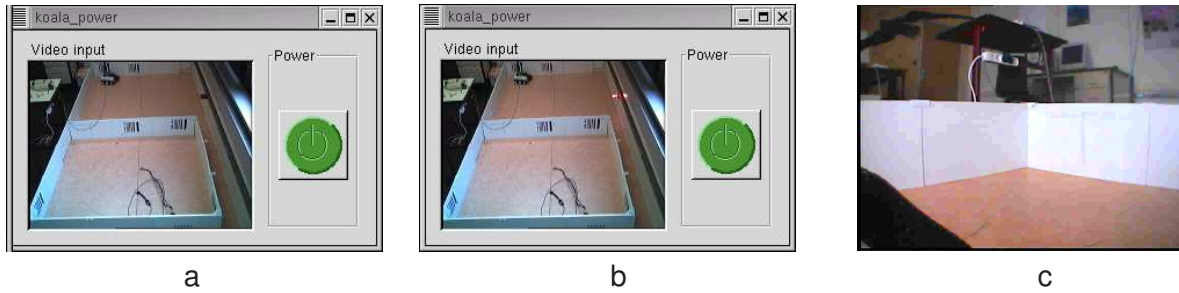
Figure 4.2: The panoramic and robot cameras in action: *(a)* The panoramic camera sees the environment, while the current is off; *(b)* The panoramic camera sees the environment, while the current is on (see the red light); *(c)* An image as seen by the robot's camera.

Walls can also be placed in this environment to create rooms or simple obstacles. The floor is covered with linoleum, which is a constant source of sliding during robot navigation, especially when performing tight turns. A panoramic camera allows a remote user to keep track of what is happening in the robot environment, while a software tool allows for instant current cut-off in case of severe error (figure 4.2, a, b). The user may also see the images as acquired by the robot (figure 4.2, c).

4.3 Robotic system hardware

This section presents our application hardware features. They are the robot, the computer and the communication network. Due to its low on-board computing power (section 4.3.1), the robot is controlled by an external computer through a serial link. When performing the self-localization, several computers may be used to process the acquired data, through LAN, ATM or Internet communication networks.

4.3.1 The robot

Figure 4.3: The Koala robot: a top-down view.

We use a `Koala` robot built by `K-Team` (figure 4.3). The size of the robot is of approximately $32cm \times 32cm \times 20cm$, for a weight of about $4kg$ without load, which means the robot is heavy enough to avoid sliding when walking forward. It has 16 Infrared proximity sensors, a

mounted rotating camera, 6 wheels, 3 on each side and 2 motors with PID control, each motor controlling a robot's side. Its on-board computing capabilities are based on a `Motorola 68331` processor at 22MHz. 1MB of `RAM` and 1MB of `ROM` ensure the working and base program memory [54]. According to the manual, the robot is capable of speeds of up to $0.6m/s$. A `Kameleon` card ensures the camera control [53]. The robot contains an `RS232` serial port and is controlled by commands issued from an external computer through a serial link.

4.3.2 The computers

The application makes constant use of at leat one computer: the dedicated PC, a Pentium-III at 600 MHz, with 128 MB RAM and a RedHat 8.0 (Linux kernel 2.4) operating system. Besides this computer, different combinations of machines are used, depending on the type of testing.

From our LAN, the machines used are given by table 4.1. The monoprocessor machines are the two `dev`'s (1 and 2) and the four `monox`'s (1 to 4). The biprocessor machines are the two `bip3n`'s (1 and 2) and the seven `iic`'s (1 to 7). The latter are used for local cluster tests. The quadriprocessor machines are the two `quad`'s (1 and 2). We also use computers belonging to our partners for remote control tests. These computers have similar abilities to our monoprocessor machines, with a Pentium-III at around 900 MHz computing power equivalent. `dev2` has been also used for results post-processing.

Table 4.1: The LAN machines used by this project

Machine name	Processor features	Cache (KB)	RAM (MB)	Operating System Linux Redhat and kernel
dev1	1 x Intel PIII 600 MHz	256	256	RH7.2, ker 2.4.9-21
dev2	1 x AMD Athlon 950MHz	256	512	RH7.2, ker 2.4.9-21
bip3n1-2	2 x Intel PIII 933 MHz	256	256	RH7.2, ker2.4.18-24.7.xsmp
quadx1-2	4 x Intel XEON PIII 700 MHz	2048	512	RH7.2, ker 2.4.18-27.7.xsmp
monox1-4	1 x Intel XEON PIV 2.40GHz	512	512	RH8.0, ker 2.4.18-27.0
iic1-7	2 x Intel PIII 700 MHz	256	256	RH7.2, ker 2.4.18-24.7.xsmp

4.3.3 The communication network

There are three types of network technologies used: `LAN`, `ATM` and `Internet`. The `LAN` communication network is used for all the parallel computing tests, while for the remote control `LAN`, `ATM` and `Internet` are all used.

LAN The `LAN` connection is used to communicate with the computers in our laboratory. It consists of `Fast Ethernet` or `Gigabit Ethernet` technologies, ensuring high-speed connections (100 Mbps and, respectively, 1000 Mbps) and very low latencies in packet delivery.

ATM The `ATM` connection is used to communicate with computers at the LORIA Nancy research center. The communication passes through a 60 Mbps ATM P2P network. As the ATM connection offers enough bandwidth, the only latency is due to distance and, sometimes, to unfortunate network traffic congestion.

Internet Establishing an accurate Internet model is one of hard task for today's networking research area. Internet is being treated as a black box, while observations being made over large periods of time are used to obtain average values, especially in telerobotics [71]. Different periods of time on the same day may give surprisingly variable results in terms of network performance (be it based on TCP [1] or UDP [2] protocols) [72]. We make our observations by using monoprocessor nodes of a PC cluster of Salerno University (Italy) through the Internet, while using the TCP protocol for reliable packets transfer.

4.4 Robotic system software architecture

Six months ago, the project was based on modules performing the needed robotic tasks. The application-robot communication was ensured through the use of a sequential library. Furthermore, the laboratory needed to share the two available robots between several users. A new library, `nono`, capable of overlapping computer and robot operations, was started by Herve Frezza-Buet and finished with the help of Olivier Rochel. Right now, at the core of the software architecture stands the `nono` library and its client/server architecture.

The other part of the application is the set of functional modules: panoramic scan, edge detection, landmark detection, optimized triangulation, and trajectory computation. However, they have not been thoroughly tested, and, furthermore, they have yet to be integrated in a unique application.

4.4.1 The client/server architecture

The client/server architecture based on the `nono` library stands at the core of our application. The machine dedicated to robot control runs a `Koala Server` program, which allows for seamless communication to the robot. All user's applications involving the `Koala` robot use are clients of this server.

Robot resources Through the client/server mechanisms, the robot is *transparently* seen by the user as a set of resources (see figure 4.4), e.g. the camera, much like in the hidden-robot described by Kheddar [57] (the graphical interface is the subject of another project, also based on the `nono` library).

Server The server uses the TCP protocol to communicate with the potential clients. Its role is to ensure that robotic resources may be used concurrently and independently by any users. Adding resources may be done for now only when compiling the server, but configuring them is interactive. Finding the available resources is available when `telnet`ing to the server. The server receives commands for the different resources it manages and dispatches them as quick as possible, in the exact order of receival. The server is also responsible for periodic update

---

[1]TCP is a reliable packet delivery protocol. If, due to network congestion or some other cases, packets do not reach the intended destination, they are automatically resent. When dealing with crowded network, resending packets can become very time consuming; thus, this protocol is used by applications where vital data is being carried over the communication network.

[2]UDP is an unreliable packet delivery protocol. No guarantee that sent packets reach their destination is being offered. This protocol is, however, being used for systems that do not need reliability, or cannot afford to wait for a given packet to safely reach the intended destination, like chat systems.
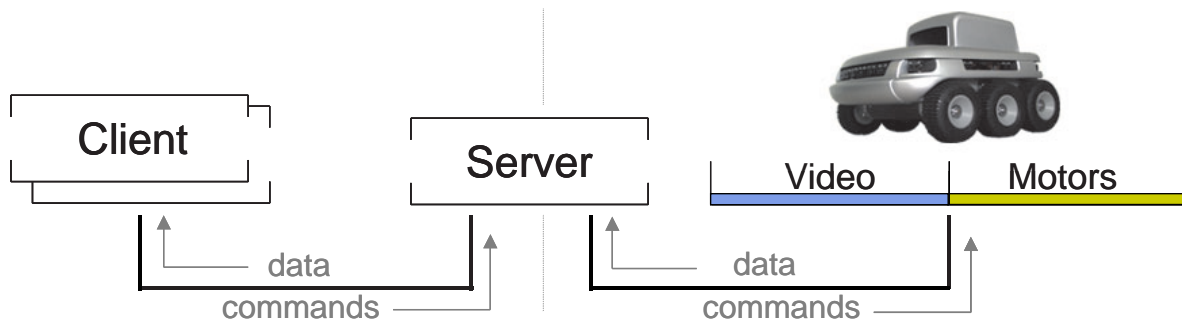
Figure 4.4: The *client/server* architecture. The robot is seen by the client as a set of resources. All commands and data pass through the server.

operations, such as robot status polling and image acquisition. This buffering mechanism ensures that clients are served as soon as they issue their requests, when not specified otherwise. Images may be transferred uncompressed or compressed JPEG to a specified quality.

Remote users   For users outside the Supelec's `LAN`, accessing the robot's dedicated server is possible through the use of `ssh` links. This was our solution to the problem of security policies and computer configurations ordeals.

4.4.2   The navigation module

This module allows the robot to safely navigate between two given points, while avoiding sliding as much as possible. To do that, two industrially-proven algorithms have been considered: one based on Dubins curves and one based on Mouaddib curves. The algorithms both try to compute trajectories which take account of the robot's lack of maneuverability on tight spaces, especially on the minimal curvature[3], and ensure that the robot will end in the desired position having the requested orientation. The Mouaddib trajectories start with a straight line, then an arc of circle followed by a straight line right to the finish (see figure 4.5, b). We considered only the Dubins trajectories that contain an arc of circle from the start and to the end, with a straight line between them (see figure 4.5, a). The difference is in the path length (Dubins trajectories are smaller) and in the number of degrees the robot has to turn (Mouaddib trajectories are generally better on this subject), and the module makes a decision based on a user-defined criteria importance coefficient. Then, the module ensures that the robots gets the appropriate commands to follow the trajectory. The computed trajectory is discretely split in a set of path points. The robot is trying to reach each the current point from this set, starting from the first. When it is very close to achieve that, the target changes to the next point in the set, and so on. This approach is almost like the famous carrot-following for rabbits.

---

[3]The minimal curvature designates the smallest radius on which the robot can follow a circle trajectory.
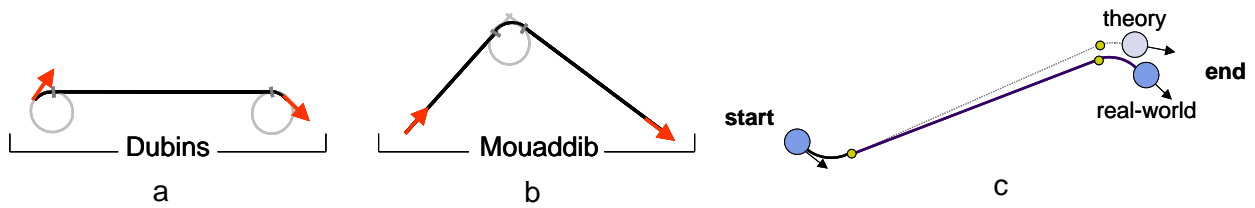
Figure 4.5: Navigation module theoretical an real trajectories. *(a)* A theoretical Dubins trajectory; *(b)* A theoretical Mouaddib trajectory; *(c)* Errors of the open-loop control based on *dead-reckoning*.

While a factory provided Proportional-Integrator-Derivative (PID) engine controller exists, without proper calibration the navigation relies only on dead-reckoning [91] (the only directly observable parameter is the odometer value). Since open-loop control does not allow for error correction (see figure 4.5, c), a self-localization need to be performed at the end of each movement to recover.

The time needed to execute a typical navigation module task depends only on the (mechanical) robot movement duration, as the trajectory computation needs are very small (just a few hundreds of arithmetic and boolean operations). Therefore, this module was not targeted for parallelization.

Unfortunately, this module has risen a great number of problems, both in conception and implementation, and complete rewriting is currently under way.

4.4.3 The self-localization module

This module allows the robot to localize itself in the testbed environment. In order to do this, our application makes use of the PSimilar artificial landmarks system developed by Daniel Scharstein in 1999 [86]. Such a landmark consists of a graphical representation of a PSimilar function and a number, in the form of a bar code. Detecting PSimilar landmarks is very robust with respect to distance, orientation and image quality. The self-localization routine returns a (x, y, θ) tuple, representing, respectively, robot's position on the x-axis, on the y-axis and the orientation against the Ox-axis.

The process of self-localization is required every time the robot has performed some lengthy straight movement or any sort of rotation and is constant in terms of required computing power. First, landmark positions are read from a landmark position file at the beginning of the execution. Then, 17 scans are required to perform a full-circle scan, with a 20° camera orientation change at each new image capture. Three steps are required for each image acquisition and treatment:

1. Two images of different sizes are caught from each position, for far and near landmark detection. We are forced to do this because the *fisheye* camera is distorting very near and very far objects, making them undetectable.

2. Each image is scanned line by line, and a first list of detected landmark is built.

3. The camera is rotated in the new position. The angle step of the camera has been experimentally fixed at 20°. This ensures that the real position is successfully detected while not acquiring too many images.

To complete the self-localization routine, two more steps are required:

1. At the end of all image scans, the landmark list is filtered to eliminate landmarks detected several times (on different images) or not matching with landmarks of the database (detection error). The latter is happening most often when several landmarks are visible during a single scan.

2. Finally, if at least three landmarks have been detected, an optimized triangulation is computed, based on classic triangulation and on Newton-Raphson iterative optimization [88].
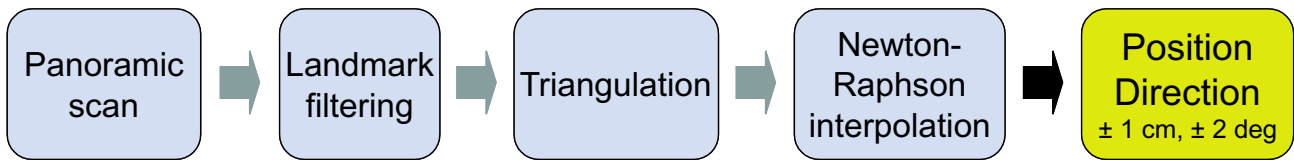
Figure 4.6: The self-localization module control flow. The result is a (x, y, θ) tuple.

In short, a self-localization routine is composed by a panoramic scan, a detected landmark filtering, a triangulation of the filtered results and a Newton-Raphson interpolation (see figure 4.6).

Having the robot localize with 5 up to 20 landmarks, on the $4m \times 2m$ area, revealed certain errors in the self-localization module. However, when being successful, the results were very accurate, with position estimation having an error close to $1cm \times 1cm$, and an orientation error of $\pm 2^o$. As the error on each landmark position is close to $0.5cm$, this final result is satisfying for our applications. However, further verifications and changes should be performed to ensure the robustness of this module.

The time needed to execute a typical self-localization module task depends on the (mechanical) robot camera movement duration, as well as on the (much lengthier) communication and computing tasks times. 17 images are acquired, transferred and treated in a lengthy process. In addition, with the old libraries the time spent in this module where much higher then time spent for the navigation. Therefore, this module was considered the *critical module* of this application and was targeted for parallelization.

4.4.4 The profiling module

Building a profiling module for real-time applications, especially having in mind parallel computing, is no easy task. Indeed, the overhead introduced by the profiler itself should be by far smaller than any expected timings of the real application routines. We have built our profiling system with speed and ease-of-use in mind. It offers means for designating as much as 1000 profiling entries, in a category/item order. An entry is given by a name, and counts for the number of calls and total time. Automatic arithmetic mean averaging is used as a simple statistical tool. A text-based tree-like displaying system has been used. This proved benefic for fast human eye comparison, and was the source of fast solutions for fine-tuning the application. Post-processing tools were also developed for transforming the profiling system's output in numbers-only data files.

4.5 Current mechanical limits

Application timing The first thing after switching to using `nono`-based libraries to control the robot was to measure the mechanical limits. This returned information on time values one could expect to obtain on the self-localization and navigation routines. Ideally, times for these two routines should be as close as possible, to allow one being performed in parallel with the other.

The important timing information for our application are:

Rotation time The minimal time for performing a complete rotation without image acquisition is 4.3 seconds. Moving any faster could possibly harm the mechanical parts of the robot, especially the camera.

Panoramic scan time The minimal time for performing a complete rotation with image acquisition is 7.4 seconds. This time is obtained after performing a full rotation and just the scan, with no actual computation. This is what we call *ideal data acquisition*.

Forward movement speed We achieved a robot speed in straight line of $0.6m/s$, which is the maximum speed defined in the manual. Higher speeds were achieved but the space proved to little to decide whether an accurate control can be obtain at such speeds. It would take about 8 seconds for the robot to get from one corner of our environment to the opposite one (approximately $4.5m$).

Curved movement speed We achieved a robot speed in curves of $0.3m/s$. Attempting to go much faster (close to the nominal maximum speed of $0.6m/s$) made the control system unstable. It would take about 10 seconds for the robot to make the biggest possible *U-turn* (approximately $3m$ in length).

Robotic mechanical testing constraints Thoroughly testing a robot control application in real environments is very difficult. Mechanical problems due to intensive use makes testing more difficult, and sometimes impossible, such as the case of breakdowns. To better preserve the mechanical parts we used delays between consecutive benchmarks, and long pauses between large series of benchmarks. We also preserved a second robot as backup solution.
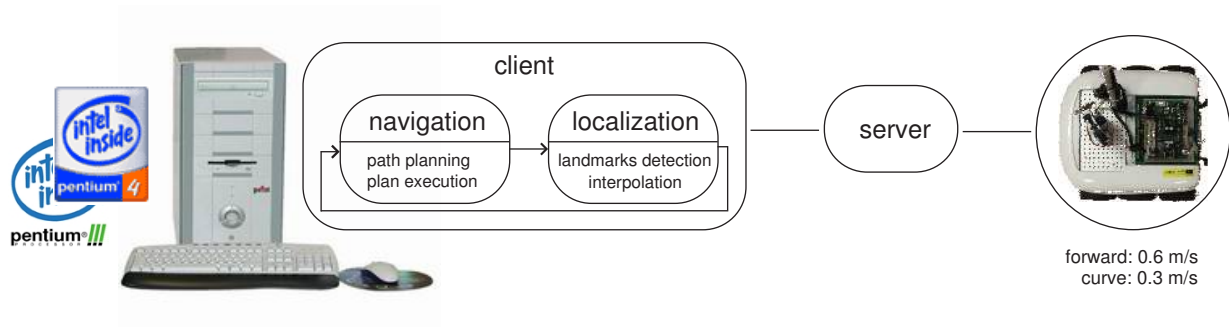
Figure 4.7: Resources diagram for our mobile robotics application.

4.6 Summing it up

In this chapter we presented our application. Its history proves that constant efforts have turned into proven results. Looking at the application structure, the hardware platform allows for parallel and distributed computing testing. Thoroughly testing can prove difficult due to mechanical problems. The software part contains a client/server kernel around which wrap our mobile robotic application modules: the navigation and the self-localization (see figure 4.7). From the two, the self-localization has been proven as the critical module of this application and is subject to parallelization and distributed computing. Our primary goal is to find is scalable speedup can be achieved for this module.

Chapter 5

Robot control algorithms for sequential architectures

5.1   A purely sequential solution

The previous version of this application included a purely sequential algorithm. That algorithm made use of direct and blocking communication with the robot, and its performance was the one of the reasons of the introduction of the client / server model into the system. Currently, such an approach is impossible, so we consider as *purely sequential* the application that uses the `Koala Server`, but issues only blocking commands to control the robot. Algorithms used in the *purely sequential* application are also considered *purely sequential*.

Purely sequential algorithm

1 Blocking init
    1 RealAngle = Robot_GetTurretAngle
    2 Init
    3 Robot_SetTurretAngle(StartAngle)
       CurrentAngle = StartAngle

2 While (CurrentAngle < EndAngle)
    1 RealAngle = Robot_GetTurretAngle
    2 LargeImage = Robot_GetLargeImageGrayscaleData
    3 CurrentAngle += AngleStep
    4 Robot_SetTurretAngle(CurrentAngle)
    5 SmallImage = Scale_and_Grayscale(LargeImage)
    6 Psimilar detection
    7 Filter detected landmarks and update global landmarks list

Figure 5.1: The purely sequential algorithm for sequential machines.

The purely sequential algorithm, presented in figure 5.1, consists of two main steps: data initialization and robot turret positioning and full panoramic scan.

The algorithm begins with the acquisition of the robot turret position from the robot (fig.5.1, point 1.1). It continues with memory allocation, data reset to default values. Being given that the camera can only rotate between $-\pi$ and 0 or between 0 and π radians (cannot pass from $-\pi$ to π directly), there are two possible starting positions: from $-\pi$ and from π. This choice is being made just once in the running application, due to the fact that camera remains at the opposite end from the start when finishing a complete panoramic scan. The algorithm continues with setting the panoramic scan starting angle such as to minimize the initial rotation (fig.5.1, point 1.2). The robot camera is now headed towards the starting angle and the `CurrentAngle` variable is set accordingly (fig.5.1, point 1.3). This concludes the *blocking init* sequence. The real panoramic scan begins (fig.5.1, point 2). It will consist of several image acquisitions and treatments, with the number of image acquisitions depending on the `AngleStep` variable value (usually 20°, in radians). Inside the scan loop, the first action is to get an *(image;angle)* pair. The acquired image is the large, 640×480 grayscale image. This operation is done through the operations points 2.1 and 2.2 in fig.5.1. The `CurrentAngle` value is then updated for the next image scan (fig.5.1, point 2.3) and the camera is rotate through a blocking command to the new position (fig.5.1, point 2.4). The small grayscaled image is then obtained from the large image through a scaling operation (fig.5.1, point 2.5). PSimilar detection on both images follows (fig.5.1, point 2.6), with the outcome of two local detected landmarks lists. These two lists are filtered to eliminate the possible errors and the valid results are updated into the global detected landmarks list (fig.5.1, point 2.7). The list is then processed through the optimized triangulation process to obtain the position-angle tuple.

5.2 Drawbacks of the purely sequential solution

This algorithm has one obvious drawback: it is *slow*. This is due to failing to use the natural parallelism between the control of the external source of data (the robot) and the computing tasks.

5.3 An overlapped approach for sequential machines

After the purely sequential version, the next approach for sequential machines is one that exploits the natural parallel features of robot control and computing tasks. Better performance was expected and obtained.

The overlapped algorithm, presented in figure 5.2, is very similar to the purely sequential algorithm (section 5.1). It has three main parts: first an initialization part, this time with non-blocking robot control commands, then an intermediate routine, to wait for robot to dispatch commands issued in the first part, and, finally, the panoramic scan routine, also containing non-blocking robot control commands.

This algorithm begins with acquisition of the robot turret position from the robot (fig.5.1, pt.1.1). The starting angle is set (fig.5.1, pt.1.2), with the next call commanding the robot

Overlapped algorithm

1. Non-blocking init
   1. RealAngle = Robot_GetTurretAngle
   2. Init, part 1: set StartAngle, based on RealAngle
   3. Robot_SetTurretAngle_nonblocking(StartAngle)
      CurrentAngle = StartAngle
   4. Init, part 2: allocate memory and set default data
2. Robot_WaitTurret
3. While (CurrentAngle < EndAngle)
   1. RealAngle = Robot_GetTurretAngle
   2. LargeImage = Robot_GetLargeImageGrayscaleData
   3. CurrentAngle += AngleStep
   4. Robot_SetTurretAngle(CurrentAngle)
   5. SmallImage = Scale_and_Grayscale(LargeImage)
   6. Psimilar detection
   7. Filter detected landmarks and update global landmarks list
   8. Robot_WaitTurret

Figure 5.2: The overlapped algorithm for sequential machines.

turret to start rotating towards the starting angle. The routine does not wait for the completion of this command (fig.5.1, pt.1.3). Instead, it continues with memory allocation and data reset to default values (fig.5.1, pt.1.4), as the second part of the *non-blocking init* sequence. The second phase contains a call for waiting the robot to complete its rotation task (fig.5.1, pt.2). The actual panoramic scan begins (fig.5.1, pt.3). An *(image;angle)* tuple is acquired, and the new `CurrentAngle` variable value is set (fig.5.1, pts. 3.1 and 3.2). A non-blocking call to the robot camera position set is being issued (fig.5.1, pt.3.4), then the usual image treatment is performed (fig.5.1, pts .3.5-3.7). Finally, the routine waits for the robot to complete its camera movement (fig.5.1, pt.3.8) and a new iteration begins. The triangulation is taking place after all images have been scanned and processed.

5.4   A hyper-threaded approach for sequential machines

Modern sequential machines have usually the ability to run more than a single application thread, making effective use of all the available processing resources. This technique, applied to sequential monoprocessor machines, is called *hyper-threading*. We expected better results from a hyper-threaded approach than for the overlapped version, because we assumed that communication could actually be performed alongside actual computing.

The idea of our approach was to create a `work pool` environment. The `work pool` is supplied with images by a `producer` and has its images dispatched by several `consumer`s. This means that the `I/O` operations are being handled in parallel with the computing parts.
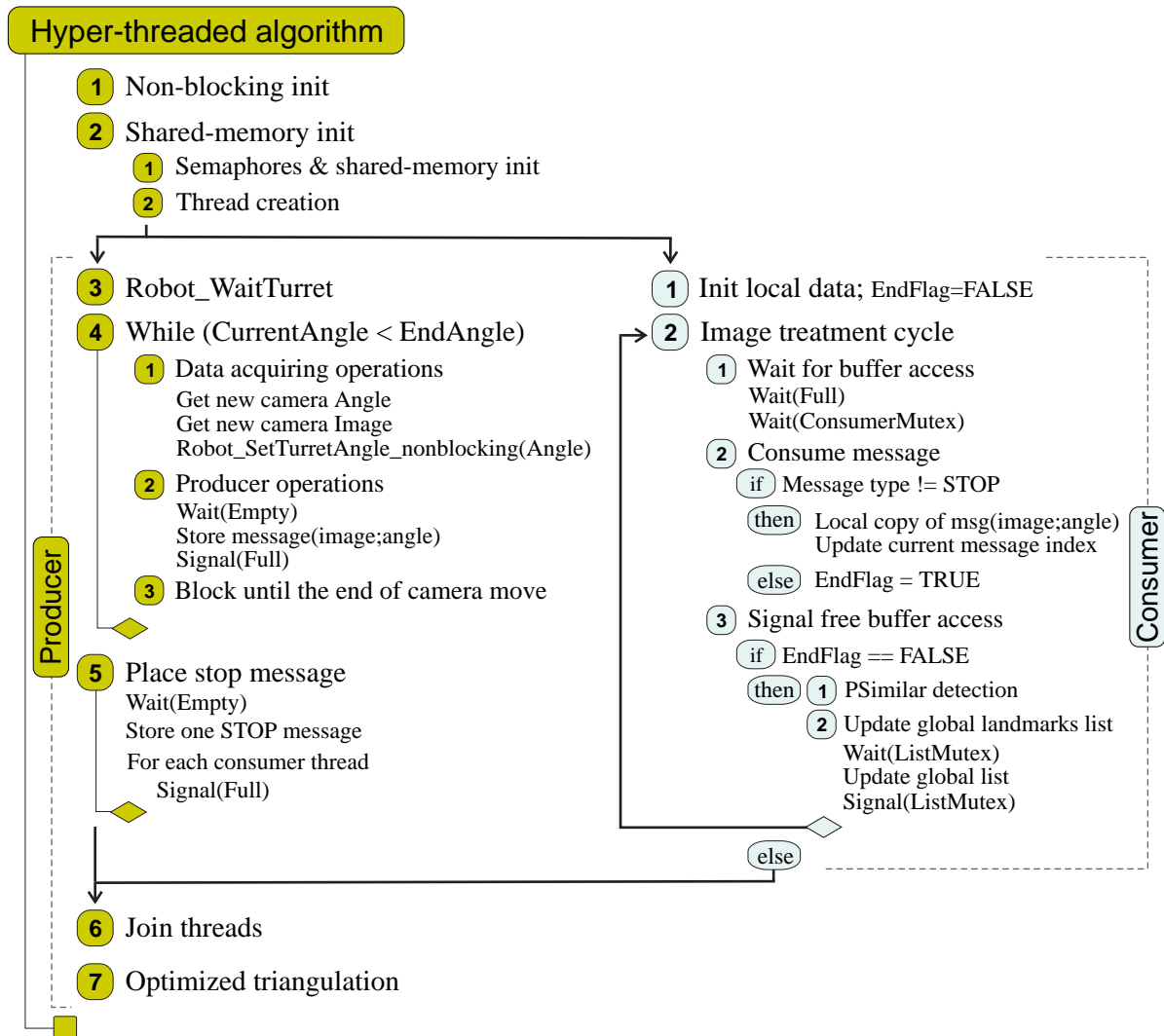
Figure 5.3: The hyper-threaded algorithm for sequential machines.

The hyper-threaded algorithm, presented in figure 5.3, is based on two routines, one for the `producer` and one for the `consumer`s. The `producer` drives the robot camera, gets the images and the associated camera angles, and places them into the working pool. The `consumer`s wait for an image to be available in the working pool, process it and update a shared landmark list. When the panoramic scan is finished, the producer stores a persistent stop message that will be received by each of the consumers ensuring they will all stop as quick as possible.

The producer then waits for all consumers to end, and finishes the localization procedure with an optimized triangulation. The `consumer`s access in turn the shared buffers, using a mutual exclusion variable, `ConsumerMutex`, to avoid the apparition of corrupt data. A common buffers circular index counter is used to indicate the last used buffer. The results from the PSimilar detection are filtered and placed in the global landmarks list, through the use of another mutex, `ListMutex`. This avoids gathering data by the producer at the end of the panoramic scan, while still reducing the local consumer data size or allocation delays.

5.5 Performance on sequential machines

5.5.1 Reference performance choice

Due to the system's client/server base, it is impossible to provide the best sequential time for speedup computation. Instead, performance values for the purely sequential algorithm version have been chosen as reference. We consider that this is the best possible with respect to the sequential property.

Table 5.1: The reference performance, obtained from the purely sequential algorithm. Times and time differences are given in seconds.

Self-localization routine	monox	bip3n	quadx	iic	dev1
T_{min}	**12.49**	14.66	17.30	17.45	18.69
T_{avg}	**12.60**	14.75	17.43	17.69	18.92
$T_{max} - T_{min}$	0.26	0.21	**0.20**	0.57	0.52

The performance of the purely sequential algorithm are presented in the table 5.1. The times show that processor performance index has a direct effect in self-localization performance. The `monox` system obtains the best time (best time $12.49s$, with an average time of $12.60s$). It has by far the most powerful processor and it was expected to be the fastest under the sequential circumstances.

Performance disturbances The differences between results are small, when compared to the total times. The `quadx` systems were the most stable, with a difference of about 1%. As the other systems have performance variations under 4%, we consider that this factor does not vary enough to affect performance considerations.

5.5.2 Tables and analysis for the overlapped approach

Testing procedure The algorithm was first validated on a `bip3n` biprocessor machine. Upon success, the series of tests begun. We tested the overlapped algorithm on the `dev1`, `bip3n`, `quadx`, `monox` and `iic` machines. To ensure results validity, each test was repeated 10 times.

Results   The overlapped approach returned much smaller total times than the reference performance base, as revealed by table 5.2. The best speedup was obtained by the `monox` systems. The value of 1.42 is due to superior `I/O` / computing overlapping of which this processor is capable. Though, even the slower machines, `iic` and especially `dev1`, achieved a respectable speedup of about 1.25. The `quadx` system surprisingly proved to be the most stable, again, after the same performance during the purely sequential tests. Further research is required in order to discover the causes. The differences between comparing the best times and the average times are small (speedups for `monox` are 1.42 and 1.40), but we will prefer using the average values.

Table 5.2: The overlapped algorithm for sequential machines performance.

Self-localization routine	monox	bip3n	quadx	iic	dev1
T_{min}	8.81s	11.11s	12.96s	13.92s	14.84s
T_{avg}	8.98s	11.43s	13.03s	14.09s	14.99s
$T_{max} - T_{min}$	0.43s	0.66s	**0.17s**	0.40s	0.37s
Sequential time (*best times*)	12.49s	14.66s	17.30s	17.45s	18.69s
Speedup (*best times*)	**1.42**	1.32	1.33	1.25	1.26
Sequential time (*avg times*)	12.60s	14.75s	17.43s	17.69s	18.92s
Speedup (*avg times*)	**1.40**	1.29	1.34	1.26	1.26

5.5.3   Tables and analysis for the hyper-threaded approach

Testing procedure   The algorithm was first successfully validated on a `bip3n` biprocessor machine. We performed hyper-threaded algorithm tests on the `dev1` and `monox` monoprocessor machines. The hyper-threaded algorithm performance parameters are the number of consumer threads and the total number of buffers in the work pool. On equal speedups we preferred selecting the version with the lowest performance variation as well as the lowest number of working threads. A test named $1 + x$ denotes having 1 producer thread and x consumer threads.

For `dev1` we varied the number of consumer threads from 1 to 3 and the number of buffers from $1 + p$ to $5 + p$ for p consumer threads. The dev1 machine proved to have the best performance in the $1 + 1$ threads case, as proven by the speedup of 1.61 and the time fluctuation of $0.10s$ (see table 5.3). The impact of the number of work pool buffers variation was not evident and it is not presented here.

For `monox` we varied the number of consumer threads from 1 to 5 and the number of buffers from $1 + p$ to $5 + p$ for p consumer threads. The dev1 machine proved to have the best performance in the $1 + 4$ threads case, with a speedup of 1.58 and the time fluctuation of $0.30s$ (see table 5.4). The impact of the number of work pool buffers variation was not evident and it is not presented here.

General considerations   As a result of observing the performance values, it is now clear that on sequential machines, if the processing power exceeds the requirements, the hyper-threading mechanism is efficient when using several computationally intensive consumer threads associated with one producer. If the processor is not powerful enough, however,

having just one consumer working on the images acquired by one producer is enough and produces good speedup results. Having more than 4-5 consumers does not help in the case of fast computers, because the producer cannot acquire the images fast enough, while the slow computers suffocate from too many computationally demanding tasks.

Acquisition thread + Computing threads	1+1	1+2	1+3
T_{avg}	**11.72s**	11.76s	11.73s
$T_{max} - T_{min}$	**0.10s**	0.28s	0.10s
Sequential execution time	18.92s		
Speedup	**1.61**	**1.61**	**1.61**

Table 5.3: The hyper-threaded algorithm performance on **dev1** machine, with $1 + p$ work pool buffers for p consumer threads.

Acquisition thread + Computing threads	1+1	1+2	1+3	1+4	1+5
T_{avg}	8.19s	8.00s	8.18s	**7.96s**	8.14s
$T_{max} - T_{min}$	0.51s	0.36s	0.48s	**0.30s**	0.47s
Sequential execution time	12.60s				
Speedup	1.54	1.58	1.54	**1.58**	1.54

Table 5.4: The hyper-threaded algorithm performance on **monox** machines, with $1 + p$ work pool buffers for p consumer threads.

5.5.4 Performance on sequential machines

Self-localization The results for the self-localization routine on sequential machines showed that, even if cheap yet powerful PCs are present on today's market, the future of robotics application will continue to depend on parallel computing. Even the `monox` systems, equipped with `Pentium-IV` processors, were performing much better when using the hyper-threaded approach (speedup of 1.42 on the overlapped approach compared to the speedup of 1.58 on the hyper-threaded approach). However, hyper-threading seems enough for such a small application, as the best time obtained by the monox system was $7.96s$, compared to the $7.4s$, the average mechanical time for a complete panoramic scan 5.4. Therefore, more robotic tasks are required to prove that high performance computing is needed for mobile robotic applications.

Global application The complete application speedup depends on the combination of navigation/self-localization calls. We assumed, for reliability reasons, that each navigation step will be followed by a self-localization. Thus, the empirical tests showed that the navigation time is generally equal to 7 seconds per move, with the best self-localization performance on fast machines at about 8 seconds per call. This means that the speedup of the complete application is comparable with the self-localization routine speedup, and it's value can be estimated as:

$$S_{app} = \frac{S_{self-loc} - 1}{2} \qquad (5.1)$$

with average values of 1.21 for the overlapped approach and 1.28 for the hyper-threaded approach.
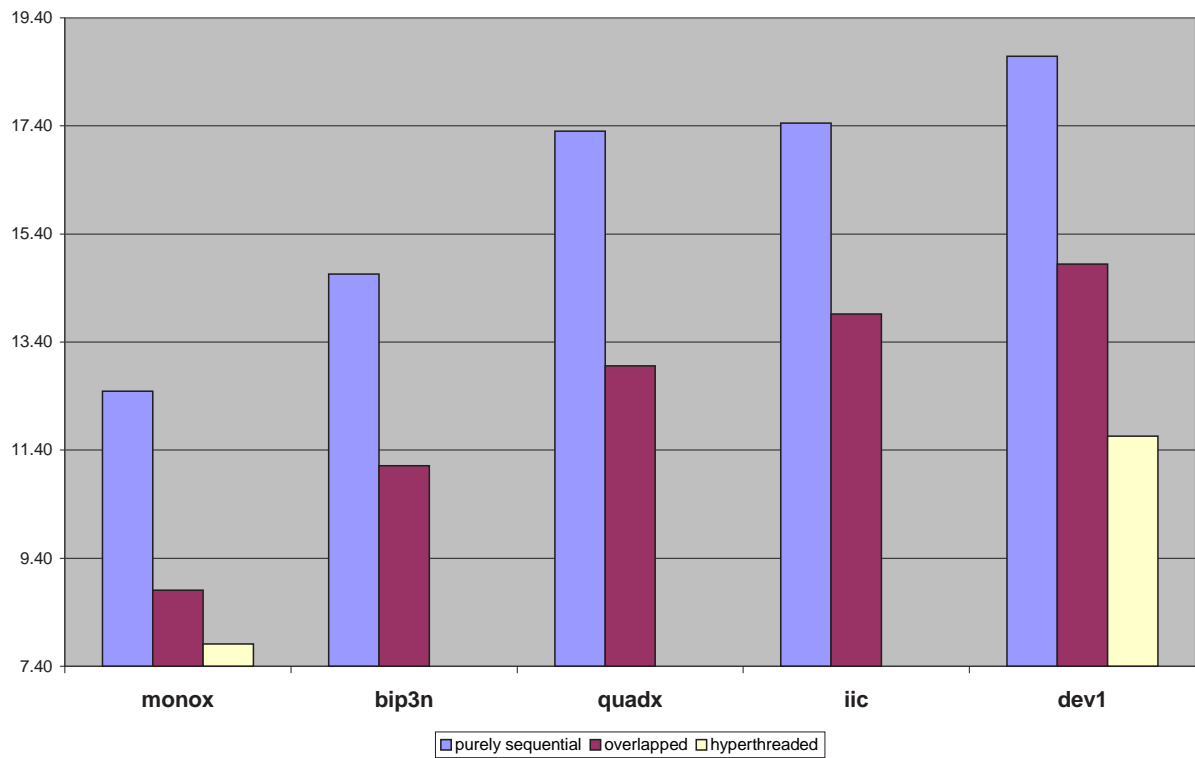
Figure 5.4: Final performance results chart for sequential architectures. The *purely sequential*, *overlapped* and *hyper-threading* versions performances are compared with the mechanical time. Smaller times are better.

Chapter 6

Robot control algorithms for parallel architectures

This chapter present two different algorithms, designed for the two main MIMD machines architectures: shared- and distributed- memory. The application containing the shared-memory algorithm is called throughout this paper the *shared-memory application*. The application built around the distributed-memory algorithm is called throughout this paper the *distributed-memory application*. The application involving any of these two algorithms and the navigation module is called the *parallel application*.

6.1 An optimized multi-threaded approach on shared-memory machines

Shared-memory machines offer a great advantage to the programmer: through the use of threads, all processing units may be used and even hyper-threaded, while still having the advantage of shared resources. Thus, we used the same algorithm implemented for the hyper-threaded approach on sequential machines (section 5.4). The only difference is that, with *shared-memory* machines, there are several processing units instead of a single one on the sequential machines. We used the `pthreads` library to implement the algorithms.

6.1.1 The *work pool* algorithm

The multi-threaded algorithm is based around the `work pool` concept, with a single `producer` thread acquiring images and placing them into the pool, from which several `consumer` threads fetching and processing the images (see figure 6.1). There is no guarantee on the order in which the `consumers` are fetching the images. The multi-threading approach ensures that the producer is continuously working. Suffocating the system, as in the case of hyper-threading, becomes hardly possible, due to the task size. Therefore, a severe decrease in the waiting time is obtained.

To keep synchronization to a minimum while efficiently using the system's memory, we tried work pool sizes from 1 up to the maximum number of images required in a panoramic scan. Multi-threading just under the system's collapsing is the best method to obtain every

drop of performance from that system. We tried a number of consumer threads varying from 1 up to 16. Complete results and a thorough analysis can be found in section 7.2.1.
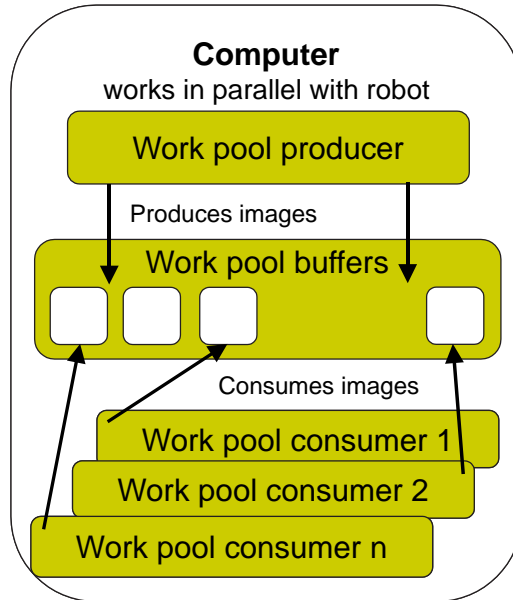
Figure 6.1: The *work pool* algorithm diagram. This algorithm runs on shared-memory systems and is based on the multi-threaded paradigm.

6.1.2 A sample *work pool* algorithm run

A sample run of the *work pool* algorithm is presented in the figure 6.2. The `producer`, marked as P, works continuously and fills the work pool buffers as quickly as possible. The `consumer`s, marked as C, fetch the available images and process them, or wait. No particular image fetching order is ensured. You can see that the second consumer waits for a long time before fetching another image, after processing the first one, and the third consumer restarts working before it.
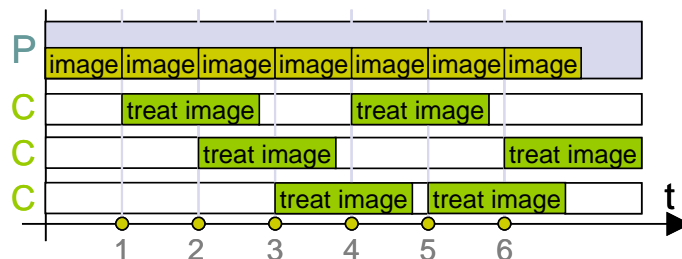
Figure 6.2: Timeline for a sample *work pool* algorithm run.

6.2   An optimized approach on distributed-memory machines

Distributed-machines do not normally offer the programmer resource sharing or easy-to-use program synchronization routines. Instead, one may successfully connect several machines into clusters, single, high performance distributed computing system. Other distributed-memory systems are suited as well. As all the computers have enough power for our kind of tasks, as showed by sequential tests, the interconnection network becomes the greatest time consumer for the self-localization module. This is why we based our approach on avoiding unnecessary messages. We used the `MPI`-based `MPICH` library to implement the algorithm.

6.2.1   The *stick-passing* algorithm

The distributed-memory version is composed of multiple processes that access the robot camera in turn. The process having camera access obtains the next image and the associated camera angle, then passes the camera control to the next process and starts processing the acquired image. The processes which do not have camera control are either treating a previously acquired camera or idle, waiting for the previous processor to give them access rights. This is why we called our algorithm *stick-passing*. Each process updates a global detected landmarks list through a `mutex` mechanism. After completing the panoramic scan and handling all acquired data, the process ranked 0 process final results by filtering the detected landmarks list and then executing trigonometric triangulations plus a Newton-Raphson iterative optimization. For more details see figure 6.3.
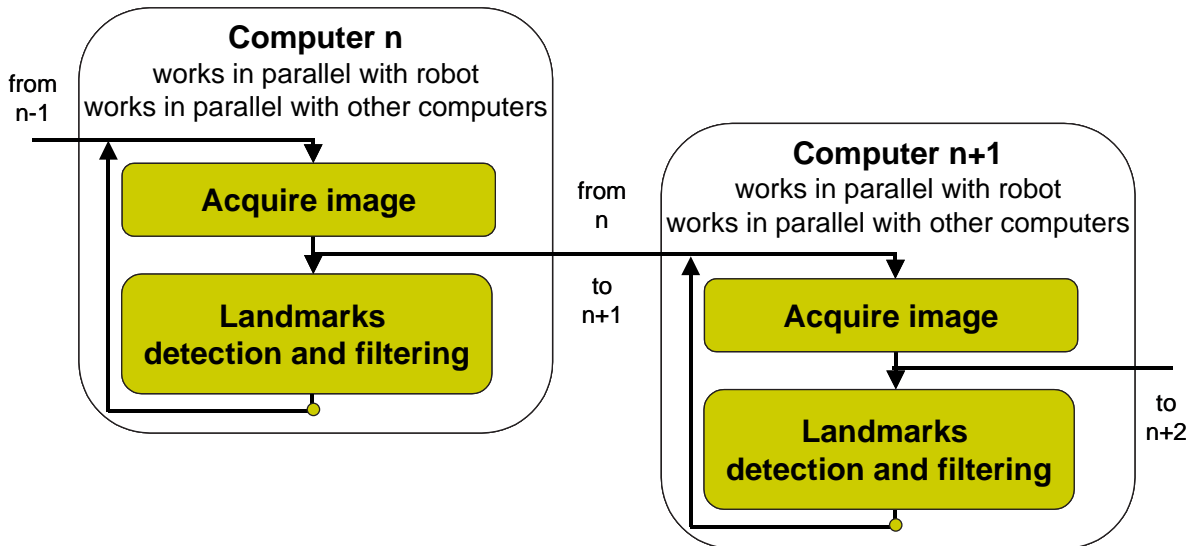
Figure 6.3: The *stick passing* algorithm diagram. This algorithm runs on distributed-memory systems.

Finally, this distributed algorithm is based on a discrete pipeline strategy, which ensures good load balancing on identical processors because each task requires the same amount

of computation (see figure 6.4). Moreover, we avoid useless communication between the processes by directly routing data, i.e. the *(images; angle)* tuples, from the robot server to the next free processor in the pipeline. In the figure one can see 4 MPI processes cycling the execution of an image-acquisition task followed by an image treatment task. You can see how each process starts in an orderly fashion, and how the absence of important performance disturbances (see the *Performance disturbances* paragraph on section 5.5.1 for more) allows the system to have good balancing.
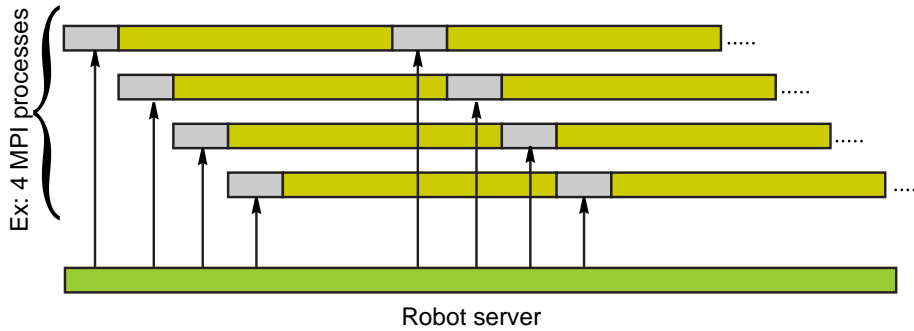
Figure 6.4: The *stick passing* algorithm principle: route data directly from robot server to the right MPI process, and keep each processor busy while there are images to process.

6.2.2 A sample *stick-passing* algorithm run

A sample *stick-passing* algorithm run is presented in the figure 6.5. Process n gets an image, then passes camera control to process $n+1$ and starts processing the image. Until process n handles the access rights, processes $n+1$, $n+2$ and $n+3$ are idle. After receiving the camera control rights from process n, process $n+1$ starts an image acquisition task, at the end of which it will handle the camera control to process $n+2$. The process continues, with process $n+2$ *passing the stick* to process $n+3$, process $n+3$ doing the same for process n and from here the cycle recommences.

Figure 6.5: Timeline for a sample *stick passing* algorithm run.

6.3 Future abilities - the *no-stop-scan* feature

Observing the detailed timing profile of our purely sequential and overlapped approaches, it has become clear that much time is being lost when waiting for the camera to reach its expected final position, before acquiring a new image. This slows down the main self-localization module, in the case of the purely sequential approach, and the same happens with the producer thread, in the case of the overlapped approach, thus limiting the parallelism. Therefore we implemented and experimented another image acquisition technique, based on *double buffering* for image acquisition. Images are regularly and automatically acquired by the `Koala Server`, and, upon request, the client application immediately receives the last image stored on the server. We call this approach the *no-stop scanning technique*, as it cuts the clients' image waiting time down to practically the network transfer time, ensuring much better results. However, right now this method does not produce the expected self-localization results, *first*, because the camera angle cannot be accurately found out during the robot turret movement, and *second*, because the image being acquired during the camera movement is becoming blurry. We call *on-demand* or *normal* image acquisition the wait-for-the-camera-to-stop-then-request-a-new-image procedure. Though our further tests revealed that the PSimilar detection functioned on these images, the first issue prevented us from obtaining valid results. However, the performance indicators show us to what extent this application can speed up in case of a successful mechanics intervention.

Chapter 7

Performance of the parallel algorithms implementation

7.1 Reference performance choice

We used the same reference as for the sequential machines: the `purely sequential` algorithm results (see section 5.5.1 for more details). However, for the optimized algorithm for shared-memory machines we also make a comparison to the overlapped approach, since the `multi-threaded` algorithm uses *non-blocking* computer-robot communication, while for the distributed algorithm for shared-memory machines such a comparison is not reliable, as the `MPI` algorithm does uses only *blocking* computer-robot communication.

7.2 Performance of the optimized algorithm for shared-memory machines

7.2.1 Tables and analysis

Testing procedure   The algorithm was first validated on a `bip3n` biprocessor machine. Upon success, we tested the *work pool* algorithm on the `bip3n`, `quadx` and `iic` machines. The performance parameters are, as in the case of the hyper-threaded algorithm, the number of consumer threads and the total number of buffers in the work pool. We chose the best version upon small speedup, low performance fluctuation and low number of threads criteria. A test named $1 + x$ denotes having 1 producer thread and x consumer threads. For each machine under test, the number of consumer threads was varied from 1 to 5, while the work pool number of buffers varied from $1 + c$ to at least $5 + c$, with c the number of consumer threads. To ensure results validity, each test was repeated 5 times. The impact of the number of work pool buffers variation was not evident and it is not presented here.

   `bip3n` machines had the best performance in the $1 + 4$ threads case, as proven by the speedup of 1.88 and the time fluctuation of $0.22s$ (see table 7.1).

   `quadx` machines had the best performance when running 5 consumer threads. The speedup of 2.17 was the same as for the $1 + 2$ and $1 + 3$ cases, but the time fluctuation in $1 + 5$'s case was jus $0.31s$ (see table 7.2).

   `iic` machines were working best in the $1 + 5$ case. The speedup raised to 2.04; however, the variation in total time was over $0.5s$ (see table 7.3), probably due to the lengthier net-

work connection effect. This makes impossible a proper selection of the best multi-threading parameters for our self-localization task. Bigger computing tasks should be tried to test for the validity of these assumptions.

Acquisition thread + Computing threads	1+1	1+2	1+3	1+4	1+5
T_{avg}	8.26s	8.16s	**7.83s**	**7.84s**	7.86s
$T_{max} - T_{min}$	0.52s	0.66s	0.27s	**0.22s**	**0.23s**
Sequential time *(avg)*			14.75s		
Speedup	1.54	1.81	**1.88**	**1.88**	**1.88**

Table 7.1: The *work pool* algorithm performance on **bip3n** machines, with $1 + p$ work pool buffers for p consumer threads.

Acquisition thread + Computing threads	1+1	1+2	1+3	1+4	1+5
T_{avg}	9.67s	8.04s	**8.03s**	8.08s	8.05s
$T_{max} - T_{min}$	**0.12s**	0.37s	0.42s	0.68s	0.31s
Sequential time *(avg)*			17.43s		
Speedup	1.80	**2.17**	**2.17**	2.16	**2.17**

Table 7.2: The *work pool* algorithm performance on **quadx** machines, with $1 + p$ work pool buffers for p consumer threads.

Acquisition thread + Computing threads	1+1	1+2	1+3	1+4	1+5
T_{avg}	9.71s	8.72s	8.70s	8.67s	**8.64s**
$T_{max} - T_{min}$	**0.05s**	0.50s	0.66s	0.76s	0.58s
Sequential time *(avg)*			17.69s		
Speedup	1.82	2.03	2.03	2.04	**2.05**

Table 7.3: The *work pool* algorithm performance on **iic** machines, with $1+p$ work pool buffers for p consumer threads.

General considerations It is clear that the multi-threading approach is suitable for shared-memory machines. By avoiding lengthy synchronization, our `work pool` algorithm achieved speedups around 2, which, given the small size of the computing task, are considered satisfactory. However, the problem size should be increased for better performance testing conditions on powerful machines.

7.2.2 Speedup on shared-memory machines

In this chapter, a complete table of the work pool performance results (see table 7.4). Results on `dev1` and `monox` machines have been included because, *one*, these machines may be considered lower-class shared-memory machines, and, *two*, the hyper-threaded and multi-threaded algorithms are exactly the same.

To further emphasize the General considerations of the previous section, table 7.4 and figure 7.1 clearly show that the hyper- and multi- threaded approach are both suitable for this application. The `work pool` algorithm led to speedups of around 1.5, for the monoprocessor

machines, and around 2, for multiprocessor machines. As opposed to the results obtained a year ago, currently the problem size is becoming too little for the available processing power. However, the problem size is not scalable. This means that to achieve scalable speedup, the number of robotic tasks should be increased.

Machine	# of threads				
type	1	2	3	4	5
dev	1.61	1.61	1.61	-	-
bip3n	1.79	1.81	1.88	1.88	1.88
quadx	1.80	2.17	2.17	2.16	2.17
monox	1.54	1.58	1.54	1.58	1.55
iic	1.82	2.03	2.03	2.04	2.05

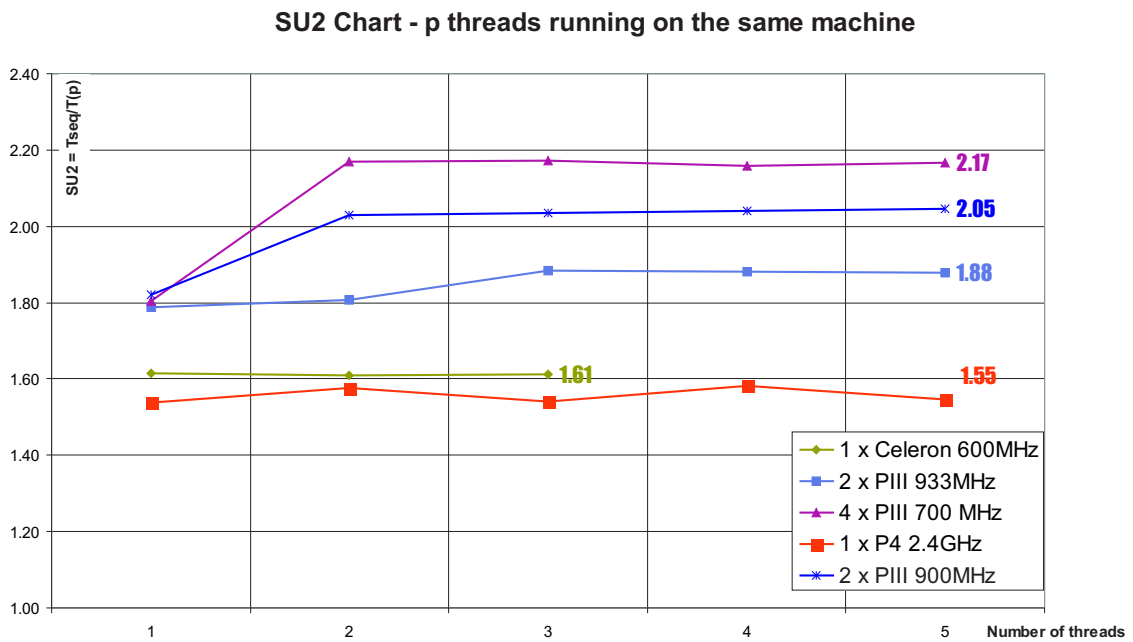Table 7.4: The speedup obtained by the *work pool* algorithm on shared-memory machines.

Figure 7.1: SU_2 for several threads running on the same machine.

# of MPI	Machines							
processes	**dev1**	**2·dev**	**bip3n**	**2·bip3n**	**quadx**	**2·quadx**	**monox**	**2·monox**
1	18.67s	-	14.69s	-	16.85s	-	12.55s	-
2	11.58s	14.59s	10.91s	8.03s	8.80s	8.93s	8.61s	8.64s
3	12.23s	10.93s	9.20s	8.09s	8.37s	8.46s	8.56s	8.89s
4	11.84s	8.68s	9.15s	8.08s	8.41s	8.54s	8.65s	8.70s
5	11.33s	8.75s	9.14s	8.18s	8.39s	8.38s	8.84s	8.80s
6	11.37s	8.89s	9.04s	8.11s	8.40s	8.46s	8.71s	8.71s
7	11.32s	8.90s	8.91s	8.09s	8.39s	8.42s	8.64s	8.61s
8	11.34s	8.74s	8.98s	8.04s	8.38s	8.38s	8.60s	8.63s

Table 7.5: The *stick passing* algorithm times on **dev**, **bip3n** and **quadx** machines, with the number of MPI processes varying from 1 to 8.

7.3 Performance of the optimized algorithm for distributed-memory machines

7.3.1 Tables and analysis

Testing procedure   The algorithm for distributed-memory machines was first validated on a `bip3n` biprocessor machine. Upon success, we thoroughly tested the *stick passing* algorithm on `dev1`, `bip3n`, `quadx`, `monox` and `iic` machines, as well for omogenous cluster. The performance parameters are the number of MPI processes. We tested the `dev1` and `bip3n` machines with a number of MPI processes varying from 1 to 8, while tests for the `quadx`, `monox` and `iic` machines used a number of 1 up to 16 MPI processes. A number of 5 tests were performed for each parameters configuration, to a total of around 1000 tests. The complete results are available, but will not all be displayed in this report, due to space contraints.

Tests on single machines and small clusters   Table 7.5 shows the self-localization routine times for 1 to 8 MPI processes run on the same machine, as well as on small homogenous clusters formed with two machines of the same kind.

The `bip3n` machine offered the most clear view of what happens when parallelizing the self-localization task for distributed-memory machines. With just 1 MPI process running, the times are similar to the *purely sequential* version ($14.69s$). When adding more MPI processes, the module execution achieves an increasing speedup ($10.91s$ for 2 MPI processes, $9.20s$ for 3 and 9.15 for $4-5$ MPI processes), up to a point where the speedup remains fairly constant (about $9s$ for 6+ MPI processes). Because of the task reduced *dimensionality*, it is of no use to add more MPI processes after the speedup stabilizes. Note that, due to the *load balancing* features of the algorithm, the systems were not over-stressed by the idle MPI processes, no matter their number. Another big issue in the MPI approach using `MPICH` was the system's *stability*. For example, on the same `bip3n` machines, performance variations of $5-29\%$ were encountered, with the most expected value around 6.5% (see table 7.6). Stability increases when using more MPI processes. For this application, using at least 4 MPI processes, no matter the cluster configuration, ensured stability. The minimum time for a complete self-localization was $8.91s$.

The 2 `bip3n` machines cluster obtained, as expected, better results than the 1 `bip3n` ma-

MPI processes	2	3	4	5	6	7	8
T_{avg}	10.91s	9.20s	9.15s	9.14s	9.04s	8.91s	8.98s
$T_{max} - T_{min}$	**3.21s**	**0.77s**	**0.61s**	**0.52s**	**0.43s**	**0.76s**	**0.66s**
Variation	29.4%	8.4%	6.6%	5.7%	4.8%	8.5%	7.3%
Speedup	1.02	1.21	1.21	1.22	1.23	1.25	1.24

Table 7.6: The *stick passing* algorithm performance variation on bip3n.

chine tests (see table 7.5, column $2 \cdot bip3n$). However, the speedup for this system stabilized immediately after running 2 MPI processes. This is due to the problem size. However, further testing with more robotic tasks is not necessary, since the speedup curves are well known. The minimum time for a complete self-localization was 8.04*s*.

The `dev1` machine presented similar results as the `bip3n` machines, only less accentuated (see table 7.5, column **dev1**). This is due to the relatively small processing power of this machine, which renders null the use of several MPI process. The average time obtained for 3 MPI processes, 12.23*s* was unexpectedly high and this is the outcome of an unusually large variation in the module performance. We explain this by the combined facts of running several MPI processes on a slow machine with a multitasking operating system, which might have been performing some other work at that point, and by the instability of the MPICH library performance over Linux. The minimum time for a complete self-localization was 11.34*s*.

Having 2 `dev` machines provided a good time and, subsequently, speedup improvement when compared to the single `dev1` machine (see table 7.5, column $2 \cdot dev$). The small cluster obtained similar results to the `bip3n` system, with a similar time and speedup curve. The minimum time for a complete self-localization was 8.74*s*.

Finally, the `quadx` machines obtained a speedup curve similar to `bip3n` machines, with an optimal time / performance variation combination achieved for 5 MPI processes, while the speedup for 2 `quadx`, 1 `monox`, and 2 `monox` systems stabilized immediately after running 2 MPI processes (see table 7.5, columns $2 \cdot quadx$, *monox* and $2 \cdot monox$). Again, the latter means that the task size should be increased to make effective use of these machines. Minimal times for a complete self-localization on `quadx` and 2 `quadx` machines were around 8.40*s*, while on `quadx` and 2 `quadx` machines they were around 8.60*s*.

Tests on clusters of workstations Homogenous clusters of workstations have thoroughly been tested. The clusters were formed by 2 `bip3n`[1] machines, or 2 `quadx`[2] machines, or 2 up to 4 `monox`[3] machines, or 2 up to 7 `iic`[4] machines. The number of running MPI processes varied from 1 to 16, with 5 tests per *machine/number of MPI processes* configuration, to

---

[1]The **bip3n** machines are Intel PIII 933 MHz biprocessors, with 256 KB cache, 256 MB RAM and running a RH7.2, ker2.4.18-24.7.xsmp operating system.

[2]The **quadx** machines are Intel XEON PIII 700 MHz quadriprocessors, with 2048 KB cache, 512 MB RAM and running a RH7.2, ker 2.4.18-27.7.xsmp operating system.

[3]The **monox** machines are Intel XEON PIV 2.40GHz monoprocessors, with 512 KB cache, 512 MB RAM and running a RH8.0, ker 2.4.18-27.0 operating system.

[4]The **iic** machines are Intel PIII 700 MHz biprocessors, with 256 KB cache, 256 MB RAM and running a RH7.2, ker 2.4.18-24.7.xsmp operating system.

ID	Machine	T_{total}	SU_1	SU_2	SU_3
1	**dev1**	18.67s	1.00	1.01	0.80
2	**2·dev**	14.59s	1.28	1.30	1.03
3	**bip3n**	14.69s	1.00	1.00	0.78
4	**2·bip3n**	8.02s	1.83	1.84	1.42
5	**quadx**	16.85s	1.00	1.03	0.77
6	**2·quadx**	8.93s	1.89	1.95	1.46
7	**monox**	12.55s	1.00	1.00	0.72
8	**2·monox**	8.64s	1.45	1.46	1.04
9	**3·monox**	8.69s	1.44	1.45	1.03
10	**4·monox**	8.78s	1.43	1.44	1.02
11	**iic**	17.22s	1.00	1.03	0.82
12	**2·iic**	9.62s	1.79	1.84	1.46
13	**3·iic**	9.38s	1.84	1.89	1.50
14	**4·iic**	9.34s	1.84	1.89	1.51
15	**5·iic**	9.31s	1.85	1.90	1.51
16	**6·iic**	9.33s	1.85	1.90	1.51
17	**7·iic**	9.30s	1.85	1.90	1.51

Table 7.7: The complete *stick passing* algorithm for distributed-memory machines performance results. T_{total} is the average time for p MPI processes running on p machines (one per machine). $SU_1 = T_{MPI,1P}/T_{MPI,NP}$, $SU_2 = T_{pseq}/T_{MMPI,NP}$, $SU_3 = T_{ovr}/T_{MMPI,NP}$, where $T_{MPI,1P}$ is the average time for a self-localization routine on 1 MPI process running on 1 machine, $T_{MPI,NP}$ is the time for n MPI processes on p machines, T_{pseq} is the time for the *purely sequential* version and T_{ovr} is the time for the *overlapped* version.

ensure that every configuration was seamlessly tested. A heterogenous cluster formed by the 2 dev[5] machines was also tested, with 1 up to 8 MPI processes. The speedups obtained for having p MPI processes running on clusters with p machines are presented in table 7.10. As in the case of speedup limitation when adding more running MPI processes on the same machine, having more machines in a cluster increases speedup up to a certain point, where it stabilizes. Again, due to *minimal communication* between MPI processes, the systems's overhead is maintained constant; thus, the speedup stabilizes at the highest value. Speedup is achieved for all systems, with best effective values running from 1.28, such as the case for the dev cluster, to 1.89, such as the case for quadx cluster. The iic cluster also achieves a 1.85 speedup.

7.3.2 Speedup and parallelizability on distributed-memory machines

Speedup The SU_2 column of table 7.10 shows that the stick passing algorithm achieves speedup. This means that more MPI processes running on more machines guarantee better times than in the case of the *purely sequential* algorithm.

---

[5]The **dev1** machine is a Intel PIII 600 MHz monoprocessor, with 256 KB cache, 256 MB RAM and running a RH7.2, ker 2.4.9-21 operating system. **dev2** machine is a AMD Athlon 950MHz monoprocessor, with 256KB cache, 512MB RAM and running a RH7.2, ker 2.4.9-21 operating system.

Parallelizability   The SU_1 column of table 7.10 shows that the `stick passing` algorithm is parallelizable. This means that having more MPI processes on more machines guarantees better times than in the case of 1 MPI program on 1 working machine. This was not obvious, for communication overhead and task's scalability might have been blocking factors. However, the small dimension of the robotic task enforced a stabilization of the parallelizability speedup values, proportional with the processing power of each machine (see figure 7.2).

The `dev` monoprocessor machines speedup limited at 1.30 (see table 7.10, IDs 1 and 2), because they do not own enough computing power to make use of alt the MPI system's advantages. Also, the 2-machines cluster was heterogenous, which might lead to a depreciation of the algorithm performance, with the more powerful processor having to wait for the other, in the case of 2 MPI processes for 2 machines.

The `bip3n` biprocessors performed much better than the two `dev`s, with the speedup limiting at 1.84 (see table 7.10, IDs 3 and 4).

The `quadx` quadriprocessors performed similarly, with a speedup limit of 1.95 (see table 7.10, IDs 5 and 6). These machines had the best speedup increase for the *stick passing* algorithm. An explanation is the superior cache size, with another being the MPICH library greater performance stability.

The `monox` systems limited after the use of 2 machines to a speedup close to 1.45 (see table 7.10, IDs 7 and 10). This is due to the task size. Thus, to make use of the full computing power capabilities of today's computers, it is very important to have a bigger robotic application, with several tasks having the same computing demands as the self-localization based on vision in our application.

Finally, the `iic` machines had an interesting performance limit since the 3 MPI processes tests (1.89), with a very slow increase until the 5 MPI processes (1.90). The 1 and 2 MPI processes tests revealed results very close to the `bip3n` machines, so we may speculate that `iic` results for 3+ MPI processes are the expected performance results for `bip3n` machines, for this application.

Comparison with the overlapped version   A comparison with the purely sequential algorithm was presented in the previous two paragraphs, but one with the overlapped algorithm is needed to complete the picture. Therefore, the SU_3 column of table 7.10 displays the ratio between the times obtained for having p MPI processes running on p machines on the times obtained for the overlapped algorithm. It is clear that, in the 1 machine case, it is always better to choose the overlapped algorithm (SU_3 is around 0.80 for each of the tested systems - the MPI process runs at about 80% of the performance of the overlapped algorithm). This was expected, due to two factors: *one*, the conception of the MPI algorithm does not make use of the overlapping mechanism, and , *two*, the MPI system has at least a communication overhead. However, once more machines have been used, the MPI algorithm obtains the lead, a marginally edge in the case of the `dev` and `monox` systems (SU_3 is 1.03 and, respectively, around 1.02), and a with good margins for the other systems (SU_3 is above 1.40).

7.3.3   Isoefficiency considerations on distributed-memory machines

It is of no use to discuss isoefficiency matters for mobile robotics tasks taken individually. Indeed, these tasks are very small or, at best, small, and nothing can be done to augment their size. However, looking at the robotics tasks as a whole, there is a big difference. In the case of mobile robotics applications that target more than just navigation, such as NASA
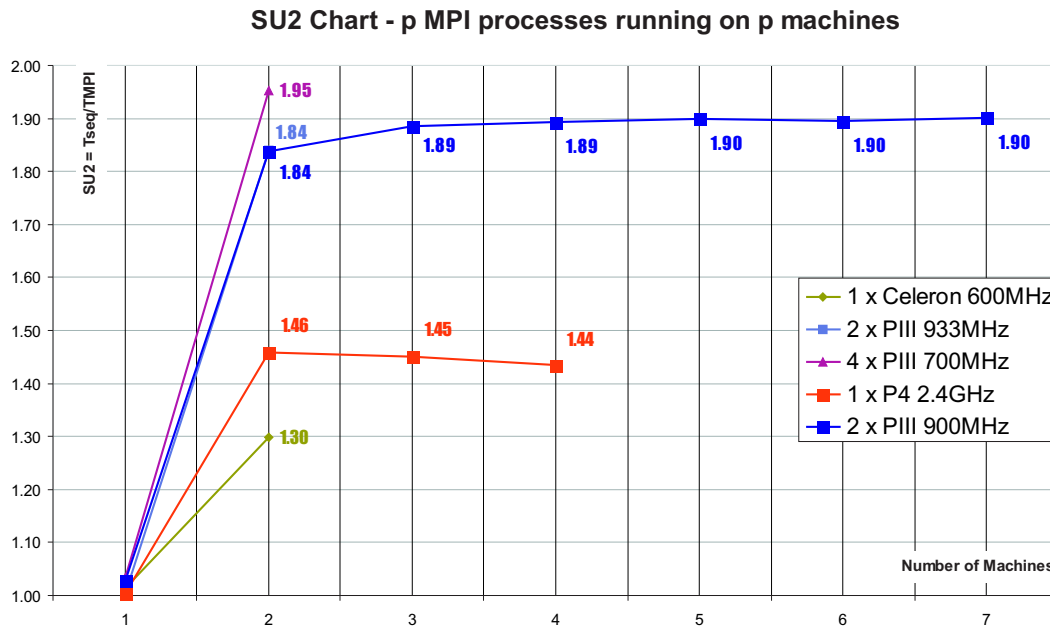
Figure 7.2: MPI algorithm performance on clusters: the SU_2 values for p MPI processes running on a p machines distributed-system.

Mars missions, we can see that the number of computationally demanding tasks grows with application's complexity. We think that robotics applications with large data acquiring and processing needs have a good scalability, with more computing tasks being solved *in-place* or in *real time*, through the use of distributed computing resources. For each resource added to the system, a task or a set of tasks that are suited for that resource, be it a multiprocessor or a sequential machine, should be added.

7.3.4 MPICH performance issues on Linux

The MPICH library implementing the MPI 1.2 standard for Linux raises several stability and performance variation problems. Using over 10 MPI processes was difficult, because the MPI system's way of handling TCP messages. We think that the delay induced by MPICH was bigger than the Linux's default limit (200 ms) and produced unnecessary resending. This explains the big time fluctuation for the complete self-localization module.

7.4 Performance for the *no-stop-scan* feature: analysis and prediction

Results analysis The results show that, in the case of *double buffered* image acquisition, the overlapped version returns average results of $10.02s$ for a `bip3n` biprocessor system, and

of 11.74s for a `quadx` quadriprocessor system. We do this because, Being given that the *no-stop-scan* feature is only a possibility, we would like to find out how the best times for this version would look like. Therefore, we chose the *overlapped* algorithm[6] results as the *reference performance choice* and we measure times obtained from the **??** and **??** algorithms. As a general fact, comparing speedups for the *normal* and *double buffered* versions, we see that the *double buffered* version achieves greater values when stressing the system with more threads, for the *work pool* algorithm, or more MPI processes, for the *stick passing* algorithm.

Acquisition thread + Computing threads	1+1	1+2	1+3	1+4	**1+5**
T_{avg}	8.18s	7.95s	**6.44s**	**6.45s**	**6.46s**
$T_{max} - T_{min}$	0.30s	1.23s	0.34s	**0.29s**	**0.26s**
Speedup	1.23	1.26	**1.56**	**1.55**	**1.55**
Speedup for *normal* image acquisition	1.38	1.40	1.46	1.46	1.46
Overlapped execution time	10.02s				

Table 7.8: The multi-threaded localization routine performance on **bip3n**, based on *double buffering* mechanism and image acquisition during camera move.

Acquisition thread + Computing threads	1+1	1+2	1+3	1+4	**1+5**
T_{avg}	9.56s	6.55s	6.65s	6.55s	**6.51s**
$T_{max} - T_{min}$	0.08s	0.30s	0.59s	**0.19s**	**0.19s**
Speedup	1.23	1.79	1.77	**1.79**	**1.80**
Speedup for *normal* image acquisition	1.35	1.62	1.62	1.61	1.62
Overlapped execution time	11.74s				

Table 7.9: The multi-threaded localization routine performance on **quadx**, based on *double buffering* mechanism and image acquisition during camera move.

MPI processes	2	3	4	5	**6**	7	8
T_{avg}	10.56s	8.18s	7.29s	7.26s	**7.10s**	7.36s	7.32s
$T_{max} - T_{min}$	**0.27s**	0.93s	0.36s	0.71s	**0.31s**	**0.24s**	0.51s
Speedup	0.95	1.23	1.37	1.38	**1.41**	1.36	1.37
Speedup for *normal* image acquisition	1.02	1.21	1.21	1.22	1.23	1.25	1.24
Overlapped execution time	10.02s						

Table 7.10: The MPI localization routine performance on **bip3n**, based on *double buffering* mechanism and image acquisition during camera move.

***Work pool* algorithm results**   Tests for this algorithm were run on `bip3n` biprocessor and `quadx` quadriprocessor machines. These were the more powerful multiprocessor machines, thus more susceptible to make full use of the data incoming more rapidly due to the *double buffering*[7] image acquisition method. Best times for both systems fall in the area of 6.5s,

---

[6]quick reminder: the *overlapped* algorithm uses just sequential computing tasks, but takes advantage of the natural parallelism between the computer and robot tasks by overlapping them

[7]quick reminder: the *double buffering* image acquisition method consists of the server constantly fetching and storing robot camera images; a client's request is therefore dispatched immediately by sending the last server-side stored image.

which is, as expected, lower than the mechanical limits of $7.4s$. This is due to the fact that the *double buffering* method does not have to wait for a new image to be acquired, which is equal to an overlapping of robot's tasks, between the image acquisition and the camera rotation. The performance variation was still around 4%, with actual minimum values of $0.26s$ for the `bip3n` system and $0.29s$ for the `quadx` system (see tables 7.8 and 7.9).

***Stick passing* algorithm results**   Tests for the *stick passing* algorithm were run after the *work pool* algorithm tests were concluded. This is why we tested just the `bip3n` machine. We point out the best time for this system, $7.10s$, with a performance variation of $0.31s$, which is obtained for 6 MPI processes (see table 7.10). We point out two aspects: *one*, this time is lower than the current mechanical limits of the application, like the case for the *work pool* algorithm using the same *double buffering* image acquisition technique, and, *two*, the best time for a distributed-memory algorithm is worse than the best time for a shared-memory algorithm, both executed on the same machine.

Prediction   We predict that both shared-memory multiprocessor machines and distributed-memory systems have enough computing power to make full use of fast image acquisition; it is now a robotics engineer's task to provide the necessary camera control for this task.

7.5   Performance on shared- and distributed-memory machines

Shared-memory vs. Distributed-memory   Watching the best times obtained by the *work pool* and the *stick passing* algorithms, it is clear that the best time for a distributed-memory algorithm is worse than the best time for a shared-memory algorithm, when both executed on the same machine (see figure 7.3 for complete comparison of shared- vs. distributed-memory self localization run times). We conclude that whenever possible, it is preferable to use the multi-threading approach than MPI. By whenever possible we mean when the computing power of the shared-memory machine is equal to the distributed-memory machine's computing power, or when the computing task size is small enough for both the shared- and the distributed-memory machine to handle.

Cost issues

Supercomputers are not for mobile robotics   The cost of dedicated parallel machines, especially supercomputers, is one important factor in the lack of use for these machines in the mobile robotics world. Indeed, having to pay far more for the computing part then for the robotics part in a robotics project seems occurred. Research oriented to using less computationally-intensive tasks, or even less tasks, with the result of having smaller and less spectacular application.

Off-the-shelf computers are cheap   The shared-memory machines and the clusters used for this application were all cheap, off-the-shelf computer, with price ranging from $USD1000$, for the `dev1` monoprocessor machine, to $USD15000$, for the `quadx` quadriprocessor machines. This is approximately the cost of the complete robotic system, including the robot, camera, tools and wires. We therefore think that using such systems for improved robotic application computing capacity is well worth the costs.
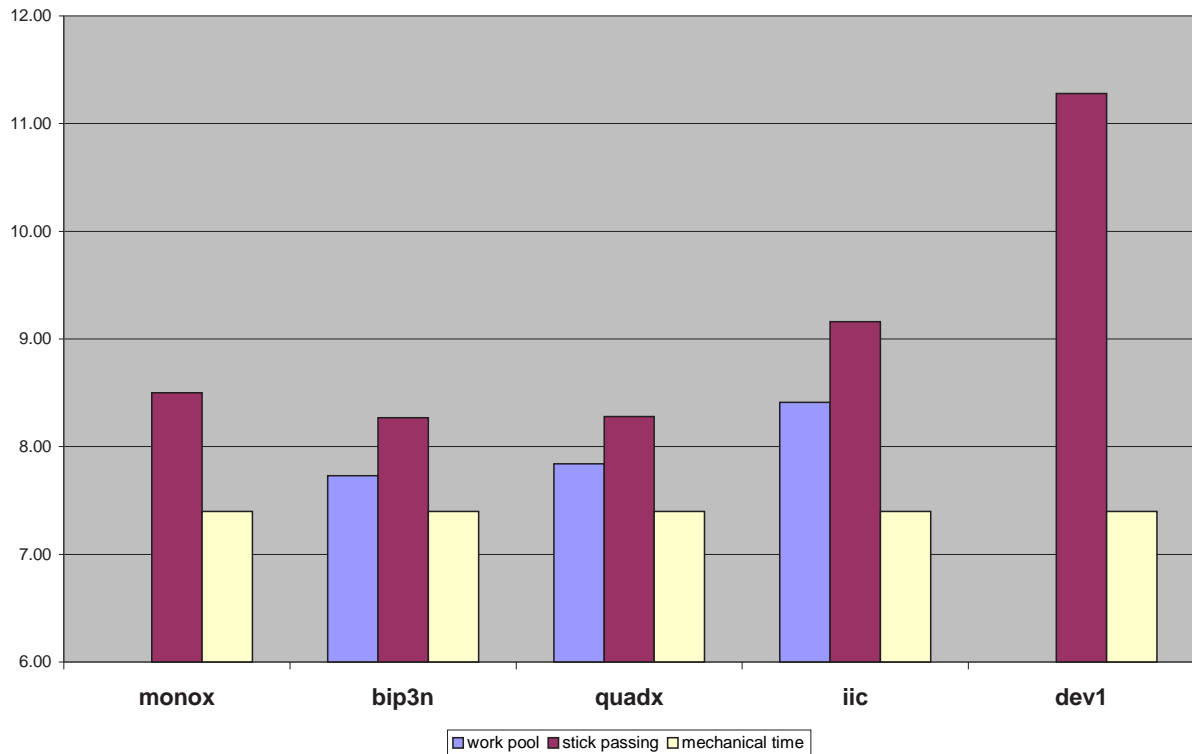
Figure 7.3: Final performance results chart for shared- and distributed-memory architectures. The *work pool* and *stick passing* versions performances are compared with the mechanical time. Smaller times are better.

Global application   Like in the sequential machines performance analysis, the complete application speedup depends on the combination of navigation/self-localization calls. The empirical tests showed that the best self-localization performance on fast machines again, at about 7.7 seconds per call. This means that the speedup of the complete application has average values similar to the sequential performance results, for very slow or very fat machines, but significantly better, i.e. around 1.40 vs. around 1.15, for the multiprocessor machines.

Chapter 8

Experiments with mobile robots remote control

8.1 Performance evaluation

For a number of years now, industrial and military applications control modules make extended use of specialized networking technologies [20]. Challenging problems in control of network-based systems are variable time-delay and data packet losses. The *variable time-delay* problem affects the application performance and, depending on the application, its results. Telesurgery is a robotics application field where time-delays may produce most severe consequences. The *packet loss* not only degrades the system's performance, but also, if not dealt with, destabilize it.

We are interested in three three main aspects of mobile remote controlled robotics applications:

1. **Availability**
   The networking problems should not affect the robot's perceived presence, e.g. the user should be able to control the robot no matter if some parts of the network are rendered inactive. Dealing with this problem includes taking into consideration the effects of time-delay and packet loss. However, verifying this property's limits is more of a networking challenge.

2. **Reliability**
   Ideally, the robotic application should perform its tasks no matter what networking problems occur. Handling this problem includes, again, dealing with time-delay and packet loss, but also defining backup plans for the undesired case of communication interruption. Imagine a mobile robot that relies on remote computing to navigate. This robot should contain some on-board control routines that would at least stop it before crashing into the walls, if the remote machine fails to deliver the required data in time.

3. **Time performance**
   Finally, a mobile robotic application should target low execution times. This is true for most of the world's applications, but real-time applications must draw as much time performance as possible.

We begin by focusing on the third issue, *time performance.* Doing so allows for establishing a good performance base for our remote tests and returns valuable information upon the problems that might arise for this specific application.

8.2 A first approach

Testbed   For *remote control applications*, using a core client/server programming paradigm comes at hand. The testbed is therefore based on the same client/server architecture, presented in section 4.4.1. The robot is transparently seen as a set of resources by the clients that connect to *Koala Server*. The communication networks under test are presented in section 4.3.3. Tests are performed on the *local area network* (LAN), with the computers in the Supelec, Metz (France) laboratory, over an ATM Peer2Peer connection, with computers from the LORIA Research Center, Nancy (France), and over the Internet, with computers from the University of Salerno (Italy) (see figure 8.1). The communication distances are under $100m$, with as much as 4 switches, for the LAN, around $50km$, over an `ATM switch`, for Nancy, and over $1000km$, for the `Internet` connection to Italy. The application times are expected to increase, thus the outcome will be a subunit speedup, or *slow-down*, of the application.
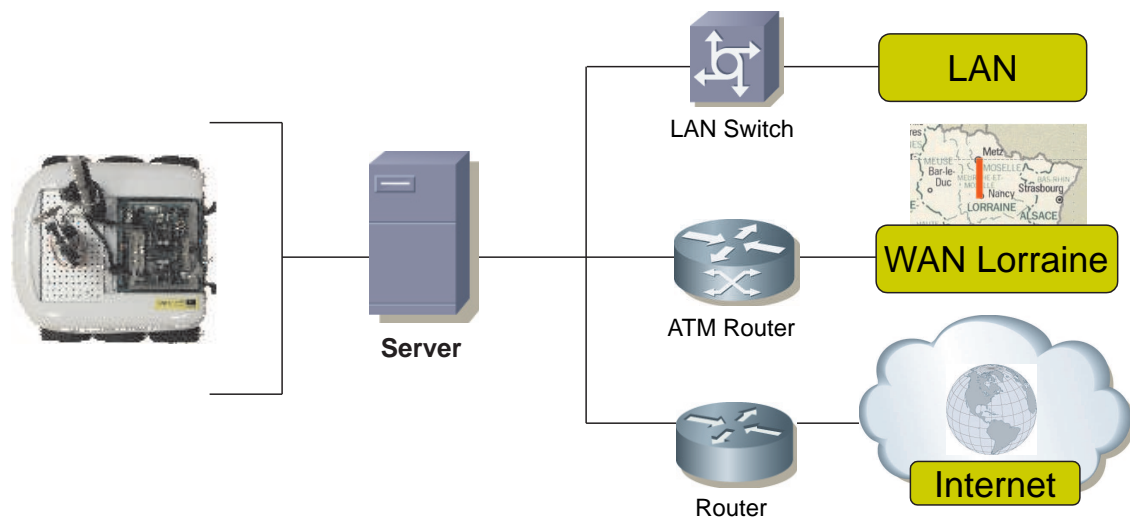
Figure 8.1: The remote application architecture. Through the server, the robot may be accessed by remote clients, be they from the LAN, over a P2P ATM connection, or over the Internet.

Image compression   In the application, the data sent over the network is of two types: robot control commands and returned data. In the case of the self-localization module, almost all of the returned data contains the acquired images. The robot camera returns 640×480 raster images, in `YUV` format. As grayscale images are needed for the self-localization

process, only the `Y` component needs to be transmitted to the client. However, on some cases, such as crowded networks, sending image data through the network should be preceded by compression to lower communication times. Focusing on lossy compression techniques, like `JPEG`, favors large amounts of data pruning. `JPEG`-compressed images have been successfully used in remote robot control applications [67], but there is a clear need to detect to which extent the PSimilarity detection resists to compression factors [97]. Hints from the previous version of the communication library tests indicate it does and offer a 10% quality factor figure for further tests.

Time issues The relationship between the times involved in the application's time performance estimation is show in equation 8.1. The total time, T_{total}, is the sum between the communication time, T_{com}, and the time for the remote routines execution, $T_{r.exec}$. Lowering one of the two components also lowers the total execution time, but failing to balance the communication and remote processing times may lead to bottlenecks for the part returning higher execution times.

$$T_{total} = T_{com} + T_{r.exec} \tag{8.1}$$

Problem statement The first approach to remote control contains two steps:

- The *first step* experimentally establishes the `JPEG` compression impact in terms of server and overall application performance, data sent over the network size and PSimilarity processing robustness.

- The *second step* establishes a base performance value for the remote control application or the *critical module* or both by using various communication networks.

8.3 Performance of our remote control approach

This section presents the results obtained for the two steps of this first approach. Results for the first step are presented in the *JPEG compression impact* section, while results for the second step are presented in the *Tests across the LAN*, *Tests across the ATM network* and *Tests across the Internet* sections. Early conclusions on the remote aspects are presented in the *Early conclusions on the remote control approach* section.

8.3.1 JPEG compression impact

The server machine performed very good when compressing 640×480 original raw data into `JPEG` images. There is no significant difference in the server application performance when compressing or not the images. Therefore, it is only the time needed for transmitting the image from the server to the client that remains important. The image sizes, depending on the compression scheme used, are presented in table 8.1. Images sent by the server as acquired, that is, in the `YUV` format, have the biggest size: $900KB$. A big improvement is obtained if only the `Y` component of the `YUV` image is sent over the network: just $300KB$. This is possible because the PSimilarity algorithm only needs grayscale images for landmarks detections. For the JPEG compression scheme, different quality rates have been tried: 10% (resulting images very blurry), 15%, 20%, 25%, 50% and 100% (resulting images very close to the originals).

Experimental tests established that images compressed using the JPEG standard, at a 20% quality are suitable for PSimilarity detection and have small sizes. Images with a quality factor of 15% are acceptable, but PSimilarity detection performed on them does not always return the expected results. Images with quality below 15% induce errors in the PSimilarity detection. During these tests, the average size for a compressed image was 24.31 KB, with a maximum of 24.71 KB. We conclude that total size required for a complete self-localization routine with optimal image compression is $425KB$ (17 images $\times$ 25 KB/image), obtained when using JPEG compression with 20% quality results.

Image	Uncompressed		Compressed JPEG quality [%]					
type	YUV	Y	10	15	20	25	50	100
Avg. size [KB]	**900**	**300**	**18.42**	**22.63**	**24.31**	**25.80**	**36.14**	**75.84**
Min. size [KB]	900	300	17.21	22.36	24.02	25.28	33.85	75.10
Max. size [KB]	900	300	20.20	23.02	24.71	26.31	37.62	76.87

Table 8.1: Size of acquired images under different formats and quality values. All images are based on 640×480 original raw data. YUV images are uncompressed images in YUV format. Y images are made of the Y component of a YUV image. $JPEG$ images are compressed versions of the full image, at various quality rates.

8.3.2   Tests across the LAN

Extensive tests on locally available machines were performed. The computers were situated on different sub-nets and had access to $100Mbps$ Fast Ethernet or $1000Mbps$ Gigabit Ethernet connections. Each complete set of tests considered the uncompressed and compressed images as data transmitted over the network. Various compression sizes were tried (the results have been presented in the previous section).

There were no significant differences between execution times when using compressed and uncompressed images, or in the execution time with regard to the sub-net hosting of the computer. We conclude that, even if `YUV` or `Y` data may be used with the same application performance, compressed JPEG images with 20% quality are the best choice, because they also reduce network traffic.

8.3.3   ATM networks performance estimation

We did not have time to perform tests for our application over the ATM network between Metz and Nancy. Comparative results obtained for previous versions of our application provide a quantitative view of what is expected to happen when running the application over this network. For the *purely sequential* self-localization routine, the times for the old remote application where with 3 up to 4.5 seconds bigger, for a module *slow-down* from 1.11 to 1.12. We expect the remote execution times for all the self-localization routine versions to lose at most the same amount of (communication) time. However, more testing is required to prove the assumptions.

8.3.4 Tests across the Internet

For Internet-based tests we only had the possibility to try some of the developed algorithms. Tight scheduling and problems regarding the MPICH installation on Italy imposed for testing the *overlapped* and *work pool* algorithms.

Overlapped results The results showed in table 8.2 indicate that remote execution induces a severe performance loss. The self-localization routine with JPEG 20% images performed in 18.95s, that is 1.34 times slower than the same routine executed on the slowest local machine in our tests. This time is however greatly superior to the time obtained in the YUV image case, 31.88s. The latter machine performed the self-localization routine 2.26 times slower than the slowest local machine. Performance fluctuation is also a more important issue over the Internet than it is on the LAN. Variations of an average 14% of the average execution time, for the JPEG 20% images, and of 10%, for the YUV images, were encountered.

Machine	Italy, JPEG 20%	Italy, YUV	Local, dev1
T_{avg}	18.95s	31.88s	14.09s
$T_{max} - T_{min}$	2.66s	3.08s	0.40s

Table 8.2: Results for the overlapped algorithm remotely executed from Italy. Times for a slow local machine, dev1, are also showed.

Work pool results The results showed in table 8.3 also show great performance degradation, when compared to local execution. The number of threads in the application does not severely influence the performance of the algorithm, because the single producer has to wait long times for each image acquisition. Being given that the processing time is lower than the acquisition time, one work pool consumer suffices. The small time difference of 0.13s between the $1 + 1$ and $1 + 2$ producer-consumer numbers in the case of JPEG 20% sustains this conclusion, while the bigger difference of 1.69 was obtained when the performance variation had been measured to 1.66s. For the work pool algorithm, the module *slow-down* was 1.27 for the compressed images version and 2.42 for the raw images version, both compared to the slowest local machine in test, `dev1`.

	Machine, image type	1+1	1+2
T_{avg}	Italy, JPEG 20%	14.87s	15.00s
$T_{max} - T_{min}$	Italy, JPEG 20%	0.49s	
T_{avg}	Italy, YUV	28.37s	26.68s
$T_{max} - T_{min}$	Italy, YUV	1.66s	
T_{avg}	Local dev1, any	11.72s	11.76s
$T_{max} - T_{min}$	Local dev1, any	0.16s	

Table 8.3: Results for the *work pool* algorithm remotely executed from Italy. Times for a slow local machine, dev1, are also showed.

8.3.5 Early conclusions on the remote control approach

We presented some early results regarding remote execution of the same algorithms developed for sequential machines and for parallel shared-memory systems. These results indicate that the use of JPEG 20% images leads to almost twice as better performance than in the case of raw YUV images. Also, the robot-computing tasks *overlapping* mechanism is promising. On the contrary, the image-flow is clearly insufficient, both for compressed and uncompressed images, to achieve high speedup values on remote shared-memory machines, using the *work pool* algorithm. Currently, suffices to associated one consumer thread to the producer thread. A more sustained flow of data is therefore required to make full use of the remote machine power, but this desiderate requires image requests to be sent in advance, or the *no-stop-scan* feature to be available.

The punctual measures introduced in this chapter are, of course, just early tests upon establishing a remote control testbed for our application. It is well known that Internet performance shows great daily variations [72]. We point out that more testing, with a full-day period split in 5 minutes quanta, is necessary to validate our results.

Chapter 9

Estimating Grid technology for mobile robotics

9.1   A brief history of Grid

The idea of Grid is linked with the evolution of Internet [1]. It is generally accepted that Grid is a sum-up of distributed systems around the world, much like Internet was for local networks at its time. Like in the Internet's case, the dynamic environment of Grid computing developed a fast growth, and has already produced two generations in ten years [81].

The first generation (1990-1995) contains the forerunners of Grid computing, embodied in projects like `I-WAY` [22] and `FAFNER`. Both projects were examples of *metacomputing*, that is, exploitation of computational resources within a national or even world wide network. The metacomputing setting involved the use of *metacomputers*, networks of heterogeneous, computational resources linked by software such that they can be used similarly to a personal computer in terms of easiness. While `I-WAY` was set to use supercomputers, `FAFNER` involved anonymous personal computers with slow computing capabilities. Both were demonstrated at the *Supercomputing'95* conference.

The second generation (1996-today) The two leading projects of the previous generation evolved into what we call today middleware Grid computing: `FAFNER` into `JXTA` (1998) and `Seti@home` and `I-WAY` into `Globus` (1996) and `Legion` (1997). *Middleware* is generally considered to be the layer of software standing between the operating system and the user applications, providing a variety of services required by an application to function correctly. As several middleware systems were made publicly available, integrated systems emerged. An example is the European `DataGrid` project conducted by `CERN`. Issues like heterogeneity, scalability and adaptability are tackled within this generation, in an attempt to establish general standards.

A *third generation* is on its way, with issues like distributed global collaboration, service-oriented approach and information layer issues embarked in the flagship [1].

9.2   Current Grid views

There are more then ten different definitions currently available in the `Grid` research community. Problems of these definitions lie in the closeness to *distributed systems*, *metacomputing*,

and *networked systems* definitions. Two very recent definitions dealing with Grid systems stand apart:

- Ian Foster's *3-point checklist* definition [32]:

  A Grid is a system that coordinates resources that are not subject to centralized control using standard, open, general-purpose protocols and interfaces to deliver nontrivial qualities of service.

- Andrew Grimshaw's *Grid hardware and user perspective* definition [42] (adapted some of the descriptions to fit phrases instead of point lists used in the article):

  From a *hardware perspective*, a Grid is a collection of distributed resources connected by a network.
  From a *user perspective* a Grid gathers together resources (e.g., CPU, data, and applications) and makes them accessible in a secure manner to users and applications.

These definitions represent the academic view (Foster) and the market-oriented view (Grimshaw) of the `Grid` phenomenon. Note that these definitions do not deal with the definition of *the* `Grid` - one `Grid` interconnecting all the others, much like the Internet did with all proprietary local networks -, the final target of the `Grid` systems research. *Bote-Lorenzo et all.* extract 10 main characteristics from the most cited Grid systems views: *large scale system*, dealing with huge number of resources, with a *wide geographical distribution*, connecting *heterogenous* software and hardware resources, allowing *sharing* between these resources, *multiple administration policies*, such as different security and control requirements, but with *resource coordination* between the different organizations using the Grid. In addition, access to these resources should be transparent, e.g. the user must perceive the Grid a single, virtual computing system, *dependable*, e.g. access falls under many types of Quality of Service rules, *consistent*, e.g. accessing resources should be done according to open, publicly available protocols, and, finally, *pervasive*, e.g. the Grid should be seen as a fail-safe system, with the best possible resource failure recovery [15].

9.3 Different kinds of Grid

As shown in the previous chapter, within the current frame of lacking a globally accepted Grid definition, grouping all kinds of Grid systems into a complete taxonomy is merely a claim. *Krauter et all.* [58] present three Grid types, depending on the resource management systems design objectives: *computational*, *data* and *service* Grids.

Computational Grids are built around the idea of using together many computers, in order to achieve an aggregate computational power that exceeds that of any single machine in the system alone. These systems are further divided into distributed supercomputing and high throughput. For *distributed supercomputing Grids*, applications are redundantly executed on several machines, to reduce the total time and also to ensure reliable execution of a certain job. Highly computationally demanding with reliability issues applications, such as weather modeling or laser simulation are examples of suitable applications for distributed supercomputing Grids. *High throughput Grids* focus on increasing the execution rate for

streams of jobs, usually on specialized architectures. Typical applications include those who require same operations to be run over and over, such as parameter range sweep in Monte Carlo simulations.

Data Grid systems provide an infrastructure for all the aspects of distributed high-throughput databases. Typical applications are large-scale data merging or specialized data mining from multiple sources.

Service Grids focus on providing the user services, which would be unavailable on one's regular working environment. Depending on the specific service delivery mode, service Grids span into on-demand, collaborative and multimedia. *On-demand* Grids aggregate, upon request, various resources, for limited periods of time. Visualization systems that increase their image output quality, given that more machines are added, are a typical example. *Collaborative* Grids focus on incarnating the term virtual organizations, i.e. globally spread humans and applications interacting in the same virtual environment. *Multimedia* Grids offers support for real-time multimedia applications. Quality of service and high-speed streaming are main issues upon building such systems.

9.4 Two Grid resource management systems

This section presents two such currently available Grid resource management systems: Globus and DIET. We argument our choice for DIET and attempt to foresee the Grid deployment process.

9.4.1 Globus

The `Globus` project [33] is the largest computational grid constructed up to date. First set up in 1997, Globus evolved into a multi-national and multi-institutional global Grid application through the completion of a first experimental platform, the *Globus Uniquitous Supercomputing Testbed* (`GUSTO`) [34]. `GUSTO` is based on the *Globus Metacomputing Toolkit*, which defines the services and capabilities of Globus. Services for resource management, security and communication are described. The most important capability is the consistent, dependable and pervasive access to high-performance computational resources.

The resource allocation and process management system is provided in `Globus` by the *Globus Resource Allocation Manager* (`GRAM`). Each `GRAM` holds responsibility for a set of local sources, accessible under the same site-specific allocation policy, provided by a third-party local resource management system or a simple *fork* daemon. Grid applications and tools make use of the resource allocation through a standard application programming interface, with requests expressed in terms of a *resource specification language* (RSL). High-level (application) RSL requests are refined into progressively more specific requirements by *resource brokers*, until the perfect match is found. The final step is to dispatch the separate resource allocation requests to the appropriate GRAM. *Co-allocation* comes in two flavors, *first*, atomic co-allocation, which means that if any of the requested resources is unavailable, the entire allocation process fails, and, *second*, allowing components of the submitted RSL expression to

be revised until the commit is issued. This architecture has a hierarchical view upon resource allocation process.

9.4.2 DIET

The *Distributed Interactive Engineering Toolbox* (`DIET`) project is a new Application Service Provider systems development tool, based on a Client/Agent/Server paradigm, i.e. the servers register to the agent, which is contacted by the client and, in turn, establishes a connection between the client and the most appropriate server, with CORBA used for communication. `DIET` provides a hierarchical agents system, with a scattered scheduler. Network Weather Service (NWS) sensors are placed on every node of the hierarchy to collect resource availability status, which are then used by Fast Agent's System Timer (FAST) [78] performance prediction tool and converted into suitable information for the scheduler.

The DIET components are: Clients, Master Agents, Local Agents and Server Daemons.

Client A client is any application that uses DIET to access computational resources.

Master Agent (MA) A Master Agent advertises services and is responsible for searching, upon request, the best server that can perform one of its listed services. Service requests may be issued from a web page as well from a compiled program. Master Agents have under control one or several Local Agents. Dispatching a Client's request involves contacting registered local agents and sending them the request, collecting the local agents valid answers and selecting the most appropriate, and, finally, returning the client the reference of a server that is the best suited to perform the service, according to the Master Agent selection policy.

Local Agent (LA) Local Agents are low-level agents, with the sole purpose of simplify the master agent's task of selecting the best computational server for a specific request. Local Agents register to a Master Agent or to another Local Agent and are responsible for directly connected resources and/or Local Agents, under a hierarchical brokerage structure. To dispatch requests, Local Agents contact the directly connected resources or Local Agents under them, issue the request, then select the valid servers and transmit the information above. No scheduling is performed at this level, only server filtering.

Server Daemon (SeD) A Server Daemon encapsulates the services offered by a computational server. Server Daemons are linked in the DIET system to a Local Agent and store information about the data available on the server, the list of provided services and system load, through the use of FAST.

All communication inside the DIET platform is based on CORBA.

Figure 9.1, presents a simple DIET hierarchy of agents. The clients contact Master Agents with regard to a listed service. The Master Agents receive the request and begins dispatching. For that, they contact the Local Agents directly connected to them and forward the requirements. The service requirements are propagated through the entire hierarchy of Local Agents to the Server Daemons. Each server that can satisfy the request launches FAST to estimate the job completion time and transmit the results back to the Local Agent. The
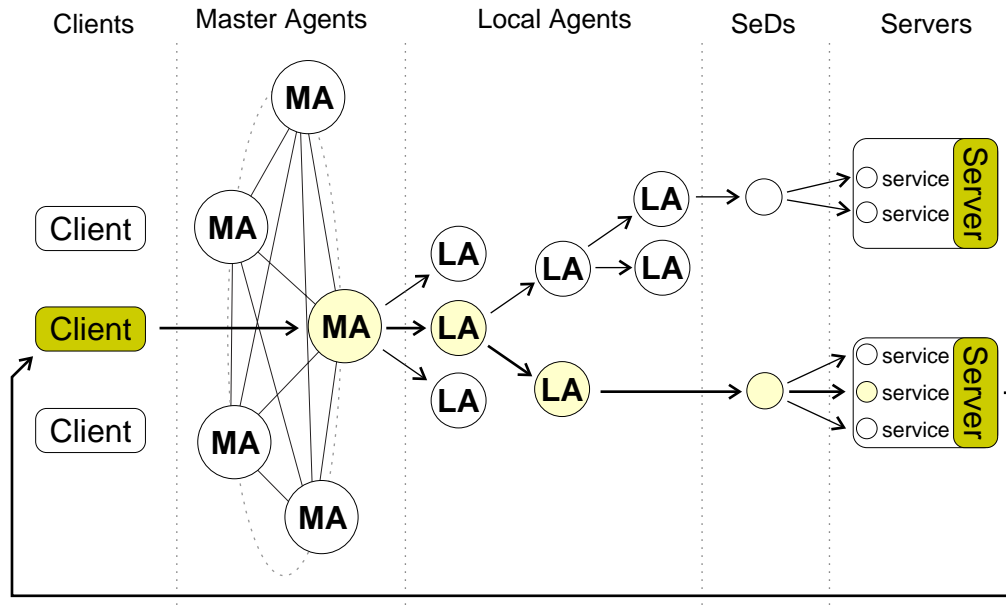
Figure 9.1: The DIET components. The client contacts the master agent (MA), which in turn makes use of the local agents (LA). The local agent contacts a computational server daemon (SeD). A virtual connection is established between the client and the computational server.

Local Agents prunes the valid server references to the fastest ones. The results are returned from the Local Agents to the Master Agents, which use schedulers and other decision-making tools to provide the client the appropriate server reference. A communication link based on CORBA is then established between the Client and the Server Daemon, allowing the service to be performed.

9.5 A Grid approach for mobile robotics

First step - completed The application was thoroughly tested in order to establish the best execution means, e.g. optimal number of processors and required memory sizes, on sequential, shared-memory and distributed-memory machines. This first step into Grid integration was presented in the previous chapters.

Second step - under way Selecting a Grid resource management system around which to build the distributed application is the next logical step. *Grid-savvy* mobile robotics applications built around this system should have the following requirements:

Highest performance The resource management system should provide a high performance, by using the best machines available with the most appropriate algorithms, being provided several algorithms, targeting different architectures.

Completely reliable If one of the machines handling a certain application task is removed, another should be fitted in the former's place. If no such host is available in place, the

resource management system should re-route the computing to a new (possibly, to be discovered) machine.

Optimally efficient Whenever possible, the resource management system should take advantage of the most lightweight hardware configuration available, in order to dispatch a certain task. Rare architectures should be preserved for future very specific demands. Ease of management of these resources should be a top priority.

Cost-effective The hardware infrastructure is considered already available. iF it is not the case, the expenses for the application hardware platform should be kept to a low. The development, testing, launch and maintenance costs altogether should be economically advantageous.

All the other Grid applications general considerations, such as good security, standard-based, scalability (to the extent of mobile robotics applications dimension restrictions) and so on, also apply.

Globus or DIET?

First MPICH tests on Globus(G1) at Supelec showed that performance is a significant problem for this implementation. Another problem was the relatively complicated Globus setup. Third-party tools and libraries, especially local resource managers, i.e. Condor, also need to be installed and evaluated. Since then, the MPICH library has greatly evolved, and now MPICH-G2 shows great improvements [56]. However, further tests should be performed until this implementation becomes mainstream. Therefore, Globus now looks promising, but involves too much pre-testing efforts to be a first choice.

A first glance at DIET's features shows that making first Grid integration tests on this platform is going to be greatly facilitated. The DIET system is small and does not rely on a big number of third-party tools and libraries. Among the software dependencies, the most important is omniORB, a free CORBA implementation, with performance close to that of sockets'. A security level is provided by the SSL module that ships with the current version of omniORB. We conclude that building on top of DIET provides the resource management system and a secure communication layer with very few efforts, and we choose DIET as the primary Grid development platform.

Chapter 10

Conclusions

In this work, we have established that global speedup for mobile robotic application can be achieved on common off-the-shelf computers, even if the tasks dimensions are very small. Furthermore, we optimized and speeded up the entire application by severely improving the self-localization routine, one *critical* tasks of the application.

Best practices have been established for sequential, shared-memory and distributed-memory architectures, with regard to the algorithm structure and preliminary considerations for mobile robotics.

Early tests were performed for remote control, with encouraging results. The theoretical transmission delay and the packet loss effects were confirmed by our tests. Speedup was achieved by using algorithms specifically designed for the remote machines and reduced communication through the use of compressed data.

The early remote control tests and a study of current Grid resource management systems turned our attention towards a DIET implementation for our application. Globus was found, for the moment, unsuitable for this task, but with the new MPICH-G2 library maturing, this situation could reverse.

10.1 The application testbed

The testbed was constructed around several cheap common-off-the-shelf single- and multi-processor computers. A simple, yet effective, profiling library was used to determine accurate run-times. This allowed us to quickly spot the points in our routines that needed urgent improvement and to eliminate the ones which were not worth the optimizing efforts.

Mechanical times for the robot and camera, as well as the optimal time for a complete panoramic scan, were established.

The *purely sequential* sequential algorithm for purely sequential machines was built as the reference for all the future versions. Three optimized algorithms, the *overlapped* algorithm, for purely sequential machines, the *work pool* algorithm, for sequential machines with hyper-threading mechanism support and for shared-memory machines, and the *stick passing* algorithm, for distributed-memory machines, were developed and thoroughly tested. All the needed values for evaluating performance on sequential and parallel architectures were obtained. Therefore, the tests on sequential and parallel architectures are complete.

Tests were also performed on the remote version, and base reference results were established for two remote sequential machines on LAN and Internet. The impact of lossy JPEG

image compression quality factor on the PSimilarity detection was empirically detected. YUV and 20% JPEG file formats were tested against transmission over various network technologies. Based on previous observations, ATM networks performance estimations were made. The application platform for the reference values is complete. More tests are required, however, to complete this study. It is very likely that new solutions would have to be designed for this approach.

We are looking forward for the Grid technical problems, many of which will coincide with the problems encountered in the remote control case.

Finally, the impact of mechanical improvements of the Koala robot camera mechanisms was estimated. Thorough tests were performed for the possible case of obtaining an accurate camera position information during movement.

10.2   Robotic mechanical testing constraints

We ran more than 2500 benchmarks for our tests, including around 500 for debugging and tune-up. From the over 2000 successful benchmarks, 25 were run for the *purely sequential* version, 75 for the *overlapped* versions, including the *no-stop-scan*, 750 went for the *work pool* algorithm, 1150 for the *stick passing* version, and, finally, 50 benchmarks were run for the remote control measurements. On a *per-machine* basis, 100 tests were run for the `dev` monoprocessor machines, 360 tests were run for the `monox` monoprocessor machines, 400 for the `bip3n` biprocessor machines, 660 for the `iic` biprocessor machines, and, finally, 450 tests were run for the `quadx` quadriprocessor machines. We conclude that the large number of tests ensures that average values have meaningful results. Delays between consecutive benchmarks, and long pauses between large series of benchmarks were used to ensure the robotic system's safety and data robustness.

10.3   Global application issues

Speedup and maximum speed   The complete application speedup currently depends on the combination of navigation/self-localization calls. We assumed, for reliability reasons, that each navigation step will be followed by a self-localization. The resulting global application speedup is given by the equation 5.1. Under the same assumptions, the self-localization application reaches as much as 95% of the maximum possible speed, as given by the mechanical times measurements.

Parallelizability and isoefficiency   The parallelizability values are affected by the small dimensionality of the robotic tasks. We see just one solution to this problem: having several moderately computationally demanding robotic tasks at hand, such as the case of complex mobile robotic applications. The same discussion suits the isoefficiency considerations. There is no isoefficiency function, but this does not mean the system would not scale. Again, adding more tasks can prove decisive to the proper execution of a certain mobile robotics application, such as a robot entering and quitting a dangerous zone.

Cost-effectiveness   The common off-the-shelf computers costs have dropped significantly since the year 1999, the year when this project was started. It is currently convenient to buy cheap biprocessor machines or even sequential processor with hyper-threading support

systems. The price range for such systems is between $USD1000$ and $USD3500$, with top quadriprocessor systems at about $USD15000$, while the robotic system would lead to spending around $USD15000$.

10.4 Architecture/algorithm lookup table

One of the goals of this work was to establish an architecture/algorithm lookup table. This table is graphically presented in figure 10.1.

Sequential	Overlapped algo	easy to do not exploiting the intrinsic multi-tasking nature of robotics
Shared-memory	Hyper- or Multi-threaded algo	fairly easy to do, well suited only one (multi-processor) machine might not be enough
Message passing	Stick-passing algo	well suited not easy to maintain
Remote	A distributed algo, machine-specific	well suited when resources are not at hand; well suited for fast & reliable networks network-dependent

Figure 10.1: The architecture/algorithm lookup table.

Sequential For the purely sequential machines, the overlapped algorithm produces the best results. These algorithms are easy to implement, but do not exploit the intrinsic multi-tasking nature of mobile robotics application. However, it makes a crude use of the computing/robot control parallelism.

Shared-memory This class also includes the sequential machines with hyper-threading capabilities. For the shared-memory machines, the work-pool algorithm is the best performer. The algorithm is fairly easy to implement, being based on a well known paradigm, and makes good use of the computing/robot control parallelism. For the sequential machines, using just a single processor is not be enough for a complex mobile robotics application. For the proper shared-memory machines there are more processing units available. However, just one machine might not be enough for complex applications.

Distributed-memory The stick passing algorithm is to be used for distributed-memory machines with message passing capabilities, i.e. clusters. This algorithm is well suited for such applications, as it minimizes the communication, the primary source of delays, and manages to obtain a balanced processing units utilization. The main drawback is the code maintainability. Indeed, working with specific message-passing libraries tends to produce tedious code, as the common routines are wrapped in parallel calls structures.

Remote   For the remote machines, a distributed algorithm with machine-specific code for the computing parts should be deployed. A simple way to achieve that is to execute on the remote machine an algorithm based on the remote machine's architecture, and to collect and transmit the results from a simple distributed interface program. The remote methods are needed when resources are not at hand and ar well suited for fast and reliable networks. However, the remote approach is still network-dependent.

10.5   Grid integration for mobile robotics applications

The application was thoroughly tested and best practices have been established for sequential, shared-memory and distributed-memory machines. Therefore, we have completed the first step into Grid integration. From the currently available Grid resource management systems, DIET looks the most promising for mobile robotics. Deployment of this application onto DIET environment in currently underway.

Chapter 11

Perspectives

The results obtained during the past six month by the research community as well as our team proved that the initial objectives may and should be broadened. In order to improve the working strategy the goals were split into three categories: short term, for goals achievable in up to one year, medium term, for goals requiring from one year to three years of work, and long term, for goals requiring more than three years. Of course, these plans are subject to (more than likely) changes, but they mainly include directions as: modules enhancement, application enrichment, Grid robot control integration and enhanced model development.

11.1 Short term: Grid integration

Grid robot control integration   The most important direction for our project as of today is the attempt to integrate the application into a Grid environment, be it DIET (more likely), Globus or any other. Integrating the current application into Grid would be very useful for gathering information about the possible problems regarding mobile use of resources. So far we are targeting reliability issues through the use of several similar processes run on different machines - if not all machines fail, the correct result will be obtained.

Module enhancement   The next step for the path-planning module is to eliminate the problems encountered so far and to include a shortest path selection for the Dubins trajectories.

Application enrichment   Finally, we plan to integrate an existing module for obstacle detection into the complete application. Having more modules is always beneficial in terms of proving the model's ability to suit both simple and complex applications, but having this particular module would allow the application to qualify as navigation-savvy.

11.2 Medium term: complex application

Grid robot control integration   Given that the Grid integration succeeds, obtaining an optimal speedup through the use of several machines (and possibly checking for the occurrence of "lucky" super-linear speedup) is the next logical step (though in practice it might not prove efficient), while having an enriched application under Grid test is another challenge.

Model development   We also plan to establish a model for integrating time-constrained mobile robotics applications into Grid.

Module enhancement   Currently the self-localization system and the navigation system act completely separate. This is not longer possible in today's mobile robotics, so making the image acquisition and processing system work during the robot movement is *the* direction to follow.

Application enrichment   We plan to add more sensors to the sensing system, and to perform sensors fusion (based on map database and AI) for more reliable results. There is no such thing as a perfect data acquisition system [49], so having more data becomes critical in some cases, such as surveillance in indoor environments.

Application enrichment   Finally, having a second camera mounted on the robot (the Koala robot's structure specifically permits it) would allow for better self-localization. The target is a system capable of splitting the panoramic scan in two parts, or having the second camera perform a different task than localization, like intruder detection.

11.3   Long term: indoor autonomous robot

Application enrichment   We plan to integrate AI in order to create random paths that cover the building while avoiding to cover the same path every time.

Enhanced model development   Establishing a model for integrating time-constrained remote control applications into Grid, as well as providing a tool for ease of use of such a model, are matters of great interest. By providing such tools we hope to have successfully contributed to a part of the Grid research.

Application enrichment   The final goal is to obtain the complete robotic application - **an indoor autonomous surveillance robot with remote computing resources**.

It is certain that these perspectives may change as one or more new technologies may come into our attention, or our predictions and assumptions may prove wrong (such as the case of Grid integration). However, we see these expectations as consistent views upon the application transformation towards the final goal: the indoor autonomous surveillance robot with remote computing resources.

Bibliography

[1] S. Adcock. How does the grid extend the internet, and what is the future vision for this development?

[2] J. S. Albus, R. Lumia, J. C. Fiala, and A. J. Wavering. Nasrem: The nasa/nbs standard reference model for telerobot control system architecture. In *Proceedings of the 20th International Symposium on Industrial Robots*, volume 4, October 1989. Tokyo, Japan.

[3] G.M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. *In Proceedings of AFIPS Conference, Reston, VA*, 30:483–485, April 1967.

[4] R. C. Arkin. Motor schema-based mobile robot navigation. *International Journal of Robotics Research*, 8(4):1989, August 92-112.

[5] D. Buttlar B. Nichols and J. Proulx Farrel. *Pthreads Programming*. O'Reilly & Associates, Inc., first edition, September 1996.

[6] P. Backes, K. Tso, and G. Tharp. Mars pathfinder mission internet-based operations using wits. In *IEEE Intl. Conference on Robotics and Automation, Leuven, Belgium*, May 1998.

[7] M. Baker. Cluster computing white paper, version 2.0, december 2000. *IEEE Task Force on Cluster Computing (TFCC) Publications*, December 2000.

[8] H. E. Bal and D. Grune. *Programming Language Essintials*. Addison-Wesley, 1994.

[9] H. E. Bal, M. F. Kaashoek, and Andrew S. Tanenbaum. Orca: a language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, 1992.

[10] J. Baus, A. Krueger, and W. Wahlster. A resource-adaptive mobile navigation system. In *ACM IUI'02, San Francisco, California*, January 2002.

[11] P. Beckman, D. Gannon, J. Gotwals, N. Sundaresan, and S. Yang. Object-parallel programming with pc++.

[12] M. Beetz. Structured reactive controllers : A computational model of everyday activity. *Proceedings of the Third International Conference on Autonomous Agents (AA99)*, 1999.

[13] E.E. Binder and J.H. Herzog. Distributed computer architecture and fast parallel algorithms in real-time robot control. *IEEE Transactions on Systems, Man, and Cybernetics*, 16(4):543–549, 1986.

[14] J. Borenstein and Y. Koren. Tele-autonomous guidance for mobile robots. *IEEE Transactions on Systems, Man, and Cybernetics, Special Issue on Unmanned Systems and Vehicles*, 20(6):1437–1443, November/December 1990.

[15] M. Bote-Lorenzo, Y. Dimitriadis, and E. Gomez-Sanchez. Grid characteristics and uses: a grid definition, 2002. Technical Report CICYT TIC2002-04258-C03-02, Univ. of Valladolid, Spain.

[16] S. Bottazzi, S. Caselli, M. Reggiani, and M. Amoretti. A software framework based on real-time corba for telerobotic systems. In *IROS'2002*, 2002.

[17] R. A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2(1):14–23, 1986.

[18] V. Chaudhary and J.K Aggarwal. Parallelism in computer vision: a review. In V. Kumar, P.S. Gopalakrishnan, and L.N. Kanal, editors, *Parallel algorithms for machine intelligence and vision*, pages 271–309. Springer Verlag, 1990.

[19] C. Chiculita and L. Frangu. A web based remote control laboratory. *6th World Multiconference on Systemics, Cybernetics and Informatics*, July 14-18 2002. Orlando, Florida.

[20] M.Y. Chow and Y.Tipsuwan. Network-based control systems: a tutorial. *Industrial Electronics Society, 2001. IECON '01. The 27th Annual Conference of the IEEE*, 3:1593 –1602, August 2001.

[21] Compaq. Guide to the posix threads library, April 2000. In Compaq Tru64 UNIX Guides Collection. Order number AA-RH9RB-TE.

[22] T. A. DeFanti, I. Foster, M. E. Papka, R. Stevens, and T. Kuhfuss. Overview of the I-WAY: Wide-area visual supercomputing. *The International Journal of Supercomputer Applications and High Performance Computing*, 10(2/3):123–131, Summer/Fall 1996.

[23] A. Denis, C. Perez, and T. Priol. Towards high performance corba and mpi middlewares for grid computing. In *Proceedings of the 2nd Intl. Wshp. on Grid Computing, Denver, Colorado*, pages 14–25, September 2002. LNCS 2242, INRIA Rennes Research Report on Networks and Systems, PARIS Project, No.4555.

[24] G. N. DeSouza and A. C. Kak. A subsumptive, hierarchical, and distributed vision-based architecture for smart robotics. *Submitted to: IEEE Transactions on Robotics and Automation*, May 2002.

[25] G. N. DeSouza and A. C. Kak. Vision for mobile robot navigation: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24, February 2002.

[26] Hank Dietz. Linux parallel processing howto, 1998.

[27] E.D. Lazowska D.L. Eager, J. Zahorjan. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers*, 38(3):408–423, March 1989.

[28] M. J. Flynn. Very high-speed computing systems. *Proceedings of IEEE*, (54(12)):1901–1909, Dec 1966.

[29] M. J. Flynn. Parallel prospects for the future... and maybe yet. *Computer*, (29(12)):151–152, Dec 1996.

[30] T. W. Fong, C. Thorpe, and C. Baur. Multi-robot remote driving with collaborative control. *IEEE Transactions on Industrial Electronics*, 2003.

[31] Message Passing Forum. Mpi: A message passing interface standard. *International Journal of Supercomputer Applications*, 8(3/4), 1994.

[32] I. Foster. What is the grid? a three point checklist. *Grid Today ezine*, 1(6).

[33] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

[34] I. Foster and C. Kesselman. The Globus project: a status report. *Future Generation Computer Systems*, 15(5–6):607–621, 1999.

[35] Fox and Coddington. Parallel computers and complex systems. In Terry R. J. Bossomaier and David G. Green, editors, *Complex Systems, Cambridge Univ. Press, 2000.* 2000.

[36] G. Fox. Software and hardware requirements for some applications of parallel computing to industrial problems, 1995.

[37] A. Geist, A. Beguelin, Jack Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing.* MIT Press, Cambridge, Mass., 1994.

[38] UNECE Geneva. Press release ece/stat/02/01. Technical report, United Nations Economic Comitee for Europe, Oct. 2002.

[39] Maria Gini. Talks. In *Dagstuhl Seminar on Plan-Based Control of Robotic Agents.* Jul 2003.

[40] Y. Goto and A. Stentz. Mobile robot navigation: The CMU system. *IEEE Expert*, 2(4):44–54, Winter 1987.

[41] J.H. Graham. Special computer architectures for robotics: Tutorial and survey. *IEEE Transactions on Robotics and Automation*, 5(5):543–554, October 1989.

[42] A. Grimshaw. What is a grid? *Grid Today ezine*, 1(26).

[43] J.L. Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31(5):532–533, May 1988.

[44] D. Hamilton. Parallel robot control using speculative computation. *Journal of Robotics and Automation*, 13(4):101–112, December 1998.

[45] D. Hamilton, J. Bennet, and I. Walker. Parallel fault tolerant robot control. *In proceedings of SPIE conference on Cooperative Intelligent Robotics in Space III*, pages 251–261, November 1992.

[46] D. Henrich. Fast motion planning by parallel processing - a review, 1997.

[47] D. Henrich and T. Honiger. Parallel processing approaches in robotics. *IEEE International Symposium on Industrial Electronics (ISIE'97)*, July 1997. Guimaraes, Portugal.

[48] D. Henrich, J. Karl, and H. Woern. A review of parallel processing approaches to robot kinematics and jacobian. Technical Report ISSN 1432-7864, University of Karlsruhe, Computer Science Department, October 1997.

[49] Joachim Hertzberg. Talks. In *Dagstuhl Seminar on Plan-Based Control of Robotic Agents*. Jul 2003.

[50] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Houston, Tex., 1993.

[51] H. Hirukawa and I. Hara. Web-top robotics. *IEEE Robotics and Automation Magazine*, 7(2):40–45, June 2000.

[52] Lei Hu and Ian Gorton. Performance evaluation for parallel systems: A survey.

[53] K-Team. *Kameleon 376 SBC User's Manual*. K-Team S.A., May 1999.

[54] K-Team. *Koala User's Manual, Silver Edition*. K-Team S.A., May 2001.

[55] A.C. Kak and G.N. DeSouza. Robotic vision: What happened to the visions of yesterday? In *Proceedings of the 2002 Int. Conference in Pattern Recognition, Quebec, Canada*, volume August 2000, 2000.

[56] N. Karonis, B. Toonen, and I. Foster. MPICH-G2: A Grid-Enabled MPI. The Globus Project documentation, www.globus.org, 2002.

[57] A. Kheddar, C. Tzafestas, and P. Coiffet. The hidden robot concept – high level abstraction teleoperation. *IEEE/RSJ Intl. Conference on Intelligent Robotics and Systems, IROS'97, Grenoble, France*, 3:1818–1824, September 1997.

[58] K. Krauter, R. Buyya, and M. Maheswaran. A taxonomy and survey of grid resource management systems for distributed computing. *Software - Practice and Experience*, 00:1-7, Winter 2001.

[59] C. Chiculita L. Frangu. Remote laboratory allowing full-range student-designed control algorithm. *9th IEEE International Conference on Electronics, Circuits and Systems - ICECS 2002*, September 15-18 2002. Dubrovnik, Croatia.

[60] C.S.G. Lee. Introduction to special issue on robot manipulators: Algorithms and architectures. *IEEE Transactions on Robotics and Automation*, 5(5):541–542, October 1989.

[61] R.B. Lee. Empirical results on the speed, efficiency, redundancy and quality of parallel computations. *Proceedings of the 1980 Intl. Conf. on Parallel Processing*, pages 91–100, August 1980.

[62] LLNL. Posix threads programming, April 2001. Available online at http://www.llnl.gov/computing/tutorials/pthreads/.

[63] A.B. Cremers M. Beetz, T. Arbuckle and M. Mann. Transparent, flexible, and resource-adaptive image processing for autonomous service robots. *Third International Conference on AI Planning Systems*, 1998.

[64] C.B. Madsen, C.S. Andersen, and J.S. Sørensen. A robustness analysis of triangulation-based robot self-positioning. *Proceedings of the International Symposium on Intelligent Robotic Systems, Stockholm*, 1997.

[65] A. Majumdar. Parallel performance study of monte carlo photon transport code on shared-, distributed-, and distributed-shared-memory architectures. In *Proceedins of the 14th Parallel and Distributed Processing Symposium, IPDPS'00*, volume May 2000, pages 93–102, 2000.

[66] D.E. Manolakis. Efficient solution and performance analysis of 3-d position estimation by trilateration. *IEEE Transactions on Aerospace and Electronic Systems*, 32(4):1239–1248, October 1996.

[67] K. Masuda, N. Tateishi, Y. Suzuki, E. Kimura, Y. Wie, and K. Ishihara. Experiment of wireless tele-echography system by controlling echographic diagnosis robot. *Medical Image Computing and Computer-Assisted Intervention - MICCAI 2002, Lecture Notes in Computer Science 2488*, 2002.

[68] D. Gelernter N. Carriero and J. Leichter. Distributed data structures in linda. *ACM Symposium on Principles of Programming Languages*, pages 236–242, 1986.

[69] L. Navarro-Serment, R. Grabowski, C. Paredis, and P. Khosla. Millibots. *IEEE Robotics and Automation Magazine*, pages 31–40, December 2002.

[70] I. Noda, M. Asada, H. Matsubara, M. Veloso, and H. Kitano. Robocup as a strategic initiative to advance technologies. In *Proceedings of the 1999 IEEE International Conference on Systems, Man, and Cybernetics (SMC '99)*, volume 6, pages 692–697, October 1999.

[71] R. Oboe and P. Fiorini. Issues on internet-based teleoperation, 1997.

[72] R. Oboe and P. Fiorini. A design and control environment for internet-based telerobotics, 1998.

[73] OpenMP Architecture Review Board. OpenMP Specifications for C/C++ v2.0, 2002.

[74] L. E. Parker. Cooperative motion control for multi-target observation. In *Proceedings of the 1997 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS '97)*, pages 1591–1598, 1997.

[75] R. Paul, C. Sayers, and J. Adams. Operabotics, 1995.

[76] W. Pfeiffer, L. Carter, A. Snavely, R. Leary, A. Majumdar, S. Brunett, J. Feo, B. Koblenz, L. Stern, J. Manke, and T. Boggess. Evaluation of a multithreaded architecture for defense applications, 1999.

[77] Martha Pollack. Talks. In *Dagstuhl Seminar on Plan-Based Control of Robotic Agents*. Jul 2003.

[78] M. Quinson. Dynamic performance forecasting for network-enabled servers in a meta-computing environment. In *Proceedings of the* Intl. Workshop on Performance, Modeling, Evaluation, and Optimization of Parallel and Distributed Systems *(PMEO-PDS'02)*, 2002.

[79] N. K. Ratha and A. K. Jain. Computer vision algorithms on reconfigurable logic arrays. *IEEE Transactions on Parallel and Distributed Systems*, 10(1):29–41, 1999.

[80] J. Rosenblatt. The distributed architecture for mobile navigation. *Journal of Experimental and Theoretical Artificial Intelligence*, 9(2/3):339–360, April-September 1997.

[81] D. De Roure, M. A. Baker, N. R. Jennings, and N. R. Shadbolt. The evolution of the grid. Research Support Article on CiteSeer, 2002.

[82] J.M. Hollerbach S. Narasimban, D.M. Siegel. Condor: An architecture for controlling the utah-mit dexterous hand. *IEEE Transactions on Robotics and Automation*, 5(5), October 1989.

[83] S. Vialle, E. Dedu and C. Timsit. ParCeL-5/parssap: A parallel programming model and library for easy and fast execution of simulations of situated multi-agent systems. In *Proceedings of the SNPD'02 International Conference on Software Engineering Applied to Networking and Parallel/Distributed Computing, Madrid, Spain*, 2002.

[84] S. Vialle, M. Bouzid, V. Chevrier and F. Charpillet. ParCeL-3: A parallel programming language based on concurrent cells and multiple clocks. In *Proceedings of the SNPD'00 International Conference on Software Engineering Applied to Networking and Parallel/Distributed Computing, Reims, France*, 2000.

[85] S. Vialle, T. Cornu and Y. Lallement. ParCeL-1: A parallel programming language based on autonomous and synchronous actors. *ACM SIGPLAN Notices*, 31(8):43–51, 1996.

[86] D. Scharstein and A.J. Briggs. Real-time recognition of self-similar landmarks. *Workshop on perception for mobile agents*, June 1999.

[87] Douglas C. Schmidt, David L. Levine, and Sumedh Mungee. The design of the TAO real-time object request broker. *Computer Communications*, 21(4):294–324, 10 April 1998.

[88] A. Siadat and S. Vialle. Robot localization, using p-similar landmarks, optimized triangulation and parallel programming. *2nd IEEE International Symposium on Signal Processing and Information Technology*, 2002. Marrakesh.

[89] S. Singh and S. Thayer. A foundation for kilorobotic exploration. In *Proceedings of the Congress on Evolutionary Computation at the 2002 IEEE World Congress on Computational Intelligence*, May 2002.

[90] M. Henshaw T. R. Collins, R. C. Arkin. Ieee intl. conference on robotics and automation. volume 1, Atlanta, Georgia, 1993. IEEE.

[91] T. Uhlin and K. Johansson. Autonomous mobile systems: A study of current research, 1996. Technical Report ISRN KTH/NA/P–96/03–SE, Dept. of Numerical Analysis and Computing Science, KTH (Royal Institute of Technology), Stockholm, Sweden.

[92] Sips Delft University. Programming languages for high performance computers.

[93] S.C. Venema and B. Hannaford. Miniature telerobots in space applications.

[94] S. Vialle. *ParCeL User's Guide and Reference Manual*. CodeME S.A.R.L., St Genis-Pouilly (France), 1993.

[95] T. Vidal, M. Ghallab, and R. Alami. Incremental mission allocation to a large team of robots. In *IEEE ICARO*, pages 1620–1625, 1996.

[96] Th. Lumpp A. Speck W. Kuchlin, G. Gruhler. Highrobot: a high-performance universal robot control on parallel workstations. *Workshop on Engineering of Computer-Based Systems in Monterey (ECBS '97)*, pages 444–451, March 1997.

[97] Gregory K. Wallace. The jpeg still picture compression standard. *Communications of the ACM*, 34(4):30–44, April 1991.

[98] L.L. Whitcomb. Advances in architectures and algorithms for high performance robot control. *Ph.D. Thesis at Yale University*, 1992.

[99] B. Wilkinson and M. Allen. *Parallel Programming: techniques and applications using networked workstations and parallel computers*. Prentice Hall, 1999.

[100] Y. Zhang, M. Schervish, E.U. Acar, and H. Choset. Probabilistic methods for robotic landmine search. *In Proceedings of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems*, October 2001.

Appendix A

Conferences and published papers

Dagstuhl The work included in this paper was presented as A. Iosup, S. Vialle, A. De-Vivo, A. Siadat, C. Giraud-Audine, *Mobile Robot Navigation and Self-Localization System: Parallel and Distributed Experiments* at the **Dagstuhl Seminar on Plan-Based Control of Robotic Agents** (Seminar no.03261), **June 22-27, 2003**.

Dagstuhl Seminar
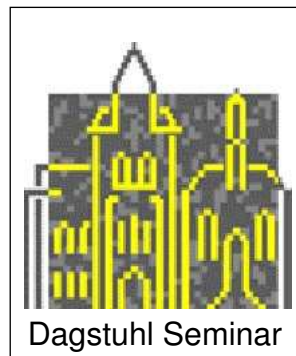
Appendix B

List of the programs created for this application, with their usage

This section presents the most important programs built and used for this project. Note that most of the applications were run using scripts.

Koala Server To get complete information on the program's parameters, run `ersdipKoalaServer`.

Usage:

```
ersdipKoalaServer [port] [config file]
```

where `port` is the server's port, and `config file` is the configuration file for the robot. If no configuration file is present, an interactive setup procedure will begin.

parNavLoc This is the complete application that contains both the navigation and the self-localization module.

Parameters:

-a angle step, $min = 5; max = 45; default = 20$;

-b benchmark flag - if present, benchmarks will be displayed;

-c path coefficient, $min = -1000; max = 1000; default = 1$;

-d debug flag - if present, several debug messages will be displayed;

-e extra buffer size, for the overlapped version, $default = 0$;

-l landmark file name;

-m method identifier: 0 for overlapped, 1 for work pool algorithm, 2 for MPI version;

-n number of consumer threads for the work pool algorithm;

-t timing flag - if present, complete timings will be performed;

-p server port number;

-s server host name;

-v verbose level, $min = 0; max = 10; default = 1$;

Robot self-localization This is the program that implements the self-localization routine. `rl` comes in two main flavors: version `1.97`, for *purely sequential*, *overlapped* and the *work pool* algorithms, and version `1.95a`, for the *stick passing* algorithm with at least 1 MPI process. Usage is similar to the usage of the `parNavLoc` program (see above).

Move client This program is used for internal server testing. `ersidpTestMoveClient` tries to use a given `nono` based server's resources to initialize and run around the robot.

Usage:

`ersdip-koala-move-client [host name] [port]`

where `host name` and `port` are the *Koala Server* host name and port number.

Image client This program is used for internal server testing and image acquisition timings. `raw_image_timing` was used at the very start of the project, to obtain useful image acquisition timings for various situations (acquiring `RGB24` and `JPEG` images with different compression modes and qualities).

Usage:

`test [host name] [port] -v -tests [no of tests] -mode [mode] -policy [policy] -quality [quality %]`

where `host name` and `port` are the *Koala Server* host name and port number, `no of tests` is the desired number of tests, `mode` is one of: 1 for `RGB24`, 3 for `JPEG`, policy is one of local and remote and specifies where is the `JPEG` compression taking place, and `quality %` is the image quality after compression, from 1 to 100.

Basic rotation timing This program was used to determine the time needed for a complete panoramic rotation, without image acquisition or image processing. All this program does is to rotate the robot camera and time this process. Results from this program have been presented in the *Current mechanical limits* section (4.5).

Usage:

`test-turn-time [host name] [port]`

where `host name` and `port` are the *Koala Server* host name and port number.

Panoramic scan timing This program was used to determine the time needed for a complete panoramic scan, with image acquisition but without any image processing. All this program does is to rotate the robot camera and acquire images and time this process.Results from this program have been presented in the *Current mechanical limits* section (4.5).

Usage:

`test-shot-time [host name] [port]`

where `host name` and `port` are the *Koala Server* host name and port number.

Navigation The navigation program is a refined version of the navigation module provided by Matthieu Massonneau. It is used just as internal test program.

Usage:

`navi [host name] [port] [paramfile]`

where `host name` and `port` are the *Koala Server* host name and port number, `paramfile` is the name of the robot position configuration file.

PSimilar landmarks This is a Win32 program that exports PSimilar landmarks as printable PPM images (use in conjunction with IrfanView or some other program - a complete tutorial is available for this combination of programs).

Usage:

`PSimilar_marks sizex sizey bits number image`

where `sizex` and `sizey` define the image size, `bits` is the number of bits used to represent the number, `number` is the number to print, and `image` is the image file name.

Results processing This Win32 program was used to post-process result files obtained from the robot self-localization programs.

Parameters:

-i input file name;

-o output file name;

-m parsing mode: 0 for sequential version results, 1 for work pool algorithm results, 2 for stick passing algorithm results, 3 for average times for multiple files stick passing algorithm results;

-d data mode: 0 get best times; 1 get average times.

Appendix C

Sample self-localization module output

```
Date/Time: Wed Jun  4 10:50:08 2003

Koala Robot parLoc, v.1.97 by A.Iosup [aiosup@yahoo.com], adapted
to nono from previous work by S. Vialle and A. de Vivo
   Info           : controls and times the koala robot localization process
[seq, seq ovr, thr ovr, mpi]
   Last compiled : Jun  4 2003
Present configuration:
    -verbose level: 10
    -timing flag: On
    -processor number: 1
    -angle step of the camera: 20deg
    -method identifier (for localization): 3

Landmark list read in file "LandmarkPosition.txt"
    -landmark 23: X = -0.45, Y = -3
    -landmark 1: X = -26.5, Y = 97.5
    -landmark 6: X = 123.7, Y = 18.3
    -landmark 19: X = 123.3, Y = -47.8
    -landmark 5: X = -14.5, Y = -76.3

Number of basic landmark detections: 11 Number of really detected
landmark: 1 Triangulation has failed

##Localization failed: less than 3 landmarks detected

Timing:
    -Localization Program: 19935.8 (ms)
    -Localization Routine: 14719.7 (ms)
        -Panoramic scan        : 14719 (ms)
        -Landmark list filtering: 0.022 (ms)
        -NRTriangulation       : 0.003 (ms)
        -Result Broadcast      : 0.001 (ms)
 |-- update_img
    |-- elapsed  2081.4150 ms
```

```
         |-- # calls        17
         |-- average 122.436179
         |-- Image size [KB]
             |-- total     419.8311
             |-- average   24.695944
|-- get_img
    |-- elapsed   189.7540 ms
    |-- # calls        17
    |-- average   11.162000
|-- RGB24 scale
    |-- elapsed    19.4050 ms
    |-- # calls        17
    |-- average    1.141471
|-- memcpy big
    |-- elapsed     0.0000 ms
    |-- # calls
    |-- average    0.000000
    |-- size [KB]
        |-- total        0.0000
        |-- average    0.000000
|-- memcpy small
    |-- elapsed     0.0000 ms
    |-- # calls
    |-- average    0.000000
    |-- size [KB]
        |-- total        0.0000
        |-- average    0.000000
|-- grayscale big
    |-- elapsed     0.0000 ms
    |-- # calls
    |-- average    0.000000
|-- grayscale small
    |-- elapsed     0.0000 ms
    |-- # calls
    |-- average    0.000000
|-- Time for image acquisition
    |-- elapsed  2271.3750 ms
    |-- # calls        17
    |-- average 133.610294
|-- Time for getting a usable image
    |-- elapsed  2290.8733 ms
    |-- # calls        34
    |-- average  67.378626
|-- Time for PSimilarity detection
    |-- elapsed  6302.9531 ms
    |-- # calls        17
    |-- average 370.761949
|-- big images PSimilarity detection
    |-- elapsed  5062.3960 ms
    |-- # calls        17
    |-- average 297.788000
|-- small images PSimilarity detection
    |-- elapsed  1240.4220 ms
```

```
    |-- # calls        17
    |-- average  72.966000
|-- Time for Panoramic Detection
    |-- elapsed 14718.9922 ms
    |-- # calls         1
    |-- average 14718.992188
|-- Time for Landmarks Filtering
    |-- elapsed     0.0200 ms
    |-- # calls         1
    |-- average   0.020000
|-- Time for Landmarks Triangulation
    |-- elapsed     0.0020 ms
    |-- # calls         1
    |-- average   0.002000
|-- Total time
    |-- elapsed 14719.7090 ms
    |-- # calls         1
    |-- average 14719.708984
|-- Total turret turning time
    |-- elapsed  5539.0269 ms
    |-- # calls        18
    |-- average 307.723714
|-- first turning
    |-- elapsed   650.0291 ms
    |-- # calls         1
    |-- average 650.029053
|-- first turning
    |-- elapsed     0.0000 ms
    |-- # calls
    |-- average   0.000000
|-- issuing turning commands
    |-- elapsed  4888.9160 ms
    |-- # calls        17
    |-- average 287.583295
|-- waiting for turret to finish
    |-- elapsed     0.0000 ms
    |-- # calls
    |-- average   0.000000
```

Appendix D

Installation tutorial

In this section a sample installation for all the libraries needed by our application is presented. It is pretty straightforward and may be performed in very few steps. Commands are explained when they are used. A complete script is presented at the end of this section.

Tutorial

1. Setting things up

   (a) You should be `root` and start in your own directory:
   ```
   su
   cd
   ```

   (b) These packages should be here: `commoncpp2-1.0.9.tar.gz`, `PSimil-devel-1.00.tar.gz` `nono-2.07.tar.gz`, `koala-2.04.tar.gz` and `nonovideo-0.9.tar.gz`. If any of these packages are missing, copy them into the root directory. For me, it worke out by running the following command:
   ```
   [root@quadx2 root]# cp /usr/users/staginfo/aiosup/*tar.gz .
   ```
   Note the point at the end of the command.

   (c) If older versions of the libraries are installed, for instance `koala-2.03`, you should `first` uninstall them. For the koala-2.03 example, this may be done by running:
   ```
   [root@quadx2 root]# cd koala-2.03/
   [root@quadx2 koala-2.03]# make uninstall
   [root@quadx2 koala-2.03]# cd ..
   [root@quadx2 root]# rm -rf koala-2.03/
   ```

2. Performing the actual install

   common c++ This is the tricky part. First, you need to unpack the library's files:
   ```
   tar zxvf commoncpp2-1.0.9.tar.gz
   ```
   Before starting to compile, the current library configuration should be refreshed:
   ```
   /sbin/ldconfig
   ```
   The compiling procedure may now proceed:

```
cd commoncpp2-1.0.9
./configure; make install
```

Now, a symbolic link must be created for the `cc++2` directory:

```
ln -s cc++2/cc++ cc++
```

Right now, a new system libraries configuration refresh should be performed:

```
/sbin/ldconfig
```

PSimil   This is the library used to detect PSimilar landmarks for a given image. Its installation is very simple:

```
tar zxvf PSimil-devel-1.00.tar.gz
cd PSimil-devel-1.00
./configure; make install
```

nono   This library controls the low-level features of the robot. It is used by both the `koala` and `nonovideo` libraries in order to communicate with the robot. To install just run:

```
tar zxvf nono-2.07.tar.gz
cd nono-2.07
./configure; make install
```

nonovideo   This library controls all the image-acquisition features for the `Koala` robot. It uses the nono library to communicate with the robot. To install you have to run:

```
tar zxvf nonovideo-0.9.tar.gz
cd nonovideo-0.9
./configure; make install
```

koala   This library ensures a high-level control of the `Koala` robot. Commands like *go forward for 50 centimeters* or *rotate camera against the y axis for 45 degrees* are possible. To install this library you have to run:

```
tar zxvf koala-2.04.tar.gz
cd koala-2.04
./configure; make install
```

The complete script   For this script you need to be root and run from your personal directory, where all the libs have to be placed (see the tutorial's first step to see which they are). Then you have to run:

```
tar zxvf commoncpp2-1.0.9.tar.gz; cd commoncpp2-1.0.9; ./configure;
make install; cd ..; rm -f commoncpp2-1.0.9.tar.gz; cd /usr/local/include;
```

```
ln -s cc++2/cc++ cc++; ls -la | grep cc++; cd; /sbin/ldconfig

tar zxvf PSimil-devel-1.00.tar.gz; cd PSimil-devel-1.00;
./configure; make install; cd ..; rm -f PSimil-devel-1.00.tar.gz

tar zxvf nono-2.07.tar.gz;cd nono-2.07; ./configure;
make install; cd ..; rm -f nono-2.07.tar.gz

tar zxvf koala-2.04.tar.gz;cd koala-2.04; ./configure;
make install; cd ..; rm -f koala-2.04.tar.gz

tar zxvf nonovideo-0.9.tar.gz;cd nonovideo-0.9; ./configure;
make install; cd ..; rm -f nonovideo-0.9.tar.gz
```

Index