An XML Virtual Machine for Distributed Image-Based Localization for Mobile Robots

E. Menegatti[2], *E. Mumolo*[1], *E. Pagello*[2], *F. Seriani*[1]
[1] DEEI, Universita' degli Studi di Trieste,
34127 Trieste, Italy, email: mumolo@univ.trieste.it
[2] DEI, Universita' degli Studi di Padova, Italy

Sommario

In questo articolo descriviamo un sistema di calcolo distribuito che localizza un robot che si muove autonomamente in un ambiente chiuso. La localizzazione viene fatta mediante confronto tra immagini omnidirezionali. Le immagini vengono rappresentate nel dominio spettrale ed il confronto é realizzato mediante una norma L_1. Il sistema distribuito utilizza XML per descrivere gli algoritmi che vengono distribuiti attraverso i nodi remoti. Il parsing dei documenti viene effettuato con il parser di Apache Xerces e l'interprete é scritto in Java. Nell'articolo sono riportati alcuni risultati sperimentali rappresentativi delle prestazioni del sistema.

Abstract

This paper describes a software system to perform distributing image-based localization for mobile robots navigation according to the 'Grid Computing' paradigm. Omnidirectional images are acquired in the environment where the robot moves, represented by Fourier signatures and compared to a set of images, acquired previously in given points of the environment. Comparison between images is based on L_1 norm. The distributed system has been realized by describing the operations using XML documents distributed among the remote nodes with the XML-RPC protocol. An assembly-like language, called XML-VM, has been developed in XML. Parsing of the XML documents is based on the parser of Apache Xerces while the interpret is written in JAVA. Representative experimental results are reported.

Keywords: Mobile robot, omnidirectional images, XML, distributed computing, Fork/Join.

1 Introduction

This paper deals with the problem of localizing a mobile robot by visual landmarks. The localization problem is clearly one of the most important problems in autonomous mobile robotics, as it is fundamental in tasks such as navigation and map building. We use a set of omindirectional images acquired at given points of the environment as a basis for localization. The localization process is performed by comparing the current image viewed by the mobile robot to the pre-acquired reference images of the set. The best matching image indicates, with a given likelihood, where the robot is located. Images are represented by Fourier signatures, i.e. the Fourier transformations of each line of the images. Of course, omnidirectional represent all the robot view in one shot; hence, there is no need to rotate the camera. Moreover, it is straightforward to see that Fourier signatures are invariant to image rotations, so the orientation of the robot does not need to be taken in consideration in the matching phase. Finally, representing the image by Fourier signatures greatly reduces the amount of data needed to represent the image. Images O_i and O_j are compared by means of the distance described in (1).

$$D(O_i, O_j) = \sum_{t=0}^{L-1} \sum_{k=0}^{M-1} |F_{it}(k) - F_{jt}(k)| \qquad (1)$$

where F_{it} is the module of the Fourier transform of the t-th row of image O_i and k represent frequency.

However, although the amount of data is reduced, the computations required by (1) can be quite demanding if the number of reference images is high. For this reason, a distributed system has been deviced as described in this paper.

This paper is structured as follows. Section 2 deals with the state of the art, describing in particular a popular Grid system and highlighting its positive aspects and defects. In Section 3 we describe the architecture of the system. In Section 4 we summarize the XML-VM language, while in Section 5 we describe some information on the parsing and interpretation of a XML-VM program. In Section 6 some experimental result are shown.

In Section 7 some final remarks are reported.

2 A distributed system for image signatures comparison

In the last decade there has been an increasing interest in the development of systems for distributed computing aiming at sharing computing resources available on a large scale. These systems exploit the unused CPU cycles of a potentially enormous number of computers available in internet, conveying to a final user a large computing power at a very low cost. These implementations originated the computing paradigm known as "GRID Computing" [1]. The work described in this paper represents a preliminary effort towars the goal of building a GRID, in the sense that we reduced the number of involved computers to a smaller number of networked machines.

Generally speaking, the computing environments for GRID computing are usually very heterogeneous from an hardware and software point of views; thus the first problem to be faced is the necessity of developing virtual machines independent from the various platforms. Of course, the most important characteristics of a virtual machine, which must be considered with care, are the easiness of use, performance, security and scalability. One of the currently most popular virtual machines is Java. In fact, the Java virtual machine has been designed for running in different computing environments, from dedicated systems to general purpose machines. However, the Java virtual machine does not behave particularly well against attacks or intrusions. Moreover, the execution time of a Java process is not deterministic, mainly because of the memory management.

However, the image-based localization problem is inherently real-time, in the sense that the image matching results should be available within a deadline imposed by the navigation module. In a GRID framework, deadline constraints can be mainly taken into consideration by balancing the load of the involved machines in order to be sure that a remote method can have a sufficient computing power for its execution, by choosing a real-time language without garbage collection and by assuring that the network loading is low, to reduce communication times. In this work, we used a cluster of 16 computers made available for this work from the Parallel Programming Lab of the University of Karlsruhe, Germany. Each machine is equipped with Pentium III biprocessors at 800 MHz, with 512 Mbyte of memory and running a Linux Operating System. In this work, the load of the machines were not monitored but we supposed that the load were not so high because the average number of processes running on the machines were quite low. For the same reason, and also because the machines communicate only inside the network, the load of the network is low. The system was used to experiment the XML-VM language described below.

In addition to the cluster described above, a machine located in the Trieste Lab has been used to represent the robot. The machine used a Pentium II processor at 400 MHz and runs a Windows 2000 Operating System. The configuration of the system is described in Fig.1.

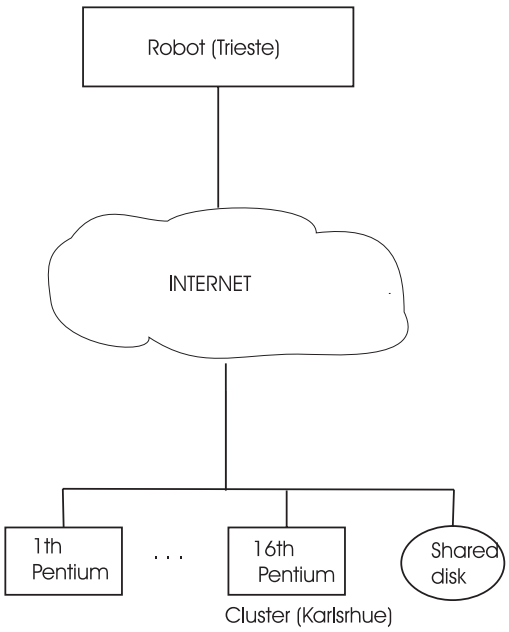

Figure 1: System architecture

Generally speaking, there are two main approaches for bulding a distributed programming system based on Java virtual machines [10]. One is to give the programmer an unique environment in which the threads are distributed on the different nodes by the operating system. This solution is quite complex to develop, since many problems arise concerning both implementation and performance. Projects in this area include the IBM cluster VM for Java, the Kaffe virtual machine and the JDSM [9].

Other solutions are based on the development of communication mechanisms such as, for example, message passing. A typical approach is RMI (Remote Method Invocation). Other approaches are based on extensions of Java with parallel programming linguistic constructs. An example of the latter approach is JavaParty, developed at the University of Karlsruhe [5]. JavaParty introduces the definition of a 'remote' class. A remote class

can be used by all the remote virtual machines of the distributed system. A remote class can be used like a standard Java class, but it can interact with other classes independently to the location. The typical use of JavaParty is described in Fig.2. More precisely, the main method, which runs on a given node, creates a number of threads on the same node. Each of the threads instantiates an object of a 'remote' class using a 'new' statement. At this point the object is distributed on a remote node. A method of the class is then started in the standard way. The threads are then joined together to collect the results. It is worth noting that there are three ways to distribute the remote objects, namely randomly, at fixed addresses and letting the system deciding the remote addresses.

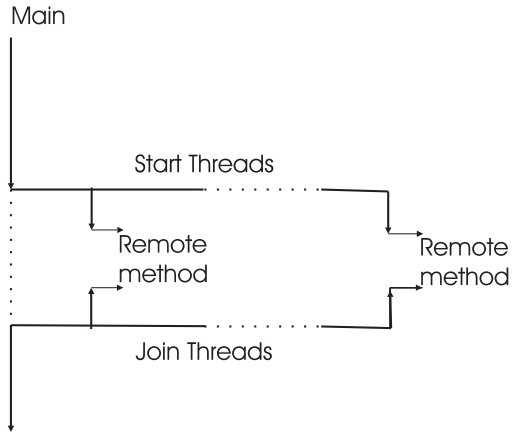

Figure 2: JavaParty remote method invocation

Of course the main advantage of JavaParty is that a sequential code can be run in a distributed computing environment using a few modifications. Let us consider however the speed-up results reported in Fig.3. The *speedup*(n) is a function of n, where n is the number of renote nodes, and it is computed as the ratio between the execution time of one program on one node and the execution time of the same program on n nodes. In this way, *speedup*(1) is equal to the execution time on a GRID made by 1 remote node, *speedup*(2) represents the increment of the time of a GRID made of 2 remote nodes, which is ideally twice the previous one, and so forth. The ideal speedup curve is therefore a 45 degrees line. In the initial speedup measurements, two programs have been used: the distributed computation of the first N Fourier coefficients of the function $f(x) = (x+1)^x$, $x \in [0, 2]$, and the multiplication of two sparse matrices.

The results shown in Fig.3 are far from the theoretical performances represented by the diamonds curve. This is mainly because of the thread join operations (the remote method running on the more loaded node make the other threads waiting for its completion), but also because a method may behave non deterministically and also due to the distribution mechanism of the methods on the remote nodes. Even if the loading on the remote nodes were accurately balanced, it should not be possible to have a deterministic behaviour. In conclusion, a real time distributed operation is not possible with Java. For this reason, a new virtual machine has been developed. In the future we build an interpret of the virtual machine using a high level language like C and by disabling all the conditions which can cause non determinism. For now, however, the interpret has been written in Java. Therefore, the current system may be used only for comparison of the distribution mechanism based on XML-RPC with the execution efficiency obtained with JavaParty.

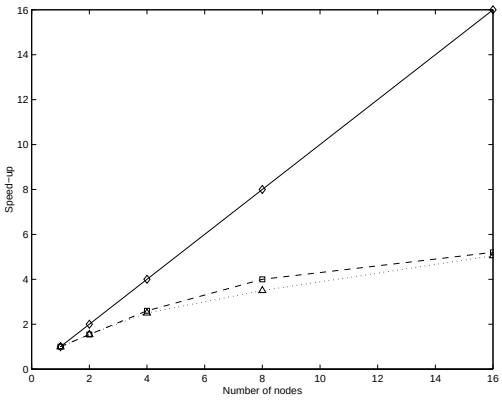

Figure 3: Speed ups measured with JavaParty: in diamond the ideal curve, in triangle the matrix multiplication, in squares the Fourier benckmark.

3 System architecture

The distributed system described in this paper and described in Fig.4 is structured in a peer-to-peer style, limiting the tasks of the central node of the system mainly to the activation of the remote methods, the collection of the result and the measurements of the performances of the system.

The central node does not know nearly anything about what is happening in the remote nodes, where the computation effectively takes place. Each algorithm decides if and when to call other remote nodes and the method to execute. On the central node a particular daemon is running, called 'name resolving daemon', which knows what remote nodes are available. When a generic node 'A' needs to fork a procedure on a remote node, it

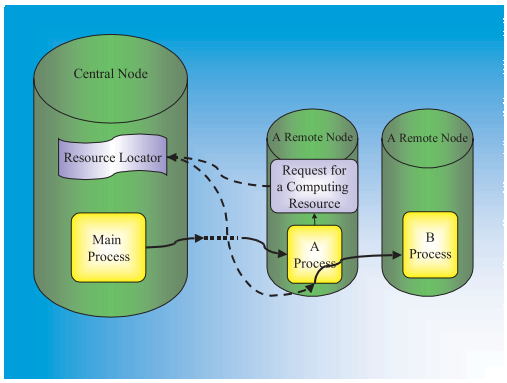

Figure 4: Architecture of the distributed system

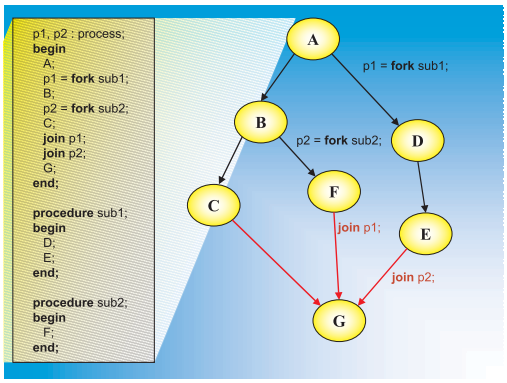

Figure 5: Example of a Fork/Join concurrent processes with a predefined data type

calls the central node for determining the address of an available remote node. At this point, node 'A' contacts directly the remote node for sending information on the distribution of the XML-VM code and its remote execution; this procedure is executed each time a remote call is needed. The remote node is therefore informed about what XML document it has to download and interpret, and downloads the related code from the central node, which acts as a Web Server.

Clearly, node 'A' must join the conclusion of the remote call by waiting for the return of results. This procedure is implemented through the use of the linguistic framework Fork/Join.

The linguistic framework Fork/Join has been introduced by M.Conway [3] and J.Dennis [4]. Starting from the initial definition, many programming languages used the Fork/Join concept in several ways. The Fork/Join operations has been largely studied from a queueing point of view [2, 8, 7]. Fork generates a concurrent thread of execution, while the Join waits for termination; in this way it is possible to build concurrent processes. In Fig.5 a system of concurrent processes is shown using an interpretation of the Fork operation based on a data type defined by the language, process, which is used as an operand of Join to specify the process to synchronize with. A similar approach for the implementation of Fork/Join has been used in this work.

4 The language XML-VM

In this Section we will summarize the main characteristics of the XML-VM language. First of all, it is worth noting that two sets of memory are generated, declared in Java as Array of Object, which simulate registry and a virtual disk available to the virtual machine. The registry is constituted by 32 cells numbered from 0 to 31, while the virtual disk is constituted by 10000 blocks of data, numbered from 0 to 9999.

All the data related operations take place on the registers. The numbers must be stored in two data cells of registry. There are no variables, and every operation must be performed specifying the involving register's cells; from this point of view, XML-VM is an assembler-like language. The storage of data on the virtual disk is performed exclusively through the STORE instruction. Instead, the LOAD instruction is the only instruction that allows to copy the content of the cells of the virtual disk into the registers. Ten different data types are implemented in the language; nine of them follow the Java data types: int, long, short, byte, float, double, boolean, char and string while the tenth data type, defined in XML-VM as "index", represents a pointer to another data cell in the virtual disk or in a register. The index data type can be used by the load, store and procedure call operations; moreover, this type is fundamental for parameter passing to the remote methods. The syntax of the language exploits the use of tags attributes as integrating parts of the instructions, while nesting of tags is rare. This decision was taken in order to facilitate the writing of XML-VM code. A simplified description of the instructions of XML-VM, grouped by type, is given below.

4.1 Mathematical tags

- <ADD target="r1" first="r2" second="r3" />
- <SUB target="r1" first="r2" second="r3" />

The Tags execute the sum (subtraction) of the registers reported as arguments, according to the involved data types.

- <CONV register="r1" target="r2" to="type" />

The Tag executes the conversion of a data contained in one register to the specified type. It reports conversion errors: if it is asked to convert a String in a number, the execution is interrupted and an error is notified.

- <DIV first="r1" second="r2" result="r3" rest="r4"/>

This Tag executes the division between two numbers; if the two numbers are integer, the register specified by the attribute "rest" contains the rest of the division.

- <ELEV target="r1" register="r2" exponent="r3"/>

The Tag performs the exponentiation operation register**exponent. It reports errors on the types of the involved data: if one of the two operators does not contain a number, but, for example, a String, the execution is interrupted and an error is notified.

- <MUL target="r1" first="r2" second= "r3"/>

The Tag executes the multiplication between two numbers. It reports errors on the types of the involved data.

- <FFT from="r1" to="r2"/>

This Tag perform a Fourier transform.

4.2 Data movement tags

- <LOAD register="r1" index="m1"/>

The Tag loads the specified register with the content of the specified cell of the virtual disk.

- <MOVE target="r1" source="r2"/>

The Tag moves the source to the target.

- <STORE to="m1" type="type"> 'value' </STORE>

This Tag stores in the "m1" location of the virtual disk, the data whose value and data type are specified respectively in the field "value" and by the attribute "type".

- <LOADIMAGE type="xxx" index="x" to="y"/>

This tag reads the image whise name is pointed to by the "x" cell and writes its content in the virtual disk. The image can be "fft" or "ppm" depending on the type of data, namely, a Fourier signature file or an image file.

- <STOREIMAGE type="xxx" index="x" from="y"/>

This tag performs the following operations: the image of the type indicated (fft or ppm as before) is written synchronously on the remote disk. The name of the remote file is pointed by to the "x" cell and the pointer to the matrix data is stored in the "x" cell. On the remote side, a daemon server waits for this type of requests.

4.3 Logical tags

- <CMP first="r1" second="r2"/>

The Tag compares first with second registers.

- <JEQ to="label"/>

The Tag performs a jump if equal.

- <JGR to="label"/>

The Tag performs a jump if greater.

- <JNEQ to="label"/>

The Tag performs a jump if not equal.

- <JNGR to="label"/>

The Tag performs a jump if not greater.

4.4 Procedure call tags

- <CALL ip="IPINDEX:NPORT" file="Path/name.xml" name="name" to="m5-m7"> <PARAM> r2 </PARAM> <PARAM> m15 </PARAM> ... </CALL>

The Tag executes the remote synchronous call: the calling node is blocked until the results from the remote node are received. The attributes IP, Path, file aim at addressing the remote procedure. The arguments indicated in the <PARAM> section are passed through XML-RPC to the remote node, while the "to=" attribute indicates where the results from the remote procedure are stored.

<FORK id="N02" file="Path/name.xml" name="nome" to="r5-r7" clone="m5-m7"/>

The Tag executes the remote asynchronous call. The attribute "id=" selects the remote node, the attributes "file", "name" addresses the procedure to execute, the attribute "to=" indicated where the results are stored and the "clone" field indicate the arguments to pass to the remote node.

- <JOIN to="r5-r7"/>

The Tag executes the synchronization of two processes. The calling procedures waits until the registers indicated in the "to" attributes contain results.

- <LOCALCALL name="name" to="m5-m7"> <PARAM> r2 </PARAM> <PARAM> m15 </PARAM> ... </LOCALCALL>

The Tag executes the local call.

- <RETURN from="m5-m7"/>

The Tag represents the term of one procedure recalled from a CALL, a FORK or from a LOCALCALL.

4.5 Miscellanea tags

- <LABEL name="name"/>

The Tag marks the position through a label, that can be recalled from a Tag of conditioned jump.

- <QUIT/>

The Tag marks the end of a document XML-VM.

- <START/>

The Tag marks the begin of a document XML-VM "stand-alone".

- <STRUCT/>

This Tag marks the beginning of a document XML-VM not "stand-alone"; this means that the document XML-VM that contains this Tag can be used only by another document through the remote calls.

5 Parsing and interpretation

The parser performs a complete analysis of the XML-VM document, expanding all the tags, attributes, values. One of the most complete and used XML parser is the Apache Xerces XML Parser. Xerces supports SAX 1.0 and 2.0; SAX stands for Simple Api for XML. Once we completed the first version of XML-VM virtual machine, embedding the XML-RPC protocol for communication between remote nodes, we noticed a decrease of performance due to the slowness of the XML parser. In order to overcome this overhead we decided to use the same parser chosen by the developers of Apache XML-RPC, that is MinML, a light and fast SAX 1.0 parser. MinML is very fast and it has been integrated in our system. The high performance obtained with MinML is mainly due to the fact that the parser does not process the Data Type Definition (DTD). The interpreter of XML-VM language has been written in Java for portability reasons; the interpreter executes the action associated to the XML tags as they are analyzed by the parser. A pseudo-code of the interpreter is outlined in the Appendix. Let us consider the following steps needed to operate the system: 1) it is necessary to install on each machine the XML-VM virtual machine; the virtual machine is started and works as a service, waiting for remote requests. 2) on the central node, we install the same XML-VM virtual machine and the name resolving daemon used by remote nodes 3) the sources of the algorithms to be executed are published through a web server reachable by all the nodes 4) now we are ready to launch the distributed execution of the program.

The pseudo-code of the XML-VM interpreter is reported below.

```
public Object startExecution (Object[] arg,
xmlvmMachine Machine) throws Exception
{
  Initialize XML-VM stack, registers,
  virtual disk;
  Verify the XML document;
  try {
     FOR(all the XML-VM tags) {
       If (Tag in [ADD, LOAD, MOV, STORE, SUB,
           MUL, DIV, COMP, LOCALCALL, CALL,
           FORK, JOIN, CONV, ELEV, OPER, FFT,
           LOADIMAGE, STOREIMAGE, RANDOM])
       Then {
         Call the related procedure;
       }
       else {
         Process the other instructions;
       }
     }
   } catch(Exception e)
   If (non ending with QUIT)
   Then error;
   Log the execution times;
}
```

6 Experimental results

The goal of the experimental analysis is to study the performance of the Grid computing system presented in this paper. In general, the efficiency of a distributed application is related to various factors, including: the network speed, the load of the remote nodes, the homogeneity of the machines which participate to the Grid, the degree of parallelism of the algorithm, the protocol used for method distribution.

The first benchmark used to test XML-VM is the Fourier benchmark. The results obtained with XML-VM are slightly better than that obtained with Java-Party, reported in Fig.3 with squares. XML-VM was then used to realize the image comparison system. The XML document interpreted on the local machine is summarized below.

```
<?xml version='1.0'?>
<XMLVM>
<START>
<!-- Initialization tags -->
<LOADIMAGE type="ppm" index="10" to="5" />
<LOAD register="r1" index="5"/>
<FFT from="r1" to "r2"/>
```

```
<STOREIMAGE type="fft" index="11" from="r2"/>
<FORK id="N01"
     file="http://140.105.50.110:80/Texel.xml"
     name=Texel" to "m8[m7]" clone="m9[m6]"/>
. . .
<FORK id="N16"
     file="http://140.105.50.110:80/Texel.xml"
     name=Texel" to "m8[m7]" clone="m9[m6]"/>
. . .
<JOIN topointed="m5[m4]"/>
<!-- next tags extract from the output cells
     best images with related distances -->
<!-- other tags sort the images with
     increasing distances -->
<QUIT/>
</XMLVM>
```

Clearly, the role of Fork is to start a section of the algorithm, in parallel to other sections, on a remote node whose address is chosen in a suitable manner.

The acquired image is first represented as Fourier signatures with the tag

```
 <FFT from="r1" to "r2"/>
```

and then is transmitted to the remote cluster where it will appear as a shared file. The writing is synchronous, so that the image comparisons which are performed thereafter can be sure to use a stable image. The FORK tags distribute the comparison tasks on the remote nodes. The cluster nodes access, through the shared disk, all the reference images previously acquired and the unknown image, all of them are given as Fourier signatures. The method "Texel.xml" performs a small number of comparisons, from 3 to 9. Since sixteen remote methods are distributed, this corresponds to 48 to 144 image comparisons.

The algorithm which runs on each remote node is described below in XML-VM:

```
<?xml version='1.0'?>
<XMLVM>
<STRUCT/>
<LABEL name="Texel" />
<!-- initialization tags -->
<LOADIMAGE type="fft" index="13" to "4">
<!--the following tags compute distance -->
<!--as indicated as follows in pseudocode-->
<!-- For the i-th reference image,  -->
<|--   <LOADIMAGE i-th/>            -->
<!--   For each row                 -->
<!--     For each column            -->
<!--        Accumulate distance     -->
<RETURN results/>
<QUIT/>
</XMLVM>
```

The function of a Fork instruction is similar to the function of a For instruction, such that simply varying the value of the constants (number to add and workload for node) it automatically varies also the number of calls and the parameters passed to the remote node. Clearly, the computing time is dependent on the number of machines used in parallel for the elaboration of the algorithm. What it is expected is an hyperbolic behavior, since the T(n) function should be of the type 1/n, where n is the number of machines involved in the distributed computation. The measured values are shown in Fig. 6 and Fig. 7. In Fig. 6, the absolute execution time versus the number of remote nodes is shown.

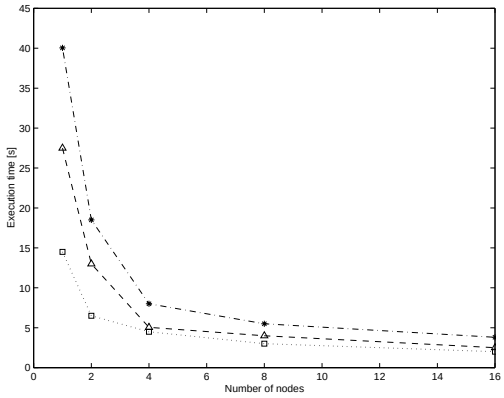

Figure 6: Absolute time versus number of nodes for the comparison of 48, 96 and 144 images

In Fig. 6, the computations with 3, 6 and 9 image per remote method "Texel.xml" is indicated with points, triangles and squares respectively. As it can be observed, the experimental measurements of Fig. 6 confirm the expected theoretical results: the absolute time versus the number of machines follows a hyperbolic curve. With 16 nodes, the comparison absolute time is approximately 2, 2.5 and 4 seconds for 48, 96 and 144 comparisons respectively. In Fig.7 the speedups related to the XML system are shown.

Again, with dots, triangles and squares the results pertaining to 48, 96 and 144 total comparisons respectively are represented. The last picture gives rise to several comments. First of all the results for a different number of comparisons, mainly for 96 and 144 images, are very irregular. Second, it appears that in same case, for example with four nodes, the speedup is greater than the ideal curve. The irregularity of the results are due to the different loading factor of all the nodes but also for another reason, that is for caching. In fact, in the first

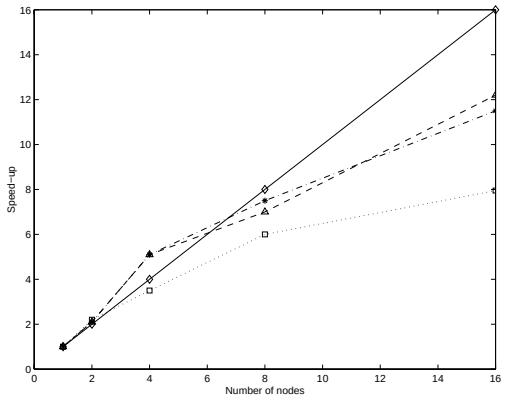

Figure 7: Speedups versus the number of remote nodes

run the first machine have to load all the required reference images from the disk, thus requiring a high number of disk I/O. If the number of nodes increases, some images are already loaded in the system and threfore the computation time is reduced, leading to a speed-up greater than the ideal one, which is based on the first run, which comprises all the I/O requests. As a second remark, slower machines in the GRID quickly lead to a performance decrement of the whole system down to the speed of the slower machines.

7 Final remarks and conclusion

In this paper we dealt with the problem of designing and developing an efficient architecture based on XML for realizing a distributed image-based localization for mobile robots based on the Grid approach. In this work we use XML for describing an algorithm. By means of XML it is possible to realize an efficient Grid system; the distribution of the algorithms, which is done by means of HTTP protocols, is very fast, the code execution is quickly performed with an interpreter which was written in JAVA, the scalability of the system is realized efficiently by means of the name resolving daemon. Due to these reasons, the system performance compare very favorably with other solutions based on the distribution of Java code [6]. An XML-VM language has been developed in order to describe algorithms, and an XML-VM interpreter has been written in order to execute these algorithms. Each remote machine runs locally the parser and the interpreter. Many problems are still open. For example, the distribution of the workload, which is related to the choice of the nodes where the tasks are distributed, has not been considered. Another open aspect is the fault tolerance of the system. Similarly, we did not consider the problems related to the programming of complex algorithms; this last problem can be mitigated by the development of a compiler from a high-level language to XML-VM.

Acknowledge

The authors wish to thank Dr. Bernhard Haumacher of the Parallel Programming Lab of the University of Karlsruhe for making available to us the cluster of machines used for implementing the system.

References

[1] C. E. Catlett, J. Toole, "Testbeds: From Research to Infrastructure", in "The Grid: Blueprint for a New Computing Infrastructure," Ian Foster and Carl Kesselman, ed., Morgan Kaufmann, August 1998.

[2] Ray Jinzhu Chen, "A Hybrid Solution of Fork/Join Synchronization in Parallel Queues", IEEE Transactions on Parallel and Distributed Systems 12(8), August 2001

[3] M Conway, "Multiprocessing system design", Proc. Of the AFIPS Fall Computer Conf., 1963

[4] J.G.Dennis, E.C.Van Horn, "Programming semantics for multiprogramming computations", Communications of ACM, March 1966

[5] http://www.ipd.uka.de/JavaParty/features.html

[6] Doug Lea, "A Java Fork/Join Framework", ACM Java Grande 2000 Conference, June 3-5 2000

[7] Y.C.Liu, H.G.Peros, "A Decomposition Procedure for the Analysis of a Closed Fork/Join Queuing System", IEEE Transactions on Computers, vol.40, n.3, march 1991

[8] R.Nelson, A.N.Tantawi, "Approximate analysis of Fork/Join Synchronization in Parallel Queues", IEEE Transactions on Computers, vol37, n.6, June 1988

[9] Y.Sohda, H.Nakada, S.Matsuoka, "Implementation of a Portable Ssoftware DSM in Java", ACM Java-Grande Int. Conference, June 2001

[10] M.Surdeanu, D.Moldovan, "Design and Performance Analysis of a Distributed Java Virtual Machine", IEEE Transactions on Parallel and Distributed Systems, Vol.13, N.6, June 2002