View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Archivio istituzionale della ricerca - Università di Trieste

# XML-VM: An XML-Based Grid Computing Middleware

Alfredo Cuzzocrea[1], Enzo Mumolo[2], Marco Tessarotto[2], Danilo Amendola[3]
[1] DIA Dept., University of Trieste and ICAR-CNR, Italy
alfredo.cuzzocrea@dia.units.it
[2] DIA Dept., University of Trieste, Italy
mumolo@units.it; marco.tessarotto@regione.fvg.it
[3] IRCCS Bonino-Pulejo, Messina, Italy
danilo.amendola@gmail.com

**Abstract.** This paper describes a novel distributing computing middleware named XML-VM. Its architecture is inspired by the 'Grid Computing' paradigm. The proposed system improves many characteristics of previous Grid systems, in particular the description of the distributed computation, the distribution of the code and the execution times. XML is a markup language commonly used to interchange arbitrary data over the Internet. The idea behind this work is to use XML to describe algorithms; XML documents are distributed by means of XML-RPC, interpreted and executed using virtual machines. XML-VM is an assembly-like language, coded in XML. Parsing of XML-VM programs is performed with a fast SAX parser for JAVA. XML-VM interpreter is coded in JAVA. Several algorithms are written in XML-VM and executed in a distributed environment. Representative experimental results are reported.

## 1 Introduction

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. At a smaller scale, they exploit the unused CPU cycles of the computers available in the current intranet. These implementations originated the computing paradigm known as "GRID Computing" [1-4].

Grid computing aims at creating the illusion of a simple yet powerful virtual computer. Actually, the computing power is provided by a large collection of connected systems. There are many important applications requiring a large amount of processing power, for example systems for weather prediction using computational models, systems for the solution of theoretical physics and astronomy problems, simulations of complex systems, financial markets prediction systems and many other. The computing environments for GRID computing are usually very heterogeneous from a hardware and software point of views; thus, the first problem to be faced is the necessity of developing virtual machines to make the computation infrastructure independent from the various platforms. Of course, the most important characteristics of a virtual machine are the easiness of use, performance, security and scalability. Java is one of the most popular virtual machine. The Java virtual machine has been designed for running in various computing environments, from dedicated systems to general-purpose machines. However, generally Java requires a remarkable amount of computing and memory resources for compilation and execution.

One of the problems of the known Grid systems is that they are typically built around a monolithic architecture: the remote computing node selected to execute a certain application must also execute locally the verification functions, the security management, compilation and optimisation. Consequently, these monolithic architectures generally are limited in terms of security and scalability, opening to a large field of research [5]. Other known problems come from the lack of grid enabled software, making it difficult to easily obtain software solutions.

This paper proposes a contribution in this field of research, developing a middleware that hides to the programmer the complexity of the available distributed architecture. The issues that we have treated designing the middleware are system performance, addressed by means of efficient distribution and interpretation tools, and the security, addressed using of SSH protocols. The architecture we propose in this paper is based on the XML meta-language, which is a programming language normally used for describing data structures. The key point of this work is that rather than using XML to describe data we use XML to describe algorithms. Distribution of XML-VM code is performed using XML-RPC high performance protocol. Parsing and interpretation of XML-VM documents is performed in JAVA.
Using XML as a framework to describe both data and algorithms we provide a common base that different platforms can manage in different ways. One way may be to use transformation sheets, and another way is to write an interpret using an appropriate programming language.

This paper is structured as follows. Section 2 deals with the state of the art, describing in particular a representative Grid system 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 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 Related Work

This project is inspired by the large scale distributed computing systems for research purposes developed over the years. The Great Internet Mersenne Prime Search (GIMPS) project [6] started in 1996. The goal of GIMPS is to find Mersenne primes, namely prime number of the form $M_n = 2^n - 1$ for some integer $n$, using computers connected via internet. The SETI@Home project [7] started in 1997 to identify non-random radio patterns generated by some form of intelligent life, excluding at the same time random signal patterns generated by natural phenomena like stars and supernovas. The Einstein@Home project [8] was officially launched in 2005 for searching signals from rotating neutron stars using gravitational-wave detectors. In each of these projects, huge amount of data is processed using a parallel computing approach: data is broken in small packets, and each packet is sent to the computers distributed across the Internet which offer their free CPU times for the processing. This is done using a special software downloaded from a web site. Usually this software acts as a traditional screen saver, however it contains the implementation of the data processing algorithms and the procedures for receiving data and transmitting results to the main server. The algorithms implemented on the GRID are always the same, while the data to be processed is varying; such a SIMD model is only suitable to particular computations. Namely, if we do not have a completely separable

problem, or if the routines being executed on the data are varying, or if a single node needs additional remote tasks, the reported GRID computational model becomes absolutely inadequate.

Other approaches for building a distributed programming system are based on Java virtual machines [9]. Some approaches 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 following this approach include the IBM cluster VM for Java, the Kaffe virtual machine and the JDSM [10].

Other approaches are based on the development of communication mechanisms such as, for example, message passing. A typical approach of this kind is RMI (Remote Method Invocation). Other approaches are based on extensions of Java with parallel programming linguistic constructs. An example of this latter approach is JavaParty, developed at the University of Karlsruhe [11].

Other approaches are based on CORBA [12]. Miro [13] is an object oriented layered client/server robot middleware system based on ACE [14] and the associated real-time CORBA ORB, TAO.

The middleware described in this paper is of SIMD type. One of its feature is that it is able to distribute the computing load transparently to the programmer.

## 3 System Architecture

The architecture of the distributed system described in this paper, namely XML-VM, is depicted in Fig.1. XML-VM is structured according to a peer-to-peer principles. A central node provides a simple terminal for 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 node 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 resolve daemon', which knows the IP of the available remote nodes. A generic node 'A', when it needs to fork a procedure on a remote node, calls the central node for the individuation and resolution of the address of an available remote node. At this point, node 'A' contacts directly the remote node for the execution of an algorithm; this procedure is executed each time a remote call is needed. Clearly, node 'A' must join the conclusion of the remote call by waiting for the return of results. In our system, the distribution of the XML code on the distributed nodes is performed using the Fork statement.

```
<FORK id clone results>
        ...data and algorithms to be executed in the remote node,
        expressed in XML-VM...
</FORK>
```

Fork implements the following actions: first, an available node is sought in the local table, then the code and data are sent to the remote node with XML-RPC.

Synchronization is performed using the Join statement. The Join tag has the following syntax: <JOIN id/> and implements the following operation: waits for the

termination of the remote node and returns the results to the calling environment using the XML-RPC response.

## 4 The Fork/Join linguistic framework

The Fork/Join linguistic framework has been introduced by M. Conway [15] and J. Dennis [16] in the '60. 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 [17-19]. The Fork/Join linguistic framework is available in Java [20]. Fork generates a concurrent thread of execution, while the Join waits for its termination; in this way it is possible to build concurrency.

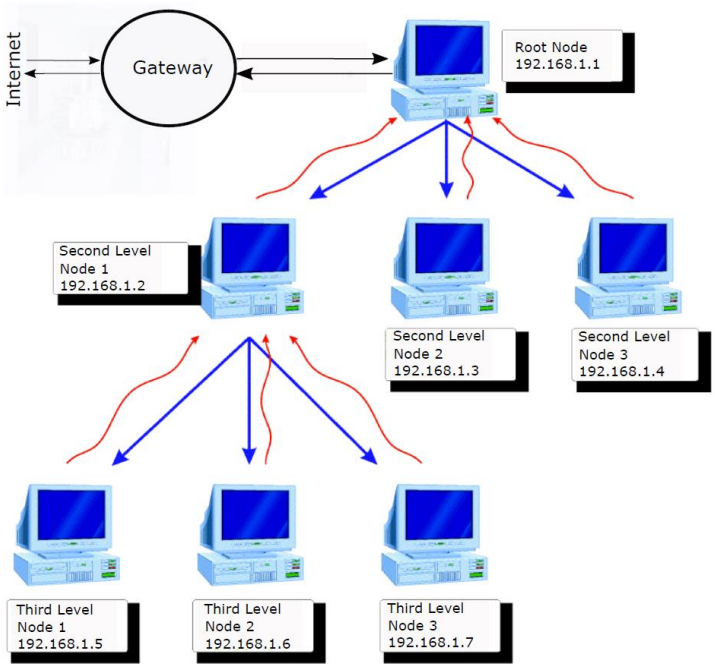


**Fig. 1. XML-VM** System Architecture. The gateway computer is the Root Node. Every other computer is a computational node and it is configured as root or leaf in a logical tree structure. Each node has a reference to its higher-level node and a to a local IP table.

In **Fig. 2** 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 is used in this work.

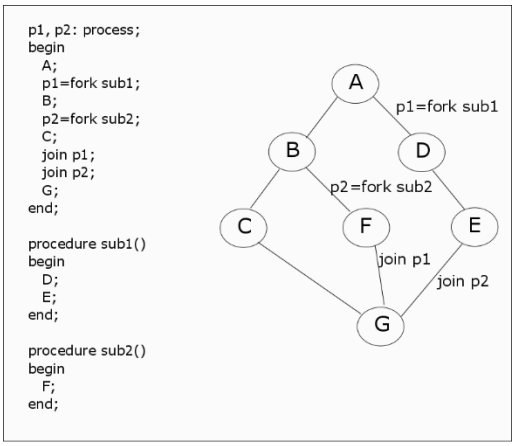


**Fig. 2.** Example of a Fork/Join concurrent processes with a predefined data type

## 5. The language XML-VM

In this Section, we 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 virtual disk; this means that, for example, if we want to make a simple sum of two numbers, the numbers must be stored in two data cells of the virtual disk. There are no variables, and every operation must be performed specifying the involved cells of memory; from this point of view, XML-VM is an assembler-like language.

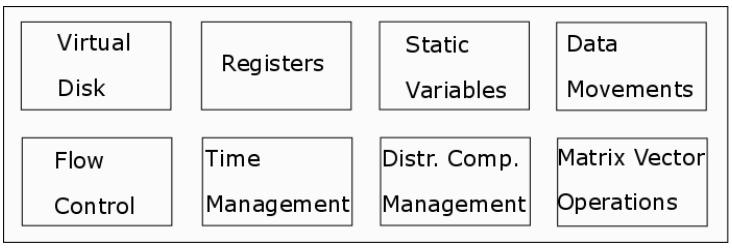


**Fig. 3**. The architecture of XML-VM

All the mathematical and logical instructions of the language take only place on registers. 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 programming loops. 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 XML-VM code writing. A simplified description of the instructions of XML-VM, grouped by type, is summarized in Appendix A.

### 5.1 Parsing and interpretation

The parser performs a complete analysis of the XML-VM document, expanding all the tags, attributes and 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 Piccolo XML Parser for Java [21], which is a light and fast SAX 1, SAX 2 and JAXP parser. Piccolo is very fast and it has been integrated in our system. The high performance reached is mainly due to the fact that the parser does not process the Data Type Definition (DTD). The interpreter of XML-VM language is written in Java for portability reasons; the interpreter executes the actions associated to the XML tags as they are analyzed by the parser.

Let us consider the architecture depicted in **Fig.1**. The following points are required:
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 resolve 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 pseudocode of the interpret is reported as follows.

```
public Object startExe (Object[]arg,xmlvm Machine)throws
Exception  {
     Initialize XML-VM stack, registry end virtual disk;
     Search the Tag Labels;
     Save the (name, position) couples in the
     LabelLis tarray;
     Verify that all the document is included in
     <XMLVM>…</XMLVM> Tags;
     Find the Tags <START/> or <STRUCT/>;
     Set 'i'=<START/>Tag position;
     try {
           for(i<tag.getChildrenCount();
           if (i-th Tag in ADD LOAD MOV STORE SUB MUL
           DIV
                           COMP LOCALCALL CALL FORK JOIN
                           CONV ELEV OPER e RANDOM)
```

```
            then
                  execute the Tag service routine;
            else{
                  if START and LABEL do nothing;
                  if STRUCT restore informations;
                  if JEQ, JNEQ, JGR, JNGR execute the
                  associated routines;
                  if RETURN execute the Return(tag)
                  routine;
                  if SHOW list registry content;
                  if QUIT exit;
              }
            }
      } catch(Exception e)
      Measure the time interval from initialization to
      last detected Tag;
      Send the time measure to the name resolution
      routine using XML-RPC;
      End routine;
}
```

In the following we report the pseudocode of the implementation of the Fork instructions.

```
public void Fork(xmlvm tag) throws Exception {
   Extract TO, IP, FILE, NAME and CLONE attributes;
   If (IP.charAt(0 == 'N'){
         Make an RPC call to the Name Resolution
         Module to get the IP corresponding to the
         identifier stored in IP;
         IP = true IP returned by the name resolution
         module;
         args = registry and virtual disk address
         indicated by CLONE;
         virtual disk = "*RESERVED*";
         ForkThread remoteCall = new ForkThread();
         RemoteCall.start();
   }
}
```

We now report the pseudocode of the implementation of the Join instruction.

```
public void Join(xmlvm tag) throws Exception {
   Extract TO and TOPOINTED attributes;
   if(TO.compareTo("") != 0)
         Verify that the TO cells are still not
         *RESERVED*;
```

```
        Otherwise, start a cycle to continuously
        monitor the cells;
  Else{
        Verify that TOPOINTED cells are still not
        *RESERVED*;
        Otherwise, start a cycle to continuously
        monitor the cells
  }
}
```

## 6. Experimental results

We report some experimental results about typical performance of the Grid computing system presented in this paper. In general, the efficiency of a distributed application is related to various factors: the network speed, the homogeneity of the machines which participate to the Grid, the degree of parallelism of the algorithm.

Our experimental study is performed on fifteen computers available in the laboratory intranet. Each computer, based on an Intel Core I5 running at **3.40 GHz**, **is** a XMLVM node. Three simple applications are written in XML-VM and executed on this network. The first application is the sum of two billions of integer numbers. The whole series of numbers is split in small sections whose summation is distributed among the nodes. The second application is the computation of the π number by solving this integral:

$$\int_0^1 \frac{4}{1+x^2} dx = \pi$$

where the [0-1] interval is divided in two billion sections distributed among the network. The third application a quicksort sorting algorithm executed on a sequence of two billion integers. The quicksort algorithm is implemented in the XML-VM distributed computing framework.

These simple problems allow us to study the behavior of XML-VM executing a large number of operations. Clearly, the computing time is dependent on the number of machines used in parallel for the elaboration of the algorithms.

The applications are executed on fifteen nodes and the execution time of each node is summed and divided by fifteen. To analyze the overhead of XML parsing, the parsing time of each node is averaged over the nodes too. The results are reported in Table 1.

**Table 1.** Medium Parsing and Execution Times and Parsing Overhead.

| Application | Average Parsing Time | Average Execution Time | Percentage |
|---|---|---|---|
| Sum | 384 ms | 5814.32 ms | 0.066% |
| Integral | 531 ms | 10662.81 ms | 0.05% |
| QuickSort | 518 ms | 7727.3 ms | 0.067% |

The first application is executed on a number of nodes from one to fifteen. What it is expected is that the execution time T(n) follows 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. 4**. The absolute time versus the number of machines follows a hyperbolic curve as expected.

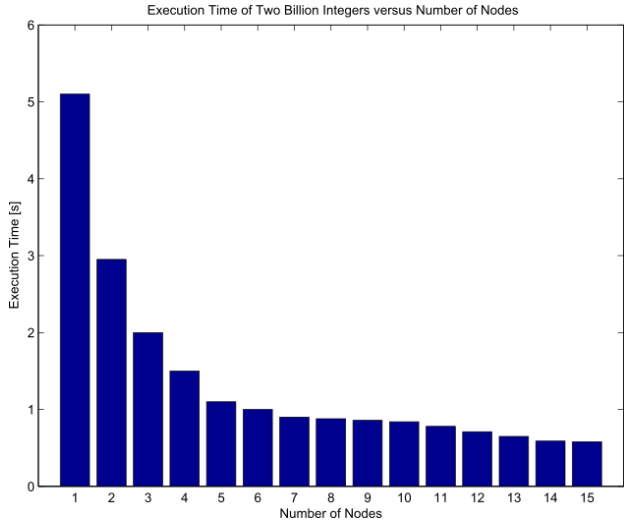


**Fig. 4** Execution time in seconds of the sum of the two billion integer versus the number of distributed nodes**.**

In **Fig. 5** we report the speedup of the execution time required by the sum operation performed on the distributed system compared with the execution time of the same operation on a single i5 computer.

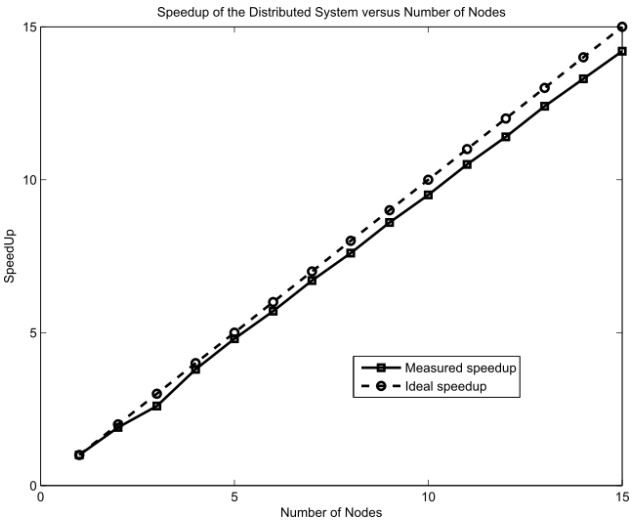


**Fig.5** Speedup of the execution time obtained with the XML-VM distributed system compared to the execution time on a single node.

The speedup approximately follows the theoretical behavior.

## 7. Conclusions and Final remarks

In this paper we dealt with the problem to design and develop an efficient architecture for realizing a computing system based on the Grid approach. We describe a Grid system based on XML. Many distributed applications are described. This work is based on the idea to use XML for describing algorithms. 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 by a fast XML parsing and by means of an interpreter we wrote in JAVA. The scalability of the system is realized efficiently with a name resolve daemon. Due to these reasons, the system performance compare very favorably with other solutions based on the distribution of Java code.

We develop a language, called XML-VM, for describing algorithms, as well an XML-VM interpreter to execute these algorithms. Each distributed machine runs locally the parser and the interpreter, whose execution is very efficient. The distribution of the methods is efficiently performed with XML-RPC and HTTPS protocols. Experimental results show that the practical behavior of the system is in agreement with the theoretical expectations.

Many problems are still open. The distribution of the workload, that is how to choose the distributed nodes, 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 problem could be mitigated by the development of an high-level language to XML-VM translator.

In conclusion, in view of the increasing importance of the Grid computing and of the possible future developments of this research, we believe that this paper can give a remarkable contribute in several theoretical and applicative fields.

## References


1. Paul Strong, “Enterprise Grid Computing”, Distributed Computing, Volume 3, issue 6, August 18, 2005
2. Sachith Gullapalli, “Atlas: an intelligent, performant framework for web-based grid computing”. ACM SIGSOFT FS,, pp.1154-1156, November 13-18, 2016
3. Shyna Sharma, Amit Chhabra, Sandeep Sharma, “Comparative analysis of scheduling algorithms for grid computing, International Conference on Advances in Computing, Communications and Informatics, Kochi, India, August 10-13, 2015
4. Leila Abidi, Christophe Cerin, Mohamed Jemni, ”Desktop Grid Computing at the Age of the Web”, Proceedings of 8th International Conference on Grid and Pervasive Computing, Seoul, Korea, May 9-11, 2013
5. R. Geetha, D. Ramyachitra, “Security Issues in Grid Computing”, International Conference on Research Trends in Computer Technologies ICRTCT – 2013, 33-37
6. GAO Quan-quan**, “**Research Development of "Great Internet Mersenne Primes Search Project”, Mathematics in Practice and Theory, CNKI Journal, 2010-5 D**.**
7. P.Anderson, J.Cobb, E. Korpela, M. Lebofsky, D. Werthimer, “SETI@home: an experiment in public-resource computing”, Communications of the ACM, Volume 45 Issue 11, November 2002, Pages 56-61

8. J. Aasi et.al, “Einstein@Home all-sky search for periodic gravitational waves in LIGO S5 data”, PHYSICAL REVIEW D 87, 042001[1-29] (2013)
9. M.A.Baker, M.Grove, A.Shafi, “Parallel and Distributed Computing with Java”, Symposym on Parallel and Distributed Computing, 2006.
10. Y.Sodha, H. Nakada, S.Natsuoka, “Implementation of a portable software DSM in Java”, Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande
11. M.Philippsen, M.Zenger , “JavaParty - Transparent Remote Objects in Java”, Concurrency Practice and Experience 9(11) · November 1997
12. Z. Yang , K. Duddy, “CORBA: a platform for distributed object computing”, Operating Systems Review, 1996
13. G. Kraetzschmar, H. Utz, S. Sablatnög, S. Enderle, G. Palm, “Miro — Middleware for Cooperative Robotics”, Lecture Notes on Computer Science, Vol.2377, 2001
14. D. C. Schmidt, “ACE: an Object-Oriented Framework for Developing Distributed Applications,” in Proceedings of the 6 th USENIX C++ Technical Conference, (Cambridge, Massachusetts), USENIX Association, April 1994.
15. M Conway, “Multiprocessing system design”, Proc. Of the AFIPS Fall Computer Conf., 1963
16. J.G.Dennis, E.C.Van Horn, “Programming semantics for multiprogramming computations”, Communications of ACM, March 1966
17. Ray Jinzhu Chen, “A Hybrid Solution of Fork/Join Synchronization in Parallel Queues”, IEEE Transactions on Parallel and Distributed Systems 12(8), August 2001
18. R.Nelson, A.N.Tantawi, “Approximate analysis of Fork/Join Synchronization in Paralle Queues”, IEEE Transactions on Computers, vol37, n.6, June 1988
19. Y.C.Liu, H.G.Peros, “A decomposition procedure for the analysis of a closed fork/join queuing system”, IEEE Transactions on Computers, vol40, n.3, march 1991
20. Doug Lea, “A Java Fork/Join Framework”, ACM Java Grande 2000 Conference, June 3-5 2000
21. Dennis M. Sosnoski, “XML documents on the run”, JavaWorld, APR 26, 2002

**Appendix A**: XML-VM tags

**XML-VM Mathematical tags**

```
<ADD target="r1" first="r2" second="r3"/>          → r1=r2+r3
<SUB target="r1" first="r2" second="r3"/>          → r1=r2-r3
<CONV register="r1" target="r2" to="type"/>        → convert type of r1 to r2
<DIV first="r1" second="r2" result="r3" rest="r4"/> → r1=r2*r3+r4
<ELEV target="r1" register="r2" exponent="r3"/>    → r1=r2^r3
<MUL target="r1" first="r2" second="r3"/>          → r1=r2*r3
```

**XML-VM Data movement tags**

```
<LOAD register="r1" index="m1"/>                   → r1=disk[m1]
<MOVE target="r1" source="r2"/>                    → r1=r2
<STORE to="m1" type="type">                        → disk[m1]=value
'value'</STORE>
```

**XML-VM Logical tags**

```
<CMP first="r1" second="r2"/>
<JEQ to="label"/>                                  → jump if equal
<JGR to="label"/>                                  → jump if greater
<JNEQ to="label"/>                                 → jump if not equal
<JNGR to="label"/>                                 → jump if not greater
```

**XML-VM Procedure call tags**

```
<CALL ip="IP:NPORT" file="Path/filename.xml" name="name" to="m5-m7">
   <PARAM>                                         → remote synch call
            r2
   </PARAM>
   <PARAM>
            m15
   </PARAM>
   …
</CALL>
<FORK id="N02" file="Path/file.xml" name="nome" to="r5-r7" clone="m5-m7"/>
<JOIN to="r5-r7"/>
<LOCALCALL name="name" to="m5-m7">                 → local call
   <PARAM>
            r2
   </PARAM>
   <PARAM>
            m15
   </PARAM>
   …
</LOCALCALL>
<RETURN from="m5-m7"/>
```

**XML-VM Miscellanea tags**

```
<LABEL name="name"/>      → labels the position
<QUIT/>                   → the end of a XML-VM document.
<START/>                  → begin of a "stand-alone" document
<STRUCT/>                 → begin of a "not stand-alone" document
```

**Appendix B:** Testing algorithm written in XML-VM

The algorithm that runs on the central node, written in XML-VM, can be described with the following pseudo-code, where the only lines written in XML-VM are that of the fork and join instructions:

```
R0=4000000;
R1=nr.increments;
R2=nr.remote.nodes;
For (n=0; n<R2;n++)                    /*for each remote node */
{
    R4=n * nr.increments;      /* 0, nr.inc., 2*nr.inc.,…*/
    R5=R4+nr.increments;
/* N00 means that the remote node is chosen randomly. */
/* R4 and R5 are 'passed' to the remote node */
    <fork "id=N00" file="url" name="sum" to="res loc" clone="R4 R5"/>
    <join to="results">
}
```

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

```
<?xml version='1.0'?>
<XMLVM>
<STRUCT/>
<LABEL name="Sum"/>
<STORE to="10" type="double">
      0
</STORE>
<LOAD register="0" index="10"/> <!-- In R[0] we put the result, set it to zero -->
<LOAD register="1" index="1"/> <!-- In R[1] we put the first parameter    -->
<LOAD register="2" index="2"/> <!-- In R[2] we put the second parameter -->
<ADD first="1" second="31"/>
<LABEL name="For"/>
<CMP first="1" second="2"/>   <!—if i > n_high then break -->
<JGR to="EndFor"/>
<ADD first="0" second="1"/>   <!-- result += i -->
<ADD first="1" second="31"/> <!-- ++i -->
<JNGR to="For"/>
<LABEL name="EndFor"/>
<RETURN from="r0"/>
<QUIT/>
</XMLVM>
```