Extension of Macrostep Debugging Methodology Towards Metacomputing Applications

Robert Lovas[1], Vaidy Sunderam[2]

[1]MTA SZTAKI Computer and Automation Research Institute,
Hungarian Academy of Sciences
P.O. Box 63, H-1518 Budapest, Hungary
rlovas@sztaki.hu

[2]Emory University, Dept. of Math & Computer Science
1784 N. Decatur Rd. #100
Atlanta, GA, 30322, USA
vss@mathcs.emory.edu

Abstract. This paper focuses on the non-deterministic behaviour and architecture dependencies of metacomputing applications from point of view of debugging. As a possible solution we applied and also extended the macrostep systematic debugging methodology for metacomputing applications. Our extended methodology is based on modified collective breakpoints and macrosteps furthermore, we introduce host-translation tables generated automatically for exhaustive testing. The prototype is developed under the Harness metacomputing framework for message box communication based applications. The main implementation issues as well as the architecture of our systematic debugger are also described as the further development of X-IDVS Harness-based metadebugger.

1 Introduction

Debugging of metacomputing applications can be much more exhausting task contrary to debugging of sequential or even parallel programs. This problem comes from the following features of metacomputing: (i) heterogeneity, (ii) dynamic behaviour of computational environment, (iii) large amount of computational resources, (iv) authorisation/authentication on different administration domains, (v) non-deterministic execution of metacomputing applications. During our previous debugging project [15] we have already given some efficient solutions for (i)-(iv) but the *systematic handling of non-determinism* was out of scope of that work. In this paper[1] we focused on the issues of the non-deterministic behaviour of metacomputing applications caused by the varying relative execution speeds of tasks as well as the architecture dependent failures. For instance, it seems a given metacomputing

[1] The work presented in this paper was supported in part by U.S. Department of Energy grant # DE-FG02-99ER25379 and National Research Grant (OTKA) registered under No. T-032226.

V.N. Alexandrov et al. (Eds.): ICCS 2001, LNCS 2073, pp. 263–272, 2001.

application always generates correct results on a particular architecture or a combination of architectures (where the programmers originally developed their application) *but* often fails on other architectures. Mostly, the reason for this behaviour is the varying relative speeds of tasks together with the hazardous and untested race conditions. Besides, these different timing conditions might be occurred more frequently in metacomputing environment than in case of dedicated clusters or traditional supercomputers because of the different implementation of the underlying operating systems/communications layers and the unpredictable network traffic, CPU loads or other dynamical changes. By metacomputing applications the above described phenomenon can be very crucial because we cannot ensure that our metacomputing application always runs on the same nodes with almost the same timing conditions.

The only way to prove the 'metacomputing-enabled' feature of an application is the usage of *systematic* testing methods in order to find the timing/architecture dependent failures in the implemented code. For this purpose we applied and also extended the macrostep systematic debugging methodology that has been introduced originally for message passing parallel programs developed by P-GRADE graphical programming environment [10]. Our prototype is under development in the Harness metacomputing framework [14] and we also applied the achievements of the earlier developed X-IDVS metadebugger tool [15].

This paper is organized as follows. In the next section we introduce briefly the Harness framework, the Java Platform Debugger Architecture (JPDA) and X-IDVS metadebugger tool as the basis of our prototype. Section 3 describes the fundamental principles of the extended macrostep debugging methodology and some implementation details. Finally, Section 4 summarizes our project and points out the most current related work.

2 Background

2.1 Harness Metacomputing Framework

Harness attempts to overcome the limited flexibility of traditional software systems by defining a simple but powerful architectural model based on the concept of a software backplane. The Harness model consists primarily of a kernel (see Figure 2) that is configured, according to user or application requirements, by attaching "plug-in" modules that provide various services. Some plug-ins are provided as part of the Harness system, while others might be developed by individual users for special situations, while yet other plug-ins might be obtained from third-party repositories.

By configuring a Harness distributed virtual machine using a suite of plug-ins appropriate to the particular hardware platform being used, the application being executed, and resource/time constraints, users are able to obtain functionality and performance that is well suited to their specific circumstances. Furthermore, since the Harness architecture is modular, plug-ins may be developed incrementally for emerging technologies such as faster networks or switches, new data compression

algorithms or visualization methods, or resource allocation schemes - and these may be incorporated into the Harness system without requiring a major re-engineering effort.

The fundamental abstraction in the Harness metacomputing framework is the *Distributed Virtual Machine* (DVM) (see Figure 1, Level 1). Any DVM is associated with a symbolic name that is unique in the Harness name space but has no physical entities connected to it. *Heterogeneous Computational Resources* may enroll into a DVM (see Figure 1, Level 2) at any time however, at this level the DVM is not ready yet to accept requests from users. To get ready to interact with users and applications the heterogeneous computational resources enrolled in a DVM need to load *'plug-ins'* (see Figure 1, Level 3). A plug-in is a software component implementing a specific *service*. By loading plug-ins a DVM can build a consistent service baseline (see Figure 1, Level 4). Users may reconfigure the DVM at any time (see Figure 1, Level 4) both in terms of computational resources enrolled by having them join or leave the DVM and in terms of services available by loading and unloading plug-ins.

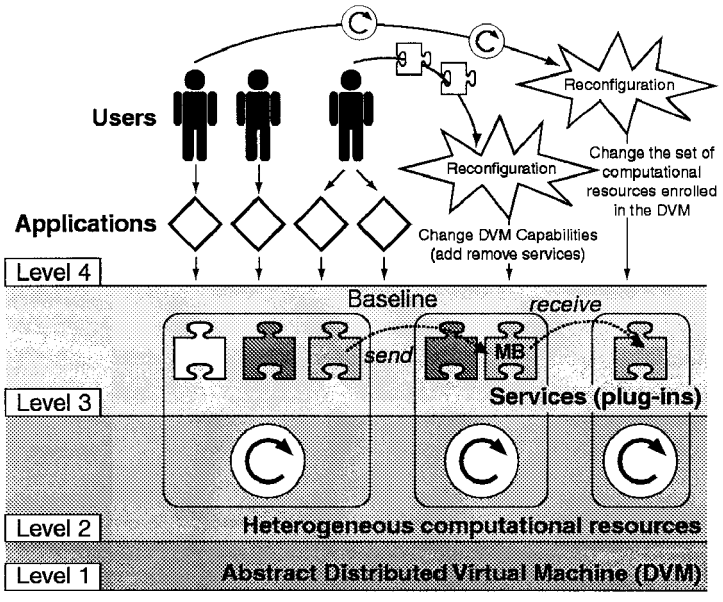

Fig. 1. Abstract Model of a Harness DVM with Message Box (MB) Service

The availability of services to heterogeneous computational resources derives from two different properties of the framework: the portability of plug-ins and the presence of multiple searchable plug-in repositories. Harness implements these properties mainly leveraging two different features of Java technology. These features are the capability to layer a homogeneous architecture such as the Java Virtual Machine (JVM) over a large set of heterogeneous computational resources, and the capability to customize the mechanism adopted to load and link new objects and libraries.

2.2 Java Platform Debug Architecture

Java Platform Debug Architecture (JPDA) is available for almost all widespread platforms as part of Java SDK 1.3. In outline, JPDA provides a high-level remote debugging interface for debuggers called Java Debug Interface (JDI). For the purpose of out-of-process debugging, JPDA gives the Java Virtual Machine Debug Interface (JVMDI) to the debuggee/target JVM. Between the JDI and the JVMDI, the Java Debug Wire Protocol (JDWP) is responsible for transporting both debug requests and debug events. Hence, JPDA can form a base of our X-IDVS debugger (see Section 2.3) by its remote debugging facilities (see Figure 2 between JVM1 and HMCPI).

2.3 Extendible Integrated Debugger & Visualization Service for Harness

In order to solve the emerging debugging issues in the field of metacomputing we already defined the fundamental principles of an extendible, programmable and integrated debugging & visualization tool [15]. The next target was to design and implement a prototype; X-IDVS (eXtendible Integrated Debugger & Visualization Service) applying the defined principles and relying on the Harness framework as well as the above described Java Platform Debugger Architecture.

In order to illustrate briefly the novelty of this work, the main features of the current X-IDVS prototype can be summarized as follows; X-IDVS was designed as a *real metacomputing application* itself hence, the debugger tool can adapt totally to the debugged application and also can take all advantages of the metacomputing environment, such as fault-tolerance, dynamic behaviour, support for heterogeneous computational environment and authorization. When a plug-in is loaded by the user's application anywhere in the metacomputer, X-IDVS can load and activate some system plug-ins on the target host for debugging/monitoring purposes (using the same authorization keys as the loaded plug-in). Moreover, for providing efficient debugging support for RMI-based plug-ins, X-IDVS offers some unique debugging capabilities for RMI communication. Firstly, during step-by-step execution X-IDVS is able to hide the differences between the traditional and remote method invocations from user's point of view. Basically, it means two automatic context switches during an RMI call (client to server/server to client side). On the other hand, X-IDVS combines some program visualization techniques with debugging methods. Hence, the user can get a big picture about the history of plug-ins with the help of an integrated semi-online visualization tool depicted the communication interacts among Harness plug-ins.

Another significant feature of the system is the *extendibility*. X-IDVS can invoke external sequential debuggers that might implement some other architecture dependent debugging facilities on a specified host/pool in the heterogeneous environment. In this way the user can choose the best tool in every phases of debugging procedure. Additional tightly integrated graphical tools are responsible for the navigation through the distributed/Java virtual machines and threads (equipped by filtering options for handling of scalability), management of breakpoint sets and establishment of new debug sessions.

Finally, X-IDVS is *programmable* with a simple macro language particularly for testing purposes. Thus, the programmer can test the startup of his application and can force the metacomputing application to run with vary timing conditions.

3 Systematic debugging in Harness

As it was described above, X-IDVS was designed originally for Harness applications built on RMI-based plug-ins. During an RMI-based interaction the invoked remote methods are executed in separated threads on the server side but the macrostep debugging methodology [7] cannot be applied in case of multithreaded applications (which might use shared objects). Thus, we had to take into consideration two options: (i) attempt to extend the macrostep debugging methodology with multithreaded/shared objects support or (ii) provide systematic debugging support for other types of Harness plug-ins, e.g. which are based on message passing paradigm. As the first stage of this project, we applied the macrostep debugging methodology on Harness plug-ins which can communicate with each other via message box. Based on these experiences and achievements we will try to solve the systematic debugging issues of multithreaded/RMI-based metacomputing applications as the next stage of this project.

In Harness the message box plug-in provides a generic send/receive/scatter/gather message passing service for Harness plug-ins via a simple interface:

- public void *send*(String senderID, String destination, Object message)
- public void *sendToAny*(String senderID, Object message)
- public H_Envelope *receive*(String myID, String senderID)
- public H_Envelope *receiveFromAny*(String myID)
- public H_Envelope *receiveAsync*(String myID, String senderID)
- public H_Envelope *receiveFromAnyAsync*(String myID)

In details, the *send* and *sendToAny* operations are always executed asynchronously but each type of *receive* operation can be either asynchronous or synchronous. As a first step we reduced these communication possibilities in order to get a similar message passing interface as in P-GRADE system where the macrostep debugging methodology has been implemented for the first time. Thus, we turned the asynchronous send operations to synchronous send and also removed both asynchronous receive operations.

The main ideas of the further developed macrostep debugging methodology can be summarized by the following concepts: (i) enhanced collective breakpoints, (ii) modified macrosteps, (iii) extended macrostep-by-macrostep execution mode, (iv) execution tree, (v) meta-breakpoints, (vi) execution tree. In the rest of this section we describe these concepts as well as some implementation issues.

In [7], a restriction was introduced on the global breakpoint sets and introduced a special version of them called **collective breakpoints**. When all the breakpoints of the global breakpoint set are placed on communication instructions, the global breakpoint set is called collective breakpoint. A formal definition of the collective breakpoints can be found in [7]. If there is at least one breakpoint for each alternative execution path of every process, the collective breakpoint is called strongly complete. In

practice, we were able to implement the strongly complete collective breakpoints by placing breakpoints on each method entries of message box interface. It means only a couple permanent breakpoints for each message box thus, we might achieve good performance that can be crucial in case of communication intensive metacomputing programs. Two problems were turned out during the design phase: (i) RMI communication between plug-ins and the message box, (ii) dynamically created message boxes. In details; the message box service was implemented as a plug-in, according to the Harness concept, and the sender/receiver plug-ins have to communicate with the message box plug-in *via RMI*. As it described in [15] JPDA has no debugging support for RMI but we have to find out which plug-in wants to send or receive a message (the *myID* and *senderID* string arguments can be defined without any restrictions by plug-ins). Hereby, we had to deal with issues of RMI debugging and to apply some RMI-related functions of X-IDVS in spite of our original plans. On the other hand, any Harness plug-in can create dynamically *new message boxes* therefore; our debugger tool must be also responsible for detecting when a new message box plug-in is loaded.

The set of executed code regions between two consecutive collective breakpoints is called a **macrostep**. Precise definition of macrostep is given in [7]. Provided that sequential program parts between communication instructions are already tested, a systematic debugging of a metacomputing program requires to debug the metacomputing program by pure macrosteps, i.e. instrumenting all the communication instructions by global breakpoints. A breakpoint of the collective breakpoint is called active if it was hit in a macrostep and its associated instruction has been completed. A breakpoint is called sleeping if it was hit in a macrostep but its associated instruction has not been completed (for example, receive instruction waiting for a message). Those breakpoints that were either active or sleeping in a macrostep are together called effective breakpoints.

After the definitions given above we can define the **macrostep-by-macrostep execution mode** of metacomputing programs. In each step either the user or the debugger runs the program until the collective breakpoint is hit. Under these conditions the metacomputing program will be executed by macrostep-by-macrostep. The boundaries of the macrosteps are defined by a series of effective global breakpoint sets. In such cases the user is interested only in checking the program state at the well-defined boundary conditions.

There is a clear analogy between the step-by-step execution mode of sequential programs realised by local breakpoints and the macrostep-by-macrostep execution mode of metacomputing programs. The macrostep-by-macrostep execution mode enables to check the progress of the metacomputing program at the points that are relevant from the point of view of parallel execution, i.e. at the message passing points. What we should ensure is that the macrostep-by-macrostep execution mode should work deterministically just like the step-by-step execution mode works in case of sequential programs. In order to ensure it, according to the original macrostep concept the debugger should store the history of collective breakpoints, the acceptance order of messages at receive instructions and the result of input operations. Additionally, in a metacomputing environment we should also store the events about the reconfiguration; when a new plug-in is loaded, unloaded or failed anywhere in heterogeneous computational environment, new host is grabbed/released or a new

message box is started by the user's application. Therefore, our debugger tool must be able to adapt to the dynamic behaviour of debugged application as well as its fault tolerance. As it was mentioned in Section 2.1, the enrolled computational resources as well as the DVM itself can be reconfigured. To handle the dynamic, reconfigurable and fault tolerant behaviour of DVM, our basic idea was the following. During the initialisation the Harness Monitor/Control Plug-In (HMCPI) places some so-called 'system breakpoints' in the Harness kernel (see Figure 2) in order to detect the changes/reconfiguration of DVM in advance. Then, HMCPI can report these events to Harness Systematic Debugger Tool (HSDT) that is responsible for storing these reconfiguration events in a trace file (see Figure 2). Basically, the fault tolerance of X-IDVS has been inherited from the Harness Framework itself.

At replay, the progress of tasks are controlled by the stored collective breakpoints and reconfiguration events and the program is automatically executed again macrostep-by-macrostep as in the execution phase. The debugger is also responsible for loading/unloading/killing the plugins, grabbing/releasing hosts and starting new message boxes during each macrostep (if it is needed). Obviously, during the replay phase it is not guaranteed that a host can be grabbed again for the distributed virtual machine or a given host is able to load the required plug-in (resource limitations, etc.). Our solution is a *host translation table* maintained by the debugger, in that each host enrolled in the original DVM can be associated to a substitute host (independently for each plug-in) where the appropriate plug-in actually run during the replay phase. The relative speed of the substitute host is unessential because the macrostep-by-macrostep execution can handle this issue. Only the architecture of the substitute host can be important if the current plug-in uses some architecture dependent features (e.g. via Java Native Interface). In this case, we have to check whether both architectures of reference and substitute hosts are the same ones.

In Harness the introduced host-translation table is used by the systematic debugger tool as well as the RMI communication core during the replay/control phases.

The execution path is a graph whose nodes represent macrosteps and the directed arcs connect the consecutive macrosteps. The **execution tree** is a generalization of the execution path, it contains all the possible execution paths of a metacomputing program assuming that the non-determinism of the current program is inherited from (wildcard) message passing communications. Nodes of the execution tree can be of four types: (i) Root node, (ii) Alternative nodes, (iii) Deterministic nodes.

The Root node represents the starting conditions of the metacomputing program. Alternative nodes indicate either a wildcard receive instructions which can choose a message non-deterministically from several processes or (as an extension of the original macrostep concept) a wildcard send instructions which can send a message non-deterministically to any process. Only alternative nodes can create new execution paths in the execution tree, deterministic nodes cannot create any new execution path.

Breakpoints can be placed at the nodes of the execution tree. Such breakpoints are called **meta-breakpoints**. The role of meta-breakpoints is analogous with the role of the breakpoints of sequential programs. A breakpoint in a sequential program means to run the program until the breakpoint is hit. Similarly, a meta-breakpoint at a node of the execution tree means to place the collective breakpoint belonging to that node and run the metacomputing application until the collective breakpoint is hit. Replay

guarantees that the collective breakpoint will be hit and the metacomputing program will be stopped at the requested node.

The task of systematic debugging or testing is to exhaustively traverse the complete execution tree with all the possible execution paths in it. Therefore, the execution tree represents a search space that should be completely explored by the debugging method. Accordingly, systematic testing and debugging of a metacomputing program require (i) generation of its execution tree (ii) exhaustive traverse of its execution tree. With the help of the extended macrostep-by-macrostep concept both of these issues can be solved and implemented in a very similar way as they have been implemented in DIWIDE [8][12]. Some minor changes are required by the wildcard send operations as well as the event tracing and replaying.

Often a Harness plug-in does not require a particular architecture for its execution. Despite of this we always have to inspect whether each plug-in has been implemented *architecture independently* if we want to get a real metacomputing application. For testing the architecture independency of plug-ins or whole applications the *systematically* generated host-translation tables are needed. It means that we have to test each architecture independent plug-in on each significantly different architecture (by exhaustive traverse of the execution tree).

We can test several plug-ins (from the aspect of architecture dependency) in one exhaustive traverse of an execution tree. In the best case we need only as much traversing of the execution tree as the number of significantly different architecture we have in the metacomputing environment. Our solution contains four steps: (1) the debugger looks for a host with a new and untested architecture for each architecture independent and not fully tested plug-in and the debugger also registers the found host into the new host-translation table, (2) if step 1 was not successful by any plug-in (after a timeout), we allowed the debugger to look for *any* host for the unsuccessfully mapped plug-ins (3) if there was at least one successfully mapped plug-in among the not fully tested plug-ins the debugger starts exhaustive traverse of execution tree, else go to Step 1, (4) if there is any not fully tested plug-in, go to Step 1.

We can decrease the time and resource requirements of the exhaustive tests by magnitude of orders in two ways. On one hand, we can take the advantage the large number of resources included in metacomputing environment by starting more (hundreds or thousands) test scenarios at the same time. On the other hand, we can try to reduce the complexity/size of metacomputing application as much as possible without losing the relevant parts of the application.

4 The architecture of systematic debugger

First of all, Harness kernels (see Figure 2) are launched with a special debug flag in order to enable the JVMDI interfaces and turn the JIT compiler off. As depicts in Figure 2, on each host enrolled in the distributed virtual machine two different types of plug-in can be found for debugging purposes: one HDPI and one HMCPI plug-in. Both of them are loaded by HSDT (with the same authorization keys as the user plug-ins) during the initialisation phase but they play different roles. In the first step of initialisation, HDPI gathers information about the enabled JVMDI interface and pass

the information to the HSDT. Therefore, in the second step the HSDT is able to load the HMCPI plug-in and attach it to the JVM of Harness kernel with the help of JDWP. Then, HMCPI is able to (i) place the system breakpoints in the Harness kernel in order to detect the reconfiguration of DVM, (ii) monitor the access the message boxes, (iii) control the execution of plug-ins.

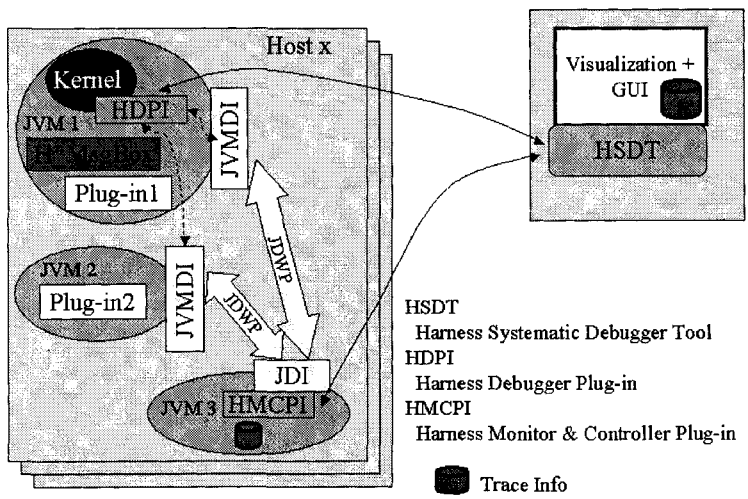

Fig. 2. Fundamental Architecture of Systematic Debugger In Harness

5 Conclusion and related work

As we studied the related work (such as DejaVu [1], DJVM [11], DIWIDE [8][12] integrated in P-GRADE [10], Macrostep-by-macrostep concept [7], TotalView, P2D2 [4]) we realized that there is a lack of an integrated graphical systematic debugger for metacomputing applications equipped by visualization techniques. Even the most relevant grid/metacomputing projects such as Globus [3], Condor [13] and Legion [6] do not give efficient solution to the emerging debugging issues. That was the main reason for the further development of the macrostep debugging methodology. The current prototype is partly implemented under the Harness Framework as an extension of X-IDVS metadebugger.

We also plan to investigate the efficiency issues of the described methodology as well as the feasible optimisation techniques that could be applied in order to reduce the complexity of exhaustive testing.

6 References

[1] Bowen Alpern, Jong-Deok Choi, Ton Ngo, Manu Sridharan, John Vlissides. A Perturbation-Free Replay Platform for Cross-Optimized Multithreaded Applications. IBM Research Report, RC 21864, 22 September 2000.

[2] Jong-Deok Choi and Harini Srinivasan. Deterministic Replay of Java Multithreaded Applications. ACM SIGMETRICS Symposium on Parallel and Distributed Tools, pages 48-59, August 1998.

[3] I. Foster and C. Kesselman. Globus: a Metacomputing Infrastructure Toolkit. International Journal of Supercomputing Application, May 1997.

[4] Robert Hood. The p2d2 project: building a portable distributed debugger. Proceedings of the SIGMETRICS symposium on Parallel and distributed tools, May 22 - 23, 1996, Philadelphia, PA USA

[5] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, B. Mancheck and V. Sunderam. PVM: Parallel Virtual Machine a User's Guide and Tutorial for Networked Parallel Computing, MIT Press, Cambridge, MA, 1994.

[6] A. Grimshaw, W. Wulf, J. French, A. Weaver and P. Reynolds. Legion: the next logical step toward a nationwide virtual computer, Technical Report CS-94-21, University of Virginia, 1994.

[7] P. Kacsuk. Systematic Macrostep Debugging of Message Passing Parallel Programs. Future Generation Computer Systems, Vol. 16, No. 6, pp. 609-624, 2000.

[8] P. Kacsuk, R. Lovas, J. Kovacs. Systematic Debugging of Parallel Programs in DIWIDE Based on Collective Breakpoints and Macrosteps. In: Proceedings. 5th International Euro-Par Conference, Toulouse, France, 1999. pp. 90-97.

[9] P. Kacsuk, G. Dózsa, T. Fadgyas, R. Lovas. GRADE: A Graphical Programming Environment for Multicomputers. Computer and Artificial Intelligence. 17 (5) :417-427. (1998)

[10] P. Kacsuk. Visual Message Passing Programming – the P-GRADE Concept. Scientific Programming Journal. 2000, Special Issue on SGI'2000

[11] Ravi Konuru, Harini Srinivasan, and Jong-Deok Choi. Deterministic Replay of Distributed Java Applications. 14th International Parallel & Distributed Processing Symphosium, pages 219-228, May 2000.

[12] J. Kovacs, P. Kacsuk. The DIWIDE Distributed Debugger on Windows NT and UNIX Platforms, Distributed and Parallel Systems, From Instruction Parallelism to Cluster Computing, Eds.: P. Kacsuk and G. Kotsis, Cluwer Academic Publishers, 2000.

[13] M. J. Litzkow, M. Livny and M. W. Mutka. Condor – A Hunter of Idle Workstations, Proc. of the 8th International Conference on Distributed Computer Systems, pp. 104-111, IEEE Press, June 1998.

[14] M. Migliardi, V. Sunderam, A.Geist, J. Dongarra. Dynamic Reconfiguration and Virtual Machine Management in the Harness Metacomputing System, Proc. of ISCOPE98, pp. 127-134, Santa Fe', New Mexico (USA), December 8-11, 1998.

[15] R. Lovas, V. Sunderam: Extendible Integrated Debugging and Visualization Service for Harness Metacomputing Framework, technical paper, available online at: http://www.sztaki.hu/~rlovas/projects/harness/docs/xidvs.pdf