Clustering Multiple and Cooperative Instances of Computational Intensive Software Tools

Dana Petcu[1,2], Marcin Paprzycki[3,4], and Maria Ganzha[5]

[1] Computer Science Department, Western University of Timişoara, Romania
[2] Institute e-Austria, Timişoara, Romania
[3] Computer Science Department, Oklahoma State University, USA
[4] SWPS, Warsaw, Poland
[5] Computer Science Dep., Gizycko Private Higher Educational Institute, Poland
petcu@info.uvt.ro, marcin@cs.okstate.edu, ganzha@pwsz.net

Abstract. In this note a general approach to designing distributed systems based on coupling existing software tools is presented and illustrated by two examples. Utilization of this approach to the development of intelligent ODE solver is also described.

1 Introduction

Developing from the scratch parallel systems to solve computationally intensive problems, while efficient, is in most cases rather difficult. Moreover, in initial stages of development, only few parallel algorithms within such a system are usually fully implemented and tested, making the resulting system too limited for practical uses. An alternative approach is to gradually add parallelism to an existing system consisting of a large number of fully tested sequential algorithms. In this case parallelism becomes an added value to an existing environment. Several software tools, based on different requirements and targeted for different hardware architectures, have been developed in this way. Such development followed the same general path leading from single-processor computers, through tightly-coupled parallel systems, to loosely-coupled distributed environments.

In this note we consider requirements imposed on system architecture that allow connecting several instances of a given software tool (possibly combined with single or multiple instances of other tools) in a cluster environment. Architecture discussed here is designed so that the system can be easily ported to the grid or to a web-based environment. The proposed architecture was implemented to couple instances of software tools from symbolic computing and from expert systems. The efficiency of implementation on a cluster is also reported.

2 Overview of the Proposed Architecture

Coupling several instances of software is an example of transforming software components into a conglomerate system. One of possible ways of accomplishing this transformation, is by implementing a *wrapper* that becomes the desired

V. Malyshkin (Ed.): PaCT 2005, LNCS 3606, pp. 452–456, 2005.

interface between components, translates external *interactions across native interfaces* and deals with *global syntax* of the system. Automatic wrapper generators for legacy codes that would be able to wrap the entire code or selected subroutines / modules, are still not available outside of the academia [8]. Here, we try to identify the requirements that have to be imposed on such a wrapper for the particular case of coupling software components within cluster environments.

We make the following assumptions about the software module that is to be interfaced with the system. (1) It has a user interactive interface, (2) it can be installed on several machines of a cluster or is accessible by each cluster machine via an NFS, (3) it has TCP/IP communication facilities, or its source code is available in a language having TCP/IP communication library, or it has I/O facilities, and (4) user knows how to split the problem into subproblems. Specifically, user wants to use the interface of an existing software module to launch several copies on nodes of a cluster and, in the next step, to send to them separate subproblems to be solved. Separately, we assume that cooperation between instances is possible. Finally, other users of the same cluster may want to use (within the same scenario) the same software module(s) within their applications. For example, each node of an 8-processor cluster has a copy of Maple running and various users may utilize groups of 1 through 8 kernels.

Designing cluster wrappers for software tools satisfying our assumptions can be done by combining two pieces of software (Figure 1). The first one is a set of simple commands, functions, procedures or methods, written in a language of that software tool; designed as user interface for controlling remote instances, sending and receiving information to and from them (parallel API – PAPI). The second one is a set of commands, functions, procedures or methods written in a language of the cluster middleware, catching the user commands and executing them (communication middleware, CMW). The PAPI is the cluster wrapper

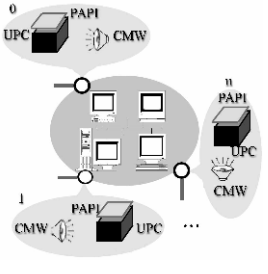

Fig. 1. Wrapping the user code (UPC) to clone it in a cluster environment: two main components, the PAPI depending on the UPC and the general CMW

component depending on the user provided software (denoted UPC), while the CMW is more general, it must be useable by several PAPIs. On a particular node of the cluster where an instance of the software tool is running, the PAPI set is loaded and any specific call to it leads to communication with the CMW

Table 1. The commands send by a PAPI to its twin CMW

Command	meaning
spawn n	CMW launches in the cluster environment n copies of the UPC
send d t "c"	CMW forwards to the dth copy of the UPC the text c to be interpreted by the UPC; the label t is used to match sends with receives;
receive s t	CMW and UPC wait until a message from the instance s is received; then forward it to the UPC which has requested the receive
probe s t	CMW tests if a message has arrived from s and responds 'true' or 'false'
kill s	Shutdown the UPC and the CMW from the node hosting the instance of s
exit	All remote UPCs and their twin CMWs are stopped
proc_no	CMW replace it with a no. representing the no. of current UPC copies
proc_id	CMW replace it with the identifier of the UPC copy

component activated on the same machine (by the classical TCP/IP or the I/O operations). A minimal language must be specified for such communication. For example, a minimal set of such messages can be as presented in Table 1. In this case, while user loads interactively the PAPI set within its UPC, the UPCs launched remotely also load the PAPI set, recognize their identifier in the system and constantly probe requests their twin CMW, i.e. if they do not work they expect to receive information from the other instances. Let us now illustrate most important details of implementation of the proposed architecture.

3 Implementation

We introduce two examples: (1) coupling several instances of Maple; (2) coupling several instances of Jess. Let us note that our main goal is to reduce the solution time when the two software tools are used to solve complex problems.

Coupling Several Instances of Maple. There exist a large number of efforts to extend Maple to parallel and distributed environments and a comprehensive review of the state-of-the-art can be found in [9]. Within last 5 years we have implemented two variants of cluster (and grid) oriented Maple: PVMaple and Maple2g.

In PVMaple [4], Maple was wrapped into an external software that manages execution of tasks. The CMW, a special binary, written in C and using PVM, is responsible for the message exchanges between Maple processes, coordinates interaction between Maple kernels via PVM daemons, and schedules tasks among nodes. The PAPI, a Maple library, consists of a set of parallel programming commands available within Maple itself and supports connections with the CMW (Table 2). Communication between Maple and CMW is achieved via text files.

The more recent wrapper – Maple2g [6] - also consists of two parts, the PAPI, the computer algebra dependent one – m2g, a Maple library of functions allowing Maple users to interact with the grid or cluster middleware – and the CMW, the grid-dependent one – MGProxy, a package of Java classes acting as

Table 2. PAPI to CMW communication in PVMaple, Maple2g, and Parallel Jess

PAPI	pvm lib for PVMaple	m2g library for Maple2g	ParJess lib
spawn n	spawn(IP,proc_no)	m2g_maple(n)	(kernels n)
send d t "c"	t:=send(d,c)	m2g_send(d, t, c)	(send d t s)
receive s t	receive(t, s)	m2g_recv(s, t)	(recv s t)
probe s t	-	m2g_prob(s, t)	(prob n t)
kill s	-	-	(kill s)
exit	exit	m2g_exit()	(exit)
proc_no	tasks	m2g_size	-
proc_id	TaskId	m2g_rank	?*p*
-	ProcId, MachId, setttime(), time(), version()	m2g_connect(), m2g_getservice(c, l) m2g_jobsubmit(t, c), m2g_jobstop(t) m2g_status(t), m2g_MGProxy_start(), m2g_results(t), m2g_MGProxy_end()	(connection)

an interface between m2g and the grid or cluster environment. MpiJava was selected as the message-passing interface for the CMW, due to ist compatibility with the Globus Toolkit. Communication between Maple and the MGProxy is achieved using socket library available in Maple. The most important features of Maple2g are summarized in Table 2. Maple2g was tested on a cluster of 9 PCs connected with a fast Myrinet switch (2Gbs) on which Maple7 was installed. To indicate an order of the efficiency, for 2 integers of 10 millions multiplied with Karatsuba algorithm (the implicit procedure in Maple7): 88% for 3 processors, 71% for 9 processors. Subsequent examples in the grid case can be found in [6]. All experiments indicate reasonable scalability of both PVMMaple and Maple2g (scalability depends primarily on network throughput).

Coupling Several Instances of Jess. Standard benchmarks [2], show that current rule-based systems, running on modern hardware, may need hours to reach an answer when the number of rules is of order of thousand. Therefore, parallel approaches are needed for real applications and first parallel implementations were already available in early 1990s [1]. There are several approaches to parallelization [10]: (a) parallel matching leads to a limited speedup caused by the sequential execution of rules; (b) multiple rule firing approach parallelizes the match phase and the act phase by firing multiple rules in parallel, but involves extra cost due to synchronization; (c) special techniques like compatible rules or analysis of data dependency graphs, can improve efficiency of parallelization; (d) task-level parallelism, used here, based on the decomposition of the problem into a hierarchy of tasks is expected to lead to best results.

Jess, a rule-based programming environment written in Java was chosen because of its active development and support, and because there is no parallel version of Jess. The proposed architecture, recently reported in [7], also follows the wrapper-based design presented in Section 2.

The CMW in Parallel Jess consists of two parts: the Connector and the Messenger. The Connector is written in Java and uses standard ServerSockets

methods of TCP/IP communication. Jess instance acts as a client and contacts (via socket) its Connector, the server. Each Messenger is associated with one local Connector and its purpose is (1) to execute commands received by the Connector, and (2) to communicate with Messengers associated with other instances of Jess. Messenger is written in Java and JPVM. Set of new commands added to Jess (the PAPI) is presented in Table 2.

In order to test Parallel Jess efficiency, we applied it to the Miss Manner problem [2] on the same cluster computer, obtaining an efficiency of 95% for 2 processors, and 45% for 8 processors. Several other examples can be found in [7] and all of then indicate reasonable efficiency of Parallel Jess.

Future Research Direction. The above described components will be used to develop a distributed cluster-based intelligent ODE solving environment. Here, the problem will be described in a user friendly environment of the latest version of the ODE numerical expert, EpODE [3]. The problem properties (stiffness, decomposability, etc) will be then analyzed using a Maple kernel residing in the cluster environment (e.g. eigenvalues of the linear part), or using EpODE facilities (e.g. Jacobian matrix). Decisions which analysis methods to apply will be made by a rule-based algorithm rewritten in Parallel Jess. Furthermore, if the problem is large, a Maple2g multiprocessor approach will be used (see also [5]).

References

1. Amaral J.N.: A Parallel architecture for serializable production systems, Ph.D. Thesis, University of Texas, Austin, 1994.
2. OPS5 Benchmark Suite, available at http://www.pst.com/benchcr2.htm, 2003.
3. Petcu D., Dragan M.: Designing an ODE solving environment, LNCSE 10, Procs. SciTools, eds. H.P. Langtangen et al(2000), 319-338.
4. Petcu D.: PVMaple – A distributed approach to cooperative work of Maple processes. LNCS 1908, eds. J.Dongarra et al., (2000), 216–224
5. Petcu D.: Numerical Solution of ODEs with Distributed Maple, LNCS 1988, Procs. NAA, eds. Lubin Vulkov et al, 666–674, 2001.
6. Petcu D., Dubu D., Paprzycki M.: Extending Maple to the Grid: Design and implementation, in Procs. ISPDC'04, J.Morrison et al. eds., IEEE CS Press, 209-216.
7. Petcu D., Parallel Jess, Proceedings for the ISPDC 2005 Conference, to appear.
8. Solomon A., Struble C.A.: JavaMath – an API for internet accessible mathematical services, Procs. 5th Asian Symposium on Computer Mathematics, (2001).
9. Schreiner W., Mittermaier C., Bosa K.: Distributed Maple – parallel computer algebra in networked environments, J. Symb. Comp. 35:3, (2003), 305–347.
10. S. Wu, D. Miranker, J. Browne: Towards semantic-based exploration of parallelism in production systems, TR-94-23, 1994.