Tools for Parallel/Distributed Mathematical Computation

Paul S. Wang[*]
pwang@mcs.kent.edu
http://icm.mcs.kent.edu/ pwang
Institute for Computational Mathematics
Kent State University
Kent, Ohio 44242-0001

January 30, 1997

Abstract

A set of software tools for connecting heterogeneous tasks on networked computers as well as MIMD parallel machines is described. The purpose is to provide a convenient and practical approach to connect mathematical/scientific computing tasks, to build complex systems by integrating existing ones, and to perform parallel/distributed processing in symbolic and algebraic computation (SAC). The tools work with the widely available PVM (Parallel Virtual Machine). Efficient exchange of mathematical data is achieved through MP, an efficient mathematical data exchange protocol. The tools work well together but can also be used independently:

- Program compilation and distribution tools for PVM
- Common Lisp and MAXIMA Interface library to PVM
- Saclib to PVM interface library
- Generic run-time task scheduling library for PVM
- C-coded MP library for efficient mathematical and scientific data transfer
- MP-related utilities for Email- and Web-based SAC applications

The tools are available by public FTP.

[*]Work reported herein has been supported in part by the National Science Foundation under Grant CCR-9503650

1 Introduction

As parallel computers and high-speed networks become more widespread, it is increasingly important to develop new computing systems by connecting and combining existing programs to run in a parallel or distributed fashion. The individual program components can be *heterogeneous* — written in different languages, run under distinct operating systems, or designed for special architectures. The arrangement allows easy reuse of existing codes and, perhaps more importantly, permits separate parts of a system to be implemented on different platforms and maintained at sites with the right expertise. For example, the user interface part may run on a graphics workstation while compute-intensive parts may execute on parallel machines and supercomputers. The strategy encourages building systems by constructing reusable components and has wide-ranging applications. In the area of symbolic and numeric computing, people are interested in constructing *problem solving environments* by connecting symbolic, numeric, visualization, document processing, and other appropriate software components. The approach can also provide medium to very coarse grain parallelism.

Researchers have been working on ways to connect heterogeneous tasks. The work on *Extended Data Representation* (XDR) [25] and *Remote Procedure Call* (RPC) [4], made early contributions. The Polylith system [19] investigated the management of tool interfaces, DSC [9], and STAR/MPI [8] experimented with distributed symbolic computing. Other systems such as *Toolbus*, *Parallel Virtual Machine* (PVM) [10], *Message Passing Interface* (MPI) [13], and *Common Object Request Broker Architecture* (CORBA) [18], [20] have made important advances in distributed/parallel computing. The subject remains an active R&D area and there are many interesting and important problems yet to be solved.

Research reported here focuses on a very practical problem: *easy connection of symbolic computing tools for parallel/distributed computing*. The goal is to make it easy for almost anyone to connect SAC engines together with other components for prototyping and experimentation. The approach taken is straightforward: build a set of tools to allow easy connection of SAC and other tasks. PVM is adopted to connect tasks because it is effective, widely available, and in the public domain. A similar approach can be taken with MPI [13].

Once compute engines are connected and able to send and receive messages, a common protocol must be used to exchange information. The MP is a protocol designed and implemented for efficient transfer of mathematical data among cooperating tasks. Work on MP addresses many central issues of efficient transmission of mathematical data. The research, together with investigations on mathematical protocols by others, should complement, aid, provide experience, and otherwise advance the *OpenMath* [1] effort to achieve a widely adopted, standard mathematical data transmission protocol.

A set of PVM enhancement tools are described first. Then, CL-PVM, a Common Lisp interface to PVM, is presented. CL-PVM allows easy interfacing of Lisp-based programs and enables CL-based SAC systems to run as PVM tasks. A Saclib to PVM interface makes C-coded symbolic systems that use Saclib easy to connect. A run-time task scheduling library provides a generic bag-of-jobs facility. The library and utilities enabling Email- and Web-based applications of MP are described.

The tools are available by anonymous FTP from

`ftp.mcs.kent.edu`.

Current status of these tools can be accessed easily on the `SymbolicNet` web site:

`http://SymbolicNet.mcs.kent.edu/`

2 PVM-ET

Parallel Virtual Machine (PVM) [10] is a convenient and widely available computing facility for connecting parallel tasks on parallel or networked computers. A PVM user can create a *personal parallel virtual machine* (*pvm*) (Figure 1) by designating, in a *hostfile*, hosts on a LAN as processing elements. The same PVM system also works on actual parallel computers with a message-passing (Cray T3D for example) or shared-memory (SGI for example) architecture. Thus, parallel programs developed on a pvm can be easily ported to run on actual parallel computers for even better performance. The PVM consists of a run-time system and library functions to support application programming in `f77` and `C`. PVM is available by public FTP (`ftp://ftp.netlib.org`). A user simply lists host names

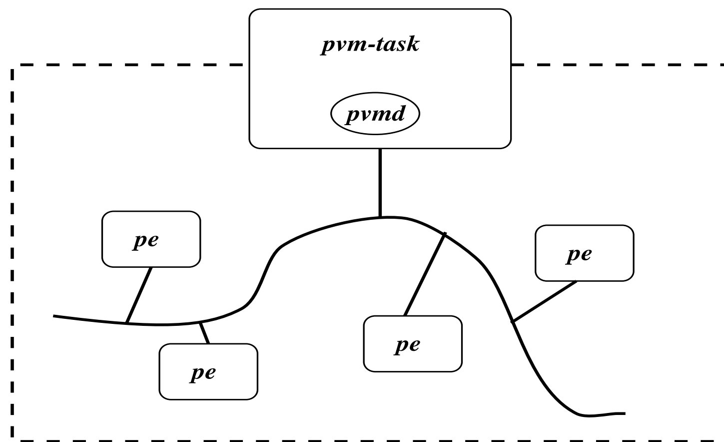

Figure 1: Parallel Virtual Machine

in a per-user configuration file to define a pvm. Starting your pvm causes a PVM daemon to run on each host and the pvm persists across login sessions. Distributed tasks can be created, deleted and otherwise controlled in an application program as well as interactively from the *PVM console* program.

PVM-ET [22] is a set of enhancement tools developed to aid the use of a pvm. The enhancements make much easier setting up new PVM users, compiling programs, creating and distributing executables, and performing other chores on a pvm. Development of PVM-ET began on the HEAT [21] workstation cluster at Sandia National Labs (Livermore, California) in 1994 during a sabbatical visit. Improvements, further testing and debugging were done subsequently at Kent/ICM.

PVM-ET Summary

PVM-ET is a set of user-level programs to aid PVM related work and experimentation and should be of general interest to PVM users. PVM-ET functionalities are implemented as individual UNIX commands. The following points apply to all PVM-ET commands.

- Each command has an on-line manual page.
- A command given with the wrong number of arguments displays its usage.
- Command options can be given in any order but must be given before other arguments and options.
- If an optional hostfile argument is not given, then the file `hostfile` in the current directory or in the `$HOME/pvm3`, in that order, is used.
- Default values are set by the tools and can be customized easily.

Each command is summarized briefly here. More information can be found in [22].

1. **pvminit** – sets up the PVM environment for a new user automatically.
2. **pvmcc1** – compiles a C-coded PVM application on the local host. It generates an executable file in a standard file location

   `$HOME/pvm3/bin/$PVM_ARCH`

   and distributes this executable to the local disk directory (`pvm3/local_disk_bin`) for the local host. Optionally, it also distributes the executable to all hosts with the same architecture as the local host. This command is useful when testing and debugging a new PVM application.
3. **pvmf771** – is the Fortran 77 counterpart of **pvmcc1**. The default compiler used is **f77**.
4. **pvmcc** – compiles a set of C source programs for your entire virtual machine by generating an executable on each host. This is done by compiling once on each different architecture and distributing to all specified hosts (by calling **pvmcc1**).
5. **pvmf77** – is the Fortran 77 counterpart of **pvmcc**.
6. **pvm_distrib** –

   distributes executables to the local disks for the hosts indicated in the given hostfile. The hostfile may contain hosts of many different architecture types, but **pvm_distrib** will distribute executables only to hosts of the architecture specified.

7. **pvmexec** –

   executes a given command string on each host specified in the hostfile. The actions are carried out in parallel and all output, including error output, is sent to the file `/tmp/$HOST.log` on each host.

8. **pvmclean** –

   helps a user clean up the local disk directories

   `$home/pvm3/local_disk_bin/`

   for specified hosts or the entire pvm.

PVM-ET introduced the use of the symbolic link

`$HOME/pvm3/local_disk_bin`

which is a key feature that deserves some discussion.

Usually, PVM is used in an environment where a group of UNIX workstations and other high speed servers are connected by a fast LAN within a single department. Each PVM user requires **rsh** privilege on each component of the parallel virtual machine. Thus, the computers are usually under the administrative control of one closely knit group. Often, the users will keep unique home directories on centralized disk servers to avoid file duplication and to simplify system maintenance and file backup. Furthermore, executable codes common to all, or a subset, of the machines are also centralized for the same reasons. PVM-ET take

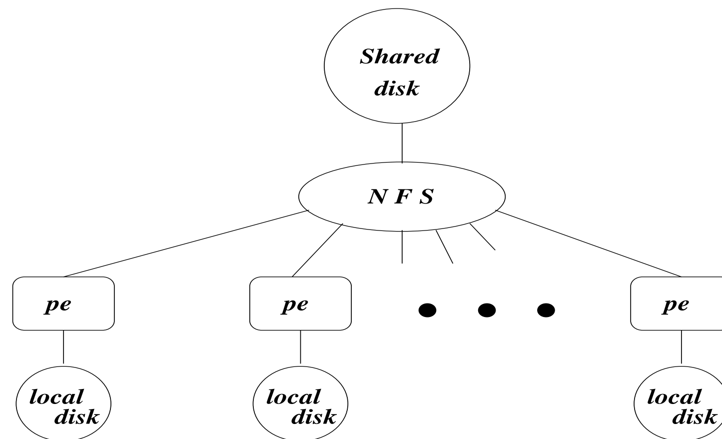

Figure 2: Disk Access Bottleneck

these conditions into account. All PVM sources, libraries, shell scripts, etc. are kept on shared disks. But, the PVM executables for each individual computer can be kept on local disks to minimize time required to spawn PVM processes and to avoid service congestion on

shared disks (Figure 2). Because loading an executable from a locally mounted disk can be many times faster than fetching it via NFS (especially during multiple parallel fetchings), this reduces the `pvm_spawn` overhead significantly. For certain applications where spawning of medium grain activities are performed repeatedly this can be very important.

The PVM-ET tools are fine, but other facilities are needed to connect SAC tasks.

3 A Common Lisp Interface to PVM

The standard PVM distribution contains libraries for C and F77 applications. The *CL-PVM package* [14], [15] enables any CL-based programs to take part in PVM applications(Fig. 3). A wide variety of useful Lisp programs exists including SAC systems, expert systems, artificial intelligence systems, knowledge-based systems, and many more.

With CL-PVM, the PVM library routines can also be invoked interactively from the Lisp top level or from Lisp programs.

CL-PVM contains a set of Common Lisp functions that interfaces Common Lisp (KCL, AKCL, or GCL) to the C-based library With CL-PVM, the PVM library routines can be invoked interactively of PVM [7]. Generally, there is one CL interface function to each PVM C library function. The CL function calls its C-based counterpart and relays data to and from the C function. For example, the CL function `pvm-spawn` calls the C function `pvm_spawn`.

Here is a sample call in Lisp

```
(pvm-spawn "hello_other" "" 0
   "condor.mcs.kent.edu" 1 'pids)
```

which spawns a pvm task `hello_other` on the host `tiger`.

CL-PVM is complete with on-line manual pages and examples. CL-PVM is available by public FTP from SymbolicNet web site at Kent/ICM. The package is also distributed together with GCL (Gnu Common Lisp) at the FTP site `ftp.ma.utexas.edu` (in `/pub/gcl/`) at the University of Texas at Austin. Please refer to [15] for more information on the design and implementation of the Lisp interface to PVM.

4 Lisp Processes as PVM Tasks

Being interactive, Lisp provides an easy way to test CL-PVM and to develop PVM applications. But, PVM applications usually require non-interactive tasks. A Lisp process can be turned into a PVM task by writing a simple shell script. Consider the `hello.lsp` program with a top-level called `(hello ...)`. The following *csh* script can be used:

```
#!/bin/csh
set logfile = /tmp/${USER}LOG.$$
set gcl_w_pvm = /usr/local/gcl/gcl+pvm
```

```
set gcl_dir = /usr/local/gcl/
echo $0 $argv INVOKED >! $logfile

set cmd = ( hello )
foreach a ( $argv )
    set cmd = ( $cmd \"$a\" )
end

$gcl_w_pvm $gcl_dir >>&! $logfile <<EOF
(if (null (load "$HOME/pvm3/local_disk_bin/hello.o"
        :if-does-not-exist nil))
        (progn (princ "loading hello.o failed.")
               (bye) )
)
( $cmd )
( bye )
EOF
echo FINISHED >>! $logfile
```

The executable `$gcl_w_pvm` is a version of GCL with CL-PVM codes already loaded.

The `hello.o` file in the above script refers to the compiled code produced from `hello.lsp`, part of the examples package distributed together with CL-PVM. This particular script takes

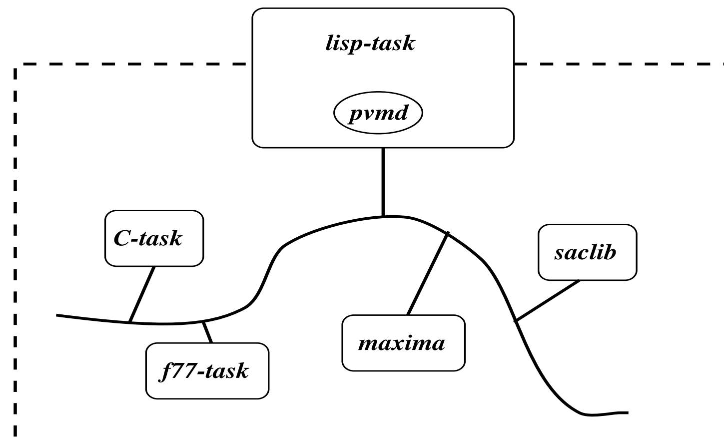

Figure 3: Tool Integration via PVM

one command-line argument indicating the host where to spawn the `hello_other` task. The command-line argument processing shown works in general for any number of arguments. Using a shell script to invoke a Lisp process has another advantage: the program is architecture independent.

Executable shell scripts like the one shown have a common structure and can be automatically generated given appropriate data. The **pvm_cltask** tool included in the package

does just that.

5 CL-Related Tools

Additional tools are available in CL-PVM to help manage Lisp tasks. These are:

- **pvmlc1** — a command to compile CL source files into `.o` files for one architecture type.
- **pvmlc** — a command to compile and distribute CL-coded files for PVM tasks on all specified hosts used in the form:

  pvmlc `[ -U` *compiler* `] [ -H` *hostfile* `]` *file*`.lsp ...`

- **pvm_cltask** — a command used to generate a PVM executable task in the form of a *csh* script when given a list of Lisp object files. It is used in the form

  pvm_cltask `[ -I` *lisp* `] [ -N` *executable* `]` *file*`.o ...`

  The script automatically invokes a user-specified Lisp function and passes to it any command line arguments. The generated script can be edited and later distributed by **pvm_distrib**.

6 MAXIMA under PVM

With CL-PVM, it is rather simple to run MAXIMA as a PVM task since the symbolic computation system is built on top of GCL/AKCL. You can load the CL-PVM interface into MAXIMA and produce, say, a **maxima+pvm** command to run MAXIMA-based PVM tasks.

The CL-PVM package also includes

- **pvmmc1** — a command to compile `.lsp` files written for MAXIMA to `.o` files on one architecture type.
- **pvmmc** — a command to compile `.lsp` files written for MAXIMA to run on a *pvm*.
- **pvm_maximatask** — a command to generate maxima-based PVM tasks.

Examples showing usages of these commands are included in the CL-PVM package.

7 Running SACLIB under PVM

SACLIB [6] is a public-domain SAC library written in C. It supports integer, modular, polynomial, and other important SAC computations. SACLIB is useful as a base for developing applications and implementing advanced algorithms in symbolic computing. Thus, being able to also connect SACLIB-based tasks is important.

SACLIB represents characters and single-precision numbers as atoms and multiprecision integers and polynomials as linked lists. Automatic garbage collection is provided. To use a SACLIB-based task under PVM, all that is needed is a way to send and receive SACLIB data structures through PVM. Four library functions have been constructed for this purpose.

- `pvm_pklist` – packs a given SACLIB list into a PVM outgoing buffer.
- `pvm_upklist` – retrieves data packed by `pvm_pklist` and recreates a SACLIB list.
- `send_poly` – sends a SACLIB polynomial to a designated PVM task.
- `recv_poly` – receives a SACLIB polynomial sent by another PVM task.
- `mcast_poly` – broadcasts a SACLIB polynomial to all other PVM tasks.

The routines are efficient and avoid any reparsing of the structures.

Applying these tools, Mr. Ajwa, a graduate student at Kent, is investigating parallel Gröbner Basis algorithms [2] using SACLIB and PVM. The work also involves developing and applying a *bag-of-jobs* library to control and synchronize SACLIB processes as PVM tasks.

8 Generic Run-time Job Control

The bag-of-jobs library (BJ), implemented in C, is designed as a generic facility to manage parallel tasks under PVM. It is implemented as a C library. A CL interface makes it usable by CL-based tasks as well. BJ provides an easy-to-use master-slave paradigm for scheduling and synchronization. BJ makes it easy for a master PVM task to spawn slaves, dispatch jobs to each slave, and synchronize them, all in an application defined manner. Each bag has its own job queue and set of slave tasks. A master can control multiple bags. A slave, in turn, can also be a master. The job requesting, dispatching, and result gathering are done asynchronously.

- `bag_Open` – called by master task to request a new bag. The master may designate hosts for slave tasks.
- `Job_FirstRequest` – called by a slave task to obtain the very first job assignment.
- `Job_NextRequest` – used by a slave task to send back results and obtain the next job assignment.

- bag_JobComplete – called by a master task to receive results sent back by slave tasks in a specified bag.
- bag_JobDispatch – used by a master task to assign a job to a ready slave in a particular bag.
- bag_Close – to close down a bag of slave tasks.

A master can also add and remove slave tasks in a bag. These are complemented by status reporting, administrative, and error handling functions. The representation of jobs and answers as well as their transmission via PVM are arbitrary and can be defined by each application. This library comes with good documentation and examples. The CL interface part of BJ is still under development.

9 Efficient Exchange of Mathematical Data

Described so far is a cohesive set of tools to integrate tasks for distributed/parallel computing. With these tools, it becomes easy to connect and control SAC compute engines in a heterogeneous environment. An important aspect yet to be considered is the efficient exchange of scientific data.

The author has been collaborating with N. Kajler and S. Gray on the Multi Protocol (MP) [11], [12]. Among the goals of MP are to design and develop a protocol for efficient communication of mathematical data among scientific computing systems (Fig. 4) and to explore how such a protocol might be used in practice. Initial deliberations on Multi started in the late 1980's at Kent and involved Peter Hintenaus, Michael Rothstein, Paul Wang, and several graduate students. Significant progress on the design and implementation of MP began in 1990 as a result of joint work with N. Kajler and S. Gray. The initial design of the Multi Protocol was done in 1992. The first achievement of this research was to provide an unambiguous protocol for the efficient exchange of mathematical expressions between scientifically-oriented packages. The initial design underwent some modifications and additions. Initial release of the C-coded MP package was in 1995. The revised MP-1.1.1. was released for Beta testing in October 1996 [12].

MP Features

The MP design strives for efficiency, flexibility, and extensibility in scientific data exchange. Here are some highlights.

- Data exchange via binary-encoded parse trees – MP defines a set of basic types: text-based types (identifier, operator, string), binary types (single and arbitrary precision integer and real), a raw type for transmitting uninterpreted data, plus the meta and MP-operator types for protocol management purposes. More complicated data such

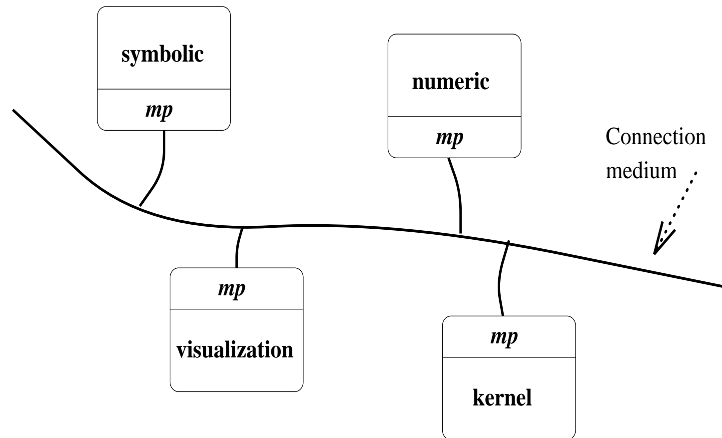

Figure 4: Data Exchange with MP

as mathematical expressions, data structures, function calls, procedures, etc. are represented by *MP annotated trees* (MaTs). An MaT is basically a binary-encoded parse tree whose nodes may have attached *annotations*. MaT is simple to implement, efficient to use, yet powerful enough for most, if not all, practical needs.

- Annotations – Each MaT node may be *annotated* with supplementary information. Annotations may provide information about how a data item is to be interpreted (e.g., units of measurement - miles, centimeters, Jules), the original source of the data, and any application specific attributes. MP-defined annotations, there are 10 currently, are understood by any system using MP. Application-defined annotations can be established to fill problem-specific needs.

- Optimizations – In addition to efficient binary encoding of basic data types, MP uses several techniques to reduce the amount of data transmitted. One is common subexpression elimination supported through the `label` and `reference` annotations. When a subexpression occurs more than once in an expression, the first occurrence of the subexpression may appear in the annotated tree with a label attached to it and all other occurrences simply contain a reference to the labeled subexpression. Also, significant optimizations can be made on blocks of data characterized by a common *format* (for example, a vector of real numbers or an array of records whose fields are of different types). This is done by preceding the data blocks with a format description called the *MP prototype*.

- Dictionaries – A scientific data exchange protocol must deal with two basic issues, the syntax and the semantics of transmitted data. The MaT deals with syntax. A separate mechanism must be used to handle semantics so communicating parties can obtain the same interpretation on the data being passed. The MP approach for semantics is to use *dictionaries*. A dictionary is an offline, human-readable document containing a list of items together with their meaning. A definition may be formal or informal, but

must be sufficiently precise for a reader to understand. Basically, a dictionary defines a name space within which names have preassigned meanings.

Each dictionary has a character string name which uniquely identifies it to users. An *MP tag* associates an MaT rooted at a node with a particular dictionary. A dictionary has a well-defined format and contains different sections for operators, constants, and annotations, etc. A dictionary entry normally consists of an *index*, a *symbol* (string name), and a *definition* that spells out the meaning of the symbol. The index ranges from 0 to 255. It is possible to have entries with no index. Here is a sample constant entry

```
17  Pi  Circumference / diameter of circle
```

- Transport layer independence – The MP library routines are written in C. The implementation can use different network transport mechanisms such as TCP/IP, PVM, MPI, etc. This is achieved by having a well-defined MP to transport layer interface, making it very easy to port MP.

Thus, MP can work with the other tools described to connect heterogeneous processes. And MP has also been used to connect distributed systems by researchers outside Kent [3]. The source for the MP.1.1.1 is available by anonymous FTP from `ftp.mcs.kent.edu` in `/pub/MP`.

10 Email and Web Communication of Mathematical Data

Not all MP applications have to do with behind-the-scenes interactions among tasks. A user should be able to access MP-enabled services directly. Some utilities have been developed allowing communication of MP-encoded data by email and World-wide Web services (Fig. 5).

MP-encoded data can be sent and received via MIME-compliant mail agents, such as **elm**, using the MIME [5] content type

```
application/mp
```

An MP viewer, `mpview`, can convert MP-encoded data into prefix, infix, and LaTeX formats. A mail browser can call an external program which employs the `mpview` to appropriately display a mail message with `application/mp` content.

A Web server can be configured to return MP-encoded documents which can be displayed by a Web browser through the external `mpview` program. The `SymbolicNet` *live demo* page has several interactive demos (Fig. 6) that return MP-encoded machine-readable data using this scheme. The viewer software is available for downloading.

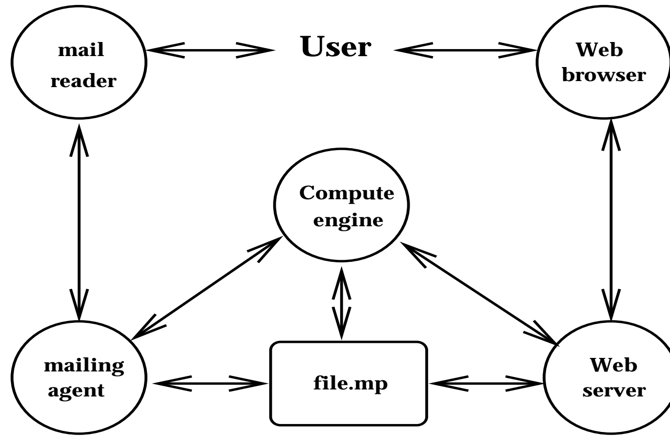

Figure 5: Communicating MP-encoded Data

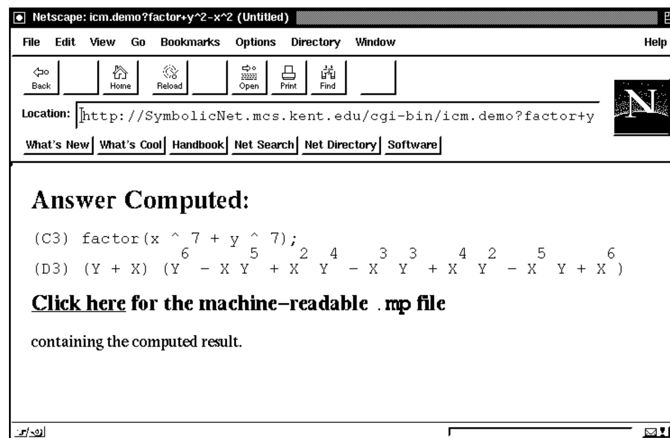

Figure 6: Web Service using MP

Initial experiments have shown the feasibility of *compute-by-mail* where a user sends email computation requests encoded in MP to a compute server that is triggered by an incoming email message and returns the computed results in MP-encoded form to the message sender (Fig. 7).

11 Summary

A set of tools have been constructed to connect symbolic, numeric, and other compute servers together using PVM. PVM-ET makes compilation and management of PVM tasks easy. CL-PVM makes possible running LISP and MAXIMA programs as PVM tasks. Another library interfaces SACLIB-based SAC systems under PVM. A generic bag-of-jobs utility makes scheduling and synchronizing PVM tasks much easier. Efficient transfer of mathe-

matical data is achieved through MP which can help establish a standard. The tools provide efficient and practical ways to conduct distributed and parallel computations and to connect heterogeneous compute servers including SAC systems. These tools address essential enabling technologies for building problem solving environments.

The tools are being used to investigate parallel algorithms for Gröbner basis, polynomial GCD, and characteristic set algorithms. Tools related to email and web access of SAC computation have great potential for a wide range of applications. All the software packages described are available by FTP.

12 Acknowledgement

Work reported here covers research by a number of collaborators over a period of time. Specifically, the author wishes to acknowledge Iyad Ajwa (BJ and SACLIB-PVM), Olaf Bachmann (MP and applications), Raymond Cline (PVM-ET), Simon Gray (MP), Norbert Kajler (MP), Liwei Li (CL-PVM), Mike Lewis (PVM-ET), and Hong Ong (BJ).

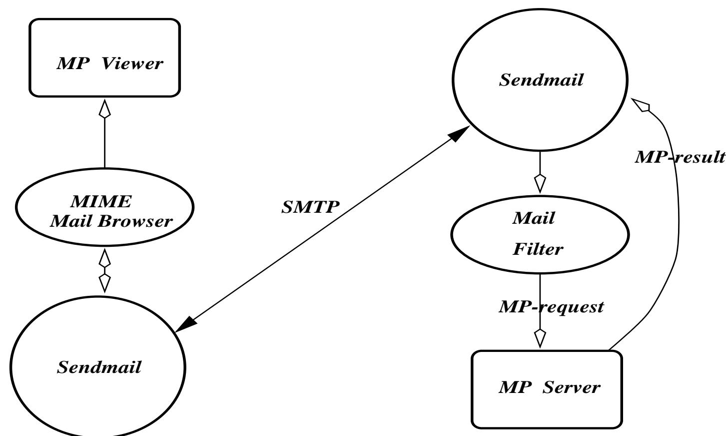

Figure 7: Compute-by-Mail

References

[1] J. Abbott, A. Diaz, and R. S. Sutor, "A Report on OpenMath," ACM SIGSAM Bulletin, pp. 21-24, March, 1996.

[2] I. Ajwa and P. S. Wang, "Parallel Gröbner Bases Algorithm with PVM," the Fourth U.S. PVM Users' Group Meeting, `http://www.cic-8.lanl.gov/pvmug96`, Santa Fe, New Mexico, Feb. 25–27, 1996.

[3] O. Bachmann, S. Gray and H. Schoenemann. "MPP: A Framework for Distributed Polynomial Computations," Proceedings, ISSAC'96, Zurich, Switzerland, pp. 103-109, July 1996.

[4] A. D. Birrell and B. J. Nelson, "Implementing Remote Procedure Calls," ACM Transactions on Computer Systems, 2-1, pp. 39-59, Feb. 1984.

[5] N. Borenstein and N. Freed, *MIME: Mechanisms for Specifying and Describing the Format of Internet Message Bodies*, the network working group RFC-1521, Sept., 1993.

[6] B. Buchberger *et al. SACLIB 1.1 User's Guide*. RISC-Report No. 93-19, Linz, Austria, 1993.

[7] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek and V. Sunderam, *PVM 3 User's Guide and Reference Manual*, Oak Ridge National Laboratory, 1993.

[8] G. Cooperman, *STAR/MPI: Binding a Parallel Library to Interactive Symbolic Algebra System. Proc. of ISSAC'95*, pp. 126-132, Montreal, Canada, July 1995, ACM Press.

[9] A. Diaz, E. Kaltofen, K. Schmitz, T. Valente, M. Hitz, A. Lobo, and P. Smyth, *DSC: A System for Distributed Symbolic Computation. Proc. of ISSAC'91*, pp. 323-332, Bonn, Germany, July 1991. ACM Press.

[10] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM 3 User's Guide and Reference Manual*. Technical Report ORNL/TM-12187, Oak Ridge National Labs, Oak Ridge, TN, May 1993.

[11] S. Gray, N. Kajler, and P. Wang, "MP: A Protocol for Efficient Exchange of Mathematical Expressions," Proceedings, ISSAC'94, Oxford, UK, ACM Press, pp. 330-335, July, 1994.

[12] S. Gray, N. Kajler and P. S. Wang, "Design and Implementation of MP, a Protocol for Efficient Exchange of Mathematical Expressions," *Journal of Symbolic Computation*, 1996 (to appear).

[13] W. Gropp, R. Lusk, and A. Skjellum, *Using MPI*, MIT Press, Cambridge, MA, 1994.

[14] L. Li and P. S. Wang, "CL-PVM: A Common Lisp Interface to PVM," Proceedings, PVM User's Meeting, Pittsburgh, PA, May 7-10, `http://www.cs.cmu.edu/Web/Groups/pvmug95.html`, 1995.

[15] L. Li and P. S. Wang, "The CL-PVM Package," SIGSAM Bulletin, Vol. 29, No. 3&4, December 1995, pp. 2-8.

[16] N. Kajler. Building a Computer Algebra Environment by Composition of Collaborative Tools. *Proc. of DISCO'92*, volume 721 of *LNCS*, pages 85–94, Bath, GB, April 1992. Springer-Verlag.

[17] N. Kajler. CAS/PI: A Portable and Extensible Interface for Computer Algebra Systems, *Proc. of ISSAC'92*, pages 376–386, Berkeley, USA, July 1992. ACM Press.

[18] Orfali, et. el., *The Essential Distributed Objects Survival Guide*. Wiley, 1996 (ISBN 0471-12993-3).

[19] J. Purtilo, "Applications of a Software Interconnection System in Mathematical Problem Solving Environments," Proceedings, 1986 Symposium on Symbolic and Algebraic Computation, ACM 1986, pp. 16-23.

[20] J. Siegel, *CORBA Fundamentals and Programming*. Wiley, 1996 (ISBN 0471-12148-7).

[21] P. S. Wang, M. J. Lewis, and R. E. Cline, Jr., "PVM Installation, Usage, and Performance on HEAT," Internal Technical Report, Sandia National Labs, Livermore, CA.

[22] P. S. Wang, "Tools to Aid PVM Users," the Fourth U.S. PVM Users' Group Meeting, Santa Fe, NM, Feb. 25–27, `http://www.cic-8.lanl.gov/pvmug96`, 1996.

[23] P. S. Wang, "PVM Guide," ICM/Kent Technical Report (ICM-199509-02), Institute for Computational Mathematics, Kent State University, `http://www.mcs.kent.edu/cgi-bin/publish/access` (1995).

[24] T. Yuasa and M. Hagiya, "Kyoto Common Lisp Dictionary," Kyoto University, KCL on-line documentation," 1986.

[25] "External Data Representation Protocol Specification," *4.3bsd UNIX System Manager's Manual, Volume 2*, University of California at Berkeley, 1986.