SOWING MPICH: A CASE STUDY IN THE DISSEMINATION OF A PORTABLE ENVIRONMENT FOR PARALLEL SCIENTIFIC COMPUTING*

WILLIAM GROPP† AND EWING LUSK‡

Abstract. MPICH is an implementation of the MPI specification for a standard message-passing library interface. In this article we focus on the lessons learned from preparing MPICH for diverse parallel computing environments. These lessons include how to prepare software for configuration in unknown environments; how to structure software to absorb contributions by others; how to automate the preparation of man pages, Web pages, and other documentation; how to automate prerelease testing for both correctness and performance; and how to manage the inevitable problem reports with a minimum of resources for support.

1. Introduction. MPI is a specification for a standard message-passing library interface. MPI was intended to facilitate widespread portability of programs among diverse parallel architectures. To help achieve this goal, at the first meeting of the MPI Forum in 1992, we volunteered to provide a reference implementation of MPI. The implementation was to begin immediately and to track the evolution of the MPI standard from meeting to meeting instead of waiting for its completion. Within a week, major parts of the initial draft were implemented, and MPICH continued to track the development through to completion.

What enabled us to implement the initial draft so quickly was considerable experience with the portable parallel message-passing systems Chameleon [15] and p4 [3]. As a result of those experiences, we decided to pursue an aggressive program of relying on automated tools—both existing ones and those we would write–to help us design, write, test, distribute, and maintain MPICH.

In our opinion, this program has been quite successful: we have developed alongside MPICH a set of procedures and tools for disseminating portable parallel software that is independent of MPICH itself. In this article we discuss our experiences; offer advice to other tool developers involved in large, multiperson, multiyear projects;

*This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

†Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439
‡Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439

and describe simple tools that users are welcome to appropriate from the MPICH distribution and adapt to their own purposes.

The article is organized as follows. In Section 2 we outline in more detail the goals of the MPICH project, which motivated the procedures and tools we used. In Section 3 we describe how we dealt with the problem of multisite development of a large system. Section 4 describes how the system was designed for portability. In particular, we describe our heavy reliance on GNU `autoconf`, which is a fundamental component of the MPICH distribution strategy. In Section 5 we describe how we manage documentation in the form of manuals, `man` pages, and Web pages. Section 6 describes how we conduct automated testing of the system, and Section 7 describes the process of releasing the system for distribution. In Section 8 we discuss our tools for managing interactions with users. We conclude with a summary of what we have learned and how we would design tools and procedures for the next portable parallel programming project.

2. Goals. The strategies and tools we describe here were directly motivated by the original goals of the MPICH project. MPICH has been both a research project and a software project. As a research project, it has explored the degree to which portable systems could achieve high performance. As a software project, it has promoted the adoption of the MPI standard by providing users with an early, free implementation and vendors with a running start on their own proprietary implementations. The architecture of MPICH, as well as the tradeoffs made between portability and performance, has been described in [13]. Here, we focus on the project goals that were independent of the particular system being implemented. Such goals presumably are shared by developers of other portable parallel environments.

Goals for the code. The goals we had for our code were the normal ones—robustness and performance—plus the extra one of extreme portability. We wished MPICH to be installable in a great variety of parallel environments, ranging from distributed-memory machines to shared-memory parallel machines to heterogeneous workstation networks. Parallel systems offer particular challenges in the area of portability, both because they must rely on a number of operating system features and because approaches to parallelism are so different in various systems.

Goals for users. Ease of configuration and installation were major goals. Documentation was critical. We also expected users to take advantage of MPICH's portability and use it in multiple parallel environments. Thus, we tried to achieve portability not only of the user's application programs, but also of the user's `Makefile`s and execution scripts.

Goals for us. We naturally wished to minimize the effort of developing parallel code for a wide variety of environments. We needed to manage a relatively large amount of code and to collaborate with others in both writing and maintaining it. We needed to minimize the overhead of release management. Finally, we wished to respond to bug reports, both to ensure user satisfaction and to increase system robustness.

3. Managing Code Development. The MPICH distribution contains 2,449 files, of which 1,188 contain the 231,220 lines of source code. While these numbers are only an approximate measurement of size and complexity, they do show that MPICH is a nontrivial project and is large enough to justify investment in automated tools and procedures.

3.1. Structuring of MPICH Code. MPICH contains many pieces: source code for the MPI standard library routines (in C and Fortran), documentation in the form of `man` pages and manuals, examples and test programs, and assorted tools and programming environment add-ons (such as profiling and graphics libraries). We decided to maintain a single directory tree containing all components. The MPICH distribution is extracted from this tree when a release is made (see Section 7).

Three particular decisions served us well. The first was to divide the implementation into layers, as described in [13]. The lower layer comprises multiple implementations of a single "abstract device." Somewhat inaccurately, we refer to these multiple implementations as "devices." Many of these devices are direct interfaces to the existing communication libraries of particular vendors (for example, the T3D, NX, and Meiko devices); others are implemented on lower-level portable systems (such as the p4, Nexus, shared-memory, and TCP devices). The main benefit of this structure is that most of the MPICH source code is shared among all the devices. Such sharing applies not only to high-level MPI routines such as datatype and topology manipulation functions (which one would expect to be portable) but also to the memory management and data structure manipulation code associated with the abstract device itself, which defines a uniform message-passing layer based on multiple protocols for messages of varying sizes. The lower-level devices are then used to implement this layer. As a result, we have been able to

- spend most of our time working on code that is common to all platforms on which MPICH runs (even much of the device code is shared among multiple devices);
- accept new device implementations from vendors and others, smoothly absorbing them into MPICH (in particular, the T3D, Meiko, NX, Convex SPP, and Nexus devices were written outside our group and plugged into MPICH via the Abstract Device Interface described in [9] with a minimum of difficulty); and
- experiment with new devices wthout impacting the stability of existing code (a recent example was a new version of the generic shared-memory device that avoided most uses of locks [12]).

Our second decision regarding the source code management concerned the Fortran-callable versions of the MPI library routines. The Fortran interface consists of a set of wrapper functions that transform their arguments into the correct form for the C versions and then call the standard C versions of the functions. Automated production of the wrapper functions was especially useful during the period when the MPI standard was changing every six weeks and the language bindings were in flux. We used the highly configurable `bfort` program, described in [6]. Particular problems that `bfort` handled automatically for us were

- adding the extra error return code argument for all Fortran MPI functions, and setting it as the return value of the corresponding MPI function;
- converting C pointers to Fortran integers and back again;
- automatically generating Fortran names (the wrapper functions are written in C but must have Fortran names, which differ from one system to another; we used `configure` to figure them out, as described in Section 4.1.)
- dealing with 64-bit problems as necessary (e.g., if a pointer fits into an integer, we just cast it, but otherwise we used a mapping; again, `configure` determined what was necessary).

The third decision involved code modularity with respect to subsystems of the MPI implementation. The modular approach made it easy to completely replace the code for dealing with datatypes, for example. We expect to be able to upgrade the algorithms used in the collective operations without impacting any other code.

3.2. Distributed Development of MPICH. Because the number of people working at any one time on MPICH was small, we did not need a sophisticated distributed source code management system. We relied on RCS, using it primarily through the Emacs interface. (Emacs itself, of course, provides considerable support for writing programs.) To handle the large number of files in many subdirectories, we wrote a few scripts to help us with the low-level RCS-related tasks:

- `needrcs` searches the whole `mpich` directory tree for files and directories that appear to require being put under RCS control.
- `torcs` puts a file, a list of files, or a whole directory under RCS control, supplying our own values as the arguments to the `rcs -i` command.
- `findlocks` searches all the RCS directories for locked files and identifies who holds the lock. This is useful for finding files left in the locked state accidentally.

We are now beginning to experiment with CVS [1], which provides more support for multi-site work by allowing checkout of whole subsystems at a time.

4. Managing Portability. MPICH runs on massively parallel processors such as the IBM SP-2, Cray T3D, Intel Delta and Paragon, Meiko CS-2, TMC CM-5, and NCUBE; shared-memory multiprocessors such as the Convex Exemplar, SGI Power Challenge, and NEC SX-4; on Cray C-90's; on workstations from Sun (both SunOS and Solaris), IBM, DEC, and HP; and on PC's running Linux, FreeBSD, or Solaris. Although these are all "Unix machines," they collectively illustrate the difficulties of writing portable Unix programs. Some of the obstacles to portability are

- different OS functionalities (e.g., how signals work);
- compiler names and flags;
- lengths of basic data types, including long integers, pointers, and long doubles;
- locations of header files and libraries (X);
- multiple, incompatible C and Fortran compilers; and
- library contents and locations (-lsocket, -lthread).

(We note that some of the major obstacles to portability of Unix applications are due to the failure of the Unix community to standardize aspects of the operating system and environment important to software developers. The mechanisms we describe in this section are a far cry from the "setup" command familiar to PC users.)

4.1. Portability for the Developers. The basic problem that we had to address was the multiplicity and variability of target environments. Although the number of systems is small enough to make a list of system names that could be tested for with `#ifdef`s, we have found this method too restrictive because of the many variations among and within the systems. These variations arise both from release-to-release variability and from local installation variations. It is far better to test for specific capabilities and properties than for a system name.

Our previous experiences led us to rule out certain mechanisms for managing portability. Chameleon used `include` statements in its makefiles to enable users to assemble the makefiles required on a given system. This mechanism proved inconvenient for users and is not supported by some `make`s (such as BSD 4.4). p4 used an environment definition file, with a set of `Makefile` variables for each environment, that was grafted onto prototype `makefile`s. This was convenient for users who had one of the supplied environments, but inconvenient for the authors, who constantly had to increase the set of supported environments, often just to provide a minor variation. We also considered using `xmkmf` but found it more cumbersome to use than `autoconf`, as well as being tied to the X distribution and local installation. We found GNU `autoconf` [1] to provide an excellent solution. GNU `autoconf` is a set of `m4` macro definitions from the Free Software Foundation that makes it easy for a developer to create a `configure` script for delivery with a software package. The recipient does not need to have `autoconf` on site; the `configure` script is a Bourne shell script that can be executed by all but the worst implementations of `sh`. When the `configure` is run in a particular user environment, it investigates the environment dynamically to determine such information as the name of the C compiler, the existence of certain header files, and the location of the X libraries. It can also run developer-supplied programs to test properties of the environment and system software. We used `configure` to determine everything that we needed to build MPICH, from the data type returned by `malloc` to the external names used by the linker to recognize Fortran functions. Our `configure` recognizes various OS versions and sets compiler flags appropriately. Some of the more interesting tests carried out by MPICH's `configure` script are

- identifying strange `make`s,
- testing for known bugs in certain compilers,

- testing for availability of international message catalogs,
- testing whether Fortran accepts `-I`,
- testing whether signals are reset when used, and
- determining the number of arguments for `gettimeofday`.

Arguments to `configure` can control MPICH options, such as which device to use, whether to build all the `mpe` support libraries, or whether to compile in various debugging modules. Even with no arguments, however, the `configure` script can deduce enough of the environment to produce a working configuration. The output of `configure` includes `Makefile`s in all MPICH subdirectories and several specially prepared scripts.

Here is a fragment of output from `configure` as run on a DEC alpha workstation:

```
Configuring with args -device=ch_shmem
Configuring MPICH Version 1.0.13.
*#
*# You should register your copy of MPICH with us by sending mail
*# to majordomo@mcs.anl.gov containing the message
*#   subscribe mpi-users
*# This will allow us to notify you of new releases of MPICH.
Trying to guess architecture ...
  configuring for "alpha" target architecture
checking for ranlib
checking gnumake... no
checking BSD 4.4 make... no - whew
checking OSF V3 make... Found OSF V3 make
The OSF V3 make does not allow comments in target code.
Using this make may cause problems when building programs.
You should consider using gnumake instead.
 . . .
checking for mmap... yes
checking for msem_init... yes
checking for mutex_init... no
checking for shmat... yes
checking for semop... yes
```

```
checking that signals work correctly... yes
checking for hostname... found /usr/ucb/hostname (1)
 . . .
checking for nl_types.h... yes
Generating message catalogs... done
checking for ANSI C header files
checking for stdlib.h... yes
checking for malloc return type... void
 . . .
Fortran externals have a trailing underscore and are lowercase
checking for Fortran include argument... supports -I
checking Fortran has pointer declaration... yes
checking for correct handling of conditionals...... yes
checking for correct handling of conditionals part 2 ...... yes
checking that compiler truncates unsigned char correctly ...... yes
 . . .
checking for size of void *... 8
checking for pointers greater than 32 bits... yes
checking for size of int... 4
checking for int large enough for pointers... no
checking for long double... yes
checking for long long int... yes
checking size of double... 8
checking size of long double... 8
 . . .
```

Assembling the master `configure.in` file, from which `autoconf` builds `configure`, was a complex and incremental task, since we encountered various flavors and releases of Unix over time. However, in many cases now, no changes need to be made. The reason is that `configure` asks specific questions about the environment and thus can adapt to a new system never seen before.

We learned from using `configure` that it is important not to associate lines of source code with specific architectures or operating systems (e.g., with `#ifdef AIX ... #endif`). Rather, the `#ifdef`'s should refer to a specific capability tested for by `configure`. This more fine-grained mechanism deals better with different compilers (vendor-supplied compiler vs. `gcc`, for example), which may align doubles differently, or different OS releases, which may requre different numbers of arguments to system routines.

4.2. Portability for Users. The advantage of our use of `autoconf` is that users can build MPICH in any environment simply by typing

```
configure
make
```

The default `make` target in the top-level directory constructs all the libraries and executable commands that are part of the MPICH programming environment, and compiles and links a small example to make sure everything is working.

We supply portability for users' *applications* by having `configure` also build three scripts—`mpicc`, `mpif77`, and `mpirun`—that encapsulate the information learned by `configure` when it was run. Hence, the user can use

```
mpicc myprog.c
mpirun -np 16 myprog
```

to compile, link, and run an MPICH application in any environment where MPICH has been installed.

Additionally, we have provided help for systems adminstrators who wish to install MPICH for use at their sites. Specifically, the `mpiinstall` script enables them to put the libraries, executable commands (such as `mpif77` and `mpirun`), and the `man` pages into standard places where their users expect to find them. Such an installation is also much smaller than the entire MPICH source distribution, which contains extensive example and test programs in addition to the source code for the MPI library itself.

5. Managing Documentation. A hallmark of high-quality software is its documentation. The specification for MPICH is given by the MPI standard itself, and we maintain an HTML version of the standard for online reference [5]. MPICH augments the standard with additional documentation, the preparation of which was facilitated by a number of automated tools.

5.1. `Man` Pages. MPICH inherited from PETSc [2] a tool for turning structured comments in the source code into `man` pages. The tool is called `doctext` [7]. In particular, source code of the form

```
/*@
MPI_Comm_rank - Find rank of calling process in the communicator

Input Parameters:
. comm - communicator (handle)

Output Parameter:
. rank - rank of the calling process in group of  'comm'  (integer)

.N fortran

.N Errors
.N MPI_SUCCESS
.N MPI_ERR_COMM
@*/
```

is turned into text using the nroff `man` macros that are recognized by all the standard `man` page display tools, such as `xman`, Emacs `man` command, or `troff`, and can also be turned into HTML text recognized by Web browers. A script, `mpiman`, is provided in MPICH as an interface to `xman` for reading the MPICH `man` pages. Nearly all the source code in MPICH uses this convention, so that the `man` pages distributed with MPICH include not only all the MPI Standard routines, but also the MPE extensions, the user commands, and the abstract device interface routines. It is easy to maintain the `man` pages automatically because their contents are embedded in the source code itself. (The `.N` notations include text that is constant from one `man` page to another.)

5.2. Manuals. We decided to separate the end user manual [11] from the installation manual [10], so that each type of user would have only what was needed. In particular, the user's manual assumes that MPICH has already been installed and instead focuses directly on how to build and run user applications. The LaTeX-to-HTML converter `tohtml` [8] enabled us to put the *MPICH User's Guide* and the *MPICH Installation Guide* on the Web for easy reference and was critical for creating a Web version of the MPI standard itself. These Web pages (and `tohtml` itself) are available for users to install on their own Web sites.

5.3. Documenting Local Installation and Performance Data. An experimental part of MPICH is the program `port`, which can be used as a wrapper for the installation process. The command `doc/port`, issued in the top-level directory instead of `configure`, not only will configure and build MPICH, but also will prepare a report (in LaTeX format) describing what `configure` learned about the environment, how long it took to install, what problems were encountered during installation, and graphical performance results based on running the `mpptest` benchmark (see [13]). Such reports provide a snapshot of the MPICH status at any time on any machine. A full set of these reports is maintained on our Web site so that remote users can compare their experiences with ours.

5.4. Examples. It has been said that "a running example is worth a thousand `man` pages." MPICH comes with a rich set of example programs to help users get started with MPI programming. The most basic examples are minimal MPI programs. The test suite (described in Section 6.1) contains examples of the use of every MPI function. More elaborate demonstration programs, discussed in the book *Using MPI* [14], are furnished; and more exotic programs, using the `mpe` parallel graphics library that comes with MPICH, provide examples of how more complex applications might be assembled.

6. Managing Testing. One goal in distributing software is that users encounter no bugs. The state of software science is such that our best defense against bugs is prerelease testing. Testing has (at least) two parts: assembling the tests themselves, and actually running them. In a project designed for extreme portability, both are much more difficult than in a simpler project. The failure of test programs may be highly system dependent, so that successful testing in one environment may not guarantee the success of the same test in another environment, even for parts of the code that are supposed to be platform independent. Because tests therefore must be carried out in multiple environments, it is difficult and time consuming to ensure that all necessary tests are done and to examine their output. When a bug is fixed as the result of a test failure, it is necessary to retest in all other environments to ensure that a fix for one machine does not introduce a new bug on another. The specter of an infinite loop looms. In this section we describe our testing procedures.

6.1. Assembling the Test Suite. Our test suite began with a few straightforward programs to exercise various MPI library routines and has grown to a fairly extensive collection that really hunts for bugs. It exercises all of the MPI library's functions and tries to exercise multiple paths within the routines. For example, it deliberately calls them erroneously and checks that they behave according to the standard (returning the right error codes) and as we expect. When a user submits a program showing a bug that has escaped our testing, we add this program to the test suite. We have also added partial test suites that some vendors have made available, which they have used for testing their own implementations.

Even so, we cannot claim that the test suite has been assembled as systematically as those of commercial systems. Putting together a systematic "validation suite" for the MPI Standard would be a separate and major project in its own right, which we would hope to contribute to. (An effort to develop such a validation suite is under way at Intel with ARPA support.)

6.2. Running Tests. During a working day, many changes may be made to many parts of the code. One usually at least compiles a change one has just made, but only on the system on which one is working. (We have multiple systems sharing a common user file system.) To ensure that daily changes are systematically tested, we rely on the "nightly build." A script launched by `cron` attempts to compile, link, and test MPICH on a large variety of systems each night. Recently the number of systems has become large enough that it takes about three nights to get through the entire cycle. We can test locally (without moving any files) the Sun-OS, Solaris, IRIX, HPUX, AIX, DEC, and SP versions; we have recently begun using `expect` (a `tcl` tool) to manage automated testing on remote machines where we must transfer files before testing. Each morning the results of the nightly build are mailed to us, and a `finderrors` script quickly locates the trouble spots in the fairly voluminous output.

Tests can also be run by hand; the `runtests` script can run a complete or partial set of tests in any of the test directories and compare the output with preprepared expected output to quickly locate discrepancies.

A special type of test is done by the `tstmachines` script. It checks a user's workstation network environment to make sure that user-supplied machine names are valid and that permissions allow the execution of the network version of MPICH.

6.3. Testing for Performance. Since high performance is an important goal of MPICH, performance bugs must be identified and repaired along with correctness bugs. The test suite distributed with MPICH contains a relatively sophisticated performance test program (`mpptest`, described in [13]) that can be used for tuning high-performance device implementations and ensuring that correctness bug fixes do not (unnecessarily) impact performance.

7. Managing Distribution. It is often difficult to tell just exactly the right time for a public release of a new version. Balancing one's natural wish to wait until new features are completed is the desire to release a new version that has fixed many bugs. We have automated the process as much as possible.

7.1. Bundling Files. Over time, the working copy of MPICH tends to become cluttered with temporary files, test input and output, and occasional core dumps. It also contains the RCS directories. Rather than periodically scrub our working files from the tree, which would have to be done quite carefully, we put together a `tar` file (either for distribution or just to move to another site) by copying the "official" files out of the working directory and tarring them up separately. The script `maketar` does this.

7.2. Creating a Release. In the case of an official distribution, building the final `tar` file for distribution is the final step of a complex process with many steps. The `makedist` script

- runs `autoconf` in several subsystems to update `configure` scripts;
- checks the working source tree to make sure no large files unexpectedly find their way into the distribution (forgotten test executables, NFS trash files of the form `.nfsxxx`, leftovers from unreliable `ars`, etc.);
- runs `doctext` to recreate the `man` pages from the structured comments in the source code, in both `man` and `html` formats;
- remakes the *User's Guide* and *Installation Guide* from the original LaTeX source text;
- builds the final `tar` file with `maketar`;
- runs `testdistrib` to rebuild and retest on selected systems; and
- runs `port` to write reports on the results of the final tests.

Since `makedist` was developed the process of creating a release has become much easier, since all the complicated checklists are gone through automatically.

After this process has completed successfully (and several tries may be needed), we run `makedist -commit`, which *commits* the release. This final step increments the release number, installs the `.Z` and `.gz` files in the publicly accessible `ftp` directory, makes a `split` version for downloading by people with unreliable connections, and installs the `README` in the `ftp` directory. It also creates the shadow source files to be used with the `makepatch` program (see Section 7.3).

7.3. Patch Management. After the release, when bugs are found and fixed, we create patches that can be fetched and installed by users. The patches are numbered according to the bug number assigned by `req`. A Web page maintains bug descriptions and the corresponding patch numbers. A script `makepatch` creates the patch and installs it on the Web.

8. Managing User Interactions. Currently, over 600 users have added themselves to our MPICH announcement list. The code itself has been downloaded thousands of times. Some users are actually meta-users, who install MPICH at their site for a large group to use. With this number of users, relying on our personal e-mail backlogs for handling user problems does not work. We needed a system that would allow multiple developers to share responsibility for bug reports, that would allow us to keep track of the dialog with a problem reporter and maintain status of the bug fix in progress, and that would be easy for users to use. After considering a number of alternatives, we chose `req` [4]. This system works entirely by observing, modifying, and responding to the `Subject:` line in ordinary e-mail. Users usually are not even aware that they are interacting with a problem-tracking system. The `req` system has a companion GUI version based on `Tcl/Tk`; we find both versions useful. Although we cannot claim zero-length backlog on responding to problems, no reports are ever lost and the e-mail dialog on a particular problem is always available in a single file. The mailing list to which `req` is attached, `mpi-bugs@mcs.anl.gov`, is managed by `majordomo`, which makes it easy for us to allow others to observe bug-report tracking if they wish.

A number of problems that do not represent bugs but, rather, failure to read the *User's Guide* occur frequently enough that we have found it useful to provide a tool for issuing a standard response (other than "RTFM") without having to reconstruct it each time. The *User's Guide* contains a "Problems" section in question-and-answer format. This part of the manual is actually constructed from a question-and-answer database that is searchable by the script `fmsg`, distributed with MPICH. For example, users with malconfigured workstation networks can receive a "permission denied" message when they run MPICH. The *User's Guide* contains a complete discussion of this problem and a number of different solutions. When we receive a bug report that we identify as pertaining to this problem, instead of merely pointing the user to the manual, we can type

```
fmsg permission
```

to obtain the full text of the discussion from the manual, which can then be pasted into an e-mail reply. This type of response to problem reports is obviously a candidate to become a Web application, bypassing e-mail altogether. We have not yet implemented this approach.

9. Conclusions.

In this section we summarize what worked well for us and what we might do differently on the next project.

9.1. What Worked Well.

The attempt to use general-purpose tools to manage the task of producing MPICH has been, in general, a success. The tools described in this article all have had heavy use, and we could not have managed nearly as efficiently without them. They fall into four categories:

- The primary tool for providing portability—`autoconf`—was critical. We enthusiastically recommend it.
- Tools for automating the production of nontrivial amounts of both code and text: `bfort`, `tohtml`, and `doctext` saved us enormous amounts of time. Without them, the Fortran interface would have been late, and the documentation would be sparser.
- Tools for managing e-mail interaction with users: we are still happy with `req` and `tkreq`.
- Small but useful tools for automating repetitive tasks: `runtests`, `maketar`, `makedist`, `makepatch`, `findlocks`, `makepatch`, `tstmachines`, `mpiinstall`, etc., were as helpful in ensuring correctness of procedures (since they could be debugged and monotonically improved) as they were in saving keystrokes.

9.2. What We Might Do Differently Next Time. Although we view the approach as successful, and we would keep most of the details, the experience has left us with a number of resolutions for the next similar project. Some of these will no doubt eventually find their way into MPICH.

- Organize and document the "project management" scripts and procedures ahead of time.
- Use CVS instead of RCS for source code management, especially if a significant number of remote co-developers are involved.
- Structure the distribution more carefully so that separate parts can be easily distributed, yet fit into an integrated whole. Not everyone needs the whole system. We might even distribute prebuilt object libraries for some systems, so that users need not configure at all.
- Use a "real" database system to manage test results, both for correctness and performance history. The test programs could be structured with their output designed for this database rather than for human consumption.
- Organize even more of the documentation and user interaction software around a Web interface.

The MPICH project has been satisfying and educational. In this article we have presented those aspects of the project that were independent of the actual *content*, and we have described the techniques and tools that might be common to any project whose goal is creating portable, parallel tools and distributing them to a user community.

All of the tools described here are freely available, either directly from their distributors (`autoconf`, `RCS`, `req`) or in the MPICH distribution (`util` subdirectory).

REFERENCES

[1] GNU manuals. `http://www.delorie.com/gnu/docs`.

[2] PETSc 2.0 for MPI. `http://www.mcs.anl.gov/petsc/petsc.html`.

[3] Ralph Butler and Ewing Lusk. Monitors, messages, and clusters: The p4 parallel programming system. *Parallel Computing*, 20:547–564, April 1994.

[4] Remy Evard. Managing the ever-growing to do list. In *USENIX Proceedings of the Eighth Large Installation Systems Administration Conference*, pages 111–116, 1994.

[5] The MPI Forum. MPI message passing interface standard, version 1.1. `http://www.mcs.anl.gov/mpi/mpi-report-1.1/mpi-report.html`.

[6] William Gropp. Users manual for `bfort`: Producing Fortran interfaces to C source code. Technical Report ANL/MCS-TM-208, Argonne National Laboratory, March 1995.

[7] William Gropp. Users manual for `doctext`: Producing documentation from C source code. Technical Report ANL/MCS-TM-206, Mathematics and Computer Science Division, Argonne National Laboratory, March 1995.

[8] William Gropp. Users manual for `tohtml`: Producing true hypertext documents from LaTeX. Technical Report ANL/MCS-TM-207, Argonne National Laboratory, March 1995.

[9] William Gropp and Ewing Lusk. An abstract device definition to support the implementation of a high-level point-to-point message-passing interface. Preprint MCS-P342-1193, Mathematics and Computer Science Division, Argonne National Laboratory, 1994.

[10] William Gropp and Ewing Lusk. Installation guide for `mpich`, a portable implementation of MPI. Technical Report ANL-96/5, Argonne National Laboratory, 1994.

[11] William Gropp and Ewing Lusk. User's guide for `mpich`, a portable implementation of MPI. Technical Report ANL-96/6, Argonne National Laboratory, 1994.

[12] William Gropp and Ewing Lusk. A high-performance MPI implementation on a shared-memory vector supercomputer. *Parallel Computing*, 1997. (to appear).

[13] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the MPI message-passing interface standard. *Parallel Computing*, 22:789–828, 1996.

[14] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 1994.

[15] William D. Gropp and Barry Smith. Chameleon parallel programming tools users manual. Technical Report ANL-93/23, Argonne National Laboratory, March 1993.