PGMAKE: A Portable Distributed Make System

Andrew Lih and Erez Zadok
Computer Science Department, Columbia University

CUCS-035-94

ABSTRACT

We describe pgmake, which extends the GNU project's *make* utility to support distributed job execution using the Parallel Virtual Machine (PVM) package from Oak Ridge National Laboratory. These two packages were chosen because of their high level of portability to different architectures, free source code distribution policy and availability to the public.

Using medium-sized farms of modest performing workstations, our system has achieved software build times faster than expensive high-speed uni- and multiprocessors. The highly portable code adddditions make this implementation easy to port among various platforms.

1 Introduction

The make utility was written by Stu Feldman [4] in the mid 1970's to automate the process of target generation based on modification of dependency files. Since that time, enhancements to make have been few in number, and limited to rule specification language enhancements.

In recent years, computing power has moved from large, centralized timesharing systems to high-powered workstations connected by high-speed local area networks. The proliferation of individual workstations on user desktops resulted in efforts to provide effective ways to combine and realize their aggregate power. The computing power of these workstations lies unharnessed when users are either inactive or performing non-CPU intensive tasks. As a result, many potentially useful CPU cycles pass by unutilized. Several projects have attempted to extend make to utilize these distributed computational resources. However, most of efforts are fairly non-portable, either because they are operating system dependent, rely on specialized transport protocols [1], or require rewriting of configuration files. pgmake consists solely of modifications to the job distribution mechanism within GNU make, and provides the exact same operational semantics with which users of GNu make are already familiar.

2 Background

Providing parallel job execution in make is not a new concept. Several commercial varieties of make exist today that support this feature, in addition to several research efforts. The original AT&T make builds a dependency tree, determines "dirty" files that need updating, and executes each update process sequentially. The second generation AT&T nmake is capable of launching multiple parallel commands with the -j N flag, where N specifies how many jobs are allowed to run concurrently on the same machine. Nmake achieves parallelism by issuing commands to a shell co-process.[5] Other make systems with specialized transport protocols include Carnegie-Mellon University parmake with *dp* [8] and Adam de Boor's pmake with *Customs*.[2, 3]

GNU Make. GNU gmake, written by Richard Stallman and Roland McGrath at the Free Software Foundation, was developed as part of the GNU Project to provide free tools familiar to UNIX users. Gmake provides a -j N option, where N defines the number of jobs to run on the same host.[10]

PVM. PVM, Parallel Virtual Machine, was developed at Oak Ridge National Laboratory in conjunction with researchers at the University of Tennessee, Carnegie Mellon University and Emory University[11]. The PVM software package *"allows heterogenous networks of parallel and serial computers to appear as one concurrent computational resource."* Machines defined within the virtual machine run a daemon called pvmd, which is available to spawn tasks on behalf of the user. PVM also provides a C and FORTRAN library that allows users to spawn and manage processes running across the virtual machine. PVM provides a message-passing based paradigm for communicating between active tasks and pvmd processes. Other features of PVM that make it attractive to use as a parallel computing platform include dynamic process groups, transport layer independence, fault tolerance, and simple load balancing. PVM has been well received in the academic community and ported to over 20 different machine architectures.

3 Design

Execution Environment Issues For gmake to faithfully execute a process remotely, it is necessary to duplicate the environment of the caller to the "callee" side. The following list of concerns must be resolved when attempting to remotely execute a (compilation) process.

1. **Architecture and Operating System.** For most operations, it is important that the actual make commands are executed on a machine of the same architecture and operating system of the target. PVM contains knowledge

of the particular host architecture of each CPU in the virtual machine and can be directed to launch jobs only on machines of a given architecture.

2. **Filesystems.** All commands generated by `make` are executed relative to the working directory from which the `make` was issued. In a networked environment, it is important to be able to "root" the remote execution unit in the correct directory before issuing commands. This brings up the problem of uniform global naming scheme for file hierarchies.

   The most widely used networked filesystem, ONC-NFS from Sun Microsystems [9], does not enforce a global namespace for filesystems, which presents a problem in our model. Therefore, it is a requirement that directories in which `pgmake` will be invoked must have the following properties.

   - The directory must be available for NFS mount to all remote nodes, and
   - the name by which it is referred must be globally defined

   Public domain automounters such as `amd` can assist in maintaining global names and maps of shared filesystems.[7]

3. **Time.** The success of any `make` utility lies in its ability to check time stamps of files and determine which ones need rebuilding. Therefore, it is necessary to implement some type of time synchronization protocol between nodes in the virtual machine. The popular Network Time Protocol, *ntp*, for example, is available in the public domain.[6]

4. **Shell Environment Variables.** Executables launched from `make` may use the users' environment variables as parameter settings. Therefore, it is important to duplicate the shell environment variables on the caller side to the remote execution side.

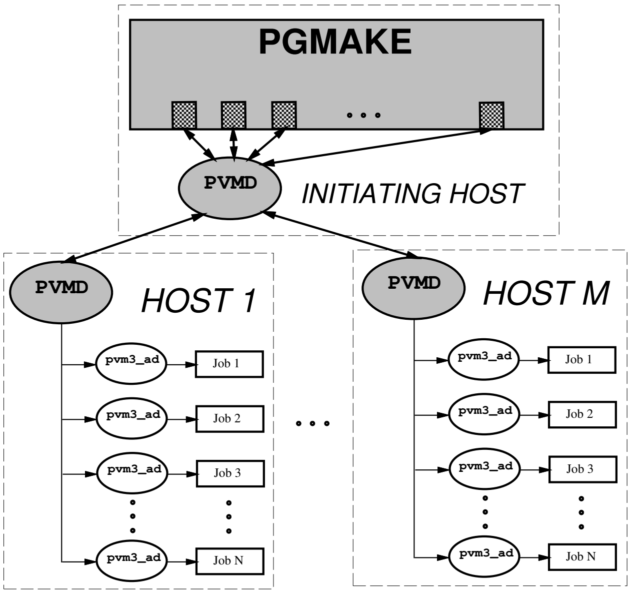

Figure 1: Flow of execution with pgmake

3.1 `gmake` Modifications and Execution Agent

In order to minimize the number of locations and software packages that have to be modified to provide distributed operation, PVM was left unmodified, and modifications were made only to `gmake`. This was also done to make `pgmake` more attractive to potential users, without adversely affecting existing installations of PVM. `Pgmake` currently works with GNU `make` version 3.64, and the latest known version of PVM, version 3.1.

Because GNU `make` already has stub provisions for remote jobs, the bulk of our work was to construct a new module for GNU `make`. Four main interface functions were supplied to provide support for remote jobs. The primary flow of execution is depicted in Figure 1.

The following GNU `make` functions were provided by us to interface with PVM:

- `start_remote_job_p`. This predicate function determines if the next job should be run locally or remotely. If the user provides the "remote" switch to `pgmake`, jobs are allowed to run remotely, and `pvm_taskid()` is called the first time to enroll the process with PVM.

- `start_remote_job`. This function is called by GNU `make`'s `jobs.c` module to execute a job remotely. The function gets passed an argument vector, environment pointer, and a standard-input file-descriptor. `Pgmake` forms a message with these pieces of information along with the current working directory and calls the PVM function `pvm_spawn()` to initiate the remote job. `Pgmake` records the thread ID, *tid*, returned from the call, into a table for future reference.

- `remote_status`. This call is invoked by `pgmake` when it has determined that it cannot issue any more jobs, and needs to wait for a job slot to empty, so that it can either fill it with a new job, or end the entire compilation. At the heart of this call is a [non]-blocking PVM call `pvm_nrecv()`, which is very similar to the UNIX `select()` system call. When a remote job terminates for any reason, its return status is collected and sent to the parent, in addition to any output it generated on *stdout* or *stderr*.

- `remote_kill`. Sends remote `pvmd`'s a notice to terminate the tasks they are currently managing (which is the execution agents.)

Our code changes to GNU `make` number roughly 400 lines, and reside almost completely in one separate and new module, `remote-pvm3.c`. A minor change was made outside this module for supporting two new GNU `make` options: `-R` tells `pgmake` to run all of its jobs remotely if possible, and `-D` turns on verbose debugging of `pvmd` and `pvm3_ad` for `pgmake`.

3.2 Agent-Daemon: `pvm3_ad`

The `pvm3_ad` sits between the remote `pvmd`'s and the actual jobs that are executing to provide the appropriate execution environment. The paradoxical name "Agent-Daemon" reflects its dual role in the `pgmake` system. `Pvm3_ad` serves as an *agent*: it assists the remote `pvmd` to fork a job, collect its status and output, and return it back to `pgmake`. Each time `pgmake` needs to start a job, it spawns a `pvm3_ad`, which in turn forks the actual job.

4 Evaluation

The pgmake system has been successfully implemented in SunOS 4.1.3 and used on a network of over thirty workstations. The code additions are highly portable and should introduce no porting problems to other operating systems that support both PVM and GNU `make`.

Goals. `Pgmake`'s main objective is to reduce the overall time to maintain groups of targets with *make*. The speed improvements must justify the extra complexity in setup and execution overhead. The following exit criteria were deemed necessary for `pgmake`'s acceptance as a viable tool:

- Low computational overhead in deciding to run a remote job.
- Low network overhead when shipping jobs to remote hosts.
- Low overhead in assembling status information obtained from remote hosts.
- Ability to quickly terminate remote jobs.

The following conditions are ideal for `pgmake` to maximize its effectiveness:

- A highly parallelizable execution hierarchy in the Makefile.
- A stable, low latency, network.
- A PVM configuration with as many reliable machines as possible.

Results. Measurements were performed with the goal of evaluating how well our design met these criteria. The overhead in deciding when to run jobs remotely is negligible. This consists of testing a boolean for each job, and a one time check to see if the local `pvmd` is running. By far, the most significant overhead of `pgmake` is shipping all the context information that is required to run a job remotely.

In the test cases (building `pgmake` with itself), using an arbitrarily large sized PVM, we observed a total of 10 seconds overhead in packing, shipping, and unpacking the context information. Note that 10 seconds is the aggregate overhead for shipping over 50 jobs to remote hosts: roughly one half second per job. In Figure 2 we see that the difference between running the entire compilation in a single thread remotely (labeled "1") and locally (labeled "LOCAL") is roughly 10 seconds. (The labels to the right and left of the plots indicate the size of the PVM that was used.)

This overhead is offset by adding just one more machine to the PVM. A PVM of size two or greater reduces the total compilation time by nearly one half. Increasing the size of the PVM to 15 nodes improves performance by 20% more. From these results, it is almost always worth parallelizing the compilation processes, even using relatively modest hardware.

4.1 Anomalies

The results obtained above and plotted in Figure 2 illustrate two interesting phenomena with non-obvious explanations:

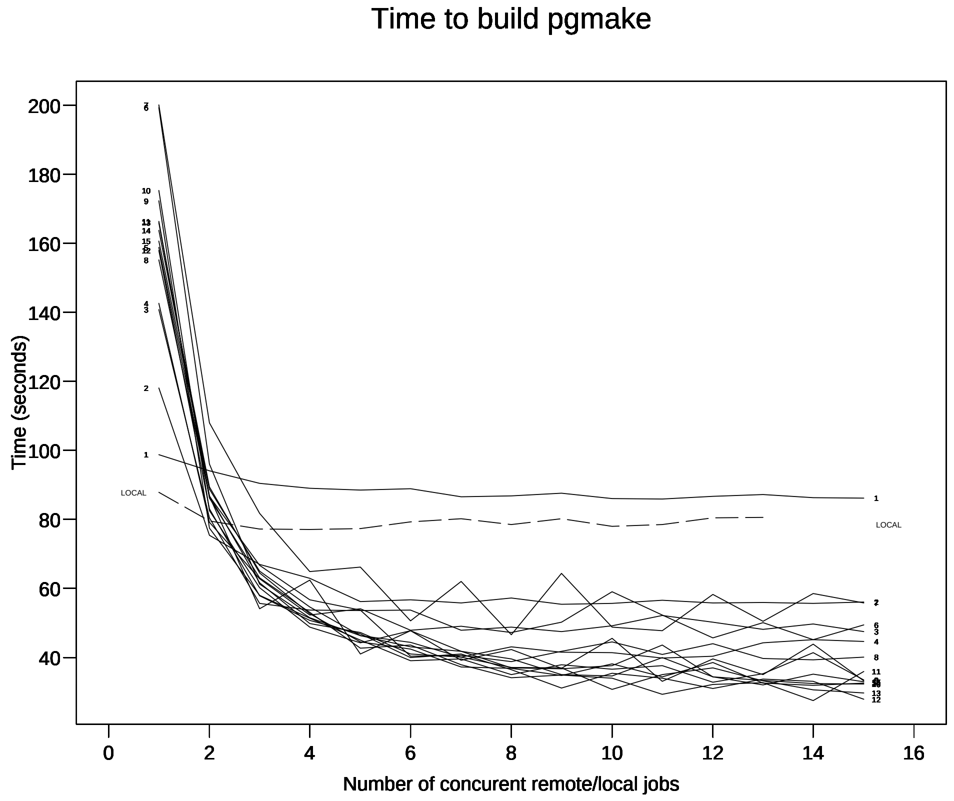

Figure 2: Times for a local and remote make vs. number of slots and size of PVM

"See-Saw" performance. Both figures show an unexplained improvement in performance when the number of concurrently running jobs is even, followed by a deterioration in performance for an odd number of parallel jobs. This may be a result of particular scheduling algorithms in the SunOS 4.1.3 operating system. This behavior needs to be investigated further.

Random behavior for PVM of size one. When we ran our tests using a `-j` value of 1, the results appeared to be random (left side of Figure 2.) There appears to be no pattern which would explain a PVM of size 6-7 machines taking twice as much to execute a single job as opposed to a PVM with one or two hosts.

We suspect that a combination of machine loads, PVM's scheduling and load-balancing algorithms, and network instabilities are at work here — but we would not be certain before we exercise more controlled experiments. Another theory which may explain these strange anomalies relates to the effects of executing commands on a machine with a *cold cache*. When processing a source file, many resources need to be dragged in to perform a compilation. In an test cases with one node, the first execution of a `make` command with a cold cache took over 60% longer than when the cache was warm. As the number of nodes in the virtual machine increases, and the job size remains one, the likelihood of spawning a task on a machine with a cold cache becomes greater. This may explain the increasing compilation times. More experiments and measurements are needed to better understand this phenomenon.

4.2 Potential Problems

NFS Bottleneck Given n remote processes, each of the processes still reads and writes to the same disk partition over NFS. This becomes a problem since most implementations of NFS are known to be lackluster in performance, and perform synchronous write commands.

Gateway and Router Concerns Also related to NFS, the performance of the virtual machine will significantly deteriorate as packets pass through more routers and gateways. It would be desirable to be able to predict what types of degradation to expect as the conditions get worse.

4.3 Experiences

GNU Make There is a general problem concerning the handling of standard input when performing parallel compilation. With multiple children and one source of standard input, only one process is allowed to have access to standard input, while the others are given a bogus, broken pipe. Therefore, GNU `make` advises users of the `-j` option not to depend on using standard input at all. In `pgmake`, we make no attempt whatsoever to give standard input to any process. Since no process should expect standard input to be valid, we do not give a valid standard input to any spawned process.

PVM PVM has performed respectably, but can give some unexpected results. For example, it is possible to execute the `pvm_spawn()` call on a machine which is legally enrolled in PVM, but simply does not have enough resources to perform the task. In this case, it returns a negative return code but gives no indication *which* node failed.

Because of this problem, we introduced a *retry loop* which attempts to respawn a failed job a specified number of times before giving up. After introducing this loop, we found our setup to be much more tolerant of bad nodes in the system.

5 Future Work

The initial version of `pgmake` serves as a proof-of-concept implementation. We are actively exploring the following areas to improve the performance and robustness of the system.

- **Distribution and load-balancing.** PVM uses a simple round-robin load-balancing scheme when scheduling tasks. It would be desirable to make more intelligent choices concerning load-balancing, taking into account the actual load on the PVM nodes.
- **Improved node metrics.** PVM maintains a *relative processor speed* number for each node, but does not seem to set it to an interesting value, nor use it for any reason. It would be useful to use this number to indicate the relative speeds of certain machines. A metric is needed to be able to distinguish between multi-processor machines from uni-processor ones.
- **Investigate see-saw behavior in performance.** We would like obtain more data samples to confirm our theories on how increasing job sizes compare to the number of processors. We would also like to investigate the seemingly "random" behavior for `make -j 1` when the PVM size changes.
- **Real-time Handling of `stdout` and `stderr`.** In the current implementation, `stdout` and `stderr` are buffered up into files and only sent back to the initiating process when the spawned task has completed. We would like to implement `pvm3_ad` to connects to its child process via two pipes, and send bursts of `stdout` and `stderr` messages back to the parent.
- **Cross Compilation.** The GNU development suite of compiler, assembler and linker provide a way to cross-compile binaries across dissimilar architectures. Combining `pgmake` with cross-compilation can provide significant performance boosts in generating binaries for architectures that are either slower in computational speed or smaller in number.

6 References

[1] E. H. Baalbergen The Design and Implementation of Parallel Make *USENIX Computing Systems*, Spring 1988, Volume 1, Number 2, pages 135-158.

[2] A. de Boor *Customs - A Load Balancing System* U.C. Berkeley, Computer Science 262, Final Project Paper, Fall 1987.

[3] A. de Boor *Pmake - A Parallel Make* U.C. Berkeley, Computer Science 262, Final Project Paper, Fall 1987.

[4] S. Feldman. *Make - A Program for Maintaining Computer Programs.* Software-Practice and Experience, pages 255-265, April 1979.

[5] G. Fowler. *The Fourth Generation Make.* Proceedings of the 1985 Summer USENIX Conference, pages 159-174, June 1985.

[6] D. Mills *Network Time Protocol (Version 3) Specification, Implementation and Analysis.* RFC 1305, IETF Network Working Group, March 1992.

[7] J. Pendry and N. Williams. *Amd - The 4.4 BSD Automounter.* Imperial College of Science, Technology, and Medicine, London, 5.3 alpha edition, March 1991.

[8] E. S. Roberts and J. R. Ellis Parmake and Dp: Experience with a distributed, parallel implementation of make *Proceedings 2nd Workshop on Large-Grained Parallelism*, pages 74-76, Carnegie Mellon University, November 1987.

[9] R. Sandberg et al. Design and Implementation of the Sun Network Filesystem. In *Proc. 1985 Summer USENIX Conf.*, pages 119–130, June 1985.

[10] R. M. Stallman and R. McGrath *GNU Make: A Program for Directing Recompilation* Free Software Foundation, Version 3.63, January 1993.

[11] V. S. Sunderam *PVM: A Framework for Parallel Distributed Computing* Concurrency: Practice & Experience, pages 315-339, December 1990.