Design and Implementation of Parallel Make

Erik H. Baalbergen

Dept. of Mathematics and Computer Science
Vrije Universiteit
Amsterdam
The Netherlands

ABSTRACT

Make is the standard UNIX† utility for maintaining programs. UNIX programmers have been using it for almost 10 years, and many UNIX programs nowadays are maintained by it. The strength of *make* is that it allows the user to specify how to compile program components, and that the system, after an update, is regenerated according to the specification and with minimum number of recompilations. With the appearance of multiple processor systems, we expect the time needed to ‘‘make’’ a program, or *target*, can effectively be reduced. Although the hardware provides parallelism, few tools are able to exploit this parallelism. The introduction of parallelism to *make* is the subject of this paper. We describe a parallel *make* and give an analysis of its performance.

1. INTRODUCTION

Large programs are often written as a collection of small files rather than as one big file so that changes to one source file only require recompilation of that one file. A large C [1] program, for example, may be split up over tens, or even hundreds of files. Code which is common to a set of source files is often placed in a single file, which is included (using a C Preprocessor ‘‘#include’’ line) in each of the source files. A consequence of the inclusion of source code is that if we change an included file, we have to recompile all files that include the file. Instead of recompiling all program components, we limit ourselves to recompiling the affected source code only. This efficiency, however, imposes a strict discipline on the programmer, who has to remember which files *depend on* (i.e. ‘‘include’’) other files, and which commands are used to regenerate components.

Make [2] is a program that keeps track of which object files are up to date and which must be regenerated by compiling their corresponding sources. Apart from the concept of efficiently regenerating components, we can achieve more efficiency by speeding up the commands that *make* executes, and adapting *make* itself. With the advent of parallel processor systems, it has become possible to speed up the operation of *make* by doing compilations in parallel. Parallelizing *make*, however, is not at all straightforward; there are numerous problems and pitfalls. These problems, their solutions, and various

† UNIX is a trademark of Bell Laboratories.

optimizations form the body of this paper. We also discuss the performance of our parallel *make* and compare it to work elsewhere.

2. SPEEDING UP THE MAKE PROCESS

Apart from parallelism, there are several methods in decreasing *make*'s response time. On the one hand, it is possible to speed up single compilations, which does not affect *make* itself. On the other hand, we can optimize *make*, by, for example, eliminating drivers, optimizing command invocation and task scheduling, and compiling description files.

2.1. Concurrent compilations

Much work has been done in the area of *concurrent* compilation. A short overview of the work is given in [3]. Strategies for concurrently compiling a single program are, among others, *pipelining* [4, 5, 6], *source-code splitting* [3], and *parallel evaluation of attribute grammars* [7]. The main purpose of running a single compilation concurrently is to decrease the response time. We can indeed speed up the *make* process by invoking concurrent compilers, but, in practice, few concurrent compilers exist yet. Moreover, *make* is routinely used to invoke − besides compilers − generators (e.g., *yacc* [8] and *lex* [9]), linkers, text formatters, and many other tools. To effectively exploit parallelism inside tools, we need concurrent tools, which still are rare. There are, however, few compilers available, which are able to do (part of) the work concurrently. A compilation in *ACK* [5], for example, is done by passing the code through several compiler components. *Make* can exploit pipelining techniques by efficiently scheduling the compiler components among the available processing power.

2.2. Eliminate Compiler Drivers

One possible improvement to *make*'s response time is to replace compiler ‘‘drivers,’’ such as *cc*, by supplying rules to call the compiler phases explicitly. Instead of having the rule

```
.c.o:
	$(CC) $(CFLAGS) −c $*.c
```

we can introduce the rules

```
.c.i:
	/lib/cpp $(CFLAGS) $*.c >$*.i
.i.s:
	/lib/ccom $(CFLAGS) $*.i $*.s
.s.o:
	/bin/as $*.s $*.o
```

This is not normally done since the traditional *make* is unable to transitively use implicit rules. More important, compiler drivers introduce some means of compatibility, since, for example, calling *cc* is portable among UNIX systems, while the invocation of the compiler phases might differ.

2.3. Optimize Command Invocation

A considerable speed-up, which is already available (or easy to implement) in existing *make*s, is optimizing the invocation of commands by shunting out the shell whenever possible. If a command line contains no shell-specific constructs (such as **;**, **&&**, **&**, **(**, **[**, **$**, etc.) and commands (*cd*, *for*, *if*, *case*, etc.), *make* can easily place the arguments in an argument array and invoke *execve* itself, instead of calling a shell to parse and execute the command line. Another strategy is to have a shell running as co-process to *make*, and write command lines to the shell via a pipe. The latter approach was adopted in *nmake* [10].

2.4. Compile Description Files

Compiling description files instead of interpreting them introduces some additional speed-up. Instead of reading and parsing the description file at each invocation, *make* compiles it into a binary format, and uses the binary description file in subsequent invocations. The binary description file may consist of *make*'s internal data structures, built when parsing the description file. *Make* has to recompile the description file, if it is more recent than its compiled binary version. The method of compiling description files is used in, for example, *nmake*.

Another strategy is to translate the description file into a compilable language, such as C, and compile the resulting program. Instead of calling *make* to update a system, we, or the *make* command itself, can invoke the generated program. Both compiled description files and generated *make* programs offer a constant decrease in response time since we only reduce the *make* overhead. However, since most of the execution time comes from executing the commands, rather than from *make* interpreting the rules, the gain here is small.

2.5. Exploit Parallelism

If multiple processors are available to execute commands for *make*, a potentially large speed-up is possible by running commands in parallel. We can get an optimum speed-up if we schedule the commands cleverly. Consider the rule

```
prog: main.o util.o prog.o
```

Assume that each of the compilations of *main.c* and *util.c* into *main.o* and *util.o*, respectively, takes *t* seconds, and the compilation of *prog.c* into *prog.o* takes *2t* seconds. If there are 2 processors available, *main.o* and *util.o* are made in parallel. The compilation of *prog.o* is postponed until one of the processors finishes, which is (at least) at time *t*. From this moment on, it still takes *2t* seconds to make *prog.o*. The *make* process totally takes *3t* seconds. If, on the other hand, we let one processor make *prog.c*, and let the other processor make *main.o* and *util.o*, as if the rule were

```
prog: prog.o main.o util.o
```

then the make only takes *2t* seconds.

The problem of running *n* independent tasks, with known execution times, on *m* similar processors with minimal response, or *finishing*, time, is studied by the theory of deterministic sequencing and scheduling [11], and is known as the $P \mid\mid C_{\max}$ problem. Since the problem is NP-complete, we are dependent on heuristics. [12] gives an easy-to-read introduction into various techniques involved with optimizing scheduling algorithms.

Known heuristics for a feasible schedule with minimum finishing time, often called the *independent task-scheduling problem*, are the *list-scheduling* algorithm, the *LPT (Largest Processing Time)* algorithm, and the *MULTIFIT* algorithm. List scheduling treats the tasks in the given order, and assigns each task in turn to a processor which is free if there are free processors, or which is the first one to finish a task if all processors are busy. A better scheduling, with lower finishing time, is *LPT*, which sorts the given jobs according to decreasing execution time, and applies list scheduling to the reordered list of jobs. *MULTIFIT* [13] is the best scheduling algorithm (for our model) yet found. The idea is to find a minimum value for the deadline, the time all tasks need to have finished. The tasks are again assigned in decreasing order of execution time, each to the lowest indexed processor, if the total execution time on that processor does not exceed the deadline. Otherwise, the next processor is taken. (This strategy is called the *first fit decreasing method*.) For any values of the deadline, the division among the processors either succeeds or fails, according to which we adapt the deadline. *MULTIFIT* uses binary search to find a minimum value for the deadline.

Should *make* itself reorder the dependency list, or does the user have to specify the order himself? The former approach requires *make* to have knowledge of the time needed to execute a command block, whereas the latter forces the user to estimate times, which may differ among various environments. It is difficult to come up with a set of heuristics for *make* to estimate the execution time (such as ‘‘compilation time is proportional to the length of the source’’), since *make* has no idea of what a command does. One solution is to keep track of execution times of command blocks in *make* runs, and use the results in future *make*s. Possible implementations are to let *make* keep the timing results in a global state file, or to enable *make* to reorder the dependency lists and overwrite a description file, or output a new description file.

3. PARALLELISM AND DISTRIBUTION

Before discussing the techniques and problems of implementing a parallel *make*, we consider the relation between parallelism and distribution. Experiments [14] have shown that running several compilations in parallel on a single processor in general does not result in a significant speed-up, since compilations are usually CPU-bound. At best, while one process is doing I/O, another one can compute.

Several practical problems arise when running compilations in parallel on a single-processor UNIX system. First, processes compete for a fixed number of CPU cycles. The more processes there are, the fewer cycles each one gets. Moreover, time needed for swapping or paging increases as the number of processes grows. The net result is that processes running in parallel slow down each other. Second, each UNIX user is allowed to have only a limited number (commonly 20 or 25) of processes running at a time. This limit reduces the number of simultaneous compilations because each one may need several processes.

If multiple processors are available, we can achieve a speed-up by running each compilation on a different processor. The trick is to arrange for this parallelism without burdening the programmer with all the details. Several approaches are discussed in [14]. In what follows we will assume that the mechanics of forking off processes to remote CPUs is handled by the operating system. Our concern is *what* should be run in parallel, not *how* parallel execution is achieved. We assume that the underlying operating system has a smart processor allocation strategy; that multiple processors (say, at least 8) are available; and that commands can be executed on any processor in the network without loosing efficiency. A distributed operating system that serves our needs is *Amoeba* [15, 16, 17].

4. PARALLELIZING DESCRIPTION FILES

In this section we consider various naive approaches in making *make* run commands in parallel by adapting description files. Apart from the conclusion that correct parallelization of *make* is not achieved by naively altering the description files, and that it is impossible to maintain compatibility with existing *make*s and description files, there are a few basic problems we have to solve in designing a parallel *make*.

A first approach in making *make* run commands in parallel is enveloping each command line in parentheses, and append an ampersand to it. The shell runs the command line in the background, and returns immediately. This simple and naive approach will not work in practice due to the following problems:

[1] The commands in a command block must execute in sequence, since a command may use the result of a previous command within the same command block.

[2] Starting a job in the background via a shell causes the shell to return immediately, and *make* to believe that the command has finished. Moreover, *make* has no facility to wait for the background process to finish, nor can it tell whether the command succeeded. It may decide to activate a rule's command block, while its dependencies still are absent or out of date.

[3] *Make* does not keep track of how many child processes are still alive. The system may refuse to execute commands due to the UNIX per-user process quantity. Letting *make* continuously try to fork off a process after a failure does not solve the problem either. If, for example, *cc* is unable to fork off a pass, it does not try again; it just reports back failure.

[4] There are commands which cannot run in parallel with each other, because they use fixed file names. *Yacc* is a standard example.

Executing the commands in a command block sequentially, and the command block as a whole in the background, solves problem [1]. This is achieved by surrounding the command block by parentheses, appending an ampersand to the closing parenthesis, and using a single shell to execute the command block as a whole. Problems [2], [3] and [4], however, still exist. Worse yet, the latter mechanism introduces another problem:

[5] Traditional *make* executes each command line of a command block in a separate shell. This means that a *cd*, ‘‘change directory’’, command has effect only within a single command line, not in the succeeding commands within the same block. To get the same behavior if the command block is executed in a single shell, we have to adapt the command block in the description file.

Problems [1] and [5] are solved by the mechanism to surround each command line by parentheses (i.e., execute it in a separate shell); place parentheses around the command block as a whole; and append an ampersand to it. This still leaves problems [2], [3] and [4] unsolved, and introduces a lot more, almost dummy, shells.

The next section discusses an approach which solves the five problems.

5. DESIGN OF PARALLEL MAKE

In order to design and develop our parallel *make*, which we have called *pmake*, we consider a few issues to make it easy to use.

5.1. Design Goals

An important issue is to maintain upwards compatibility; existing description files still should be accepted and interpreted properly, and *pmake* description files should be accepted by *make*. A second important issue is to hide the parallelism completely from the description file writer. Programmers and *pmake* invokers should not be confronted with the use of complicated constructs to exploit parallelism. Unfortunately, there are several serious obstacles which prevent our goals of compatibility and transparency from being achieved completely in parallel *make*. The problems and possible solutions are discussed in the remainder of this section.

5.2. Virtual Processors

To overcome the problems discussed in the previous section, we introduce virtual processors as the basis of *parallel make*. For each command block to be executed, *pmake* creates a child process, which we call a *virtual processor*. *Pmake* does not wait for the virtual processor to finish, but continues processing the list of dependencies in which the target appeared, which caused the command block's execution. The virtual processor controls the execution of the command lines in the command block. The strategy is depicted globally in Figure 1.

```
make(target):
   let R be the rule
      target : dependency-list
         command-block
   for each dependent in dependency-list do make(dependent)
   if all makes succeed and any dependent is newer than target then
      allocate virtual processor (i.e. check number of child processes)
      fork child: (* this is the virtual processor *)
          report execute(command_block)
   return (* do not wait for the virtual processor to finish *)

execute(command_block):
  for each command in command_block do
     execute command via a shell
     wait for the shell to finish
     if command failed then report false
  report true
```

Figure 1.

The algorithm shows that synchronization among the update commands is driven by the dependency graph.

A virtual processor runs the command lines one after another, each in a separate shell environment. This mechanism solves problems [1] and [5]. As soon as one of the commands fails, the virtual processor exits with status *false*. If all commands succeed, then the virtual processor stops with status *true*.

When the target, which is the command block's result, is needed, (i.e., when all dependencies of the parent rule are checked), *pmake* waits for all virtual processors dealing with the dependencies to finish, before deciding whether to make the parent target. This strategy solves the synchronization problem [2].

To avoid problem [3], the number of available virtual processors within a single *pmake* run is limited. If *pmake* has to allocate a virtual processor while the maximum number of virtual processors is running, it has to wait until at least one of them is ready. The number of available virtual processors must be carefully chosen, since each one can also fork off processes. The user can explicitly specify how many virtual processors may be used by using the **–Pn***num* command-line option. Otherwise, a default number is chosen, and it is up to the system administrator to select an efficient number, depending on the number of available physical processors. The virtual processor mechanism introduces a minor overhead because an extra process is introduced for each virtual processor to control the execution of the command block.

It is not possible to solve problem [4] in a transparent way, since *pmake* cannot deduce which commands may not run in parallel in the same directory. *Concurrent make*, or *cmake* [18], provides a facility to specify which command blocks should execute mutually exclusively, i.e., only one command block in a certain group of command blocks may run at a time. To do so, one has to define a *mutex* on the group of command blocks, by declaring a rule having target name *.MUTEX*, with the targets of the command blocks as dependents. The *lex* example becomes

```
prog: a.o b.o
.MUTEX: a.c b.c
a.c: a.l
        lex a.l
        sed 's/yy/ayy/g' lex.yy.c > a.c
b.c: b.l
        lex b.l
        sed 's/yy/byy/g' lex.yy.c > b.c
```

To force *cmake* to run only one command at a time (i.e., to behave like *make*), the description file should contain the rule

```
.MUTEX:
```

We adapted the *.MUTEX* mechanism in *pmake*. Although it is certainly not transparent to the user, it is a convenient way to solve problem [4]. In practice, there is hardly ever a need to define a mutex.

Note that a description file with a mutex is still compatible with the traditional *make*.

5.2.1. Command Failures

Since the dependents of a target are checked and possibly updated in parallel, we cannot prevent a dependent from being made if one of its predecessors in the dependency list has failed. This behavior is similar to *make* with the ‘‘–k’’ flag, which indicates that the *make* process should continue even if a command fails. One way to get rid of the ‘‘–k’’ behavior is to stop checking the dependency list as soon as possible. This leaves us in a situation where a nondeterministic number of dependencies has already been made. To prevent the nondeterminism as much as possible, we do not examine the success or failure of making a dependent until we need the results; this results in ‘‘–k’’ behavior.

5.2.2. Multiple-target Rules

Make lacks a means of specifying that a command block produces multiple files. It seems, however, able to deal with multiple-output-files commands, through nothing but a coincidence. We erroneously take the rule

```
y.tab.c y.tab.h: grammar.y
        yacc grammar.y
```

as a specification of how to produce both *y.tab.c* and *y.tab.h*, using a single *yacc* command. *Make*, however, considers the rule to be a short-hand specification of

```
y.tab.c: grammar.y
        yacc grammar.y
y.tab.h: grammar.y
        yacc grammar.y
```

In practice, we can hardly tell the difference, since if one of *y.tab.c* and *y.tab.h* appears as dependent and is updated, the other one is created too, at the same time. If the latter file appears as dependency, then the file already exists and *make* decides not to create or update the file. Worse, programmers sometimes tacitly assume that *y.tab.h* is created when using the rule

```
y.tab.c:  grammar.y
        yacc grammar.y
```

which indeed produces *y.tab.h*, although *make* does not note that it has been created. The programmer must take care that *y.tab.c* is created before any other file is made, that depends on *y.tab.h*. In *pmake*, however, neither the ‘‘*y.tab.c y.tab.h*’’ rule nor the ‘‘*y.tab.c*’’ rule will always work correctly. For example, the construct

```
prog: y.tab.o lex.o
        $(CC) –o prog y.tab.o lex.o
lex.o: y.tab.h
y.tab.o: y.tab.c
y.tab.c y.tab.h: parse.y
        yacc parse.y
```

is treated correctly by any sequential *make*. *Pmake*, however, tries to ‘‘make’’ *lex.o* and *y.tab.o* in parallel, independently from each other. Both actions require the last rule to be applied, resulting in two *yacc* processes running in parallel, both writing output to the same files, *y.tab.c* and *y.tab.h*. As soon as one of the commands has finished, the result is used in creating *lex.o* or *y.tab.o*, while the other *yacc* process still may be busy in writing output to *y.tab.c* and *y.tab.h*, which files are supposed to be complete. Omitting *y.tab.h* from the last rule's target list is correctly dealt with in sequential *make*, but *pmake* may complain about *y.tab.h*'s absence. *Pmake* does not know that *y.tab.h* is created as side effect in creating *y.tab.c*. Therefore, we have to take care that at least each file that is created appears as target. Furthermore, to prevent simultaneous execution of the *yacc* commands, we can define a mutex on *y.tab.c* and *y.tab.h*, or, even better, on *lex.o* and *y.tab.o*. The former mutex does not prevent *yacc* being invoked twice, while the latter does.

A better solution, without changing *make*, is to introduce an *intermediate* target, which is an empty file, and which is newer than the other files created by the commands in the command block. The following code shows the use of an intermediate target, called *yacc_done*.

```
prog: y.tab.o lex.o
        $(CC) –o prog y.tab.o lex.o
lex.o: y.tab.h
y.tab.o: y.tab.c
y.tab.h y.tab.c: yacc_done
yacc_done: parse.y
        yacc parse.y
        touch yacc_done
```

The *touch* command creates or updates file *yacc_done*, and ensures that it becomes more recent than any of the files produced by the preceding *yacc* command.

6. IMPLEMENTATION OF PARALLEL MAKE

Instead of building a parallel *make* from scratch in order to get experience with the use of *parallel make*, we developed a module which implements parallelism and which can be plugged into traditional *make*s. The module uses the command block (i.e. the sequence of commands belonging to a rule) as the basic unit of execution. The idea is to collect the commands of a command block and keep administration of the target which causes the rule to be selected. As soon as the last command of a command block is encountered, the *make* process forks off a virtual processor, which executes the commands in sequence.

We equipped the UNIX System V Release 2 *Make* [19] with the parallel module, and ran the result on the *Amoeba* distributed operating system. The configuration consisted of a processor pool, a disk, and a terminal (actually a window on a SUN), connected by a 10 Mbps Ethernet.

6.1. Implementation Problems in *Pmake*

Apart from implementation problems such as the absence of explicit dependencies in the internal data structure of implicit rules, signal handling (signal propagation to virtual processors), and synchronization (the modification time of a target which is being created), there is the problem of multiplexing diagnostic output.

The diagnostic output from the commands is multiplexed arbitrarily, leaving the user with an incoherent collection of messages or, even worse, characters. If the commands produce output on standard output, and especially if the characters are not buffered, the resulting list of diagnostics may look messy. One solution is to gather output from commands via pipes and present them to the user, or an ‘‘intelligent’’ editor (e.g., *emacs*), in an ordered manner. Since *make* may read several directories at a time, using file descriptors, we have to be careful with using file descriptors for pipes. A better approach is to redirect the output from commands onto files, and present the contents of the files, preceded with an indication of the source.

6.2. Problems in the Test Environment

Amoeba is still under development and undergoing changes. We made use of a beta version of *Amoeba*, which suffers from various problems. *Amoeba* currently lacks load balancing among the processors in the pool. Processes are assigned to physical processors at random. The chance that some processors are overloaded while other processors are idle is nonnegligible. The random selection takes place if either a new program is loaded into memory using the *exec* system call, or if a process forks itself, using *fork*, and the current processor does not have enough space for creating a copy of the process.

Another problem is the inefficient use of memory, because *Amoeba* currently lacks a shared text facility. Forking a process causes the kernel to copy both text and data space in memory, which results in both time-expensive forks and waste of memory. There is only need to copy data space when forking in shared text environments.

We take care of the influence of ‘‘random’’ processor assignment under *Amoeba* by taking the minimum of the experimental results; we believe that load balancing in *Amoeba* will eventually result in a better distribution of processes among physical processors.

6.3. Timing Results

The timing test consists of compiling a large set (several tens) of C source files into corresponding object code, using the *ACK* [5] C compiler running on *Amoeba*. We use the real time, measured under perfect conditions, which means single user, no background processes. Furthermore, we deliberately do not include the link *ld* phase in the measurements, since it cannot be done in parallel. Finally, we manually reordered the dependency lists according to the *LPT* algorithm.

Table 1 shows the speed-up factor acquired when using the indicated number of virtual processors. During the test, *Amoeba* ran on a pool of ten 68020 processors running at 16 MHz with a file server and a directory server.

number of virtual processors	*speed-up*
1	1.00
2	1.85
3	2.53
4	3.18
5	3.60
6	3.74
7	3.97
8	3.86
9	3.74
10	3.81

Table 1

The table shows that the speed-up is far below linear. This is not surprising since we have to deal with some overhead in starting a compilation. *Pmake runs* its compilations in parallel but has to *start* them sequentially. Consider $H+C$ the response of time of a compilation, where H indicates the time needed to start the compilation (i.e. to fork off the compiler driver), and C the time needed for executing the compilation. If *pmake* has to run N compilations, we cannot expect it to fork them off simultaneously. In the ideal case, when processors are available when needed, the N-th compilation is started after $N-1$ compilations have been started. *Pmake*'s minimum response time then becomes $N*H+C$, which is much more than the response time with linear speed-up $H+C$. If the number of compilations exceeds the number of processors, then *pmake* has to wait from time to time until a processor becomes available, in which case the total response time increases and the speed-up decreases.

7. COMPARISON WITH OTHER PARALLEL MAKES AND DISCUSSION

This section gives a short overview of how several makes supply parallelism. Besides compatibility and transparency issues, we compare each *make* with *pmake*.

7.1. Nmake

Nmake [10], or *New Make* or *4-th generation make*, was developed at AT&T Bell Laboratories. *Nmake* executes the update commands by sending the command blocks to the shell *sh*, which runs as co-process. As a consequence of sending complete command blocks to the shell, a *cd* command, or the introduction of a shell variable, remains effective during execution of the command block, as opposed to Feldman's *make* and *pmake*. If the user takes no special action, each command is executed by the shell, denoted as the *foreground* shell. If, however, the user specifies that update commands may execute in parallel, the foreground shell starts a subshell, called the *background* shell, for each update command block to be run. In *pmake*, commands are always executed in a background shell. Parallelism in *nmake* is activated by specifying **−j***n* (or **−j**) as command line option, which means that up to *n* (default 3) background shells may be active at a time. Like in *pmake*, the dependency graph is used for synchronizing the jobs.

Specifying *.FOREGROUND*, or its synonym *.WAIT*, in the dependency list of a target, causes the update command block of the target to execute in the foreground shell, which in turn causes *nmake* to block until the commands have finished.

Nmake does not provide an explicit facility to prevent certain commands from being executed mutually exclusively. It is possible, however, to run the critical commands in the foreground shell, thus imposing sequential execution.

7.2. Concurrent Make

Concurrent Make [18], or *cmake*, was developed primarily to reduce the time needed to run a *make* process, not to increase *make*'s functionality. It is written in *Concurrent C* [20], and is based on the Version 8 UNIX *make*. The latter requires the user to indicate explicitly which commands can execute in parallel, whereas *cmake* runs the update commands in parallel by default. Furthermore, the explicit indication in Version 8 *make* introduces a non-portable construct in a description file.

The main difference between *cmake* and other parallel *makes*, including *pmake* and *Version 8 make*, is that *cmake* takes care of distribution among several processors. The major assumption is that executing a command remotely has the same result as executing it locally. The rule

.LOCAL: [*target* ...]

forces *cmake* to run the update commands for the targets appearing in its dependency list on the local machine. An empty dependency list in a *.LOCAL* rule forces *cmake* to run all update commands locally. *Pmake* and other parallel *make*s do not take distribution into account. They silently assume that the underlying operating system provides efficient parallel execution among the available processors.

To prevent certain command blocks from executing in parallel, the rule

.MUTEX: [*target* ...]

causes the update commands of the targets, which appear as dependency, to execute mutually exclusively. Parallelism can be suppressed by specifying a *.MUTEX* rule with an empty dependency list. The *.MUTEX* mechanism has also been adopted in *pmake*.

Cmake description files, like that of *pmake*, are silently accepted by *Version 7*-compatible *make*s. The compatibility results from the use of *make*'s syntax to specify parallel constructs and options. *Make* interprets, for example, a *.MUTEX* rule as a rule to be applied when creating a target *.MUTEX*, instead of a special command.

7.3. Mk

Mk [21], like *nmake*, is an enhanced version of the original *make*. The number of jobs run in parallel is user-settable by defining the macro *$NPROC*. The number of concurrent jobs is 1 by default, which implies serial execution of the commands. Unlike *nmake*, *cmake* and *pmake*, *mk* has no provision for the mutual exclusive execution of commands, although commands can be executed one after another, using serial execution. To use parallelism and to deal with, for example, several *yacc* commands in a single *mk* run, the programmer has to take care of name clashes explicitly. An implicit rule for creating a linkable object file out of a *yacc* specification file is

```
%.o: %.y
        mkdir /tmp/$nproc; cp $stem.y /tmp/$nproc
        (cd /tmp/$nproc; yacc $stem.y; mv y.tab.c $stem.c)
        $CC $CFLAGS −c /tmp/$nproc/$stem.c
          rm −rf /tmp/$nproc
```

Although the implicit rule is artificial, there is now no need to prevent several *yacc* commands to run in parallel with each other.

7.4. Parmake

Parmake [22] is an extension of the traditional *make*, and provides concurrent execution of the operations which have no mutual dependencies. *Parmake* has been implemented at DEC's System Research Center on a local area network of shared-memory multiprocessor *Firefly* workstations. The processing power of idle workstations is supplied by a distant process facility *dp*. *Parmake* itself orders independent jobs by topologically sorting the dependency graph in the description file. The description file is compatible with the traditional *make*, although a syntactic mechanism is introduced to force left-to-right evaluation of the dependencies.

A set of heuristics, controlled by parameters which reflect the relative cost of the operations, is used to balance the local load, while *dp* schedules the distant processes, based on machine-load statistics.

Experiments in using *parmake* in recompiling a large set of Modula-2+ files have shown a maximum speed-up of 2.2, using the 5 local processors only, and 13.5, using 20 concurrent local and distant processes. An important observation is that the speed-up strongly depends on the nature of the jobs being executed; the performance advantage increases along with the ratio of computation to I/O. A much smaller and faster compiler (for example, a C compiler) turned out to be limited by the disk speed. Using 20 or more local and distant processes showed a speed-up of only 5.8.

7.5. DYNIX Make

DYNIX Make [23] provides a mechanism to activate parallelism explicitly. If the string which separates a target from its dependencies is ‘‘:&’’ or ‘‘::&’’, then the command blocks to make the dependents can execute simultaneously. If two dependents are separated by ‘‘&’’, those two can be created in parallel. The rule

```
target : dep1 dep2 dep3
```

causes *make* to update dep1, dep2, and dep3 sequentially. The rule

```
target :& dep1 dep2 dep3
```

implies that dep1, dep2 and dep3 may be updated in parallel to each other. The construct

```
target : dep1 & dep2 dep3
```

updates dep1 and dep2 in parallel, and then updates dep3. The number of simultaneously active commands is controlled by the **–P***num* command-line argument. By default, three commands can run in parallel.

Only the last command line of a multi-line command block is eligible for asynchronous execution, while the other commands are executed sequentially. In contrast with *pmake*, we have to combine multiple command lines into a single command line explicitly (by appending backslashes to all but the last lines), to force the command block as a whole being run asynchronously.

To preserve compatibility with *make*s in other systems, variable expansion is done before the parsing for parallel constructs. This allows rules to be written as

```
target :$(PAR) dep1 dep2 dep3
```

which is accepted by any *make* if $(PAR) is not defined, and which is accepted and interpreted as parallel construct if we invoke *DYNIX Make* by specifying

```
make "PAR='&'"
```

DYNIX make uses the technique of preceding the error messages from asynchronously executing commands with process identifiers, thus enabling the programmer to find the source of errors.

8. Conclusions

We believe that *pmake* satisfies our requirements in that it shows a considerable speed-up, its description files are compatible with Feldman's *make*, and parallelism and the problems that come with parallelism are almost transparent to the user.

The experiments have shown a considerable speed-up in using *pmake* on a multiprocessor environment. We believe that it is hardly possible to achieve linear speed-up even when we disregard the shortcomings in our test environment as discussed in section 6.2. First, bookkeeping and the virtual processor mechanism introduce a minor overhead in *pmake*. Second, it is hard to determine an optimal distribution of tasks among the virtual processors. Applying the *LPT* algorithm, discussed in section 2.4, might help but requires external information. *Pmake* does not implement the *LPT* algorithm, but we observed a minor speed-up when we manually reordered dependency lists. Third, we have to deal with ''bottle necks'' in practice. Compiling a multi-source-file program, for example, requires a link phase, which does not run in parallel with any of the compiling phases, although it could be made to run incrementally (so it would finish soon after the last object came in.) Fourth, if any of the available processors are multiprogrammed, we have to deal with processes competing for CPU cycles. The ideal situation is to have multiple monoprogrammed processors.

The syntactic compatibility with *make* description files is maintained. *Pmake* description files are accepted and interpreted correctly by *make*. *Make* description files, however, need revision if commands, like *lex* and *yacc*, should run mutually exclusively. Either we have to take care of non-clashing file names, by running commands in separate directories or by forcing the commands to use non-standard file names, or we need to define a *.MUTEX* on a group of commands. In practice, we have not yet encountered a legal *make* description file which is treated incorrectly by *pmake*.

Cmake and *parmake* description files, too, are compatible with *make* description files, but a major disadvantage is the explicit treatment of distribution. *Pmake* assumes the underlying operating systems supplies efficient parallel processing on multiple processors, while *cmake* has to distribute the commands itself.

9. Acknowledgements

I would like to thank Eduardo Krell, who gave me information about *nmake*; Andrew Hume, who sent me documentation on *mk*; Barton Schaefer, who supplied information on *DYNIX Make*; the *Amoeba* team at the Vrije Universiteit, who made the *Amoeba* system usable, and who had to suffer from my complaints about *Amoeba*'s shortcomings; and Andy Tanenbaum and Dick Grune for their suggestions, useful comments and critical readings of this paper.

10. REFERENCES

References

1. B.W. Kernighan and D.M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

2. S.I. Feldman, ‘‘Make—A Program for Maintaining Computer Programs,’’ *Software—Practice and Experience*, vol. 9, no. 4, pp. 255-265, April 1979.

3. V. Seshadri, I.S. Small, and D.B. Wortman, ‘‘Concurrent Compilation,’’ *Proc. IFIP Conference Distributed Processing*, Amsterdam, Netherlands, October 1987.

4. P. Robinson, W. Huen, O. El-Dessouki, E. Huske, and M. Evens, ‘‘A Pipelined DYNAMO Compiler,’’ *Proc. of the International Conference on Parallel Processing*, pp. 57-66, Cambridge University Computer Laboratory, Cambridge, UK, August 1977.

5. A.S. Tanenbaum, J.M. van Staveren, E.G. Keizer, and J.W. Stevenson, ‘‘A Practical Toolkit for Making Portable Compilers,’’ *Comm. of the ACM*, vol. 26, no. 9, pp. 654-660, September 1983.

6. J.A. Miller and R.J. LeBlanc, ‘‘Distributed compilation: a case study,’’ *Proc. 3th IEEE Int. Conference on Distributed Computing Systems*, pp. 548-553, October 1982.

7. H.J. Boehm and W. Zwaenepoel, ‘‘Parallel Attribute Grammar Evaluation,’’ internal report, Dept. of Computer Science, Rice University, Houston, Texas, October 1986.

8. S.C. Johnson, *Yacc: Yet Another Compiler-Compiler*, Bell Laboratories, Murray Hill, NJ, July 1978.

9. M.E. Lesk and E. Schmidt, *Lex—A lexical Analyzer Generator*, Bell Laboratories, Murray Hill, NJ, 1978.

10. G.S. Fowler, ‘‘The Fourth Generation Make,’’ *Proc. USENIX Summer Conference*, pp. 159-174, Portland, Oregon, June 1985.

11. E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnooy Kan, ‘‘Recent Developments in Deterministic Sequencing and Scheduling: a Survey,’’ in *Deterministic and Stochastic Scheduling*, ed. M.A.H. Dempster et al., Nato Advanced Study Series, Dordrecht, The Netherlands, July 1981.

12. M.R. Garey, R.L. Graham, and D.S. Johnson, ‘‘Performance Guarantees for Scheduling Algorithms,’’ *Operations Research*, vol. 26, no. 1, pp. 3-21, 1978.

13. E.G. Coffman, Jr., M.R. Garey, and D.S. Johnson, ‘‘An Application of Bin Packing to Multiprocessor scheduling,’’ *SIAM Journal on Computing*, vol. 7, no. 1, p. 1, February 1978.

14. E. H. Baalbergen, ‘‘Parallel and Distributed Compilations in Loosely-Coupled Systems: A Case Study,’’ *Proc. Workshop on Large Grain Parallelism*, Providence, RI, October 1986.

15. S. J. Mullender, ‘‘Principles of Distributed Operating System Design,’’ PhD. dissertation, SMC, Amsterdam, October 1985.

16. S. J. Mullender and A. S. Tanenbaum, ‘‘The Design of a Capability-Based Distributed Operating System,’’ *The Computer Journal*, vol. 29, no. 4, pp. 289-300, March 1986.

17. A. S. Tanenbaum, S. J. Mullender, and R. van Renesse, ‘‘Using Sparse Capabilities in a Distributed Operating System,’’ *Proc. of the 6th Int. Conf. on Distr. Computing Systems*, pp. 558-563, Cambridge, MA, May 1986.

18. B. Cmelik, ‘‘Concurrent Make: The Design and Implementation of a Distributed Program in Concurrent C,’’ in *Concurrent C Project*, AT&T Bell Laboratories, Murray

Hill, NJ, 1986.

19. AT&T, ‘‘A Program for Maintaining Computer Programs (Make),’’ in *System V Support Tools Guide*, pp. 11-40, June 1982.
20. N.H. Gehani and W.D. Roome, ‘‘Concurrent C,’’ *Software—Practice and Experience*, vol. 16, no. 9, pp. 821-844, September 1986.
21. A. Hume, ‘‘Mk: A Successor to Make,’’ Computing Science Technical Report No. 141, AT&T Bell Laboratories, Murray Hill, NJ, November 1987.
22. E.S. Roberts and J.R. Ellis, ‘‘Parmake and Dp: Experience with a Distributed, Parallel Implementation of make,’’ *Proc. 2nd Workshop on Large-Grained Parallelism*, pp. 74-76, Carnegie-Mellon University, Pittsburgh, Pennsylvania, November 1987. Available as Tech. Rep. CMU/SEI-87-SR-5
23. Sequent, ‘‘DYNIX Make Manual Page,’’ in *DYNIX Programmer’s Manual—Revision 1.15*, August 1987.