Makefile::Parallel Dependency Specification Language

Alberto Simões[**], Rúben Fonseca, and José João Almeida

Departamento de Informática
Universidade do Minho, Braga, Portugal,
{ambs|rubenfonseca|jj}@di.uminho.pt

Abstract. Some processes are not easy to be programmed from scratch for parallel machines (clusters), but can be easily split on simple steps. Makefile::Parallel is a tool which lets users specify how processes depend on each other.
The language syntax resembles the well known Makefile[1] format, but instead of specifying files or targets dependencies, Makefile::Parallel specifies processes (or jobs) dependencies.
The scheduler reads the specification and submits jobs to the cluster scheduler (in our case, Rocks PBS) waiting them to end. When each process finishes, dependencies are calculated and direct dependent jobs are submitted.
Makefile::Parallel language includes features to specify parametric rules, used to split and join processes dependencies: some tasks can be split into smaller jobs working on different portions of files, and at the end, another process can be used to join results.

1 Introduction

More and more, researchers have access to multi-processors machines and, as well, clusters. The problem is that most researchers do not have time to learn how to program for parallel machines. Thus, they use their usual programs on clusters taking advantage just on the processor speeds and the big amount of available memory.

We propose a tool, Makefile::Parallel, to specify how small processes (or programs) depend on each other, to create parallelism at the level of the program, instead of the usual parallelism at the instruction level. Makefile::Parallel main goals are:

- use a compact language to specify dependencies: in our main case study we are dealing with more than one hundred jobs. To specify their dependencies manually is time consuming and error prone;
- reuse a well known language syntax that is being used in related tasks for years: the Makefile syntax.

[**] Partially supported by grant POSI/PLP/43931/2001 from Fundação para a Ciência e Tecnologia (Portugal), co-financed by POSI.

– embed other languages to reuse their expressiveness. In our pmakefiles — the name we gave to the text specification files — we can define actions both in Bash and Perl.
– support parametric rules: in some situations we want to instantiate rules with different values, accordingly with results from a previous job. Thus, these rules need to be instantiated in runtime.
– create information for profiling pmakefiles and create reports.

Next section will present the Makefile::Parallel Domain Specific Language, showing the language grammar and explaining its versatility. Follows a section on the scheduler implementation and how it interacts with the main cluster scheduler. Before the final remarks, there is a section detailing our case study.

2 Makefile::Parallel Language

Makefile::Parallel language syntax is heavily inspired on Makefile's syntax. The main difference is that instead of defining dependencies between files or targets, we define dependencies between jobs. The main idea can be seen as the formalization of a PERT[2][1] network.

Figure 1 shows a simplified BNF version of the Makefile::Parallel grammar. The grammar was implemented in YAPP[3], a Perl version of the well known `yacc` parser generator.

To explain the language we will use the example shown on figure 2: suppose we have a program we want to test with a different parameter, an integer ranging from 3 to 10. We need to create a directory to save the data, and at the end we want to remove some temporary files.

To solve the problem presented above we defined three rules:

– There is a job to *prepare* the output directory. This same rule defines a variable to be used on the next rule, a parametric one, named *run* (see section 2.2 for detailed explanation on these rules). The variable is defined as a set of values that the variable p can have.
– The parametric rule `run$p` will instantiate with values from 3 to 10, and run the program being tested with different parameters.
– Finally, the last rule `cleanup` depends on the *run* set of rules, and cleanups some temporary files.

Values between parenthesis indicate the expected walltime. This time is used by ROCKS[4] to schedule the job in an adequate queue, and to kill the job in case it is taking too long.

Following the walltime (a required parameter) there is an optional parameter in square brackets: the number of CPUs needed for the job. Because we are parallelizing processes, does not mean we can not have processes that, themselves, are parallel, thus needing more than one processor.

[1] PERT (Program Evaluation and Review Technique) charts were first developed in the 1950s by the Navy to help manage very large, complex projects with a high degree of intertask dependency.

$$
\begin{array}{rcl}
jobs & \rightarrow & job\ \ jobs \\
 & \rightarrow & job \\
job & \rightarrow & jobName\ \ \text{‘:’}\ \ deps\ \ walltime\ \ nrCpus\ \ actions \\
walltime & \rightarrow & \text{‘(’}\ \ \mathbf{TIME}\ \ \text{‘)’} \\
nrCpus & \rightarrow & \text{‘[’}\ \ \mathbf{INT}\ \ \text{‘]’} \\
 & \rightarrow & \epsilon \\
jobName & \rightarrow & \mathbf{ID} \\
 & \rightarrow & \mathbf{ID}\ \ \mathbf{VAR} \\
deps & \rightarrow & jobName\ \ deps \\
 & \rightarrow & \epsilon \\
actions & \rightarrow & action \\
 & \rightarrow & actions\ \ action \\
action & \rightarrow & shellCommand \\
 & \rightarrow & perlCommand \\
 & \rightarrow & setDefinition \\
shellCommand & \rightarrow & \mathbf{TAB}\ \ \mathbf{SHELL} \\
perlCommand & \rightarrow & \mathbf{TAB}\ \ \text{‘}\texttt{sub\{}\text{’}\ \ \mathbf{PERL}\ \ \text{‘\}’} \\
setDefinition & \rightarrow & \mathbf{TAB}\ \ \mathbf{VAR}\ \ \text{‘}\texttt{<-}\text{’}\ \ \mathbf{SHELL} \\
 & \rightarrow & \mathbf{TAB}\ \ \mathbf{VAR}\ \ \text{‘}\texttt{<-}\text{’}\ \ \text{‘}\texttt{sub\{}\text{’}\ \ \mathbf{PERL}\ \ \text{‘\}’}
\end{array}
$$

Fig. 1. Makefile::Parallel DSL grammar.

```
prepare:  (5:00)
      mkdir OutputData
      p <- sub{ print "$_\n" for (3..10) }

run$p: prepare (20:00:00) [2]
      runMyProgram -p $p InputData > OutputData/run.$p

cleanup: run$p (5:00)
      for a in @p; do rm -f OutputData/run.${a}.tmp; done
```

Fig. 2. Simple Makefile::Parallel example.

2.1 Support for both Shell and Perl actions

Although most processes are binary programs, sometimes we need some specific small actions to put files in some special places and to change some file contents. With that in mind we support Bash (as makefile) and, because we work especially with Perl (and the scheduler is implemented in Perl), we also support actions written in Perl.

In the example above all actions are in Bash for simplicity.

2.2 Support for parametric rules

It is quite usual to test some specific programs with different algorithms or different parameters. In other cases, we need to split a big job on small chunks to be processed independently.

Both situations need the use of parametric rules: rules where variables get replaced by a set of values. For instance, on the example above, the variable `$p` on the *run* rule gets replaced by the values of the `p` set. This set is defined on the *prepare* rule with some Perl code. That Perl code returns a text file (in the *stdout*) with a value per line. These values are the `p` set elements.

So, `p` gets the values from 3 to 10: $p = \{3, 4, 5, 6, 7, 8, 9, 10\}$. Then, the rule *run$p* is replaced by:

```
run3: prepare(20:00:00) [2]
     runMyProgram -p 3 InputData > OutputData/run.3

run4: prepare(20:00:00) [2]
     runMyProgram -p 4 InputData > OutputData/run.4
...

run10: prepare(20:00:00) [2]
     runMyProgram -p 10 InputData > OutputData/run.4
```

In the *cleanup* rule, something similar happens. The dependency list is expanded with `p` values, and the special list variable `@p` in the rule action is also expanded to all values it can take. Note that `@p` is expanded correctly accordingly with its context (Bash or Perl). If we did not have parametric rules we would need to write:

```
cleanup: run3 run4 run5 run6 run7 run8 run9 run10 (5:00)
      for a in run3 run4 run5 run6 run7 run8 run9 run10; do
              rm -f OutputData/run.${a}.tmp; done
```

Note that the variable `${a}` is a standard Bash variable that will be instantiated during run-time.

3 Makefile::Parallel Scheduler

The scheduler — pmake — is written in the Perl language. It takes a specification file and schedules jobs accordingly with their dependencies.

Since the beginning we had in mind to develop more than one scheduler subsystem to submit jobs. While most clusters use ROCKS, there are other scheduler systems. Also, a simple SSH scheduler could be created. Thus, the code was modularized: an abstract class to represent any scheduler, and a set of subclasses implementing all the methods for a specific system.

```
do {
    launch_rules_with_satisfied_dependencies()
    terminated_processes = gather_terminated_processes()
    for( process in terminated_processes ) {
        if ( defines_parameters( process ) ) {
            parameters = calculate_parameters( process )
            expand_dependency_graph(parameters)
        }
    }

    save_journal()
    sleep(10)
} while(! all_processes_executed())

generate_profiling_information()
print_report()
```

Fig. 3. Scheduler engine behavior algorithm.

3.1 Scheduler Behavior

The basic scheduler behavior is specified on figure 3. The scheduler visits the dependency graph specification generated by the parser and, for every job with fulfilled dependencies, send them to run on the selected subsystem. The scheduler then waits for any of the running jobs to die or to end, by asking their state to the subsystem.

When a process ends, the scheduler gets information from the subsystem (return code, CPU time, memory used). If the return code does not indicate failure, all variables defined in the rule are instantiated (evaluating the Perl or Bash definition), and the dependency graph is modified on-the-fly to reflect the instantiated variables. Then, all processes not yet executed are browsed, and if any is found with all dependencies fulfilled, it is submitted to the system. The process continues all over again until no processes need to be executed.

From time to time, the scheduler saves some part of his internal state to permanent storage (hard disk), in the form of a journal. This is useful in case the specification is stopped by an error, or some kind of problem exists with the subsystem (as power shortages). Later, the user can simply pass an option to `pmake`, and the scheduler will bypass all processes ended correctly.

Two subsystem implementations were developed for now: a PBS designed for clusters running mainly ROCKS, and a Local, designed for desktop processing.

3.2 PBS Subsystem

High-performance clusters are the computing tool of choice for a wide range of scientific disciplines. Yet straightforward software installation, management, and monitoring for large-scale clusters have been consistent and nagging problems for non-cluster experts. The free ROCKS cluster distribution takes a fresh perspective on cluster installation and management to dramatically simplify version tracking, cluster management and integration[4]. The toolkit centers around a Linux distribution based on the Red Hat Enterprise line, and includes work from many popular cluster and grid specific projects.

One utility found in any cluster toolkit is a Portable Batch System (PBS). Basically, scheduling software let the cluster run like a batch system, allowing the allocation of cluster resources, such as CPU time and memory, on a job-by-job basis. Jobs are queued and run as resources become available, subject to the priorities established[5]. PBS is a powerful and versatile system.

When the scheduler emits a process to run on this subsystem, a PBS script is generated and sent to the cluster queue using the command `qsub`. To check if a process is still running Makefile::Parallel uses the `qstat` program with the appropriate parameters.

Eventually some processes finish. When the scheduler detects that (using the previous call to this subsystem), it asks for additional information about the dead process using the output of the `tracejob` program. This program returns detailed information, including (real) CPU time, return code, and memory usage of the process.

If the scheduler needs to stop a running process, the subsystem can call `qdel` to kill the process if it already running or to remove it from the queue otherwise.

While ROCKS supports dependencies between jobs (you can specify jobs dependency when submitting a job) it is not versatile enough for most users needs.

3.3 Local Scheduler Subsystem

Since most of modern computers are becoming multiprocessor by nature, a local scheduler is a good way of exploring parallelization on small to medium workflows.

With this in mind, a local scheduler subsystem was implemented. Running jobs on the development desktop machine allowed faster bug tracking and less time of coding. At the same time, one could not have access to a multicomputer cluster so this subsystem can be used in a variety of situations.

pmake local scheduler operation resembles the *GNU make* program on many aspects. First of all, if invoked with no parameters, pmake takes the specification and run it sequentially — the jobs are run as if a single pipeline exists. This could be optimal on a desktop machine.

However, as the level of multiprocessors rises the user may want to use the additional processor power available. For this reason, this subsystem was designed to accept a parameter that specifies how many parallel pipelines the scheduler must support on this execution.

This subsystem uses the *fork-exec-perror* paradigm and trust the operating system the correct map of the job to a free processor (both logical or physical).

4 Case Study

We have at our disposal a multicomputer cluster formed by approximately 140 CPUs and 50 nodes. The cluster runs Linux and ROCKS. This means that we can use the TORQUE Resource Manager to schedule our jobs.

Although we are using Makefile::Parallel on two different research fields (Bio Informatics and Natural Language Processing) we just present here the later, because it is the most interesting and was the real motivation for this work.

The code shown on figure 4 is part of a bigger pmakefile, working in production for the word-alignment[6] of big parallel texts (bitexts: texts and respective translations) and extraction of translation examples[7]. The word-alignment task needs to create big sparse matrixes in memory and for big texts (with more than 300 MBytes of text files) this matrix does not fit on main memory. Thus the solution is to split the big text in smaller pieces, and process them independently. At the end the processing result is merged up.

Figure 6 shows graphically this process. Important to note that this graph is generated by Graphviz[8] at run time with times for each task, and their dependencies. In fact we are getting 25% to 15% of the time needed in sequential mode in our tasks (this value highly depends on the current cluster load, as expected).

During process pmake will output progress information to a log file as shown in figure 5. This log shows the processes being launched and finishing, as well as the time spent for each one. Together with the graph shown in figure 6, pmake generates also a file with timing information for easy profiling as can be seen on figure 7.

5 Conclusions

Makefile::Parallel proven to be a useful tool to specify jobs dependencies and to schedule them efficiently.

Being able to perform a concise description makes it easier to define dependencies correctly than defining them by hand.

```
codify: (20:00:00)
        nat-codify -id=EurLex EurLex-PT EurLex-EN
        i <- sub{ $nr = ‘cat EurLex/nat.cnf |grep nr-chunks|cut -f 2 -d "="‘;
                  printf("%03d\n",$_) for (1..$nr); }

initmat$i: codify  (20:00:00)
        nat-initmat EurLex/source.$i.crp EurLex/target.$i.crp EurLex/mat.$i.in

ipfp$i: initmat$i (20:00:00)
        nat-ipfp 5 EurLex/source.$i.crp EurLex/target.$i.crp \
                   EurLex/mat.$i.in EurLex/mat.$i.out
        rm -f EurLex/mat.$i.in

postipfp$i: ipfp$i (20:00:00)
        nat-mat2dic EurLex/mat.$i.out EurLex/dict.$i
        rm -f EurLex/mat.$i.out

postbin$i: postipfp$i (20:00:00)
        nat-postbin EurLex/dict.$i \
                    EurLex/source.$i.crp.partials EurLex/target.$i.crp.partials \
                    EurLex/source.lex EurLex/target.lex \
                    EurLex/source-target.$i.bin EurLex/target-source.$i.bin
        rm -f EurLex/dict.$i

dicA: postbin$i (20:00:00)
        for a in @i; do \
           nat-dict add EurLex/source-target.bin EurLex/source-target.${a}.bin; \
        done
        for a in @i; do rm -f EurLex/source-target.${a}.bin; done

dicB: postbin$i (20:00:00)
        for a in @i; do \
           nat-dict add EurLex/target-source.bin EurLex/target-source.${a}.bin; \
        done
        for a in @i; do rm -f EurLex/target-source.${a}.bin; done

dump: dicA dicB (20:00:00)
        nat-dumpDicts -self EurLex
```

Fig. 4. Subset of NATools pmakefile.

```
[...]
2006/12/12 10:49:22 The job "ipfp005" is ready to run. Launching
2006/12/12 10:49:22 Launched "ipfp005" (23996)
2006/12/12 10:49:52 Process 23996 (ipfp005) has terminated [30s]
2006/12/12 10:49:52 The job "postipfp005" is ready to run. Launching
2006/12/12 10:49:52 Launched "postipfp005" (23997)
2006/12/12 10:50:02 Process 23997 (postipfp005) has terminated [10s]
[...]
```

Fig. 5. Makefile::Parallel log output.

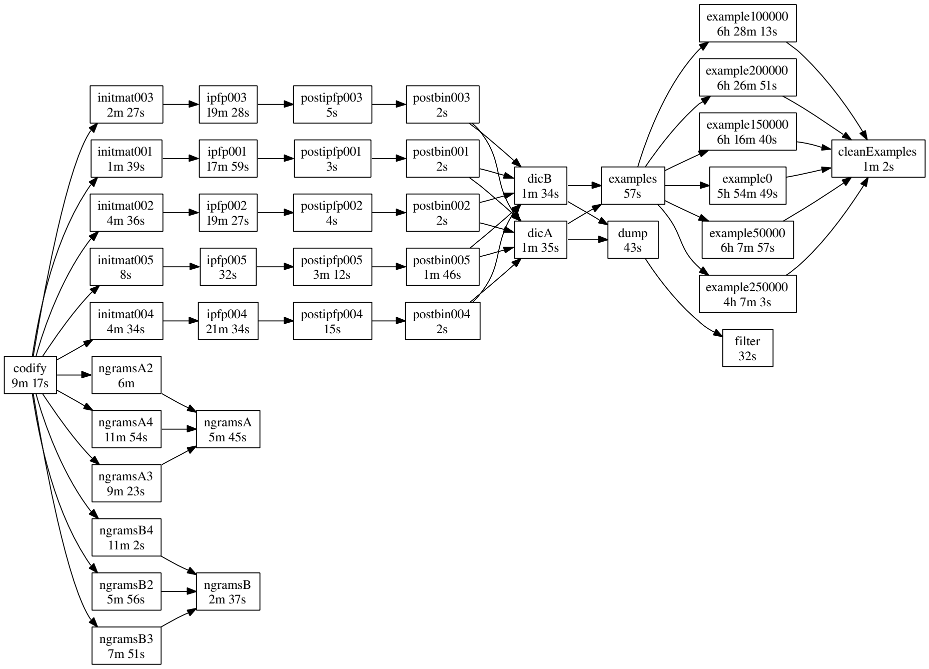

Fig. 6. Process dependency graph.

```
ID          Start Time          End Time            Elapsed
codify      2006-12-12T10:41:10 2006-12-12T10:49:11 8m 1s
ngramsA     2006-12-12T10:49:11 2006-12-12T11:07:46 18m 34s
ngramsB     2006-12-12T10:49:11 2006-12-12T11:05:44 16m 33s
initmat001  2006-12-12T10:49:11 2006-12-12T10:50:12 1m
initmat002  2006-12-12T10:49:11 2006-12-12T10:50:43 1m 31s
initmat003  2006-12-12T10:49:11 2006-12-12T10:51:03 1m 51s
[...]
```

Fig. 7. Makefile::Parallel report output.

The syntax is versatile enough to be applied in more than one research area. The parser is quite simple, and the fact of supporting the embed of external languages makes it yet more powerful.

Report facilities and graph generation are useful for week report generation about cluster usage and project evolution. The generated graphs are useful to explain how software works, how the different jobs interact together, and what are the critical jobs (and paths).

Worth saying that Makefile::Parallel has almost no running cost since it spends most of the time sleeping, waiting for events to happen.

References

1. Campbell, D., Grevstad, C.: A tutorial for make. In: ACM'85: Proceedings of the 1985 ACM annual conference on The range of computing : mid-80's perspective, New York, NY, USA, ACM Press (1985) 374–380
2. Douglas, D.E.: Pert and simulation. In: WSC '78: Proceedings of the 10th conference on Winter simulation, Piscataway, NJ, USA, IEEE Press (1978) 89–98
3. Desarmenien, F.: Parse::Yapp — perl extension for generating and using lalr parsers. Perl module (2001) http://search.cpan.org/dist/Parse-Yapp/.
4. Sacerdoti, F.D., Chandrai, S., Bhatia, K.: Grid systems deployment & management using rocks. IEEE International Conference on Cluster Computing, San Diego (September 2004)
5. Sloan, J.D.: High Performance Linux Clusters with OSCAR, Rocks, OpenMosix, and MPI. O'Reilly (2004)
6. Simões, A.M., Almeida, J.J.: Natools – a statistical word aligner workbench. SEPLN (Sep. 2003)
7. Simões, A., Almeida, J.J.: Combinatory examples extraction for machine translation. In Lønning, J.T., Oepen, S., eds.: 11th Annual Conference of the European Association for Machine Translation, Oslo, Norway (19–20, June 2006)
8. Gansner, E.R., North, S.C.: An open graph visualization system and its applications to software engineering. Software — Practice and Experience **30**(11) (2000) 1203–1233