A Parallel Version of the Boyer-Moore Prover

Matt Kaufmann
Matt Wilding

Technical Report 39 February 1989

This research was supported in part by ONR Contract N00014-88-C-0454. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of Computational Logic, Inc., the Office of Naval Research or the U.S. Government.

Table of Contents

1. Introduction

The idea of this system is to use idle processors to increase the speed of NQTHM. There were many design considerations, including:

- **No shared file system.** We want to allow the dispatcher to use processors besides the ones connected to our file system. This also allows us to implement a distributed system that has a chance to recover from any problems that occur.
- **Easy to kill jobs on a given host.** Ultimately, if all else fails, we want a way to kill ALL processes we have created. This also gives machine users some measure of control.
- **Easy to prevent job assignment on a given host.** Machine users must be able to remove their processor from the shared pool if they wish.
- **Robust.** The system should be able to recover from most kinds of processor or network problems.
- **Nice user interface.** Plenty of flexibility with reasonable defaults, output that tells the user what he needs to know, etc.

We wish to use the system to process an NQTHM event list in parallel, with a given "granularity". ~~If we have~~ a granularity of **n**, then the first job will process the first **n** PROVE-LEMMAs. Each subsequent job will process **n** PROVE-LEMMAs. Since an arbitrary subsequence of PROVE-LEMMAs may require a previous definition, each job will process all previous definitions. A subsequence of PROVE-LEMMAs may also require PROVE-LEMMAs from a previous subsequence, so each job will process all PROVE-LEMMAs from previous subsequences, treating each PROVE-LEMMA as an ADD-AXIOM.

This system allows the user to run a job on multiple machines in parallel. The issue of machine use has been discussed at Computational Logic and a preliminary policy adopted. It is the policy at Computational Logic to allow users to run parallel jobs using the *dispatcher*. Spawned jobs are to be run at the default (nice) priority level, and it is OK for any user to "kick" spawned parallel jobs off ~~his~~ local machine if he wishes (without feeling bad about it!)

Most users will only need or want to read Subsection 2.1.

Section 2 is a User's Manual for the system. Section 2.1 describes basic use of the system and includes everything most people will need to use the system. Section 2.2 describes options to the system. Section 2.3 describes some hooks that allow customization of the system. Section 2.4 describes the use of the *dispatcher*, the part of the system that distributes the work to other processors that may be used on work other than NQTHM jobs.

Section 3 is a System Guide. Section 3.1 describes how the dispatcher is implemented. Section 3.2 describes how events files are broken up and how the dispatcher is applied to this problem.

Section 4 includes results and conclusions. Section 4.1 compares the system's performance against sequential runs. Section 4.2 suggests future work. Section 4.3 summarizes our conclusions.

Appendix A describes the instrumentation added to the system to keep track of its use. Appendix B describes an experiment where the dispatcher is used to compile code in parallel. Appendix C is the code.

Acknowledgements: We'd like to thank Bob Boyer and Ross Overbeek for their help. Ross wrote a program that gave us the idea to build this system. Bob created the "nuke-all" shell script for us and suggested the basic paradigms for processing nqthm events and for compiling nqthm in parallel. Both provided advice and encouragement throughout the project.

2. A User's Manual

2.1 Basic Use

2.1-A An example

The following example shows how the function DO-FILE-PARALLEL may be called to run NQTHM events in parallel. Comments are inserted in italics. Later subsections explain which directories and files need to be present for this function to execute successfully, and some commands that control parallel jobs.

```
cli:wilding[46]% akcl     {start up Lisp}
AKCL (Austin Kyoto Common Lisp)  Version(1.57) Thu Sep 29 21:27:15 CDT 1988
Contains Enhancements by W. Schelter

>(load "/usr/local/src/parallel/top.lsp") {Load in the parallel NQTHM-manipulation code}
Loading /usr/local/src/parallel/top.lsp
Loading /usr/local/src/parallel/from-nqthm.o
start address -T 191000 Finished loading /usr/local/src/parallel/from-nqthm.o
Finished loading /usr/local/src/parallel/top.lsp
T

>(load-dispatcher)  {Load in parallel NQTHM code}
Loading /local/src/parallel/dispatch.o
start address -T 1c8800 Finished loading /local/src/parallel/dispatch.o
Loading /local/src/parallel/bm.o
start address -T 1d0800 Finished loading /local/src/parallel/bm.o
NIL

>(do-file-parallel "/usr/home/kaufmann/demo-permutationp.events" 2)
{Execute events in file with 2 PROVE-LEMMA events per job}
Clearing directory jobs/.
Creating jobs files... done.

Invoking dispatcher.
Initializing job information... done.
Clearing directory output/.
Clearing directory temp/.

Hosts requested:  ("client12" "cli" "anderson" "jingles" "scarab"
"decaf" "client13" "elgin" "oscar").
Hosts currently blocked:  NONE.

starting>> client12 : DELETE (job# 1)                                0:00
starting>> cli : DELETE-COMMUTATIVITY (job# 2)                      0:00
starting>> anderson : PERMUTATIONP-PRESERVES-MEMBER (job# 3)        0:00
starting>> jingles : PERMUTATIONP-TRANSITIVE (job# 4)               0:00
completed>> anderson : PERMUTATIONP-PRESERVES-MEMBER (job# 3)       0:01
completed>> client12 : DELETE (job# 1)                              0:01
completed>> cli : DELETE-COMMUTATIVITY (job# 2)                     0:01
completed>> jingles : PERMUTATIONP-TRANSITIVE (job# 4)              0:01
All events have been run successfully.
T
>
```

2.1-B Environmental requirements

```
          jobs/      temp/       output/             hosts (optional)

        <event1>   finish.1   <event1>.output
        <event2>   finish.2   <event1>.status
          ...        ...      <event2>.output
                   output.1   <event2>.status
                   output.2        ...
                     ...
                   status.1
                   status.2
                     ...
```

The system uses files in three subdirectories of the current working directory. By default, these subdirectories are called **jobs/**, **temp/**, and **output/**. A user may also optionally have a file named **hosts** in the current working directory.

The **jobs/** subdirectory contains the input file associated with each of the processes to be executed. It is created automatically from the input events file and the granularity (the number of PROVE-LEMMAs requested per job). The jobs are assigned to processors in alphabetical order. The name of each job is the name of the first PROVE-LEMMA in the file.[1]

```
cli:wilding[47]% ls jobs
DELETE-COMMUTATIVITY            PERMUTATIONP-PRESERVES-MEMBER
MEMBER-APPEND                   PERMUTATIONP-TRANSITIVE
```

The **temp/** subdirectory contains files used temporarily during the execution of the jobs. Each scheduled job is assigned a unique number. There are three files associated with each scheduled job contained in the **temp/** subdirectory. The output.n file is the output created by job n. The status.n file communicates whether the nqthm events were successfully (or unsuccessfully) processed to completion, or whether the job was terminated before it could complete. The finish.n file is created when the remote job finishes.

By default, when the finish.n file is created the output.n and status.n files are moved to the output/ subdirectory described below. Note that if the system does not have to recover from errors, there will be as many scheduled jobs as there are files in the jobs/ directory.

```
cli:wilding[48]% ls temp
finish.1        finish.4        output.2        status.1        status.4
finish.2        junk.lsp        output.3        status.2
finish.3        output.1        output.4        status.3
```

The **output/** subdirectory contains the output from jobs, and information about the status of each job. If scheduled job n executes job X from the jobs/ subdirectory without error, then X.output in the output/ directory is identical to output.n in the temp/ directory and X.status in the output/ directory is identical to status.n in the temp/ directory.

[1]If the file contains BOOT-STRAPs or NOTE-LIBs, then job naming and assigning is slightly different. Events of that type are always the first events in a job, so the previous job may have fewer than PROVE-LEMMAs than the granularity. Job names are the first event name in the job followed by a "." followed by a number that is the location in the file of the most recent BOOT-STRAP or NOTE-LIB. This is necessary since an events file with BOOT-STRAPs or NOTE-LIBs may have duplicate event names.

```
cli:wilding[49]% ls output
DELETE-COMMUTATIVITY.output          PERMUTATIONP-PRESERVES-MEMBER.output
DELETE-COMMUTATIVITY.status          PERMUTATIONP-PRESERVES-MEMBER.status
MEMBER-APPEND.output                 PERMUTATIONP-TRANSITIVE.output
MEMBER-APPEND.status                 PERMUTATIONP-TRANSITIVE.status
```

If the current directory contains a file named (by default) "hosts", then it is used to find hosts upon which to run. Each host name must appear on a separate line. The number of times a hostname appears in the host file will be the number of processes that may be simultaneously placed on it. (For example, if "cli" appears twice in the host file, then the host cli may have two simultaneous processes running on it.)

If a hosts file does not exist in the local directory, a system default is used.

```
cli:wilding[50]% cat hosts
client12
cli
anderson
jingles
scarab
decaf
client13
elgin
oscar
```

2.1-C Problems That are Detected

There are two kinds of errors that are detected by the system. First, if a job finishes but does not produce the appropriate output, then the system concludes that the remote job ended abnormally and reschedules the work. (To construct the following example, one remote process was killed. The system detected the problem and rescheduled the job.)

```
>(do-file-parallel "/usr/home/kaufmann/demo-permutationp.events" 3
:kill-if-no-progress 60)
Clearing directory jobs/.
Creating jobs files... done.

Invoking dispatcher.
Initializing job information... done.
Clearing directory output/.
Clearing directory temp/.

Hosts requested:  ("client12" "cli" "anderson" "jingles" "scarab"
"decaf" "client13" "elgin" "oscar").
Hosts currently blocked:  NONE.

starting>> client12 : DELETE (job# 1)                               0:00
starting>> cli : MEMBER-DELETE-OTHER (job# 2)                       0:00
starting>> anderson : PERMUTATIONP-TRANSITIVE (job# 3)              0:00
***NOT completed>> anderson : PERMUTATIONP-TRANSITIVE (job# 3)      0:00
starting>> anderson : PERMUTATIONP-TRANSITIVE (job# 4)              0:00
completed>> client12 : DELETE (job# 1)                              0:01
completed>> cli : MEMBER-DELETE-OTHER (job# 2)                      0:01
completed>> anderson : PERMUTATIONP-TRANSITIVE (job# 4)             0:01
All events have been run successfully.
T

>
```

The other type of error detected by the system is when no progress is made. If the output file is not written to for too long, the system kills the local process (in case it is not already dead) and restarts the job. (To construct this example, all local processes related to parallel jobs were killed on the local host.

After a while the system detected the problem and took action.)

```
>(do-file-parallel "/usr/home/kaufmann/demo-permutationp.events" 3
:kill-if-no-progress 60)
Clearing directory jobs/.
Creating jobs files... done.

Invoking dispatcher.
Initializing job information... done.
Clearing directory output/.
Clearing directory temp/.

Hosts requested:  ("client12" "cli" "anderson" "jingles" "scarab"
"decaf" "client13" "elgin" "oscar").
Hosts currently blocked:  NONE.

starting>> client12 : DELETE (job# 1)                               0:00
starting>> cli : MEMBER-DELETE-OTHER (job# 2)                       0:00
starting>> anderson : PERMUTATIONP-TRANSITIVE (job# 3)              0:00
*** KILLED>> client12 : DELETE (job# 1)                             0:02
*** KILLED>> cli : MEMBER-DELETE-OTHER (job# 2)                     0:02
*** KILLED>> anderson : PERMUTATIONP-TRANSITIVE (job# 3)            0:02
starting>> client12 : DELETE (job# 4)                               0:02
starting>> cli : MEMBER-DELETE-OTHER (job# 5)                       0:02
starting>> anderson : PERMUTATIONP-TRANSITIVE (job# 6)              0:02
completed>> client12 : DELETE (job# 4)                              0:03
completed>> cli : MEMBER-DELETE-OTHER (job# 5)                      0:03
completed>> anderson : PERMUTATIONP-TRANSITIVE (job# 6)             0:03
All events have been run successfully.
T

>
```

2.1-D The Par commands

There are several commands that are useful for controlling parallel jobs. They are used from the shell, and none of them has arguments. Note that it is the policy at Computational Logic that it is OK for any user to do execute any of these commands whenever he wishes.

- **parstop** -- block the local machine from getting more par jobs assigned to it (blocks are removed at 8PM and 8AM every day) and kill all par jobs already running on this machine.[2]
- **parblock** -- block the local machine from getting more par jobs assigned to it (blocks are removed at 8PM and 8AM every day)
- **parunblock** -- free the local machine for par jobs
- **parshowblocked** -- show which machines are currently blocked
- **parkill** -- kill all par jobs on this machine
- **parcount** -- give the number of jobs owned by user par on the local processor (this is usually twice the number of rsh's on the local machine)

Here's an example of these commands in action:

[2] This command is simply a parblock followed by a parkill

```
client12:wilding[11]% parshowblocked
jingles
client12:wilding[12]% parstop
client12 blocked
All PAR jobs killed on client12
client12:wilding[13]% parshowblocked
client12   jingles
client12:wilding[14]% parunblock
client12 unblocked
client12:wilding[15]% parcount
3 PAR jobs executing
client12:wilding[16]% parkill      kill the current jobs but allow later ones
client12:wilding[17]% parcount
0 PAR jobs executing
client12:wilding[18]% parcount
2 PAR jobs executing
client12:wilding[19]% parstop      kill the current jobs and prohibit later ones
client12 blocked
All PAR jobs killed on client12
client12:wilding[20]% parcount
0 PAR jobs executing
client12:wilding[21]%
```

2.2 DO-FILE-PARALLEL options

There are several options available when using DO-FILE-PARALLEL to run NQTHM jobs in parallel. They are passed as key parameters in the function invocation. The general form of an invocation of DO-FILE-PARALLEL is:

```
(do-file-parallel infile granularity
                  :jobs-directory-name <directory-name>
                  :output-directory-name <directory-name>
                  :hosts-file-name <file-name>
                  :local-host-first <t | nil>
                  :kill-if-no-progress <number of seconds>
                  :command-name <command-name>
                  :front-end <file-name>
                  :delay <nil | number of seconds>
                  :nice-flag <t | nil>              )
```

2.2-A :jobs-directory-name

Default: "jobs/"

The jobs directory name is a string that contains the name of the subdirectory that is to be used to hold the input files for the dispatcher. If the parameter given is not a real subdirectory name for the current directory, or if the parameter does not end with a '/' character, then an error is reported. The job-directory-name subdirectory is cleared with every run of DO-FILE-PARALLEL.

2.2-B :output-directory-name

Default: "output/"

The output directory name is a string that contains the name of the subdirectory that is to be used to hold the output files from the jobs. If the parameter given is not a real subdirectory name for the current directory, or if the parameter does not end with a '/' character, then an error is reported. The output-directory-name subdirectory is cleared with every run of DO-FILE-PARALLEL.

2.2-C :hosts-file-name

Default: "hosts"

The hosts file name is a string that contains the name of a file that contains the hosts upon which to run the parallel jobs. As described in section 2, if the host file name exists (either the default "hosts" in the current directory or the file specified with the :hosts-file-name key parameter) then it is used to provide the list of host names. If the host file name does not exist, than a system-wide default is used instead.

The hosts file contains one host per line. Each host name should be a valid host accessible to the user with an rsh command from the local host. A host may have as many processes assigned to it as there are occurrences of its name in the hosts file.

2.2-D :local-host-first

Default: t

A list of hosts is maintained by the system for use in assigning jobs when necessary. Since jobs are assigned to the hosts on this list starting from the beginning, the order of the hosts in the list affects the job. This is particularly important when one considers system errors - a job that does not complete successfully is reassigned to the first host on the list that does not have a job assigned to it, even if that host was the one that just failed! If the first host is inaccessible for some reason, the system will loop forever reassigning a job to that host.

Since we are already relying on the local host, in general we want the local host to appear first in the host list. If :local-host-first is non-nil, then the host list lists the hosts in the same order as was found in the hosts file, except that occurrences of the local host are moved to the front of the list. If :local-host-first is nil, then the host list is in the same order as was found in the hosts file.

2.2-E :kill-if-no-progress

Default: 200

As described in section 2, remote jobs that do not produce output for too long are killed and their work rescheduled. The minimum "safe" time in seconds for before a job may be killed is given with the parameter :kill-if-no-progress.

If the value of this parameter is n, then the processor checks every n seconds to see if any processes should be killed. If a process has not written to its output file in the last n seconds, then it is killed and the work rescheduled.[3]

The garbage collection message is turned on in remote jobs (in the default front-end file - see :front-end below) when they are set up to guarantee that long periods of time between output updates really means trouble and is not just NQTHM taking a long time.

2.2-F :command-name

Default: "pc-nqthm"

The command name is the command that is to be run on the remote hosts. It must be accessible on the remote host.

Probably the only time a user would want to change the default command is to use another version of the theorem prover, for example "nqthm".

2.2-G :front-end

Default: "/local/src/parallel/front-end.lsp"

The front-end is a string that is the name of a file containing forms to be sent to the remote processes before the job-specific info and the events. The default contains code that helps set up the remote process for running the subsequence of events that need to be run by that job.

Most users will not want to mess with this parameter.

[3]This means that a process may take longer than n seconds to be killed once it stops producing output, but no longer than 2*n seconds.

2.2-H :delay

Default: nil

The parameter describes how often the local host will check to see if there are jobs that have completed. If the number n is provided, then the local host will sleep n seconds between checking for completions.

If nil is provided (or the default is used) then the delay will be set equal to the granularity. Thus, if each job processes 20 PROVE-LEMMAS and the delay is set to nil, then the local processor will sleep 20 seconds after checking for completed jobs.

2.2-I :nice-flag

Default: t

If nice-flag is non-nil, then the jobs run on remote machines will be run using "nice". That is, they will run so that they have a lower priority than most other jobs on the system. If nice-flag is nil, then the spawned jobs will be run at normal priority.

Note that even nice jobs take resources, so running at a lower priority will not guarantee that running a parallel job will have no effect on the systems used.

It is the policy at Computational Logic to run jobs at the default (nice) priority level.

2.3 Some Implementation Hooks

This section describes global variables whose values are used by the system. Using them, the user may customize the system somewhat.

The following variables may be set by the user.

- ***output-completed-string*** (Default: "Boyer-Moore job terminated") This string appears in the status file of a job. It signifies that the job completed. Whether the job completed successfully or with failure is communicated on the line after this line.
- ***no-io-in-parallel-flag*** (Default: nil) This flag directs whether the remote processes should produce the complete NQTHM output or an abbreviated version. If non-nil the abbreviated version is produced.
- ***delete-jobs-flag*** (Default: nil) This flag directs whether the job input files should be deleted after they are all processed. If nil the job input files are retained.
- ***clear-query-flag*** (Default: nil) This flag directs whether the the user should be queried before the jobs, temp, and output subdirectories are cleared. If non-nil the query is performed.
- ***local-host-warning*** (Default t) This flag directs whether the user should receive a warning if the first host in the host list is not his local host. This is checked after the host list is possibly rearranged to move the local host to the front of the host list (as described in section 2.2). As noted before, if the first host is inaccessible for some reason, the system will loop forever reassigning a job to that host.
- ***save-temp-files-flg*** (Default nil) This flag directs whether the files in the temp/ subdirectory should be copied or moved to the output directory when a job completes. If non-nil, the temp files are copied (and therefore saved).
- ***kill-dispatcher-upon-seeing-failure*** (Default nil) This flag directs whether the parallel job should finish as soon as failure is detected. If non-nil, the job stops as soon as any of the jobs returns a failure.
- ***system-parallel-directory*** (Default "/local/src/parallel/") This directory name contains various files the dispatcher needs to operate. These files are discussed in Section 3.1.
- ***parkill-command*** (Default "/local/bin/parkill") This string is submitted to the system if the system ends abnormally (e.g. the user aborts)
- ***protected-hosts-subdirectory*** (Default "protected-hosts/") This subdirectory of *system-parallel-directory* contains the "block" files used by the dispatcher. (See Section 3.1.)
- ***lock-file-name*** (Default "lock-out-others.par") This is the name of the lock file that is used to lock the current directory from use as a parallel job current directory for someone else.
- ***output-info-subdirectory*** (Default "statistics/runs/") This subdirectory of *system-parallel-directory* holds the files that record job progress. These files are designed to help track system use. (See Appendix A.)
- ***all-valid-hosts-names*** (Default nil) If non-nil, host names from the hosts file are checked to make sure that they appear in this list. If they do not appear, a warning message appears.

2.4 Dispatcher Use

This section is intended for someone who wishes to use the dispatcher part of this system independently. We have tried to keep the dispatcher functionally separate in order to make it applicable to other problems than parallel NQTHM. If you just want to use this system to run NQTHM jobs in parallel, this subsection probably will NOT do you any good.

DON'T LET THIS SECTION CONFUSE YOU IF YOU SIMPLY WANT TO USE THIS SYSTEM TO RUN NQTHM JOBS IN PARALLEL. It is intended for those who want to build systems to do parallel work with other kinds of jobs.

Section 3.1 describes the dispatcher's implementation, and section 3.2 describes how the system is built on top of the dispatcher. Appendix B contains an example of using the dispatcher to compile in parallel.

Dispatcher Use Overview

The dispatcher takes work and passes that work out to some processors. When a processor is done with a task, the dispatcher updates its records and assigns a new job to the processor. Eventually all the work will be completed and the dispatcher will return with a value reflecting whether all the jobs were successful.

The dispatcher expects there to be in the current directory a jobs directory (default name : "jobs/") that contains the work to be done. By default, each file in the jobs directory contains the input to the command to be run remotely.

During the execution of the dispatcher, a temp subdirectory (name: "temp/") is used to keep track of the jobs as they progress.

After a job completes, its standard output is moved from the temp/ subdirectory to the the output subdirectory (default name "output/"). If the name of the job file is X, then X.output will contain the task's output and will appear in the output directory. X.status will contain the standard error stream output by the job and will also appear in the output directory.

The hosts to use for remote processing can be found in the hosts file (default name : file "hosts" in the current directory).

Dispatcher Use Example

```
cli:wilding[12]% ls jobs
job1        job2     job3     job4     job5
cli:wilding[13]% more jobs/job2
hostname; date; rsh cli date
cli:wilding[14]% more hosts
anderson
client12
oscar
cli
cli:wilding[15]% akcl
AKCL (Austin Kyoto Common Lisp)  Version(1.57) Thu Sep 29 21:27:15 CDT 1988
Contains Enhancements by W. Schelter

>(load "/local/src/parallel/from-nqthm.lsp")
Loading /local/src/parallel/from-nqthm.lsp
Finished loading /local/src/parallel/from-nqthm.lsp
T
```

```
>(load "/local/src/parallel/dispatch")
Loading /local/src/parallel/dispatch.o
start address -T 1c5800 Finished loading /local/src/parallel/dispatch.o
25392
>(dispatcher :command-name "sh" :completion-function #'(lambda (x)
                                                          (cons 'success nil))
:front-end "" :back-end "")
Hosts requested:  ("cli" "anderson" "client12" "oscar").
Hosts currently blocked:  NONE.

starting>> cli : job1 (job# 1)                                          0:00
starting>> anderson : job2 (job# 2)                                     0:00
starting>> client12 : job3 (job# 3)                                     0:00
starting>> oscar : job4 (job# 4)                                        0:00
completed>> cli : job1 (job# 1)                                         0:00
completed>> anderson : job2 (job# 2)                                    0:00
completed>> client12 : job3 (job# 3)                                    0:00
completed>> oscar : job4 (job# 4)                                       0:00
starting>> cli : job5 (job# 5)                                          0:00
completed>> cli : job5 (job# 5)                                         0:01
T

>(by)
Bye.
cli:wilding[16]% ls output
job1.output        job2.output     job3.output     job4.output     job5.output
job1.status        job2.status     job3.status     job4.status     job5.status
cli:wilding[18]% more output/job2.output
16538
anderson
Tue Feb 21 12:47:45 CST 1989
Tue Feb 21 12:56:29 CST 1989
cli:wilding[19]%
```

Dispatcher Invocation

A dispatcher invocation has the form

```
                                                     default
(dispatcher :jobs-directory-name <directory-name>    "jobs/"
            :output-directory-name <directory-name>  "output/"
            :hosts-file-name <file-name>             "hosts"
            :local-host-first <t | nil>              t
            :delay <number>                          15
            :kill-if-no-progress <number>            600
            :command-name <command-name>             "pc-nqthm"
            :completion-function <function>          #'nqthm-completed
            :front-end <file-name>                   "/local/src/parallel/front-end.lsp"
            :back-end <file-name>                    "/local/src/parallel/back-end.lsp"
            :nice-flag <t | nil>)                    t
            :job-list <job-list | nil>          )    nil
```

Most of these parameters are the same as those to DO-FILE-PARALLEL (see section 2.2), with the following exceptions:

- **:host-file-name** Like the :host-file-name parameter name to DO-FILE-PARALLEL except if the file does not exist an error occurs.
- **:completion-function** This is a function to be applied to the status file of a job to tell whether it completed successfully or not. The status file contains the output from the standard error stream of the job. This function will be applied when the remote process terminates. It is expected to return either nil (the job apparently aborted) or a pair of the form ('success . message) or (code . message) where code may be anything except 'success, and message may either be nil or a list suitable as the arglist for the function FORMAT. If this function returns nil for a job, that job is rescheduled. If all input jobs eventually produce status files that indicate success in this sense, then dispatcher returns t, otherwise

the dispatcher returns nil.

- **:front-end** This file is sent to the remote process as input before the job input file. If the empty string is used, then no front-end file is sent.
- **:back-end** This file is sent to the remote process as input after the job input file. If the empty string is used, then no back-end file is sent.
- **:job-list** If non-nil, this provides a list of job names in the jobs subdirectory to be run. This may be useful if only a subset of the jobs subdirectory is to be used, or if the jobs are to be assigned in a particular order. If nil, all jobs in the jobs directory will be assigned in alphabetical order.

3. Systems guide: implementation

This section contains a description of our implementation. Let us re-emphasize that it should be completely unnecessary to read this section if one simply wants to be able to use the system. Rather, we have included this section for those who would like to know how this all works at the lower levels, perhaps so that they can create variants of this system. One might even think of this section as documentation for the code; the code itself may be found in Appendix C.

The first subsection below is a guide to the implementation of the *dispatcher*, which has nothing to do with NQTHM but is a general-purpose program for running independent jobs in parallel. (The dispatcher's *use* is documented in Subsection 2.4.) The second subsection below describes the implementation of the parallel version of the Boyer-Moore prover on top of the dispatcher. We conclude this section with an explanation of the system front-end file.

3.1 Dispatcher implementation

The main function for the general *dispatcher* is the Lisp function DISPATCHER. The code and its comments (see Appendix C) are the ultimate reference. In this section we describe the algorithm it uses and some of the subsidiary functions.

Dispatcher algorithm:

1. **Initialize various environmental variables.** These include the starting time, the local host name, the user name, and the suffix to use for the filename where statistics of the run will be collected, e.g. the '567489' in "/local/src/parallel/statistics/runs/wilding.567489".
2. **Set up locking.** Only one run of the dispatcher is allowed in a given directory at a given time, in order to avoid clashing use of common directories. The file "lock-out-others.par" is created in the current working directory whenever the dispatcher is entered. The dispatcher starts by checking to see if the directory is already "locked" in this sense; if not, it locks the directory.
3. **Check if jobs exist.** If a job-list is provided, make sure that the jobs all exist in the jobs/ subdirectory.
4. **Set up the jobs.** The job names are simply the file names from the directory `jobs-directory-name` ("jobs/" by default), unless they are provided by the :job-list option.
5. **Set up the initial `hosts-jobs-alist`.** This is an association list which associates jobs with hosts. Initially each host is associated with `NIL`, indicating that no job has yet been assigned.
6. **Enter main loop.**
    - **Update completed jobs records.** Remove the terminated jobs from the `hosts-jobs-alist`. Tack those that didn't complete back on to the end of the list of unassigned jobs. (More on this below.)
    - **Possibly look for and kill bombed jobs.** If it has been longer than `kill-if-no-progress` seconds since we last looked for "bombed jobs", then kill all the jobs which haven't output any characters in the last `kill-if-no-progress` seconds and put them back on the list of unassigned jobs. In such cases, a message headed with "*** KILLED" will appear on the terminal.
    - **Assign jobs.** Assign jobs to hosts which are currently not busy, appropriately adjusting the `hosts-jobs-alist` and the list of unassigned jobs. Avoid hosts that are currently blocked (except for the local host). Print an appropriate "starting"

message to the terminal for each new job started.

- **Check for completion.** If no hosts are busy, return from the loop. Otherwise sleep for `delay` seconds.

7. **Report failed jobs.** Return `NIL` if there are any failed jobs and otherwise return `T`.
8. **Clean up.** Remove the lock (i.e. delete the file "lock-out-others.par") and report completion to statistics files. If execution didn't complete normally ther ˙) `parkill` to remove all local jobs owned by user `par`.[4]

One complicated thing about the code is how jobs are started. The Lisp function `SYSTEM` (as it exists in KCL and AKCL at CLInc) takes a string which is then given to the Shell to execute. Our function `SYSTEM-JOB-COMMAND` produces a string that, when given to `SYSTEM`, creates a job. This causes execution of the Shell command `parcsh`, which calls the Shell on its arguments after changing ownership of the process to the user `par`. The argument list for `parcsh` is of the form

```
PAR <host-name> <command-name> <unique-number> <front-end> <job> <back-end>
```

`PAR` is a c-Shell script (see Appendix C) that:

1. Write the process number to the file temp/output.n, where n is the <unique-number> supplied above, i.e. the unique job number.
2. Call `rsh` (remote Shell) with host <host-name> and command <command-name>, piping the concatenation of the files <front-end>, <job>, and <back-end> to its standard input stream. Send the standard output of this process to the file temp/output.n, and send its error output to the file temp/status.n, where (as above) n is the <unique-number>.
3. Create the file temp/finish.n (same n as above).

Notice that since we start by writing the process number to the file temp/output.n, we can kill a bombed job by first reading its process id from the first line of the output file and then issuing the appropriate kill command.

As mentioned above, we need to update completed jobs records. We have just explained that `PAR` uses a remote Shell call (i.e., `rsh`) to fire up a job on a remote machine, after which it creates a "finish" file. One may consider a job to be *terminated* if its corresponding "finish" file has been created. (The Lisp function `job-completed`, which should perhaps be called `job-terminated`, does this check.) Such a "terminated" job is to be removed from `hosts-jobs-alist`. But first it must be decided whether the job *completed* or not; if not, it should be put back on the list of unassigned jobs. This determination is up to the *completion function*, which by default is the function `nqthm-completed`. Recall from Subsection 2.4 that this function expects a file name, which in this case is the status file temp/status.n (where n is the job's unique number), and returns either `NIL` (which means that it was "unable to give a reliable answer") or a pair of the form `('success . message)` or `(code . message)`. In the former case (where `NIL` is returned) the job is considered to have failed to complete (and the message "***NOT completed" is printed out), and it is put on the list of unassigned jobs. Otherwise the job is considered to have completed (and the message "completed" is printed out), the message (if any) is printed out, and the output and status files are moved to the output/ subdirectory. If the first component of this pair is anything other than `'success` then the job is added to the list `*failed-job-names*`. When the dispatcher finally returns, if this list is not `NIL` then the list is printed out with an appropriate message and the dispatcher returns `NIL`. If all jobs succeed, then the dispatcher returns `T`.

[4]In many cases this will kill remote `par` jobs as well.

3.2 Parallel nqthm implementation on top of the dispatcher

As in the previous subsection, we leave to the code documentation the task of giving detailed specifications. What follows here is an overview of the execution of the main function, `DO-FILE-PARALLEL`.

1. **Set up locking.** This works just as it did in the dispatcher. We want the current working directory reserved for only this run of `DO-FILE-PARALLEL` since even before the dispatcher is called we will be writing to one subdirectory, namely (by default) jobs/.
2. **Check that appropriate files and directories exist.** These include the system's `front-end` file, the jobs/ subdirectory, the output/ subdirectory, the temp/ subdirectory, and the `hosts` file (or whatever the user supplied in place of these defaults).
3. **Set delay.** If the `:delay` keyword argument has not been supplied by the user, then set the delay to the granularity of the call to `DO-FILE-PARALLEL`.
4. **Reset.** Set the `*current-job-unique-number*` back to 0 and clear the relevant subdirectories (these are jobs/ and output/ by default, together with temp/).
5. **Create the jobs files.** These are the input files to be shipped to the remote hosts inbetween the front end and the back end. Note that jobs with events which follow a BOOT-STRAP or NOTE-LIB in the main file are suffixed with a natural number, i.e. they look like `<identifier>.n` where `n` is the position of the applicable BOOT-STRAP or NOTE-LIB in the main file.
6. **Check directory.** Be sure that the current working directory is what we started with; if not, change to it.
7. **Remove the locking.** Otherwise we won't be able to run the dispatcher!
8. **Run dispatcher.** Return what it returns and print a happy message if it returns `T`. Clean up by returning to the working directory that we started with in case that differs from the current working directory.

The dispatcher is called with `:back-end` set to the filename argument (for the events list) of the call of `DO-FILE-PARALLEL`. We'll omit discussion of the defaults, as these are documented earlier (see Section 2.2, DO-FILE-PARALLEL-OPTIONS), except to discuss briefly the function `NQTHM-COMPLETED`. Recall from the previous subsection that the `:completion-function` argument to the dispatcher takes a filename argument (which is supposed to be the name of a status file) and returns either NIL or a pair. The function `NQTHM-COMPLETED` in fact looks for a line that equals the `*output-completed-string*`, "Boyer-Moore job terminated", in the given file, and then reads the next line. If the first 7 characters of that next line are "success" (when converted to lower case), then it returns the pair `('success . NIL)`. Otherwise it returns the list `('failure . (<line>))`, where <line> is that line.

Note that the appropriate messages to the status file are placed there by the remote job. The top-level loop function `PAR-NQTHM-TOP-LEVEL` in the system's front-end file in fact uses a system call to `echo2` to print the string " FAILURE: The event <event-name> failed." to the error stream (and hence to the status file) in this case, where <event-name> is the name of the failed event; otherwise it prints "Success!!"

3.3 The system front-end file.

Recall that the default "front-end" file for `DO-FILE-PARALLEL` is the file /local/src/parallel/front-end.lsp. The file /local/src/parallel/front-end-with-doc.lsp is a version of that file with comments, so complete documentation may be found in that code. In this subsection we give describe that code (which may be found in Appendix C).

Recall that the front-end file is the first file sent into the standard input stream of the **nqthm** or **pc-nqthm** process. That is, a remote host will be reading in and executing the forms from this file. After all the forms in this file are read, the particular job file will be read in. The last form in the job file is **(PAR-NQTHM-TOP-LEVEL)**, where the function **PAR-NQTHM-TOP-LEVEL** is defined in the front-end file. It is a top-level loop which will process the forms in the back-end file, i.e. the file of events. The central thing to understand from the front-end file is the definition of **PAR-NQTHM-TOP-LEVEL**. That function executes a loop after which it "cleans up"[5]. Here is what its main loop does.

1. **Read the next form.**
2. **If there are no more events to process, return T.** There are no more events to process if we are either (a) at end-of-file or (b) at the event ***finish-name*** (set in the job-file to be the first event that we should *not* process).
3. **Print the next event, evaluate it, and print its value.** However, we turn PROVE-LEMMAs into ADD-AXIOMs until we find the starting event. The variable ***start-name*** is initialized to the starting event's name in the job file. In case there is a preceding BOOT-STRAP or NOTE-LIB the variable ***start-position*** is also set in that file, and all events before the appropriate BOOT-STRAP or NOTE-LIB are ignored.
4. **If the value is NIL, exit the loop with value NIL.**

The first part of the "cleanup" phase has already been described in the subsection above: A success message is printed to the error stream if all events evaluated to non-**NIL**, and otherwise a failure message is printed. (We also handle the case that a **READ** fails, i.e. and end-of-file is encountered during the process of reading the next form.) Finally, we exit in the C language tradition, i.e. with status 0 if all events evaluated to non-**NIL** and 1 otherwise. Our current implementation does not use that status information, however.

The only slightly tricky part of this strategy is that the "cleanup forms" are not evaluated in Lisp when an error is caused until control is returned to the built-in top-level loop. Fortunately, in KCL there is a global variable ***break-enable*** which one may initialize to **NIL** in order to avoid entering the break loop when an error occurs. This is the first thing we do in the front-end file.

The front-end file also contains the form **(setq sys::*notify-gbc* t)**, which turns on garbage-collection notification. This feature should make it virtually impossible for an NQTHM job to "bomb" simply because it's not putting out characters fast enough; if all other output is slow, still there are likely to be frequent garbage collection messages![6]

[5]in Lisp jargon, it executes the cleanup-forms of an UNWIND-PROTECT

[6]One exception is compilation, where certain phases of the code generation can take a long time without a garbage collect. In this case one may wish to specify a large number of seconds for the **:kill-if-no-progress** parameter, as illustrated in the example in Appendix B, where we use 1200.

4. Results and Conclusions

4.1 Trial Runs

We've run several tests to try to break the error handling capability of the system. These included killing processes in the middle of a job, adding sleep commands to jobs to make them "killable", and even turning off a remote processor before it finishes. In all these tests the problem was detected and the parallel job recovered.

The dispatcher's utility has been demonstrated separately from the problem of doing NQTHM runs in parallel. It was used to compile code in parallel. That experiment is described in Appendix B.

We've run several different nqthm files for testing. The largest nqthm job we've run in parallel so far has been the events that create the various shared libraries created by Bill Bevier. With 7 Sun 3/60 processors (including the processor that ran the dispatcher) the job took 2 hr 22 min, compared with 10 hr 31 min for one dedicated processor running pc-nqthm from the shell.

The speedup of about 4 1/2 is about 63% of the theoretical maximum. The following lines from the parallel run's output show that only a fairly small portion of of the 37% loss is due to uneven finish times of the jobs.

```
completed>> scarab : T (job# 63)                                      2:08
completed>> jingles : PUT-WITH-LARGE-INDEX (job# 52)                  2:09
completed>> elgin : TIMES-DISTRIBUTES-OVER-DIFFERENCE (job# 64)       2:09
completed>> decaf : PUTS-WITH-LARGE-INDEX (job# 55)                   2:10
completed>> oscar : QUOTIENT-DIFFERENCE-LESSP-ARG2 (job# 57)          2:17
completed>> client12 : PUTS-PUTS3 (job# 54)                           2:21
completed>> client13 : QUOTIENT-DIFFERENCE1 (job# 58)                 2:21
All events have been run successfully.
T
```

4.2 Future Work

There are several things we'd like to do (someday) that would increase the utility of this coarse approach to parallelism in NQTHM.

- **Integrate with J Moore's library utilities.** When NQTHM with efficient library utilities is released, it will have two possible impacts on our system. First, it may allow remote systems to avoid redoing the DEFNS and old PROVE-LEMMAS for each job. Second, and more importantly, it will be very desirable for our system to produce endorsed "books".
- **Include the notion of dependencies.** We should investigate better ways to create the job files. Some events depend on others, like PROVE-LEMMAs after a BOOT-STRAP, and some events take longer and should have processing resources devoted to them early.
- **Find the bottlenecks.** We're not sure right now what is keeping us from getting better performance. (63% may be as good as it gets, but we should at least know what the important factors are.)
- **Try some big runs.** The system has limits. (100 remote hosts would surely fill the dispatcher's process table, for example) We should find out what these are.
- **Try to run remotely** The system has been designed to run on machines that are not in the local area network. We should try it.

4.3 Conclusions

It's not difficult to take advantage of idle processors.

GNU EMACS with KCL running under Unix is a wonderful development environment.

The basic idea of running NQTHM event files in parallel by sending remote processors subsequences of events seems to work fairly well. With large runs we have obtained close to 2/3 of the theoretical speedup.

Appendix A
Instrumentation

In order to get a handle on parallel job usage, some instrumentation has been added to the code. There are several files that are updated when parallel work is done.

A.1 /local/src/parallel/statistics/blocks

This file contains information about the creation and removal of block files. There are 4 types of messages.

- **USER-BLOCK** A user has blocked a processor.
- **USER-UNBLOCK** A user has unblocked a processor.
- **USER-UNBLOCK-FAILURE** A user tried to unblock an unblocked processor.
- **SYSTEM-UNBLOCK** The system unblocked some processors. The list of block files follows.
- **SYSTEM-UNBLOCK-FAILURE** The system tried to unblock processors but there were none to unblock.

example block file:

```
+++02/20/89 17:13:10 USER-UNBLOCK kaufmann cli
+++02/20/89 19:15:16 USER-BLOCK wilding cli
+++02/21/89 16:28:16 SYSTEM-UNBLOCK
total 1
-rw-rw-r--  1 wilding        2 Feb 20 19:06 cli

+++02/21/89 16:28:47 SYSTEM-UNBLOCK-FAILURE
+++02/21/89 16:28:49 USER-BLOCK wilding cli
```

A.2 /local/src/parallel/statistics/job-info

This file contains information about the parallel jobs run. The information contains the date and time, the user, the dispatcher host, the run code number (see the next section), the requested hosts, and the blocked hosts.

example job-info file excerpt:

```
+++02/24/89 17:37:17 PAR-START kaufmann client12 355006
hosts: ("client12" "scarab" "elgin" "client13" "decaf")
blocked: ("anderson", "cli" "jingles")
+++02/24/89 17:48:38 PAR-END kaufmann client12 355006
hosts: ("client12" "scarab" "elgin" "client13" "decaf")
blocked: ("anderson" "cli" "jingles")
+++02/26/89 16:15:17 PAR-START wilding client12 522876
hosts: ("client12" "cli" "anderson" "jingles" "scarab" "decaf"
        "client13" "elgin" "oscar")
blocked: NIL
+++02/26/89 16:16:30 PAR-END wilding client12 522876
hosts: ("client12" "cli" "anderson" "jingles" "scarab" "decaf"
        "client13" "elgin" "oscar")
blocked: NIL
```

A.3 /local/src/parallel/statistics/runs

This directory contains a trace of all runs using the parallel system. There is a file in this directory of the form <user-name>.<code-number> for each parallel run. The code-number is the number found in the job-info file.[7] Each file contains the header information of the job-info file plus progress messages that the user received while he ran the job.

Example runs directory:

```
cli:wilding[4]% cd ~kaufmann/dispatcher
cli:wilding[5]% cd /local/src/parallel/statistics/runs
cli:wilding[6]% ls
kaufmann.345071 kaufmann.346447 kaufmann.355006 wilding.524027
kaufmann.345195 kaufmann.348576 wilding.522876  wilding.524251
kaufmann.345253 kaufmann.352817 wilding.523562  wilding.524815
cli:wilding[7]% cat wilding.522876
+++02/26/89 16:15:17 PAR-START wilding client12 522876
hosts: ("client12" "cli" "anderson" "jingles" "scarab" "decaf"
       "client13" "elgin" "oscar")
blocked: NIL
starting>> client12 : DELETE (job# 1)                              0:00
starting>> cli : DELETE-COMMUTATIVITY (job# 2)                     0:00
starting>> anderson : PERMUTATIONP-PRESERVES-MEMBER (job# 3)       0:00
starting>> jingles : PERMUTATIONP-TRANSITIVE (job# 4)              0:00
completed>> anderson : PERMUTATIONP-PRESERVES-MEMBER (job# 3)      0:01
completed>> client12 : DELETE (job# 1)                             0:01
completed>> cli : DELETE-COMMUTATIVITY (job# 2)                    0:01
completed>> jingles : PERMUTATIONP-TRANSITIVE (job# 4)             0:01
+++02/26/89 16:16:30 PAR-END wilding client12 522876
hosts: ("client12" "cli" "anderson" "jingles" "scarab" "decaf"
       "client13" "elgin" "oscar")
blocked: NIL
cli:wilding[8]%
```

[7]This code-number is actually the file-server's "universal time" in seconds, modulo 1,000,000.

Appendix B
Parallel compilation

In this appendix we describe an experiment using the dispatcher for parallel compilation of NQTHM. This method could be generalized to solving the problem of compiling arbitrary systems. However, we've chosen simply to build a reasonably optimal parallel NQTHM compiler at this point. We describe the parallel compiler and how fast it is.

B.1 How Parallel Compilation is Done

The NQTHM code is broken into the file sloop.lisp (which is Bill Schelter's loop macro definition) and 10 other files. Three of those 10 must be compiled in sequence first. The remaining 7 files may then be compiled in parallel. The function COMPILE-NQTHM-SEQ defined below is like the existing function COMPILE-NQTHM except that it doesn't compile sloop.lisp or the final 7 files.

```
(DEFUN COMPILE-NQTHM-seq ()
  ;;; **** This is all done before saving a core image
  (FLET ((LF (N)
             (LOAD (EXTEND-FILE-NAME N FILE-EXTENSION-BIN)))
         (CF (N)
             (COMPILE-FILE (EXTEND-FILE-NAME N FILE-EXTENSION-LISP))))
        (PROCLAIM-NQTHM-FILES)
        (system "date")
        (format t "~&Completed proclaiming nqthm files.~&")
        ;;; (CF "sloop") ***** We assume that sloop exists in any reasonable system!
        (LF "/local/src/nqthm/sloop")
        (CF "basis")
        (LF "basis")
        (CF "genfact")
        (LF "genfact")
        (CF "events")
        (LF "events")))
```

We don't compile sloop.lisp because the object file sloop.o does not change very often (i.e. we view it as being a file provided by the Lisp system)

The idea is to save a core image after compiling and loading the "sequential" files, and then run that core image in parallel, compiling one of the remaining 7 files in each job. The top-level call of this compiler is a shell command that calls akcl twice: first for the initial sequential compilation of the first 3 files, using the function COMPILE-NQTHM-SEQ shown above, and then for the parallel compilation of the remaining 7 files.

The file nqthm-par.lisp is the same as existing Boyer-Moore file nqthm.lisp, except that it includes the definition of COMPILE-NQTHM-SEQ and contains the following definition.

```
(DEFUN COMPILE-one-NQTHM-file (filename)
  ;; **** This is all done after saving a core image from (compile-nqthm-seq)
  ;; Hence we may assume that all the proclamations are already around.
  (COMPILE-FILE (EXTEND-FILE-NAME filename FILE-EXTENSION-LISP)))
```

Other than nqthm-par.lisp, the following files comprise the crucial parallel compiler code.

```
----------------------------------------
```
The file `compile-nqthm-shell-script:`

```
compile-nqthm-shell-script:

date
akcl < compile-nqthm-seq.lisp
date
rsh scarab w
rsh elgin w
rsh client13 w
rsh decaf w
rsh client12 w
date
rm /usr/home/kaufmann/compiletest/output/*
rm /usr/home/kaufmann/compiletest/temp/*
akcl < compile-nqthm-par.lisp
date
------------------------------
```
The file `compile-nqthm-seq.lisp:`

```
(when (probe-file "lock-out-others.par")
      (error "Dispatcher won't work -- please remove lock-out-others.par"))

;; nqthm-par.lisp is just like nqthm.lisp, except that instead of
;; COMPILE-NQTHM it has COMPILE-NQTHM-SEQ and COMPILE-ONE-NQTHM-FILE,
;; and also LOAD-NQTHM is omitted and sloop is loaded but not compiled.
(load "nqthm-par.lisp")

(system "date")
(format t "~%Begin compiling sequential part of nqthm~%")
(compile-nqthm-seq)

(system "date")
(format t "~%Save core image~%")
(save "/usr/tmp/nqthm-compilation-midpoint")
------------------------------
```
The file `compile-nqthm-par.lisp:`

```
(load "/local/src/parallel/top.lsp")

(load-dispatcher)

(system "date")
(format t "~%Now starting dispatcher run~%")
(setq *output-completed-string* "Compile job terminated")
(dispatcher :command-name "/usr/tmp/nqthm-compilation-midpoint"
            ;; use nqthm completion function
            :front-end "compile-nqthm-front-end.lisp" :back-end ""
            :kill-if-no-progress 1200
            :job-list '("code-1-a" "code-b-d" "code-e-m" "code-n-r" "code-s-z"
                        "ppr" "io"))
------------------------------
```

The file compile-nqthm-front-end.lisp:

```
(setq sys::*notify-gbc* t)

(defvar *output-completed-string*)
(setq *output-completed-string* "Compile job terminated")

(defun format2 (string &rest args)
  (system
    (concatenate 'string
                 "echo2 '"
                 (apply #'format nil string args)
                 "'")))

(defun format-nqthm-status (string &rest args)
  (apply #'format2 (concatenate 'string *output-completed-string* "~&" string) args))

(SETQ *DEFAULT-NQTHM-PATH* "/usr/home/kaufmann/compiletest/")
------------------------------
```

A typical job, e.g. jobs/code-1-a:

```
(compile-one-nqthm-file "code-1-a")

(cond (*break-enable*
       (if (probe-file "/usr/home/kaufmann/compiletest/code-1-a.o")
           (format-nqthm-status "Success!!")
         (format-nqthm-status "FAILURE -- did not end in a break, but file does not
                              exist.")))
      (t (format-nqthm-status "FAILURE -- ended in a break.")))
----------------------------------------
```

B.2 Results from Using the Parallel Compiler

```
Summary of Times

Total sequential time:  2688 sec.
Total parallel time:   1191 sec.
Real Speedup:  (/ 2688 1191.0)) = 2.26

Parallel run breakdown (times in seconds):

load nqthm-par.lisp               4
Proclaim                        109
compile-load sequential part    226
save core image                 124
[rsh ... w]                      24   {To see which processors were busy -- none were}
load dispatcher code             13
Run dispatcher                  691
---------
Total                          1191
```

An abbreviated shell transcript

```
17:28:48
>Loading nqthm-par.lisp
Finished loading nqthm-par.lisp
17:28:52
```

```
Begin compiling sequential part of nqthm

17:30:41
Completed proclaiming nqthm files.
Loading /local/src/nqthm/sloop.o
start address -T 20e800 Finished loading /local/src/nqthm/sloop.o
Compiling basis.lisp.
.....
start address -T 2b0000 Finished loading events.o
14744

17:34:27
Save core image

17:36:31
[rsh ... w]
17:36:55
AKCL (Austin Kyoto Common Lisp)  Version(1.57) Thu Sep 29 21:27:15 CDT 1988
Contains Enhancements by W. Schelter

>Loading /local/src/parallel/top.lsp
.....
17:37:08

Now starting dispatcher run
NIL

Hosts requested:  ("client12" "scarab" "elgin" "client13" "decaf").
Hosts currently blocked:  ("anderson" "cli" "jingles").

starting>> client12 : code-l-a (job# 1)                               0:00
starting>> scarab : code-b-d (job# 2)                                 0:00
starting>> elgin : code-e-m (job# 3)                                  0:00
starting>> client13 : code-n-r (job# 4)                               0:00
starting>> decaf : code-s-z (job# 5)                                  0:00
completed>> client12 : code-l-a (job# 1)                              0:07
completed>> decaf : code-s-z (job# 5)                                 0:07
starting>> client12 : ppr (job# 6)                                    0:07
starting>> decaf : io (job# 7)                                        0:07
completed>> client12 : ppr (job# 6)                                   0:08
completed>> scarab : code-b-d (job# 2)                                0:09
completed>> elgin : code-e-m (job# 3)                                 0:09
completed>> decaf : io (job# 7)                                       0:10
completed>> client13 : code-n-r (job# 4)                              0:11
T

>Bye.
17:48:39
```

One measure of "efficiency" is in terms of how many total CPU seconds are used in the parallel vs. the sequential run. That is, this measure should be an indication of the overhead in setting up the dispatcher run. We measure this kind of efficiency in (A), below, with a slight variation in (B). In part (C) we measure the actual REAL speedup comparing the parts of the two runs that can actually be made parallel, i.e. the compilation of the files code-1-a, code-b-d, code-e-m, code-n-r, code-s-z, ppr, io.

(A) From the point of view of total CPU seconds used (on all processors).

First we calculate the expected total CPU seconds for a theoretical "optimal" parallel run. More precisely, this is the total seconds for the sequential compilation run together with the additional operations done in the parallel run that don't correspond to actions taken in the sequential run. (Notice that our notion of "additional operations" does not include time required to fire up processes or other overhead incurred in running the dispatcher.)

```
[total sequential run time] + [save core image] + [rsh ... w] + [load dispatcher code] =
2688 + 124 + 24 + 13 =
2849
```

On the other hand, if we note early completions,

```
     completed>> client12 : ppr (job# 6)                                        0:08
     completed>> scarab : code-b-d (job# 2)                                     0:09
     completed>> elgin : code-e-m (job# 3)                                      0:09
     completed>> decaf : io (job# 7)                                            0:10
     completed>> client13 : code-n-r (job# 4)                                   0:11
```

we get the estimated total CPU seconds for the entire actual parallel run by subtracting off the minutes that all but client13 sat idle (note the gross rounding here!):

```
[{total parallel run time} - {dispatcher_time}] + (* 5 {dispatcher_time})
        - (* 60 (3+2+2+1)) =
(- 1191 691) + (* 5 691) - 480 =
3475
```

Estimated actual efficiency from the point of view of total CPU seconds used:

```
(/ 2849.0 3475) = 82%.
```

(B) From the point of view of total CPU seconds used (on all processors), but only for the part of compilation potentially done in parallel

Sequential (Real) Time for code-1-a to the end (note nqthm-par.lisp and nqthm.lisp are similar in load times): Roughly,

```
[sequential total] - {[load nqthm-par.lisp] + [Proclaim]
      + [Compile-load sequential part]} =
(- 2688 (+ 4 109 226)) =
2349

Parallel run (as explained above, roughly 8 minutes less than (* 5 691) ):
(- (* 5 691) (* 60 8)) =
2975

Efficiency:  (/ 2349 2975.0) =
79%
```

(C) From the point of view of looking at REAL TIME, but only for the compilation potentially done in parallel:

```
Sequential run, code-1-a to the end (as shown in (B) above):
2349

Parallel Real Time for running dispatcher:
691

REAL speedup for code-1-a to the end:
(/ 2349 691.0) =
3.40

Efficiency for code-1-a to the end:
(/ 3.40 5) = 68%
```

Appendix C
Code

We give here the contents of the following files in directory /local/src/parallel/: bm.lsp, dispatch.lsp, top.lsp, front-end-with-doc.lsp, and PAR.

```
;; bm.lsp
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;; CODE TO CREATE BOYER-MOORE INPUT FILES ;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;Here's an example of what an input stream might look like (whitespace added here).

#|
;; front-end.lsp
(defun do-event .... )
(defun par-nqthm-top-level ... )
;; [and requisite DEFVARs and auxiliaries]

;; jobs file
;(defun no-io () nil)
;(setq io-fn #'no-io)
;(setq blurb-flg nil)
(defvar *output-completed-string* "Boyer-Moore job terminated")
(setq *start-name* 'DELETE)
(setq *finish-name* 'MEMBER-DELETE)
(setq *start-position* 0)
(par-nqthm-top-level)

;; back-end.lsp
[actual events]
|#

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;; VARIABLE DECLARATIONS ;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defvar *output-completed-string* "Boyer-Moore job terminated")

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;; FLAGS ;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defvar *no-io-in-parallel-flag* nil)

(defvar *delete-jobs-flg* nil)          ;if non-NIL, delete the files in JOBS/.

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;; THE REST ;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun create-jobs-files (infile suc-size jobs-directory-name)
  ;; Here INFILE is the list of nqthm events, SUC-SIZE is the granularity, i.e. the
  ;; number of events to be run each time (after bringing the chronology up to speed),
  ;; and JOBS-DIRECTORY-NAME and OUTPUT-DIRECTORY-NAME are the subdirectories of the
  ;; current directory for the input and output files (respectively).
  (WITH-OPEN-FILE
   (INSTREAM INFILE
           :DIRECTION :INPUT   .
           :IF-DOES-NOT-EXIST :ERROR)
   (let ((start-info (start-info instream suc-size)))
     ;; *** We haven't thought enough about how to handle two jobs with the same name.
     ;; Perhaps we should put something here checking for duplcate file names.
     ;; But probably that's better left for when we deal with books and such.
     (iterate for tail on start-info
           when                          ;a bit of error-checking
           (if (atom (caar tail)) t
             (error "Encountered non-atom in start-names, ~S." (caar tail)))
           collect
           (create-one-job-file (car tail) (cadr tail) jobs-directory-name)))))

(defun start-name-of (form)
  (if (new-dispatch-call form) (car form) (cadr form)))

(defun lemma-form (form)
  ;; This is a recognizer for the class of forms which determine our notion of granularity.
  ;; For example, if CONSTRAIN becomes an event then we should include that too.
  (member (car form) '(prove-lemma lemma) :test #'eq))
```

```
(defun new-dispatch-call (form)
  ;; *** Should be modified when we deal with books and such.
  (or (eq (car form) 'boot-strap)
      (eq (car form) 'note-lib)))

(defun start-info (instream suc-size)
  ;; Returns a list of pairs (<event-name> . n), where n >= 0 indicates the starting
  ;; position (from a boot-strap or note-lib call or the beginning of file) for this
  ;; event-name.  (I could return the position as well, but I want the event-names
  ;; to create nice file names, and then the position ought to be irrelevant.)
  ;;   Notice that every suc will end with a lemma, except when the accumulation process
  ;; is aborted by a boot-strap or note-lib.
  (iterate for position from 1
           with form and i = 0 and suffix = 0 and ready-flg = t and ans
           ;; i is the number of lemmas we've got so far in the suc we're accumulating
           ;; when ready-flg is set we know we're ready to start a new suc
           while (not (eq (setq form (read instream nil a-very-rare-cons))
                          a-very-rare-cons))
           do (progn
                (cond ((not (and (consp form)
                                 (or (null (cdr form))
                                     (consp (cdr form)))))
                        (error " Encountered atom in event file:~& ~S" form))
                       ((new-dispatch-call form)
                        (setq ready-flg nil) ;in case we were already ready anyhow
                        (setq i 0)
                        (setq suffix position)
                        (setq ans (cons (cons (start-name-of form) suffix) ans)))
                       (ready-flg
                        (setq ready-flg nil)
                        (setq ans (cons (cons (start-name-of form) suffix) ans)))
                       (t nil))
                (when (lemma-form form)
                       (setq i (1+ i))
                       (when (= i suc-size)
                             (setq i 0)
                             (setq ready-flg t))))
           finally (return (nreverse ans))))

(defun create-one-job-file
  (start-info finish-info jobs-directory-name
              &aux (job-file-name-from-pair (job-file-name-from-pair start-info)))

  ;; Writes out an input file in the subdirectory JOBS-DIRECTORY-NAME
  ;; which is appropriate for nqthm or pc-nqthm.  If START-INFO is
  ;; (<event-name> . <position>), that input file is intended for
  ;; running events in the file from the position <position> in INFILE
  ;; up to (but not including) FINISH-NAME, with output to be written
  ;; to a file in OUTPUT-DIRECTORY-NAME.  Lemmas before the event
  ;; named <event-name> (after <position>), however, are to be taken
  ;; as axioms.
  ;;   The function actually returns the new input file's name.
  (with-open-file
   (outstream (concatenate 'string jobs-directory-name job-file-name-from-pair)
              :direction :output)
   (when *no-io-in-parallel-flag*
         (format outstream "(defun no-io () nil)~%")
         (format outstream "(setq io-fn #'no-io)~%")
         (format outstream "(setq blurb-flg nil)~%"))
   (format outstream "(defvar *output-completed-string* ~S)~%"
           *output-completed-string*)
   (format outstream "(setq *start-name* '~S)~%" (car start-info))
   (format outstream "(setq *finish-name* '~S)~%" (car finish-info))
   (format outstream "(setq *start-position* ~S)~%" (cdr start-info))
   (format outstream "(par-nqthm-top-level)~%" (cdr start-info)))
  job-file-name-from-pair)

(defun job-file-name-from-pair (start-info)
  ;; Here, as usual, START-INFO is a pair (symbol . number), where symbol is an event-name
  (if (= (cdr start-info) 0)
      (string (car start-info))
    (concatenate 'string (string (car start-info)) "."
                 (prin1-to-string (cdr start-info)))))
```

```
(defun nqthm-completed (file-name)
  ;; returns non-nil if we find the header *output-completed-string* (which isn't then
  ;; immediately followed by end-of-file)
  (cond
   ((not (probe-file file-name))
    (cons 'failure
          (list "UNUSUAL FAILURE:  File ~A should exist, but does not! ~&"
                file-name)))
   (t
    (with-open-file (stream file-name)
                    (iterate with input-string
                             do
                             (if (setq input-string (read-line stream nil nil))
                                 (when (equal input-string *output-completed-string*)
                                       (let ((success-string
                                              (read-line
                                               stream nil
                                               "UNUSUAL FAILURE:  End of status file encountered.")))
                                         (cond
                                          ((and (> (length success-string) 6)
                                                (equal (string-downcase (subseq success-string 0 7))
                                                       "success"))
                                           (return (cons 'success nil)))
                                          (t (return (list 'failure success-string))))))
                               (return nil)))))))

(defun status-file-name-from-string (directory-name job-name)
  (concatenate 'string directory-name job-name ".status"))

(defun current-directory ()
  (directory-namestring (truename "")))

(defun reset-directory (target-directory-name)
  (when (not (equal (current-directory) target-directory-name))
        (sys:chdir target-directory-name)))

(defun do-file-parallel (infile suc-size &key
                                (jobs-directory-name "jobs/")
                                (output-directory-name "output/")
                                (hosts-file-name
                                 (if (probe-file "hosts")
                                     "hosts"
                                   (concatenate 'string *system-parallel-directory*
                                                "hosts.all")))
                                (local-host-first t)
                                (kill-if-no-progress 200)
                                (command-name "pc-nqthm") ;might be "nqthm", etc.
                                (front-end (concatenate 'string *system-parallel-directory*
                                                        "front-end.lsp"))
                                (delay nil)
                                (nice-flag t)
                                &aux (temp-directory-name "temp/")
                                jobs-files (current-directory (current-directory)))

  ;; Runs a Boyer-Moore event list in parallel by creating job files
  ;; with CREATE-JOBS-FILES (each of which calls DO-FILE-WITH-SUCCESS)
  ;; and then calling the dispatcher.  The first four forms in the PROGN
  ;; below relate only to clearing out the input and output subdirectories.
  ;;   Returns T if and only if all jobs cause T to be written to the ".status"
  ;; files.  (Recall also that DO-FILE-WITH-SUCCESS causes T to be thus written
  ;; if and only if all of its events complete successfully.)  If any job fails
  ;; in this sense, the failed jobs are reported to the standard output.
  ;;   Notice that we do the locking on par jobs here because we work with the
  ;; jobs subdirectory even before starting up the dispatcher.

  (unwind-protect
      (progn

        ;; Get control over the current directory right away.
        (check-lock-on-par-jobs)
        (lock-on-par-jobs)

        ;; Set up the jobs.
        (unwind-protect
            (progn

              (chk-file-or-directory-exists front-end)
              (chk-file-or-directory-exists hosts-file-name)
              (chk-file-or-directory-exists jobs-directory-name t)
              (chk-file-or-directory-exists output-directory-name t)
              (chk-file-or-directory-exists temp-directory-name t)
          (when (null delay) (setq delay suc-size))
          (clear-directory jobs-directory-name *clear-query-flg*)
          (setq *current-job-unique-number* 0)
          ;; Create jobs files.
          (format t "~&Creating jobs files...")
          (setq jobs-files (create-jobs-files
                            infile suc-size jobs-directory-name))
          (format t " done.~%~%")
          ;; Run jobs in parallel.
          (reset-directory current-directory))
      (remove-lock-on-par-jobs))

    ;; Call the dispatcher.
    (format t "~&Invoking dispatcher.~&")
    (when (dispatcher :jobs-directory-name jobs-directory-name
                      :output-directory-name output-directory-name
                      :hosts-file-name hosts-file-name
                      :local-host-first local-host-first
                      :kill-if-no-progress kill-if-no-progress
                      :command-name command-name
                      :delay delay
                      :completion-function #'nqthm-completed
                      :front-end front-end
                      :back-end infile
                      :nice-flag nice-flag)
          (format t "~&All events have been run successfully.~%")
          t)
    )

;; Clean up.
(when *delete-jobs-flg*
      (iterate for x in jobs-files
             do (delete-file x)))
(reset-directory current-directory)))
```

```
;; dispatch.lsp

;; This version assumes that nqthm is loaded.  Otherwise one should
;;   (load "/usr/local/src/nqthm/sloop") and
;;   (defmacro iterate (&rest args) `(sloop::sloop ,@args)).
;; There are probably a couple of other functions below that would
;; need to be defined in akcl, e.g. no-duplicatesp.

;; We assume that only one parallel job is being run in the current
;; directory at any one time.

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;; STRUCTURES ;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Here is the only structure we introduce:

(defstruct job
  name number)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;; FLAGS ;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defvar *local-host-warning* t)

(defvar *save-temp-files-flg* nil)

(defvar *kill-dispatcher-upon-seeing-failure* nil)

(defvar *clear-query-flg* nil)          ;used for CLEAR-DIRECTORY

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;; VARIABLE DECLARATIONS ;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defvar *failed-job-names*)

(defvar *system-parallel-directory* "/local/src/parallel/")

(defvar *parkill-command* "/local/bin/parkill")

(defvar *protected-hosts-subdirectory* "protected-hosts/")

; The following might have been in /tmp, but what if other run dispatcher
; jobs at the same time?  So we put the "junk" in the current directory,
; where the locking mechanism should save us.
(defvar *junk-file* "dispatcher-junk.lsp")

(defvar *lock-file-name* "lock-out-others.par")

;; for the run file, e.g. "/local/src/parallel/statistics/runs/wilding.1"
(defvar *long-job-info-filename*)

;; e.g. the number 1 in the pathname above
(defvar *uniq-value*)

(defvar *user-name*)                    ;e.g. "wilding"

(defvar *short-job-info-filename*       ;i.e. "/local/src/parallel/statistics/job-info"
  (concatenate 'string *system-parallel-directory*
               "statistics/job-info"))

(defvar *output-info-subdirectory* "statistics/runs/")

(defvar *start-time*)                   ;set in dispatcher when it's called

(defvar *local-host-name*)              ;used to make the hosts-jobs-alist and to
                                        ;protect against blocking spawns on local host

(defvar *all-valid-host-names* nil)     ;not actually used here, but could be set in an init-file

(defvar *current-job-unique-number* 0)  ;not automatically reset on new dispatcher call

(defvar *file-server-time-disparity*)   ; seconds file server's clock is ahead
                                        ; (could be negative if behind)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;; FILE UTILITIES ;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun chk-file-or-directory-exists (filename &optional directory-flg)
  (when (not (probe-file filename))
        (if directory-flg
            (error "Bad news -- directory ~A doesn't exist!" filename)
          (error "Bad news -- file ~A doesn't exist!" filename)))
  (when (and directory-flg
             (not (equal (aref filename (1- (length filename))) #\/)))
        (error "Bad news -- directory ~A didn't end with a '/' !" filename)))

(defun non-files-in-directory (directory-name list-of-files)
  (iterate for name in list-of-files
           when (or (not (stringp name))
                    (not (probe-file (concatenate 'string directory-name name))))
           collect name))

(defun system-output (command)
  ;; Submits a shell command writes its output to the "junk file".
  ;; Should work if command is in syntax you'd give when typing to the shell.
  ;; Probably don't need to clear *junk-file* first because SYSTEM will direct SOMETHING there.
  (system (concatenate 'string "( " command " ) > " *junk-file*)))

(defun read-one-form (filename)
  ;; returns NIL if file doesn't exist or if the read fails
  (let ((instream (open filename :direction :input :if-does-not-exist nil)))
    (and instream
         (unwind-protect
             (read instream nil nil)
           (close instream)))))

(defun get-system-output (command)
  ;; assumes output is a single line
  (system-output command)
  (with-open-file (infile *junk-file* :direction :input)
                  (read-line infile)))

(defun lines-from (filename)
  ;; Returns the list of lines (as strings) from the given file.
  (with-open-file (stream filename)
                  (iterate with ans and temp
                           when
                           (progn (setq temp (read-line stream nil nil))
                                  (when (null temp) (return ans))
                                  (not (equal (setq temp (string-trim " \t" temp))
                                              "")))
                           collect temp into ans)))

(defun format-to-file (filename string &rest args)
  (with-open-file (outfile filename
                           :direction :output
                           :if-does-not-exist :create
                           :if-exists :new-version)
                  (apply #'format outfile string args)))

(defun princ-to-end-of-file (filename string)
  (with-open-file (stream filename :direction :output
                          :if-exists :append
                          :if-does-not-exist :error)
                  (princ string stream)))

(defun list-directory (directory-name)
  ;;; ***** We should change this to avoid the problem of two jobs dealing
  ;; with the same junk file, once DIRECTORY works right in AKCL.
  ;; Returns the listing (as strings) of the given directory name,
  ;; sorted by time, most recently written ones being at the end.
  (progn (system-output (format nil "ls '~A'" directory-name))
         (lines-from *junk-file*)))

(defun clear-directory (directory-name &optional query &aux files)
  ;; Returns T unless we decide to abort, and then NIL.
  (when (setq files (list-directory directory-name))
        (cond
         ((or (not query)
              (y-or-n-p "Directory ~A not empty!  OK to clear it?"
                        directory-name))
          ;;(iterate for file in files do
          ;;  (delete-file (concatenate 'string directory-name file)))
          (format t "~&Clearing directory ~A.~&" directory-name)
          (system (format nil "rm ~A/*" directory-name)))
         (t (error "Directory ~A not empty!  Quitting...."
                   directory-name)))))

(defun job-file-name (directory-name job)
  (concatenate 'string directory-name (job-name job)))

(defun output-file-name (directory-name job)
  (concatenate 'string directory-name (job-name job) ".output"))
```

```
(defun status-file-name (directory-name job)
  (concatenate 'string directory-name (job-name job) ".status"))

(defun temp-finish-file-name (directory-name job)
  (concatenate 'string directory-name "finish." (prinl-to-string (job-number job))))

(defun temp-status-file-name (directory-name job)
  (concatenate 'string directory-name "status." (prinl-to-string (job-number job))))

(defun temp-output-file-name (directory-name job)
  (concatenate 'string directory-name "output." (prinl-to-string (job-number job))))

(defun check-lock-on-par-jobs ()
  (when (probe-file *lock-file-name*)
        (error "Already locked by ~A!~%~%~
                 [Sometimes bogus lock files aren't removed --~% ~
                  If you REALLY want to remove the lock file (CAREFUL!!!!),~% ~
                  (REMOVE-LOCK-ON-PAR-JOBS).]~%"
                (file-author *lock-file-name*))))

(defun lock-on-par-jobs ()
  (format-to-file *lock-file-name* "locked~%"))

(defun remove-lock-on-par-jobs ()
  (delete-file *lock-file-name*))

(defun blocked-hosts ()
  ;; just for the user, perhaps
  (list-directory (concatenate 'string *system-parallel-directory*
                               *protected-hosts-subdirectory*)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;; OTHER RANDOM UTILITIES ;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun hostname ()
  (get-system-output "hostname"))

(defun username ()
  (get-system-output "whoami"))

(defun set-file-server-time-disparity ()
  (format-to-file *junk-file* "xxx")
  (setq *file-server-time-disparity*
        (- (file-write-date *junk-file*) (get-universal-time))))

(defun get-file-server-time ()
  (+ *file-server-time-disparity* (get-universal-time)))

(defun output-host-job (status host-job)
  ;; Here STATUS is a string
  (let* ((now (- (get-universal-time) *start-time*))
         (string (format nil
                         (concatenate 'string "~&" status
                                      ">> ~A : ~A (job# ~D)~70,0T~D:~D~D~&")
                         (car host-job) (job-name (cdr host-job))
                         (job-number (cdr host-job)) (floor now 3600)
                         (mod (floor now 600) 6) (mod (floor now 60) 10))))
    (output-long-job-info string)
    (princ string)
    (force-output)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;; METERING STUFF ;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun digits-to-string (number size)
  (subseq
   (prinl-to-string (+ (expt 10 size) number))
   1 (1+ size)))
```

```
(defun initialize-job-info ()
  (setq *start-time* (get-universal-time))
  (set-file-server-time-disparity)
  (setq *local-host-name* (hostname))
  (setq *user-name* (username))
  (setq *uniq-value* (mod (get-file-server-time) 1000000))
  (setq *long-job-info-filename* (concatenate 'string *system-parallel-directory*
                                              *output-info-subdirectory*
                                              *user-name* "."
                                              (digits-to-string *uniq-value* 6)))
  (let ((temp))
    (unwind-protect
        (when (not (setq temp (open *long-job-info-filename*
                                    :direction :output
                                    :if-exists nil
                                    :if-does-not-exist :create)))
              (error (concatenate 'string *long-job-info-filename* " exists")))
      (if temp (close temp))))

  (when (not (probe-file *short-job-info-filename*))
        (error (concatenate 'string *short-job-info-filename*
                            " does not exist "))))

(defun output-long-job-info (string)
  (princ-to-end-of-file *long-job-info-filename* string))

(defun output-short-job-info (string)
  (princ-to-end-of-file *short-job-info-filename* string))

(defun job-info-message (message host-names)
  ;; e.g. if message is PAR-START and host-names is ("cli" "client12" "anderson"),
  ;; and anderson is currently the only blocked host, we get:
  ;;      +++02/17/89 16:26:53 PAR-START kaufmann client12 1
  ;;      hosts: ("cli" "client12" "anderson")
  ;;      blocked: ("anderson")
  (multiple-value-bind
   (sec min hour date month year)
   (get-decoded-time)
   (format nil
           "~&+++~D~D/~D~D/~D~D ~D~D:~D~D:~D~D ~A ~A ~A ~D~%hosts: ~S~%blocked: ~S~%"
           (floor month 10) (mod month 10) ; AKCL bug(?) makes us do contortions
           (floor date 10) (mod date 10)
           (mod (floor year 10) 10) (mod year 10)
           (floor hour 10) (mod hour 10)
           (floor min 10) (mod min 10)
           (floor sec 10) (mod sec 10)
           message *user-name* *local-host-name* *uniq-value*
           host-names
           (list-directory (concatenate 'string *system-parallel-directory*
                                        *protected-hosts-subdirectory*)))))

(defun output-job-info (message host-names)
  (let ((mess (job-info-message message host-names)))
    (output-short-job-info mess)
    (output-long-job-info mess)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;; MAIN DISPATCHER CODE ;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; The following function, DISPATCHER,

;; takes input files from JOBS-DIRECTORY-NAME
;; and runs COMMAND-NAME on each of these files,
;; using the hosts in HOSTS-FILE-NAME
;; by employing a scheduler which engages roughly every DELAY seconds,
;; and sends the output from each job to the directory TEMP-DIRECTORY-NAME
;; (using the same filenames as the input "jobs" files).  If the job
;; completes, then the output file in TEMP-DIRECTORY-NAME is copied
;; to OUTPUT-DIRECTORY-NAME.  Note that COMPLETION-FUNCTION should return
;; either NIL, which indicates that the job died in the middle somehow,
;; or else a pair -- either ('success . <message>) or (<other> . <message>),
;; where <message> is either NIL or a list suitable for FORMAT, i.e. a list
;; of the form (string . args).

;; The auxiliary (&AUX) variables may be thought of as follows.
;; HOSTS-JOBS-ALIST: An association list with one pair (HOST . JOB)
;;      for each occurrence of HOST in the HOSTS-FILE-NAME.  The JOB
;;      component is either NIL, which indicates that the HOST is
;;      currently idle, or else is a job built using the name of the input file
;;      currently assigned (via the rsh command) to HOST.
;; JOBS-UNASSIGNED: The list of input files which have not yet been
;;      assigned to any host.
```

```
(defun make-local-host-first (host-names local-host-name)
  (iterate for x in host-names
           with ans
           when (not (equal x local-host-name))
           collect x into ans
           finally (return (nconc (iterate for i from 1 to (- (length host-names) (length ans))
                                           collect local-host-name)
                                  ans))))

(defun make-hosts-jobs-alist (host-names local-host-first)
  ;; Note that *local-host-name* is already initialized by the time this is called.
  (cond
   ((not (consp host-names))
    (error "Empty list of hosts"))
   (t
    (when local-host-first (setq host-names (make-local-host-first host-names *local-host-name*)))
    (when (and *local-host-warning*
               (not (member *local-host-name* host-names :test #'equal)))
          (format t "~&WARNING:  This run has been set up such that the local host, ~
                     ~A, is NOT~&in the host list.~&" *local-host-name*))
    (iterate for host-name in host-names
             when (cond
                   ((or (null *all-valid-host-names*)
                        (member host-name *all-valid-host-names* :test 'equal))
                    t)
                   (t (format t "~&WARNING:  Host name ~A not found in *all-valid-host-names*.~&"
                              host-name)
                      nil))
             collect (cons host-name nil)))))

(defun dispatcher (&key (jobs-directory-name "jobs/")
                        (output-directory-name "output/")
                        (hosts-file-name "hosts")
                        (local-host-first t)
                        (delay 15)
                        (kill-if-no-progress 600)
                        (command-name "pc-nqthm")
                        (completion-function #'nqthm-completed)
                        (front-end (concatenate 'string *system-parallel-directory*
                                                "front-end.lsp"))
                        (back-end (concatenate 'string *system-parallel-directory*
                                               "back-end.lsp"))
                        (nice-flag t)
                        (job-list nil)
                        &aux
                        (temp-directory-name "temp/") ;auxiliary because we use it in the shell command
                        (time-since-last-progress-check 0)
                        temp-time
                        (bad-jobs nil)
                        *start-time* *local-host-name* *user-name* *uniq-value* *long-job-info-filename*
                        (*failed-job-names* nil)
                        hosts-jobs-alist
                        jobs-unassigned
                        (finished-normally nil))
  ;; set *start-time*, *local-host-name*, *user-name*, *uniq-value*, and *long-job-info-filename*
  ;; NOTE:  Must do these early for the OUTPUT-JOB-INFO in the unwind-protect
  (format t "~&Initializing job information...")
  (initialize-job-info)
  (format t " done.~&")

  (unwind-protect
      (progn

        (check-lock-on-par-jobs)
        (lock-on-par-jobs)

        (chk-file-or-directory-exists jobs-directory-name t)
        (chk-file-or-directory-exists output-directory-name t)
        (chk-file-or-directory-exists temp-directory-name t)

        (clear-directory output-directory-name *clear-query-flg*)
        (clear-directory temp-directory-name *clear-query-flg*)
        (terpri)
```

```lisp
   (when job-list
       (let ((bad-files
               (non-files-in-directory jobs-directory-name job-list)))
        (when bad-files
                (error "Bad Job File Names Provided:~%~A" bad-files))))

   ;; Set up local non-special variables
   (when nice-flag
       (setq command-name (concatenate 'string "nice " command-name)))
   (setq jobs-unassigned
       (if job-list
         job-list
        (list-directory jobs-directory-name)))
   (setq hosts-jobs-alist           ;uses *local-host-name*
       (make-hosts-jobs-alist
        (lines-from hosts-file-name)
        local-host-first))

   (format t "~&Hosts requested:  ~S.~%"
        (iterate for host-job in hosts-jobs-alist
              collect (car host-job)))
   (format t "~&Hosts currently blocked:  ~S.~%~%"
        (let ((x (remove *local-host-name* (blocked-hosts) :test #'equal)))
         (or x 'none)))
   (output-job-info "PAR-START"
              (iterate for host-job in hosts-jobs-alist
                    collect (car host-job)))

   (loop
    ;; the first form is ignored for time-since-last-progress-check, but that's OK.

    (let ((temp (update-completed-jobs-records
                completion-function hosts-jobs-alist jobs-unassigned
                temp-directory-name output-directory-name)))
     (setq hosts-jobs-alist (car temp))
     (setq jobs-unassigned (cdr temp)))

    (when (and *kill-dispatcher-upon-seeing-failure* *failed-job-names*)
       (format t "~%~%********* KILLING ENTIRE JOB -- if you don't like abortion,~%~
               (SETQ *KILL-DISPATCHER-UPON-SEEING-FAILURE* NIL).")
       (return nil))
    (when (> time-since-last-progress-check kill-if-no-progress)
       (setq time-since-last-progress-check 0)
       (when (setq bad-jobs (find-bombed-jobs hosts-jobs-alist
                                  kill-if-no-progress temp-directory-name))
           (kill-processes bad-jobs temp-directory-name)
           (setq hosts-jobs-alist
               (remove-processes-from-hosts-jobs-alist
                hosts-jobs-alist bad-jobs))
           (setq jobs-unassigned
               (append jobs-unassigned
                     (iterate for job in bad-jobs
                           collect (job-name job))))))

    (setq temp-time (get-universal-time))

    (let ((temp (assign-jobs hosts-jobs-alist jobs-unassigned command-name
                     jobs-directory-name front-end back-end)))
     (setq hosts-jobs-alist (car temp))
     (setq jobs-unassigned (cdr temp))
     (if (all-jobs-completed hosts-jobs-alist)
       (return t)
      (sleep delay)))

    (setq time-since-last-progress-check (+ time-since-last-progress-check
                               (- (get-universal-time) temp-time))))
   ;; end of loop
   (setq finished-normally t)

   (cond (*failed-job-names*
       (format t "~%~%FAILED JOBS ~S~&" *failed-job-names*)
       nil)
       (jobs-unassigned
       (format t "~%~%JOBS LEFT TO BE DONE - NO HOSTS AVAILABLE!!")
       (format t "~&~%jobs left to do:~%  ~S" jobs-unassigned)
       nil)
       (t t)))
(remove-lock-on-par-jobs)
(output-job-info "PAR-END"
           (iterate for host-job in hosts-jobs-alist
                  collect (car host-job)))
(when (not finished-normally)
    (format t "~%Now killing any outstanding --par jobs.~%")
    (system *parkill-command*))))
```

```
(defun find-bombed-jobs (hosts-jobs-alist kill-if-no-progress temp-directory-name)
  ;; returns a list of all jobs which have made no progress since "kill-time"
  (let ((kill-time (-  (get-file-server-time)  kill-if-no-progress)))
    (iterate for host-job in hosts-jobs-alist
            when (and
                  (cdr host-job)
                  (let ((temp-output-file (temp-output-file-name temp-directory-name (cdr host-job))))
                    (or
                     (not (probe-file temp-output-file))
                     (< (file-write-date temp-output-file) kill-time))))
            collect (cdr host-job))))

(defun kill (pid)
  (system (concatenate 'string "kill -KILL " (prin1-to-string pid))))

(defun get-pid (job temp-directory)
  ;; pid is first line of temp output file
  (iterate for i from 1 to 2            ;could be greater than 2 if we want to try more often
          with ans
          until ans
          do
          (cond
           ((setq ans (read-one-form (temp-output-file-name temp-directory job)))
            (return ans))
           (t (format t "~&Trying again to read pid of ~S~&" job)
              (sleep 3)))
          finally (error "Unable to get pid of ~S" job)))

(defun kill-processes (bad-jobs temp-directory)
  (iterate for job in bad-jobs
          do (kill (get-pid job temp-directory))))

(defun remove-processes-from-hosts-jobs-alist (hosts-jobs-alist bad-jobs)
  ;; Note that bad-jobs is in same order as hosts-jobs-alist

  (iterate for host-job in hosts-jobs-alist
          collect
          (cond
           ((null bad-jobs) host-job)
           ((and
             (cdr host-job)
             (eq (job-number (car bad-jobs)) (job-number (cdr host-job))))
            (setq bad-jobs (cdr bad-jobs))
            (output-host-job "*** KILLED" host-job)
            (cons (car host-job) nil))
           (t host-job))))

(defun block-spawn (host)
  (and
   (not (equal host *local-host-name*))
   (probe-file (concatenate 'string *system-parallel-directory*
                          *protected-hosts-subdirectory* host))))

(defun assign-jobs
  (hosts-jobs-alist jobs-unassigned command-name jobs-directory-name
                   front-end back-end)
  ;;   This function assumes that updating of HOSTS-JOBS-ALIST (to reflect completion
  ;; of jobs) has already been performed.  That is:  it's not this function's job
  ;; to do that updating.  At this point we already know that if a job may be
  ;; assigned to the CAR of the pair then the CDR of the pair is NIL, and vice-versa.
  ;;   This returns (CONS <new hosts-jobs-alist> <new-jobs-unassigned>).  Its side
  ;; effect is to assign new jobs (from JOBS-UNASSIGNED) to those hosts
  ;; which are currently idle, additionally instructing the
  ;; hosts to create .finish files upon completion.

  (cons (iterate for pair in hosts-jobs-alist
         with next-job
         collect
         (cond
          ((null jobs-unassigned) pair)
          ((and (null (cdr pair))
                (not (block-spawn (car pair))))
           (setq next-job (make-job :name (car jobs-unassigned)
                                    :number (setq *current-job-unique-number*
                                                 (1+ *current-job-unique-number*))))
           (setq jobs-unassigned (cdr jobs-unassigned))
           (system (system-job-command (car pair) next-job command-name
                                       jobs-directory-name front-end back-end))
           (output-host-job "starting" (setq pair (cons (car pair) next-job)))
           pair)
          (t pair)))
        jobs-unassigned))
```

```
(defun system-job-command (host-name new-job command-name jobs-directory-name front-end back-end)
  ;; Grotesque, but it works.  Notice that \ appears as \\ inside quotes (" ... "), and
  ;; the single quotes are there to protect names with "funny characters" like # from the
  ;; shell command processor.
  (format nil "/local/bin/parcsh ~A \\''~A'\\' \\''~A'\\' \\''~A'\\' \\''~A'\\' \\''~A'\\' \\''~A'\\' &"
          (concatenate 'string *system-parallel-directory* "PAR")
          host-name
          command-name
          (job-number new-job)
          front-end
          (job-file-name jobs-directory-name new-job)
          back-end))

(defun job-completed (job temp-directory-name)
  (probe-file (temp-finish-file-name temp-directory-name job)))

(defun move-temp-files-to-output (job temp-directory-name output-directory-name)
  ;; Moves the output and status files from the temp directory to the output directory.
  ;; If the flag *save-temp-files-flg* is non-nil then this does a copy instead of a
  ;; move (for debugging only, probably).
  (system (format nil
                  (if *save-temp-files-flg*
                      "cp '~A' '~A' ; cp '~A' '~A'"
                    "mv '~A' '~A' ; mv '~A' '~A'")
                  (temp-output-file-name temp-directory-name job)
                  (output-file-name output-directory-name job)
                  (temp-status-file-name temp-directory-name job)
                  (status-file-name output-directory-name job))))

(defun update-completed-jobs-records
  (completion-function hosts-jobs-alist jobs-unassigned temp-directory-name
                 output-directory-name &aux success-status)
  ;; Returns new versions of HOSTS-JOBS-ALIST and JOBS-UNASSIGNED to reflect
  ;; known job completions (and failures).
  (cons
   (iterate for pair in hosts-jobs-alist
            collect
            (if (cdr pair)
                (cond
                 ((job-completed (cdr pair) temp-directory-name)
                  (if (setq success-status
                            (funcall completion-function
                                     (temp-status-file-name
                                      temp-directory-name (cdr pair))))
                      (progn
                        (output-host-job "completed" pair)
                        (when (not (eq (car success-status) 'success))
                              (setq *failed-job-names*
                                    (cons (job-name (cdr pair)) *failed-job-names*)))
                        (when (cdr success-status)
                              (fresh-line)
                              (format t "  (Job #~D) " (job-number (cdr pair)))
                              (apply #'format t (cdr success-status))
                              (fresh-line))
                        (move-temp-files-to-output (cdr pair) temp-directory-name
                                                   output-directory-name))
                    (progn
                      (output-host-job "***NOT completed" pair)
                      (setq jobs-unassigned (append jobs-unassigned
                                                    (list (job-name (cdr pair)))))))
                  (cons (car pair) nil))

                 (t pair))
              pair))
   jobs-unassigned))

(defun all-jobs-completed (hosts-jobs-alist)
  (iterate for x in hosts-jobs-alist
           always (null (cdr x))))
```

```lisp
;; top.lsp

;; Most of this file is pilfered (and modified) from an rcl file, which
;; in turn borrows freely from Boyer and Moore's file nqthm.lisp.

(defun nqthm-loaded ()
  (fboundp 'iterate))

(when (not (nqthm-loaded))
      (load "/usr/local/src/parallel/from-nqthm"))

(defvar dispatcher-code-files
  '("/local/src/parallel/dispatch" "/local/src/parallel/bm"))

(defun already-compiledp (filename &aux
                                   (lisp-filename (concatenate 'string filename ".lsp"))
                                   (bin-filename (concatenate 'string filename ".o")))
  (and (probe-file lisp-filename)
       (probe-file bin-filename)
       (< (file-write-date lisp-filename) (file-write-date bin-filename))))

(defun first-file-to-compile (filenames)
  (iterate for name in filenames
           when (not (already-compiledp name))
           do (return name)))

(DEFUN COMPILE-dispatcher (&aux (first-file-to-compile (first-file-to-compile dispatcher-code-files)))
  (if (null first-file-to-compile)
      (format t "~&All dispatcher files are already compiled.~&")
    (FLET ((LF (N)
               (LOAD (concatenate 'string n ".o")))
           (CF (N)
               (COMPILE-FILE (concatenate 'string n ".lsp"))))
          ;; could have PROCLAIM form here, as in nqthm
          (iterate for file in dispatcher-code-files
                   with compile-flg
                   do (cond (compile-flg
                             (CF file) (LF file))
                            ((equalp file first-file-to-compile)
                             (setq compile-flg t)
                             (CF file) (LF file))
                            (t (LF file)))))))

;  Invoking (load-dispatcher) is all it takes to build a runnable version of
;  this system, assuming that you have compiled it.
(defun load-dispatcher (&aux badfile)
  (when (setq badfile (first-file-to-compile dispatcher-code-files))
        (format t "WARNING:  The file ~A should be compiled."
                badfile))
  (FLET ((LF (N)
             (LOAD (concatenate 'string n ".o"))))
        (iterate for file in dispatcher-code-files
                 do (LF file))))
```

```lisp
;; /local/src/parallel/front-end-with-doc.lsp

(setq *break-enable* nil)
(setq sys::*notify-gbc* t)

;;;;; Values to be input for particular job:
(defvar *start-position*)                 ;position of applicable note-lib or boot-strap
(defvar *start-name*)                     ;name of first event to be possibly proved
(defvar *finish-name*)                    ;name of event which terminates the job (maybe NIL, for EOF)
;;;;;

;; The following might be modified in the particular input file.
(defvar *par-directory-name* "/usr/home/par/")

(defvar *event-index* 0)                  ;which event we're currently looking at

(defvar *axiom-stage* t)                  ;when t, we should replace PROVE-LEMMAs by ADD-AXIOMs

(defvar *last-event-name*)                ;name of event most recently read from the input stream

;(sleep 30)

(defun do-event (form)
  (setq *event-index* (1+ *event-index*))
  (setq *last-event-name* (cadr form))
  (cond
   ((< *event-index* *start-position*)
    t)
   (t
    (when *axiom-stage*
          (cond
           ((eq (cadr form) *start-name*)
            (setq *axiom-stage* nil))
           ((eq (car form) 'prove-lemma)
            (setq form
                  `(add-axiom
                    ,(cadr form) ,(caddr form) ,(cadddr form))))
           ((eq (car form) 'lemma)
            (setq form
                  `(axiom
                    ,(cadr form) ,(caddr form) ,(cadddr form))))))
    (ppr form nil)
    (terpri nil)
    (eval form))))

(defun format2 (string &rest args)
  ;; notice that the call to echo2 guarantees that we'll overwrite the err
  (system
    (concatenate 'string
                 "echo2 '"
                 (apply #'format nil string args)
                 "'")))

(defun format-nqthm-status (string &rest args)
  (apply #'format2 (concatenate 'string *output-completed-string* "~&" string) args))

(defun par-nqthm-top-level (&aux next-par-form init success)
  (unwind-protect
      (loop
       (setq init nil)
       (setq next-par-form (read *standard-input* nil a-very-rare-cons))
       (setq init t)
       (if (or (eq next-par-form a-very-rare-cons)
               (and *finish-name* (eq (cadr next-par-form) *finish-name*)))
           (return (setq success t))
         (or (prog1 (print (do-event next-par-form))
               (terpri nil) (terpri nil))
             (return (setq success nil)))))
    (cond
     ((null init)
      (format-nqthm-status "Failure:  Unsuccessful read after ~S" (cadr next-par-form)))
     (success
      (format-nqthm-status "Success!!"))
     (t (format-nqthm-status "  FAILURE:  The event ~S failed."
                             *last-event-name*)))
    (bye (if success 0 1))))
```

```
:: PAR

#!/bin/csh
# par [host-name] [command-name] [unique_number] [file1] [file2] [file3]
# Sends process number of the PAR call followed by standard output of rsh, all to the standard output
echo $$ > temp/output.$3
(cat $4 $5 $6 | rsh $1 $2 '; echo2 $status' >> temp/output.$3) >& temp/status.$3
echo " " > temp/finish.$3
```