



damien.francois@uclouvain.be
UCL/CISM - FNRS/CÉCI



An introduction to **checkpointing** for scientific applications

November 2013
CISM/CÉCI training session





What is
checkpointing



Without checkpointing:

```
$ ./count  
1  
2  
3^C  
$ ./count  
1  
2  
3
```

With checkpointing:

```
$ ./count  
1  
2  
3^C  
$ ./count  
4  
5  
6
```

Without checkpointing:

Checkpointing:

'saving' a computation
so that it can be resumed later
(rather than started again)

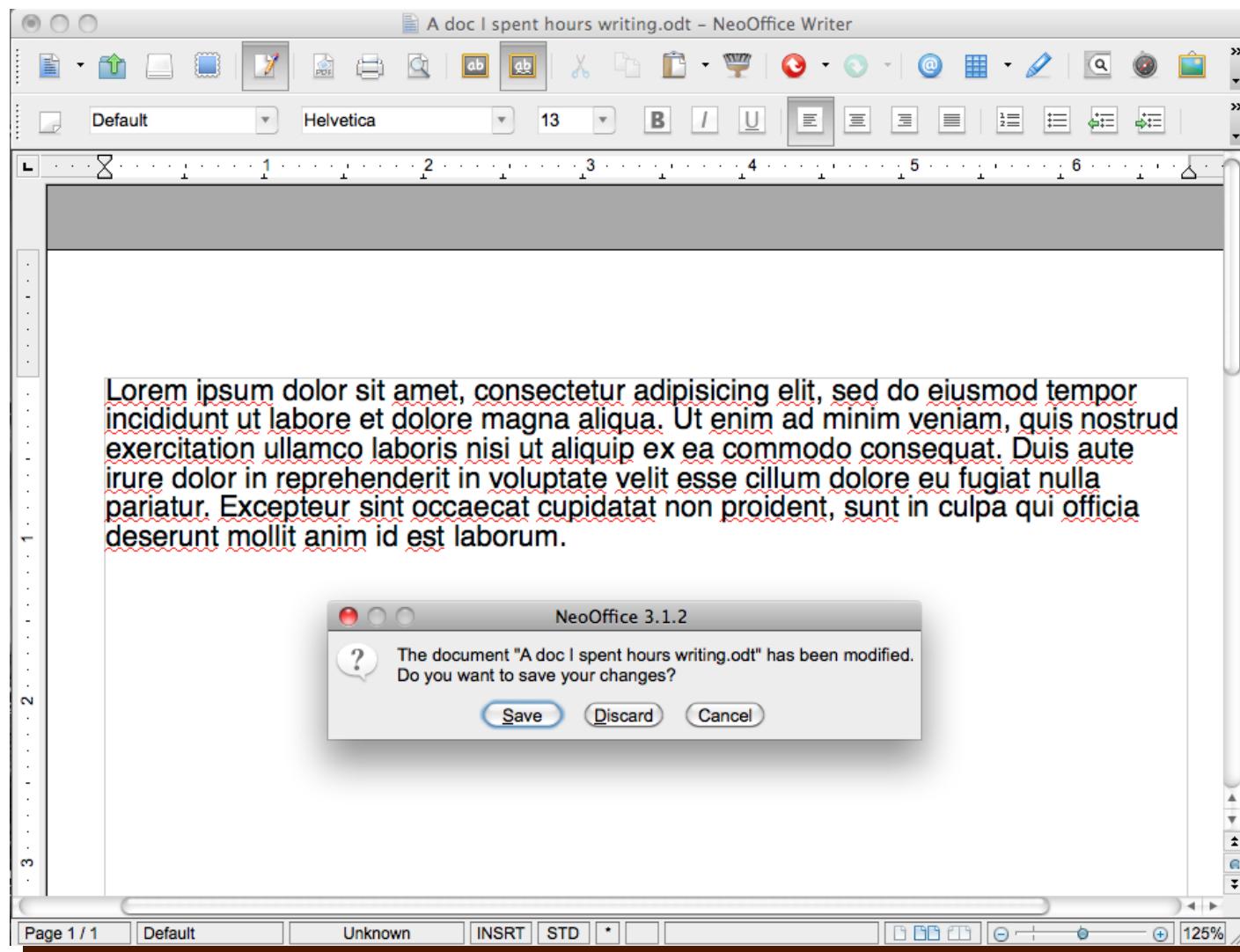
With checkpointing:



Why do we need
checkpointing



Imagine a text editor without 'checkpointing' ...



Goals of checkpointing in HPC:

1. Fit in time constraints
2. Debugging, monitoring
3. Cope with NODE_FAILs
4. Gang scheduling and preemption

The idea:

Save the program state
every time a checkpoint is encountered
and restart from there upon (un)planned stop
rather than bootstrap again from scratch

Values in variables
Open files
...

Position in the code
Signal or event
...

starting loops at iteration 0
creating tmp files
...

The key questions ...

Transparency for developer	Portability to other systems	Size of state to save	Checkpointing overhead
Do I need to write a lot of additional code ?	Can I stop on one system and restart on another ?	How many GB of disk does it require ?	How many FLOPs lost to ensure checkpointing ?

Who's in charge of all that ?

	Transparency for developer	Portability to other systems	Size of state to save	Checkpointing overhead
the application itself	--	+++	--	-
a library	-	++	--	-
the compiler	+	++	-	+
a run-time	+	+	++	+
the OS	++	-	++	++
the hardware	+++	--	+++	+++

Today's agenda:

1 How to make your program checkpoint-able

- > concepts and examples

- > recipes (design patterns)

Slurm integration

2 How to make someone else's program checkpoint-able

- > BLCR

- > DMTCP



Part One: Checkpointing when you have the code

So you can play

On hmem: ~dfr/checkpoint.tgz





1

Making a program checkpoint-able by saving its state every iteration and looking for a state file on startup.

```
#! /bin/env python

from time import sleep

the_start = 1
the_end = 10

for i in range(the_start, the_end):
    # Heavy computation
    print i
    sleep(1)
```

1. dfr@manneback (ssh)

```
#! /bin/env python

from time import sleep

try:
    # Try to recover current state
    with open('state', 'r') as file:
        the_start = int(file.read())
except:
    # Otherwise bootstrap at 1
    the_start = 1

the_end = 10

for i in range(the_start, the_end):
    # Save current state
    with open('state', 'w') as file:
        file.write(str(i))

    # Heavy computations
    print i
    sleep(1)
```

Python recipe

count.py

1,1

All crcount.py

22,5

All

```
# R --slave --vanilla < count.R
the_start <- 1
the_end <- 10
for (i in seq(the_start, the_end)){
  print(i)
  Sys.sleep(1)
}

# Try to recover current state
the_start<-try(as.integer(read.table("state")),
               silent=TRUE )

# Otherwise bootstrap at 1
if (class(the_start) == "try-error")
  the_start <- 1

the_end <- 10

for (i in seq(the_start, the_end)){
  # Save current state
  write.table(i, "state", col.names=FALSE,
              row.names=FALSE)

  # Heavy computations
  print(i)
  Sys.sleep(1)
}
```

R recipe

count.R 5,0-1 All crcount.R 4,1 All

"crcount.R" 20L, 438C

Octave recipe

```
! gfortran count.f90 -o count && ./count
program count
integer :: i, the_start, the_end

the_start = 1
the_end = 10

do i = the_start, the_end
  write(*, '(i2)' ) i
  call Sleep(1)
end do

end program count
~
```

```
! gfortran crcount.f90 -o crcount && ./crcount
program crcount
integer :: i, the_start, the_end
integer :: n, stat

! Try to recover current state
open (1, file='state', status='old', &
      action='READ', iostat=stat)
if (stat .eq. 0) then
  read(1,*), the_start
else
  ! Otherwise bootstrap at 1
  the_start = 1
end if
close(1)

the_end = 10

do i = the_start, the_end
  ! Save current state
  open (1, file='state', status='REPLACE', &
        action='WRITE', iostat=stat)
  write(1, '(i2)' ) i
  close(1)

  ! Heavy computations
  write(*, '(i2)' ) i
  call Sleep(1)
end do
end program crcount
```

count.f90 6,1 All crcount.f90 19,0-1 All

Fortran recipe

1. dfr@manneback (ssh)

```
// gcc count.c -o count && ./count
#include <stdio.h>
void main()
{
    int i, the_start, the_end;

    the_start = 1;
    the_end = 10;

    for (i=the_start; i<=the_end; i++)
    {
        printf("%d\n", i);
        sleep(1);
    }
}

// gcc crcount.c -o crcount && ./crcount
#include <stdio.h>
void main()
{
    int i, the_start, the_end;
    FILE * file;

    // Try to recover current state
    file = fopen("state", "r");
    if (file)
    {
        fscanf(file, "%d", &the_start);
        fclose(file);
    }
    else
    {
        // Otherwise bootstrap at 1
        the_start = 1;
    }

    the_end = 10;

    for (i=the_start; i<=the_end; i++)
    {
        // Save current state
        file = fopen("state", "w");
        fprintf(file, "%d", i);
        fclose(file);

        // Heavy computations
        printf("%d\n", i);
        sleep(1);
    }
}
```

C recipe

count.c 12,5 All crcount.c 3,1 Top

The general recipe

1. Look for a state file

(name can be hardcoded, or,
better, passed as parameter)

2. If found, then restore state

(initialize all variables with content
of the file state)

Else, bootstrap (create initial state)

3. Periodically save the state

In the previous example : The state is just an integer

Periodically means at each iteration

2

Using UNIX signals to reduce overhead : do not save the state at each iteration -- wait for the signal.

UNIX processes can receive 'signals' from the user, the OS, or another process

SIGHUP	1	Exit	Hangup
SIGINT	2	Exit	Interrupt
SIGQUIT	3	Core	Quit
SIGILL	4	Core	Illegal Instruction
SIGTRAP	5	Core	Trace/Breakpoint Trap
SIGABRT	6	Core	Abort
SIGEMT	7	Core	Emulation Trap
SIGFPE	8	Core	Arithmetic Exception
SIGKILL	9	Exit	Killed
SIGBUS	10	Core	Bus Error
SIGSEGV	11	Core	Segmentation Fault
SIGSYS	12	Core	Bad System Call
SIGPIPE	13	Exit	Broken Pipe
SIGALRM	14	Exit	Alarm Clock
SIGTERM	15	Exit	Terminated
SIGUSR1	16	Exit	User Signal 1
SIGUSR2	17	Exit	User Signal 2
SIGCHLD	18	Ignore	Child Status
SIGPWR	19	Ignore	Power Fail/Restart
SIGWINCH	20	Ignore	Window Size Change
SIGURG	21	Ignore	Urgent Socket Condition
SIGPOLL	22	Ignore	Socket I/O Possible
SIGSTOP	23	Stop	Stopped (signal)
SIGTSTP	24	Stop	Stopped (user)
SIGCONT	25	Ignore	Continued
SIGTTIN	26	Stop	Stopped (tty input)
SIGTTOU	27	Stop	Stopped (tty output)
SIGVTALRM	28	Exit	Virtual Timer Expired
SIGPROF	29	Exit	Profiling Timer Expired
SIGXCPU	30	Core	CPU time limit exceeded
SIGXFSZ	31	Core	File size limit exceeded
SIGWAITING	32	Ignore	All LWPs blocked
SIGLWP	33	Ignore	Virtual Interprocessor Interrupt for Threads Library
SIGAIO	34	Ignore	Asynchronous I/O

UNIX processes can receive 'signals' from the user, the OS, or another process

^C	SIGHUP	1	Exit	Hangup	—	kill -9
^D	SIGINT	2	Exit	Interrupt	—	kill
	SIGQUIT	3	Core	Quit	—	fg , bg
	SIGILL	4	Core	Illegal Instruction	—	
	SIGTRAP	5	Core	Trace/Breakpoint Trap	—	
	SIGABRT	6	Core	Abort	—	
	SIGEMT	7	Core	Emulation Trap	—	
	SIGFPE	8	Core	Arithmetic Exception	—	
	SIGKILL	9	Exit	Killed	—	
	SIGBUS	10	Core	Bus Error	—	
	SIGSEGV	11	Core	Segmentation Fault	—	
	SIGSYS	12	Core	Bad System Call	—	
	SIGPIPE	13	Exit	Broken Pipe	—	
	SIGALRM	14	Exit	Alarm Clock	—	
	SIGTERM	15	Exit	Terminated	—	
	SIGUSR1	16	Exit	User Signal 1	—	
	SIGUSR2	17	Exit	User Signal 2	—	
^Z	SIGCHLD	18	Ignore	Child Status	—	
	SIGPWR	19	Ignore	Power Fail/Restart	—	
	SIGWINCH	20	Ignore	Window Size Change	—	
	SIGURG	21	Ignore	Urgent Socket Condition	—	
	SIGPOLL	22	Ignore	Socket I/O Possible	—	
	SIGSTOP	23	Stop	Stopped (signal)	—	
	SIGTSTP	24	Stop	Stopped (user)	—	
	SIGCONT	25	Ignore	Continued	—	
	SIGTTIN	26	Stop	Stopped (tty input)	—	
	SIGTTOU	27	Stop	Stopped (tty output)	—	
	SIGVTALRM	28	Exit	Virtual Timer Expired	—	
	SIGPROF	29	Exit	Profiling Timer Expired	—	
	SIGXCPU	30	Core	CPU time limit exceeded	—	
	SIGXFSZ	31	Core	File size limit exceeded	—	
	SIGWAITING	32	Ignore	All LWPs blocked	—	
	SIGLWP	33	Ignore	Virtual Interprocessor Interrupt for Threads Library	—	
	SIGAIO	34	Ignore	Asynchronous I/O	—	

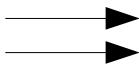
UNIX processes can receive 'signals' from the user, the OS, or another process

e.g.

SIGHUP	1	Exit	Hangup
SIGINT	2	Exit	Interrupt
SIGQUIT	3	Core	Quit
SIGILL	4	Core	Illegal Instruction
SIGTRAP	5	Core	Trace/Breakpoint Trap
SIGABRT	6	Core	Abort
SIGEMT	7	Core	Emulation Trap
SIGFPE	8	Core	Arithmetic Exception
SIGKILL	9	Exit	Killed
SIGBUS	10	Core	Bus Error
SIGSEGV	11	Core	Segmentation Fault
SIGSYS	12	Core	Bad System Call
SIGPIPE	13	Exit	Broken Pipe
SIGALRM	14	Exit	Alarm Clock
SIGTERM	15	Exit	Terminated
SIGUSR1	16	Exit	User Signal 1
SIGUSR2	17	Exit	User Signal 2
SIGCHLD	18	Ignore	Child Status
SIGPWR	19	Ignore	Power Fail/Restart
SIGWINCH	20	Ignore	Window Size Change
SIGURG	21	Ignore	Urgent Socket Condition
SIGPOLL	22	Ignore	Socket I/O Possible
SIGSTOP	23	Stop	Stopped (signal)
SIGTSTP	24	Stop	Stopped (user)
SIGCONT	25	Ignore	Continued
SIGTTIN	26	Stop	Stopped (tty input)
SIGTTOU	27	Stop	Stopped (tty output)
SIGVTALRM	28	Exit	Virtual Timer Expired
SIGPROF	29	Exit	Profiling Timer Expired
SIGXCPU	30	Core	CPU time limit exceeded
SIGXFSZ	31	Core	File size limit exceeded
SIGWAITING	32	Ignore	All LWPs blocked
SIGLWP	33	Ignore	Virtual Interprocessor Interrupt for Threads Library
SIGAIO	34	Ignore	Asynchronous I/O

UNIX processes can receive 'signals' from the user, the OS, or another process

e.g.



SIGHUP	1	Exit	Hangup
SIGINT	2	Exit	Interrupt
SIGQUIT	3	Core	Quit
SIGILL	4	Core	Illegal Instruction
SIGTRAP	5	Core	Trace/Breakpoint Trap
SIGABRT	6	Core	Abort
SIGEMT	7	Core	Emulation Trap
SIGFPE	8	Core	Arithmetic Exception
SIGKILL	9	Exit	Killed
SIGBUS	10	Core	Bus Error
SIGSEGV	11	Core	Segmentation Fault
SIGSYS	12	Core	Bad System Call
SIGPIPE	13	Exit	Broken Pipe
SIGALRM	14	Exit	Alarm Clock
SIGTERM	15	Exit	Terminated
SIGUSR1	16	Exit	User Signal 1
SIGUSR2	17	Exit	User Signal 2
SIGCHLD	18	Ignore	Child Status
SIGPWR	19	Ignore	Power Fail/Restart
SIGWINCH	20	Ignore	Window Size Change
SIGURG	21	Ignore	Urgent Socket Condition
SIGPOLL	22	Ignore	Socket I/O Possible
SIGSTOP	23	Stop	Stopped (signal)
SIGTSTP	24	Stop	Stopped (user)
SIGCONT	25	Ignore	Continued
SIGTTIN	26	Stop	Stopped (tty input)
SIGTTOU	27	Stop	Stopped (tty output)
SIGVTALRM	28	Exit	Virtual Timer Expired
SIGPROF	29	Exit	Profiling Timer Expired
SIGXCPU	30	Core	CPU time limit exceeded
SIGXFSZ	31	Core	File size limit exceeded
SIGWAITING	32	Ignore	All LWPs blocked
SIGLWP	33	Ignore	Virtual Interprocessor Interrupt for Threads Library
SIGAIO	34	Ignore	Asynchronous I/O

UNIX processes can receive 'signals' with an associated default action

SIGHUP	1	Exit	Hangup
SIGINT	2	Exit	Interrupt
SIGQUIT	3	Core	Quit
SIGILL	4	Core	Illegal Instruction
SIGTRAP	5	Core	Trace/Breakpoint Trap
SIGABRT	6	Core	Abort
SIGEMT	7	Core	Emulation Trap
SIGFPE	8	Core	Arithmetic Exception
SIGKILL	9	Exit	Killed
SIGBUS	10	Core	Bus Error
SIGSEGV	11	Core	Segmentation Fault
SIGSYS	12	Core	Bad System Call
SIGPIPE	13	Exit	Broken Pipe
SIGALRM	14	Exit	Alarm Clock
SIGTERM	15	Exit	Terminated
SIGUSR1	16	Exit	User Signal 1
SIGUSR2	17	Exit	User Signal 2
SIGCHLD	18	Ignore	Child Status
SIGPWR	19	Ignore	Power Fail/Restart
SIGWINCH	20	Ignore	Window Size Change
SIGURG	21	Ignore	Urgent Socket Condition
SIGPOLL	22	Ignore	Socket I/O Possible
SIGSTOP	23	Stop	Stopped (signal)
SIGTSTP	24	Stop	Stopped (user)
SIGCONT	25	Ignore	Continued
SIGTTIN	26	Stop	Stopped (tty input)
SIGTTOU	27	Stop	Stopped (tty output)
SIGVTALRM	28	Exit	Virtual Timer Expired
SIGPROF	29	Exit	Profiling Timer Expired
SIGXCPU	30	Core	CPU time limit exceeded
SIGXFSZ	31	Core	File size limit exceeded
SIGWAITING	32	Ignore	All LWPs blocked
SIGLWP	33	Ignore	Virtual Interprocessor Interrupt for Threads Library
SIGAIO	34	Ignore	Asynchronous I/O

UNIX processes can receive 'signals' and handle ('trap') them

```
SIGNAL(2)          Linux Programmer's Manual      SIGNAL(2)

NAME
    signal - ANSI C signal handling

SYNOPSIS
#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);

DESCRIPTION
    The behavior of signal() varies across Unix versions, and has also varied his-
    torically across different versions of Linux.  Avoid its use: use sigaction(2)
    instead.  See Portability below.

    signal() sets the disposition of the signal signum to handler, which is either
    SIG_IGN, SIG_DFL, or the address of a programmer-defined function (a "signal
    handler").
```

Previous C recipe

```
// gcc count.c -o count && ./count
#include <stdio.h>
void main()
{
    int i, the_start, the_end;

    the_start = 1;
    the_end = 10;

    for (i=the_start; i<=the_end; i++)
    {
        printf("%d\n", i);
        sleep(1);
    }
}
```

1. dfr@manneback (ssh)

```
// gcc crcount.c -o crcount && ./crcount
#include <stdio.h>
void main()
{
    int i, the_start, the_end;
    FILE * file;

    // Try to recover current state
    file = fopen("state", "r");
    if (file)
    {
        fscanf(file, "%d", &the_start);
        fclose(file);
    }
    else
    {
        // Otherwise bootstrap at 1
        the_start = 1;
    }

    the_end = 10;

    for (i=the_start; i<=the_end; i++)
    {
        // Save current state
        file = fopen("state", "w");
        fprintf(file, "%d", i);
        fclose(file);

        // Heavy computations
        printf("%d\n", i);
        sleep(1);
    }
}
```

count.c

12,5

All crcount.c

3,1

Top

```
// gcc crsigxcount.c -o crsigexcount && ./crsig  
excount  
  
#include <stdlib.h>  
#include <signal.h>  
#include <stdio.h>  
  
volatile sig_atomic_t i = 1;  
FILE * file;  
  
void catch_signal(int sig)  
{  
    // Save current state  
    file = fopen("state", "w");  
    fprintf(file, "%d", i);  
    fclose(file);  
  
    // Exit  
    exit(0);  
}  
  
void main()  
{  
    int the_start, the_end;  
  
    // Register signal handler  
    signal(SIGINT, catch_signal);  
  
    // Try to recover current state  
    file = fopen("state", "r");  
    if (file)  
    {  
        fscanf(file, "%d", &the_start);  
        fclose(file);  
    }  
    else  
    {  
        // Otherwise bootstrap at 1  
        the_start = 1;  
    }  
  
    the_end = 10;  
  
    for (i=the_start; i<=the_end; i++)  
    {  
        // Heavy computations that might  
        // be interrupted  
        printf("%d\n", i);  
        sleep(1);  
    }  
}
```

C signal recipe

```
// gcc crsigvacount.c -o crsigvacount && ./crsigvacount

#include <stdlib.h>
#include <signal.h>
#include <stdio.h>

volatile sig_atomic_t interrupted = 0;

void catch_signal(int sig)
{
    interrupted = 1;
}

void main()
{
    int i, the_start, the_end;
    FILE * file;

    // Register signal handler
    signal(SIGINT, catch_signal);

    // Try to recover current state
    file = fopen("state", "r");
    if (file)
    {
        fscanf(file, "%d", &the_start);
        fclose(file);
    }
    else
        the_start = 0;
    the_end = 10;

    for (i=the_start; i<=the_end && !interrupted;
         i++)
    {
        // Heavy computations that
        // might be interrupted
        printf("%d\n", i);
        sleep(1);
    }

    // Iterations are over or
    // have been interrupted
    // Anyway save the state.
    file = fopen("state", "w");
    fprintf(file, "%d", i);
    fclose(file);
}
```

C signal recipe

Fortan signal recipe

```
! gfortran crsigvacount.f90 -o crsigvacount&& . /crsigvacount

subroutine catch_signal
    logical :: interrupted
    common interrupted
    interrupted = .true.
end subroutine catch_signal

program crcount
    integer :: i, the_start, the_end
    integer :: stat

    logical :: interrupted
    common interrupted
    external catch_signal

    integer, parameter :: SIGINT = 2

    call signal(SIGINT, catch_signal)

    ! Try to recover current state
    open (1, file='state', status='old', &
          action='READ', iostat=stat)
    if (stat .eq. 0) then
        read(1,*), the_start
    else
        ! Otherwise bootstrap at 1
        the_start = 1
    end if
    close(1)

    the_end = 10
    interrupted = .false.

    do i = the_start, the_end
        if (interrupted) exit
        ! Heavy computations
        write(*, '(i2)' ) i
        call Sleep(1)
    end do

    ! Save current state
    open (1, file='state', status='REPLACE', &
          action='WRITE', iostat=stat)
    write(1, '(i2)' ) i
    close(1)

end program crcount
```

Fortan signal recipe

```
1. dfr@manneback (ssh)
! wget https://bitbucket.org/nxg/libsigwatch/do|the_end = 10
wnloads/libsigwatch-1.0.tar.gz
! tar xf libsigwatch-1.0.tar.gz
! (cd libsigwatch-1.0 && ./configure && make)
! gfortran libsigwatch-1.0/.libs/sigwatch.o crs|do i = the_start, the_end
igvalibcount.f90 -o crsigvalibcount && ./crsig| sig = getlastsignal()
alibcount
if (sig.eq.99) exit

! Heavy computations
write(*, '(i2)' ) i
call Sleep(1)
end do

! Save current state
open (1, file='state', status='REPLACE', action=|close(1)

end program crcount

! Try to recover current state
open (1, file='state', status='old', action='RE|AD', iostat=stat)
if (stat .eq. 0) then
  read(1,*), the_start
else
  ! Otherwise bootstrap at 1
  the_start = 1
end if
close(1)

stat = watchsignalname("INT", 99)
crsigvalibcount.f90          26,0-1           Top crsigvalibcount.f90      41,1
"crsigvalibcount.f90" 55L, 962C written
```

```
#! /bin/env python

from time import sleep
import signal, os

interrupted = False
def signal_handler(signum, frame):
    global interrupted
    interrupted = True

try:
    # Try to recover current state
    with open('state', 'r') as file:
        the_start = int(file.read())
except:
    # Otherwise bootstrap at 1
    the_start = 1

signal.signal(signal.SIGINT, signal_handler)
the_end = 10

for i in range(the_start, the_end):
    if interrupted:
        break
    # Heavy computations
    print i
    sleep(1)

# Save current state
with open('state', 'w') as file:
    file.write(str(i))
~
```

Python
signal
recipe

1. dfr@manneback (ssh)

```
% octave --silent --no-window-system < crsigcdcount.m
% killall -SIGTERM octave

% set core dump filename
octave_core_file_name('state')

% Try to recover current state
try
  load('state');
catch
  % Otherwise bootstrap at 1
  i = 1;
end_tryCatch

the_end = 10;

while i <= the_end
  % Heavy computations
  disp(i)
  sleep(1)
  i = i + 1;
end

~  
~  
~  
~  
~  
~  
~  
~  
~  
~  
~  
~  
~  
~  
~  
~  
~  
~
```

Octave
signal
recipe

R signal recipe

The general recipe

1. Register a signal handler
(a function that will modify a global variable when receiving a signal)
2. Test the value of the global variable periodically
(At a moment when the state is consistent an easy to recreate)
3. If the value indicates so, save state to disk
(and optionally gracefully stop)

In the previous example : The state is just an integer
Periodically means at each iteration



3

Use Slurm signaling abilities to manage checkpoint-able software in Slurm scripts on the clusters.

scancel is used to send signals to jobs

```
1. dfr@manneback (ssh)
SCANCEL(1)          Slurm components          SCANCEL(1)

NAME
    scancel - Used to signal jobs or job steps that are under the control
    of Slurm.

SYNOPSIS
    scancel      [OPTIONS...]           [job_id[_array_id][.step_id]]
    [job_id[_array_id][.step_id]...]

DESCRIPTION
    scancel is used to signal or cancel jobs, job arrays or job steps. An
    arbitrary number of jobs or job steps may be signaled using job speci-
    fication filters or a space separated list of specific job and/or job
    step IDs. If the job ID of a job array is specified with an array ID
    value then only that job array element will be cancelled. If the job
    ID of a job array is specified without an array ID value then all job
    array elements will be cancelled. A job or job step can only be sig-
    naled by the owner of that job or user root. If an attempt is made by
    an unauthorized user to signal a job or job step, an error message will
    be printed and the job will not be signaled.

OPTIONS
:|
```

Example: use scancel --signal USR1 \$SLURM_JOB_ID
to force state dump for reviewing/debugging

The screenshot shows a terminal window titled "1. dfr@manneback (ssh)". The code in the terminal is as follows:

```
#!/bin/env python

from time import sleep
import signal, os

interrupted = False
dump_requested = False

def signal_handler(signum, frame):
    global interrupted, dump_requested
    if signum == 15 or signum == 2:
        interrupted = True
    if signum == 16:
        dump_requested = True

try:
    # Try to recover current state
    with open('state', 'r') as file:
        the_start = int(file.read())
except:
    # Otherwise bootstrap at 1
    the_start = 1

signal.signal(signal.SIGINT,
             signal_handler)
the_end = 10

for i in range(the_start, the_end):
    if dump_requested or interrupted:
        # Save current state
        with open('state', 'w') as file:
            file.write(str(i))
        dump_requested = False
    if interrupted:
        break
    # Heavy computations
    print i
    sleep(1)
```

On the right side of the terminal window, the text "Python signal recipe" is displayed vertically.

At the bottom of the terminal window, there are status indicators: "<sigdumpusr1count.py 1,1" on the left, "Top <rsigdumpusr1count.py 45,0-1" in the center, and "Bot" on the right.

--signal to have Slurm send signals automatically before the end of the allocation

```
1. dfr@manneback (ssh)
This option may result the allocation being granted sooner than if the --share option was not set and allow higher system utilization, but application performance will likely suffer due to competition for resources within a node.

--signal=<sig_num>[@<sig_time>]
When a job is within sig_time seconds of its end time, send it the signal sig_num. Due to the resolution of event handling by SLURM, the signal may be sent up to 60 seconds earlier than specified. sig_num may either be a signal number or name (e.g. "10" or "USR1"). sig_time must have integer value between zero and 65535. By default, no signal is sent before the job's end time. If a sig_num is specified without any sig_time, the default time will be 60 seconds.

--sockets-per-node=<sockets>
Restrict node selection to nodes with at least the specified number of sockets. See additional information under -B option above when task/affinity plugin is enabled.

--switches=<count>[@<max-time>]
When a tree topology is used, this defines the maximum count of switches desired for the job allocation and optionally the maxi-
/sbatch man page
```

Example: send SIGINT 60 seconds before job is killed
(so, here, after 2 minutes)

```
#!/bin/bash

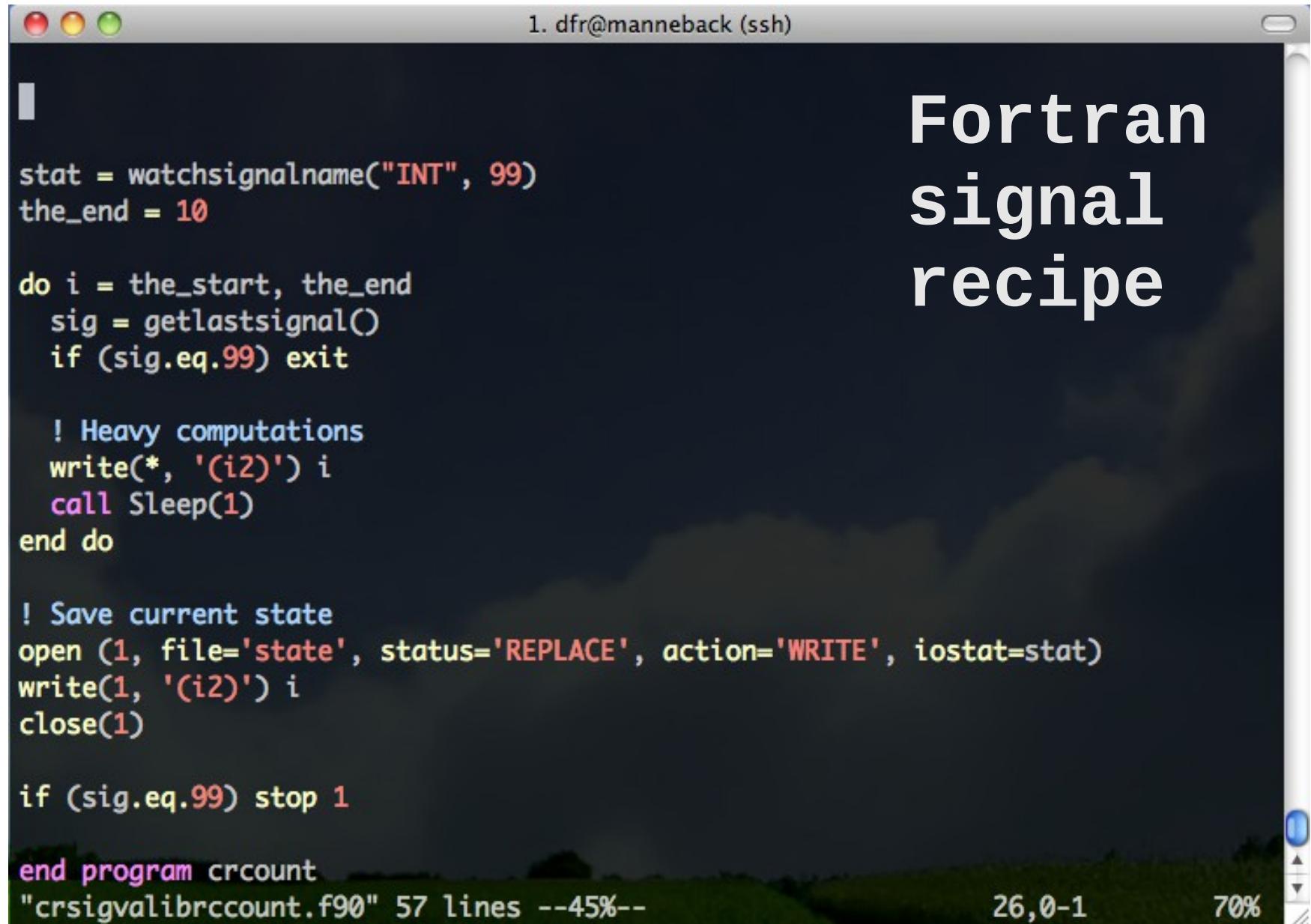
#SBATCH --job-name=test

#SBATCH --output=res

#SBATCH --time=0-00:03:00
#SBATCH --signal=INT@60
#SBATCH --mem-per-cpu=500

#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=1
```

Set non-zero return code when stopping because of a received signal



The image shows a terminal window with a dark background and light-colored text. The title bar reads "1. dfr@manneback (ssh)". On the right side of the window, the text "Fortran signal recipe" is displayed vertically. The main content of the window is a block of Fortran code:

```
stat = watchsignalname("INT", 99)
the_end = 10

do i = the_start, the_end
    sig = getlastsignal()
    if (sig.eq.99) exit

    ! Heavy computations
    write(*, '(i2)' ) i
    call Sleep(1)
end do

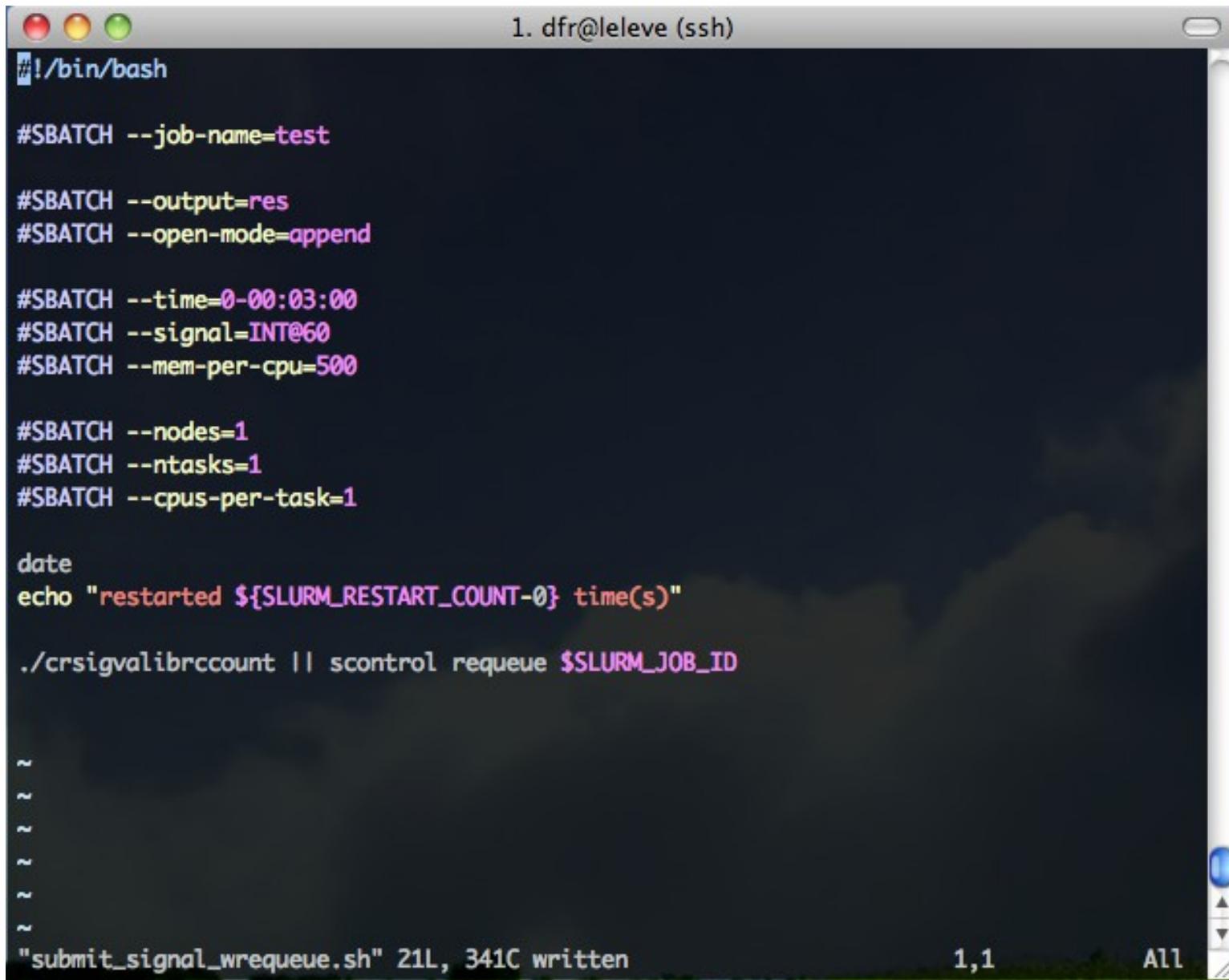
! Save current state
open (1, file='state', status='REPLACE', action='WRITE', iostat=stat)
write(1, '(i2)' ) i
close(1)

if (sig.eq.99) stop 1

end program crcount
"crsigvalibrccount.f90" 57 lines --45%--
```

The bottom right corner of the terminal window shows a progress bar at 70% completion.

Then you can have your job re-queued automatically



The screenshot shows a terminal window titled "1. dfr@leleve (ssh)". The window contains a bash script with the following content:

```
#!/bin/bash

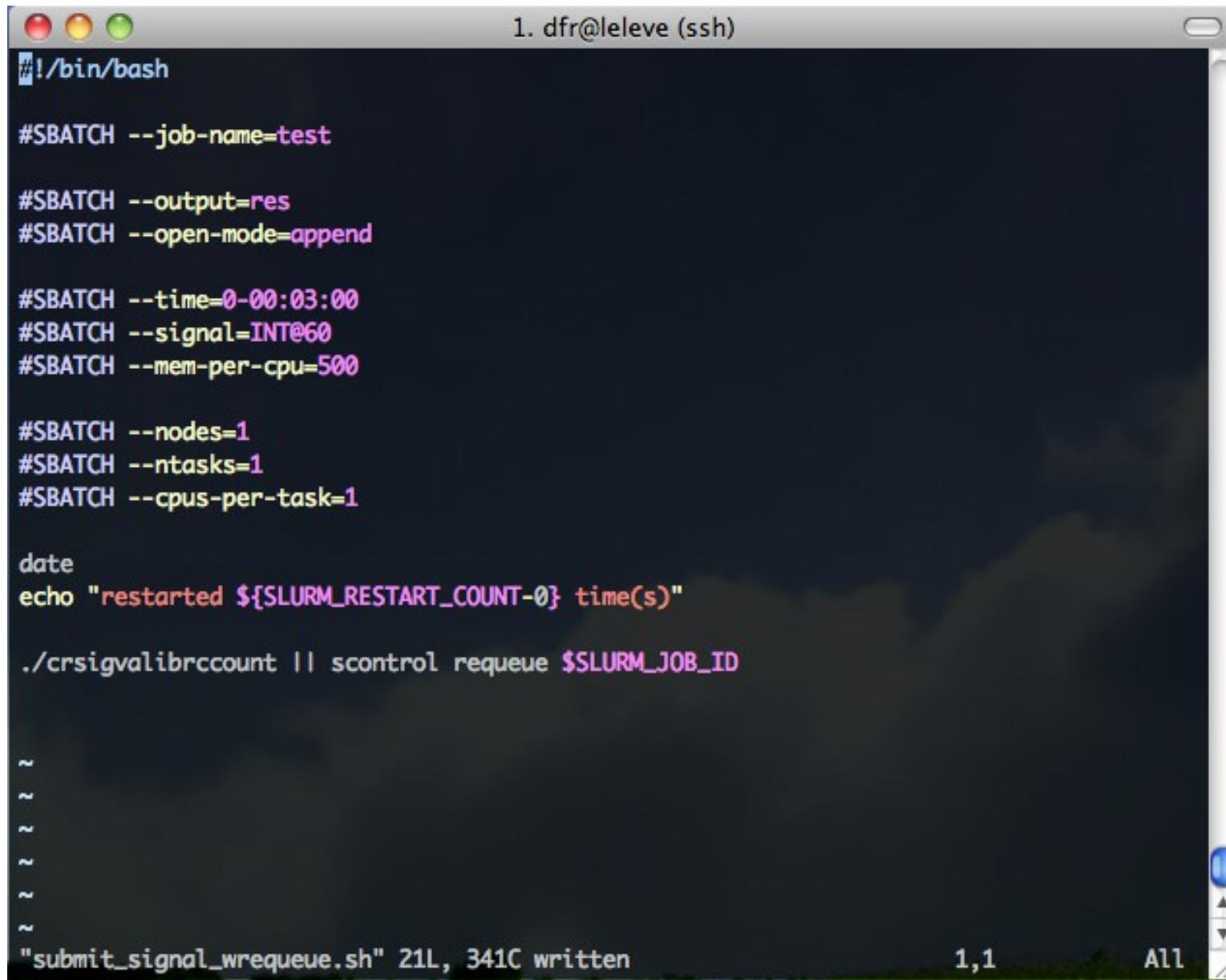
#SBATCH --job-name=test
#SBATCH --output=res
#SBATCH --open-mode=append
#SBATCH --time=0-00:03:00
#SBATCH --signal=INT@60
#SBATCH --mem-per-cpu=500
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=1

date
echo "restarted ${SLURM_RESTART_COUNT-0} time(s)"

./crsigvalibrcount || scontrol requeue $SLURM_JOB_ID
```

At the bottom of the terminal, there are several tilde (~) characters followed by the message: "submit_signal_wrequeue.sh" 21L, 341C written. The status bar at the bottom right shows "1,1" and "All".

Note the --open-mode=append



A screenshot of a Mac OS X terminal window titled "1. dfr@leleve (ssh)". The window contains a bash script with several Slurm directives (SBATCH) and some standard output. The Slurm directives include job name, output file, open mode (set to append), time limit, signal handling, memory per CPU, nodes, tasks, and cpus-per-task. The script also includes a date command and an echo command printing the restart count. At the bottom, it shows the command submitted and its statistics.

```
#!/bin/bash

#SBATCH --job-name=test
#SBATCH --output=res
#SBATCH --open-mode=append

#SBATCH --time=0-00:03:00
#SBATCH --signal=INT@60
#SBATCH --mem-per-cpu=500

#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=1

date
echo "restarted ${SLURM_RESTART_COUNT-0} time(s)"

./crsigvalibrcount || scontrol requeue $SLURM_JOB_ID

~
~
~
~
~
~

"submit_signal_wqueue.sh" 21L, 341C written          1,1      All
```

Or chain the jobs...



-d, --dependency=<dependency_list>

Defer the start of this job until the specified dependencies have been satisfied completed. *<dependency_list>* is of the form *<type:job_id[:job_id][,type:job_id[:job_id]]>*. Many jobs can share the same dependency and these jobs may even belong to different users. The value may be changed after job submission using the scontrol command.

after:job_id[:jobid...]

This job can begin execution after the specified jobs have begun execution.

afterany:job_id[:jobid...]

This job can begin execution after the specified jobs have terminated.

afternotok:job_id[:jobid...]

This job can begin execution after the specified jobs have terminated in some failed state (non-zero exit code, node failure, timed out, etc).

afterok:job_id[:jobid...]

This job can begin execution after the specified jobs have successfully executed (ran to completion with an exit code of zero).

expand:job_id

Resources allocated to this job should be used to expand the specified job. The job to expand must share the same QOS (Quality of Service) and partition. Gang scheduling of resources in the partition is also not supported.

singleton

This job can begin execution after any previously launched jobs sharing the same job name and user have terminated.

-D, --workdir=<directory>

Set the working directory of the batch script to *directory* before it is executed.

Set a non-zero exit code

C: `exit(1)`

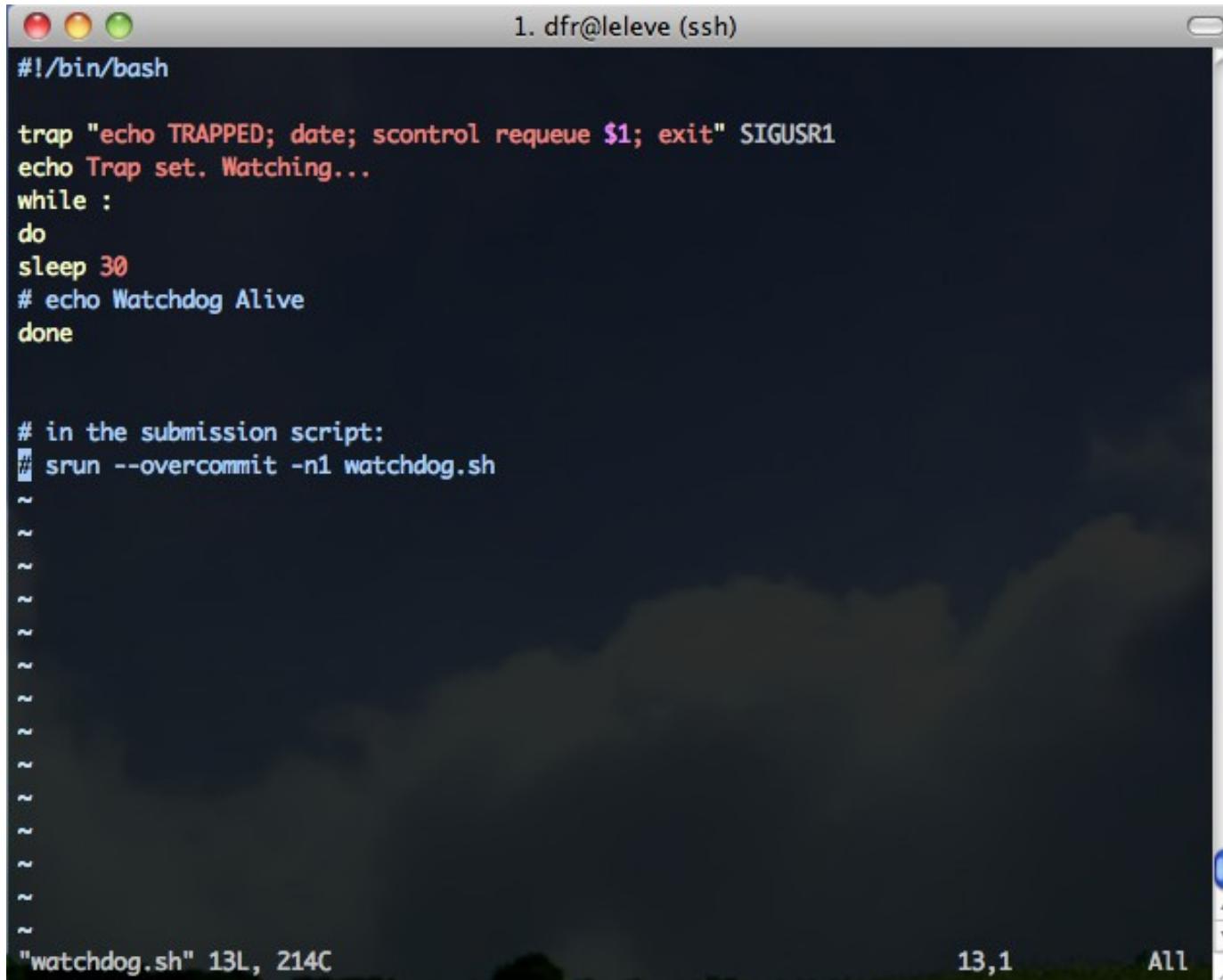
Fortran: `stop 1`

Octave: `exit(1)`

R: `quit(status=1)`

Python: `sys.exit(1)`

Using a signal-based watchdog to re-queue the job just before it is killed



The screenshot shows a terminal window titled "1. dfr@leleve (ssh)". The window contains a bash script named "watchdog.sh". The script starts with "#!/bin/bash" and includes a trap command to handle SIGUSR1 signals. It then prints a message and enters a loop that sleeps for 30 seconds and checks if the job is alive. If it's not, it requeues the job using "scontrol requeue \$1". The submission script at the bottom shows the command "srun --overcommit -n1 watchdog.sh". The terminal window has a dark background with light-colored text and a scroll bar on the right.

```
#!/bin/bash

trap "echo TRAPPED; date; scontrol requeue $1; exit" SIGUSR1
echo Trap set. Watching...
while :
do
sleep 30
# echo Watchdog Alive
done

# in the submission script:
# srun --overcommit -n1 watchdog.sh
~
```

"watchdog.sh" 13L, 214C 13,1 All



4

Use serialization tools and
libraries for efficient and
persistent data storage on disk

Standard data file format allow browsing, postprocessing and transmitting intermediate data

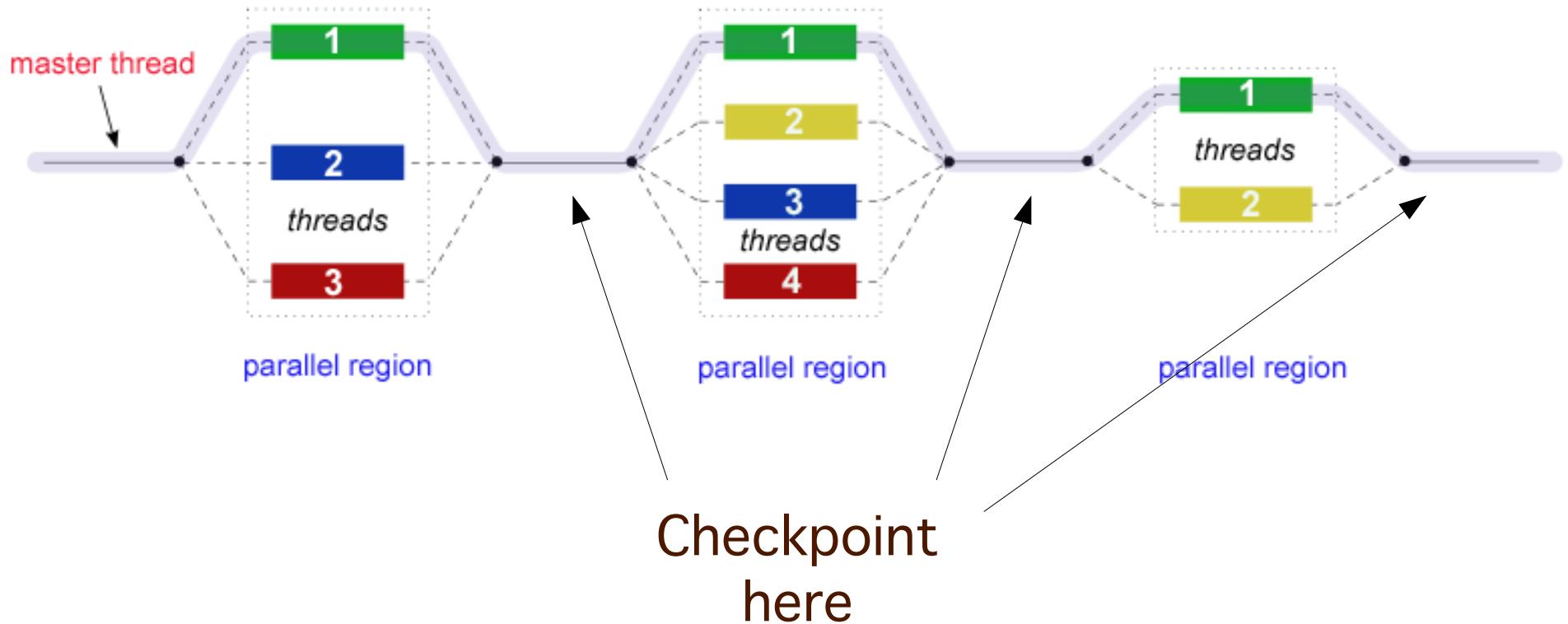
Data size	Storage type
~10MB	CSV
~10GB	Zipped CSV or Binary
~100GB	HDF5, sqlite
~ 10TB	MongoDB, Postgres



5

Parallel programs are better
checkpointed after a global
synchronization.

In the fork-join model, checkpoint after a join and before a fork



Easily ensure state consistency
Allows restarting with a different number of threads

A survey of checkpointing algorithms for parallel and distributed computers

S KALAISELVI and V RAJARAMAN^a

Supercomputer Education and Research Centre (SERC), Indian Institute of Science, Bangalore 560 012, India

^aAlso at Jawaharlal Nehru Centre for Advanced Scientific Research, Indian Institute of Science Campus, Bangalore 560 012, India
e-mail: rajaram@serc.iisc.ernet.in

MS received 27 August 1998; revised 8 June 2000

Abstract. *Checkpoint* is defined as a designated place in a program at which normal processing is interrupted specifically to preserve the status information necessary to allow resumption of processing at a later time. *Checkpointing* is the process of saving the status information. This paper surveys the algorithms which have been reported in the literature for checkpointing parallel/distributed systems. It has been observed that most of the algorithms published for checkpointing in message passing systems are based on the seminal article by Chandy and Lamport. A large number of articles have been published in this area by relaxing the assumptions made in this paper and by extending it to minimise the overheads of coordination and context saving. Checkpointing for shared memory systems primarily extend cache coherence protocols to maintain a consistent memory. All of them assume that the main memory is safe for storing the context. Recently algorithms have been published for distributed shared memory systems, which extend the cache coherence protocols used in shared memory systems. They however also include methods for storing the status of distributed memory in stable storage. Most of the algorithms assume that there is no knowledge about the programs being executed. It is however felt that in development of parallel programs the user has to do a fair amount of work in distributing tasks and this information can be effectively used to simplify checkpointing and rollback recovery.

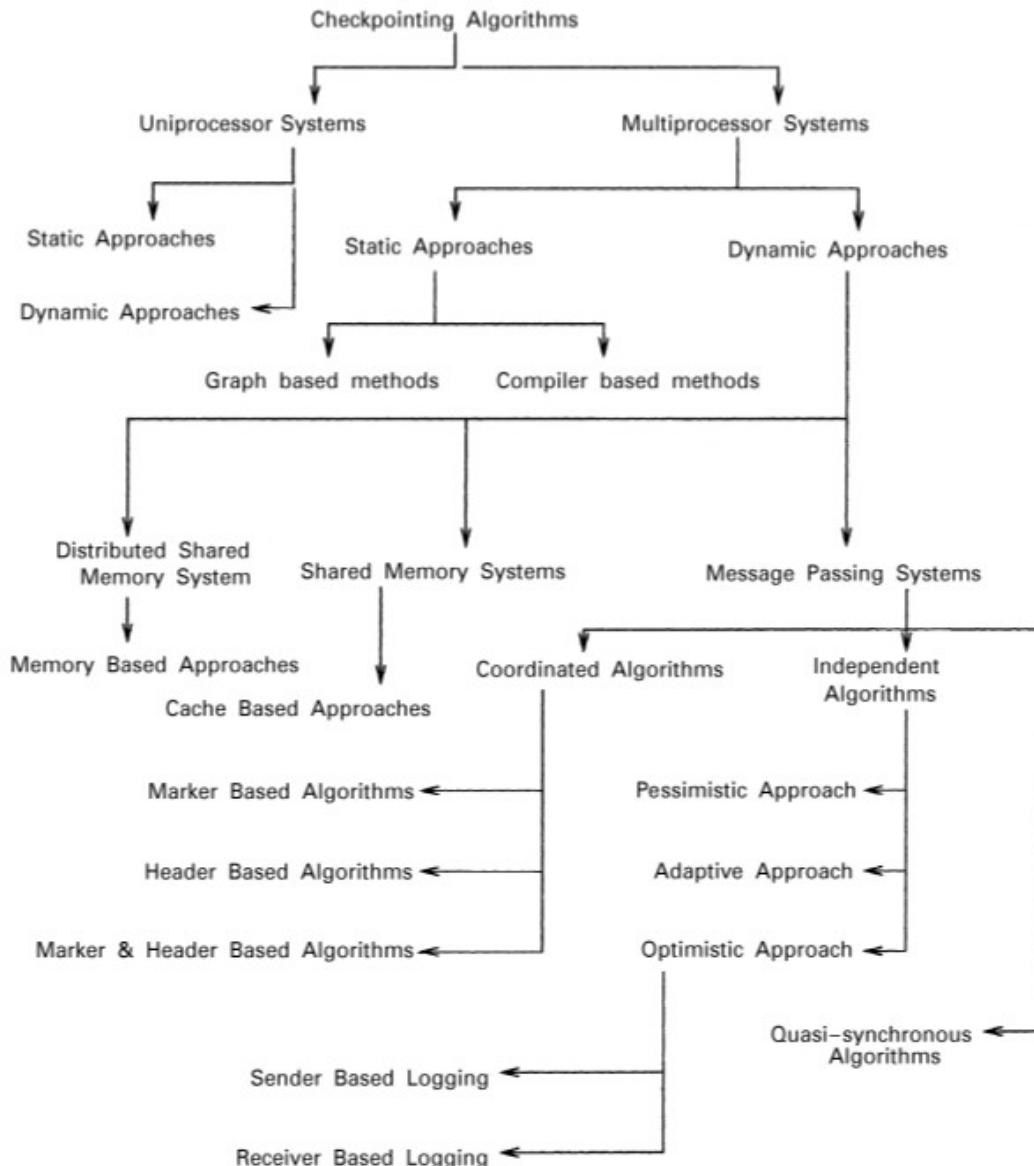


Figure 2. Classification of checkpointing algorithms.



FAQ: Fault tolerance for parallel MPI jobs

| [Home](#) | [Support](#) | [FAQ](#) | | all just the FAQ »

3. Does Open MPI support checkpoint and restart of parallel jobs (similar to LAM/MPI)?

Yes. The v1.3 series was the first release series of Open MPI to include support for the transparent, coordinated checkpointing and restarting of MPI processes (similar to LAM/MPI).

Open MPI supports both the the [BLCR](#) checkpoint/restart system and a "self" checkpointer that allows applications to perform their own checkpoint/restart functionality while taking advantage of the Open MPI checkpoint/restart infrastructure. For both of these, Open MPI provides a coordinated checkpoint/restart protocol and integration with a variety of network interconnects including shared memory, Ethernet, InfiniBand, and Myrinet.

The implementation introduces a series of new frameworks and components designed to support a variety of checkpoint and restart techniques. This allows us to support the methods described above (application-directed, BLCR, etc.) as well as other kinds of checkpoint/restart systems (e.g., Condor, libckpt) and protocols (e.g., uncoordinated, message induced).

Note: The checkpoint/restart support was last released as part of the v1.6 series. The v1.7 series and the Open MPI trunk do not support this functionality (most of the code is present in the repository, but it is known to be non-functional in most cases). This feature is looking for a maintainer. Interested parties should inquire on the developers mailing list.



Part Two: Checkpointing when you do not have the code



6

Use programs and libraries that enable other programs with checkpoint/restart capabilities.

Such program needs to:

1. Access the process' memory
(the c/r program forks itself as the process, or uses a kernel module)
2. Access the processor state at any moment
(it uses signals to interrupt the process and provoke storage of the registers on the stack)
3. Track the state changing actions (fork, exec, system, etc.)
(wrap standard library functions with LD_PRELOAD'ed custom functions)
4. Inject checkpointing code in the program
(LD_PRELOAD a library with signal handlers)

LD_PRELOAD magic

1. dfr@leleve (ssh)

```
#include <stdio.h>
main()
{
    printf("Hello world");
}
~
```

hello.c 1,1 All

```
#define _GNU_SOURCE
#include <stdio.h>
#include <dlfcn.h>
#include <stdlib.h>
#include <stdarg.h>

int printf(const char *format, ...)
{
    va_list ap; char *args; typeof_printf) *real_printf;

    // need to process args to forward them to the real printf
    va_start(ap, format); vasprintf(&args, format, ap); va_end(ap);

    real_printf = dlsym(RTLD_NEXT, "printf"); // get a pointer to the real printf
    return (*real_printf)(-- %s --\n", args); // we call the real printf
}
```

~

myprintf.c 1,9 All

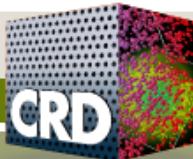
LD_PRELOAD magic

```
1. dfr@leleve (ssh)
dfr@leleve:~/Checkpointing/lddmagic $ make
gcc -fPIC -o hello hello.c
gcc -shared -fPIC -Wl,-soname -Wl,libmyprintf.so -ldl -o libmyprintf.so myprintf.c
dfr@leleve:~/Checkpointing/lddmagic $ ./hello
Hello worlddfr@leleve:~/Checkpointing/lddmagic $ ldd hello
linux-vdso.so.1 => (0x00007fff9f7ff000)
libc.so.6 => /lib64/libc.so.6 (0x00007fe02499c000)
/lib64/ld-linux-x86-64.so.2 (0x00007fe024d45000)
dfr@leleve:~/Checkpointing/lddmagic $ LD_PRELOAD=./libmyprintf.so ldd hello
linux-vdso.so.1 => (0x00007fff1dd63000)
./libmyprintf.so (0x00007f934a843000)
libc.so.6 => /lib64/libc.so.6 (0x00007f934a49a000)
libdl.so.2 => /lib64/libdl.so.2 (0x00007f934a296000)
/lib64/ld-linux-x86-64.so.2 (0x00007f934aa45000)
dfr@leleve:~/Checkpointing/lddmagic $ LD_PRELOAD=./libmyprintf.so ./hello
-- Hello world --
dfr@leleve:~/Checkpointing/lddmagic $
```



7

BLCR : the Berkely Labs Checkpoint/ Restart for Linux works with a kernel module and a shared library



HOME

ABOUT

NEWS & PUBLICATIONS

GROUPS & DEPTS

SEMINARS

SEARCH...

»GO

Home » Groups & Depts » Future Technologies Group » Research » Current Projects » Checkpoint/Restart

GROUPS & DEPTS

Advanced Computing for Science Dept.

Applied Numerical Algorithms Group

Biological Data Management & Technology Center

CCSE

Computational Cosmology Center

Future Technologies Group

Research

Current Projects

Other Projects

CoDEx

GASNet

FastOS

Green Flash

TOP500

BeBOP

Berkeley UPC

CAL

ASIM

Berkeley Lab Checkpoint/Restart (BLCR) for LINUX

Future Technologies Group researchers are developing a hybrid kernel/user implementation of checkpoint/restart. Their goal is to provide a robust, production quality implementation that checkpoints a wide range of applications, without requiring changes to be made to application code. This work focuses on checkpointing parallel applications that communicate through MPI, and on compatibility with the software suite produced by the SciDAC Scalable Systems Software ISIC. This work is broken down into 4 main areas:

- Checkpoint/Restart for Linux (CR)
- Checkpointable MPI Libraries
- Resource Management Interface to Checkpoint/Restart
- Development of Process Management Interfaces

News

January 29, 2013

Version 0.8.5 is now available from the [Checkpoint Downloads](#) page.

This version fixes several bugs, and extends support to kernels through 3.7.1.

January 14, 2013

TABLE OF CONTENTS

1. News
2. Documentation
3. Publications
4. Downloads
5. Other Resources
6. Features

Advertised Features

- Fully SMP safe
- Rebuilds the virtual address space and restores registers
- Supports the NPTL implementation of POSIX threads (LinuxThreads is no longer supported)
- Restores file descriptors, and state associated with an open file
- Restores signal handlers, signal mask, and pending signals.
- Restores the process ID (PID), thread group ID (TID), parent process ID (PPID), and process tree to old state.
- Support save and restore of groups of related processes and the pipes that connect them.
- Should work with nearly any x86 or x86_64 Linux system that uses a 2.6 kernel (see FAQ for most recent info). Verified to work on SuSE Linux 9.x and up; Red Hat 8 and 9; Red Hat Enterprise Linux version 3, 4 and 5; Fedora Core 5 through 10; and many vanilla Linux kernels (from kernel.org) from 2.6.0 on up (and many more).
- Experimental support is present for PPC, PPC64 and ARM architectures. We consider this support experimental mainly because of our limited ability to test it.
- Xen dom0 and domU are both supported with Xen 3.1.2 or newer.
- Tested with the GNU C library (glibc) versions 2.1 through 2.6

Recall the non-checkpointable program

```
1. dfr@manneback (ssh)
// gcc count.c -o count && ./count
#include <stdio.h>
void main()
{
    int i, the_start, the_end;

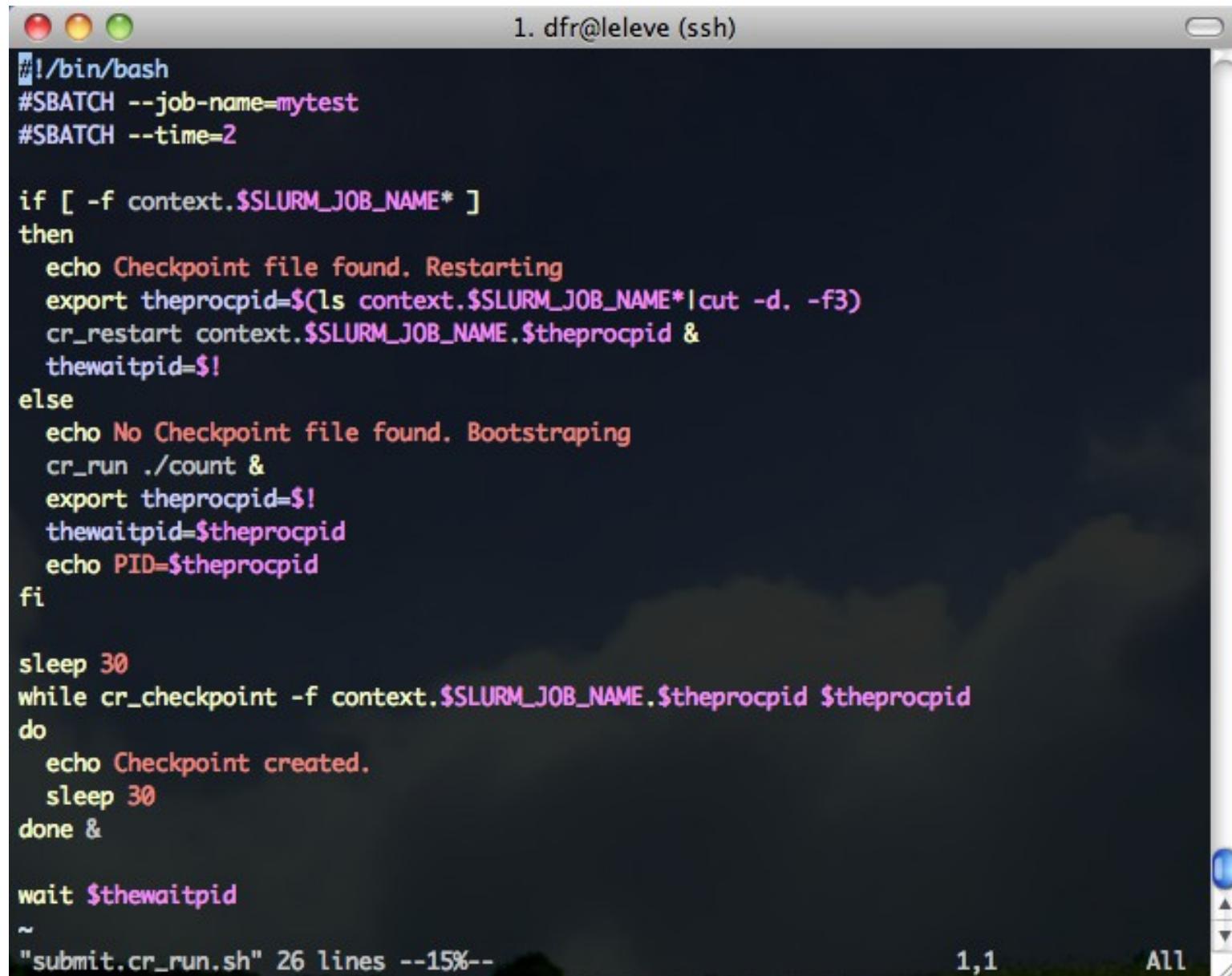
    the_start = 1;
    the_end = 10;

    for (i=the_start; i<=the_end; i++)
    {
        printf("%d\n", i);
        sleep(1);
    }
}
~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~ ~
"count.c" 15L, 219C           1,1           All
```

Run with cr_run ; restart with cr_restart

```
1. dfr@manneback (ssh)
dfr@manneback:~/Checkpointing $ cr_run ./count & export THEPID=$! ; sleep 4 ;
cr_checkpoint $THEPID ; kill $THEPID
[1] 26053
1
2
3
4
5
[1]+  Terminated          cr_run ./count
dfr@manneback:~/Checkpointing $ ls -rtl|tail -1
-r----- 1 dfr grppan 172590 Oct  8 15:57 context.26053
dfr@manneback:~/Checkpointing $ cr_restart context.$THEPID
5
6
7
8
9
10
dfr@manneback:~/Checkpointing $
```

The submission script looks for checkpoint and cr_runs or cr_restarts accordingly



A screenshot of a terminal window titled "1. dfr@leleve (ssh)". The window contains a bash script with syntax highlighting. The script starts with "#!/bin/bash" and "#SBATCH --job-name=mytest #SBATCH --time=2". It then checks if a file named "context.\$SLURM_JOB_NAME*" exists. If it does, it prints a message about finding a checkpoint file and restarting, then exports the process ID of the current job as "theprocpid", runs "cr_restart" with the context and process ID, and stores the wait PID as "thewaitpid". If the file does not exist, it prints a message about no checkpoint file found and bootstrapping, runs "cr_run ./count &", exports the process ID as "theprocpid", sets "thewaitpid" to the process ID, and prints the PID. After this, it sleeps for 30 seconds. A loop follows, where it checks for a checkpoint file every 30 seconds using "cr_checkpoint -f context.\$SLURM_JOB_NAME.\$theprocpid \$theprocpid". If the file exists, it prints a message about creating a checkpoint and sleeps again. Once the loop exits, it prints "done &". Finally, it waits for the process with PID "thewaitpid". The script ends with a tilde (~) and the command "submit.cr_run.sh" followed by its line count (26) and a progress indicator ("--15%--"). The status bar at the bottom right shows "1,1" and "All".

```
#!/bin/bash
#SBATCH --job-name=mytest
#SBATCH --time=2

if [ -f context.$SLURM_JOB_NAME* ]
then
    echo Checkpoint file found. Restarting
    export theprocpid=$(ls context.$SLURM_JOB_NAME* | cut -d. -f3)
    cr_restart context.$SLURM_JOB_NAME.$theprocpid &
    thewaitpid=$!
else
    echo No Checkpoint file found. Bootstrapping
    cr_run ./count &
    export theprocpid=$!
    thewaitpid=$theprocpid
    echo PID=$theprocpid
fi

sleep 30
while cr_checkpoint -f context.$SLURM_JOB_NAME.$theprocpid $theprocpid
do
    echo Checkpoint created.
    sleep 30
done &

wait $thewaitpid
~
"submit.cr_run.sh" 26 lines --15%--
```

Two jobs are submitted

A checkpoint is created periodically

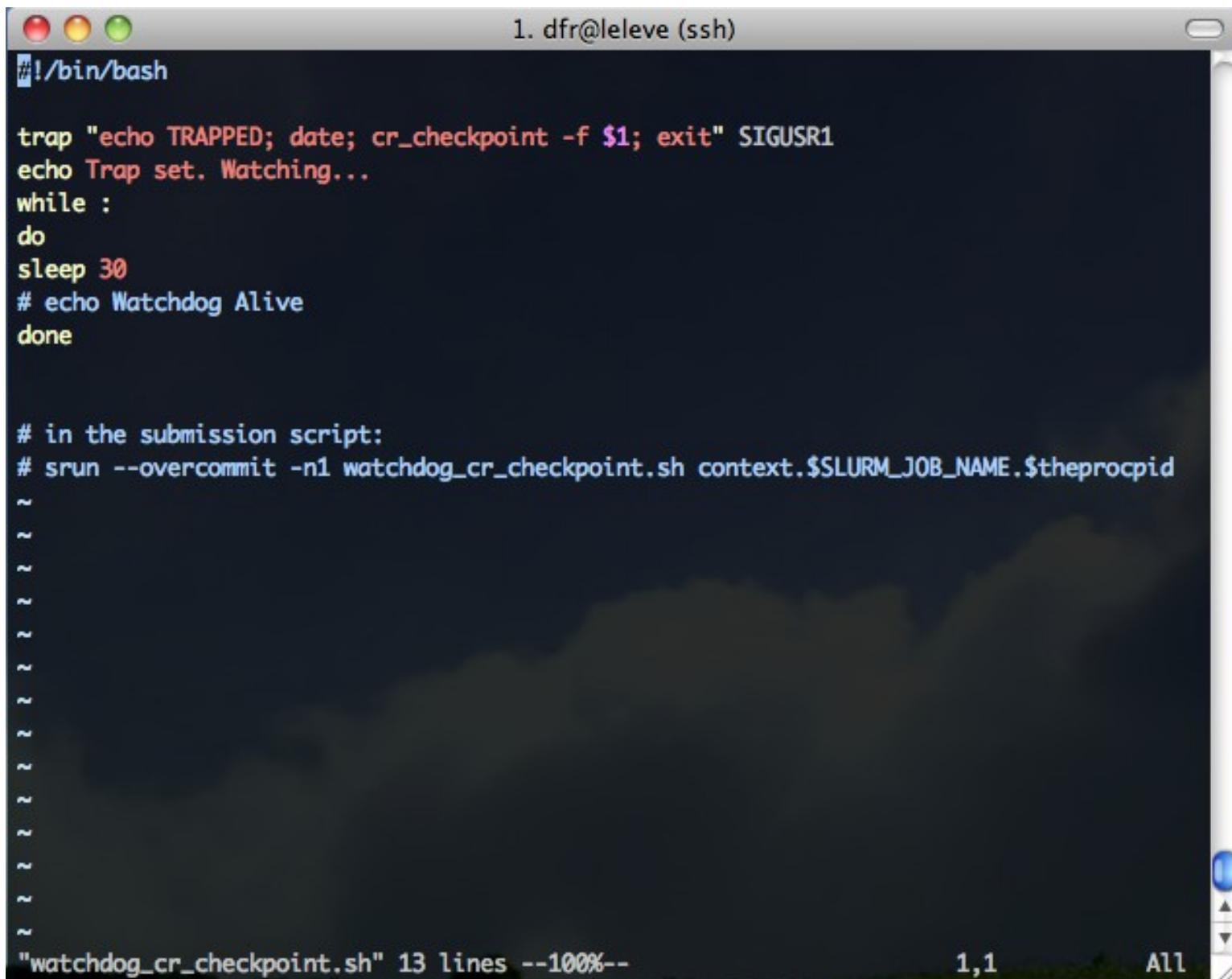
```
1. dfr@leleve (ssh)
dfr@leleve:~/Checkpointing $ sbatch --output res1 submit.cr_run.sh
Submitted batch job 76
dfr@leleve:~/Checkpointing $ sbatch --output res2 --dependency afternotok:76 submit.cr
_run.sh
Submitted batch job 77
dfr@leleve:~/Checkpointing $ squeue
      JOBID PARTITION     NAME     USER   ST      TIME  NODES NODELIST(REASON)
          77      Def  mytest    dfr  PD      0:00      1 (Dependency)
          76      Def  mytest    dfr    R      0:09      1 leleve02
dfr@leleve:~/Checkpointing $ tail -f res1
No Checkpoint file found. Bootstrapping
PID=18407
1
2
3
4
5
6
7
8
9
10
11
12
13
Checkpoint created.
```

At restart, note ./count still write to res1
while the submission script writes to res2

```
16
17
Checkpoint created.
slurm[leleve02]: error: *** JOB 76 CANCELLED AT 2013-10-10T16:01:31 DUE TO TIME LIMIT
***  

tail: res1: file truncated
17
18
19
20
21
22
23
24
25
26
27
28
29
30
^C
dfr@leleve:~/Checkpointing $ cat res2
Checkpoint file found. Restarting
Checkpoint created.
Checkpoint created.
Checkpoint created.
Checkpoint failed: no processes checkpointed
dfr@leleve:~/Checkpointing $
```

Alternatively, use a signal watchdog



The screenshot shows a terminal window titled "1. dfr@leleve (ssh)". The window contains a bash script named "watchdog_cr_checkpoint.sh". The script sets up a trap to handle SIGUSR1 signals, which triggers a checkpoint and exits. It then enters a loop that sleeps for 30 seconds and prints a message every iteration. A comment indicates it's used in a submission script with srun.

```
#!/bin/bash

trap "echo TRAPPED; date; cr_checkpoint -f $1; exit" SIGUSR1
echo Trap set. Watching...
while :
do
sleep 30
# echo Watchdog Alive
done

# in the submission script:
# srun --overcommit -n1 watchdog_cr_checkpoint.sh context.$SLURM_JOB_NAME.$thePROCID
~
```

"watchdog_cr_checkpoint.sh" 13 lines --100%-- 1,1 All



slurm
workload manager

Version 2.6

About

- [Overview](#)
- [Meetings](#)
- [What's New](#)
- [Publications](#)
- [Testimonials](#)
- [SLURM Team](#)

Using

- [Tutorials](#)
- [Documentation](#)
- [FAQ](#)
- [Getting Help](#)
- [Mailing Lists](#)

Installing

- [Download](#)
- [Installation Guide](#)
- [Platforms](#)



Google™ Custom Search



SLURM Checkpoint/Restart with BLCR

Overview

SLURM version 2.0 has been integrated with [Berkeley Lab Checkpoint/Restart \(BLCR\)](#) in order to provide automatic job checkpoint/restart support. Functionality provided includes:

1. Checkpoint of whole batch jobs in addition to job steps
2. Periodic checkpoint of batch jobs and job steps
3. Restart execution of batch jobs and job steps from checkpoint files
4. Automatically requeue and restart the execution of batch jobs upon node failure

The general mode of operation is to

1. Start the job step using the `srun_cr` command as described below.
2. Create a checkpoint of `srun_cr` using BLCR's `cr_checkpoint` command and cancel the job. `srun_cr` will automatically checkpoint your job.
3. Restart `srun_cr` using BLCR's `cr_restart` command. The job will be restarted using a newly allocated jobid.

NOTE: checkpoint/blcr cannot restart interactive jobs. It can create checkpoints for both interactive and batch steps, but **only batch jobs can be restarted**.

NOTE: BLCR operation has been verified with MVAPICH2. Some other MPI implementations should also work.

User Commands

The following documents SLURM changes specific to BLCR support. Basic familiarity with SLURM commands is assumed.

`srun`

Several options have been added to support checkpoint restart:

- **--checkpoint:** Specifies the interval between creating checkpoints of the job step. By default, the job step will have no checkpoints created. Acceptable time formats include

8

DMTCP : Distributed MultiThreading
CheckPointing works with an independent
monitoring process and a shared library

DMTCP: Distributed MultiThreaded CheckPointing

[Home](#)

[Downloads](#)

[SF project page](#)

[Browse Source](#)

[Demo](#)

[Supported Apps](#)

[Condor Integration](#)

[Manual/Documentation](#)

[API](#)

[FAQ](#)

[Publications](#)

[Contact Us](#)

About DMTCP:

DMTCP (Distributed MultiThreaded Checkpointing) is a tool to transparently checkpoint the state of multiple simultaneous applications, including multi-threaded and distributed applications. It operates directly on the user binary executable, without any Linux kernel modules or other kernel modifications.

Among the applications supported by DMTCP are Open MPI, MATLAB, Python, Perl, and many programming languages and shell scripting languages. Starting with release 1.2.0, DMTCP also supports [GNU screen](#) sessions, including vim/emacs. With the use of TightVNC, it can also checkpoint and restart X Window applications, as long as they do not use extensions (e.g.: no OpenGL, no video). See the [QUICK-START](#) file for further details.

DMTCP supports InfiniBand internally as of Aug., 2013, and will soon be released.

DMTCP is also the basis for [URDB, the Universal Reversible Debugger](#). URDB was an experimental project for reversibility for four debuggers: gdb, MATLAB, python (pdb), and perl (perl -d). It is now obsolete, and work is continuing on a newer internal project, which will be released as open source in the future.

[News](#) | [See Also](#) | [Authors](#) | [Acknowledgement](#)

Announcement!

We are currently looking for well qualified applicants who are interested in joining a Ph.D. program in order to do research on checkpointing and reversible debugging. Interested applicants should write to Gene Cooperman (gene@ccs.neu.edu) at Northeastern University.

Advertised Features

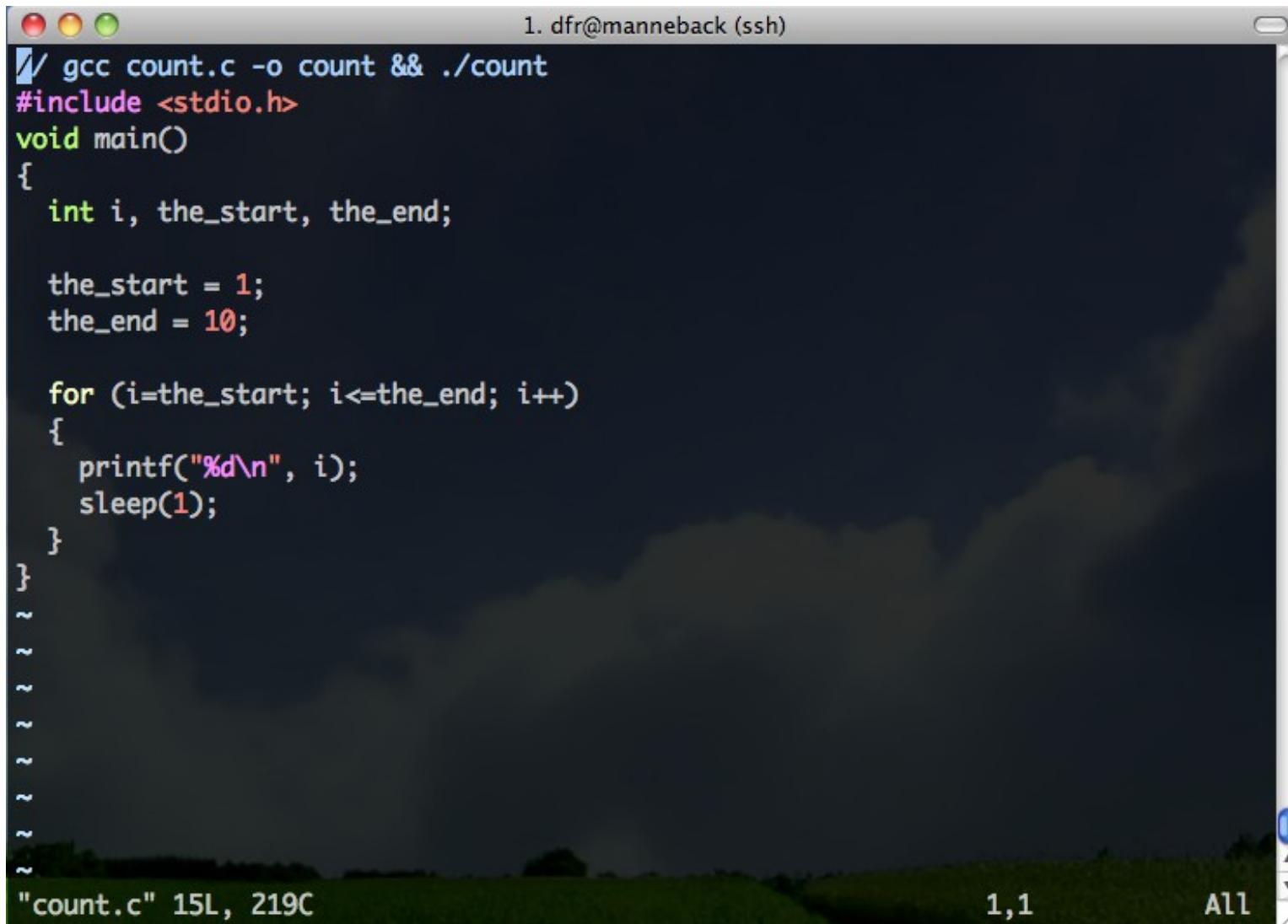
- Distributed Multi-Threaded CheckPointing
- Works with Linux Kernel 2.6.9 and later
- Supports sequential and multi-threaded computations across single/multiple hosts
- Entirely in user space (no kernel modules or root privilege)
- Transparent (no recompiling, no re-linking)
- Written at Northeastern U. and MIT and under active development for 4+ years
- LGPL'd and freely available
- No remote I/O
- Supports threads, mutexes/semaphores, forks, shared memory, exec, and many more

From their FAQ:

“ What types of programs can DMTCP checkpoint?

It checkpoints most binary programs on most Linux distributions. Some examples on which users have verified that DMTCP works are: Matlab, R, Java, Python, Perl, Ruby, PHP, Ocaml, GCL (GNU Common Lisp), emacs, vi/cscope, Open MPI, MPICH-2, OpenMP, and Cilk. See [Supported Applications](#) for further details. Our goal is to support DMTCP for all vanilla programs. If DMTCP does not work correctly on your program, **then this is a bug in DMTCP**. We would be appreciative if you can then file a bug report with DMTCP.

Recall the non-checkpointable program



The screenshot shows a terminal window titled "1. dfr@manneback (ssh)". The window contains the following text:

```
/# gcc count.c -o count && ./count
#include <stdio.h>
void main()
{
    int i, the_start, the_end;

    the_start = 1;
    the_end = 10;

    for (i=the_start; i<=the_end; i++)
    {
        printf("%d\n", i);
        sleep(1);
    }
}
~
```

The terminal prompt is at the bottom left, and the status bar at the bottom right shows "1,1 All".

Run with dmtcp_launch (runs monitoring daemon if necessary)

```
1. dfr@leleve (ssh)
dfr@leleve:~/Checkpointing $ dmtcp_launch ./count & sleep 4 ; dmtcp_command --quiet
--checkpoint ; sleep 1 ; dmtcp_command --quiet --quit
[1] 2976
dmtcp_launch (DMTCP + MTCP) version 2.0
Copyright (C) 2006-2013 Jason Ansel, Michael Rieker, Kapil Arya, and
Gene Cooperman
This program comes with ABSOLUTELY NO WARRANTY.
This is free software, and you are welcome to redistribute it
under certain conditions; see COPYING file for details.
(Use flag "-q" to hide this message.)

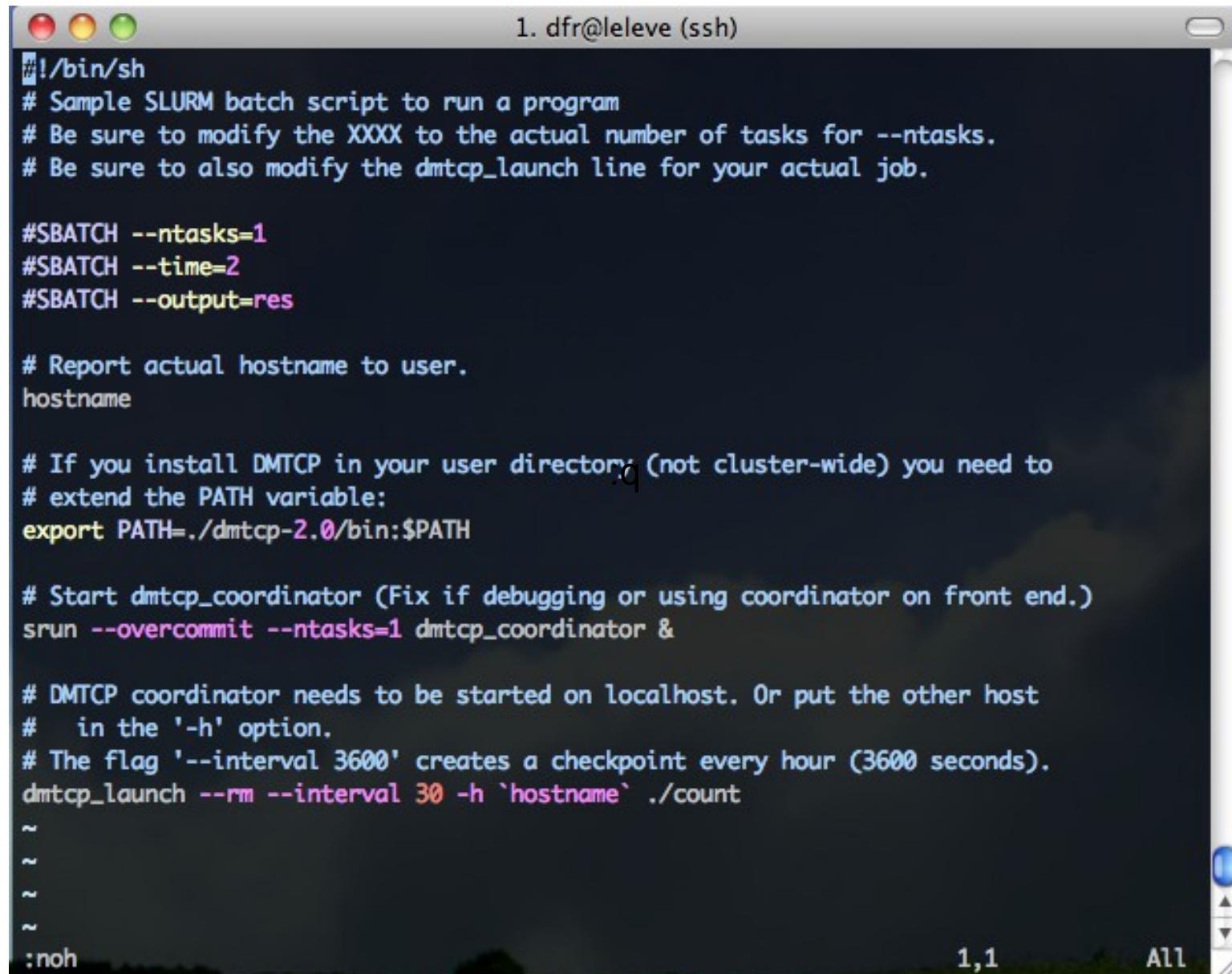
dmtcp_coordinator starting...
Host: leleve.cism.ucl.ac.be (0.0.0.0)
Port: 7779
Checkpoint Interval: disabled (checkpoint manually instead)
Exit on last client: 1
Backgrounding...
1
2
3
4
5
[1]+ Done                  dmtcp_launch ./count
dfr@leleve:~/Checkpointing $ ls -rtl|tail -1
-rwxrw-r-- 1 dfr dfr 5167 Oct 15 11:51 dmtcp_restart_script_1dcda56f5a2723b6-40000-
525d1005.sh
dfr@leleve:~/Checkpointing $
```

Restart with dmtcp_restart_script.sh

```
1. dfr@leleve (ssh)
5
[1]+ Done dmtcp_launch ./count
dfr@leleve:~/Checkpointing $ ls -rtl|tail -1
-rwxrw-r-- 1 dfr dfr 5167 Oct 15 11:52 dmtcp_restart_script_1dcda56f5a2723b6-40000-
525d1043.sh
dfr@leleve:~/Checkpointing $ ./dmtcp_restart_script.sh
dmtcp_restart (DMTCP + MTCP) version 2.0
Copyright (C) 2006-2013 Jason Ansel, Michael Rieker, Kapil Arya, and
Gene Cooperman
This program comes with ABSOLUTELY NO WARRANTY.
This is free software, and you are welcome to redistribute it
under certain conditions; see COPYING file for details.
(Use flag "-q" to hide this message.)

dmtcp_coordinator starting...
Host: leleve.cism.ucl.ac.be (0.0.0.0)
Port: 7779
Checkpoint Interval: disabled (checkpoint manually instead)
Exit on last client: 1
Backgrounding...
5
6
7
8
9
10
^C
dfr@leleve:~/Checkpointing $
```

Launch the coordinator and the program with automatic checkpointing every 30 seconds



The screenshot shows a terminal window titled "1. dfr@leleve (ssh)". The window contains a sample SLURM batch script. The script starts with a shebang line "#!/bin/sh". It includes comments about modifying the number of tasks and the dmtcp_launch line. It sets the number of tasks to 1, specifies a run time of 2 hours, and outputs results to a file named "res". It then reports the actual hostname. If DMTCP is installed in the user directory, it needs to be added to the PATH. The script then starts the dmtcp_coordinator with overcommit enabled and runs the dmtcp_launch command with a 30-second interval, specifying the current host name and a count program.

```
#!/bin/sh
# Sample SLURM batch script to run a program
# Be sure to modify the XXXX to the actual number of tasks for --ntasks.
# Be sure to also modify the dmtcp_launch line for your actual job.

#SBATCH --ntasks=1
#SBATCH --time=2
#SBATCH --output=res

# Report actual hostname to user.
hostname

# If you install DMTCP in your user directory (not cluster-wide) you need to
# extend the PATH variable:
export PATH=./dmtcp-2.0/bin:$PATH

# Start dmtcp_coordinator (Fix if debugging or using coordinator on front end.)
srun --overcommit --ntasks=1 dmtcp_coordinator &

# DMTCP coordinator needs to be started on localhost. Or put the other host
# in the '-h' option.
# The flag '--interval 3600' creates a checkpoint every hour (3600 seconds).
dmtcp_launch --rm --interval 30 -h `hostname` ./count
~  
~  
~  
~  
:noh
```

Launch coordinator and restart program

```
1. dfr@leleve (ssh)
#!/bin/sh
# Sample SLURM batch script for restart
# You'll also need to customize the SBATCH lines below.
# The script, ./dmtcp_restart_script.sh, will have been created for you
# by a checkpoint during the initial run.

#SBATCH --ntasks=1
#SBATCH --output=res
#SBATCH --open-mode=append

# Report actual hostname to user.
hostname

# If you install DMTCP in your user directory (not cluster-wide), you'll
# need to extend PATH variable:
export PATH=./dmtcp-2.0/bin:$PATH

# Start dmtcp_coordinator (Fix if debugging or using coordinator on front end.)
srun --overcommit dmtcp_coordinator &
export DMTCP_HOST=`hostname`

# The flag '--interval 3600' creates a checkpoint every hour (3600 seconds).
./dmtcp_restart_script.sh --interval 30
~  
~  
~  
~  
"submit.dmtcp.restart.sh" 23L, 742C
```



1. dfr@leleve (ssh)

```
leleve01.cism.ucl.ac.be
dmtcp_launch (DMTCP + MTCP) version 2.0
[...]
dmtcp_coordinator starting...
Backgrounding...
1
srun: error: leleve01: task 0: Exited with exit code 99
2
3
[...]
16
17
slurm[leleve01]: error: *** JOB 93 CANCELLED AT 2013-10-15T15:28:04 DUE TO TIME LIMIT
***

leleve01.cism.ucl.ac.be
dmtcp_restart (DMTCP + MTCP) version 2.0
[...]
dmtcp_coordinator starting...
Backgrounding...
srun: error: leleve01: task 0: Exited with exit code 99
17
18
19
[...]
25
26
27
"res" 29 lines --3%--
```

1,1

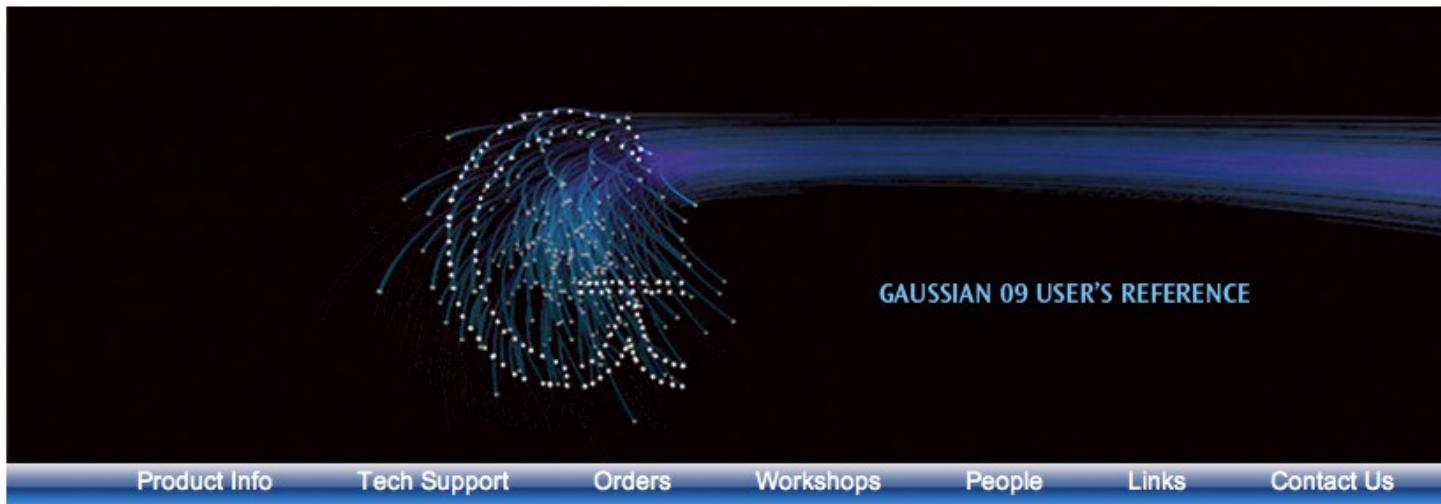
Top





9

Check whether your scientific software is checkpointable. Many of them are...



Structure of the Formatted Checkpoint File

This file is designed to be machine independent with a structure that makes it easy for post-processors to extract required data and ignore the remainder. The latter fact is important for extensibility as future additions will not interfere with applications designed for previous revisions. Typically a job is run specifying a `.chk` file, which is the binary file containing results from a calculation which are potentially useful in later calculations or for post-processing, and then after Gaussian 09 has completed, the `formchk` utility is run to generate the text `.fchk` file from the binary `.chk` file. There is also a utility, `unfchk`, to reverse the process. For backwards compatibility, running `formchk` without any options produces a subset of the full information. This document describes the results of running `formchk -3 chkfile fchkfile`, which produces a version 3 formatted checkpoint file (the current and most full-featured version).

Here is a description of the data in Fortran formatted form, although there is no particular reason to use Fortran as opposed to other languages to read the data.

Using Gaussian Checkpoint Files

Since the queues on the scientific servers have time limits, the Gaussian calculations may be terminated prematurely due to running out of time. To restart such a prematurely terminated calculation, one can use the checkpointing facilitated via the Gaussian supported checkpointing file. However, note here that the primary use of a checkpoint file is to use the results of one calculation as the starting point for a second calculation.

When a Gaussian calculation is restarted using information from a checkpoint file, the new calculation results (or the continuation from the uncompleted run) will be placed in the exact same checkpoint file, overwriting the original checkpoint file. **THUS IT IS ALWAYS SAFER TO MAKE A BACKUP COPY OF THE CHECKPOINT FILE.** Note that it is possible for a checkpoint file to become corrupted (i.e. if a calculation dies while writing to the checkpoint file).

Gaussian will use a checkpoint file if the command:

```
%Chk=file_name
```

appears before the route card in the input file. If the specified file does not exist, it will be created. If the specified file does exist, information to be used in the present calculation can be read from it.

Commands for reading from the checkpoint file

A calculation can be started using information from the checkpoint file by including one of the following commands in the route card.

ChkBasis	Read the basis set from the checkpoint file.
SCF=Restart	Restart an SCF calculation from the checkpoint file. This is normally used when an SCF calculation failed finish for some reason.
IRC=Restart	Restarts an IRC calculation that did not complete, or restarts an IRC calculation for which additional points along the reaction path are desired.
Scan=Restart	Restarts a potential energy surface scan which did not complete.
Freq=Restart	Restarts a numerical frequency calculation which did not complete. Analytic frequency calculations cannot be restarted.
Polar=Restart	Restarts a numerical polarizability calculation which did not complete.
CIS=Restart	Restarts a CIS (Configuration Interaction – Single excitation) calculation which did not complete.
Opt=Restart	Restarts an geometry optimization which did not complete.
Geom=Checkpoint	Reads the molecular geometry from the checkpoint file.
Geom=AllCheckpoint	Reads the molecular geometry, charge, multiplicity and title from the checkpoint file. This is often used to start a second calculation at a different level of theory.
Guess=Read	Reads the initial guess from the checkpoint file. If the basis set specified is different from the basis set used in the job which generated the checkpoint file, then the wave function will be projected from one basis to the other. This is an efficient way to switch from one basis to another.
Density=Checkpoint	Reads the density from the checkpoint file. This implies Guess=Only so that no integrals or SCF are computed. This is used to compute the population analysis or create cube files from a wave function without rerunning the job.
Field=Checkpoint	Reads the 34 multipole components of a finite field from the checkpoint file.
Field=EChk	Reads the 3 electric dipole field components from the checkpoint file.
Charge=Check	Reads point charges from the checkpoint file.
ReArchive	This option is used to generate an archive entry from the information in the checkpoint file. No calculation is run.

Abaqus 6.11

Abaqus/CAE User's Manual

 **SIMULIA**



RESTARTING AN ANALYSIS

Parallelization

Use the **Parallelization** tabbed page to configure the parallel execution of the Abaqus analysis jobs during the optimization process, such as the number of processors to use and the parallelization method. For more information, see "Controlling job parallel execution," Section 19.8.8.

Precision

Use the **Precision** tabbed page to choose the precision of nodal output that is written to the output database during the Abaqus analysis. For more information, see "Controlling job precision," Section 19.8.9.

19.6 Restarting an analysis

If your model contains multiple steps, you do not have to analyze all of the steps in a single analysis job. Indeed, it is often desirable to run a complex analysis in stages. This allows you to examine the results and confirm that the analysis is performing as expected before continuing with the next stage. The restart files generated by an Abaqus analysis allow you to continue the analysis from a specified step. For more information, see "Restarting an analysis," Section 9.1.1 of the Abaqus Analysis User's Manual.

This section describes the restart capability in Abaqus/CAE. The following topics provide some background information:

- "Controlling a restart analysis," Section 19.6.1
- "Files required to restart an analysis," Section 19.6.2
- "Rules governing a restart analysis," Section 19.6.3
- "The relationship between the model and the restart analysis," Section 19.6.4

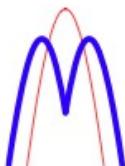
The following topics describe examples of the most common uses for restart analysis:

- "Restarting after adding more analysis steps to the model," Section 19.6.5
- "Restarting after modifying existing analysis steps," Section 19.6.6
- "Restarting from the middle of a step," Section 19.6.7
- "Visualizing results from restart analyses," Section 19.6.8
- "Recovering an Abaqus/Standard analysis," Section 19.6.9
- "Remote submission of restart jobs," Section 19.6.10

19.6.1 Controlling a restart analysis

By default, no restart information is written for an Abaqus/Standard or an Abaqus/CFD analysis and restart information is written only at the beginning and end of each step for an Abaqus/Explicit analysis.

MOLPRO



Users Manual Version 2012.1

H.-J. Werner

Institut für Theoretische Chemie
Universität Stuttgart
Pfaffenwaldring 55
D-70569 Stuttgart
Federal Republic of Germany

P. J. Knowles

School of Chemistry
Cardiff University
Main Building, Park Place, Cardiff CF10 3AT
United Kingdom

SHA1 0034bbaeb6d4f13a081e892956fbcd55b2271035

(Copyright ©2012 University College Cardiff Consultants Limited)

6 PROGRAM CONTROL

31

6.2 Ending a job (---

The end of the input is signaled by either an end of file, or a

--- card. All input following the --- card is ignored.

Alternatively, a job can be stopped at some place by inserting an EXIT card. This could also be in the middle of a DO loop or an IF block. If in such a case the --- card would be used, an error would result, since the ENDDO or ENDIF cards would not be found.

6.3 Restarting a job (RESTART)

In contrast to MOLPRO92 and older versions, the current version of MOLPRO attempts to recover all information from all permanent files by default. If a restart is unwanted, the NEW option can be used on the FILE directive. The RESTART directive as described below can still be used as in MOLPRO92, but is usually not needed.

RESTART, $r_1, r_2, r_3, r_4, \dots$

The r_i specify which files are restarted. These files must have been allocated before using FILE cards. There are two possible formats for the r_i :

- a) $0 < r_i < 10$: Restart file r_i and restore all information.
- b) $r_i = name.nr$: Restart file nr but truncate before record $name$.

If all $r_i = 0$, then all permanent files are restarted. However, if at least one r_i is not equal to zero, only the specified files are restarted.

Examples:

RESTART;	will restart all permanent files allocated with FILE cards (default)
RESTART, 1;	will restart file 1 only
RESTART, 2;	will restart file 2 only
RESTART, 1, 2, 3;	will restart files 1-3
RESTART, 2000, 1;	will restart file 1 and truncate before record 2000.

6.4 Including secondary input files (INCLUDE)

INCLUDE, $file$ [,ECHO];

Insert the contents of the specified $file$ in the input stream. In most implementations the file name given is used directly in a Fortran open statement. If $file$ begins with the character '/', then it will be interpreted as an absolute file name. Otherwise, it will be assumed to be a path relative to the directory from which the Molpro has been launched. If, however, the file is not found, an attempt will be made instead to read it relative to the system lib/include directory, where any standard procedures may be found.

If the ECHO option is specified, the included file is echoed to the output in the normal way, but by default its contents are not printed. The included file may itself contain INCLUDE commands up to a maximum nesting depth of 10.



damien.francois@uclouvain.be
UCL/CISM - FNRS/CÉCI

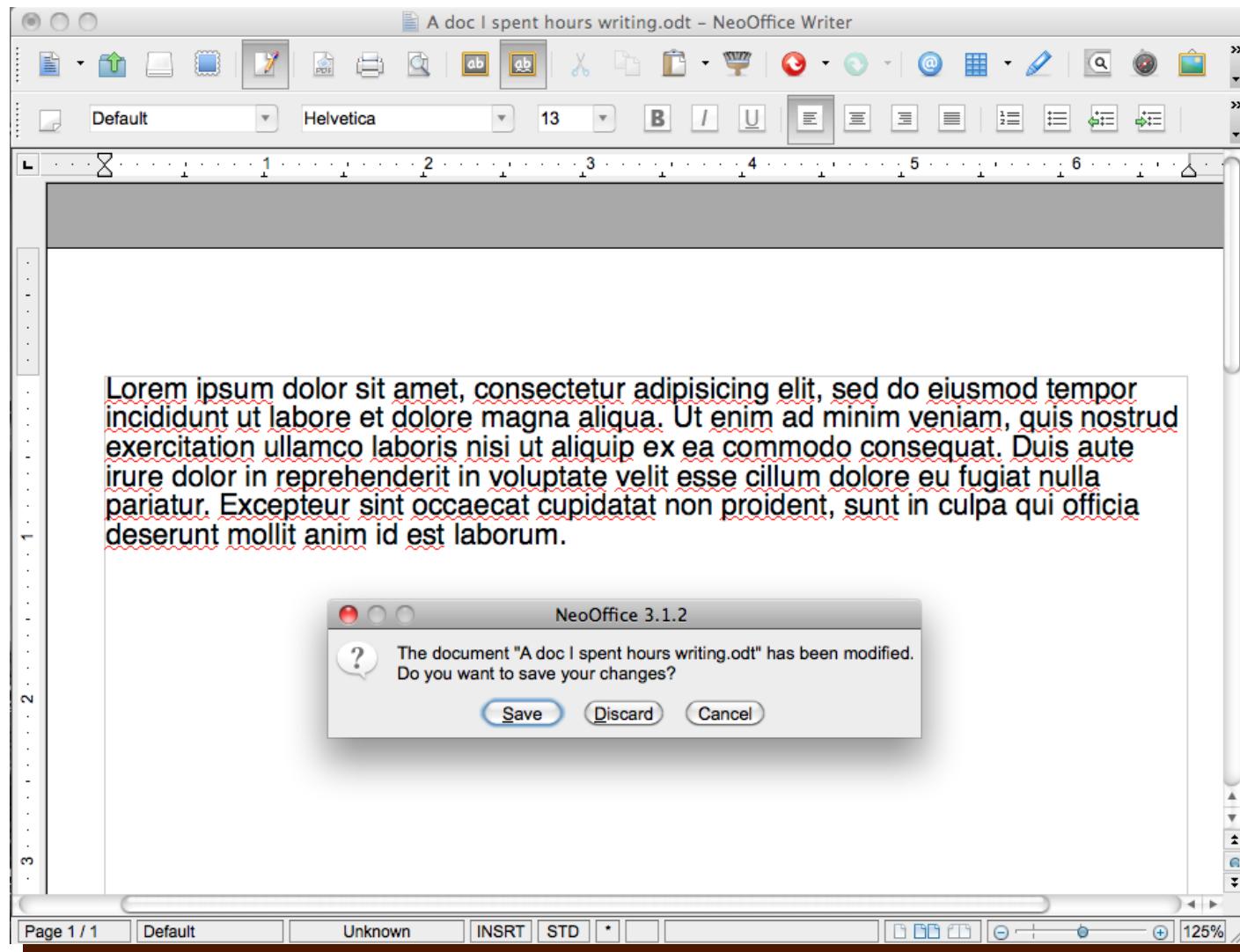


Summary, Wrap-up and Conclusions.

November 2013
CISM/CÉCI training session



Never click 'Discard' again...



- Application-based checkpointing
 - Efficient: save only needed data
 - Coarse temporal granularity: Good for fault tolerance, bad for preemption
 - Requires effort by programmer
- Library-based (DMTCP)
 - Portable across platforms
 - Transparent to application
 - Can't restore all resources
- Kernel-based checkpointing (BLCR)
 - Not portable
 - Transparent to application
 - Needs root access to install
 - Can save/restore all resources

- If you're the developer:
 - Make initializations conditional
 - Save minimal reconstructable state periodically
 - Save full workspace upon signal
 - Checkpoint after a synchronization

The submission script(s)

- Either one big one or two small ones
- Checkpoint periodically or --signal
- Requeue automatically
- Open-mode=append

BLCR, DMTCP, own recipe...

