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:  

```
1
2
3
```

With checkpointing:  

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

```
1
2
3
```

Without checkpointing:  

```
4
5
6
```

With checkpointing:  

```
1
2
3
```

```
4
5
6
```
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.
The key questions ...

<table>
<thead>
<tr>
<th>Transparency for developer</th>
<th>Portability to other systems</th>
<th>Size of state to save</th>
<th>Checkpointing overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Do I need to write a lot of additional code?</td>
<td>Can I stop on one system and restart on another?</td>
<td>How many GB of disk does it require?</td>
<td>How many FLOPs lost to ensure checkpointing?</td>
</tr>
</tbody>
</table>
Who's in charge of all that?

<table>
<thead>
<tr>
<th></th>
<th>Transparency for developer</th>
<th>Portability to other systems</th>
<th>Size of state to save</th>
<th>Checkpointing overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>the application itself</td>
<td>--</td>
<td>+++</td>
<td>--</td>
<td>-</td>
</tr>
<tr>
<td>a library</td>
<td>-</td>
<td>++</td>
<td>--</td>
<td>-</td>
</tr>
<tr>
<td>the compiler</td>
<td>+</td>
<td>++</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>a run-time</td>
<td>+</td>
<td>+</td>
<td>++</td>
<td>+</td>
</tr>
<tr>
<td>the OS</td>
<td>++</td>
<td>-</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>the hardware</td>
<td>+++</td>
<td>--</td>
<td>+++</td>
<td>+++</td>
</tr>
</tbody>
</table>
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
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 computations
    print(i)
    sleep(1)

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)
R recipe

```r
# R --slave --vanilla < count.R

the_start <- 1
the_end <- 10

for (i in seq(the_start, the_end)) {
  print(i)
  Sys.sleep(1)
}
```

```r
# R --slave --vanilla < count.R

# 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)
}
```
Octave recipe

count.m

doctave --silent --no-window-system < count.m

the_start = 1;
the_end = 10;

for i=the_start:the_end
  disp(i)
sleep(1)
end

% count.m
% Try to recover current state
try
  the_start = dlmread('state');
catch
  % Otherwise bootstrap at 1
  the_start = 1;
end_try_catch

the_end = 10;

for i=the_start:the_end
  % Save current state
dlmwrite('state', i)
  % Heavy computations
disp(i)
sleep(1)
end

crcount.m
% octave --silent --no-window-system < crcount.m

m

% Try to recover current state
try
  the_start = dlmread('state');
catch
  % Otherwise bootstrap at 1
  the_start = 1;
end_try_catch

the_end = 10;

for i=the_start:the_end
  % Save current state
dlmwrite('state', i)
  % Heavy computations
disp(i)
sleep(1)
end
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

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

end program crcount
```c
#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 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
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

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

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

<table>
<thead>
<tr>
<th>Signal</th>
<th>Number</th>
<th>Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGHUP</td>
<td>1</td>
<td>Exit</td>
<td>Hangup</td>
</tr>
<tr>
<td>SIGINT</td>
<td>2</td>
<td>Exit</td>
<td>Interrupt</td>
</tr>
<tr>
<td>SIGQUIT</td>
<td>3</td>
<td>Core</td>
<td>Quit</td>
</tr>
<tr>
<td>SIGILL</td>
<td>4</td>
<td>Core</td>
<td>Illegal Instruction</td>
</tr>
<tr>
<td>SIGTRAP</td>
<td>5</td>
<td>Core</td>
<td>Trace/Breakpoint Trap</td>
</tr>
<tr>
<td>SIGABRT</td>
<td>6</td>
<td>Core</td>
<td>Abort</td>
</tr>
<tr>
<td>SIGEMT</td>
<td>7</td>
<td>Core</td>
<td>Emulation Trap</td>
</tr>
<tr>
<td>SIGFPE</td>
<td>8</td>
<td>Core</td>
<td>Arithmetic Exception</td>
</tr>
<tr>
<td>SIGKILL</td>
<td>9</td>
<td>Exit</td>
<td>Killed</td>
</tr>
<tr>
<td>SIGBUS</td>
<td>10</td>
<td>Core</td>
<td>Bus Error</td>
</tr>
<tr>
<td>SIGSEGV</td>
<td>11</td>
<td>Core</td>
<td>Segmentation Fault</td>
</tr>
<tr>
<td>SIGSYS</td>
<td>12</td>
<td>Core</td>
<td>Bad System Call</td>
</tr>
<tr>
<td>SIGPIPE</td>
<td>13</td>
<td>Exit</td>
<td>Broken Pipe</td>
</tr>
<tr>
<td>SIGALRM</td>
<td>14</td>
<td>Exit</td>
<td>Alarm Clock</td>
</tr>
<tr>
<td>SIGTERM</td>
<td>15</td>
<td>Exit</td>
<td>Terminated</td>
</tr>
<tr>
<td>SIGUSR1</td>
<td>16</td>
<td>Exit</td>
<td>User Signal 1</td>
</tr>
<tr>
<td>SIGUSR2</td>
<td>17</td>
<td>Exit</td>
<td>User Signal 2</td>
</tr>
<tr>
<td>SIGCHLD</td>
<td>18</td>
<td>Ignore</td>
<td>Child Status</td>
</tr>
<tr>
<td>SIGPWR</td>
<td>19</td>
<td>Ignore</td>
<td>Power Fail/Restart</td>
</tr>
<tr>
<td>SIGWINC</td>
<td>20</td>
<td>Ignore</td>
<td>Window Size Change</td>
</tr>
<tr>
<td>SIGURG</td>
<td>21</td>
<td>Ignore</td>
<td>Urgent Socket Condition</td>
</tr>
<tr>
<td>SIGPOLL</td>
<td>22</td>
<td>Ignore</td>
<td>Socket I/O Possible</td>
</tr>
<tr>
<td>SIGSTOP</td>
<td>23</td>
<td>Stop</td>
<td>Stopped (signal)</td>
</tr>
<tr>
<td>SIGTSTP</td>
<td>24</td>
<td>Stop</td>
<td>Stopped (user)</td>
</tr>
<tr>
<td>SIGCONT</td>
<td>25</td>
<td>Ignore</td>
<td>Continued</td>
</tr>
<tr>
<td>SIGTTIN</td>
<td>26</td>
<td>Stop</td>
<td>Stopped (tty input)</td>
</tr>
<tr>
<td>SIGTTOU</td>
<td>27</td>
<td>Stop</td>
<td>Stopped (tty output)</td>
</tr>
<tr>
<td>SIGVTALRM</td>
<td>28</td>
<td>Exit</td>
<td>Virtual Timer Expired</td>
</tr>
<tr>
<td>SIGPROF</td>
<td>29</td>
<td>Exit</td>
<td>Profiling Timer Expired</td>
</tr>
<tr>
<td>SIGXCPU</td>
<td>30</td>
<td>Core</td>
<td>CPU time limit exceeded</td>
</tr>
<tr>
<td>SIGXFSZ</td>
<td>31</td>
<td>Core</td>
<td>File size limit exceeded</td>
</tr>
<tr>
<td>SIGWAITING</td>
<td>32</td>
<td>Ignore All LWP's blocked</td>
<td></td>
</tr>
<tr>
<td>SIGLWP</td>
<td>33</td>
<td>Ignore</td>
<td>Virtual Interprocessor Interrupt for Threads Library</td>
</tr>
<tr>
<td>SIGAI0</td>
<td>34</td>
<td>Ignore</td>
<td>Asynchronous I/O</td>
</tr>
</tbody>
</table>
UNIX processes can receive 'signals' from the user, the OS, or another process.

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

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

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 historically 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").
```c
#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);
    }
}
```

```c
// 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
#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
#include <stdio.h>
#include <signal.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 {
        // Otherwise bootstrap at 1
        the_start = 1;

        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);
    }
}
```
subroutine catch_signal
    logical :: interrupted
    common interrupted
    interrupted = .true.
end subroutine catch_signal

program crscount

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 crscount
program crcount

integer :: i, the_start, the_end
integer :: stat, sig

integer, parameter :: SIGINT = 2

integer :: watchsignalname, getlastsignal

! 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)

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))
Octave signal recipe

```octave
% 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_try_catch

the_end = 10;

while i <= the_end
  % Heavy computations
  disp(i)
  sleep(1)
  i = i + 1;
end
```
R --slave --restore < crsigcount.R
# killall -SIGUSR1 R

# Otherwise bootstrap at 1
if (!exists("i")) { i <- 1}

the_end <- 10

# transform in while loop
while (i <= the_end){
  # Heavy computations
  print(i)
  Sys.sleep(1)
  i <- i + 1
}
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 and 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

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

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

```bash
#!/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

Fortran signal recipe

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

d o  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
Then you can have your job re-queued automatically

```bash
#!/bin/bash
#SBATCH --job-name=test
#SBATCH --output=res
#SBATCH --open-mode=append
#SBATCH --time=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)"

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

"submit_signal_wrequeue.sh" 21l, 341C written
Note the `--open-mode=append`
Or chain the jobs...

- `--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).

- `--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.

```bash
#!/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
```
Use serialization tools and libraries for efficient and persistent data storage on disk
Standard data file formats allow browsing, postprocessing, and transmitting intermediate data.

<table>
<thead>
<tr>
<th>Data size</th>
<th>Storage type</th>
</tr>
</thead>
<tbody>
<tr>
<td>~10MB</td>
<td>CSV</td>
</tr>
<tr>
<td>~10GB</td>
<td>Zipped CSV or Binary</td>
</tr>
<tr>
<td>~100GB</td>
<td>HDF5, sqlite</td>
</tr>
<tr>
<td>~10TB</td>
<td>MongoDB, Postgres</td>
</tr>
</tbody>
</table>
Parallel programs are better checkpointed after a global synchronization.
In the fork-join model, checkpoint after a join and before a fork.

Checkpoint here

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

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

```c
#include <stdio.h>

main()
{
    printf("Hello world");
}

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
}
```
LD_PRELOAD magic

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 world
dfr@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 $
BLCR: the Berkely Labs Checkpoint/Restart for Linux works with a kernel module and a shared library.
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 kermita through 3.7.1.

January 14, 2013
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 (TGID), 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 an 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

```c
// 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);
    }
}
```
Run with `cr_run`; restart with `cr_restart`
The submission script looks for checkpoint and `cr_runs` or `cr_restarts` accordingly.

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

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

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

wait $thwaitpid
```

"submit.cr_run.sh" 26 lines --15%--
Two jobs are submitted
A checkpoint is created periodically
At restart, note `./count` still write to `res1` while the submission script writes to `res2`.
Alternatively, use a signal watchdog

```bash
#!/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.$SLURM_JOB_NAME.$theprocpid
```
SLURM Checkpoint/Restart with BLR

Overview

SLURM version 2.0 has been integrated with Berkeley Lab Checkpoint/Restart (BLR) 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 BLR's `cr_checkpoint` command and cancel the job. `srun_cr` will automatically checkpoint your job.
3. Restart `srun_cr` using BLR'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:** BLR operation has been verified with MVAPICH2. Some other MPI implementations should also work.

User Commands

The following documents SLURM changes specific to BLR 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...
DMTCP : Distributed MultiThreading CheckPointing works with an independent monitoring process and a shared library
DMTCP: Distributed MultiThreaded CheckPointing

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/cescope and 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@cs.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/escape, 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

```c
/*
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);
    }
}
*/
```
Run with dmtcp_launch
(runs monitoring daemon if necessary)
Restart with dmtcp_restart_script.sh
Launch the coordinator and the program with automatic checkpointing every 30 seconds
Launch coordinator and restart program

```
#!/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
```
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

slurmd[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%--
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.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChkBasis</td>
<td>Read the basis set from the checkpoint file.</td>
</tr>
<tr>
<td>SCF=Restart</td>
<td>Restart an SCF calculation from the checkpoint file. This is normally used when an SCF calculation failed finish for some reason.</td>
</tr>
<tr>
<td>IRC=Restart</td>
<td>Restarts an IRC calculation that did not complete, or restarts an IRC calculation for which additional points along the reaction path are desired.</td>
</tr>
<tr>
<td>Scan=Restart</td>
<td>Restarts a potential energy surface scan which did not complete.</td>
</tr>
<tr>
<td>Freq=Restart</td>
<td>Restarts a numerical frequency calculation which did not complete. Analytic frequency calculations cannot be restarted.</td>
</tr>
<tr>
<td>Polar=Restart</td>
<td>Restarts a numerical polarizability calculation which did not complete.</td>
</tr>
<tr>
<td>CIS=Restart</td>
<td>Restarts a CIS (Configuration Interaction - Single excitation) calculation which did not complete.</td>
</tr>
<tr>
<td>Opt=Restart</td>
<td>Restarts an geometry optimization which did not complete.</td>
</tr>
<tr>
<td>Geom=Checkpoint</td>
<td>Reads the molecular geometry from the checkpoint file.</td>
</tr>
<tr>
<td>Geom=AllCheckpoint</td>
<td>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.</td>
</tr>
<tr>
<td>Guess=Read</td>
<td>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.</td>
</tr>
<tr>
<td>Density=Checkpoint</td>
<td>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 resuming the job.</td>
</tr>
<tr>
<td>Field=Checkpoint</td>
<td>Reads the 34 multipole components of a finite field from the checkpoint file.</td>
</tr>
<tr>
<td>Field=EChk</td>
<td>Reads the 3 electric dipole field components from the checkpoint file.</td>
</tr>
<tr>
<td>Charge=Check</td>
<td>Reads point charges from the checkpoint file.</td>
</tr>
<tr>
<td>ReArchive</td>
<td>This option is used to generate an archive entry from the information in the checkpoint file. No calculation is run.</td>
</tr>
</tbody>
</table>
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 19.8.4.

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 model 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 point. 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:

• “Restarting 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 analysis,” 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.
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 any 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 STOP or EXIT cards would not be found.

6.3 Restarting a job (RESTART)

In contrast to MOLPRO2 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 NO RESTART option
can be used on the EXEC directive. The RESTART directive as described below can still be used as
in MOLPRO2, but is usually not needed.

RESTART, n1, n2, n3, ...

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

a) 0 < ni < 10: Restart file ni and remove all information.

b) ni = name.nnn: Restart file name.nnn but truncate before record name.

If all ni = 0, then all permanent files are restarted. However, if at least one ni 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([,zero])

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 the 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 NO RESTART 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.
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...