ITALC: Interactive Tool for Application-Level Checkpointing

HUST’17 @ SC17
11/11/17

PRESENTED BY:
Robert McLay
Email: mclay@tacc.utexas.edu

Ritu Arora
Email: rauta@tacc.utexas.edu

Trung Nguyen Ba
Email: tantrungnb@gmail.com
Sorry that we could not be present at the workshop in 3-D!

Ritu Arora, High Performance Computing group, TACC

Trung Nguyen Ba, previously at TACC, now at the University of Massachusetts-Amherst
As HPC Systems Evolve…

New policies governing the system usage are put in place

• How many compute nodes can a user access at a given time, and for how long?

• How many MPI tasks could be running on a compute node at a given time?

What happens when an application does not complete running within the maximum allowed time for a job in a system queue?

• We get queries as shown on the following slide 😊
Samples of User Requests for Increasing the Time Limit in the Queues

[Category] Running jobs or Using TACC Resources
[Resource] Lonestar5

October 2, 2017

Hi,

We have a job to run on one node that is hitting the 48 hour time limit. Is it possible to extend that limit to 96 hours for this one job?

Thanks,

---

[Category] Running jobs or Using TACC Resources
[Resource] Lonestar5

October 5, 2017

Hi,

I was running some simulations on lonestars and found that some simulations ended before converge due to the 48 hour limit on the queued jobs.

I was wondering whether there is a way for me to run simulations without this time limit.
While We Do Accommodate Special Cases for Extending Job Time Limits, We Mostly Advice as Follows:

“

We are afraid we do not change the time limits on jobs as the requests would quickly become unmanageable given our user volume. We strongly recommend that you use any checkpointing/restart capabilities of your code to avoid wasting computational time.

”
There is Also a Need for Making the Applications Fault-Tolerant

As we get ready for the exascale computing era, the Mean Time Between Failure (MTBF) of hardware components is likely to decrease

- What happens to a job when there is a transient disruption to the services due to a network failure?

- Mechanisms for making the applications resilient to certain types of expected or unexpected failures (such as, time out from the job queue, interconnect failure, or a compute node crash) are needed for saving the compute cycles and preventing an enormous increase in the overall time-to-results

- Checkpointing is one such mechanism
What is Checkpointing?

• If a job gets timed out from the queue while leaving its computation incomplete, a subsequent job can be submitted to resume the computation using one of the previously saved states of the application

• The periodically saved execution state of an application that can be used to resume it after an interruption is known as a checkpoint

• Resuming the execution of an application using a previously saved state (instead of starting all over again) is referred to as the restart phase

• There are different types of checkpointing such as system-level, user-level or library-level, and application-level
Types of Checkpointing

- **System-Level Checkpointing**
  - **Pros:** Convenient to use, no code changes needed, user only specifies the checkpointing frequency
  - **Cons:** Large memory-footprint of checkpoints as the entire execution state of the application and the operating system processes are saved during checkpointing, system administrator level privileges needed for installation
  - **Example:** Berkeley Lab Checkpointing and Restart (BLCR)

- **Library-Level or User-Level Checkpointing**
  - **Pros:** Useful for checkpointing applications without requiring any changes to their source-code or the operating system kernel
  - **Cons:** Users may need to load the checkpointing library before starting their applications, and then, would need to dynamically link the loaded library to their applications, checkpoints have a large memory-footprint
  - **Example:** DMTCP

- **Application-Level Checkpointing:** next slide
What is Application-Level Checkpointing?

• When the checkpoint-and-restart mechanism is built within the application itself, it is called Application-Level Checkpointing
  • An efficient implementation of this would require saving and reading the state of only those variables or data that are necessary for recreating the state of the entire application. Such variables or data are referred to as critical variables/data

```c
int main(){
    int x = 4;
    int y = sqrt(x);
    int z, i;
    j = x*y;
    for (i =0; i< 100; i++){
        z += j* myFct(randomNumber* i);
    }
    return 0;
}
```

i and z are critical variables as their values are updated and cannot be derived easily to recreate the execution state of an interrupted program
Pros and Cons of Application-Level Checkpointing

• It does not rely on the availability of any external libraries or tools, and hence, is useful for writing portable applications.

• While an efficient implementation of this technique will generate checkpoints with smaller memory footprint and incur lesser I/O overheads as compared to other types of checkpointing, the onus is on the user (or the developer) to manually implement it on a per application basis.
  • This implies that the users should understand the code of the applications that they are checkpointing.
  • Manual reengineering of existing code to insert checkpoint-restart logic is required, and this can be an error-prone and a time-consuming activity.
Interactive Tool for Application-Level Checkpointing (ITALC)

- ITALC can reengineer the existing serial and parallel applications (C/C++/MPI/OpenMP) to insert the checkpoint-and-restart functionality in them
  - Fortran and Python applications will be supported in future

- It is a command-line tool that depends upon the user-specifications to reengineer the existing code

- Users provide the checkpointing specifications – that is, what to checkpoint, where to checkpoint, and the checkpointing frequency – in an interactive manner
ITALC in Action

1. ITALC requires the application source code and the checkpointing specifications as user-input.

2. On the basis of the input provided to it, ITALC performs static-code analysis and generates the output code that can checkpoint-and-restart.

3. It uses the ROSE source-to-source compiler and the build-in heuristics to carry out the required code transformation.
Using ITALC

1. Invoking the ALC tool with file named md.c

2. Choosing the code region to checkpoint

Which would you like to checkpoint?
(0) for-loops only
(1) individual lines only
(2) both

3. Choosing the loop to insert checkpoint blocks

REGION 3 (from function dist, line 168 to 171)

for (i = 0; i < nd; i++) {
    dr[i] = (r1[i] - r2[i]);
    d = (d + (dr[i] * dr[i]));
}

Would you like to checkpoint this for loop system? (y/n) y

Variables/Arrays that can be checkpointed in this loop system above:
(1) dr
(2) d

Are there any variables in this list that you would NOT like to checkpoint? (for
mat: #, #, #... or no for no input)

4. Generated code (with ALC support) that can be compiled and executed

c557-003.stampede(9)$ ls rose_md.c
rose_md.c
Steps for Running the Code Checkpointed Using ITALC, Interrupting it, and Resuming

Running the generated code first time and killing it with CTRL+C:
c558-901.stampede(28)$ ibrun -np 6 rpn
TACC: Starting up job 8165029
25 January 2017 09:20:16 PM
PRIME_MPI
C/MPI version
The number of processes is 6

<table>
<thead>
<tr>
<th>N</th>
<th>Pi</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0.021126</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0.003351</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>0.003321</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>0.010931</td>
</tr>
<tr>
<td>16</td>
<td>6</td>
<td>0.003320</td>
</tr>
<tr>
<td>32</td>
<td>11</td>
<td>0.003278</td>
</tr>
<tr>
<td>64</td>
<td>18</td>
<td>0.020993</td>
</tr>
<tr>
<td>128</td>
<td>31</td>
<td>0.003285</td>
</tr>
<tr>
<td>256</td>
<td>54</td>
<td>0.003277</td>
</tr>
<tr>
<td>512</td>
<td>97</td>
<td>0.108019</td>
</tr>
<tr>
<td>1024</td>
<td>172</td>
<td>0.003502</td>
</tr>
<tr>
<td>2048</td>
<td>309</td>
<td>0.004040</td>
</tr>
<tr>
<td>4096</td>
<td>564</td>
<td>0.796040</td>
</tr>
<tr>
<td>8192</td>
<td>1028</td>
<td>0.010968</td>
</tr>
<tr>
<td>16384</td>
<td>1900</td>
<td>0.035597</td>
</tr>
</tbody>
</table>

^C[c558-901.stampede.tacc.utexas.edu:mpirun_rsh][signal_processor] Caught signal 2, killing job
TACC: MPI job exited with code: 1

Running the generated code second time with the restart flag:
c558-901.stampede(29)$ ibrun -np 6 rpn --restart
TACC: Starting up job 8165029
25 January 2017 09:20:33 PM
PRIME_MPI
C/MPI version
The number of processes is 6

<table>
<thead>
<tr>
<th>N</th>
<th>Pi</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>32768</td>
<td>3512</td>
<td>3.084043</td>
</tr>
<tr>
<td>65536</td>
<td>6542</td>
<td>0.461787</td>
</tr>
<tr>
<td>131072</td>
<td>12251</td>
<td>1.732492</td>
</tr>
<tr>
<td>262144</td>
<td>23000</td>
<td>55.405050</td>
</tr>
</tbody>
</table>

PRIME_MPI - Master process:
Normal end of execution.

25 January 2017 09:21:34 PM
TACC: Shutdown complete. Exiting.
Run-time Comparison: Code without Checkpointing, Manual Checkpointing, & Using ITALC

NC: No Checkpointing
CHKPT-I: Checkpoint Using ITALC
CHKPT-M: Checkpoint Manually
Comparison of Number of Lines of Code Inserted Manually and with ITALC

<table>
<thead>
<tr>
<th>Application</th>
<th>NC (# LoC)</th>
<th>CHKPT-I (# LoC)</th>
<th>CHKPT-M (# LoC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD, Serial</td>
<td>712</td>
<td>812</td>
<td>724</td>
</tr>
<tr>
<td>MD, MPI</td>
<td>558</td>
<td>875</td>
<td>763</td>
</tr>
<tr>
<td>MD, OpenMP</td>
<td>626</td>
<td>869</td>
<td>773</td>
</tr>
<tr>
<td>Prime Number, MPI</td>
<td>247</td>
<td>545</td>
<td>434</td>
</tr>
<tr>
<td>Dijkstra, OpenMP</td>
<td>588</td>
<td>813</td>
<td>726</td>
</tr>
</tbody>
</table>

NC: No checkpointing
CHKPT-I: Checkpointing using ITALC
CHKPT-M: Manual checkpointing
Files Generated Using ITALC Have Smaller Memory Footprint as Compared to DMTCP

<table>
<thead>
<tr>
<th>Name</th>
<th>Date Modified</th>
<th>Size</th>
<th>Kind</th>
</tr>
</thead>
<tbody>
<tr>
<td>rose_prime_number.c</td>
<td>1:06 AM</td>
<td>19 KB</td>
<td>C source code</td>
</tr>
<tr>
<td>.save.LISTOFFUNCTION.o.txt</td>
<td>1:07 AM</td>
<td>38 bytes</td>
<td>text</td>
</tr>
<tr>
<td>.save.LISTOFLoop.o.txt</td>
<td>1:07 AM</td>
<td>104 bytes</td>
<td>text</td>
</tr>
<tr>
<td>.save1.prime_number.o.txt</td>
<td>1:08 AM</td>
<td>448 bytes</td>
<td>text</td>
</tr>
<tr>
<td>.save2.prime_number.o.txt</td>
<td>1:08 AM</td>
<td>384 bytes</td>
<td>text</td>
</tr>
<tr>
<td>NUMFILE_main.txt</td>
<td>1:05 AM</td>
<td>4 KB</td>
<td>text</td>
</tr>
<tr>
<td>ROSE_SAVE_FUNCTION_prime_number.o.txt</td>
<td>1:08 AM</td>
<td>128 bytes</td>
<td>text</td>
</tr>
<tr>
<td>VarLog_.save1.prime_number.o.txt</td>
<td>1:05 AM</td>
<td>57 bytes</td>
<td>text</td>
</tr>
<tr>
<td>VarLog_.save2.prime_number.o.txt</td>
<td>1:05 AM</td>
<td>37 bytes</td>
<td>text</td>
</tr>
</tbody>
</table>

DMTCP

- rwx------ 1 rauta G-25072 2.3M Nov 9 2016 ckpt_dmtcp1_452457cb737c5a44-40000-377a61f1b7365.dmtcp
- lrwxrwxrwx 1 rauta G-25072 60 Nov 9 2016 dmtcp_restart_script.sh -> dmtcp_restart_script_452457cb737c5a44-40000-377a62c2e244a.sh
- rwx------ 1 rauta G-25072 6.6K Nov 9 2016 dmtcp_restart_script_452457cb737c5a44-40000-377a62c2e244a.sh
References


Thank You!

We are grateful for the support received through:

• TACC STAR Scholars program

• Extreme Science and Engineering Discovery Environment (XSEDE) - NSF grant # ACI-105357
Any Question, Comments, or Concerns about ITALC?

For Details, Please Contact:
Ritu Arora at the following email: rauta@tacc.utexas.edu
Pseudo-code for checkpointing a for-loop

```c
for (int i = 0; i < 100; i++) { b(); }

->

int checkpoint_frequency;
int rose_restart;
/*additional code for setting the value of variable rose_restart to the value of i read from the restart file and setting the value of the variable checkpoint_frequency */
for (int i = rose_restart; i < 100; i++) {
    /*additional code for tracking the call to the function b()—where exactly in b() should the code start executing again—and setting any values if required */
    b();
    if (i % checkpoint_frequency == 0){
        //code for saving the critical variables to a file
    }
}
```