TEXAS ADVANCED COMPUTING CENTER
WWW.TACC.UTEXAS.EDU

TEXAS
The University of Texas at Austin

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

```
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

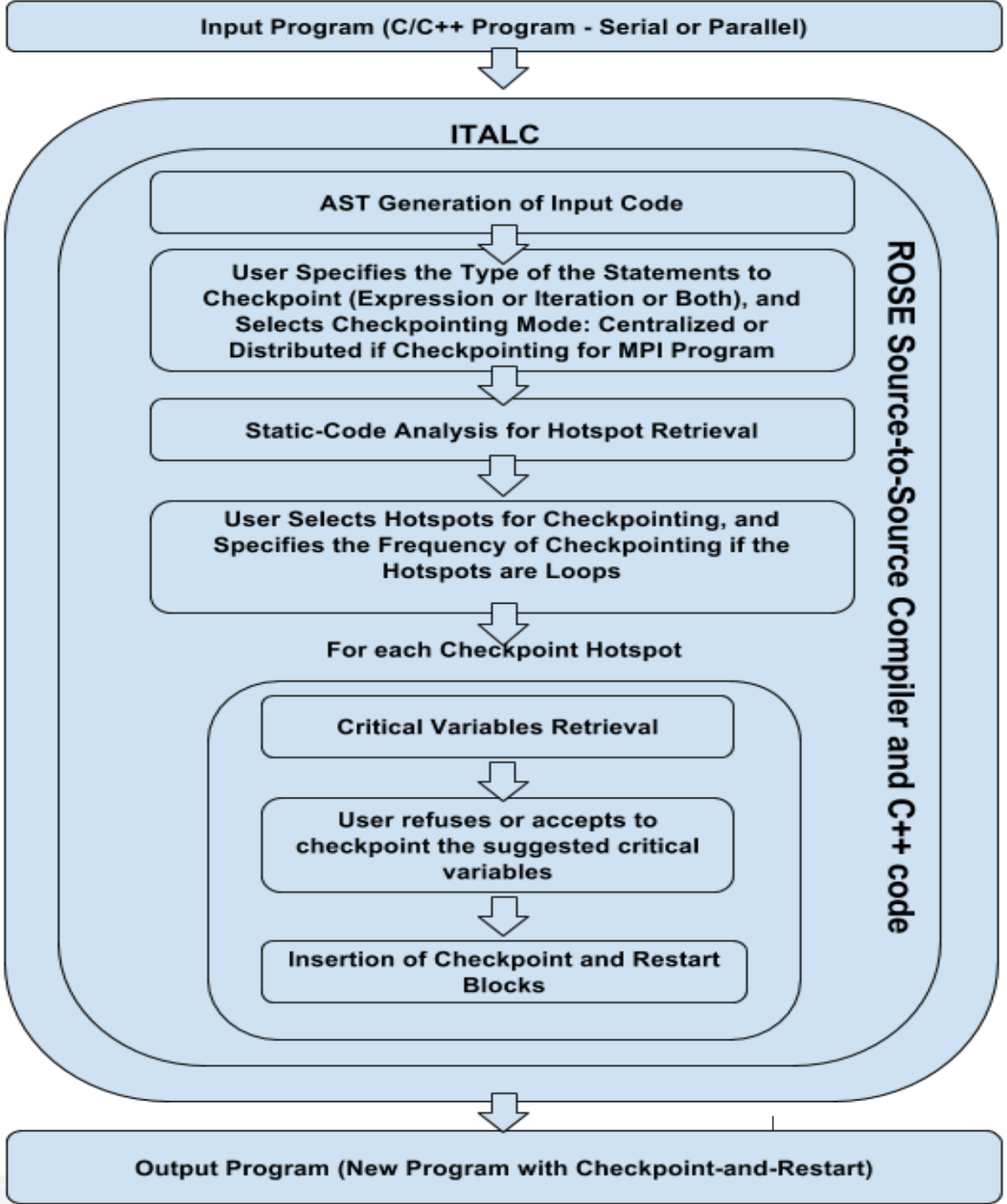
- Users provide the checkpointing specifications – that is, what to checkpoint, where to checkpoint, and the checkpointing frequency – in an interactive manner

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 checkpoined 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

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

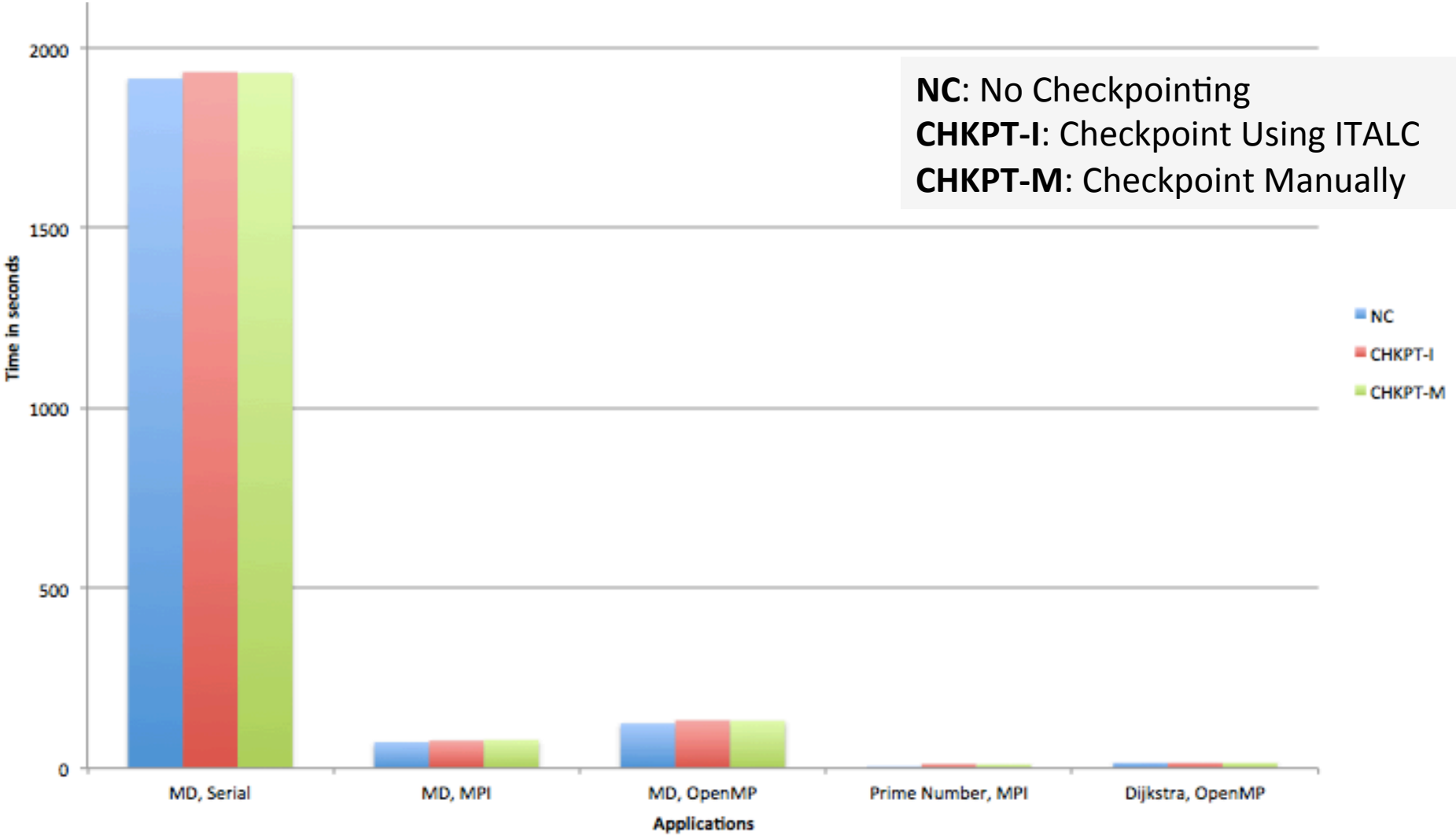
```
c558-901.stampede(29)$ ibrun -np 6 rpn --r
TACC: Starting up job 8165029
25 January 2017 09:20:33 PM
PRIME_MPI
  C/MPI version
  The number of processes is 6

         N        Pi          Time
     32768      3512      3.084043
     65536      6542      0.461787
    131072     12251      1.732492
    262144     23000     55.405050

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



Comparison of Number of Lines of Code Inserted Manually and with ITALC

Application	NC (# LoC)	CHKPT-I (# LoC)	CHKPT-M (# LoC)
MD, Serial	712	812	724
MD, MPI	558	875	763
MD, OpenMP	626	869	773
Prime Number, MPI	247	545	434
Dijkstra, OpenMP	588	813	726

NC: No checkpointing

CHKPT-I: Checkpointing using ITALC

CHKPT-M: Manual checkpointing

Files Generated Using ITALC Have Smaller Memory Footprint as Compared to DMTCP

ITALC

Name	Date Modified	Size	Kind
rose_prime_number.c	1:06 AM	19 KB	C source code
.save.LISTOFFUNCTION.o.txt	1:07 AM	38 bytes	text
.save.LISTOFLOOP.o.txt	1:07 AM	104 bytes	text
.save1.prime_number.o.txt	1:08 AM	448 bytes	text
.save2.prime_number.o.txt	1:08 AM	384 bytes	text
NUMFILE_main.txt	1:05 AM	4 KB	text
ROSE_SAVE_FUNCTION_prime_number.o.txt	1:08 AM	128 bytes	text
VarLog_.save1.prime_number.o.txt	1:05 AM	57 bytes	text
VarLog_.save2.prime_number.o.txt	1:05 AM	37 bytes	text

DMTCP

```
-rw-------  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

1. Trung Nguyen, Ritu Arora, “A Tool for Semi-Automatic Application-Level Checkpointing”: http://sc16.supercomputing.org/sc-archive/tech_poster/poster_files/post228s2-file3.pdf

2. Ritu Arora, Purushotham Bangalore, Marjan Mernik: A technique for non-invasive application-level checkpointing. The Journal of Supercomputing 57(3): 227-255 (2011)

3. Ritu Arora, Marjan Mernik, Purushotham Bangalore, Suman Roychoudhury, Saraswathi Mukkai: A Domain-Specific Language for Application-Level Checkpointing. ICDCIT 2008: 26-38

4. Jason Ansel, Kapil Arya, and Gene Cooperman. 2009. DMTCP: Transparent checkpointing for cluster computations and the desktop. In Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing (IPDPS '09). IEEE Computer Society, Washington, DC, USA, 1-12. DOI=10.1109/IPDPS.2009.5161063

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

```
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
  }
}
```