A Tool for Semi-Automatic Application-Level Checkpointing

Trung Nguyen Ba, Ritu Arora

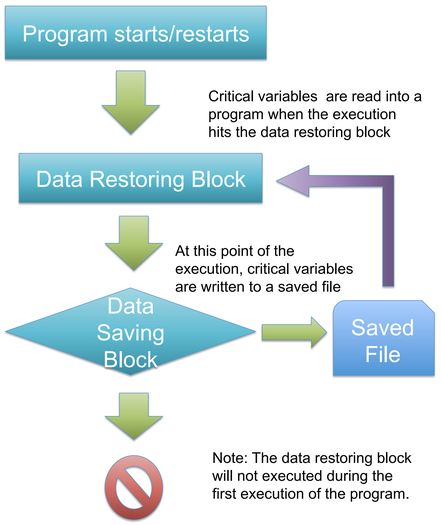
Texas Advanced Computing Center, The University of Texas at Austin

Introduction

High-Performance Computing (HPC) resources at open-science data centers often have general system policies according to which a computational job has limitations in terms of the maximum number of compute nodes it can use and the maximum amount of time it can stay in the job queue. However, many large-scale application runs are unable to complete within the allowed limits. To overcome the limitations of queue policies, the applications can be checkpointed so that their execution state can be saved on the hard disk before it times out of the queue. The application can then be restarted from the saved state to resume the computation from the point where it stopped. Application-level checkpointing is useful for this purpose.

Application-Level Checkpointing (ALC)

ALC is useful for periodically saving the state of critical variables in an application. If a checkpointed application is interrupted during its execution, then it can resume the computation from one of the saved states instead of starting from scratch. ALC saves the state of only critical variables and hence, is more memory efficient as compared to system-level checkpointing.



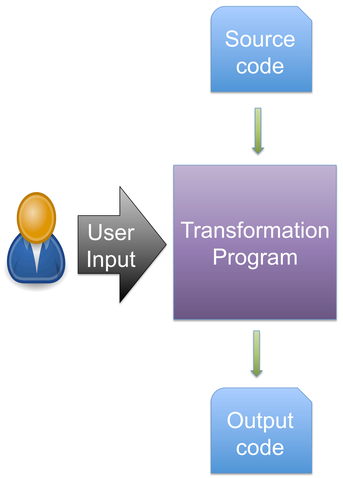
General Structure of Checkpointed Program

Overview of Code Transformation for Checkpointing

The user provides the source code to checkpoint and provides other required specifications for ALC. The user can choose to checkpoint : (1) for-loops, (2) program statements other than for-loops, and (3) both for-loops and other program statements.

The tool prompts the user for specifications such as: (1) the region of code to checkpoint, (2) the frequency of checkpointing, and (3) the choice to accept or decline checkpointing critical variables that it has short-listed.

On the basis of the specifications provided, the program transformation engine that is part of the tool generates a new output file with the checkpoint and restart code inserted.



The Tool in Action

Example of Checkpointing For-Loops

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

```
)$ alc ./testing/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
```

Application Restart

The program is interrupted at one point during the 16th iteration

```
c557-002.stampede(13)$ ./rose_md 2 1000 1000 0.1
Not running in restart mode.
22 July 2016 12:10:56 AM

MD
  C version
  A molecular dynamics program.

  ND, the spatial dimension, is 2
  NP, the number of particles in the simulation, is 1000
  STEP_NUM, the number of time steps, is 1000
  DT, the size of each time step, is 0.100000

  At each step, we report the potential and kinetic energies.
  The sum of these energies should be a constant.
  As an accuracy check, we also print the relative error
  in the total energy.

      Step      Potential       Kinetic        (P+K-E0)/E0
                Energy P        Energy K       Relative Energy Error

         0    489081.460437         0.000000    0.000000e+00
         1    489081.460437       111.618768    2.282586e-04
         2    487997.031659      1084.568916    2.865784e-07
         3    485499.101699      3071.793884   -8.803345e-04
         4    481386.095649      7265.514138   -7.135984e-04
         5    478773.428419     14017.533496    7.749460e-03
         6    481188.084938     21325.046874    7.763115e-02
         7    484257.504958     26586.656346    4.466797e-02
         8    485357.562295     29096.511834    5.205818e-02
         9    486036.558637     30332.795991    5.596690e-02
        10    486967.496679     31170.429626    5.958360e-02
        11    488069.666861     31658.256068    6.283511e-02
        12    490085.758295     31622.137550    6.688413e-02
        13    491750.368101     31052.397652    6.912312e-02
        14    493047.838709     30116.052867    6.986160e-02
        15    494025.707490     29245.496157    7.008107e-02
        16    494742.862686     28591.529496    7.021020e-02
        17    495371.637900     28171.258302    7.063667e-02
        18    495960.122114     27832.543324    7.114745e-02
^C
c557-002.stampede(14)$ 
```

During the restart mode, the program begins computation from the last checkpoint that it saved.

```
c557-002.stampede(14)$ ./rose_md 2 1000 1000 0.1 --r
Restart initated.
22 July 2016 12:15:01 AM

MD
  C version
  A molecular dynamics program.

  ND, the spatial dimension, is 2
  NP, the number of particles in the simulation, is 1000
  STEP_NUM, the number of time steps, is 1000
  DT, the size of each time step, is 0.100000

  At each step, we report the potential and kinetic energies.
  The sum of these energies should be a constant.
  As an accuracy check, we also print the relative error
  in the total energy.

      Step      Potential       Kinetic        (P+K-E0)/E0
                Energy P        Energy K       Relative Energy Error

Restarting from file string1.
        10    486967.496679     31170.429626    5.958360e-02
        11    488069.666861     31658.256068    6.283511e-02
        12    490085.758295     31622.137550    6.688413e-02
        13    491750.368101     31052.397652    6.912312e-02
        14    493047.838709     30116.052867    6.986160e-02
        15    494025.707490     29245.496157    7.008107e-02
        16    494742.862686     28591.529496    7.021020e-02
        17    495371.637900     28171.258302    7.063667e-02
        18    495960.122114     27832.543324    7.114745e-02
```

Sample Code Checkpointed by the Tool

Original Code

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

Note: The inserted code was manually modified for simplification and clarity.

Code after checkpoint and restart blocks are inserted

```
{
    /** DATA RESTORING BLOCK **/
    int Rose_lowerLimit_0;
    FILE *ROSE_FILE;
    int arrIterator;
    char fname[16] = ".save3.md.o.txt";
    // variable to determine if this loop is ever checkpointed or not
    int written_loop_ = 0;
    char *string1;
    string1 =(char *) malloc((strlen(fname) + 1) * sizeof(char));
    strcpy(string1, fname);
    if (ROSE_RESTART_FLAG != 0) // if restart mode is turned on
    {
        ROSE_FILE = fopen(string1, "rb");
        if (ROSE_FILE == NULL) {
            printf("Error opening file string1. Cannot restart. Defaulting to Normal mode.\n");
            ROSE_RESTART_FLAG = 0;
        } else {
            int fread_checking_ =
                fread(&written_loop_, sizeof(int), 1,ROSE_FILE);
            if (written_loop_ == 1 && fread_checking_ == 1) {
                printf("Restarting from file string1.\n");
                fread(&Rose_lowerLimit_0, sizeof(int), 1, ROSE_FILE);
                /** RESTART d dr **/
                fread(&d, sizeof(double), 1, ROSE_FILE);
                fread(dr, sizeof(double), (nd), ROSE_FILE);
                fclose(ROSE_FILE);
                /** END RESTART d dr **/
            } else {
                printf("file empty or cannot open file: %s\n",string1);
            }
        }
    }
    for (i =(written_loop_ ) ? Rose_lowerLimit_0 : 0; i < nd; i++) {
        {
            Rose_lowerLimit_0 = i;
        }

        dr[i] = (r1[i] - r2[i]);
        d = (d + (dr[i] * dr[i]));
        /** DATA SAVING BLOCK **/
        if ((i % (10) == 0) && FUNCTION_RELATING_CONDITION_dist) {
            ROSE_FILE = fopen(string1, "wb");
            //Turn written on
            int __w__ = 1;
            fwrite(&__w__, sizeof(int), 1, ROSE_FILE);
            Rose_lowerLimit_0 +=1;
            fwrite(&Rose_lowerLimit_0, sizeof(int), 1, ROSE_FILE);
            /** SAVE d dr **/
            fwrite(&d, sizeof(double), 1, ROSE_FILE);
            fwrite(dr, sizeof(double), (nd),ROSE_FILE);
            fclose(ROSE_FILE);
            /** END SAVE d dr **/
        }

    }
    /** END CHECKPOINTED BLOCK **/ }
```

Data restore block (reading the checkpoint during restart)

```
/** DATA RESTORING BLOCK **/
int Rose_lowerLimit_0;
FILE *ROSE_FILE;
int arrIterator;
char fname[16] = ".save3.md.o.txt";
// variable to determine if this loop is ever checkpointed or not
int written_loop_ = 0;
char *string1;
string1 =(char *) malloc((strlen(fname) + 1) * sizeof(char));
strcpy(string1, fname);
if (ROSE_RESTART_FLAG != 0) // if restart mode is turned on
{
    ROSE_FILE = fopen(string1, "rb");
    if (ROSE_FILE == NULL) {
        printf("Error opening file string1. Cannot restart. Defaulting to Normal mode.\n");
        ROSE_RESTART_FLAG = 0;
    } else {
        int fread_checking_ =
            fread(&written_loop_, sizeof(int), 1,ROSE_FILE);
        if (written_loop_ == 1 && fread_checking_ == 1) {
            printf("Restarting from file string1.\n");
            fread(&Rose_lowerLimit_0, sizeof(int), 1, ROSE_FILE);
            /** RESTART d dr **/
            fread(&d, sizeof(double), 1, ROSE_FILE);
            fread(dr, sizeof(double), (nd), ROSE_FILE);
            fclose(ROSE_FILE);
            /** END RESTART d dr **/
        } else {
            printf("file empty or cannot open file: %s\n",string1);
        }
    }
}
```

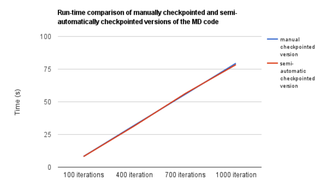
Data saving block (writing the checkpoint)

```
/** DATA SAVING BLOCK **/
if ((i % (10) == 0) && FUNCTION_RELATING_CONDITION_dist) {
    ROSE_FILE = fopen(string1, "wb");
    //Turn written on
    int __w__ = 1;
    fwrite(&__w__, sizeof(int), 1, ROSE_FILE);
    Rose_lowerLimit_0 +=1;
    fwrite(&Rose_lowerLimit_0, sizeof(int), 1, ROSE_FILE);
    /** SAVE d dr **/
    fwrite(&d, sizeof(double), 1, ROSE_FILE);
    fwrite(dr, sizeof(double), (nd),ROSE_FILE);
    fclose(ROSE_FILE);
    /** END SAVE d dr **/
}
```

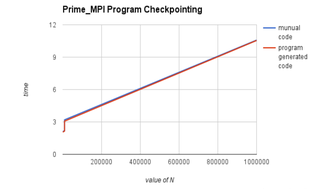
Results and Discussion

There was no significant variation in the performance of the semi-automatically checkpointed version and manually checkpointed version of the serial and parallel source code.

Graph showing the performance of two checkpointed version of "md.c" code which have the same checkpoint location



Graph showing the performance of two checkpointed version of "prime_mpi.c" code which have the same checkpoint location



Currently only C/C++ programs are supported by the tool. In future, the program will also be able to support Fortran and Python programs.

Currently, the tool has limited support for checkpointing MPI programs. More work is required in order to generalize the usage of the tool for checkpointing MPI and OpenMP programs.

Acknowledgments

We are grateful to TACC and XSEDE for providing the resources for this project. XSEDE is funded by the National Science Foundation (NSF) through award ACI-1053575.