CSC 548: Project

PRASUN RATN (pratn@ncsu.edu)
http://www4.ncsu.edu/~pratn/548

Implementing computation timing deltas on the MPI Trace Compression Framework replay engine

Background

This project builds on the work that has been done to capture MPI call information and record them in a compressed form. These traces are later replayed using the replay engine. The aim of the project was to add timing information into MPI traces and use that information while replaying the traces. This essentially simulates the computation time between any two MPI calls.e

Timing delta

First of all, the timing information is captured during the MPI program execution. This is done within the record engine. This work is being done in a separate project.
(See http://www4.ncsu.edu/~akdalmia/Projects/itdmpit.html)
This work intends to cover using the timing delta information while replaying the trace.

A little bit of introduction is required as to what exactly is meant by capturing timing deltas. A program whose trace is required is linked against the record engine and allowed to run. The record engine has wrappers, through which it catches all MPI calls. Within these wrappers, the amount of time spent in computation since the last MPI call is computed. This value needs to be stored. However, the values differ from node to node and thus it is not possible to store timing information for all nodes.

Capturing timing delta

The simple solution is to capture this information statistically by computing minimum, average and maximum values. This is added as one line per MPI call entry in the trace file in the form
TIME: <min> <avg> <max>. For example -

`TIME: 13926 180540 383426` (all values in microseconds).

The alternate solution is to store a histogram, instead of storing just the average. Here, the minimum is chosen as the base, and a stride is chosen depending upon the number of buckets. The number of buckets can be a parameter to the program. The format now becomes
TIME: <base> <stride> <size1> <size2> ...<size5>. For example -

`TIME: 5736 211 3 1 0 6 4` (all values in microseconds).

Here the bucket values indicate the number of nodes that fall in that particular range.

Replaying timing delta

Once we have the timing information in the trace, they need to be replayed. Here it is assumed that the replay engine is compiled with the same option (min-avg-max or histogram) that was used to generate the traces. It is trivial to put a indicator flag based on which the replay engine can be notified of the format. The first change in the replay engine is in the trace parser. The parse needs to look for timing information while parsing the trace file (needless to say, the appropriate data structures must be already modified). This timing information is stored in the RSD list. At the time of actual replay, these values are read and usleep is called to simulate computation delay.

Replaying min-avg-max trace

Here we need to mention the exact time used to sleep. For the min-avg-max traces, **only the average value is used** at all times. This results in a situation where all the MPI tasks sleep for the exact same time. Due to this, the load imbalance effects are not seen.

Replaying Histogram trace

When we have a histogram trace, a random value between 0-<#nodes> is generated. This value is used to **select the appropriate bucket** in the histogram. The sleep time is then simply computed from the base,stride and the bucket index. This value is then used to simulate computation delay.

Implementation

The only data structure modified was the replay_op structure. A few fields were added to store the min-avg-max deltas and the histogram values as follows :-

```
#if defined(TIMING_DELTA)
enum
{
  MIN_TIME,
  AVG_TIME,
  MAX_TIME,
  TIME_MAX
};
#elif defined(TIMING_HISTO)
#define TIMING_HISTO_BUCKET_LEN 5
#endif
/* binary data structure for ops */
typedef struct {
      ..............
#if defined(TIMING_DELTA)
  /* timing info */
  int time[TIME_MAX];
#elif defined(TIMING_HISTO)
  int timing_base;
  int timing_stride;
  int histo_values[TIMING_HISTO_BUCKET_LEN];
#endif
  ..............
} replay_op;
```

Implementation details

The first step was to compile and run the existing record/replay engines.

The existing source consisted of the following :-
`record.tgz` - the new record engine
`replay.tgz` - the new replay engine
`src.tar.gz`- the old record and replay engine.

Firstly, the old record engine was compiled and run to generate trace files. In this process, a few minor bugs were discovered and fixed . However, it was later observed that those bugs were already fixed in the new record engine code. So it was decided that the new record engine be used.. In the meantime, the other team informed that the new replay engine did not have implementation of all the MPI calls, so the old replay engine was used.

Record engine:

- First of all, the record makefile needed to be modified in order for the source to be compiled. The main changes were in the source path prefix and the compiler path and flags. Also **minor changes were made to fix the way the makefile was written** (the default target was 'clean'!).
- After doing this the source was compiled and the record engine are run for small sample programs. It was noticed that the program did not execute to completion. Some debugging indicated that this was due to some code under one of the debug flags. The **debug flag was turned off** and the program was able to run to completion.
- On doing this, the trace files were produced. It was noticed that the **trace files had certain binary characters in the signature** field. The stack signature is generated using the Program Counters from the call stack. The location of PC, FP etc are platform dependent, and macros are used to access these values. These **macros needed to be tweaked** a little. At this point, a working record engine was ready.
- This work was done on an x86 laptop. When the same code was run on the OS cluster, the program failed to run, crashing due to a segmentation failure. On much debugging, it was found that due to compiler optimizations, the stack signature was not being read correctly and the program was trying to read from invalid memory addresses. This was fixed by **turning off optimizations while compiling the replay engine**.
- **Timing delta information:** Once a working version of the record engine was created, the code was modified to output timing delta information. The actual capture of timing delta was done in a separate project, **so some test code was added to generate random deltas** in the range 0-500ms. There are two timing delta formats :- min/avg/max and histogram. The code for generating each type of timing information is under a different compiler switch.

Replay engine

- Similar to the record engine, **the makefile was modified** to properly update source path prefix, compiler path and flags etc.
- The replay engine builds two targets : replay and replay_mpip. The **replay_mpip target is linked with the mpiP library**. This is useful in generating a report of the MPI calls and this verifying the correctness of the trace and replay engine. The replay engine was compiled and run on smaller benchmarks.
- To replay traces which had MPI calls with source/destination, some modifications were needed. The trace file stores the source/destination as offsets, and the actual node ranks need to be calculated using these offsets to correctly address source and destination nodes. The **code modifications were done to calculate node ranks using offset.**
- There was another small caveat here. Some MPI receive calls use MPI_ANY_SOURCE(defined as -2) to specify the source. This value was incorrectly interpreted as the offset and had to be handled appropriately. This is an interesting problem, where some modifications might need to be done in the record engine to **differentiate between a -2 offset and MPI_ANY_SOURCE.**
- Another problem was noticed in the MPI_Wait call, where the request handles are looked up from a table. It is important for the lookup tables to be of the same size both in the record and replay engine, because the offset is stored towards the end of the array. So changes were made in the replay engine so that the **lookup table size matched the value defined in the record engine**.
- Another problem was observed while replaying traces with cross-node compression. A function called member_rank is used to determine whether a particular MPI call belongs to a particular node. This function had no return statement at the end, so the results were unpredictable. This was fixed by simply adding a **returning a proper value at the end of member_rank**.
- With these changes, we had a replay engine which worked with traces from a small set of benchmarks.
- **Timing delta information:** At this point, the **timing information was parsed from the trace** and the timing delta information was read. The **program was made to sleep** (using usleep) for the given amount of time. As mentioned earlier, the code is written to accommodate **two types of formats** for timing delta :- **min/avg/max** and **histogram**. These two are **compiled under separate compiler switches**. The replay engine must be **compiled specifically for either one** of these formats.

Results

The replay engine was modified only so much so that it could run a few basic test cases. Since the main objective was to implement timing delta replay, effort was spent only so that a few running cases were available. This might change later when fixes from the other team are merged.

Thus far, the replay engine timing delta changes have not been merged. So the **results show values which are randomly generated.**

1. MPI call profile capture using mpiP

The figure below shows the mpiP profile report for the sample `irecv-isend-ok2.c` . As can be seen, the sample uses the Isend/Irecv(highlighted) calls. The time consumed by the `usleep()` calls can be seen under the AppTime column. As can be seen, these calls were made from the `lookup_prsd_call()` function, which is responsible for replaying the MPI calls.

```
---------------------------------------------------------------------------
@--- MPI Time (seconds) ---------------------------------------------------
---------------------------------------------------------------------------
Task    AppTime    MPITime     MPI%
   0       29.9       0.01     0.03
   1       29.9       1.91     6.38
   *       59.8       1.92     3.21
---------------------------------------------------------------------------
@--- Callsites: 5 ---------------------------------------------------------
---------------------------------------------------------------------------
 ID Lev File/Address        Line Parent_Funct             MPI_Call
  1   0 prsd_utils.c        1276 lookup_prsd_call         Isend
  2   0 prsd_utils.c        1212 lookup_prsd_call         Irecv
  3   0 prsd_utils.c        1620 lookup_prsd_call         Wait
  4   0 prsd_utils.c         795 lookup_prsd_call         Barrier
  5   0 0x0804ac26             main                       Barrier
---------------------------------------------------------------------------
@--- Aggregate Time (top twenty, descending, milliseconds) ----------------
---------------------------------------------------------------------------
Call                 Site       Time    App%    MPI%     COV
Wait                    3   1.82e+03    3.04   94.69    1.41
Barrier                 4       99.7    0.17    5.20    1.18
Isend                   1       1.65    0.00    0.09    0.69
Irecv                   2      0.435    0.00    0.02    0.09
Barrier                 5      0.031    0.00    0.00    0.23
---------------------------------------------------------------------------
```

Figure 1: mpiP profile report.

2. **Timing delta distribution (min-average-max)**

    3. The figure below shows the distribution of sleep times between consecutive MPI calls. These values were generated using remainders of random values divided by a constant value. A pattern is seen here because the random **values generated aren't truly random.**

Figure 2: Sleep times between MPI calls using average values.

3. Timing delta distribution (histogram)

2. The figure below shows the distribution of sleep times between consecutive MPI calls using the histogram in the traces recorded. These values were generated using remainders of random values divided by a constant value. Again a pattern is seen here because the random **values generated aren't truly random.**

lFigure 3: Sleep times between MPI calls using average values

Pending Items

- integrate record engine timing delta implementation
- integrate the latest replay engine fixes and test in larger benchmarks
  - Update:The above changes could not be done due to non-availability of respective source codes
- implement the number of buckets as a parameter
- detect timing format (min-max-avg or histogram) to automatically handle each type.
  - Update: The above changes have been done.

Problems

Record and replay engine are separately stored and compiled. However, there are dependencies such as compiler switches, header files, pre-defined values that should ideally be shared in common. The code might need to be re-organized to achieve this.

Acknowledgements

I thank Aditya for collaboration on the data structure changes

I also thank Vivek and Apoorva for the initial record engine information and some of the initial changes on which we worked together.