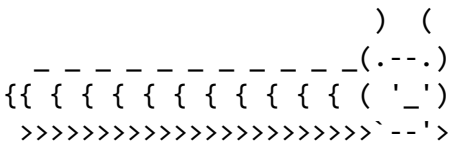
Using ATP, STAT, and LGDB

Oh no! Bugs moved in this example program. Only you are able to a fix the situation.



Introduction

The `dbgMPIapp` program provided in the example directory `dbgMPI/` is a MPI application with basic communication, calculation and process of dependent data. This code is structured in a master--worker scheme. The program takes two arguments, first arguments represents the problem size and the second a flag. Both arguments should be selectable within the range of positive numbers. During the run time, the processes print several statuses, e.g. initialization phase, calculation, and finalization phase, which should be consistent and meaningful.

In the present version multiple programming errors are included. These bugs should be investigated and corrected **without any additional print statements**. Instead, exercise the usage of the debugging tools:

- STAT,
- ATP,
- LGDB.

The goal is to find and correct the bugs and create a bug free program.

Please start debugging in a working directory (on Lustre) with 8 processes and the two arguments: “10000 4” as a first guess. Monitor the job using

```
> squeue -u <user_name>
```

and

```
> tail -f output.out
```

Please **compile on the front end** (login node) by means of the compiler driver wrapper

```
> cc dbgMPIapp.c –o dbgMPIapp
```

and **run / inspect** the program using the batch script `job.slurm`

```
> sbatch job.slurm
```

or from within an interactive session:

```
> ssh gateway<id>
> salloc –N 1 –p workq –t 30
> srun –n 8 ./dbgMPIapp 10000 4 &> output.out &
```

In the following, the mentioned tools are used to point out some shortcomings of the source code.

1. STAT: For When Things Appear To Be Hanging

The Stack Trace Analysis Tool (STAT) was designed to provide a snapshot of a running application by merging the back traces of all processing elements, specifically when it appears to deadlock or hung for some reason. STAT can be activated either as part of a batch job, as via of an interactive session (treated in this example), or directly using the GUI. From an interactive session do

```
> sbatch job.slurm
[1] 20496
> stat-cl 20496
```

This generates a directory stat_results/ containg *.dot files which can be inspected with stat-view. Conversely, the stat-gui command invokes the Stack Trace Analysis Tool GUI that drives stat-cl and allows you to interactively control the sampling of stack traces from your parallel application. The stat-gui is built on stat-view and provides the same call tree manipulation operations. For more information on these operations see stat man page. When finished and if the walltime is not yet exceeded, please stop the program by killing the srun process in the interactive session. Using the STAT to attach to a job is not discussed here but information will be provided.

STAT inspects the application state including the stack traces of all processes, presented in a tree graph. Check this tree with the associated ranks. You will discover a separation of processes. Almost all ranks entered `MPI-Barrier`, except one rank which entered `MPI_Allreduce`. Reminding the correct handling of both (collective) routines, they have to be called by ALL ranks in the communicator (here MPI_COMM_WORLD). To enable the desired functionality, the *worker* routine also has to call `MPI_Allreduce` or even better the routine `calc_crosssum` (called in the `master` routine calling MPI_Allreduce). The `MPI_Barrier` (!not called from the master routine!) is not necessary here, since `MPI_Allreduce` already involves synchronization, thus it can be deactivated in the worker routine.

After correcting this issue, recompile and rerun the program. Now, unfortunately you encounter a segmentation fault, which could be inspected by ATP.

2. ATP (Abnormal Termination Processing): *For When Things Go Wrong Unexpectedly*

ATP is helpful when an application issues a signal indicating it has encountered a problem. This can range from a segmentation violation to a termination signal indicating that the application has run to long. In order to be used, ATP requires the setting of environment variable ATP_ENABLED. Please add "`export ATP_ENABLED=1`" to the batch script job.slurm and rerun the job. In the associated output file we find ATP messages like:

```
…
ATP Stack walkback for Rank 2 starting:
  _start@start.S:113
  __libc_start_main@libc-start.c:242
  main@dbgMPIapp.c:159
  worker@dbgMPIapp.c:98
  foo@dbgMPIapp.c:54
ATP Stack walkback for Rank 2 done
…
```

The ATP Stack walkback points to a certain routine/file and line, here the following line in routine `foo`:

```
res = 2 * *(arr+rank + t0);
```

As also described in the source code comment, it is intend to use the array element at position ‘rank’, added to `t0` and multiplied by 2. Thus, an error exists in calling the array element. Instead the line could be corrected as:

```
res = 2 * (*(arr+rank) + t0);
```

or simply

```
res = 2 * (arr[rank] + t0);
```

For more complex problems, on one hand, `ulimit -c unlimited` can be set (in the batch script), which creates core files, one for each process state. These core files can be examined using debuggers, e.g. lgdb, gdb or ddt.
A re-run of the program still results in a segmentation fault at the same line. Again one can use ATP and inspect the core files as described or directly use the lgdb…

3. LGDB: The Command-Line Parallel Debugger

The command line debugger lgdb can be used to analyse a parallel program at runtime. The following example shows a possible inspection of the mentioned problem in an interactive session:

```
> module load cray-lgdb
> lgdb
dbg all> launch $pset{8} --args="10000 3" dbgMPIApp
dbg all> cont
...
pset{2..5}: Program received signal SIGBUS.
pset{2..5}: In foo at /pathToApp/dbgMPIApp2.c:44
...
dbg all> list
pset{0..1,6..7}: *** The application is running
pset{2..5}: 44      res = 2 * (arr[rank] + t0); #let’s inspect the variables
pset{2..5}: 45          printf("foo calculated %d ", res );
pset{2..5}: 46          // run the loop
pset{2..5}: 47          loop(res);
pset{2..5}: 48          // throw the status
pset{2..5}: 49      //  raise(rank);
pset{2..5}: 50      }
pset{2..5}: 51
pset{2..5}: 52      // should calculate the cross sum over all ranks
pset{2..5}: 53      int calc_crosssum(int rank)
dbg all> print rank
pset{0..1,6..7}: *** The application is running
pset{2..5}: 10000                           # we just used 8 processes
dbg all> print t0
pset{0..1,6..7}: *** The application is running
pset{2}: 2                                  # for rank 2-6 varies …
pset{3}: 3                                  # seems to be the rank
pset{4}: 4
pset{5}: 5
dbg all> list foo         # err in foo ->    # print routine foo …
...
pset{0..7}: 37      void foo(int t0, int rank)      # especially the header
...
dbg all> up               # go back (one level up to the worker routine)
```

```
pset{0..1,6..7}: *** The application is running
pset{2..5}: #1 0x000000000040201c in worker at /pathToApp/dbgMPIApp.c:87
dbg all> list 87
...
pset{0..7}: 87           foo(rank, loopCount);
...
```

First the program is set and started. Then after crashing the source code lines at the actual state are printed. The variable `rank` and `t0` are inspected as well as the function header of `foo` and the call of `foo`. Comparing the variables with the expected values, and the routine call with its definition, one could realize an inconsistency in the order of the call with `rank` and `loopCount`, which are swapped.

After correcting this issue and re-running the application, another deadlock seems to occur. As already described the application can be inspected using STAT (see section 1) or with lgdb. As a result we observe most processes hanging in a loop. This loop represents a calculation and is adjusted very large by `loopCount`. Drastically reducing the "problem size" by selecting a small first argument of the program is equivalent to selecting a "really simple example". Thus we modify the first argument:

```
srun -n 8 ./dbgMPIapp 1 4
```

Now the program should run through without any exception. But please check the program output carefully (especially the line with "bar"), use further additional compiler output or inspection tools e.g. memory debugging with Allinea DDT or Valgrind to find further bugs and resolve them.