Debugging Parallel Algorithms and MPI

<https://www.open-mpi.org/faq/?category=debugging>

So this is more of a tools lecture and deals with much more advanced concepts than we necessarily have time for in detail. But you should know what’s going on.

Debugging MPI has two problems – first, conceptualising how parallel problems fails, and secondly, MPI specific tools/commands.

# Parallel Problems

Conceptually parallel debugging is a matter of thinking through the sequence of operations as they happen, and where messages are passed too and when. There are several common problems that arise: Data arriving too early or too late, tasks starting before they are ready, processes that hang around after they should be finished, and lastly non-determinism (because the same program won’t necessarily run the same way when run in parallel, so problems that exist may not happen all the time). Parallel debugging also introduces the general parallel problems we’ve talked about, of deadlocks and race conditions.

There are several different tools and tech for this that go well beyond what we’d have time for. There are tools for capturing the running state of the program and replaying errors that occur, there are tools for trying to globally manage the running environment of a parallel program (these slow the program down, sometimes dramatically), breakpoints as in serial problems pose weird problems in parallel ones sometimes, that sort of thing. We’re interested today in runtime debugging tools that are built in to MPI (which is not the next section but the one after).

Parallel Debugging Tools (Just read the paragraph)

gdb is a serial debugger, that means it works with single threaded applications, but not parallel ones. In principle you could run some serial debugger (such as gdb) on each process of an MPI application and see what happens, but uh… that problem doesn’t scale well to hundreds or thousands of threads. Doing essentially that is fine for our assignments but there are better tools in MPI (we’ll get to those).

In principle there are parallel debugging tools that let you treat one MPI application (not each process) as a single entity and essentially see what the threads are, that level of abstraction is handy for parallel computing in general. I think the two biggest parallel debuggers are TotalView and DDT. These are extremely sophisticated and while very handy, outside the scope of our class unfortunately.

<https://computing.llnl.gov/tutorials/totalview/> - <https://www.roguewave.com/products-services/totalview>

<https://www.arm.com/products/development-tools/server-and-hpc/forge/ddt>  
  
Neither is free, and, in practice, far more complicated than we need.

Sharcnet actually has a tutorial on how to do DDT debugging - <https://www.youtube.com/watch?v=Y3Pnw3lZtcI> which would be great, if we could use Sharcnet, and had problems that made that worth bothering with.

# MPI Debugging

Note: It is possible, but *extremely* unwise to try and build with MPI debug mode itself. You can basically build a program using MPI debug headers and or source code, but then you must step through the MPI libraries… which is not actually our problem.

MPI also comes with features for debugging your regular programs (so you can compile and run your code in debug mode but have regular MPI functions which do stuff for you).

From: <https://www.open-mpi.org/faq/?category=debugging>

* **mpi\_param\_check:** If set to true (any positive value), and when Open MPI is compiled with parameter checking enabled (the default), the parameters to each MPI function can be passed through a series of correctness checks. Problems such as passing illegal values (e.g., NULL or MPI\_DATATYPE\_NULL or other "bad" values) will be discovered at run time and an MPI exception will be invoked (the default of which is to print a short message and abort the entire MPI job). If set to 0, these checks are disabled, slightly increasing performance.
* **mpi\_show\_handle\_leaks:** If set to true (any positive value), OMPI will display lists of any MPI handles that were not freed before MPI\_FINALIZE (e.g., communicators, datatypes, requests, etc.).
* **mpi\_no\_free\_handles:** If set to true (any positive value), do not actually free MPI objects when their corresponding MPI "free" function is invoked (e.g., do not free communicators when MPI\_COMM\_FREE is invoked). This can be helpful in tracking down applications that accidentally continue to use MPI handles after they have been freed.
* **mpi\_show\_mca\_params:** If set to true (any positive value), show a list of all MCA parameters and their values during MPI\_INIT. This can be quite helpful for reproducibility of MPI applications.
* **mpi\_show\_mca\_params\_file:** If set to a non-empty value, and if the value of **mpi\_show\_mca\_params** is true, then output the list of MCA parameters to the filename value. If this parameter is an empty value, the list is sent to stderr.
* **mpi\_keep\_peer\_hostnames:** If set to a true value (any positive value), send the list of all hostnames involved in the MPI job to every process in the job. This can help the specificity of error messages that Open MPI emits if a problem occurs (i.e., Open MPI can display the name of the peer host that it was trying to communicate with), but it can somewhat slow down the startup of large-scale MPI jobs.
* **mpi\_abort\_delay:** If nonzero, print out an identifying message when MPI\_ABORT is invoked showing the hostname and PID of the process that invoked MPI\_ABORT, and then delay that many seconds before exiting. A negative value means to delay indefinitely. This allows a user to manually come in and attach a debugger when an error occurs. Remember that the default MPI error handler — MPI\_ERRORS\_ABORT — invokes MPI\_ABORT, so this parameter can be useful to discover problems identified by **mpi\_param\_check**.
* **mpi\_abort\_print\_stack:** If nonzero, print out a stack trace (on supported systems) when MPI\_ABORT is invoked.
* **mpi\_ddt\_<foo>\_debug**, where **<foo>** can be one of **pack**, **unpack**, **position**, or **copy**: These are internal debugging features that are not intended for end users (but ompi\_info will report that they exist).

So how do you set these?

Recall that to run MPI programs at all you need

export PATH=/usr/mpich-**3.2**/bin:$PATH

export PATH=/usr/lib64/openmpi/bin:$PATH

So now let’s say we want to run a.out, and we want to set a parameter.

In this case running program a.out with 4 processes

(Don’t include the shell$ part, that’s just whatever your terminal prompt is)

shell$ **mpirun** --mca mpi\_show\_handle\_leaks 1 -np 4 a.out

Or

|  |  |
| --- | --- |
| 1 | shell$ **mpirun** --mca param – np 4 filename |

Slightly more sophisticated – you can set these as environ variables.

shell$ OMPI\_MCA\_mpi\_show\_handle\_leaks=1

shell$ **export** OMPI\_MCA\_mpi\_show\_handle\_leaks

shell$ **mpirun** -np 4 a.out

# Activities for Today

I’ve also attached an MPI quicksort routine I found on the web for you to poke at. It turned out to be too much to make into a lab for us, but it’s a useful more interesting program for something to poke at here for part 3 particularly.

1. With your existing MPI lab or an assignment program, set the **mpi\_show\_mca\_params 1** And then run it. This just tells you which parameters are set and there are a LOT of them most of which are utterly irrelevant to us.   
     
   You need to figure out how (based on what’s above) to get it to set parameters, and enumerate them at this step and see that the ones you actually want to use are enabled. It’s not actually hard.
2. Try **mpi\_param\_check,** **mpi\_show\_handle\_leaks,** **mpi\_no\_free\_handles:**   
     
   You should deliberately break your program to see how these tools help you.

mpi\_param\_check will let you look for invalid data types and that sort of thing. In Lab4 I gave you an MPI program that transmits part of an array, try sending a string or a float when it expects an int and watch what mpi\_param\_check does (note that the most correct but a relatively large PITA way to test this is to have two separate programs, one which sends and one which receives and then the receiver will fail when it gets invalid data).

mpi\_show\_handle\_leaks is will show you when you’ve got opening MPI handles (e.g. communications or data types etc.) that aren’t freed before MPI\_FINALIZE. Create some MPI\_INT’s or the like and after a finalize and see what happens!

Mpi\_no\_free\_handles – this is sort of the reverse of the previous one, it’s for when you may have a program that relies on handles that have been initialised but that you’ve already closed with an MPI\_FINALIZE

1. Try **mpi\_abort\_print\_stack** 1 as well, it’s a bit different,

For this, what you need is some code for MPI\_ABORT – usually you put this in your code as an error check, e.g.

MPI\_Bcast (send\_data, count, type, MASTER, GROUP);

And then

if (send\_data ==1)

MPI\_Abort(MPI\_COMM\_WORLD, 1);

// note that the “1” don’t’ have to be the same but MPI\_ABORT does take an integer error code I think

Either way, you can also just randomly stick an MPI\_ABORT in your code and have it spit out the stack trace of your MPI functions at that time.

Mpiquicksort.c, hopefully I attached this as a file, but if not, feel free to copy/paste.

|  |
| --- |
| /\*  Solution available at:  http://monismith.info/cs599/examples.html  http://monismith.info/cs599/examples/quicksortMPI.c  When you run this mpirun -np X filename make sure that the number of processors is a power of 2 (2,4, 8, 16, 32, 64)  \*/  #include <stdio.h>  #include <stdlib.h>  #include <mpi.h>  #define N 2000000  #define X 1000  // Headers, don't touch these  void quicksort(int \*, int, int);  int partition(int \*, int, int);  int choosePivot(int \*, int, int);  void swap(int \*, int \*);  //Main this is the method you'll need to mess with  int main(int argc, char \*\* argv)  {      // boring stuff we'll need      int rank, size, pivot, partner, recvSize;  double start, end;  MPI\_Status status;  MPI\_Init(&argc, &argv);  MPI\_Comm\_size(MPI\_COMM\_WORLD, &size);  MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank);  // random number seed  srand(123456 + 10000\*rank);  // array creation  int \* newArr;    // equivalent to int \*arr = new int [N]  int \* arr = (int \*) malloc(sizeof(int)\*N/size);    int \* recvBuffer = (int \*) malloc(sizeof(int)\*N/size);    // Creates the array with random numbers  int i, j;  for(i = 0; i < N/size; i++)  arr[i] = rand()%X;  // this is some stuff for seeing how long the whole thing takes and starting us off. It picks the mid point element in the array  if(rank == 0)  {  start = MPI\_Wtime();  pivot = choosePivot(arr, 0, N/size-1);  printf("The pivot is %d\n", pivot);  }  // YOU NEED TO WRITE CODE HERE - YOU NEED TO BROADCAST (MPI\_Bcast (parmeter1, 1, MPI\_INT, parameter2, MPI\_COMM\_WORLD)  MPI\_Bcast(&pivot, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);  //Assume that the number of processes is a power of 2  int storeIdx = 0;  int arrSize = N/size;  for(partner = size/2; partner > 0; partner = partner >> 1)  {  storeIdx = 0;  for(i = 0; i < arrSize; i++)  {  if(arr[i] < pivot)  {  swap(&arr[i], &arr[storeIdx]);  storeIdx++;  }  }  printf("storeIdx = %d in process %d\n", storeIdx, rank);  //0...7 8...15 16/2 = 8  //  //rank / (size/partner)  //0 vs 1  //  //0..3 4..7 8..11 12..15 16/4 = 4  //lower upper lower upper  //0 1 2 3  int flag = 0;  MPI\_Request request, requestSend;  if( (rank / partner) % 2 == 0)  {  printf("rank + partner is %d\n", rank + partner);  int sendVal = arrSize-storeIdx;  recvSize = 0;  MPI\_Isend(&sendVal, 1, MPI\_INT, rank+partner, partner+size, MPI\_COMM\_WORLD, &requestSend);  MPI\_Irecv(&recvSize, 1, MPI\_INT, rank+partner, partner+size, MPI\_COMM\_WORLD, &request);  MPI\_Wait(&request, &status);  if(arrSize-storeIdx > 0)  {  MPI\_Isend(arr+storeIdx, arrSize - storeIdx, MPI\_INT, rank + partner, partner,MPI\_COMM\_WORLD, &requestSend);  }  printf("recvsize is %d in process %d\n", recvSize, rank);  if(recvSize > 0)  {  free((void \*) recvBuffer);  recvBuffer = (int \*) malloc(sizeof(int)\*recvSize);  MPI\_Irecv(recvBuffer, recvSize, MPI\_INT, rank + partner, partner,  MPI\_COMM\_WORLD, &request);  MPI\_Wait(&request, &status);  }  }  else  {  int sendVal = storeIdx;  recvSize = 0;  MPI\_Isend(&sendVal, 1, MPI\_INT, rank-partner, partner+size, MPI\_COMM\_WORLD, &requestSend);  MPI\_Irecv(&recvSize, 1, MPI\_INT, rank-partner, partner+size, MPI\_COMM\_WORLD, &request);  MPI\_Wait(&request, &status);  printf("rank - size is %d\n", rank - partner);  if(storeIdx > 0)  {  MPI\_Isend(arr, storeIdx, MPI\_INT, rank - partner, partner, MPI\_COMM\_WORLD, &requestSend);  }  printf("recvsize is %d in process %d\n", recvSize, rank);  if(recvSize > 0)  {  free((void \*) recvBuffer);  recvBuffer = (int \*) malloc(sizeof(int)\*recvSize);  MPI\_Irecv(recvBuffer, recvSize, MPI\_INT, rank - partner, partner,  MPI\_COMM\_WORLD, &request);  MPI\_Wait(&request, &status);  }  }  MPI\_Barrier(MPI\_COMM\_WORLD);  if(recvSize > 0)  {  //Merge arrays  if((rank / partner) % 2 == 0) //Keep smaller elements  {  newArr = (int \*) malloc(sizeof(int)\*(recvSize+storeIdx));  for(i = 0; i < storeIdx; i++)  newArr[i] = arr[i];  for(j = 0, i = storeIdx; i < recvSize+storeIdx; i++, j++)  newArr[i] = recvBuffer[j];  free((void \*) arr);  arr = newArr;  newArr = NULL;  arrSize = recvSize+storeIdx;  }  else //Keep larger elements  {  newArr = (int \*) malloc(sizeof(int)\*(recvSize+(arrSize-storeIdx)));  for(j = 0, i = storeIdx; i < arrSize; i++, j++)  newArr[j] = arr[i];  for(j = 0, i = arrSize-storeIdx; i < recvSize+(arrSize-storeIdx); i++, j++)  newArr[i] = recvBuffer[j];  free((void \*) arr);  arr = newArr;  newArr = NULL;  arrSize = recvSize+(arrSize-storeIdx);  }  }  else  {  arrSize = 0;  }  if(rank % partner == 0)  {  pivot = choosePivot(arr, 0, arrSize-1);  for(i = 1; i < partner; i++)  MPI\_Send(&pivot, 1, MPI\_INT, rank+i, partner+1, MPI\_COMM\_WORLD);  }  else  {  MPI\_Recv(&pivot, 1, MPI\_INT, partner\*(rank/partner), partner+1, MPI\_COMM\_WORLD, MPI\_STATUS\_IGNORE);  }  }  if(arrSize > 0)  quicksort(arr, 0, arrSize-1);  int \* sizeArr, \* fullArr, \* displacement;  if(rank == 0)  {  sizeArr = (int \*) malloc(sizeof(int)\*size);  fullArr = (int \*) malloc(sizeof(int)\*N);  displacement = (int \*) malloc(sizeof(int)\*size);  }  MPI\_Gather(&arrSize, 1, MPI\_INT, sizeArr, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);  if(rank == 0)  {  i = 0;  displacement[0] = 0;  printf("displacement[%d] = %d\n", i, displacement[i]);  //Perform a scan on sizeArr to determine the displacement of each data location.  for(i = 1; i < size; i++)  {  displacement[i] = sizeArr[i-1] + displacement[i-1];  printf("displacement[%d] = %d\n", i, displacement[i]);  }  }  MPI\_Gatherv(arr, arrSize, MPI\_INT, fullArr, sizeArr, displacement, MPI\_INT, 0, MPI\_COMM\_WORLD);  MPI\_Barrier(MPI\_COMM\_WORLD);  if(arrSize > 0)  free((void \*) arr);  if(rank == 0)  {  // for(i = 0; i < N; i++)  // printf("%d ", fullArr[i]);  end = MPI\_Wtime();  printf("Time required was %lf seconds\n", end-start);  free((void \*) fullArr);  free((void \*) sizeArr);  free((void \*) displacement);  }  MPI\_Finalize();  return 0;  }  // Don't change quicksort  void quicksort(int \* arr, int lo, int hi)  {  if(lo < hi)  {  int p = partition(arr, lo, hi);  quicksort(arr, lo, p - 1);  quicksort(arr, p + 1, hi);  }  }  //Don't change this code  int partition(int \* arr, int lo, int hi)  {  int i;  int pivotIdx = choosePivot(arr, lo, hi);  int pivotVal = arr[pivotIdx];  swap(&arr[pivotIdx], &arr[hi]);  int storeIdx = lo;  for(i = lo; i < hi; i++)  {  if(arr[i] < pivotVal)  {  swap(&arr[i], &arr[storeIdx]);  storeIdx++;  }  }  swap(&arr[storeIdx], &arr[hi]);  return storeIdx;  }  //Don't change this code either  void swap(int \* x, int \* y)  {  int temp = \*x;  \*x = \*y;  \*y = temp;  }  //Don't change this code either  //Select the median of arr[lo], arr[hi], and arr[(lo+hi)/2]  int choosePivot(int \* arr, int lo, int hi)  {  int mid = (lo+hi)/2;  int temp;  if(arr[lo] > arr[hi])  {  temp = lo;  lo = hi;  hi = temp;  }  if(arr[mid] < arr[lo])  {  temp = mid;  mid = lo;  lo = temp;  }  if(arr[hi] < arr[mid])  {  temp = mid;  mid = hi;  hi = temp;  }  return arr[mid];  } |