PARC source code

```
//
//
//    PARC  - a parallel and concurrent data encoding method using MPI and Parallel I/O support
//
//
//


#define _XOPEN_SOURCE 1               /* Required under GLIBC for nftw() */
#define _XOPEN_SOURCE_EXTENDED 1      /* Same */

#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <getopt.h>
#include <ftw.h>        /* gets <sys/types.h> and <sys/stat.h> for us */
#include <limits.h>     /* for PATH_MAX */
#include <unistd.h>     /* for getdtablesize(), getcwd() declarations */
#include <curl/curl.h>
#include <fcntl.h>
#include <string.h>
#include <mpi.h>
#include <curl/curl.h>
#include <netdb.h>
#include <time.h>
#include <sys/times.h>
#include <dirent.h>

extern int h_errno;

#include "parc.h"


char *output_fname = NULL;
char *config_fname = NULL;
char *sourceFile   = NULL;    // example /memfs/36MB001

int  numEntry      = DEFAULT_NUM__ENTRY;
int  numPDM        = 1;
int  numGateway    = 1;
int  allENTRYHANDLED = FALSE;
int  overWriteFlag = NO;
```

```
int  fileSize      = 37748736;
int  numShared     = 8;
int  numNeeded     = 5;
int  destDirFlag   = NO;
int  prefixFlag    = NO;
int  suffixFlag    = NO;
int  helpFlag      = NO;

char *overWrite = (char *) "";
char *destDir   = (char *) "./";
char *srcDir    = (char *) ".";
char *prefix    = (char *) "";
char *suffix    = (char *) "";

struct workloadInfo      workloadList[MAX_NUM_ENTRY];
struct dirent    *fileList[DEFAULT_NUM__ENTRY];
struct fileInfo fileProcessedList[DEFAULT_NUM__ENTRY];

char *sourceFile;

char PARCCmd[256];

clock_t startTimeProcess;  // from times()
clock_t finishTimeProcess; // from times()
double  cpu_time_used;  // usec
struct tms stimeProcess, etimeProcess; //from times()

time_t  startUploadTimeProcess;// from time()
time_t  endUploadTimeProcess;  // from time()
time_t  uploadTimeProcess;     // delta time
struct tm *loctimeProcessStart;
struct tm *loctimeProcessFinish;


int master(int, int);
int producer(int, int);
int explorer(int, int);
int PARCAgent(int, int);
int processInputArgument(int , char **);
int workloadGenerator(int , struct dirent **fileList);
int initData(int ,char *,char *,struct dirent **,int ,int );
int PARCDecoder(char *,int ,int, char **);
int PARCEncoder(char *,int ,int ,int ,int ,char [],char *);

// usage
```

```
// mpirun -np numProcs --hostfile hostlist PARC -p prefix -s suffix -n numFiles -m numShared -
k numNeeded -f -s srcDir -d destDir
//

int main(argc, argv)
   int  argc;
   char *argv[];
{
  int  myRank;
  int  numProcs;
  int  rc  = 0;
  char hostname[256];

  // Initiaize MPI
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numProcs); // number of process used in this job
  MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

  printf("num process = %d\n", numProcs);
  printf("My rank     = %d\n", myRank);

  /* error handler */
  MPI_Comm_set_errhandler(MPI_COMM_WORLD, MPI_ERRORS_RETURN);

  memset(hostname, ' ', sizeof(char)*256);
  gethostname(hostname, sizeof(char)*256);

  printf("hostname = %s\n", hostname);

  startUploadTimeProcess = time(NULL); // currentl local time
  startTimeProcess = times(&stimeProcess);
  loctimeProcessStart = localtime (&startUploadTimeProcess);

  switch (myRank) {
    case MASTER: //rank 0
        rc = processInputArgument(argc , argv);
        if(helpFlag == YES) {
          printf("Please try it later\n");
          return(0);
        }

        printf("Master: myRank=%d srcDir[%s],destDir[%s],M=%d,k=%d\\n",
myRank,srcDir,destDir,numShared,numNeeded);
        rc = initData(myRank, srcDir, destDir, fileList, numShared, numNeeded);
        printf("Master: myRank=%d\n", myRank);
        rc = master(myRank, numProcs);
```

```
        break;
    case PRODUCER: // rank 1
        printf("Poducer: myRank=%d\n", myRank);
        rc = producer(myRank, numProcs);
        break;
    case EXPLORER: // rank 2
        printf("Explorer: myRank=%d\n", myRank);
        rc = explorer(myRank, numProcs);
        break;
    default: // rank 3 and ramining
        printf("PARC agent: myRank=%d, enter PARCAgent \n", myRank);
        rc = PARCAgent(myRank, numProcs);
        break;
  }

  // generate report
  //

  finishTimeProcess = times(&etimeProcess);
  cpu_time_used = ((double) (finishTimeProcess - startTimeProcess)) / CLOCKS_PER_SEC;


  MPI_Finalize();       /* cleanup MPI */

  return(rc);
}


int processInputArgument(int argc , char *argv[]) {
    int opt = 0;
    int rc  = 0;

  int  M  = 8;
  int  K  = 2;
  int  forceFlag = 0;
  char *prefix = "" ;
  char *suffix = "" ;

  int checkARGV = 0;
  int prefixFlag = 0;
  int suffixFlag = 0;
  char pstr[16];
  char sstr[16];
  char fstr[16];
  char str[64];
```

```
/*
   -n  number of data entry
   -w  fixed data size from input data
   -o  redirect output to a file
   -d  dest Directory Path
   -s  source Directory Path
   -m  total encoding shared data chunk will be created  : default is 8
   -k  the number of  shared data chunk required to reconstruct the original data: default is 3
*/

 overWrite   = NO;
 numShared   = 8;
 numNeeded   = 5;
 destDirFlag = NO;

 memset(str,  ' ', 64*sizeof(char));
 memset(pstr, ' ', 16*sizeof(char));
 memset(sstr, ' ', 16*sizeof(char));
 memset(fstr, ' ', 16*sizeof(char));
 memset(PARCCmd, ' ', 256*sizeof(char));



 while ((opt = getopt(argc, argv, "h:n:o:d:m:k:p:s:f")) != EOF) {
   switch(opt) {
     case 'n':
       numEntry = (int ) atoi(optarg);
       printf("\n number of entry =%d\n", numEntry);
       break;
     case 'o': //
       srcDir = optarg;
       printf("\nsourceDir = %s\n", srcDir);
       break;
     case 'd': // destination of encoded share3d file objects
       destDir = optarg;
       destDirFlag = YES;
       printf("\ndestDir = %s\n", destDir);
       break;
     case 'm':
       numShared = atoi(optarg);
       printf("\nNumber of shared chunk created =%d\n", numShared);
       break;
     case 'k':
       numNeeded = atoi(optarg);
       printf("\nNumber of data chunk required to recover the original file =%d\n", numNeeded);
```

```
        break;
      case 'p':
        prefix = optarg;
        printf("\nPrefix=%s\n", prefix);
        prefixFlag = 1;
        sprintf(pstr, " -p %s ", prefix);
        strcat(str, pstr);
        break;
      case 's':
        suffix = optarg;
        printf("\nSuffix=%s\n", suffix);
        suffixFlag = 1;
        sprintf(sstr, " -s %s ", suffix);
        strcat(str, sstr);
        break;
      case 'f':
        forceFlag = 1;
        sprintf(fstr, " -f ");
        strcat(str, fstr);
        break;
      case 'h':
        printf("USAGE\n");
        printf("mpirun -np numProcs --hostfile hostlist PARC -p prefix -s suffix -n numFiles -m
numShared -k numNeeded -f -s srcDir -d destDir \n");
        printf("sPARC -h : how to run this sPARC\n");
        helpFlag = YES;
        break;

      default:
        rc = -1;
        printf("\nunknow argv   =%s\n", optarg);
        break;
    }
  }


  if( (prefixFlag == 1) || (suffixFlag == 1) || (forceFlag == 1)) {
    sprintf(PARCCmd, "PARC -d %s %s -m %d -k %d  ", destDir, str, numShared, numNeeded);
  }
  else {
    sprintf(PARCCmd, "PARC -d %s -m %d -k %d ", destDir, numShared, numNeeded);
  }

  printf("processsInput: PARCCmd=%s\n", PARCCmd);

  return (rc);
```

```
}

/*
int   myRank              my process Rank
char  *srcDir             source directory
char  *destDir            destination directory
struct dirent **fileList   file list scanned from soruce directory
int   M                   number of shared chunks
int   K                   number of code chunk
*/
int initData(int myRank,char *srcDir,char *destDir,struct dirent **fileList,int numShared,int
numNeeded)
{
  int  i;
  int  rc =0;
  DIR  *dirDest;
  DIR  *dirSrc;
  struct dirent *dp;
  struct stat  dirStat;
  struct stat  fileStat;
  int  dirFlag = 0;
  char *file_name;
  char filepath[256];
  char real_file_name[256];
  int  numFile =0;
  int  fileInode;
  char *localSrcDir = srcDir;
  char *localDestDir = destDir;
  int  K;
  int  M;

  K = numNeeded;
  M = numShared = numNeeded;

  printf("\ninitData: srcDir[%s], destDir[%s], M=%d, K=%d\n", srcDir, destDir, M,K);

  // explore source directory and create fileProcessedList
  dirSrc = opendir(srcDir);

  // use readir to get the data set
  i = 0;
  while ((dp=readdir(dirSrc)) != NULL) {
    file_name = dp->d_name;
    fileInode = dp->d_ino;
    printf("file inode  %ld\n", fileInode);
```

```
   printf("filename    %s\n",  file_name);
   memset(filepath, '\0', 256);
   strcat(filepath, srcDir);
   strcat(filepath, file_name);
   printf("real filename=[%s]\n", filepath);
   printf("real srcDir=[%s]\n", srcDir);



  if (strcmp(file_name, ".") == 0)  {
   printf("filename=.\n");
  }
  else
   if (strcmp(file_name, "..") == 0)
   {
    printf("filename=..\n");
   }
   else {
    fileList[i] = dp;
    rc = stat(filepath, &fileStat);

    memset(fileProcessedList[i].filename,'\0', 256 * sizeof(char));
    memset(fileProcessedList[i].destDir, '\0', 256 * sizeof(char));
    memset(fileProcessedList[i].PARCCmd, '\0', 256 * sizeof(char));

    strcpy(fileProcessedList[i].filename, filepath);
    strcpy(fileProcessedList[i].destDir,  destDir);
    strcpy(fileProcessedList[i].PARCCmd,  PARCCmd);

    printf("\n002 initData: srcDir[%s], destDir[%s], M=%d, K=%d\n", srcDir, destDir,
numShared, numNeeded);
    printf("\n002 initData: localSrcDir[%s], localDestDir[%s]\n", localSrcDir, localDestDir);

printf("\nfileProcessedList[i].destDir[%s],destDir[%s]\n",fileProcessedList[i].destDir,destDir);
    printf("strcpy(fileProcessedList[%d].PARCCmd=[%s]\n", i,
fileProcessedList[i].PARCCmd);

    fileProcessedList[i].inode            = fileInode;
    fileProcessedList[i].size             = fileStat.st_size;
    fileProcessedList[i].rank             = myRank;
    fileProcessedList[i].numShared        = numShared;
    fileProcessedList[i].numNeeded        = numNeeded;
    fileProcessedList[i].K                = K;
    fileProcessedList[i].M                = M;
    fileProcessedList[i].totalEncodingSize = 0;
    fileProcessedList[i].encodingBandwidth = 0.0;
```

```
    fileProcessedList[i].startTime         = 0;
    fileProcessedList[i].finishTime        = 0;
    fileProcessedList[i].processTime       = 0.0;
    numFile++;
    i++;
   }
}

closedir(dirSrc);
numEntry = numFile;

printf("workloadGenerator:  number of entry = %d\n", numEntry);

for(i=0; i < numEntry; i++) {
  memset(workloadList[i].objName,'\0', 256 * sizeof(char));
  memset(workloadList[i].destDir,'\0', 256 * sizeof(char));
  memset(workloadList[i].PARCCmd,'\0', 256 * sizeof(char));

  strcpy(workloadList[i].objName,fileProcessedList[i].filename);
  strcpy(workloadList[i].destDir,fileProcessedList[i].destDir);
  strcpy(workloadList[i].PARCCmd, fileProcessedList[i].PARCCmd);

  workloadList[i].objSize         = fileProcessedList[i].size;
  workloadList[i].numShared       = fileProcessedList[i].numShared;
  workloadList[i].numNeeded       = fileProcessedList[i].numNeeded ;
  workloadList[i].K               = fileProcessedList[i].K;
  workloadList[i].M               = fileProcessedList[i].M;
  workloadList[i].workloadID      = i;
  workloadList[i].assignedStatus  = UNASSIGNED;
  workloadList[i].uploadStatus    = UNDERUPLOADED;
  workloadList[i].myRank          = 0;
  workloadList[i].startUploadTime = 0;
  workloadList[i].endUploadTime   = 0;
  workloadList[i].uploadTime      = 0;
  workloadList[i].uploadBandwidth = 0.0;
}

for(i=0; i < numEntry; i++) {
  printf("fileProcessedList[%d].filename = %s\n", i, fileProcessedList[i].filename);
  printf("fileProcessedList[%d].destDir  = %s\n", i, fileProcessedList[i].destDir);
  printf("fileProcessedList[%d].inode    = %ld\n",i, fileProcessedList[i].inode);
  printf("fileProcessedList[%d].size     = %d\n", i, fileProcessedList[i].size);
  printf("fileProcessedList[%d].rank     = %d\n", i, fileProcessedList[i].rank);
  printf("fileProcessedList[%d].K        = %d\n", i, fileProcessedList[i].K);
  printf("fileProcessedList[%d].M        = %d\n", i, fileProcessedList[i].M);
  printf("workloadList[%d].objSize       = %d\n", i, workloadList[i].objSize);
```

```
   printf("workloadList[%d].objName       = %s\n", i, workloadList[i].objName);
  }

  return(rc);
}



int master(int myRank, int numProcs)
{
 int ntasks, rank;
 int exceptRank;
 int workAssigned;
 int workDone;
 int work = -1;
 int workID ;
 struct workloadInfo  workloadData;
 struct workloadInfo  workloadReceived;
 struct workloadInfo  workloadSent;
 int rc = 0;
 MPI_Status   status;
 int error;
 int curEntry = -1;
 int i;
 struct dirent **namelist;

 //      Communication protocol between: Master, Producer, Explorer, and PARCAgent
 //
 //      [1] Master    ----> work  ---> PARCAgent  move item(work) to destination
 //          send INITWROK to all processes
 //          while(1) {
 //            receive any incoming message
 //          switch (status.MPI_SOURCE) {
 //             PRODUCER:
 //                  send  reply
 //             EXPLORER:
 //                  send reply
 //             PDM:
 //           pdmRank = status.MPI_SOURCE
 //           get next work item
 //           send next work item to pdmRank
 //         }
 //
 //      [2] PARCAgent   ----> work
 //
```

```
//          if(tag == WORKTAG){
//             if(work == INITWORK)
//                send a request to get next job assignment from Master
//             else
//                call curl-upload item(work)
//                record upload result (start time, end time, size of data upload)
//                send a request to get next job assignment from Master
//          else if tag == ENDTAG
//                return to the main program and finish this MPI process
//
//        [3] producer
//
//        [4] explorer
//

 // STEP 1: send an initial message to each MPI process except the producer and explorer
processes

 for (rank = 1; rank < numProcs; ++rank) {
    work  = INITWORK;
    workloadSent.workloadID = INITWORK;

    error = MPI_Send(
            &workloadSent,   /* message buffer           */
            1,              /* one data item            */
            MPI_INT,        /* data item is an integer  */
            rank,           /* destination process rank */
            INITTAG,        /* user chosen message tag  */
            MPI_COMM_WORLD); /* always use this          */

    if(error != MPI_SUCCESS)
      printf("MPI_Send: Master Process %d Failed on sedning an initial mesage to  process %d,
rc=%d \n", myRank, rank, error);
    else
      printf("MPI_Send: Master sending an Initial  message to process %d\n", rank);
 }


 /* STEP 2:
  * Receive a result from any procs and dispatch a new work request
  * until all workload has  been assigned and processed
  */

  while (allENTRYHANDLED == FALSE) {

    // assign next workload entry ID to this PDM process
```

```
   curEntry++;

   if(curEntry > numEntry) {
     printf("Master: All %d entry are handled, final work entry %d is reached\n", numEntry,
work);
     allENTRYHANDLED = TRUE;
   }
   else {
     printf("Master: Next dataEntry  %d \n",work);
     error = MPI_Recv(
          &workloadReceived,           /* message buffer              */
          1,           /* one data item               */
          MPI_INT,        /* data item is a double real  */
          MPI_ANY_SOURCE, /* receive from any sender     */
          MPI_ANY_TAG,    /* receive any type of message */
          MPI_COMM_WORLD, /* always use this             */
          &status         /* info about received message */
            );

     if(error != MPI_SUCCESS)
       printf("MPI_Recv: Master Process %d Failed on receiving a mesage, rc=%d \n", myRank,
error);
     else  {
       printf("MPI_Recv: Master Received a message from process %d, TAG=%d\n",
status.MPI_SOURCE, status.MPI_TAG);
       workloadID = workloadReceived.workloadID;
     }

     // check who is sending me message
     switch(status.MPI_SOURCE ) {
       case PRODUCER:
         printf("Should not recevie a message from producer\n");
         break;
       case EXPLORER:
         printf("Should not recevie a message from explorer\n");
         break;
       default: // from one of PDM process
         if(status.MPI_TAG == WORKDONETAG || status.MPI_TAG == WORKTAG) {
           if(status.MPI_TAG == WORKDONETAG) {
             printf("MPI_Recv: Master Received a WORKDONEATAG message from process %d,
on entry %d\n", status.MPI_SOURCE, work);
             // report processed status and save processed result information
             // workloadID is processed successfully
             //
           }
```

```
          else
            if(status.MPI_TAG == WORKTAG)
              printf("MPI_Recv: Master Received a WORKTAG message from process %d, on
entry %d\n", status.MPI_SOURCE, work);

          work = curEntry;
          workloadList[work].uploadStatus    = UPLOADED;
          workloadList[work].endUploadTime   = time(NULL);
          workloadList[work].assignedStatus  = ASSIGNED;
          workloadList[work].uploadStatus    = UNDERUPLOADED;
          workloadList[work].myRank          = status.MPI_SOURCE;
          workloadList[work].startUploadTime = time(NULL);

          printf("Master: Assgin work entry %d to PARCAgent %d\n", work,
status.MPI_SOURCE);

        error = MPI_Send(
                    &workloadList,
                    1,
                    MPI_INT,
                    status.MPI_SOURCE,
                    WORKTAG,
                    MPI_COMM_WORLD
                   );

          if(error != MPI_SUCCESS)
            printf("MPI_Send: Master Process %d Failed on sedning a WORKTAG mesage to
process %d, rc=%d \n", myRank, rank, error);
          else
            printf("MPI_Send: Master sending a WORKTAG message to process %d, dataEntry
%d\n", status.MPI_SOURCE, work);

        }
        else {
          printf("MPI_Recv: Master Received an unexpected message from process %d, tag %d\n",
status.MPI_SOURCE, status.MPI_TAG);
        }
        break;

     } // switch

   }

  } // whilte loop
```

```
  printf("Finalize Master Proc ......\n");

  /*
   * Receive results for outstanding work requests.
   */
  printf("Receive results for outstanding work requests\n");

  int  exitCount = 0;

  for (rank = 3; rank < numProcs; ++rank) {
    error = MPI_Recv(
              &work,
              1,
              MPI_INT,
              MPI_ANY_SOURCE,
              MPI_ANY_TAG,
              MPI_COMM_WORLD,
              &status
             );

     printf("Master: receive PARCAgent %d final message\n", status.MPI_SOURCE);

    if(error != MPI_SUCCESS)
      printf("MPI_Recv: Master Process %d Failed on receiving a final mesage, rc=%d \n",
myRank, error);
    else  {
      printf("MPI_Recv: Master Received a final message from process %d\n",
status.MPI_SOURCE);
      printf("Master:  send PARCAgent %d final message\n", rank);
      error = MPI_Send(0, 0, MPI_INT, rank, ENDTAG, MPI_COMM_WORLD);
      if(error != MPI_SUCCESS)
        printf("MPI_Send: Master Process %d Failed on sedning a ENDTAG mesage to  process
%d, rc=%d \n", myRank, rank, error);
      else
        printf("MPI_Send: Master sending a ENDTAG message to process %d\n",
status.MPI_SOURCE);
      exitCount++;
    }

  }


  printf(" Master received %d closeout messages \n", exitCount);

  //  Statistic data
  /*
```

```
   for(i=0; i < numEntry; i++) {


   }
  */

  printf(" Master exit\n");

  return (rc);

}



int producer(int myRank, int numProcs) {
 int rc = 0;
 MPI_Status   status;
 int error;
 int work;
 struct workloadInfo workLoadReceive;


 error = MPI_Recv(
        &workloadReceive,/* message buffer              */
        1,             /* one data item               */
        MPI_INT,         /* data item is a double real  */
        MPI_ANY_SOURCE,  /* receive from any sender     */
        MPI_ANY_TAG,     /* receive any type of message */
        MPI_COMM_WORLD,  /* always use this             */
           &status);         /* info about received message */

   if(error != MPI_SUCCESS)
     printf("MPI_Recv: Producer Process %d Failed on receiving a mesage from unknown
process, rc=%d\n", myRank, error);
   else
     printf("Producer: Producer %d Received a message from process %d\n", myRank,
status.MPI_SOURCE);


  // Now: doing nothing
  printf("Producer Process %d is finished\n", myRank);
  return (rc);


}
```

```
int explorer(int myRank, int numProcs) {
 int rc = 0;
 MPI_Status   status;
 int error;
 int work;
 struct workloadInfo workLoadReceive;

 error = MPI_Recv(
        &workloadReceive,/* message buffer i            */
        1,             /* one data item               */
        MPI_INT,         /* data item is a double real  */
        MPI_ANY_SOURCE,  /* receive from any sender     */
        MPI_ANY_TAG,     /* receive any type of message */
        MPI_COMM_WORLD,  /* always use this             */
           &status);         /* info about received message */

   if(error != MPI_SUCCESS)
     printf("MPI_Recv: EXPLORER Process %d Failed on receiving a mesage from unknown
process, rc=%d \n", myRank, error);
   else
     printf("Explorer: EXPLORER Process %d Received a message from process %d\n", myRank,
status.MPI_SOURCE);

  // Now: doing nothing
  printf("Exploer Process %d is finished\n", myRank);
  return (rc);

}


/*
  PARCAgent request workload from the Master and waiting to receive workload assignment
from the master

*/

int PARCAgent(int myRank, int numProcs) {
  int rc = 0;
  MPI_Status   status;
  int error;
  int work;
  long objSize;
  int  delay;
  clock_t start, end;
  double cpu_time_used;
```

```
  char cmd[1000];
  double  uploadSpeed;
  double  uploadTime;
  struct workloadInfo workLoadReceive;
  struct workloadInfo workLoadSend;
  static destDirCreated = NO;
  int    workloadID;
  char    *destDirLocal;
  struct stat dirStat;

  // STEP 1:
  // receive an initial message from the MASTER
  error = MPI_Recv(
            &workloadReceive,
            1,
            MPI_INT,
            0,
            MPI_ANY_TAG,
          MPI_COMM_WORLD,
            &status);

  /*
   * Check the tag of the received message.
   */
  if(error != MPI_SUCCESS)
    printf("MPI_Recv: PARCAgent Process %d Failed on receiving a mesage from unknown
process, rc=%d \n", myRank, error);
  else {
    printf("PARCAgent: Process %d Received a message from process %d\n", myRank,
status.MPI_SOURCE);

    // STEP 2: replay message
    if(status.MPI_TAG == ENDTAG) {
       // ending this PARCAgent process and exit
       printf("pdaAgent %d is ending and return to main stream  \n", myRank);
       return 0;
    }

    if(status.MPI_TAG == INITTAG) {
      // PARCAgent receive the first message from Master
      // prepare for routine processing

      printf("PARCAgent %d receive the first message from Master \n", myRank);
      printf("PARCAgent %d send a WORKTAG message backto Master \n", myRank);

      work = -1;
```

```
      workloadSend.workloadID = -1;
      error = MPI_Send(&workloadSend, 1, MPI_INT, MASTER, WORKTAG,
MPI_COMM_WORLD);
      if(error != MPI_SUCCESS)
        printf("MPI_Send: PDM Process %d Failed on sedning a WORKTAG mesage to
MASTER process, rc=%d \n", myRank, error);
      else
        printf("MPI_Send: PDM Process %d send Master sending a WORKTAG message to
MASTER process\n", myRank);

    }
  }



  // STEP 3:  Regular processing
  for (;;) {
    error = MPI_Recv(
            &workloadReceive,
            1,
            MPI_INT,
            0,
            MPI_ANY_TAG,
          MPI_COMM_WORLD, &status);
    /*
     * Check the tag of the received message.
     */
    if(error != MPI_SUCCESS) {
      printf("MPI_Recv: PARCAgent Process %d Failed on receiving a mesage from unknown
process, rc=%d \n", myRank, error);
    else
      printf("PARCAgent: PARCAgent Process %d Received a message from process %d\n",
myRank, status.MPI_SOURCE);

    }

    if (status.MPI_TAG == ENDTAG) {
      printf("PARCAgent Process %d reveived an ENDTAG message from process %d\n",
myRank, status.MPI_SOURCE);
      printf("PARCAgent Process %d is finished and returned to main", myRank);
      return 0;
    }

    if (status.MPI_TAG == WORKTAG) {
      printf("PDM Process %d reveived an WORKTAG message from process %d on entery %d
\n", myRank, status.MPI_SOURCE, work);
```

```
      // workload information
      workloadID   = workloadReceived.workloadID;
      filename     = (char *) workloadList[i].objName;
      destDirLocal = (char *) workloadList[i].destDir;
      prefix       = (char *) workloadList[i].prefix;
      suffix       = (char *) workloadList[i].suffix;
      objectSize   = workloadList[i].objSize;
      numShared    = workloadList[i].numShared;
      numNeeded    = workloadList[i].numNeeded;

      //
      // call PARC
      //
      // check the destDir creation status:
      // only need to created once
      //
      if(static destDirCreated == NO) {
        rc = stat(destDiriLocal, &dirStat);
        if(S_ISDIR(dirStat.st_mode)) {
          printf("destination Directory is exist destDir[%s]\n", destDirLocal);
          dirFlag = 1;
          destDirCreated = YES;
        }
        else {
          // destination Directory is not exist, create one
          printf("destination Directory is not exist, create one destDir[%s]\n", destDirLocal);
          rc  = mkdir(destDirLocal, S_IRWXU | S_IRWXG | S_IROTH | S_IXOTH);
          if(rc != 0) {
            printf("ERROR: parFEC initData - Cannot create Destination directory [%s]\n",
destDirLocal);
          }
          else
          destDirCreated = YES;
        }
      }

      start = times(&stime);
      rc =
PARCEncoder(destDirLocal,numShared,numNeeded,forceFlag,prefixFlag,suffixFlag,char
str[],filename)
      end = times(&etime);
      cpu_time_used2 = ((double) (end - start)) / CLOCKS_PER_SEC;

      workloadSent.workloadID      = i;
      workloadSent.assignedStatus  = UNASSIGNED;
      workloadSent.uploadStatus    = UNDERUPLOADED;
```

```
      workloadSent.myRank          = 0;
      workloadSent.startUploadTime = 0;
      workloadSent.endUploadTime   = 0;
      workloadSent.uploadTime      = 0;
      workloadSent.uploadBandwidth = 0.0;


      error = MPI_Send(&workloadSent, 1, MPI_INT, MASTER, WORKDONETAG,
MPI_COMM_WORLD);
      if(error != MPI_SUCCESS)
        printf("MPI_Send: PDM Process %d Failed on sedning a WORKDONETAG mesage to
MASTER process, rc=%d \n", myRank, error);
      else
        printf("MPI_Send: PDM Process %d Master sending a WORKDONETAG message to
MASTER process, current processed entry %d\n", myRank, work);

    }

  } //for loop


  return 0;

}
```

```
/*---------------------------------------------------------------------------
usage: PARC [-h] [-d D] [-p P] [-s S] [-m M] [-k K] [-f] [-v] [-q] [-V] INF

Encode a file into a set of share files, a subset of which can later be used
to recover the original file.

positional arguments:
  INF                   file to encode or "-" for stdin

optional arguments:
  -h, --help            show this help message and exit
  -d D, --output-dir D  directory in which share file names will be created
                        (default ".")
  -p P, --prefix P      prefix for share file names; If omitted, the name of
                        the input file will be used.
  -s S, --suffix S      suffix for share file names (default ".fec")
  -m M, --totalshares M
                        the total number of share files created (default 8)
```

```
  -k K, --requiredshares K
                    the number of share files required to reconstruct
                    (default 3)
  -f, --force          overwrite any file which already in place an output
                    file (share file)
  -v, --verbose        print out messages about progress
  -q, --quiet          quiet progress indications and warnings about silly
                    choices of K and M
  -V, --version        print out version number and exit


    PARC -d destDir -p prefix -s suffix -m M -k K -f -i inputfile


*/



int PARCEncoder(char *destDir,int M,int K,int forceFlag,int prefixFlag,int suffixFlag,char
str[],char *input_filename)
{

  char cmdstr[256];


  printf("str  = %s\n", str);

  if( (prefixFlag == 1) || (suffixFlag == 1) || (forceFlag == 1)) {
    sprintf(cmdstr, "PARC -d %s %s -m %d -k %d  %s ", destDir, str, M,K,input_filename);
  }
  else {
    sprintf(cmdstr, "PARC -d %s -m %d -k %d %s ", destDir, M,K,input_filename);
  }


  printf("\ncmdstr=[%s]\n", cmdstr);

  //system(cmdstr);
  return (0);

}
int PARCDecoder(char *destfile,int force,int numSharedFile, char *sharedFile[])
{
  int rc = 1;

  // Future work: implement decoder function here
```

```
  return rc;
}
```