

The background of the slide is a scenic photograph of a mountain range. In the foreground, there is a dense forest of evergreen trees. Behind the forest, a series of mountain peaks are visible, some with patches of snow or light-colored rock. The sky is a pale, hazy blue. The overall tone of the image is soft and natural.

Bag of Tasks Parallelism

Timothy H. Kaiser, Ph.D.

The background of the slide is a scenic photograph of a mountain range. In the foreground, there is a dense forest of green coniferous trees. Behind the trees, a series of rugged, rocky mountain peaks are visible, some with patches of snow or light-colored rock. The sky is a pale blue with soft, white clouds. The overall tone of the image is serene and natural.

Slides:

<https://github.com/timkphd/slides>

Source

```
hexagon:~ tkaiser$ mkdir tut
hexagon:~ tkaiser$ cd tut
hexagon:tut tkaiser$ git clone https://github.com/timkphd/examples.git
Cloning into 'examples'...
remote: Enumerating objects: 83, done.
remote: Counting objects: 100% (83/83), done.
remote: Compressing objects: 100% (63/63), done.
remote: Total 1236 (delta 39), reused 45 (delta 19), pack-reused 1153
Receiving objects: 100% (1236/1236), 2.47 MiB | 3.64 MiB/s, done.
Resolving deltas: 100% (390/390), done.
hexagon:tut tkaiser$ mv examples/r .
hexagon:tut tkaiser$ rm -rf examples
hexagon:tut tkaiser$ cd r

hexagon:r tkaiser$ ./getdat.py
downloading file
True
split -l 158563 start start
hexagon:r tkaiser$ tar -xzf laser.tgz
hexagon:r tkaiser$

hexagon:r tkaiser$ Rscript doinstall.R
. . .
. . .
hexagon:r tkaiser$ R CMD SHLIB mapit.c
```


What if you have?

- A bunch of tasks to do
- They are all independent
- Similar, maybe just different input files
- Often called bag of task parallelism or embarrassingly parallel

bot.R

- Starts MPI
- ~~Splits the processors into two groups/communicators 0-(N-2) and (N-1)~~
- Processor (N-1) waits for “ready” from other processors, then sends work
- Rest of processors loop
 - send requests for work
 - do work (in this case generate plots)
 - send results

Bag of Tasks

This is a bag-of-tasks program. We define a manager task
that distributes work to workers. Actually, the workers
request input data. The manager sits in a loop calling
Iprobe waiting for requests for work.

In this case the manager reads input. The input is a list
of file names. It will send a entry from the list as
requested. When the worker is done processing it will
request a new file name from the manager. This continues
until the manager runs out of files to process. The
manager subroutine is just "manager"

The worker subroutine is "worker". It receives file names
form the manager.

#

The files in this case are outputs from an optics program
tracking a laser beam as it propagates through the atmosphere.
The workers read in the data and then create an image of the
data by calling the routine doplot This should work
with arbitrary 2d files except the size in mkview.plotit is
currently hard coded to 64 x 64.

~~# We use the call to "Split" to create a separate communicator
for the workers. This is not important in this example but
could be if you wanted multiple workers to work together.~~

- Do initialization

```
MPI.Wtime<-function(){
  return(0.0)
}
if (!is.loaded("mpi_initialize")) {
  library("Rmpi")
}
mpi_comm_world<-0
myid <- mpi.comm.rank(comm=mpi_comm_world)
numprocs <- mpi.comm.size(comm=mpi_comm_world)
myname <- mpi.get.processor.name()
paste("I R am",myid,"of",numprocs,"on",myname)
# num_used is the # of processors that are part of the new communicator #
# for this case hardwire to not include 1 processor #
  num_used<-numprocs-1
  mannum=0;
  if(myid == mannum){
    group<-0
  }else{
    group<-1
  }
}
```


- Create a communicator that contains everyone but the one process

```
# Rmpi does not support this....  
# Split will create a set of communicators. All of the  
# tasks with the same value of group will be in the same  
# communicator. In this case we get two sets one for the  
# manager and one for the workers. The manager's version  
# of the communicator is not used.  
#  
# DEFINED_COMM=mpi_comm_world.Split(group,myid)  
#  
# new_id=DEFINED_COMM.Get_rank()  
# worker_size=DEFINED_COMM.Get_size()  
# print("old id = %d    new id = %d    worker size = %d" %(myid,new_id,worker_size))  
    DEFINED_COMM<-(-1)
```


- Manager is not part of “worker group” so she manages

```
if(group == 0){
  todo=1000
# if not part of the new group do management. #
  manager(num_used,todo)
  print("manager finished")
  mpi.barrier(mpi_comm_world)
  bonk<-mpi.finalize()
}else{
```


- Worker does this...

```
}else{  
# part of the new group do work. #  
mannum=0;  
ts<-MPI.Wtime()  
idid=worker(DEFINED_COMM,mannum)  
te<-MPI.Wtime()  
print(paste("worker",myid,"finished",idid,  
            "tasks in",te-ts,"seconds"))  
mpi.barrier(mpi_comm_world)  
bonk<-mpi.finalize()  
}
```


Manager function initialization

```
manager<-function(num_used,TODO){  
# our "data"  
# Our worker is expecting a single word  
mydat<-read.csv("infile",header=F)  
todo<-nrow(mydat)  
mydat<-as.vector(mydat$V1)  
# counters  
igot<-0  
isent<-0
```

Gets input from stdin

Manager function main loop

```
while(isent < todo){  
# wait for a request for work #  
  mystat<-as.integer(6789)  
  flag<-mpi.iprobe(mpi.any.source(), tag=1234, comm = mpi_comm_world, status = mystat)  
  if(flag){  
# where is it coming from #  
    gotfrom<-mpi.get.sourcetag(status = mystat)  
    sendto<-gotfrom  
    x=as.integer(-1)  
    mystat<-as.integer(6789)  
    x<-mpi.recv(x, 1, gotfrom, tag=1234, comm=mpi_comm_world, status=mystat)  
    print(paste("worker",gotfrom,"sent",x))  
    if(x > -1){  
      igot<-igot+1  
      print(paste("manager got",igot))  
    }  
    if(isent < TODO){  
# send real data #  
      send_msg=mydat[isent+1]  
      mpi.send(send_msg, 3, sendto, tag=2345, comm=mpi_comm_world)  
      isent<-isent+1  
    }  
  }  
}
```

- Waits for ready
- Sends work

Manager finish up

- Tells everyone to quit when work is finished

```
# tell everyone to quit #
    print("tell everyone to quit")
for (i in 1:(numprocs-1)){
    send_msg="stop"
    print(paste("telling",i,send_msg))
    mpi.send(send_msg, 3, i, tag=2345, comm=mpi_comm_world)
}
return( TRUE)
}
```


Worker's loop

```
worker<-function(THComm_WORLD,managerid){
  x<-as.integer(0)
  comm<-mpi_comm_world
  ic<-as.integer(0)
  while(TRUE){
# send message says I am ready for data #
    send_msg<-x
    mpi.send(send_msg, 1, managerid, tag=1234, comm=mpi_comm_world)
# get a message from the manager #
    buffer=""
    mystat<-as.integer(9876)
    buffer<-mpi.recv(buffer, 3, managerid, tag=2345,
                     comm=mpi_comm_world, status=mystat)
    print(paste(myid,"got",buffer))
    if (grepl("stop",buffer)){return(ic)}
    ic<-ic+1
    ##### Do something here
    doplot(buffer)
    #Sys.sleep(4)
  }
}
```

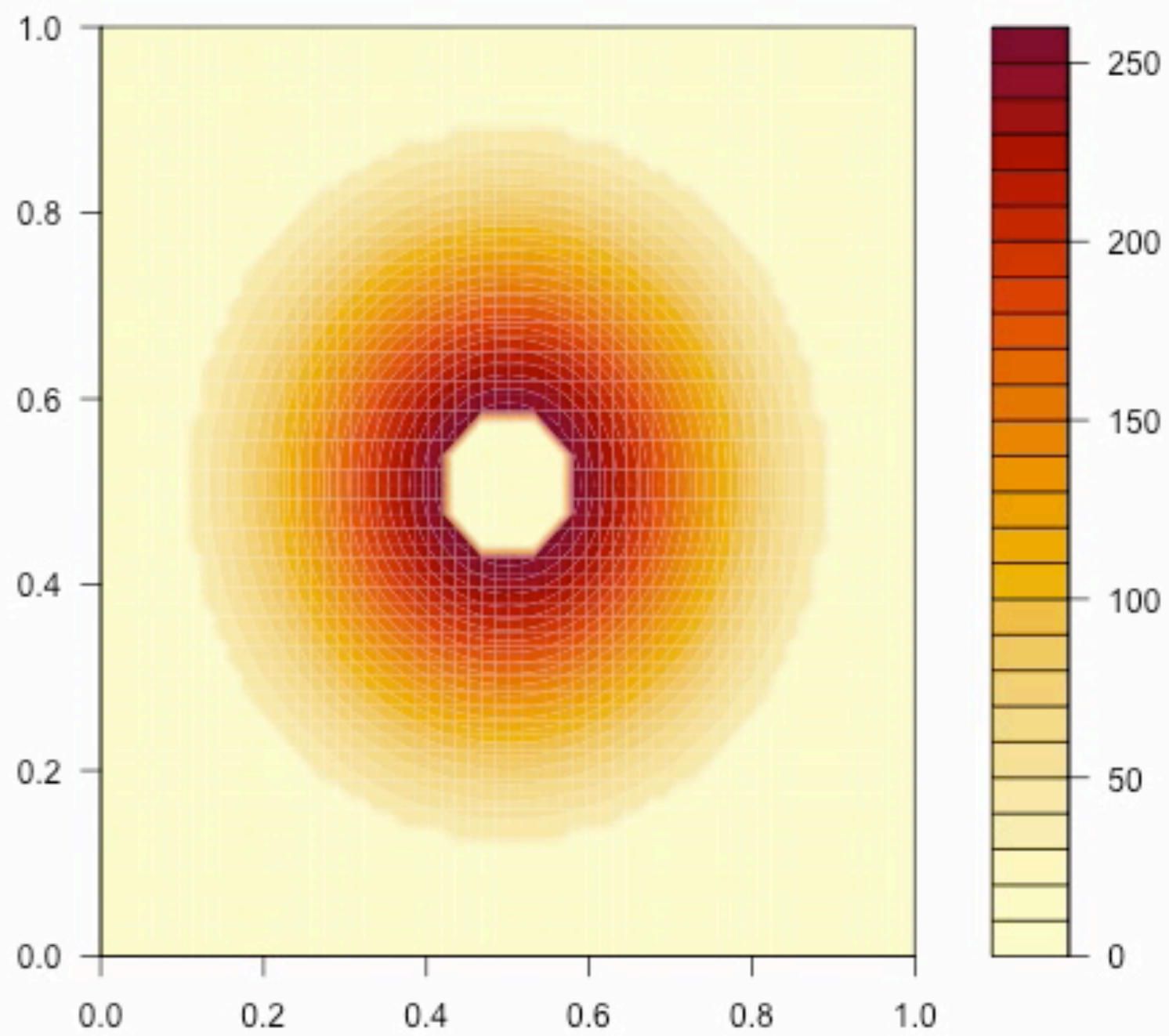
- Workers tell manager they are ready for (more) work

- Get work

- Do work

Worker's "work" function

```
doplot<-function(infile) {  
  library(gplots)  
  outfile<-paste(infile,".png",sep="")  
  bonk<-readLines(infile)  
  bonk<-bonk[-1]  
  #im<-read.delim(infile,sep="")  
  im<-read.delim(text=bonk,sep="")  
  mygrid<-data.matrix(im)  
  png(filename=outfile)  
  filled.contour(mygrid)  
  dev.off()  
}
```

Timings for 4 and 8 tasks

srun -n 4 ./bot.R

```
worker 1 finished 66 tasks in 8.75 seconds  
worker 2 finished 67 tasks in 8.78 seconds  
worker 3 finished 68 tasks in 8.83 seconds
```

srun -n 8 ./bot.R

```
worker 1 finished 29 tasks in 4.11 seconds  
worker 2 finished 28 tasks in 4.13 seconds  
worker 3 finished 29 tasks in 4.04 seconds  
worker 4 finished 29 tasks in 4.13 seconds  
worker 5 finished 28 tasks in 4.04 seconds  
worker 6 finished 29 tasks in 4.13 seconds  
worker 7 finished 29 tasks in 4.12 seconds
```