

# Lecture 21: Manager-worker algorithm

1. Introduction
2. Elements of the manager-worker algorithm
3. Example

# Introduction

- Problems that can easily be parallelized: computational workload can be divided into  $n$  equal parts with little communication.
- More difficult to parallelize: Workload consists of many individual steps that differ considerably in size (i.e. execution time)
- Examples:
  - Finding roots of a family of functions that depend on one or several parameters
  - Integration of oscillating functions where no constant step size can be chosen
  - Structure calculations of isotopic chains of different lengths
  - Computation of Mandelbrot sets
  - SETI (Search for Extra-Terrestrial Intelligence)
- In such cases, the manager-worker algorithm is ideal. One manager distributes work to the workers. These complete the work, send the result back to the manager, and receive new workloads. There's no communication between the workers.

# Example: Matrix-vector product

- Manager process hands out tasks to worker processes
- Workers are assigned new tasks after they complete previous task
- Matrix multiplication example
  - manager process does no work
  - task is to multiply a vector times a matrix
  - initially each process receives the vector and one row of the matrix
  - processes receive additional rows to compute after they send back answer for current row
- Structure of matrix multiplication example
  1. Manager gives work to every worker
  2. Manager receives result and replies by sending more work, or a 'end-of-work' message
  3. Worker receive work or end-of-work message. The former is performed, and the result is send back to manager. The latter leads to an MPI\_Finalize by the worker

```
program main
```

```
include 'mpif.h'
```

```
integer, parameter:: MAX_ROWS = 1000, MAX_COLS = 1000, MAX_PROCS = 32
```

```
real(8):: a(MAX_ROWS,MAX_COLS), b(MAX_COLS), c(MAX_COLS)
```

```
real(8):: buffer(MAX_COLS), ans
```

```
integer:: procs(MAX_COLS), proc_totals(MAX_PROCS)
```

```
integer:: myid, master, numprocs, ierr, status(MPI_STATUS_SIZE)
```

```
integer:: i, j, numsent, numrcvd, sender, job(MAX_ROWS)
```

```
integer:: rowtype, anstype, donetype
```

```
call MPI_INIT( ierr )
```

```
call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
```

```
call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
```

```
if (numprocs .lt. 2) then
```

```
    print *, "Must have at least 2 processes!"
```

```
    call MPI_ABORT( MPI_COMM_WORLD, 1 )
```

```
    stop
```

```
else if (numprocs .gt. MAX_PROCS) then
```

```
    print *, "Must have 32 processes or less."
```

```
    call MPI_ABORT( MPI_COMM_WORLD, 1 )
```

```
    stop
```

```
endif
```

```
print *, "Process ", myid, " of ", numprocs, " is alive"
```

```
! A few useful definitions for the TAG used in point-to-point communications.
```

```
rowtype = 1
```

```
anstype = 2
```

```
donetype = 3
```

```
master = 0
```

```
! Actual size of matrix
```

```
rows = 100
```

```
cols = 100
```

```

if(myid. eq. master) then ! master initializes and then dispatches
do i = 1,cols !initialize matrix A and vector B
  b(i) = 1
  do j = 1,rows
    a(i,j) = i
  enddo
enddo
numsent = 0
numrcvd = 0
call MPI_BCAST(b, cols, MPI_DOUBLE_PRECISION, master, MPI_COMM_WORLD, ierr) ! Send vector to other processes
do i = 1,numprocs-1
  do j = 1,cols
    buffer(j) = a(i,j)
  enddo
  call MPI_SEND(buffer, cols, MPI_DOUBLE_PRECISION, i, rowtype, MPI_COMM_WORLD, ierr) ! Send row of matrix to each process
  job(i) = i
  numsent = numsent+1
enddo
do i = 1,rows
  call MPI_RECV(ans,1, MPI_DOUBLE_PRECISION, MPI_ANY_SOURCE, anstype, MPI_COMM_WORLD, status, ierr)
  sender = status(MPI_SOURCE)
  c(job(sender)) = ans
  procs(job(sender))= sender
  proc_totals(sender+1) = proc_totals(sender+1) +1
  if (numsent .lt. rows) then ! there's still some work to do
    do j = 1,cols
      buffer(j) = a(numsent+1,j)
    enddo
    call MPI_SEND(buffer, cols, MPI_DOUBLE_PRECISION, sender, rowtype, MPI_COMM_WORLD, ierr)
    job(sender) = numsent+1
    numsent = numsent+1
  else ! No work left. send the 'done' message.
    call MPI_SEND(1, 1, MPI_INTEGER, sender, donetype, MPI_COMM_WORLD, ierr)
  endif
enddo
! print out the answer
do i = 1,cols
  write(6,809) i,c(i),procs(i)
enddo
do i=1,numprocs
  write(6,810) i-1,proc_totals(i)
enddo
809 format('c(',i3,') =' ,f8.2,' computed by proc #',i3)
810 format('Total answers computed by processor #',i2,' were ',i3)

```

```

else !worker
! compute nodes receive b, then compute dot products until done message
  call MPI_BCAST(b, cols, MPI_DOUBLE_PRECISION, master, MPI_COMM_WORLD, ierr)
90  call MPI_RECV(buffer, cols, MPI_DOUBLE_PRECISION, master, MPI_ANY_TAG, &
      MPI_COMM_WORLD, status, ierr)
  if (status(MPI_TAG) .eq. donetype) then ! no more work to do
    go to 200
  else ! multiply matrix row with vector
    ans = 0.0
    do i = 1,cols
      ans = ans+buffer(i)*b(i)
    enddo
    call MPI_SEND(ans, 1, MPI_DOUBLE_PRECISION, master, anstype, MPI_COMM_WORLD, ierr)
    go to 90
  endif
endif

200 call MPI_FINALIZE(ierr)

end program main

```

# Notes on manager-worker algorithms

- Though the computational task is simple, the program is rather complex. In particular, one has to make sure that program ends properly.
- This program is automatically load-balancing: fast workers do more work than slow workers, provided that the amount of calculations to be performed is much larger than the number of workers.
- The example is somewhat artificial for a machine with identical processors but natural for a cluster of different computers.

# Summary

- Manager/worker algorithm useful
  - Computational workload cannot be divided into pieces of equal size
  - CPUs in parallel cluster vary in performance
- Basic algorithm is more complex
  - Based on point-to-point communications
  - TAG variable is used to differentiate between work and end-of-work instructions sent from manager
  - Care has to be taken to match number of Send and Receive