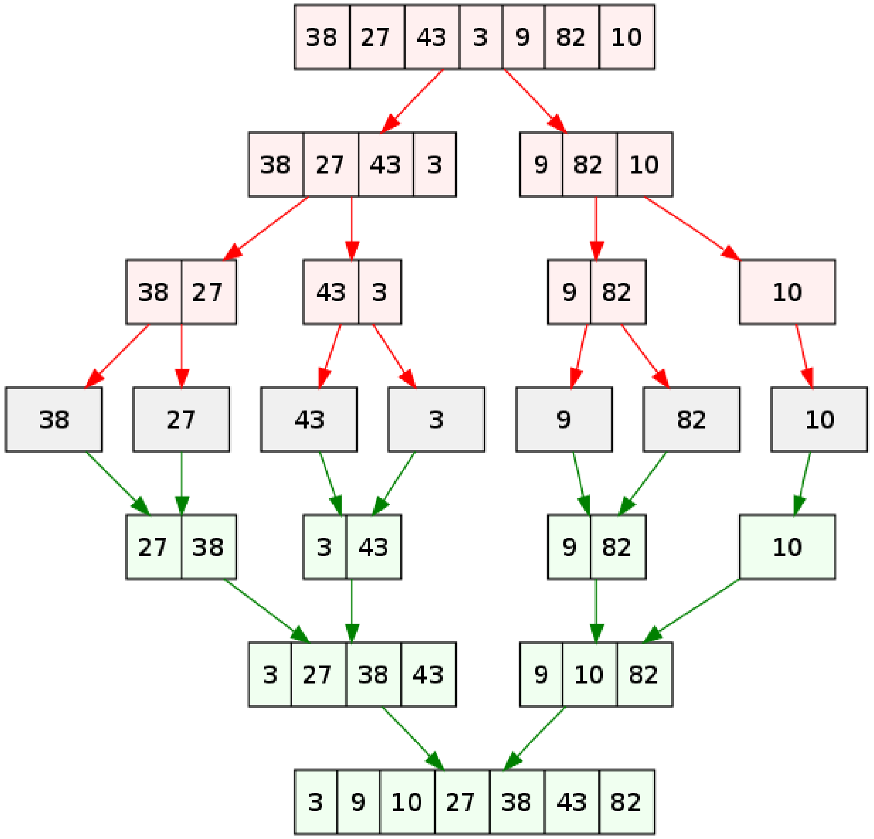
Practical five: Mergesort

Wrap up

Mergesort

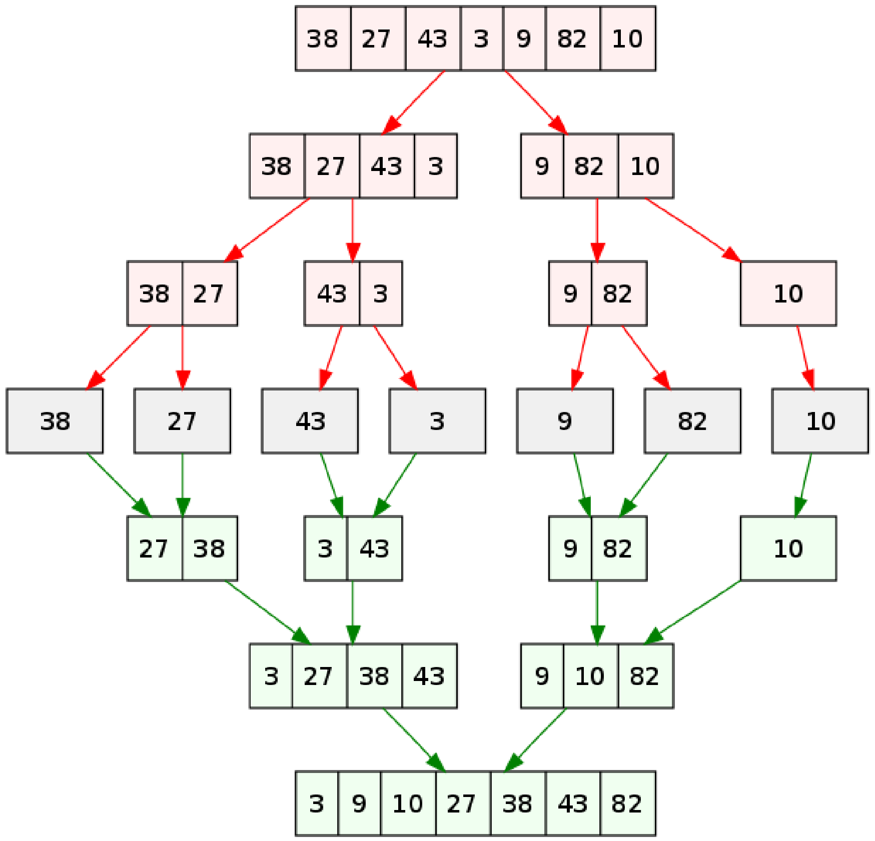
- Starting from some randomly generated, unsorted data.



- Repeatedly divide the data (problem) up until it is trivial to solve
- Then merge the small answers together to form the overall sorted list of numbers

- Maps very well to D&C pattern

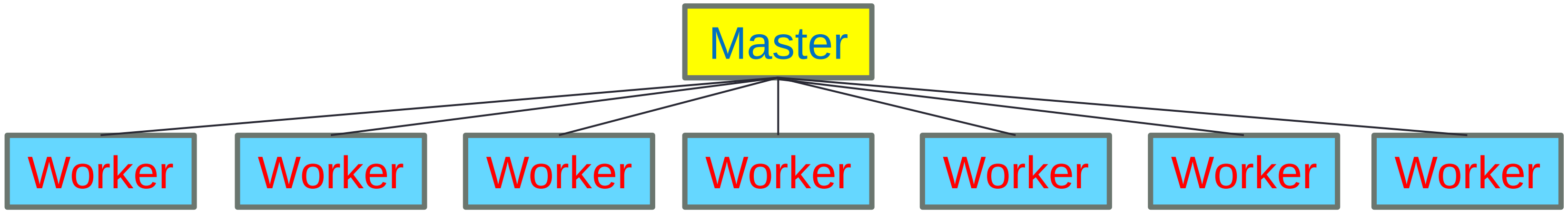
Parallel mergesort



- Each division is a task, working down to some serial threshold (in the image this is 1, but in reality you probably want it to be higher than this.)
- Remember from the lecture, only create one task for the first half of the data and use the existing task for the second half

Skeleton code is provided, your task is to hook it all up with MPI!

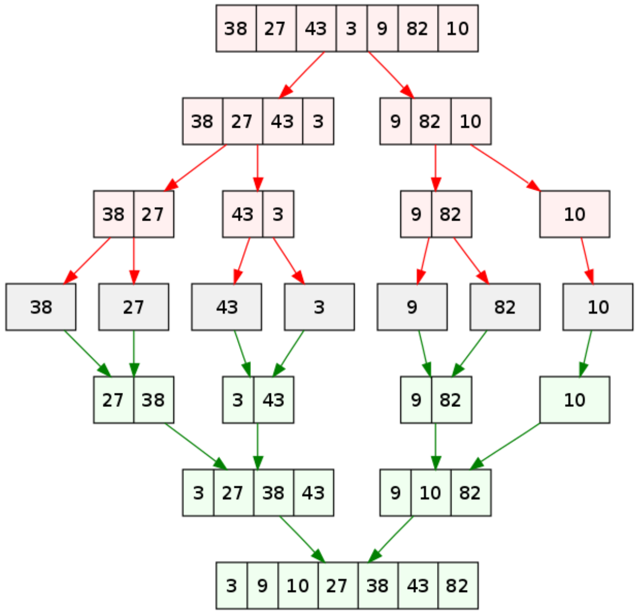
How to do the task generation?



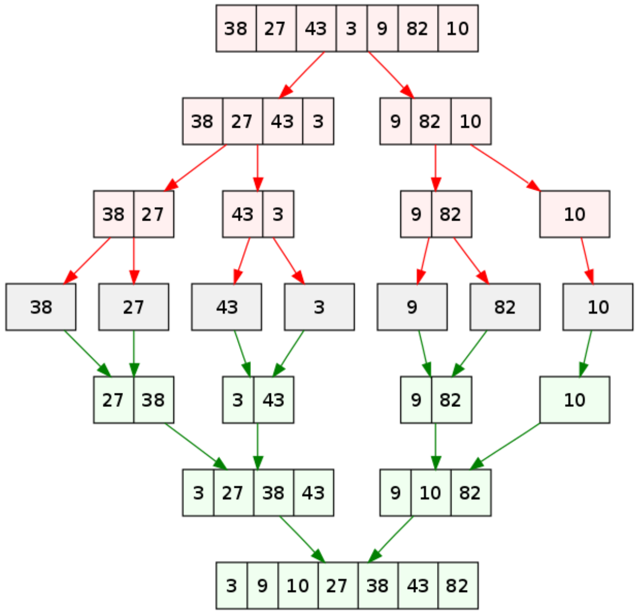
- Provide you with a process pool which implements the master worker pattern
  - The master keeps track of which worker UEs are currently busy
  - Workers sit there and wait for a command from the master to start
  - When a task requests a new worker from the master, the master sends back the rank of this new worker. The new worker is provided with the rank of its parent when it is started and from this the two UEs can communicate
    - i.e. the parent can tell the new worker what data it needs to process

Function	Description
int processPoolInit()	Initialises the process pool (1=worker, 2=master)
void processPoolFinalise()	Finalises and process pool (called from all)
int masterPoll()	Master polls to determine whether to continue or not
int workerSleep()	Worker waits for new task (1=new task, 0=stop)
int startWorkerProcess()	Starts a new worker task and returns the rank of this
int getCommandData()	Retrieves the rank of the task created this one
void shutdownPool()	Called by anyone to shut down the pool

```
static int startMergeSort(double * data, int length) {
  int workerPid = startWorkerProcess();
  /*
   * This initialises the first worker - here the master will need to send the overall size of the data
   * and the data itself to the worker that it has just created
   */
  return workerPid;
}

static void workerCode(int serial_threshold) {
  int workerStatus = 1;
  int parentId, dataLength;
  double * data;
  while (workerStatus) {
    parentId = getCommandData();
    /*
     * Here you will need to receive the data and amount of data to sort from the parent. You
     * will need to allocate the data variable to hold this.
     */
    sort(data, dataLength, serial_threshold);
    if (parentId==0) shutdownPool();
    /*
     * Here you will need to send the sorted data back to the parent
     */
    workerStatus=workerSleep();
  }
}

static void sort(double * data, int length, int serial_threshold) {
  if (length < serial_threshold) {
    qsort(data, length, sizeof(double), qsort_compare_function);
  } else {
    int workerPid = startWorkerProcess();
    /*
     * Need to figure out the pivot, split the data, send half to the newly created worker.
     * Then this process should call sort with the other half of the data. Once that has completed
     * need to get the sorted data back from the worker and merge the two together (using merge function)
     */
  }
}
```



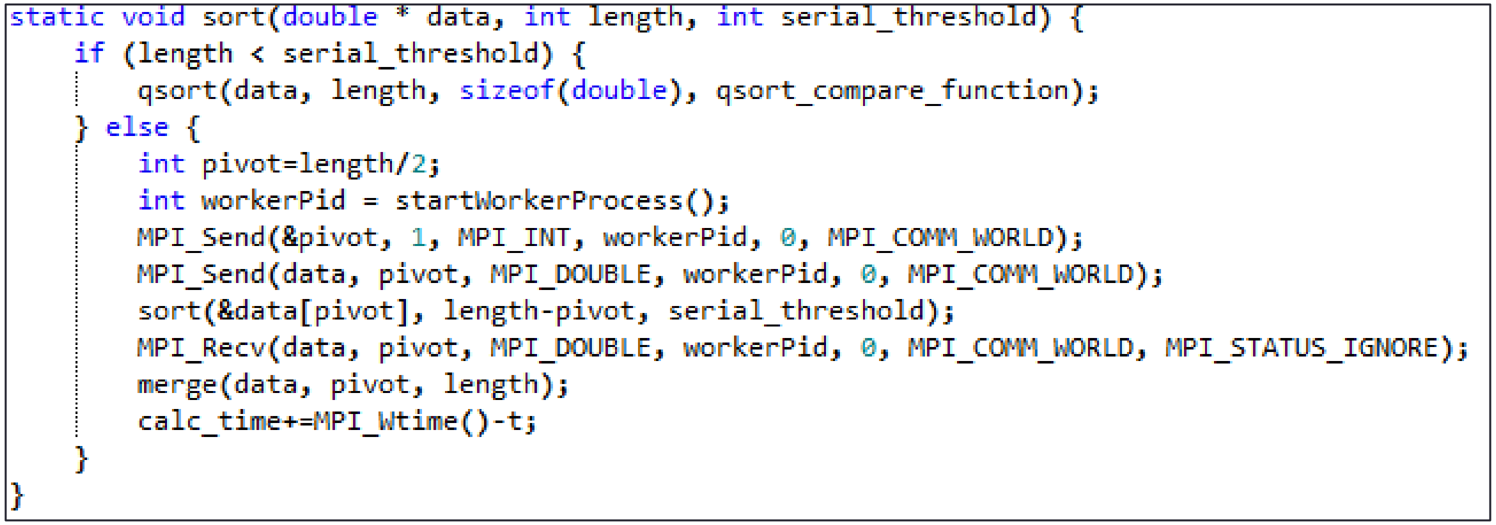
```c
static int startMergeSort(double * data, int length) {
  int workerPid = startWorkerProcess();
  MPI_Send(&length, 1, MPI_INT, workerPid, 0, MPI_COMM_WORLD);
  MPI_Send(data, length, MPI_DOUBLE, workerPid, 0, MPI_COMM_WORLD);
  return workerPid;
}

static void workerCode(int serial_threshold) {
  int workerStatus = 1;
  int parentId, dataLength;
  double * data;
  while (workerStatus) {
    parentId = getCommandData();
    MPI_Recv(&dataLength, 1, MPI_INT, parentId, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    data=(double*) malloc(sizeof(double) * dataLength);
    MPI_Recv(data, dataLength, MPI_DOUBLE, parentId, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    sort(data, dataLength, serial_threshold);
    if (parentId==0) shutdownPool();
    MPI_Send(data, dataLength, MPI_DOUBLE, parentId, 0, MPI_COMM_WORLD);
    free(data);
    workerStatus=workerSleep();
  }
}

static void sort(double * data, int length, int serial_threshold) {
  if (length < serial_threshold) {
    qsort(data, length, sizeof(double), qsort_compare_function);
  } else {
    int pivot=length/2;
    int workerPid = startWorkerProcess();
    MPI_Send(&pivot, 1, MPI_INT, workerPid, 0, MPI_COMM_WORLD);
    MPI_Send(data, pivot, MPI_DOUBLE, workerPid, 0, MPI_COMM_WORLD);
    sort(&data[pivot], length-pivot, serial_threshold);
    MPI_Recv(data, pivot, MPI_DOUBLE, workerPid, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    merge(data, pivot, length);
  }
}
```



Sample solutions also have some timing code to generate time spent in compute and comms

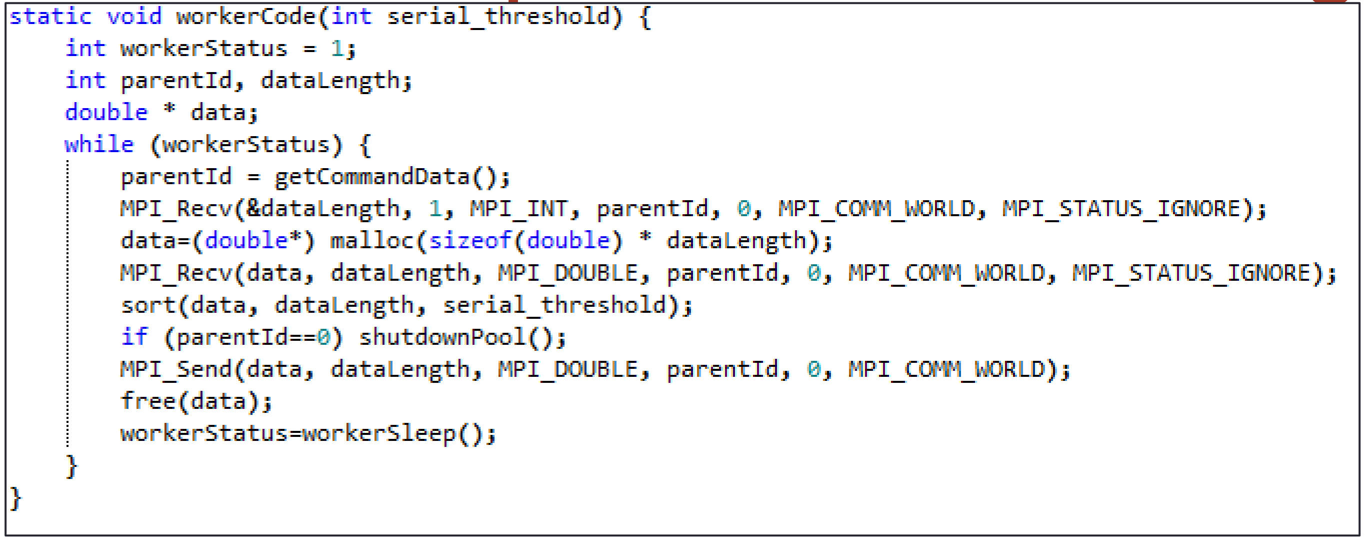
Reuse of process for half of data

- The fact that the existing worker is *reused* for half of the data is important
  - As otherwise workers would be sitting idle waiting for their children to complete and wasted concurrency

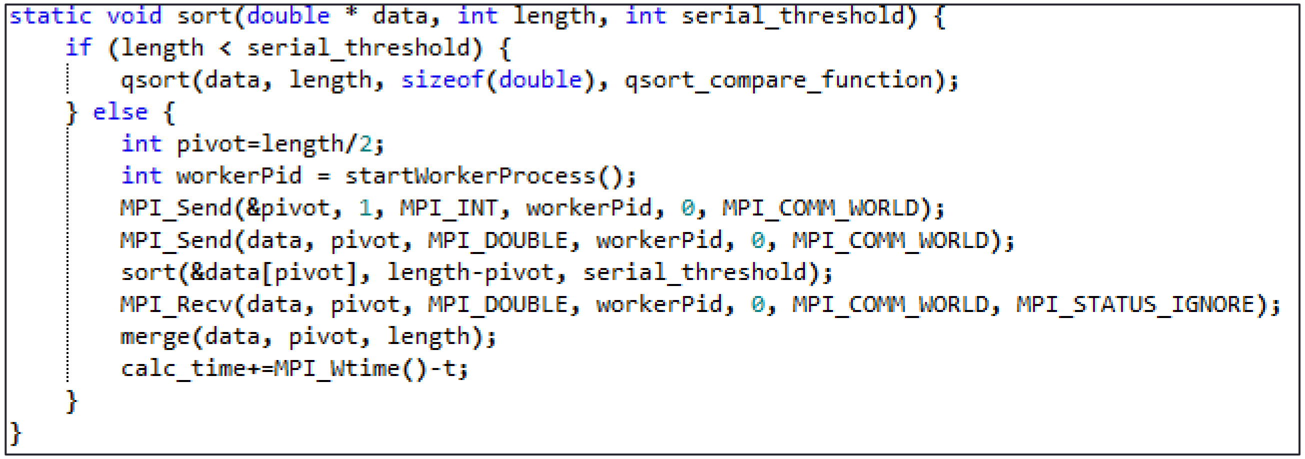


```c
static void sort(double * data, int length, int serial_threshold) {
    if (length < serial_threshold) {
        qsort(data, length, sizeof(double), qsort_compare_function);
    } else {
        int pivot=length/2;
        int workerPid = startWorkerProcess();
        MPI_Send(&pivot, 1, MPI_INT, workerPid, 0, MPI_COMM_WORLD);
        MPI_Send(data, pivot, MPI_DOUBLE, workerPid, 0, MPI_COMM_WORLD);
        sort(&data[pivot], length-pivot, serial_threshold);
        MPI_Recv(data, pivot, MPI_DOUBLE, workerPid, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        merge(data, pivot, length);
        calc_time+=MPI_Wtime()-t;
    }
}
```

Could improve P2P messages further



```
static void workerCode(int serial_threshold) {
    int workerStatus = 1;
    int parentId, dataLength;
    double * data;
    while (workerStatus) {
        parentId = getCommandData();
        MPI_Recv(&dataLength, 1, MPI_INT, parentId, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        data=(double*) malloc(sizeof(double) * dataLength);
        MPI_Recv(data, dataLength, MPI_DOUBLE, parentId, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        sort(data, dataLength, serial_threshold);
        if (parentId==0) shutdownPool();
        MPI_Send(data, dataLength, MPI_DOUBLE, parentId, 0, MPI_COMM_WORLD);
        free(data);
        workerStatus=workerSleep();
    }
}
```



```
static void sort(double * data, int length, int serial_threshold) {
    if (length < serial_threshold) {
        qsort(data, length, sizeof(double), qsort_compare_function);
    } else {
        int pivot=length/2;
        int workerPid = startWorkerProcess();
        MPI_Send(&pivot, 1, MPI_INT, workerPid, 0, MPI_COMM_WORLD);
        MPI_Send(data, pivot, MPI_DOUBLE, workerPid, 0, MPI_COMM_WORLD);
        sort(&data[pivot], length-pivot, serial_threshold);
        MPI_Recv(data, pivot, MPI_DOUBLE, workerPid, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        merge(data, pivot, length);
        calc_time+=MPI_Wtime()-t;
    }
}
```

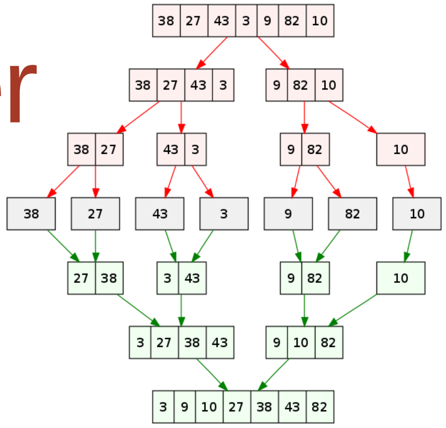
- Could potentially improve this by moving to one MPI message rather than two
  - Do a probe and grab the number of elements out of the MPI status

Computation vs overhead

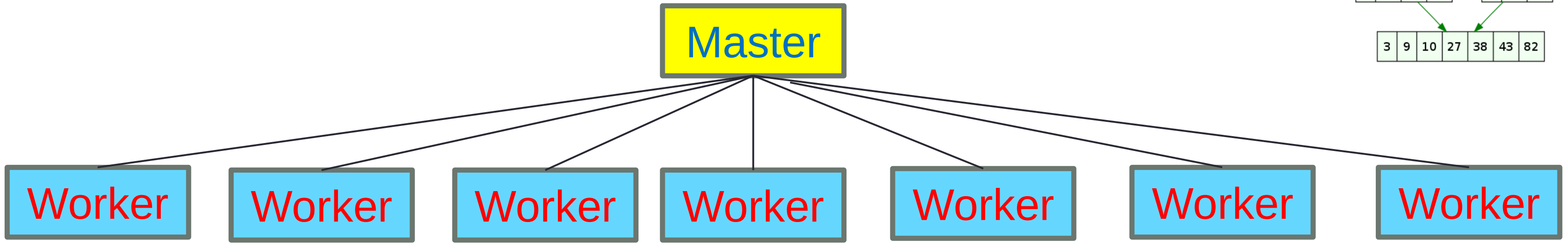
Data size	Serial threshold	Number workers	Task start up overhead	Comm time (s)	Compute Time (s)	Runtime (s)
100	10	16	6e-6	3.6e-5	2.0e-6	0.00125
1000	100	16	6e-6	4.0e-5	8.0e-6	0.00131
10000	2000	8	5e-6	1.2e-4	5.7e-4	0.00228
10000	1000	16	6e-6	8.4e-5	5.24e-4	0.00215
10000	100	128	1e-3	5.6e-3	2.2e-5	0.01559
1000000	100000	16	7e-6	3.1e-3	1.1e-2	0.05768
1000000	50000	32	5.7e-4	9.3e-3	5.6e-3	0.07008
1000000	10000	128	6.2e-4	1.3e-2	2.3e-3	0.08194
100000000	10000000	16	2.0e-5	0.34	1.50	6.27
100000000	1000000	128	5.8e-5	0.083	0.187	5.45

- Start up, communication and compute time are the average per task

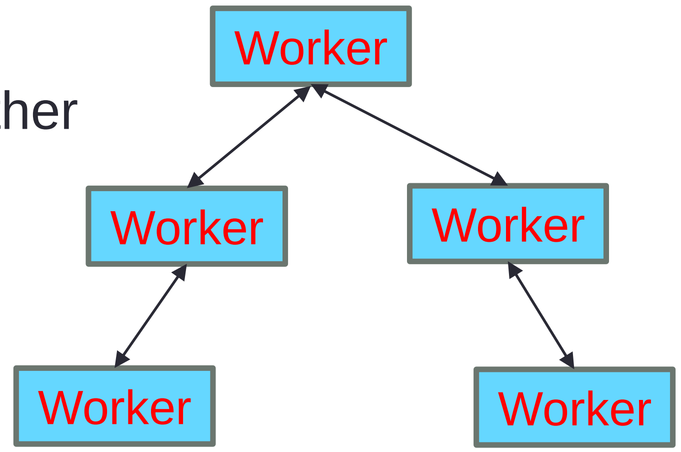
Process pool vs master/worker

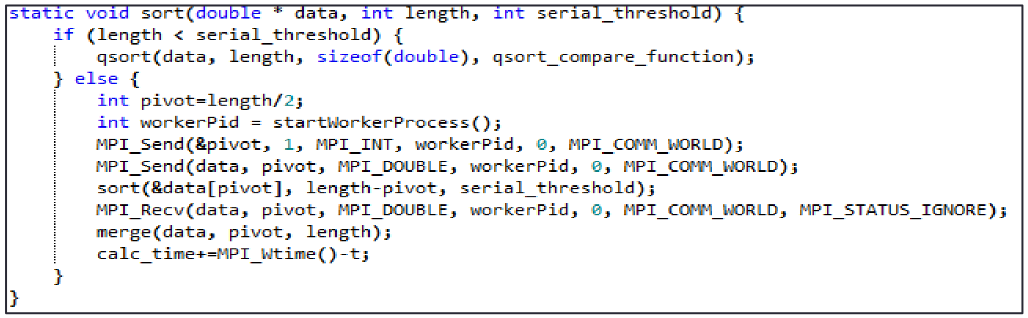


- The process pool is a (restricted) implementation of master/worker



- The limitation here is that there must be a process idle (and hence the number of processes must equal the maximum number needed).
  - Workers directly communicate with each other

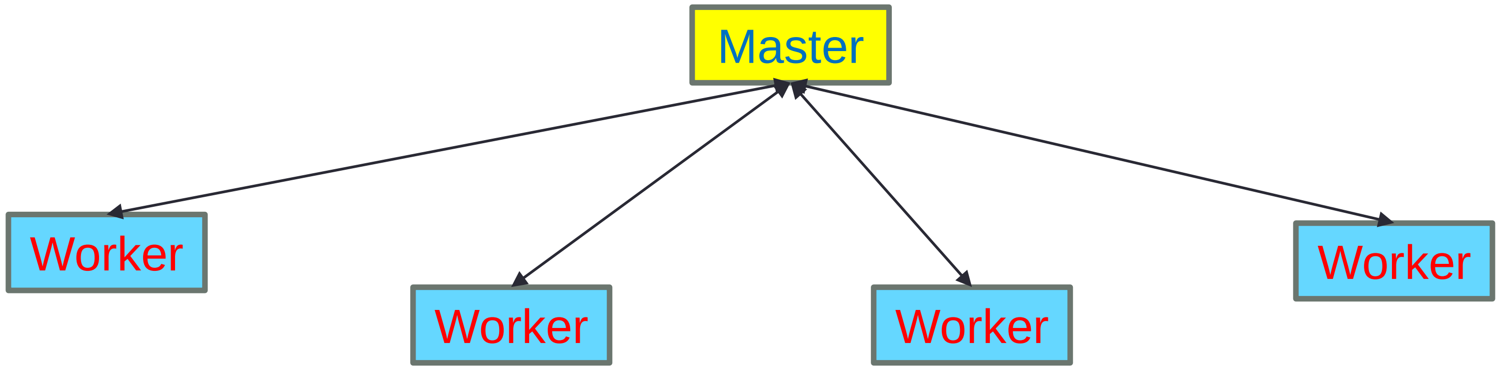




```
static void sort(double * data, int length, int serial_threshold) {
    if (length < serial_threshold) {
        qsort(data, length, sizeof(double), qsort_compare_function);
    } else {
        int pivot=length/2;
        int workerPid = startWorkerProcess();
        MPI_Send(&pivot, 1, MPI_INT, workerPid, 0, MPI_COMM_WORLD);
        MPI_Send(data, pivot, MPI_DOUBLE, workerPid, 0, MPI_COMM_WORLD);
        sort(&data[pivot], length-pivot, serial_threshold);
        MPI_Recv(data, pivot, MPI_DOUBLE, workerPid, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        merge(data, pivot, length);
        calc_time+=MPI_Wtime()-t;
    }
}
```

Process pool vs *full* master/worker

- It would be better to decouple this, and for units of work (the data to be sorted) to be sent to the master instead
  - The master could then store these and send them to an idle worker when one is available



  - The big benefit of this is that it decouples the problem size from the number of UE needed
  - However, will add to code complexity, not least on the merging of results