

# Matrix multiply Course project overview

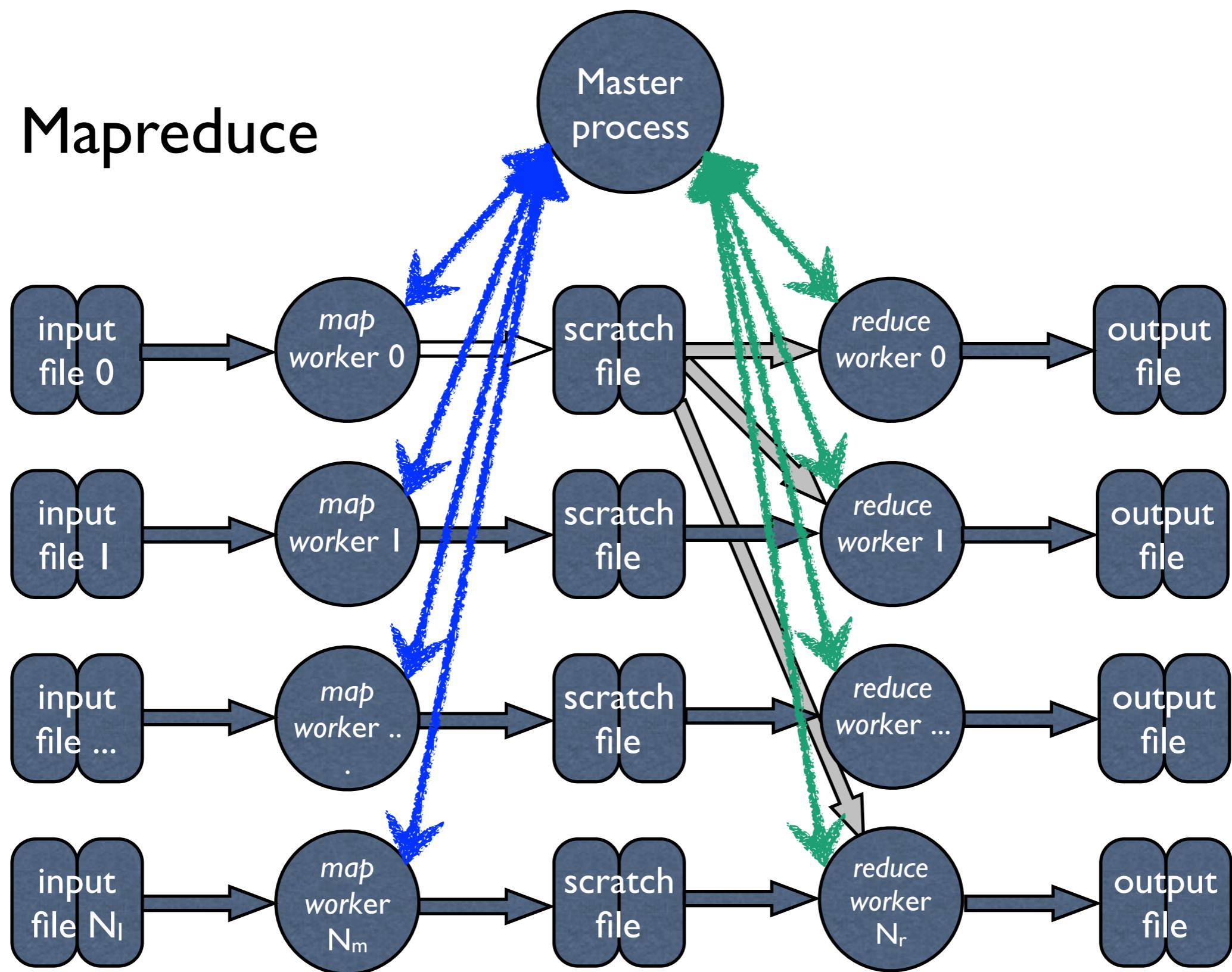
- One person teams
- implement a shared memory and distributed memory matrix multiply.
- This will be due the week before dead week
- There will be some reward, as yet determined and perhaps only psychological, for the top 10% fastest project
- You should use cache and communication strategies to achieve good results.
  - I will go over these.

# Map Reduce Course

## project overview

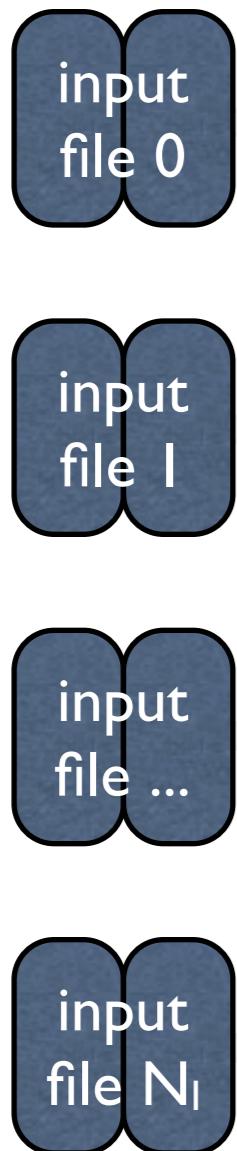
- Two person teams allowed
- Let me know if you are working on a team as soon as possible
  - Both team members should contact me
  - An easy way to do this is for one team member to send email copying the other, and the other to respond affirmatively
- This is the default project -- you can propose a different one
  - Let me know as soon as possible if you are doing this, and we should finalize the project by the end of February

# Mapreduce



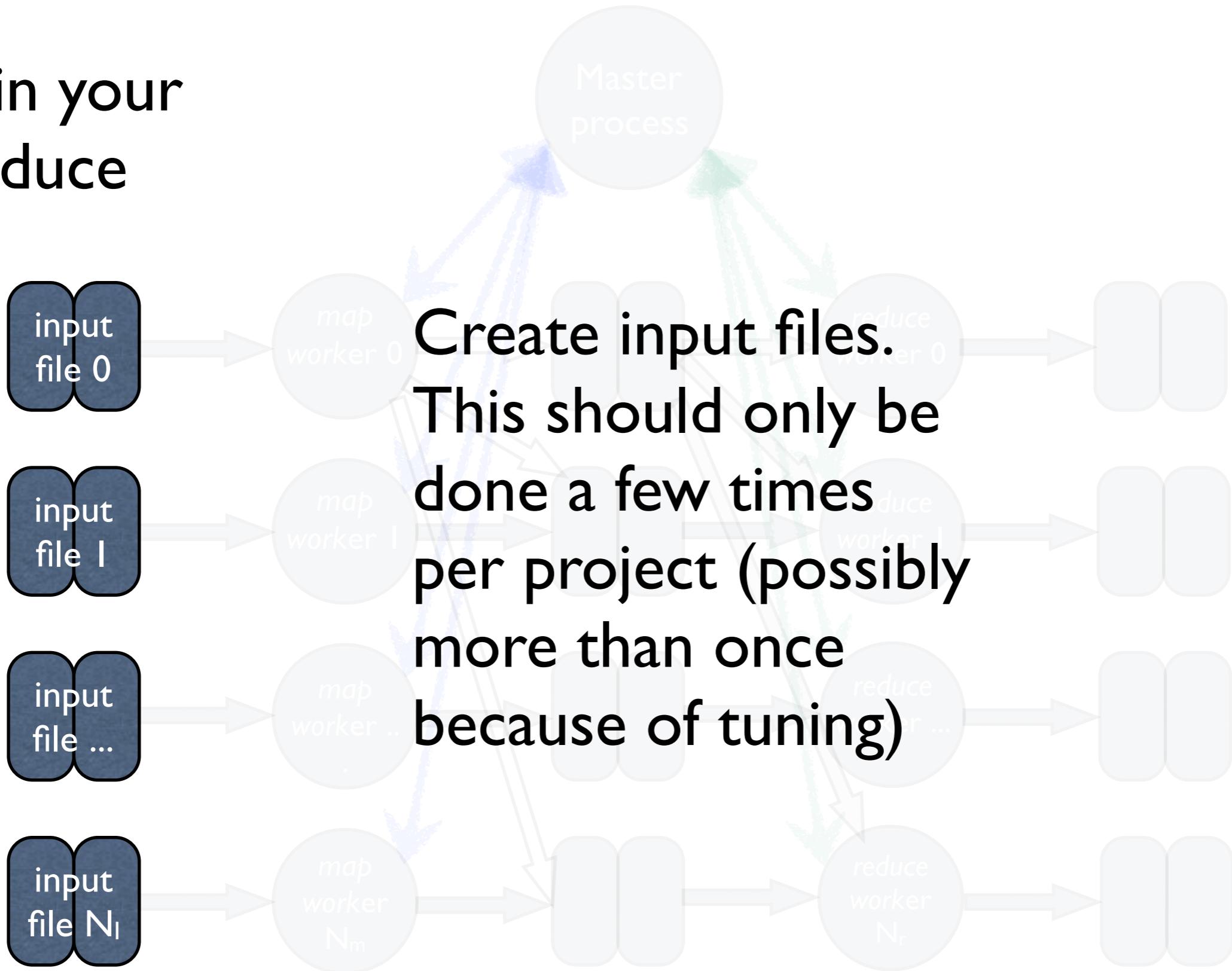
*Even though this shows one input file for mappers and reducers, there should be more than one.*

# Program startup and initialization

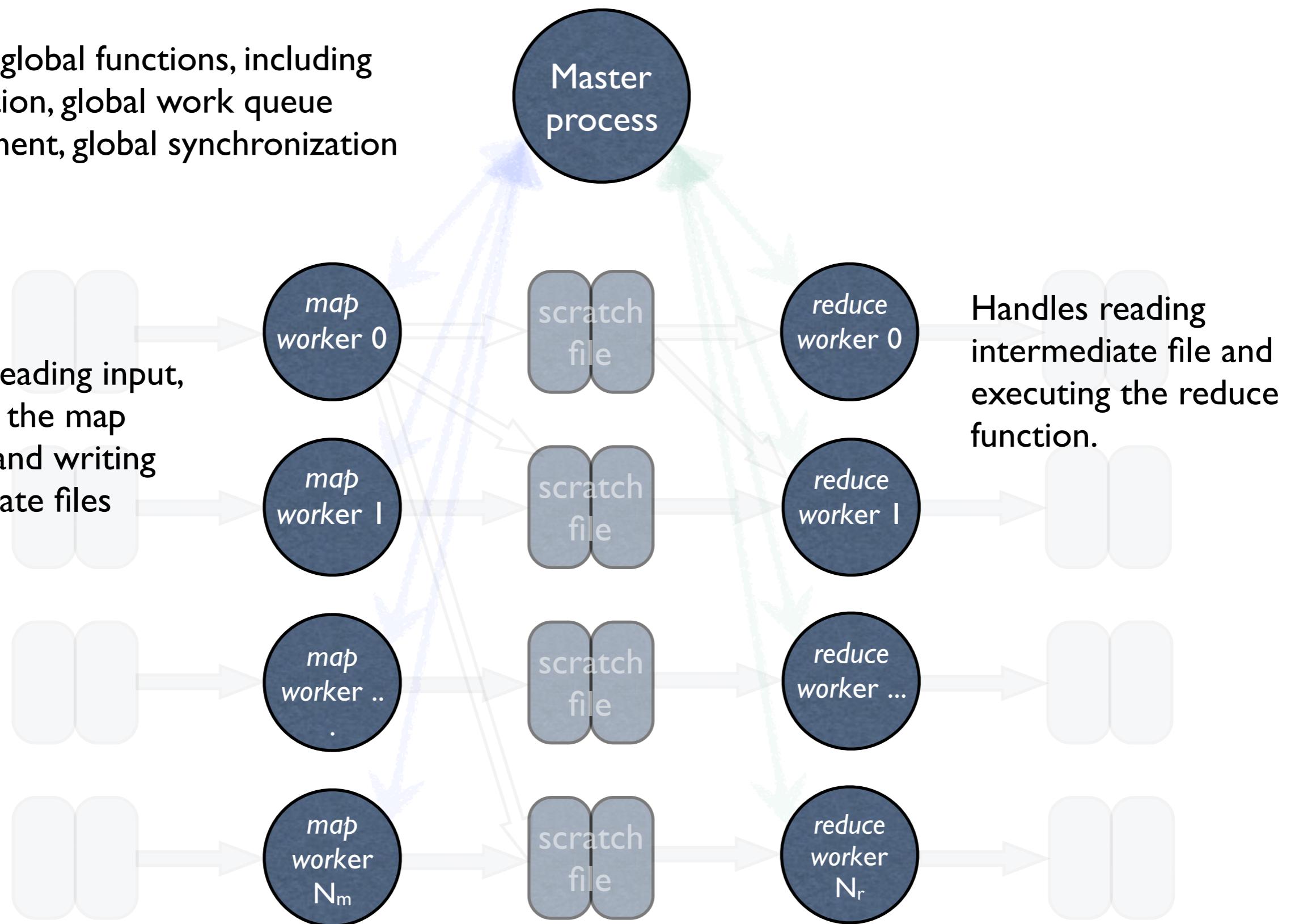


- Input is divided into multiple files prior to program execution beginning
  - No processing of data when dividing it into multiple files.
    - You can do the file dividing before running the program.
    - In any case dividing the files does not need to be counted in your run time.
    - I will supply raw input, you will need to divide

# Steps in your mapreduce

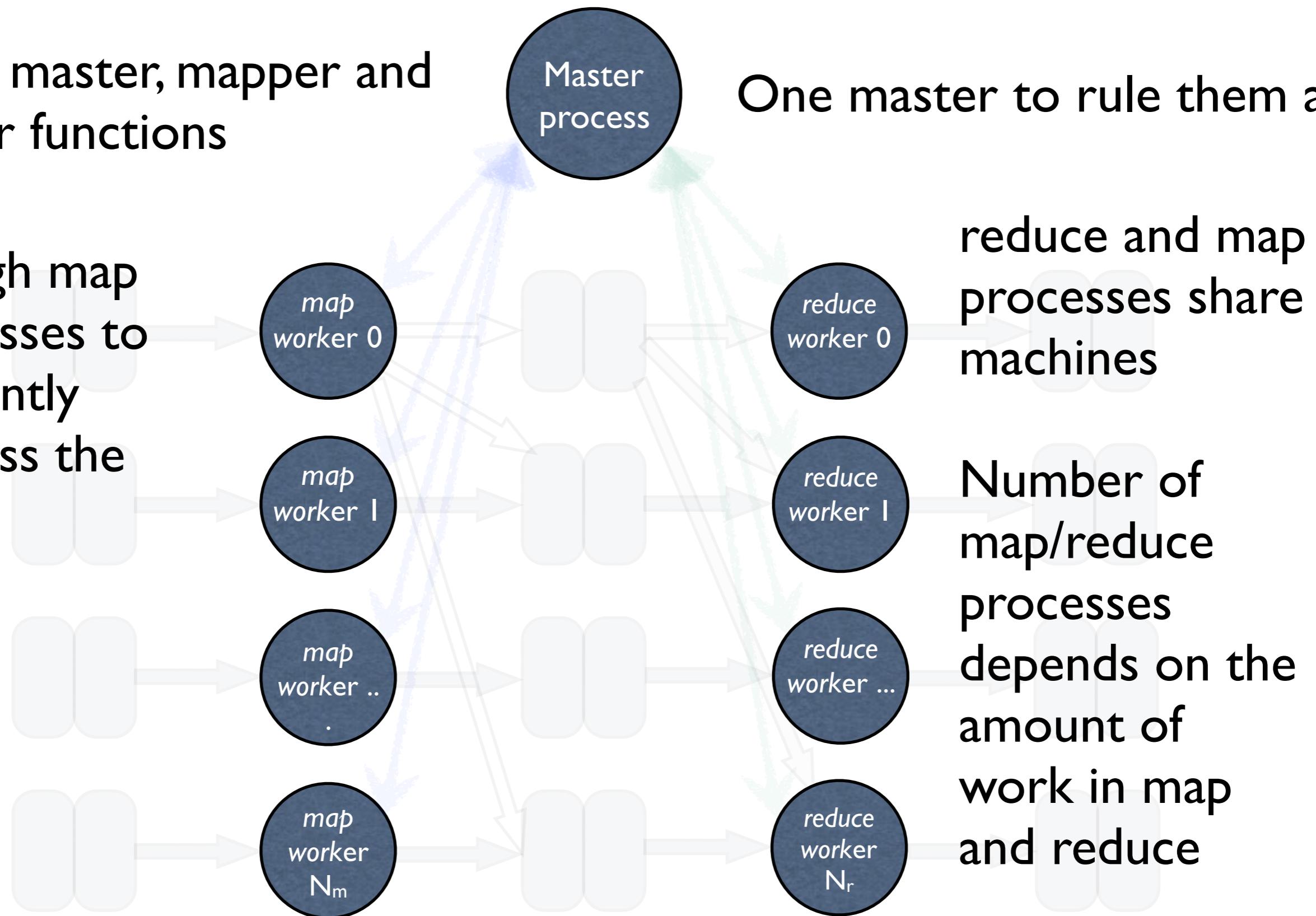


Handles global functions, including initialization, global work queue management, global synchronization



Intermediate files allow buffered high volume communication between mappers and reducers

Create master, mapper and reducer functions



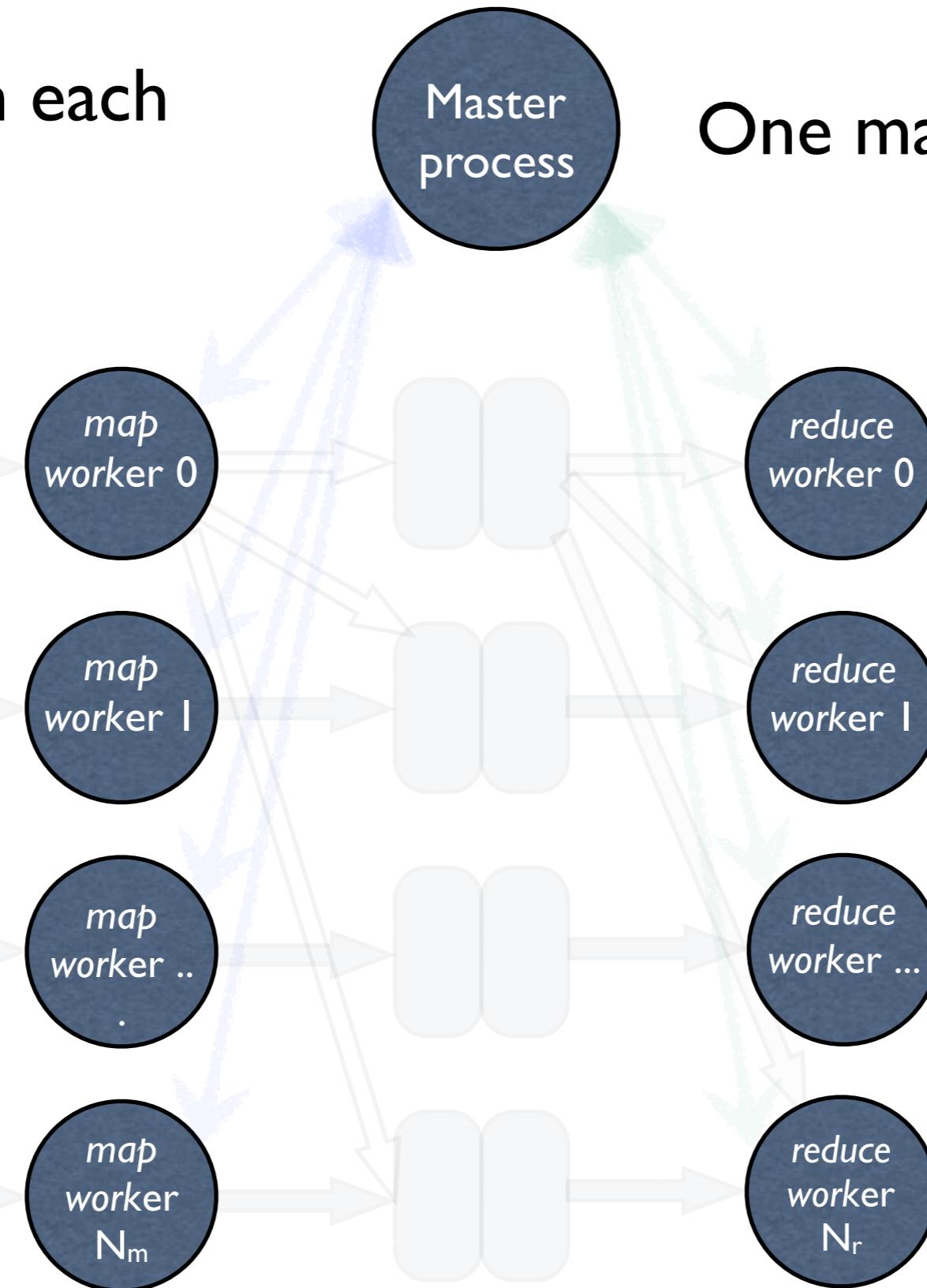
$$\text{Total processes} = 1 + \text{reduce procs} + \text{map procs}$$

# Threads needed in each process

$k_{MapRead}$  threads to read input. May be more than number of cores for best performance because of waiting for I/O to finish. Has an effect on file types and number of input files.

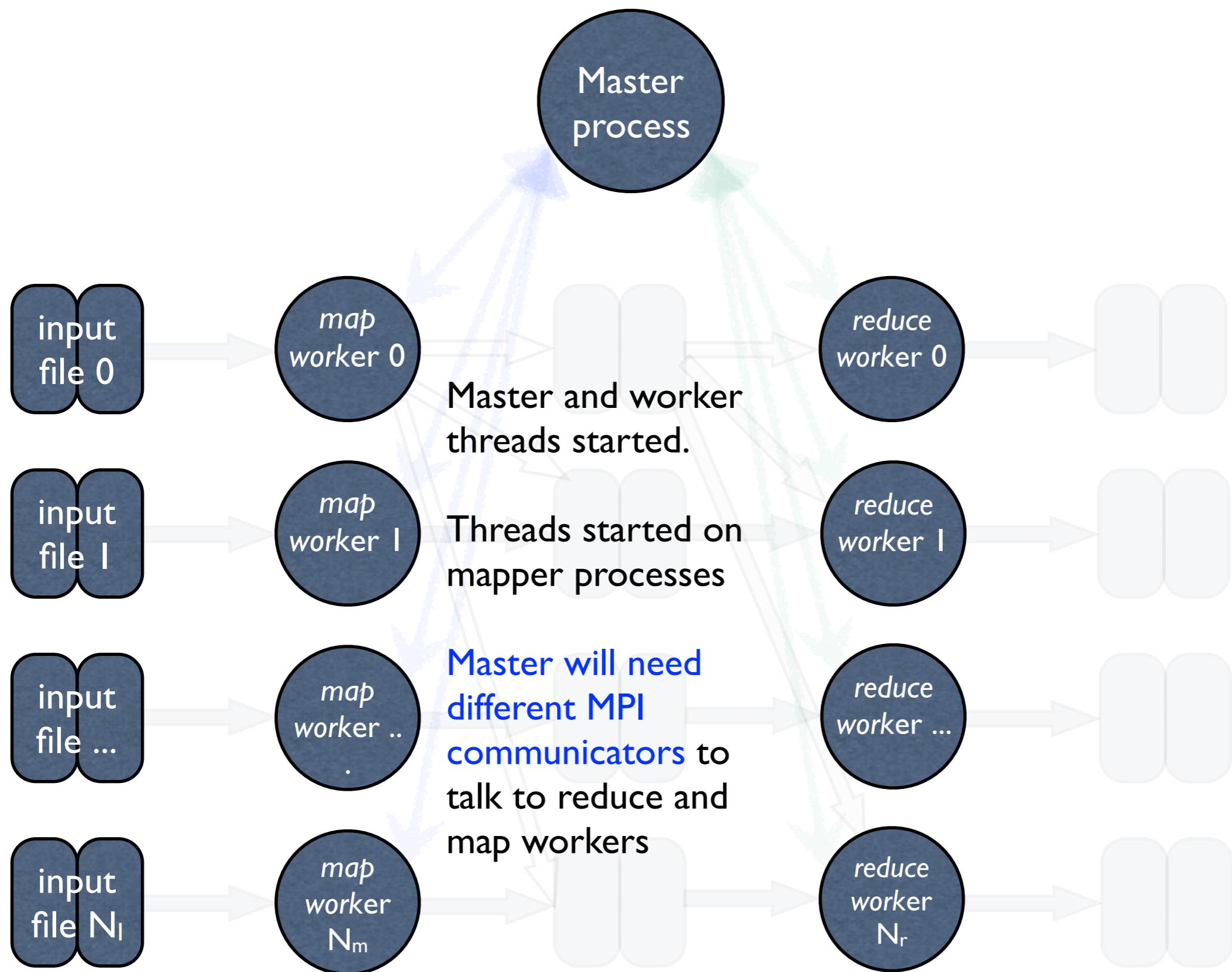
$k_{MapWrite}$  threads to write intermediate files. Issue concerning the number of threads are the same as for  $k_{MapRead}$

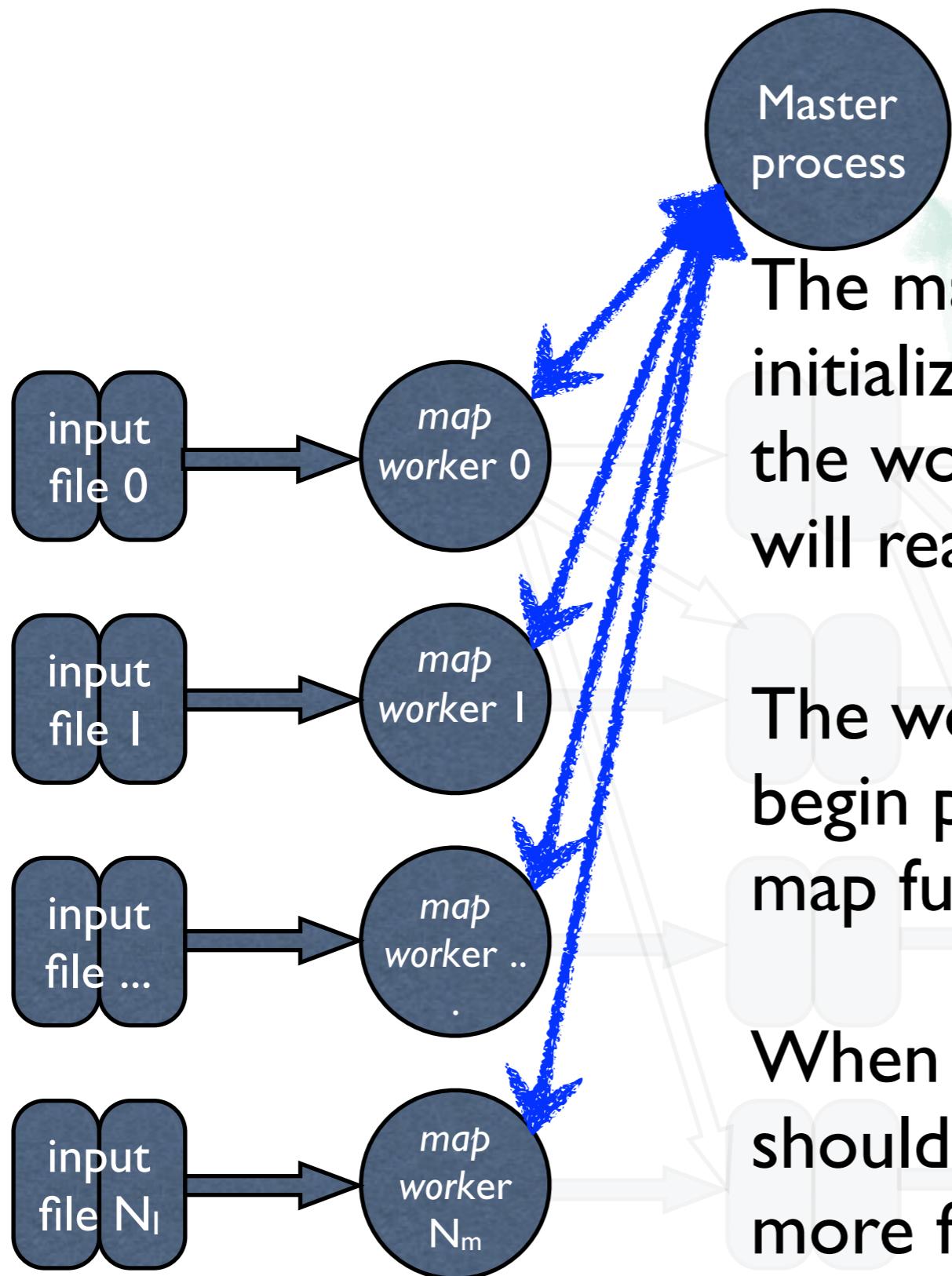
$k_{Map}$  threads to perform map. One per core is probably right.



## One master thread

The same considerations for mappers need to be taken into account for threads on reduce processors

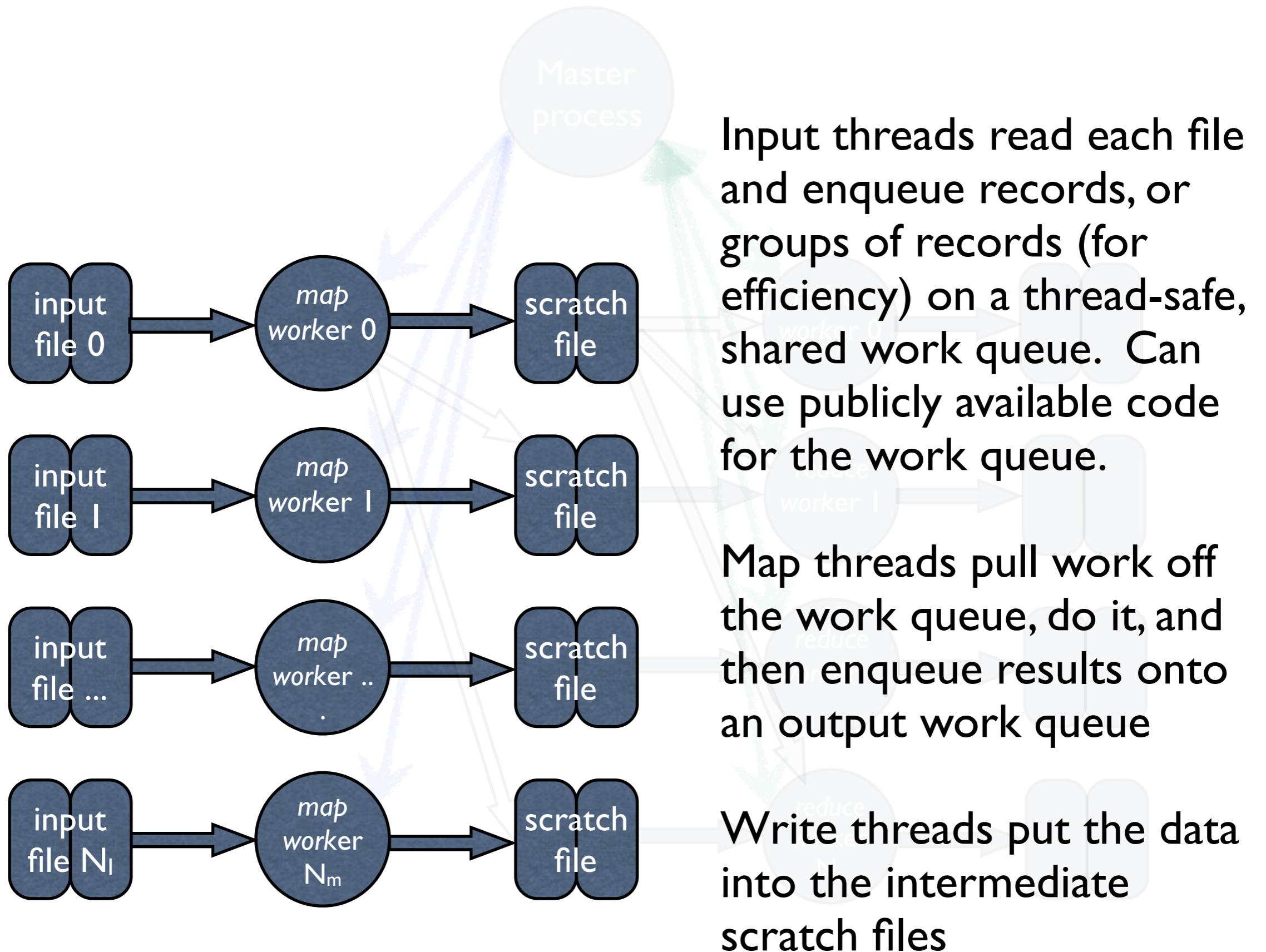


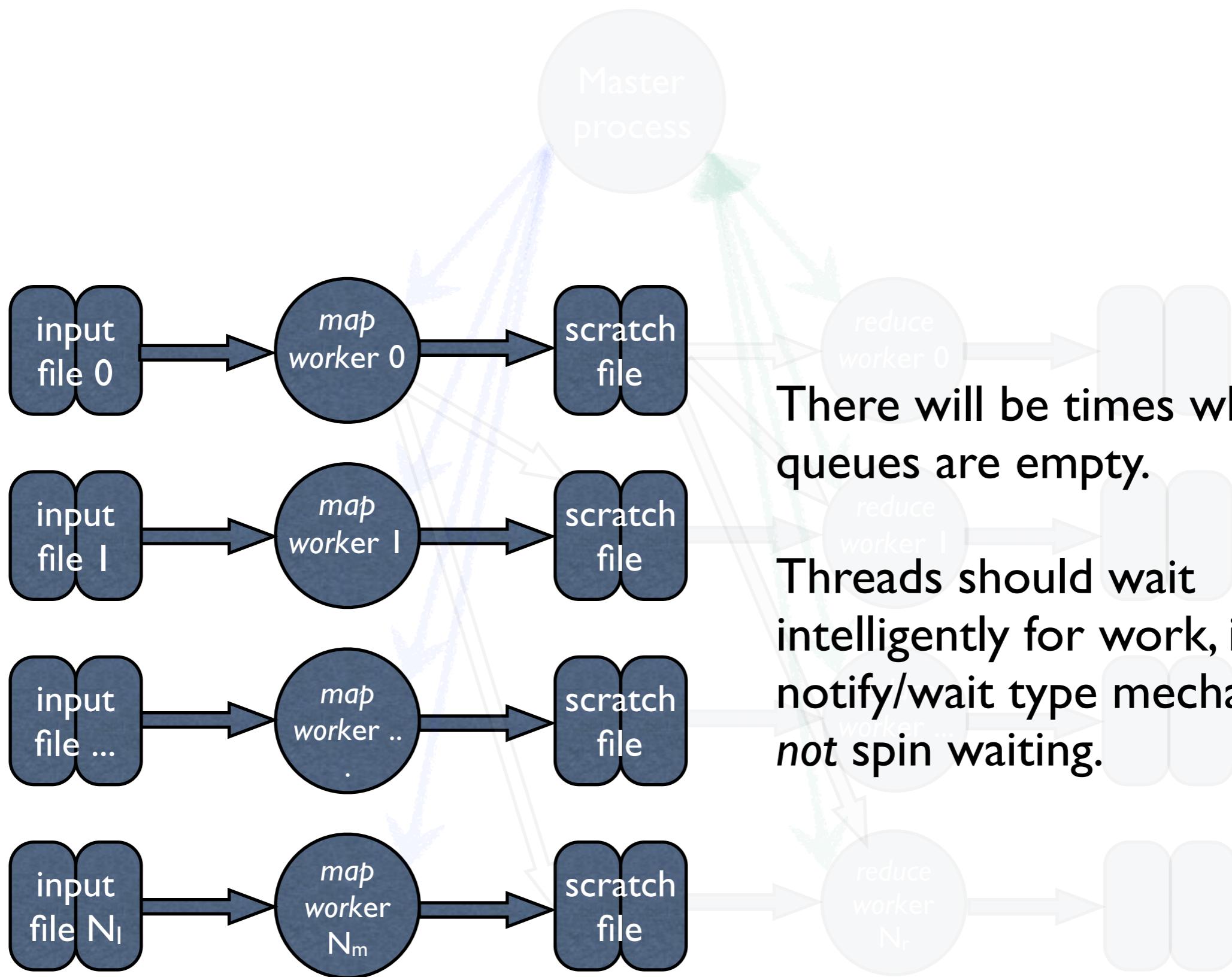


The master thread performs MPI initialization and communicates to the workers the first file(s) each will read.

The workers read the file and begin processing the data with the map function

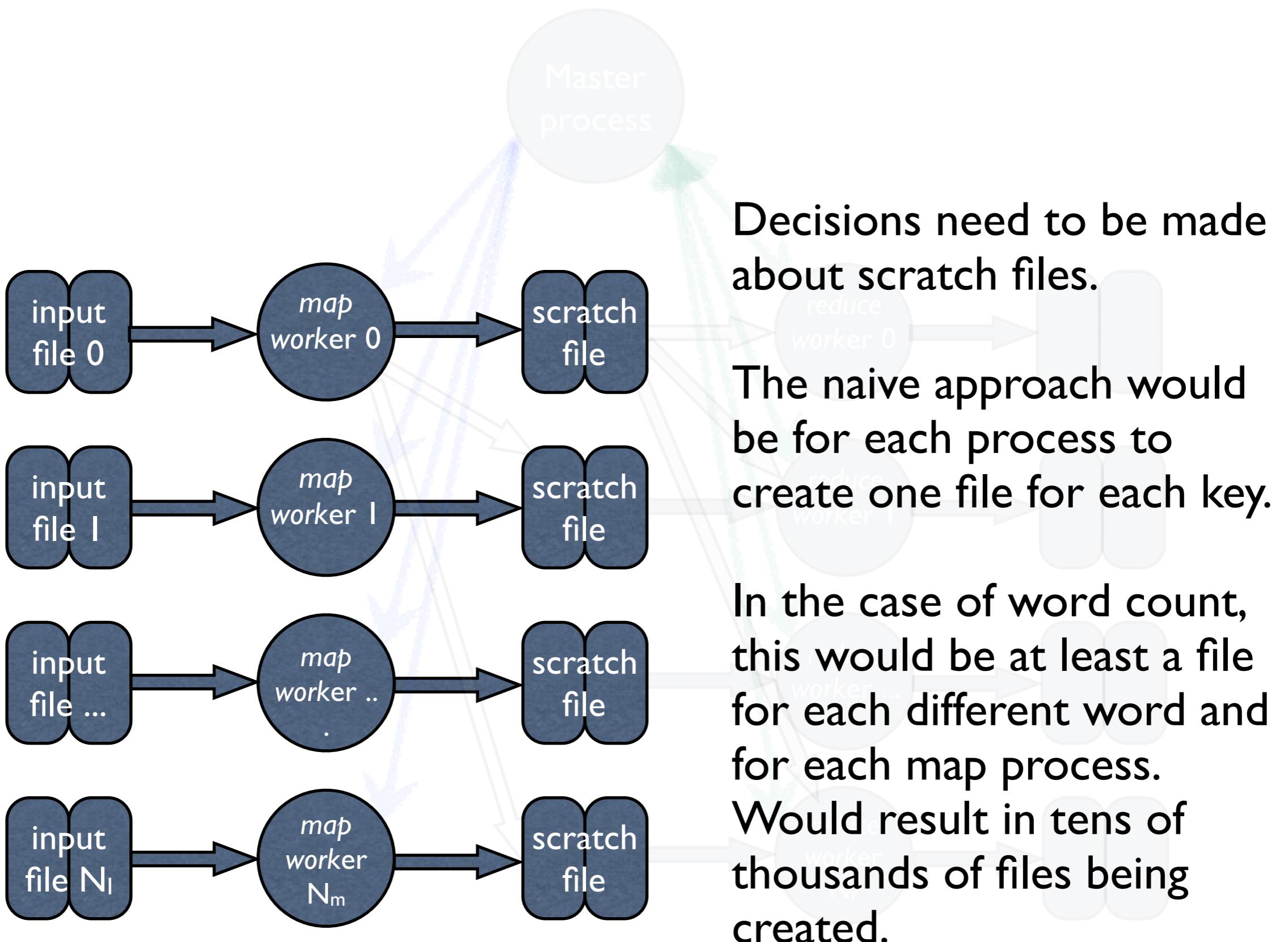
When done with a file they should go to the Master to see if more files need to be read. The master serves as a work queue.

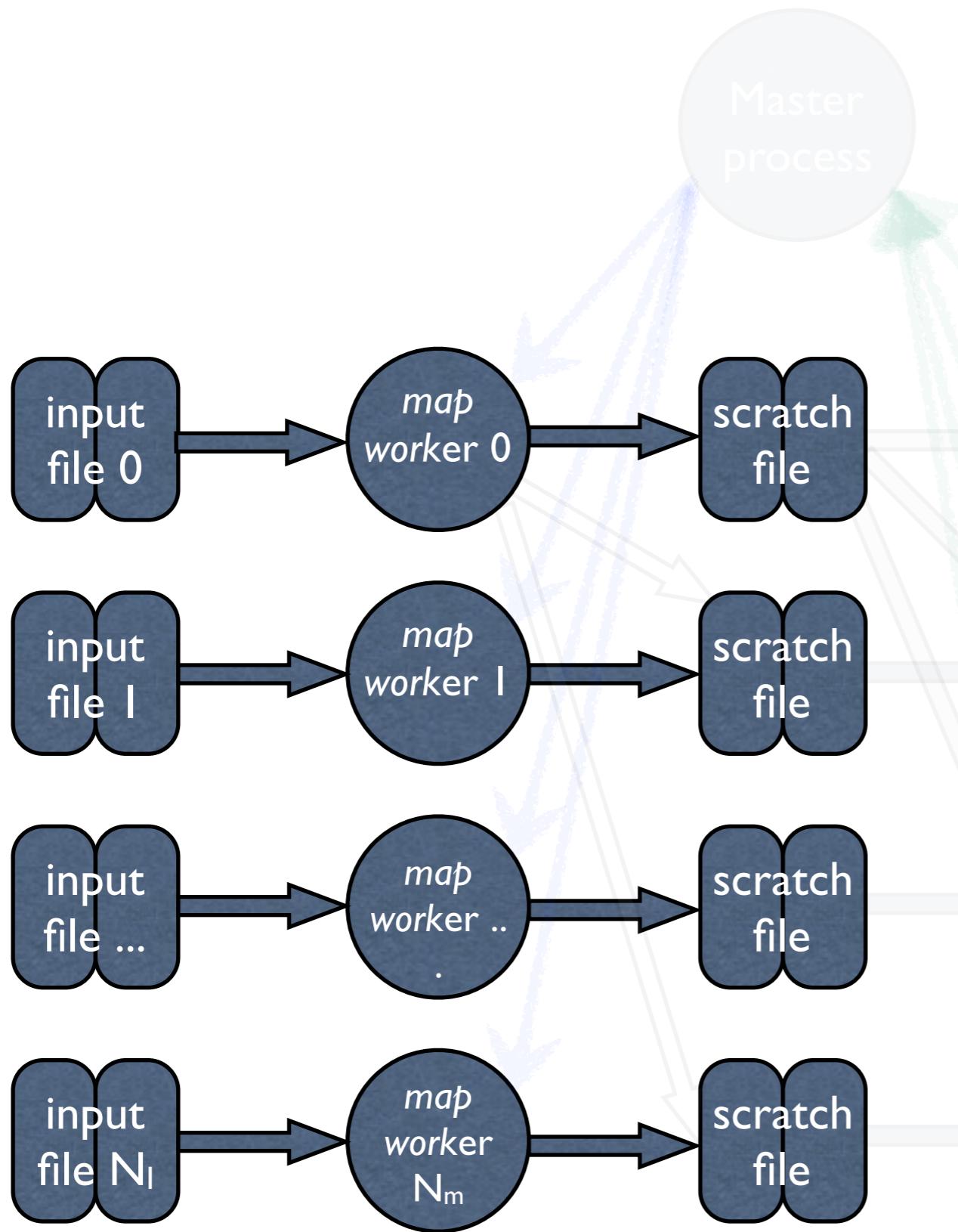




There will be times when queues are empty.

Threads should wait intelligently for work, i.e. use notify/wait type mechanisms, not spin waiting.

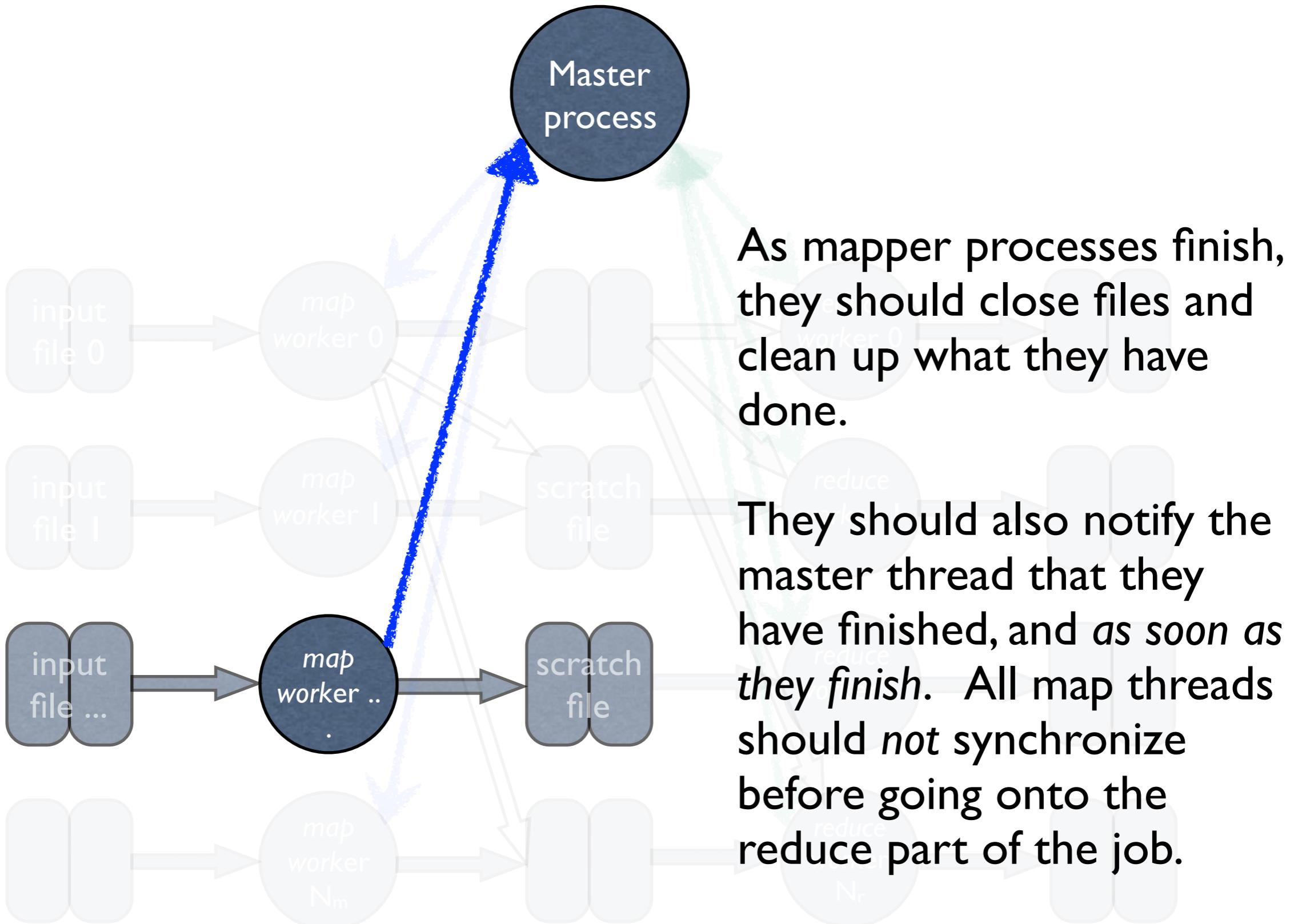




We also do not want one file created by each map process for each reduce process (i.e. approximately  $\lceil |keys|/|reducers| \rceil$  keys per file. Would be bad for load balancing purposes.

We want more files than there are reduce processes.

Want  
 $\lceil |keys|/(k_{ReduceFiles} * |reducers|) \rceil$   
 keys/file, where  $k_{ReduceFiles}$  is a number you determine.

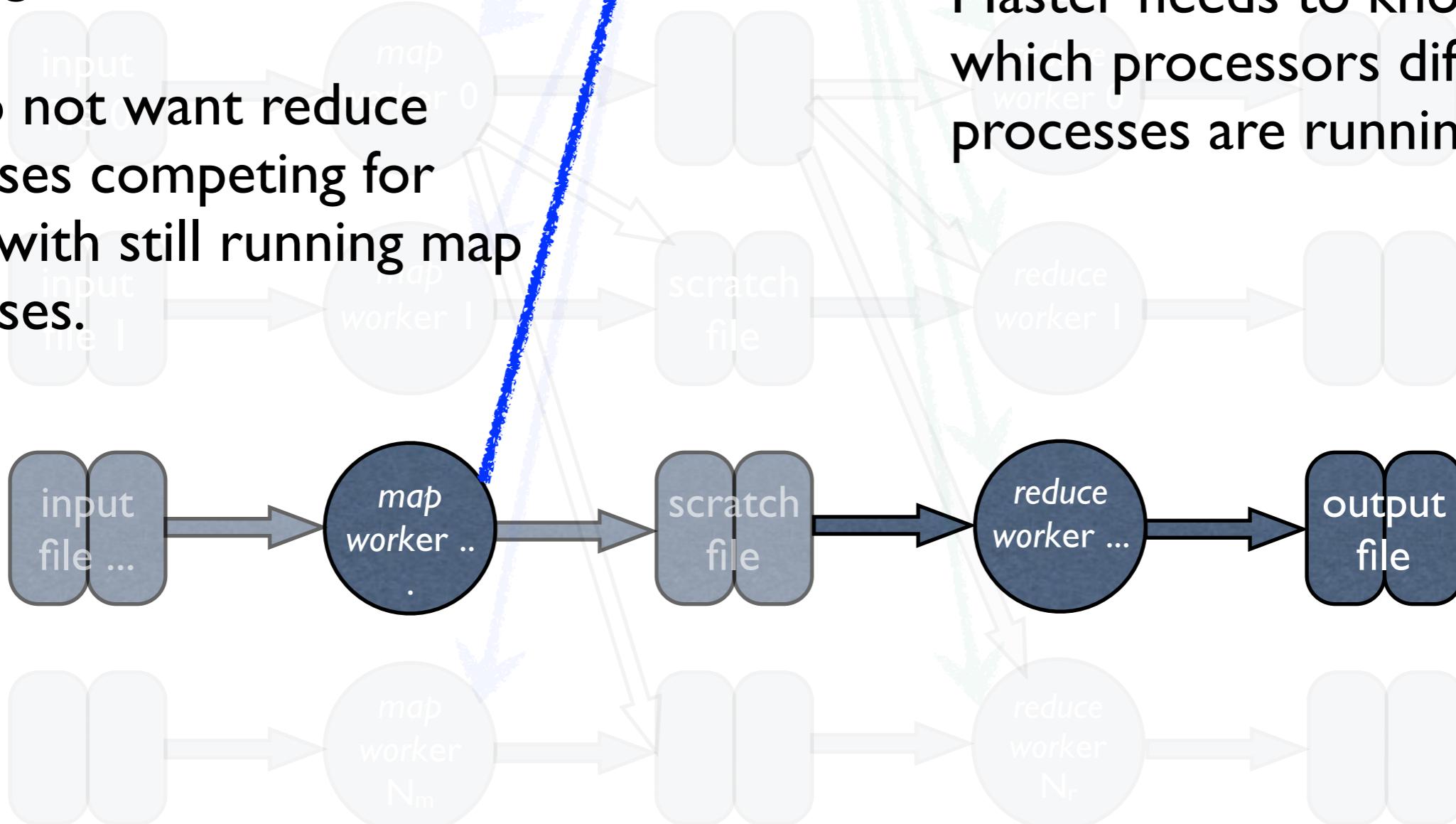


Only the reduce process(es) running on the same host as the map thread that is finishing should be started.

You do not want reduce processes competing for cycles with still running map processes.



Master needs to know which processors different processes are running on.



# what machines are executing what maps and reduces? processes execute on

- Given a map process running on an MPI process with rank  $r$ , what reduce functions (with rank  $r' \neq r$ ) run on the same physical process?
- `gethostname()` returns the standard hostname for the system
- The master process can build a mapping from MPI processes to machine names using this

# hostname() on linux should be Windows binding

```
dynamo 186% cat test2.c
```

```
#include <stdio.h>
```

```
#include <unistd.h>
```

```
#include <string.h> // this allows us to manipulate text strings
```

```
int main(int argc, char* argv[]) {
```

```
    char name[256];
```

```
    int rc;
```

```
    rc = gethostname(name, 256);
```

```
    if (!rc) printf("%s\n", name); // prints out greeting to screen
```

```
}
```

```
dynamo 187% ./a.out
```

```
dynamo.ecn.purdue.edu
```

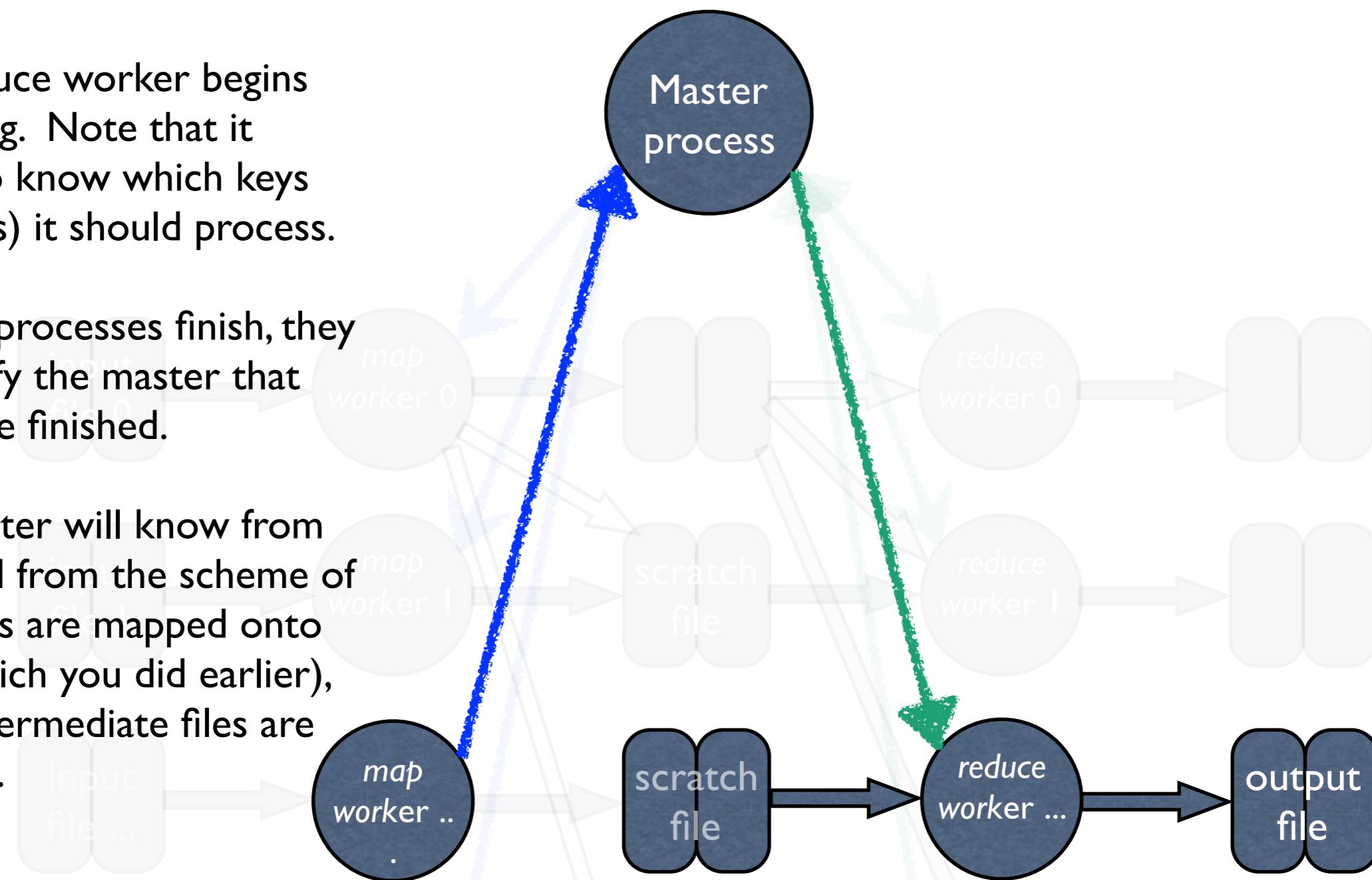
char array to hold  
the host's name

Length of the array --  
max hostname length is 256  
characters

The reduce worker begins executing. Note that it needs to know which keys (and files) it should process.

As map processes finish, they will notify the master that they have finished.

The master will know from this, and from the scheme of how keys are mapped onto files (which you did earlier), what intermediate files are available.



The master process will use the scheme to map keys (I'll call it the hash function). The hash function will map all keys onto  $kReduceFiles * |reducers|$  different files. When a reducer is ready for work, it will give one of the  $kReduceFiles * |reducers|$  different sets of file files to that reducer. When all those files are consumed, the reducer can ask for another file.

When all work is handed out, the master will respond that no more work is available, and the reduce process will terminate.

# Deliverables

- Working code and a possible demonstration of it for me before dead week.
  - You will only need to do word count
- a writeup, no more than ten pages. Grading will not be by length, so don't use more space than necessary. The writeup should contain
  - Information about issues encountered
  - Performance bottlenecks -- i.e. why your speedup isn't perfect
  - performance numbers showing ...
- The best projects will get a chance to explain what they did during dead week lectures

# Performance numbers

- load imbalance within a node, among threads,
- load imbalance across nodes, within maps and reduces
- curves showing I/O performance vs numbers of threads (look for the knee in the curve)
- numbers showing performance (speedup) within nodes with different number of map and reduce (i.e. not I/O) threads
- Overall performance

# Grading criteria

- 50% of the grade will come from having working code
- 45% for the writeup, in particular I need to see that you understand why your application performs the way it does. This is about as important as raw speedup.
- 5% for not being “too far away” from acceptable in speedup. This will be a moving target, *overall in your favor*
  - if no one gets speedups, not having slowdowns “significantly worse” than others will be ok
  - If the average is linear speedups, getting within a factor of some  $c$  to be determined will be ok.

# A possible implementation strategy —you may have a better way

- Implement a single map
  - Determine how many data read threads are best ( $k_{MapRead}$ ) (collect data)
  - Write a map, integrate with a work queue
    - determine number map computation threads are needed ( $k_{Map}$ ) (collect data)
    - At this point  $k_{MapRead}$  is fixed
  - Determine how many output threads are needed ( $k_{MapWrite}$ ) (collect data)
- Integrate with a master thread

# A possible implementation strategy (2)

- Implement a single reduce
  - Determine how many data read threads are best ( $k_{MapRead}$ ) (collect data)
  - Write a reduce, integrate with a work queue
    - determine number map computation threads are needed ( $k_{Map}$ ) (collect data)
    - At this point  $k_{MapRead}$  is fixed
  - Determine how many output threads are needed ( $k_{MapWrite}$ ) (collect data)
- Integrate with a master thread

# A possible implementation strategy (3)

- Get the handoff between map and reduce threads working
- Final system tuning, data collection, report writing, etc.

# Clean up after runs

- Do not leave zombie processes, dead files, etc. in public /tmp and other spaces
- This will sabotage other's efforts and lead to bad results for you if it happens enough.