Power point project overview

This is a suggestion – you make the final decision on how to implement this.

Ground rules

- Two person teams are allowed
  - If you are working on a team, let me know as soon as possible
  - One team members should send me an email letting me know what the team is, and copy the other other team member on the email
- This is the default project, I'm happy to have you propose another one
  - If you have something closer to your research project, or that just sounds like more fun, let me know
  - It should have an MPI and an OpenMP component
  - We should finalize by the end of February

The project is a *map-reduce* project

- Map-reduce has two main parts
  - A mapper collects object together that will be reduced, and sends them to the appropriate reducer
  - Mappers can also do combining, which is a form of early reducing
  - A reducer takes the items mapped onto it, and reduces them
- Let's look at an example

The mapper

- Mappers always map, and they can also combine
- Our map-reduce will count words
- The combine function of our word-count mapper is to combine the same words processed by a mapper into a single entry, along with the count
- Mapper 0 “the”, mapper 3 “dog” have been combined

Mapper 0	Mapper 1	Mapper 2	Mapper 3
dog	the	and	have
cat	pool	no	seen
horse	and	one	dog
bill	cat	knows	and
the	pig	the	bill
and	dog	trouble	pig
the	head	I	dog

Mapper 0

dog	1
cat	1
horse	1
bill	1
the	2
and	1

Mapper 1

the	1
pool	1
and	1
cat	1
pig	1
dog	1
head	1

Mapper 2

and	1
no	1
one	1
knows	1
the	1
trouble	1
I	1

Mapper 3

have	1
seen	1
dog	2
and	1
bill	1
pig	1

The mapper (2)

dog	1
cat	1
horse	1
bill	1
the	2
and	1

Mapper 0

the	1
pool	1
and	1
cat	1
pig	1
dog	1
head	1

Mapper 1

and	1
no	1
one	1
knows	1
the	1
trouble	1
I	1

Mapper 2

have	1
seen	1
dog	2
and	1
bill	1
pig	1

Mapper 3

- Mappers also map!
- The mapping should ensure that all combined instance of some word *w* end up on the same reducer r_w
- An easy way to do both the combining and the mapping is to use a hash function.

Hash functions H(w)

- A hash function int H(…) maps an input argument into an int called a *hash code*.
- Given a has function that maps words into an int, *H(w) == H(w)*, i.e., the result of the hash function for identical inputs is always the same.
- The value returned the hash function, *h*, such that $0 \leq h \leq 2^N - 1$, where N is the number of bits in the hash function result
- If $0 \leq h \leq 127$, and we have want a hash value from $0 \leq h' \leq v$, we can use the value $h' = h \bmod v$.
- An easy hash function for words is to exclusive or characters, or pairs of characters, together to form an 8 or 16 bit hash code

A simple hash function that yields 8 bits

letter	a	l (el)	u	m
Value, 8 bit ascii	01100000	01101100	01110101	01101110

a	01100000
l	01101100
xor	00001100

	01100000
u	01110101
xor	00010101

	00010101
m	01101110
h	01111011

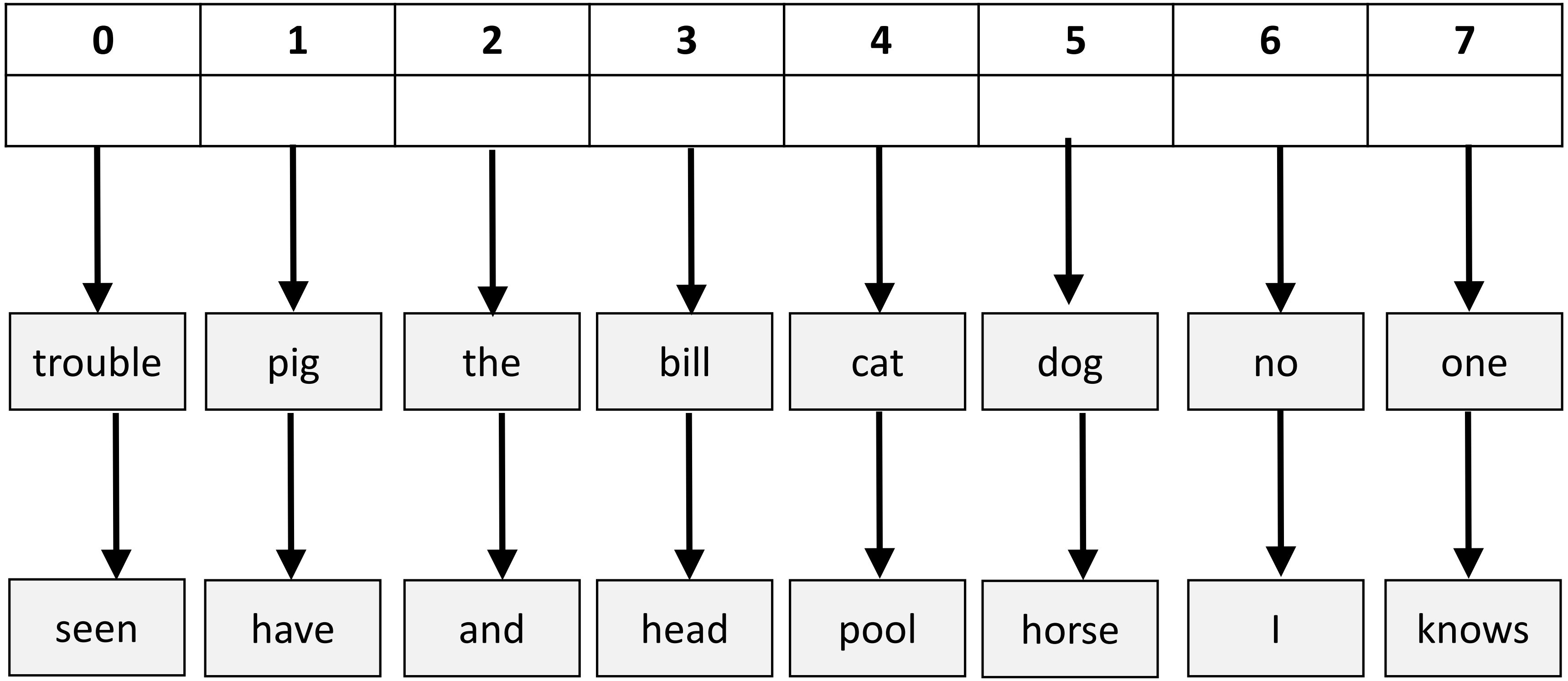
h = 64 + 32 +16 + 8 + 3
= 123

A simple hash function that yields 16 bits

letter	a	l (el)	u	m
Value, 8 bit ascii	01100000	01101100	01110101	01101110

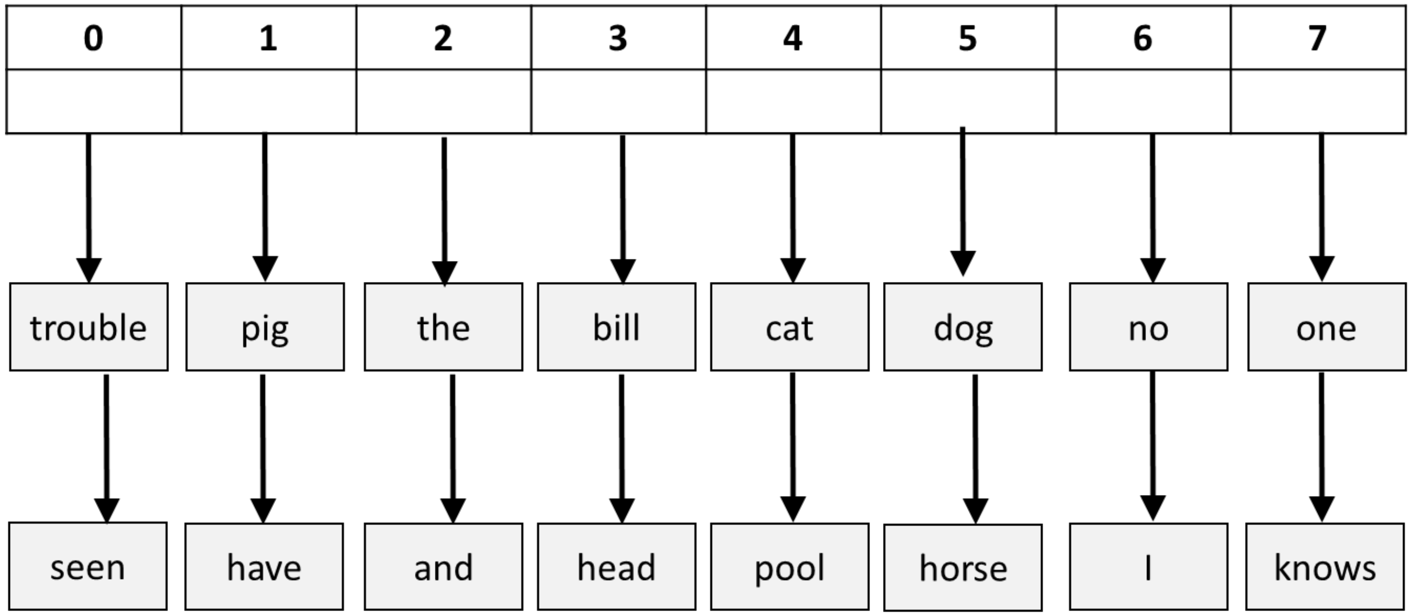
a l	01100000	01101100
u m	01110101	01101110
xor	00010101	00000010

Hash maps



word	h
dog	5
cat	4
horse	5
bill	3
the	2
and	2
pool	4
pig	1
head	3
no	6
one	7
knows	7
trouble	0
I	6
have	1
seen	0

Hash maps



By increasing the number of buckets, the number of words per bucket can be decreased. We can do searches in O(1) time!

- Hash maps allow rapid lookup of words
- Hash a word, and then look in the *bucket* to see if the word is there
  - If so, it is found and the entry can be returned or updated
  - If not, it is not there and can be added
- The word in this case is the key
- Hashmaps work with any key that can be used to form a value h

dog
cat
horse
bill
the
and
the

Mapper 0

the
pool
and
cat
pig
dog
head

Mapper 1

and
no
one
knows
the
trouble
I

Mapper 2

have
seen
dog
and
bill
pig
dog

Mapper 3

word	h
dog	1
cat	0
horse	1
bill	1
the	0
and	0
pool	0
pig	1
head	1
no	0
one	1
knows	1
trouble	0
I	0
have	1
seen	0

0	**1**
cat/1	dog/1
the/1	horse/1
and/1	bill/1

dog
cat
horse
bill
the
and
the

Mapper 0

the
pool
and
cat
pig
dog
head

Mapper 1

and
no
one
knows
the
trouble
I

Mapper 2

have
seen
dog
and
bill
pig
dog

Mapper 3

0	**1**
cat/1	dog/1
the/2	horse/1
and/1	bill/1

Combining with hash tables

0	**1**
the/1	pig/1
pool/1	dog/1
and/1	head/1
cat/1	

0	**1**
and/1	one/1
no/1	knows/1
the/1	head/1
trouble/1	
I/1	

0	**1**
seen/1	have/1
and/1	dog/2
the/1	bill/1

word	h
dog	1
cat	0
horse	1
bill	1
the	0
and	0
pool	0
pig	1
head	1
no	0
one	1
knows	1
trouble	0
I	0
have	1
seen	0

Now we need to map

- Assume we have 4 reducers
- Let $h' = h \bmod 4$. Then h' gives the reducer that a word goes to.
- Thus, all dogs go to reducer 1, cat to reducer 0, horse to reducer 1, bill to reducer 3, and so forth.
- In reality, we will have P processes, each with R reducers.
    - $h \bmod P$ gives the process a word is sent to
    - $h \bmod R$ gives the reducer within process P that a word is sent to

word	$\boldsymbol{h}$	$\boldsymbol{h} \textbf{ mod } \mathbf{4}$
dog	5	1
cat	4	0
horse	5	1
bill	3	3
the	2	2
and	2	2
pool	4	0
pig	1	1
head	3	3
no	6	2
one	7	3
knows	7	3
trouble	0	0
I	6	2
have	1	1
seen	0	0

Mapping the words on the right

- Let’s assume we have 4 processes with 7 reducers each.
- Then “dog:” goes to process 1 and reducer 5 within process 1
- “bill” goes process 3 and reducer 3 within process 3
- “knows” goes to process 3 and reducer 7 within process 3

word	h	$h \bmod 4$
dog	5	1
cat	4	0
horse	5	1
bill	3	3
the	2	2
and	2	2
pool	4	0
pig	1	1
head	3	3
no	6	2
one	7	3
knows	7	3
trouble	0	0
I	6	2
have	1	1
seen	0	0

The reducer

dog	the	and	have
cat	pool	no	seen
horse	and	one	dog
bill	cat	knows	and
the	pig	the	bill
and	dog	trouble	pig
the	head	I	dog

- Reducers get words and count pairs from each mapper
- Reducers *reduce* the multiple same words from different mappers by adding the counts
- Each reducer then contains the final count of the words it is responsible for

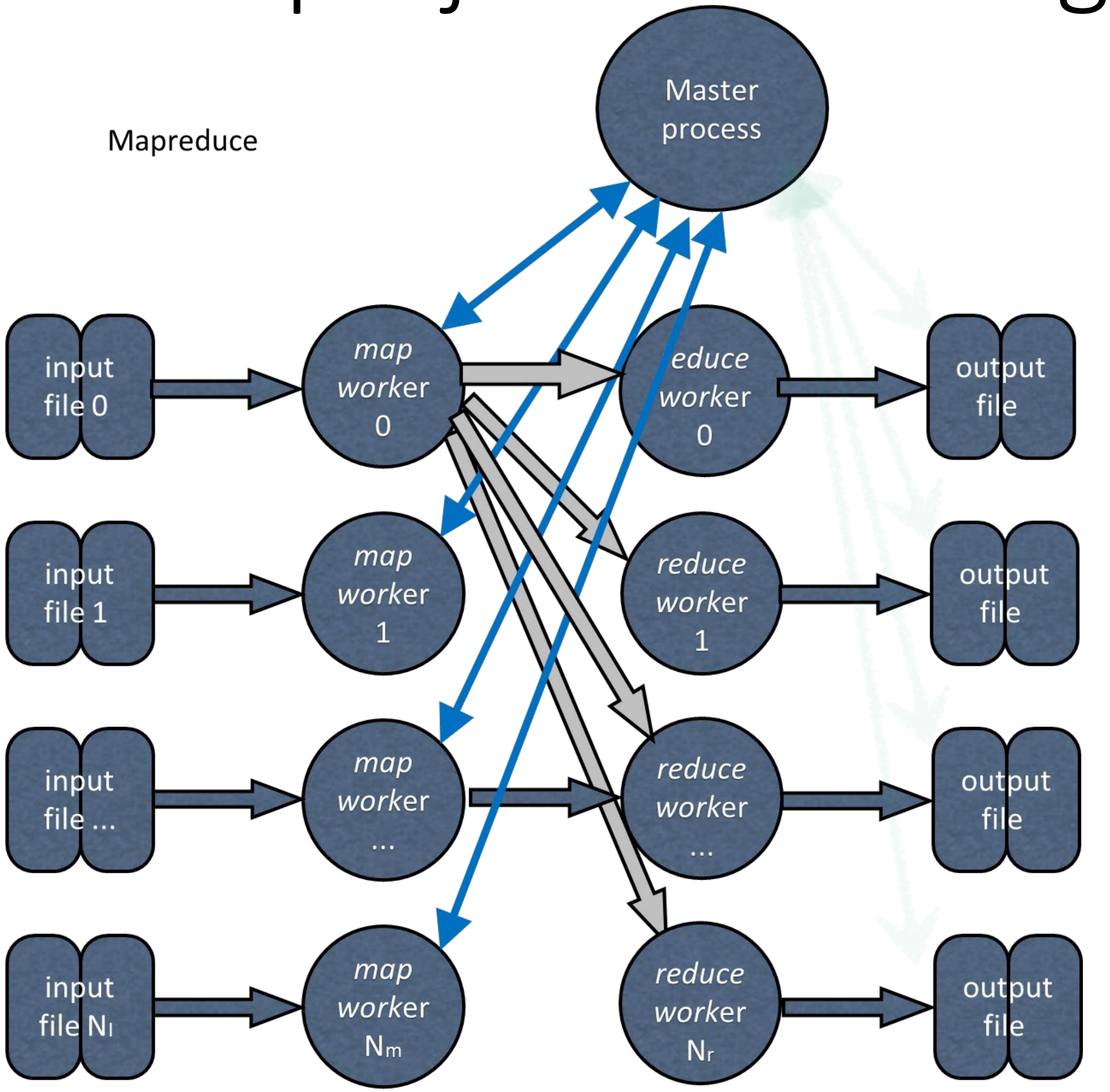
word	h	$h \bmod 4$	**count**
dog	5	1	4
cat	4	0	2
horse	5	1	1
bill	3	3	2
the	2	2	4
and	2	2	4
pool	4	0	1
pig	1	1	2
head	3	3	1
no	6	2	1
one	7	3	1
knows	7	3	1
trouble	0	0	1
I	6	2	1
have	1	1	1
seen	0	0	1

The project: Step 1

- Create a parallel map-reduce that runs as an OpenMP program
  - A master thread will maintain a list of files with words to count
    - *Reader* threads (see below) will request a file to read when it has no work to do
  - *Reader* thread(s) will read the files and put lines of the file into queues to be used by the *mapper* threads
  - *Mapper* threads will *combine* words and then put them onto input queues for *reducers*.
    - There will be a separate input queue for each reducer
  - At the end of the program, each reducer will write its list of words and counts into a file.
- The program is done when every reducer's input queue is empty, and every mapper is finished executing

The project from a high level

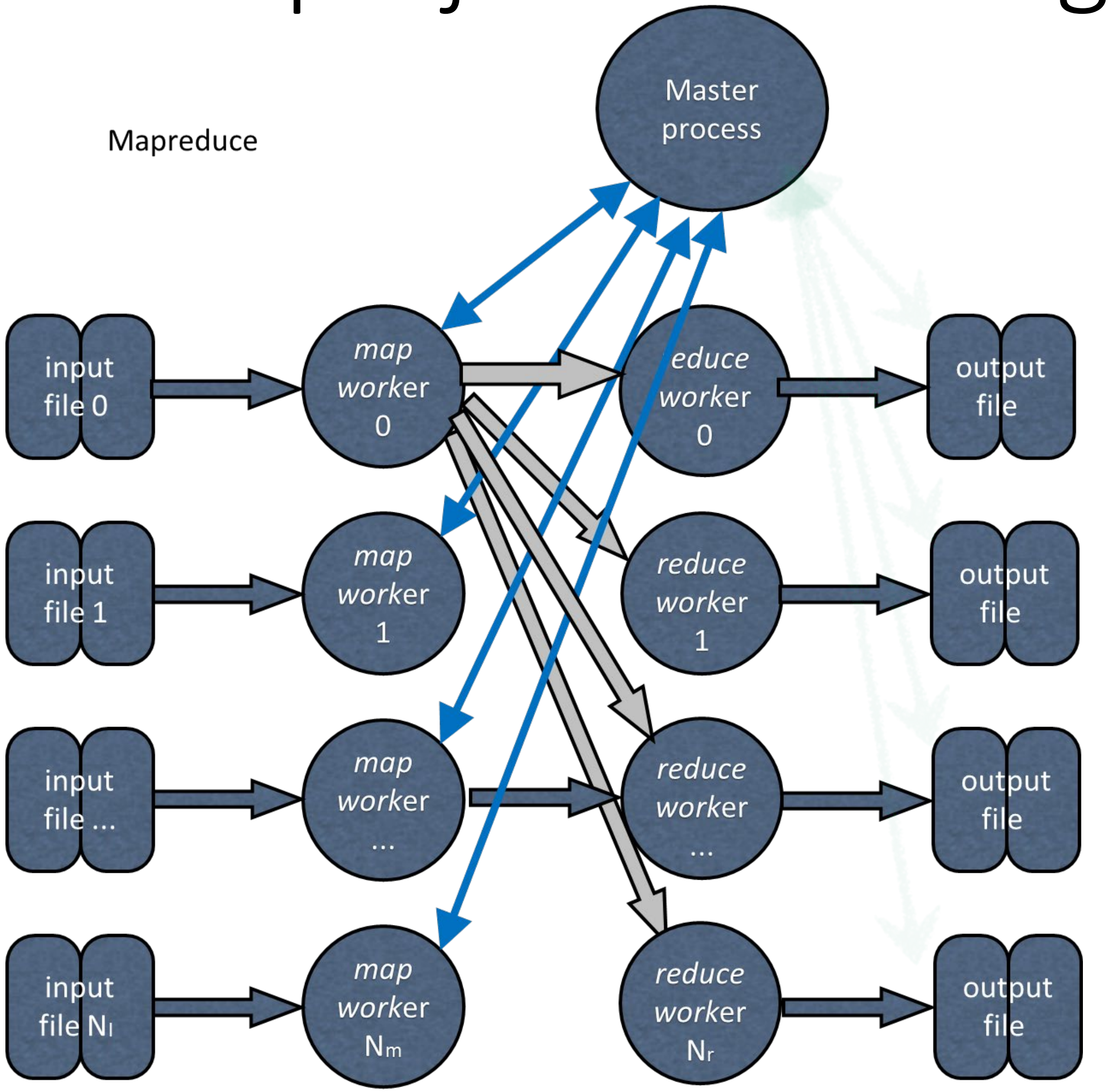
Mapreduce



The project, Step 2

- Write a hybrid MPI/OpenMP program
- Each MPI process runs on a node
- Each MPI process executes an OpenMP program
  - Mappers, reducers and readers are OpenMP threads
  - Mappers will map words to the correct process
    - Processes will map received words to the correct reducer within the process
    - MPI will work best if there is a single OpenMP thread that does all inter-process communication
    - The master thread is a good one to pick for this
- A master process will distribute files to processes to read and map
- The master process will let all processes know when all mappers in all processors have finished
- The master process should also do mapping and reducing

The project from a high level



- Let the reduce workers be processes that will do a reduction
- Then the structure for Step 2 is, at a high level, similar to the structure for Step 1.

Project Step 3

- A report based on Project Step 2. See the project document for details.