Document Classification

WWW

- WWW contains millions of text documents
- Many questions answered by retrieving the right documents
- But, automated search engines needed to find the documents most likely to contain relevant information
- Usually vectors are used to represent the "fit" between document and concept.

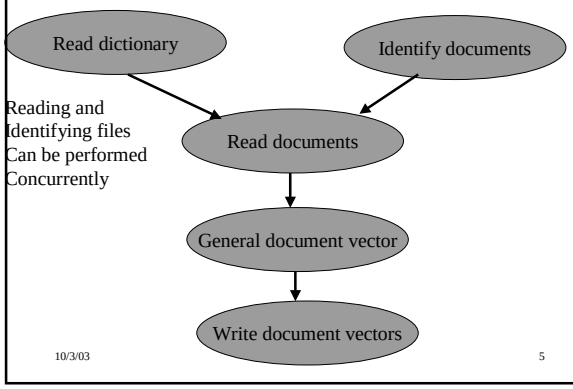
Application

- Reads a dictionary of key words
- Locates a set of text documents
  - A user given directory search
  - Files could be .html, ,tex ot .txt
- For each file:
  - Program opens the file
  - Reads the contents
  - Generates a profile vector→indicates how many times the text document contains each word appearing in the dictionary.
  - Writes the profile vectors in a file.

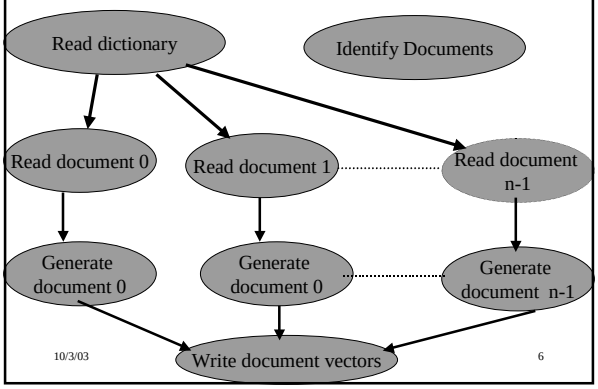
Problem

- Amenable to functional decomposition
- Develop manager/worker style parallel program to solve

Dependency graph



Partitioning and Communications



Agglomeration and Mapping

- Number of tasks not known at compile time
- Tasks do not communicate with each other
- The time to process each document may vary--.html files may be more difficult to process
- We will map processes at run time.

Manager/Worker paradigm

- Run-time allocation of processes
- One process, **manager**
  - Keeps track of assigned and unassigned tasks
  - Allocates tasks on demand to
- Other processes, Workers
  - Send results back to the document
- Termination condition:
  - When no work documents available in the manager

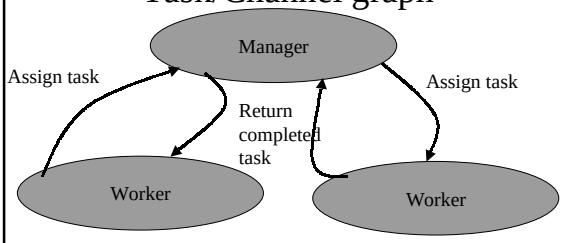
Manager/Worker style

- Advantage: load balancing
- Disadvantage:
  - Additional communication overhead if tasks are allocated one at a time
  - Increases execution and decreases speedup
- Moves away from SPMD model
  - Manager executes one function while the workers execute another
  - Responsibilities are different

Which tasks will be done by manager and which by workers

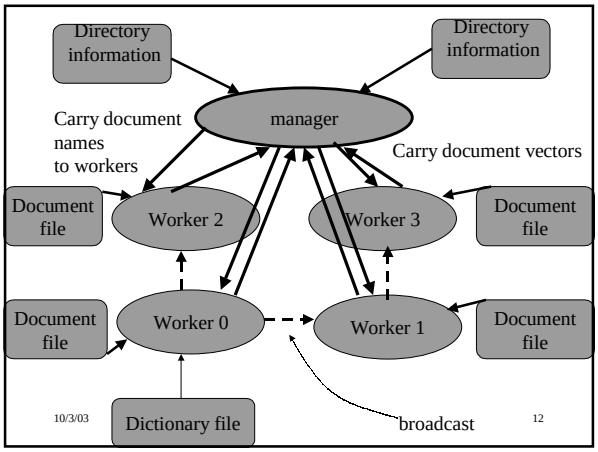
- Manager
  - Identify documents
    - Assign file names to workers
  - Gather the document vectors and write the resulting files
- Worker
  - Reading dictionary
    - Will construct the profile vectors
  - Read the document given the filename
  - Produce document profile vector

Task/Channel graph



Either manager can assign tasks by asking workers if they one
Or worker sends a message to manager when it needs one

We decide to have the worker start the message – we do not know when the MPI processes will finish.



Manager process Algorithm

- *a: array showing document assigned to each process*
- *d: documents assigned*
- *j: id of worker requesting document*
- *k: document vector length*
- *n: number of documents*
- *p: total number of processes*
- *s: storage area containing document vectors*
- *t: terminated workers*
- *v: individual document vector*

Manager process Algorithm

- Identify n documents from user specified directory
- Receive dictionary of size k from worker 0
- Allocate s (nxk dimensions) to store document vectors
- d← 0 (no documents assigned)
- t← 0 (no workers terminated)

Manager process Algorithm

```
Repeat (/* until all workers terminated*/)
    Receive message from worker j
    If message contains document vector v store the vector in the
       appropriate place in s
    Else
        Message is first request for work (do nothing)
    If there are any documents left (d < n)
        Send name of document d to worker j
        Record in array a which document is assigned
        Increment d, the number of documents
    Else
        No more work—sends termination message to worker j
        Increments termination count, t.
Until t = p-1 (terminated all workers)
Write s to output file
```


MPI_Abort

- Manager needs to set up matrix while other processes doing other things
- If memory allocation problem, terminate execution of MPI program
- int MPI_Abort(MPI_Comm comm, int error_code)
  - Aborts the processes in the communicator *comm*.
  - Returns *error_code* to calling environment

Worker Processes algorithm

- Every worker needs a copy of the dictionary
- Solution 1:
  - One worker reads by opening file and broadcasts to others
- Solution 2:
  - Each worker opens the dictionary file and reads
- *f: filename*
- *k: dictionary size*
- *v: document vector*

Worker Processes Algorithm

```
First send the request for work to manager
    /* while this is sent out, the worker can go with the dictionary setup.
       Overlapping computation with communication*/
    If worker 0 then
        Read dictionary form file
        Broadcast dictionary
        Each worker constructs a hash table from dictionary elements
        If worker 0 then
            Send dictionary size k to manager
    Repeat
        Receive filename,f, from manager
        If filename, f, indicates termination then exit.
        Else
            Read document from file f
            Generate document vector v
            Send v  to manager.
    forever
```


Partitioning Processes

- MPI_COMM_SPLIT:
  - partitions the processes into disjoint sets
  - Each group executes a different program
  - These processes communicate amongst themselves without fear of conflict with other concurrent computations
  - This communicator supports parallel composition
- MPI_COMM_SPLIT(comm,color,key,newcomm)

Partitioning Processes

- MPI_COMM_SPLIT(comm,0,0,newcomm)
  - All processes have the same color and key.

Example

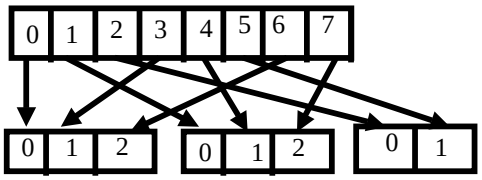
```
MPI_COMM comm,newcomm;
int myid, color;
MPI_COMM_RANK(comm,&myid)
color = myid % 3;
MPI_COMM_SPLIT(comm,color,
                        myid,&newcomm);
```

Comm contains 8 processes:

processes 0,3,6 form a new communicator of size 3

processes 1,4,7 form a new communicator of size 3

processes 2,5 form a new communicator of size 2

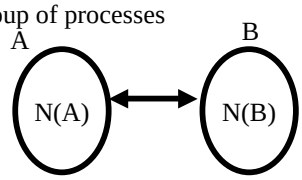


Example

```
MPI_COMM comm,newcomm;
int myid, color;
MPI_COMM_RANK(comm,&myid)
If (myid < 8) /* Select first 8 processes */
        color = 1
Else
        color = MPI_UNDEFINED /*others not in group */
MPI_COMM_SPLIT(comm,color,myid,&newcomm)
```


Communicating between processes

- MPI_COMM_SPLIT can be used to communicate within a group of processes. It is called an intracommunicator
- Intercommunicator: communicate between the group of processes



Creating workers only communicator

- int id;
- MPI_Comm worker_comm;
- If (!id) /*Manager*/
  MPI_Comm_split(MPI_COMM_WORLD,MPI_UNDEFINED,id,&worker_comm);
- else
  MPI_Comm_split(MPI_COMM_WORLD,0,id,&worker_comm)