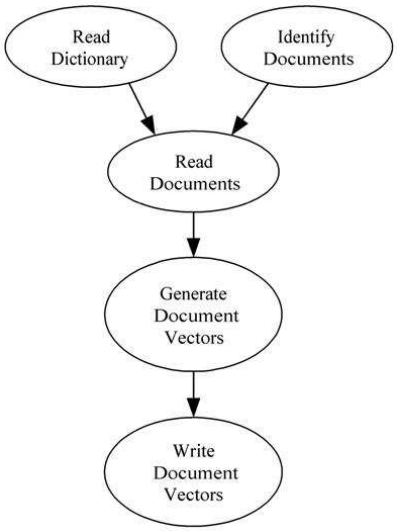
Document Classification

Introduction

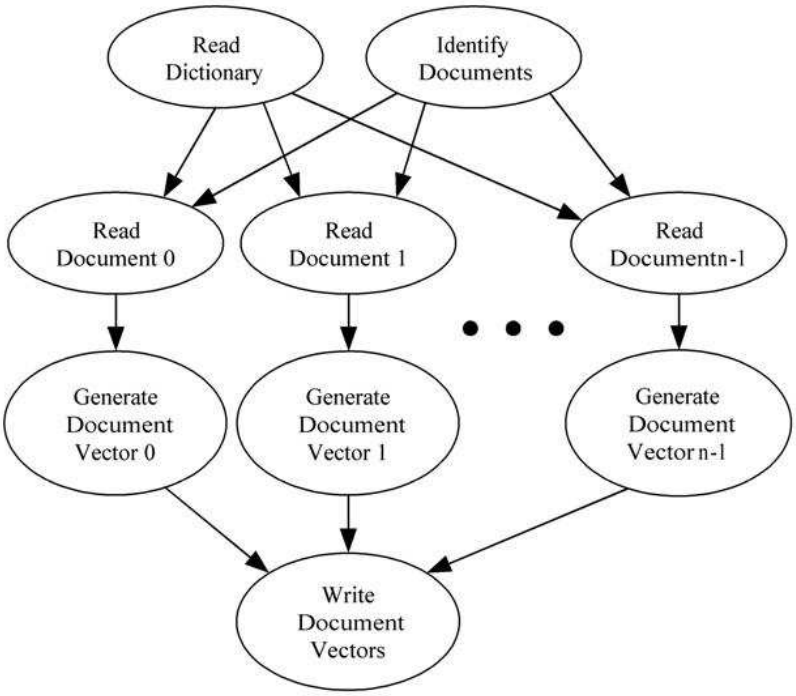
- Search engine on web
  - Search directories, subdirectories for documents
  - Search for documents with extensions `.html`, `.txt`, and `.tex`
  - Using a dictionary of key words, create a profile vector for each document
  - Store profile vectors
- Problem to be solved with manager/worker style parallel program

Parallel algorithm design

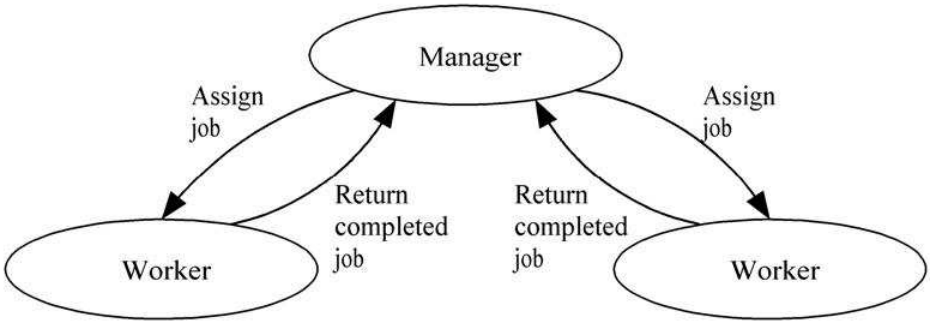
- Profile vector
  - Commonly used words eliminated by a *stop list*
  - Words stripped down to their roots
    - Store, Storage, Storing, Stored – All forms of the same word *store*
  - Weight assigned to each keyword in the document
  - Term frequency
  - Inverse document frequency
  - Data dependence diagram



- Partitioning and Communication
  - Most time spent on reading documents and generating profile vectors
  - Two primitive tasks for each document
    1. Read the document file
    2. Generate the vector

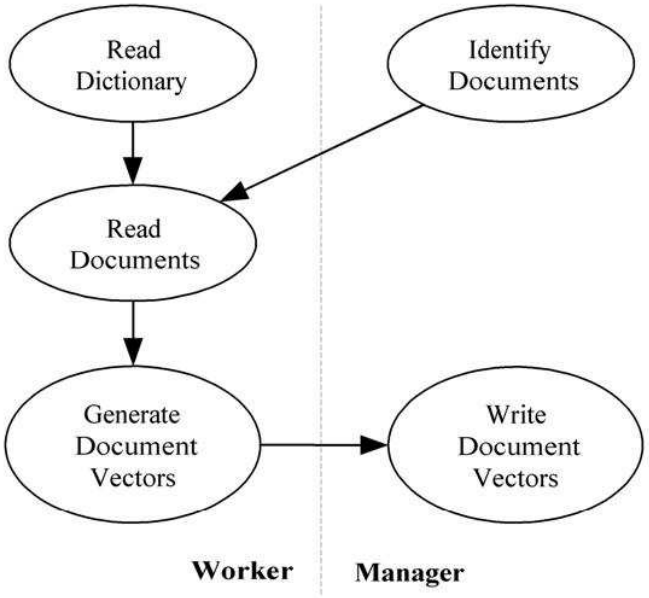


- Agglomeration and mapping
  - Number of tasks not known at compile time
  - Tasks do not communicate with each other
  - Time needed to perform each task may vary widely
    - Different size of documents
    - Files may contain markup (`.html` and `.tex` files compared to `.txt` files)
  - Strategy: map tasks to processes at run time
- Manager/worker paradigm
  - Can also be viewed as domain partitioning with run-time allocation of data to tasks



  - Manager
    - Responsible to keep track of assigned and unassigned data
    - Assigns tasks to workers and retrieves results back from them
  - Allocating only a single task at a time to each worker balances workload
  - Worker
    - Done when it completes a task and manager has no more tasks to assign

    - No worker has more than one task to complete
  - Disadvantage of allocating a single task
    - Introduces additional communication overhead
    - Increases execution time
    - Lowers speedup
  - SPMD Style
    - Single Program Multiple Data
    - Almost all the processes we have worked on so far
    - Every process executes the same functions
      - A designated process may be responsible for file or user I/O
  - Manager/worker paradigm
    - Does not follow SPMD style
      - Manager process has different responsibilities than worker processes



    - MPI manager/worker system splits control flow early on
      - Manager and workers perform completely different functions
    - Helps in balancing workload for high efficiency
    - Decide the tasks to be performed by the manager and workers
      - Reading dictionary should be a job for the workers
      - Worker reads a document file and creates the profile vector
      - Manager responsible for gathering the documents vectors and writing the results file
    - Interaction cycle between the manager and each worker
      - Manager provides a worker with a task
      - Worker does the job and sends the report (completed task) to manager
      - The cycle repeats
    - Start with the worker who sends a message to manager that he is ready for some work
      - Makes sure that manager only sends tasks to workers it knows are active

- Manager process
  - Identifies n plain text documents

- Receives the document vector size k from process 0 to allocate space for $n \times k$ matrix s to store vectors received from workers
- Initializes variables d and t to show that no documents have been assigned and no workers have been terminated, respectively
- Pseudocode for manager

```
a -- Array showing documents assigned to each process
k -- Document vector length
n -- Number of documents
s -- Storage array containing document vectors

Identify n documents in user-specified directory
Receive dictionary size k from worker 0
Allocate nxk matrix s to store document vectors
d = 0    // Documents assigned
t = 0    // Terminated workers

// Repeat loop until all workers are terminated

do
    Receive message from worker j
    if message contains document vector v
        s[a[j]] = v         // Store document vector in s
    // else worker is indicating that it is ready for a document
    // Only happens once per worker

    if d < n                // more documents?
        Send file name d to worker j
        a[j] = d            // Record in a the document assigned
        d++
    else
        Send worker termination message
        t++
while workers present
Write document vectors s to file
```

- Function `MPI_Abort`
  - A quick and dirty way for one process to terminate all processes in specified communicator
  - Manager allocates space for an $n \times k$ matrix to store document vectors
  - In case of allocation failure, the code should be terminated
  - `MPI_Abort` makes a *best effort* attempt to terminate all processes in the specified communicator

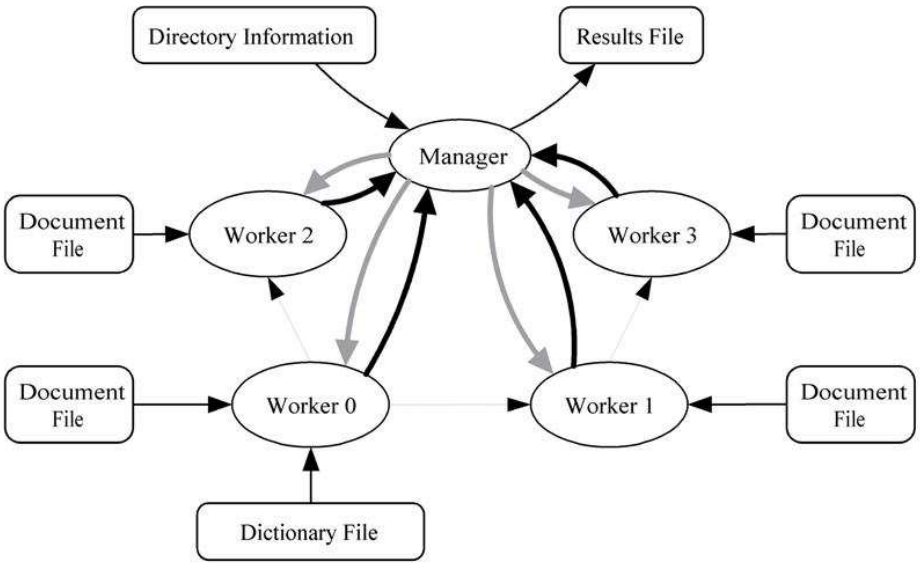
```
int MPI_Abort (
    IN MPI_Comm comm,    // Communicator of tasks to abort
    IN int error_code    // Error code to return to invoking environment
    );
```

- Worker process
  - Worker needs to read dictionary using one of the following two options
    1. Each worker opens dictionary file and reads it from filesystem
    2. One worker reads the dictionary and broadcasts it to other workers
       - Better option if broadcast bandwidth inside the parallel computer is greater than bandwidth between parallel computer and file server

- Pseudocode for worker

```
Send first request for work to manager
if worker 0
    Read dictionary from file

Broadcast dictionary among workers
Build hash table from dictionary
if worker 0
    Send dictionary size k to manager

// Reading/sharing dictionary overlaps communication with manager

do
    Receive file name f from manager
    if file_name == NULL
        terminate

    Read document from file f
    Generate document vector v
    Send document vector v to manager
while ( 1 )
```

- Task/channel graph



- Creating a worker-only communicator
  - Dictionary is broadcast among workers while manager searches native directory structure for document files
  - Use `MPI_Bcast` for collective broadcast
  - Need workers-only communicator for workers-only broadcast
  - Use `MPI_Comm_split` to split the current communicator into manager and workers
  - Manager passes `MPI_UNDEFINED` as the value of `color`, meaning it will not be part of any new communicator
    - Return value of `new_comm` will be `MPI_COMM_NULL` for manager
  - Usage example

```
int          id;
MPI_Comm     worker_comm;
...
if ( ! id )             // Manager
    MPI_Comm_split ( MPI_COMM_WORLD, MPI_UNDEFINED, id, &worker_comm );
else                    // Worker
    MPI_Comm_split ( MPI_COMM_WORLD, 0, id, &worker_comm );
```

Nonblocking communications

- Three phases of manager process
  1. Find plain text files; receive dictionary size from worker 0, allocate space (2D array) for profile vectors
  2. Allocate documents to workers and collect profile vectors
  3. Write the complete set of profile vectors to a file
- Phase I
  - Manager must search directory and receive a message from worker 0
  - Need to overlap the two activities
  - `MPI_Send` and `MPI_Recv` are blocking operations
    - `MPI_Send` does not return until message is copied to a system buffer or the message has been sent
    - `MPI_Recv` does not return until message has been copied into buffer specified by user
    - Blocking send and receive may limit the performance of a parallel program
  - Operations can be *initiated* by `MPI_Isend` and `MPI_Irecv`
  - `MPI_Wait` blocks until the operation is complete
  - Calls can be made early
    - `MPI_Isend` as soon as value(s) assigned
    - `MPI_Irecv` as soon as buffer available
  - Can eliminate a message copying step
  - Allows communication/computation overlap
- Manager's communication
  - Manager needs to receive the dictionary size from worker even though it does not use it until after it has identified document files to be processed
- Function `MPI_Irecv`
  - Begins a nonblocking receive

```
int MPI_Irecv (
    void        * bufr,     // Address of receive buffer
    int           count,    // Number of elements in receive buffer
    MPI_Datatype  datatype, // Datatype of each element in receive buffer
    int           src,      // Rank of source process
    int           tag,      // Message tag
    MPI_Comm      comm,     // Communicator
    MPI_Request * request   // Communication request
    );
```

  - The first six parameters are the same as `MPI_Recv`

  - Last parameter (`request`) *returns* a pointer to an `MPI_Request` object to identify the communication operation that is initiated
  - Only initiates receive; you cannot access `\bufr` until a matching call to `MPI_Wait` has returned
  - Does not return a pointer to an `MPI_Status` object because receive is not yet completed

- Function `MPI_Wait`
  - Wait for an MPI request to complete

```
int MPI_Wait (
    MPI_Request * request,  // Communication request
    MPI_Status  * status    // Status object; may be MPI_STATUS_IGNORE
    );
```

  - Function blocks until the operation associated with `request` completes
  - For send operation, buffer may be assigned a new value
  - For receive operation, buffer may be referenced; `status` points to object containing information about received message

- Workers' communications
  - Each worker initially notifies the manager that it is active
  - Send a notification to manager and proceed to read/broadcast of dictionary and build hash table
  - Worker receives file name from manager
    - File names may be deeply nested in the directory structure and worker may not know the number of characters in incoming filename
    - How to handle it?
      - Allocate a huge buffer
      - Check the length of incoming message and then, allocate buffer

- Function `MPI_Isend`
  - Initiate a nonblocking send

```
int MPI_Isend (
    IN  void         * bufr,     // Address of send buffer
    IN  int            count,    // Number of elements in send buffer
    IN  MPI_Datatype   datatype, // Datatype of each element in send buffer
    IN  int            dest,     // Rank of destination process
    IN  int            tag,      // Message tag
    IN  MPI_Comm       comm,     // Communicator
    OUT MPI_Request  * request   // Communication request
    );
```

    - The parameter `request` returns a handle to an object created by run-time system to identify communication request
    - The message buffer may not be reused until the matching call to `MPI_Wait` has returned

- Function `MPI_Probe`
  - Blocking test for a message
  - Blocks until a message matching the source and tag specifications is available to be received

```
int MPI_Probe (
    IN  int          src,   // Source rank, or MPI_ANY_SOURCE
    IN  int          tag,   // Tag value, or MPI_ANY_TAG
    IN  MPI_Comm     comm,  // Communicator
    OUT MPI_Status * status // Status object
    );
```

    - `status` returns information about the source, tag, and length of message
  - Blocks until the message matching the source and tag specifications is available to be received
  - *Does not actually receive message*
  - `MPI_ANY_SOURCE` allows you to probe for a message from any other process
  - `MPI_ANY_TAG` allows you to probe for any message from the process specified as `src`
  - Best to keep source and tag specification as narrow as possible
    - Minimize mismatch bugs that occur when messages arrive in unexpected order
  - In the current problem, worker knows both source and tag of message expected from manager
- Function `MPI_Get_count`
  - Get the number of *top level* elements, or number of elements in message

```
int MPI_Get_count (
    IN  MPI_Status   * status,    // Return status of receive operation
    IN  MPI_Datatype   datatype,  // Datatype of each element in receive buffer
    OUT int          * count      // Number of received elements
    );
```

    - If the size of datatype is 0, `count` is 0
    - If the amount of data in `status` is not an exact multiple of datatype size, `count` is `MPI_UNDEFINED`

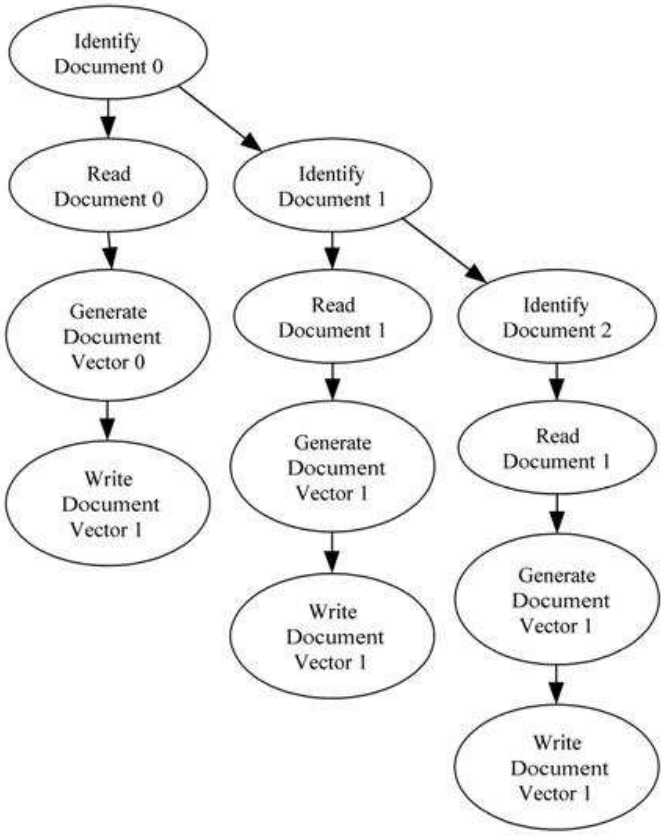
Parallel Program

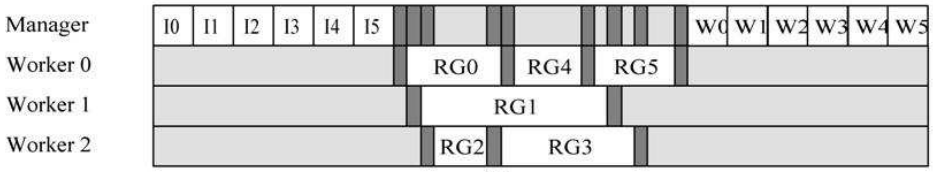
- Four types of messages being sent and received: dictionary size, file name, profile, empty
  - Allows a process to receive messages from another process in a different order than they were sent
  - Worker 0 sends an initial request to manager for work
  - After it has read, broadcast, and processed the dictionary, it sends dictionary size to manager
  - Manager needs to to allocate the document vector profile storage area before handling requests for work from workers
    - Must know dictionary size from worker 0 before it receives the initial request for work from worker 0
  - Two messages get different tags, enabling out-of-order reception

Enhancements

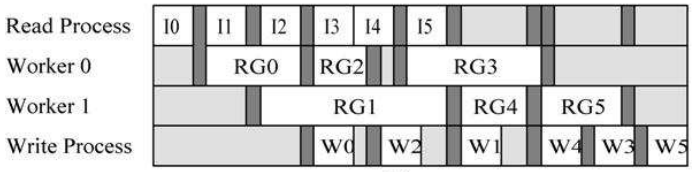
- Assigning groups of documents
  - Preallocation of data to processes may result in imbalanced workload
  - We allocated data to processes at run time to balance workload
  - Our process introduced additional communications overhead, lowering speedup
  - Middle ground between pre-allocation and one-at-a-time allocation
  - Assign k documents at a time to workers
- Pipelining
  - We identify all documents before we start processing any of them
    - No document files are processed until the manager has identified all of them
    - If the time to perform these tasks is not negligible, our design will not scale well to a larger number of processes

– Detriment to scaling over a number of processors





(a)



(b)