Parallel Programming
in C with MPI and OpenMP

Michael J. Quinn

Chapter 9

Document Classification

Chapter Objectives

- Complete introduction of MPI functions
- Show how to implement manager-worker programs

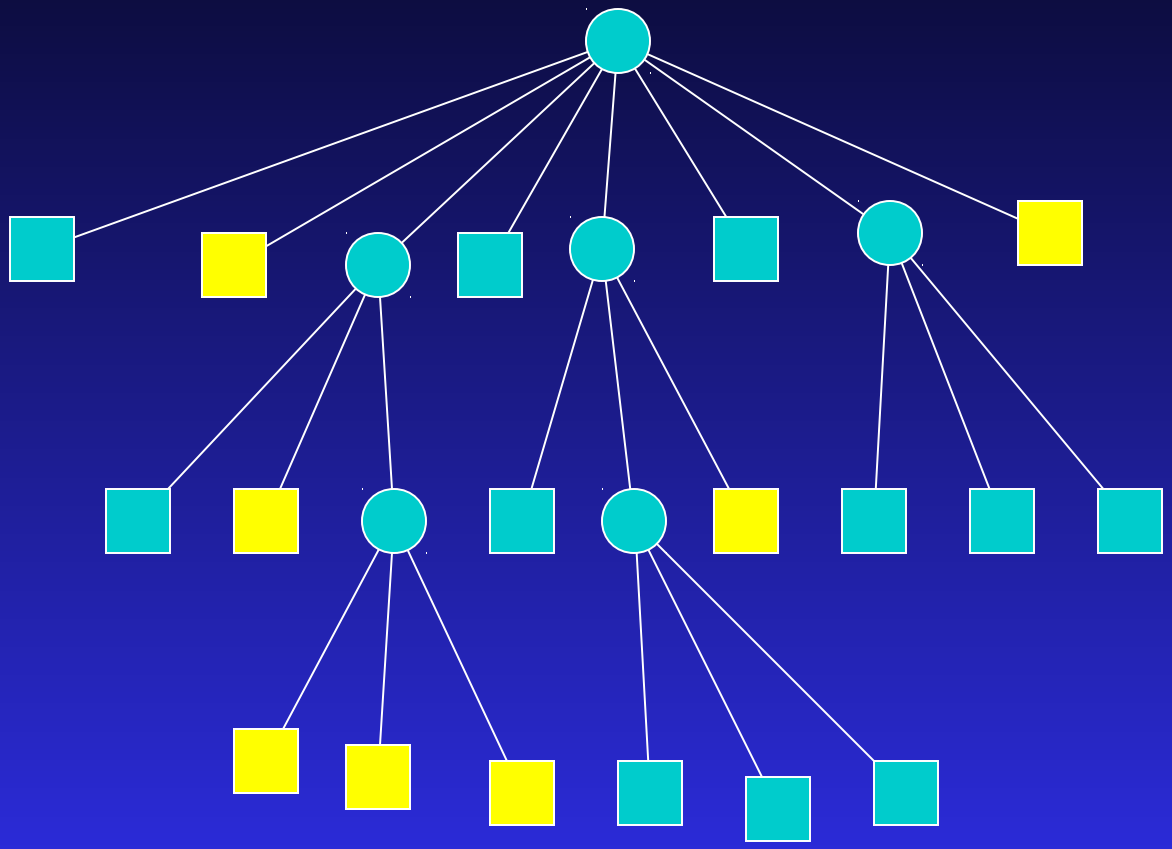
Outline

- Introduce problem
- Parallel algorithm design
- Creating communicators
- Non-blocking communications
- Implementation
- Pipelining

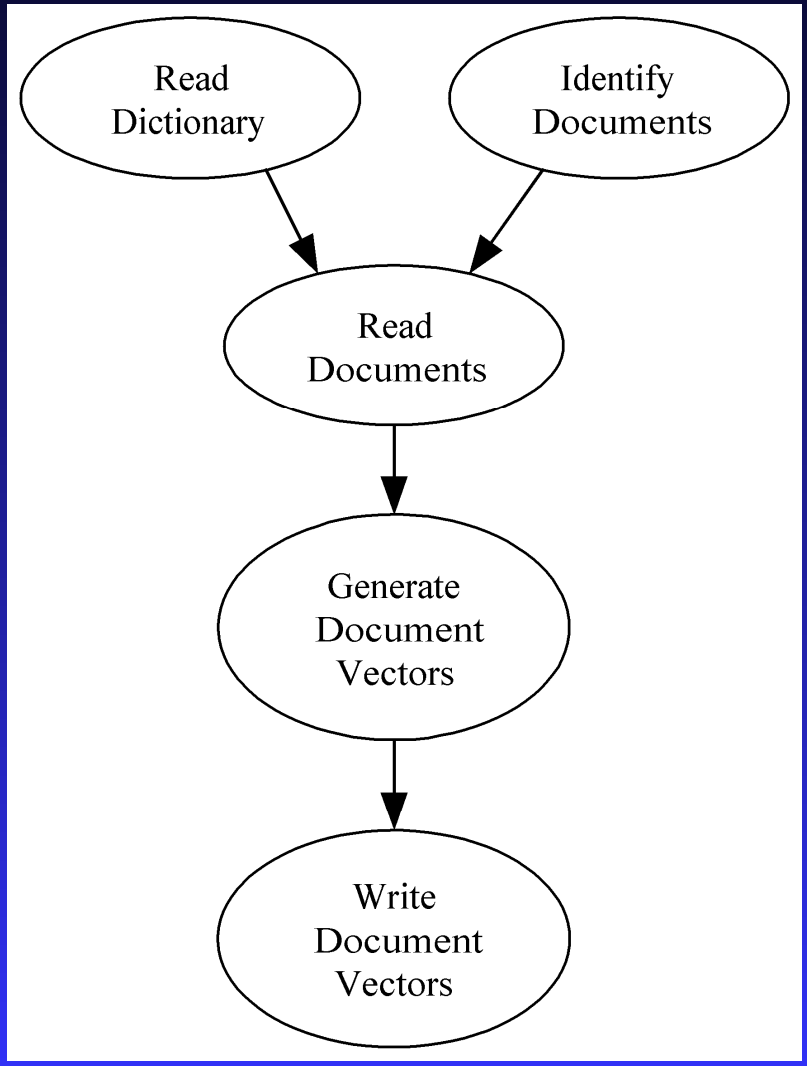
Document Classification Problem

- Search directories, subdirectories for documents (look for .html, .txt, .tex, etc.)
- Using a dictionary of key words, create a profile vector for each document
- Store profile vectors

Document Classification Problem



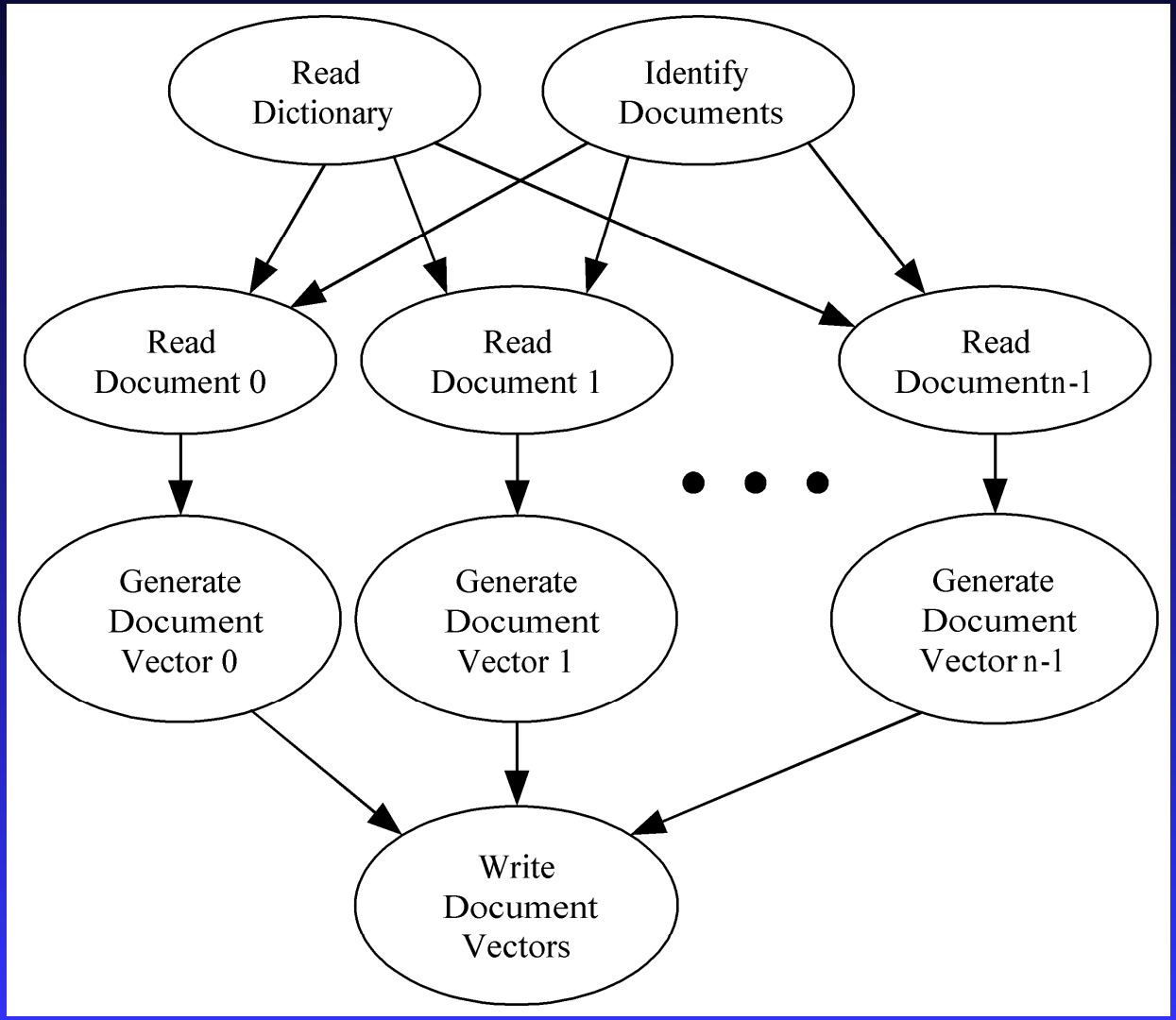
Data Dependence Graph (1)



Partitioning and Communication

- Most time spent reading documents and generating profile vectors
- Create two primitive tasks for each document

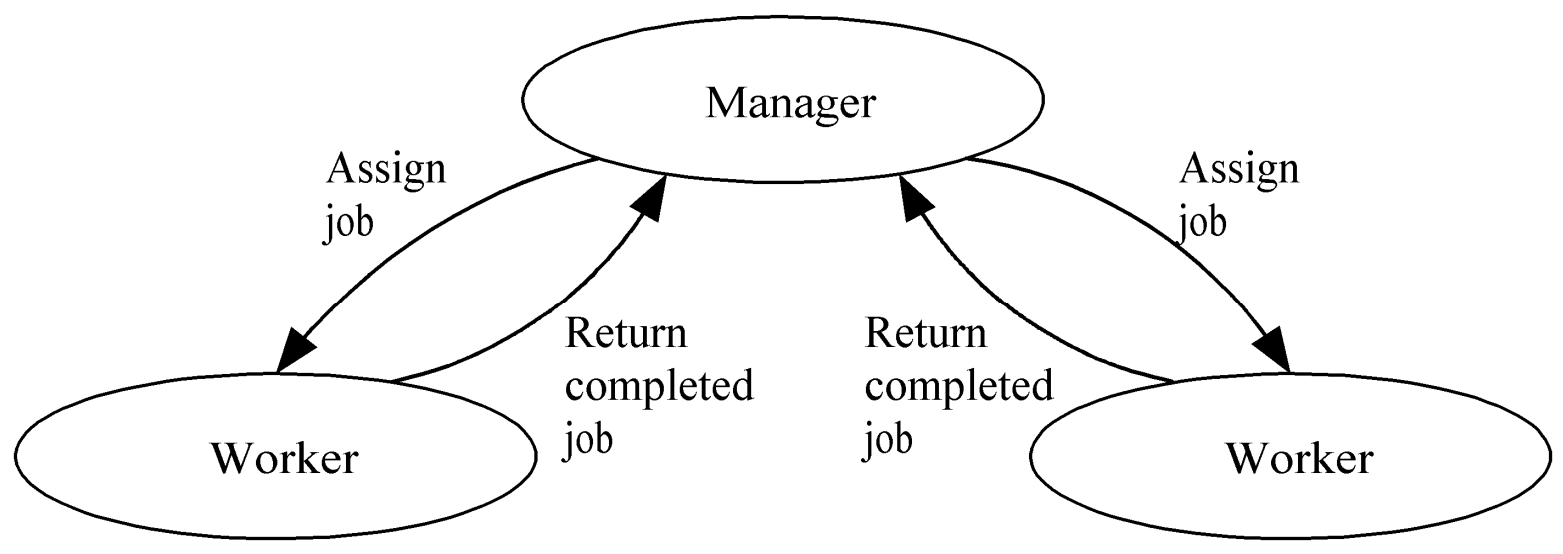
Data Dependence Graph (2)



Agglomeration and Mapping

- Number of tasks not known at compile time
- Tasks do not communicate with each other
- Time needed to perform tasks varies widely
- Strategy: map tasks to processes at run time

Manager/worker-style Algorithm

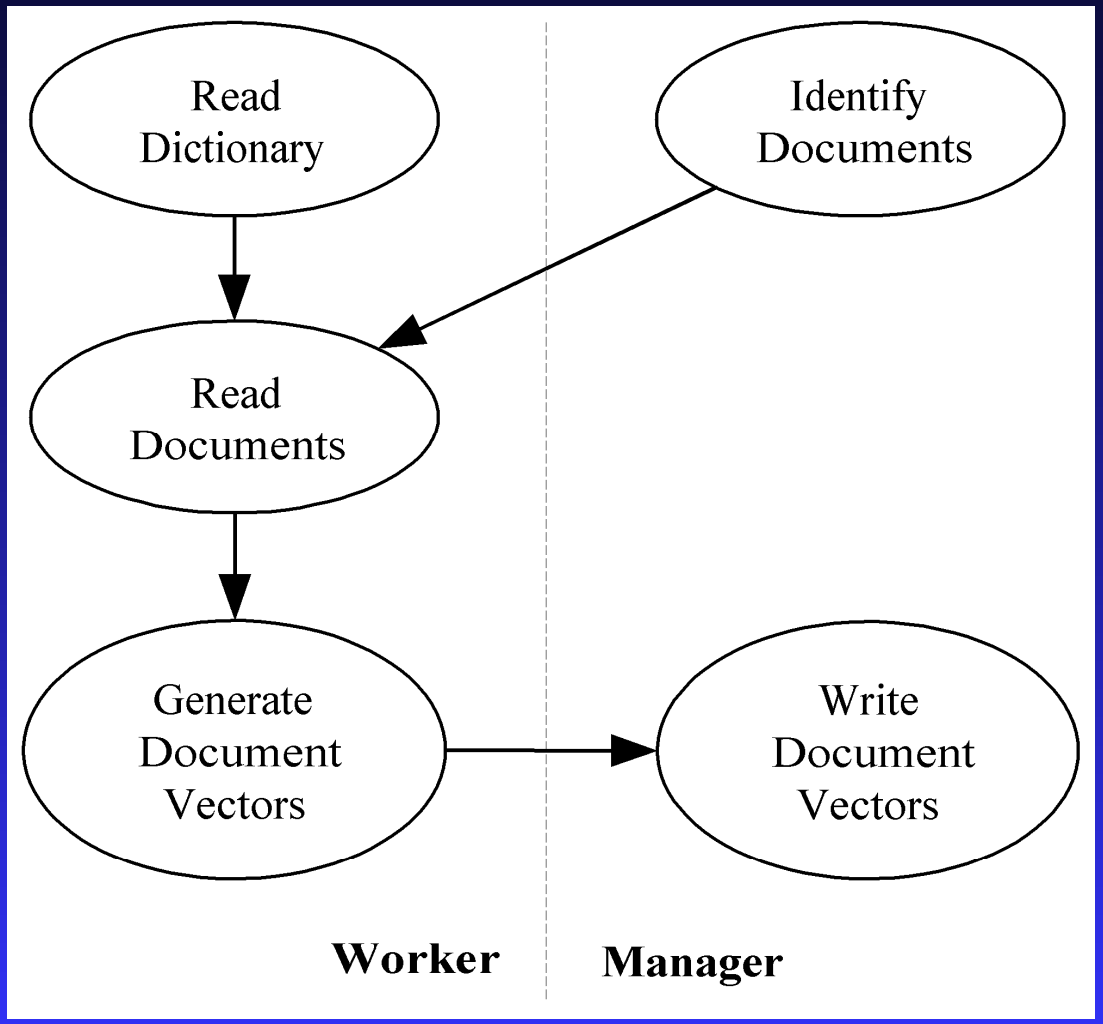


Can also be viewed as domain partitioning with run-time allocation of data to tasks

Manager/Worker vs. SPMD

- SPMD (single program multiple data)
  - Every process executes same functions
  - Our prior programs fit this mold
- Manager/worker
  - Manager process has different responsibilities than worker processes
  - An MPI manager/worker program has an early control flow split (manager process one way, worker processes the other way)

Roles of Manager and Workers



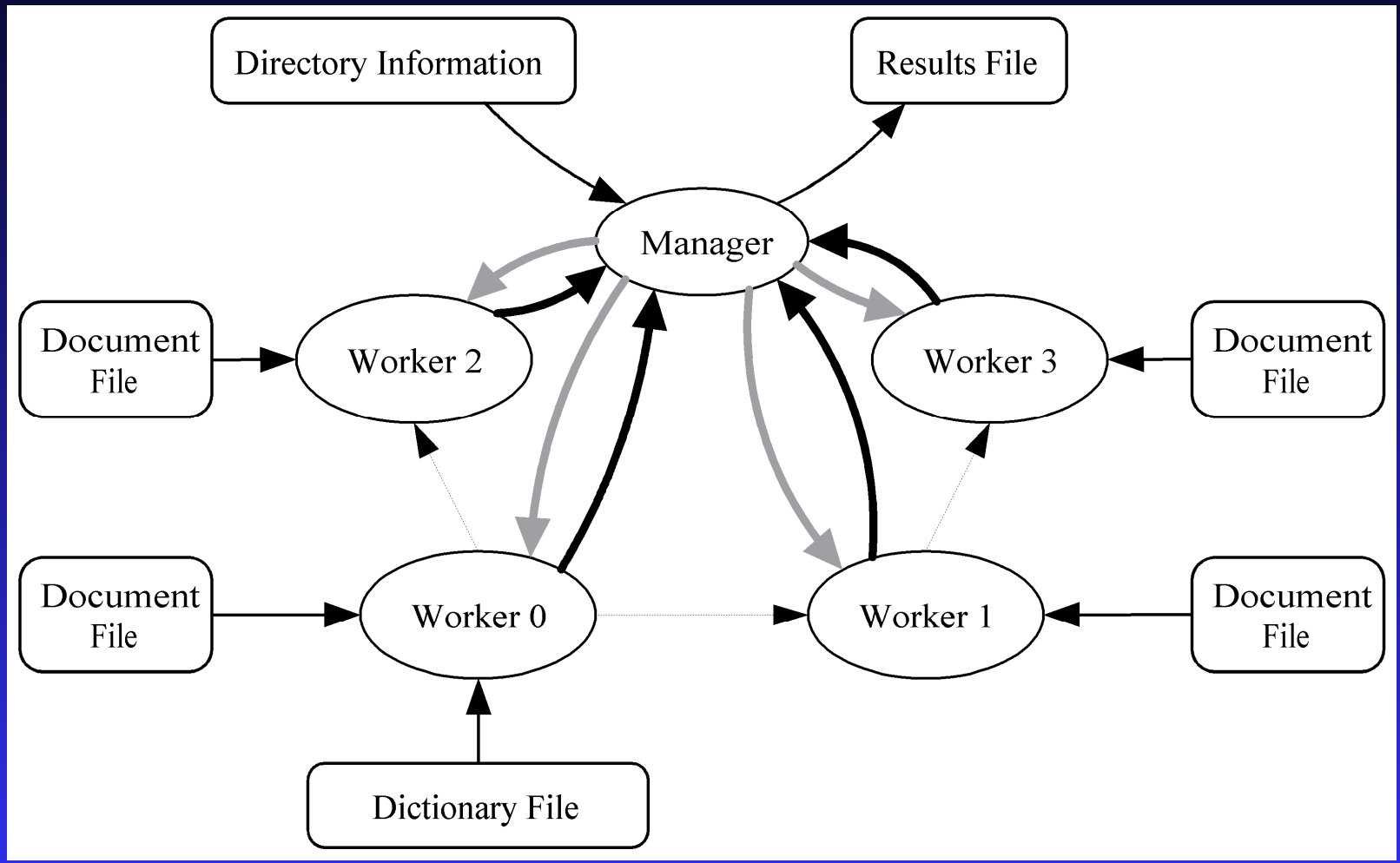
Manager Pseudocode

```
Identify documents
Receive dictionary size from worker 0
Allocate matrix to store document vectors
repeat
        Receive message from worker
        if message contains document vector
                Store document vector
        endif
        if documents remain then Send worker file
name
        else Send worker termination message
        endif
until all workers terminated
Write document vectors to file
```

Worker Pseudocode

```
Send first request for work to manager
if worker 0 then
        Read dictionary from file
endif
Broadcast dictionary among workers
Build hash table from dictionary
if worker 0 then
        Send dictionary size to manager
endif
repeat
        Receive file name from manager
        if file name is NULL then terminate endif
        Read document, generate document vector
        Send document vector to manager
forever
```

Task/Channel Graph



MPI_Abort

- A “quick and dirty” way for one process to terminate all processes in a specified communicator
- Example use: If manager cannot allocate memory needed to store document profile vectors

Header for MPI_Abort

```
int MPI_Abort (

    MPI_Comm comm,   /* Communicator */

    int error_code)  /* Value returned to
                        calling environment */
```

Creating a Workers-only Communicator

- Dictionary is broadcast among workers
- To support workers-only broadcast, need workers-only communicator
- Can use MPI_Comm_split
- Manager passes MPI_UNDEFINED as the value of split_key, meaning it will not be part of any new communicator

Workers-only Communicator

```
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 Send / Receive

- MPI_Isend, MPI_Irecv initiate operation
- MPI_Wait blocks until operation 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

Function MPI_Irecv

```
int MPI_Irecv (
      void           *buffer,
      int             cnt,
      MPI_Datatype    dtype,
      int             src,
      int             tag,
      MPI_Comm        comm,
      MPI_Request    *handle
)
```

Pointer to object that identifies communication operation

Function MPI_Wait

```
int MPI_Wait (
        MPI_Request *handle,
        MPI_Status *status
)
```

Function MPI_Isend

```
int MPI_Isend (
      void           *buffer,
      int             cnt,
      MPI_Datatype    dtype,
      int             dest,
      int             tag,
      MPI_Comm        comm,
      MPI_Request    *handle
)
```

Pointer to object that identifies communication operation

Receiving Path Name

- Worker does not know length of longest path name it will receive
- Alternatives
  - Allocate huge buffer
  - Check length of incoming message, then allocate buffer
- We'll take the second alternative

Blocks until message is available to be received

Function MPI_Probe

```
int MPI_Probe (
      int          src,
      int          tag,
      MPI_Comm     comm,
      MPI_Status *status
)
```

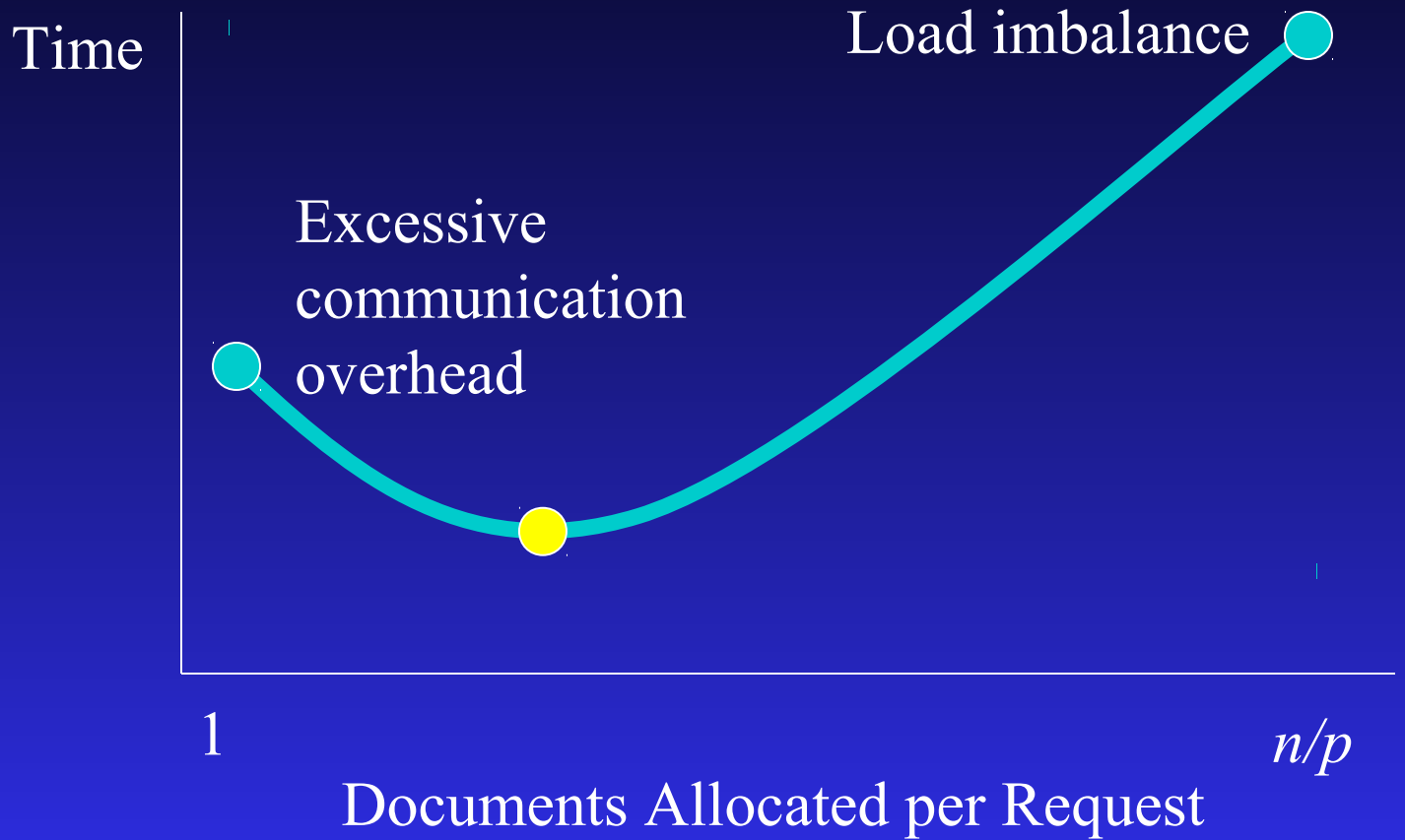
Function MPI_Get_count

```
int MPI_Get_count (
      MPI_Status *status,
      MPI_Datatype dtype,
      int *cnt
)
```

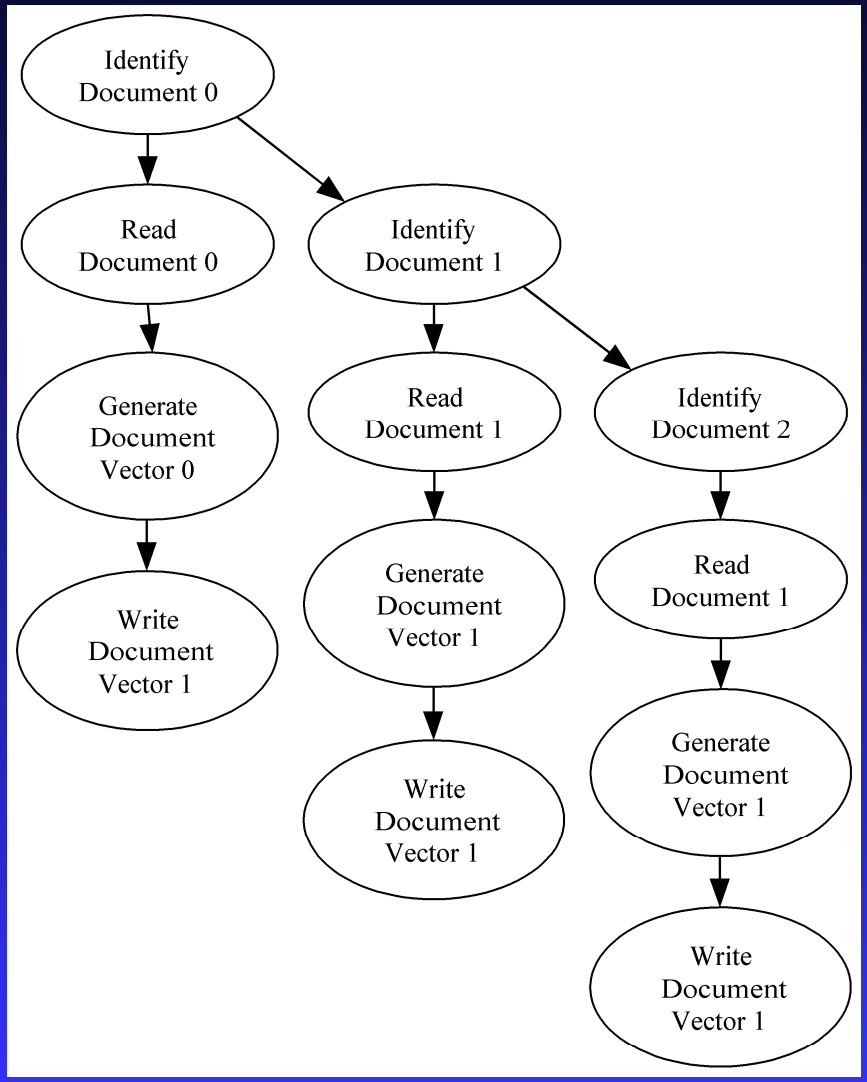
Enhancements

- Middle ground between pre-allocation and one-at-a-time allocation
- Pipelining of document processing

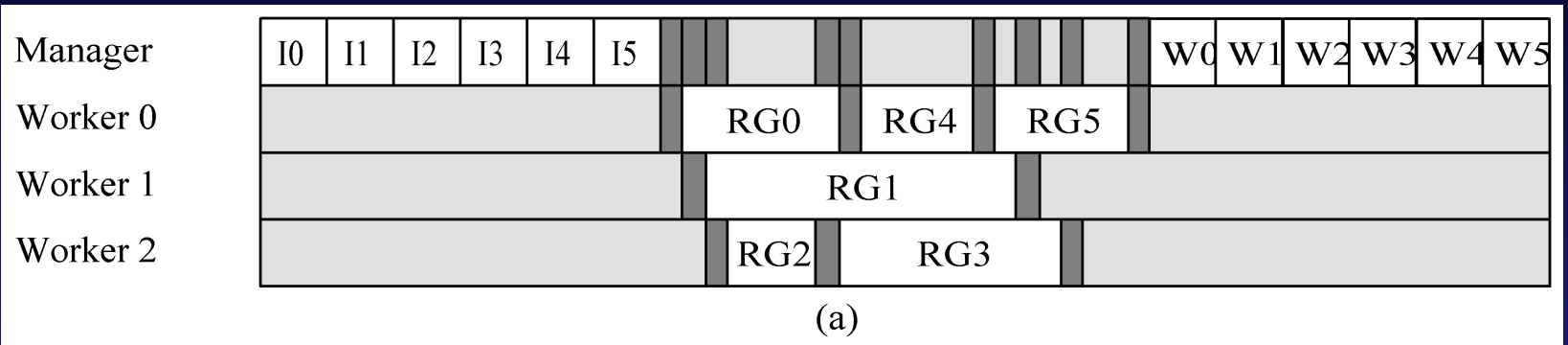
Allocation Alternatives



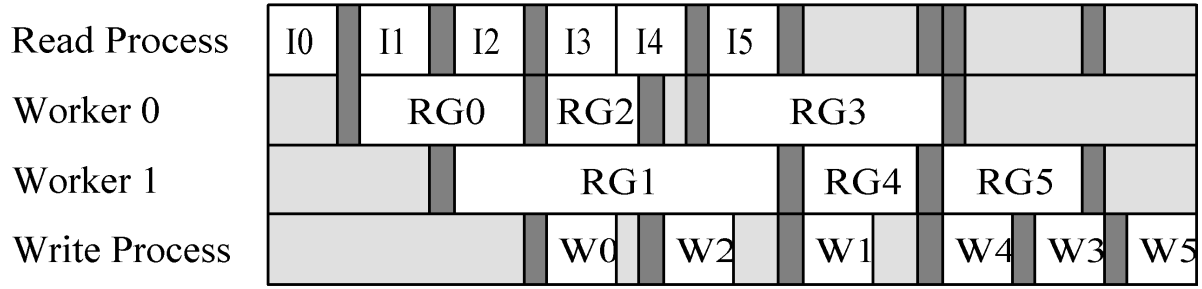
Pipelining



Time Savings through Pipelining



(a)



(b)

Pipelined Manager Pseudocode

```
a ← 0  {assigned jobs}
j ← 0  {available jobs}
w ← 0  {workers waiting for assignment}
repeat
        if (j > 0) and (w > 0) then
                assign job to worker
                j ← j – 1; w ← w – 1; a ← a + 1
        elseif (j > 0) then
                handle an incoming message from workers
                increment w
        else
                get another job
                increment j
        endif
until (a = n) and (w = p)
```

Function MPI_Testsome

```
int MPI_Testsome (

      int in_cnt,  /* IN - Number of
         nonblocking receives to check */

       MPI_Request *handlearray,  /* IN -
         Handles of pending receives */

       int *out_cnt, /* OUT - Number of
         completed communications */

       int *index_array, /* OUT - Indices of
         completed communications */

       MPI_Status *status_array) /* OUT -
         Status records for completed comms */
```

Summary

- Manager/worker paradigm
  - Dynamic number of tasks
  - Variable task lengths
  - No communications between tasks
- New tools for “kit”
  - Create manager/worker program
  - Create workers-only communicator
  - Non-blocking send/receive
  - Testing for completed communications