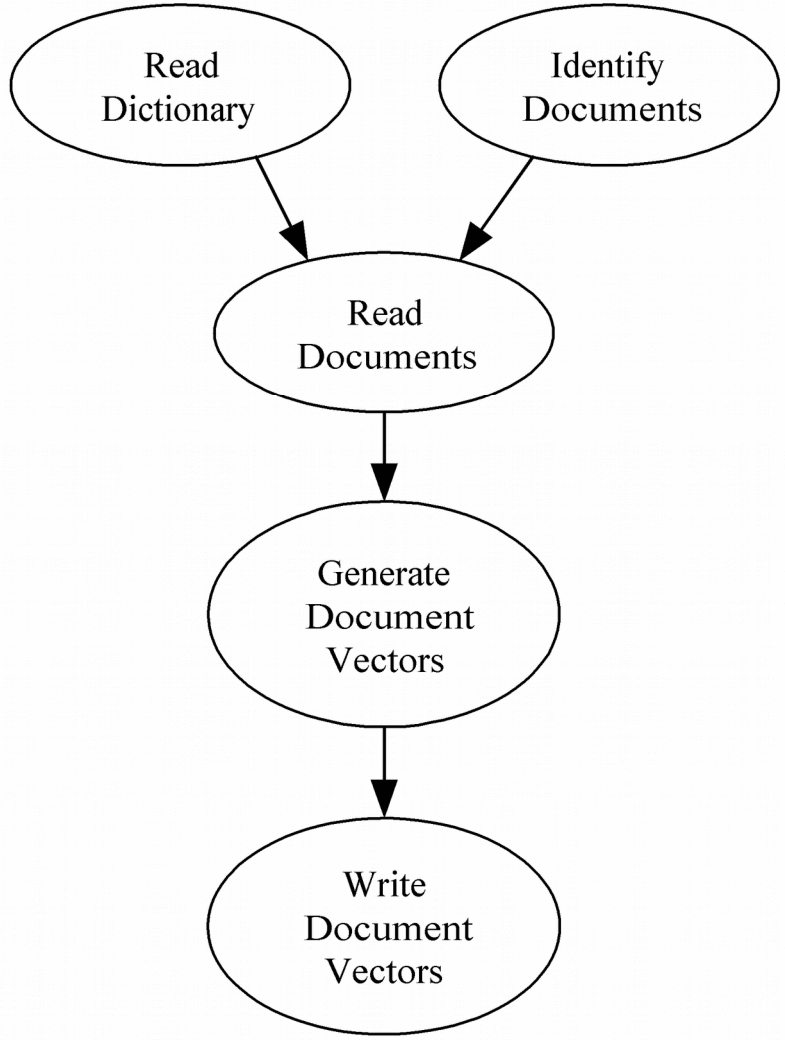
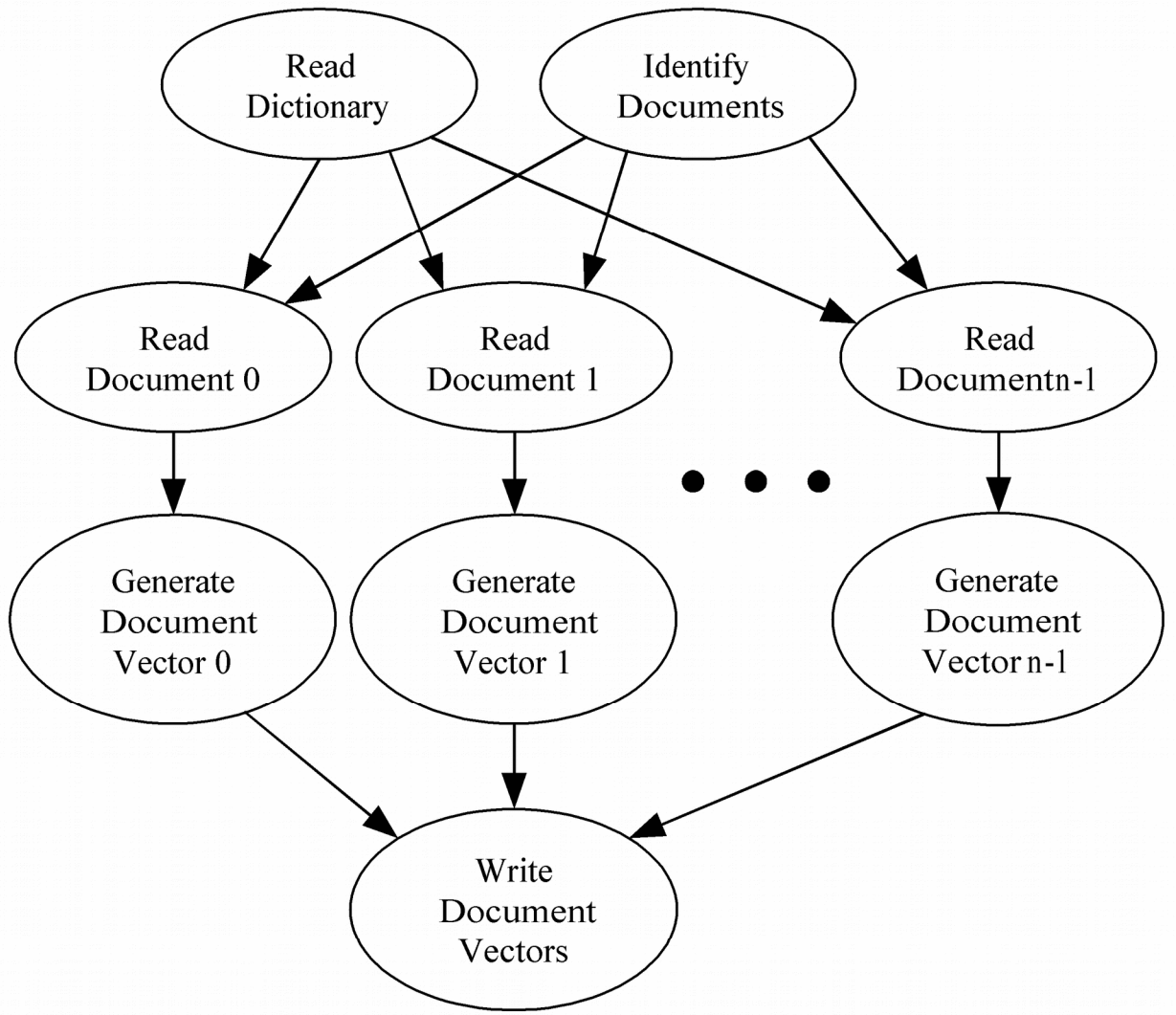
Document Classification Problem

- 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

Data Dependence Graph (1)



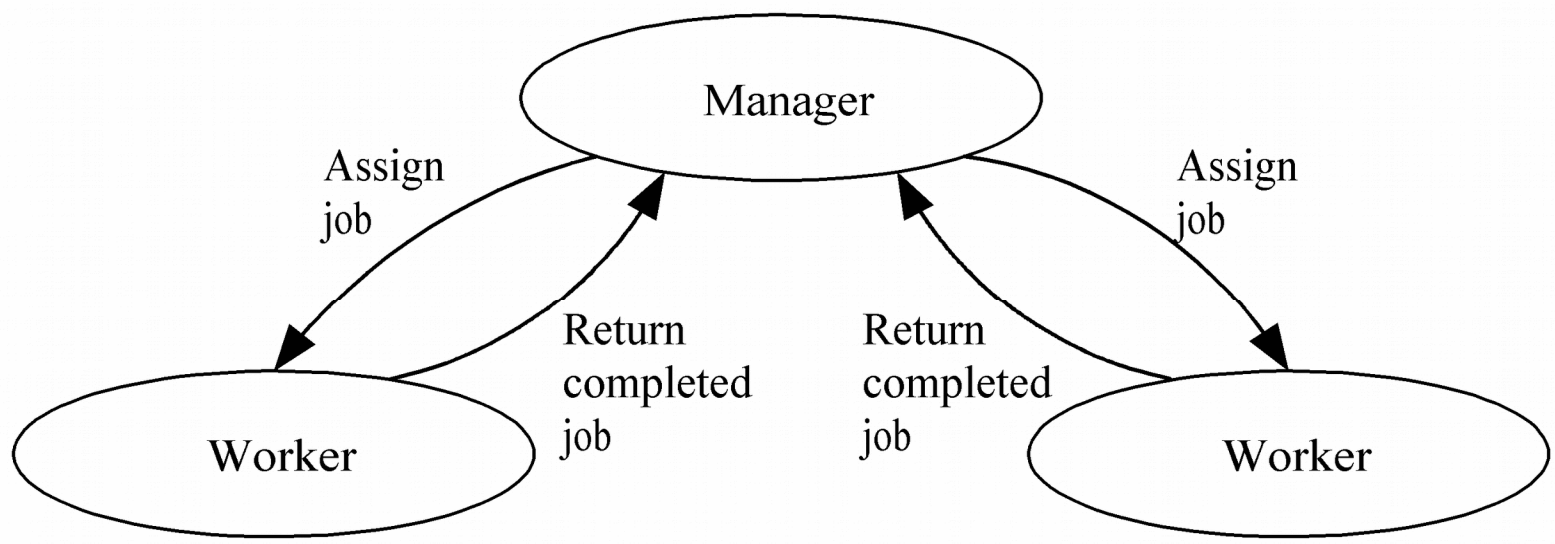
Data Dependence Graph (2)



Agglomeration and Mapping

- Number of tasks not known at compile time
  - Can not do static allocation of tasks
- Tasks do not communicate with each other
  - Embarassingly parallel taks
- Time needed to perform tasks varies widely
  - Load balancing becomes an important issue
- Strategy: map tasks to processes at run time

Manager/worker-style Algorithm

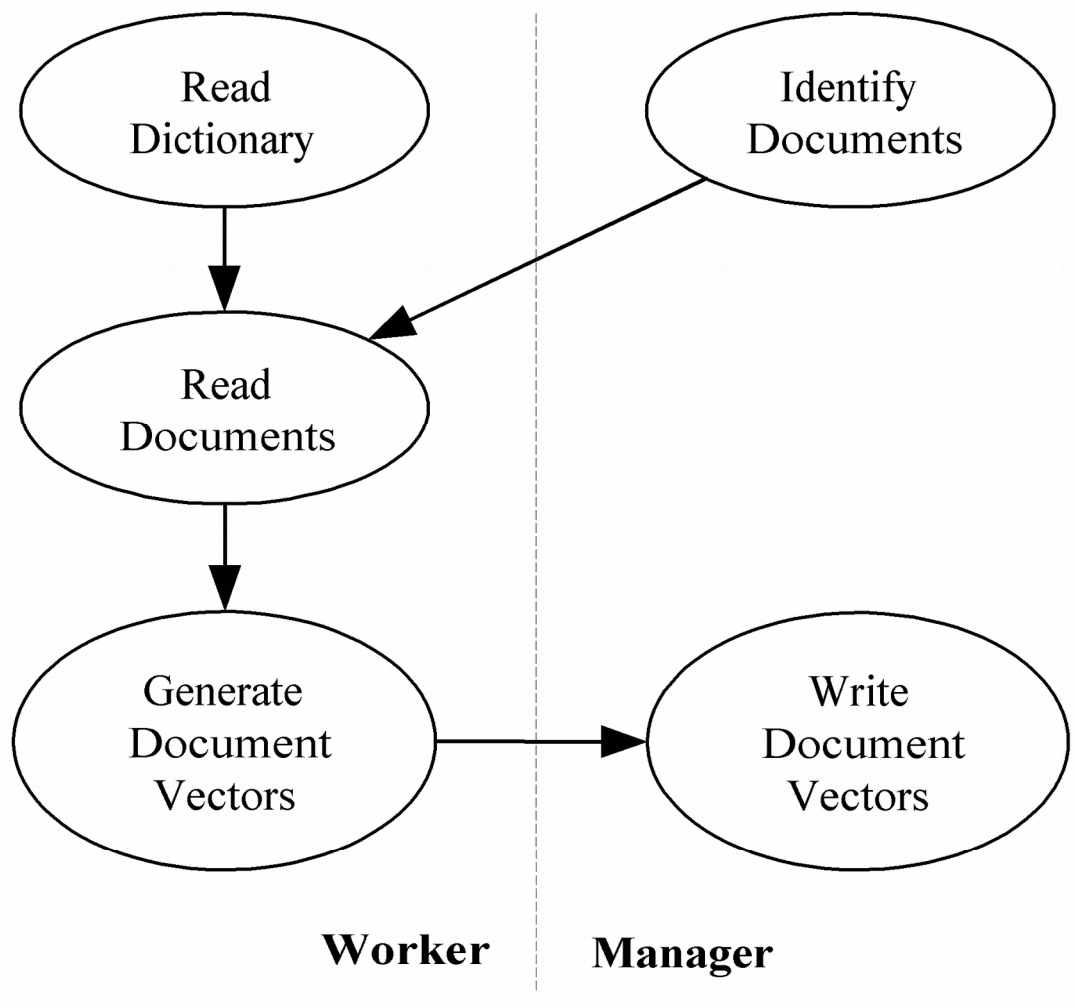


Can also be viewed as domain partitioning with run-time allocation of data to tasks

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



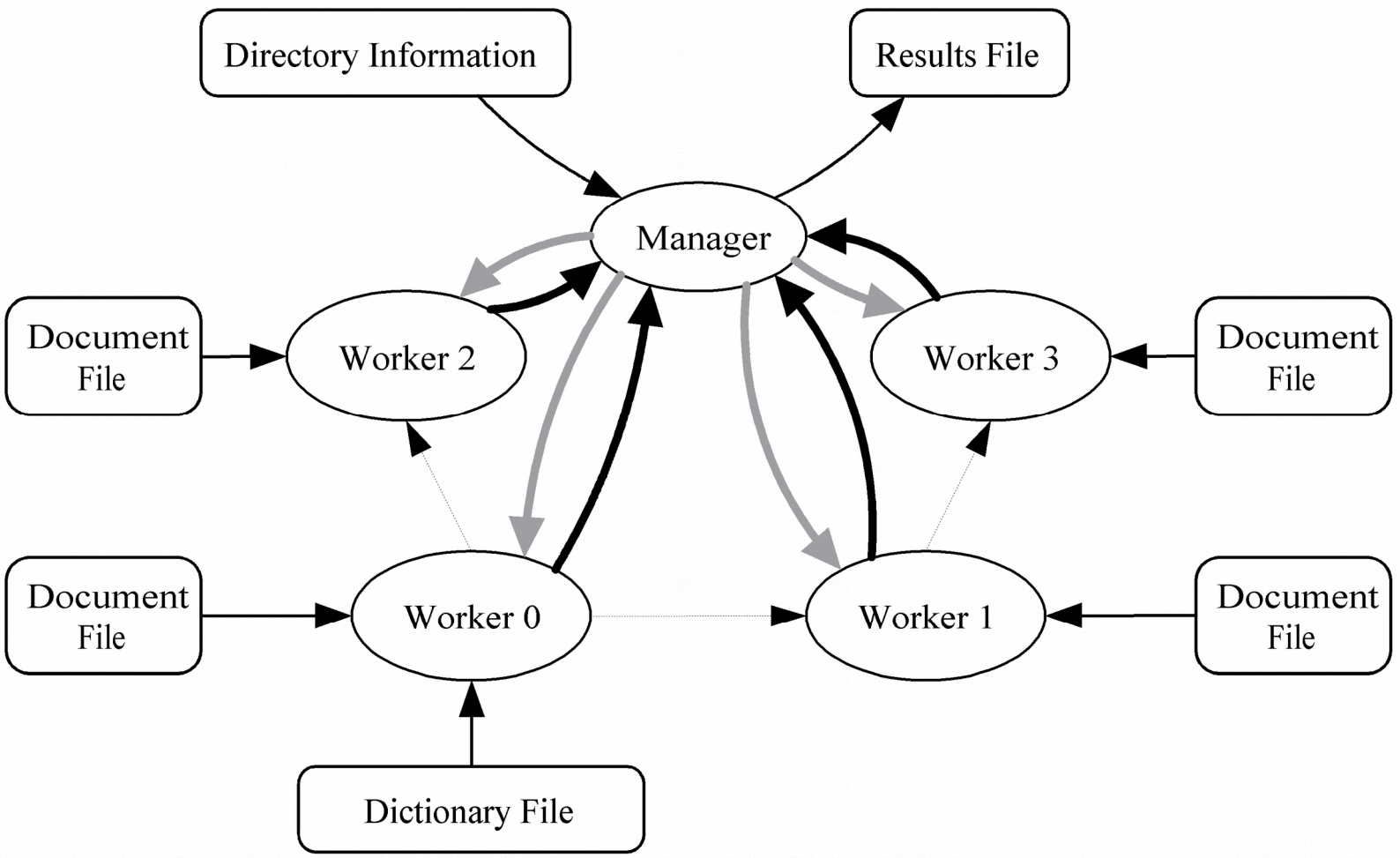
Manager Pseudocode

```
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



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_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

Function MPI_Wait()

A call to MPI_Wait() returns when the operation identified by handle is complete.

```
int MPI_Wait (

      MPI_Request *handle,

      MPI_Status *status

)
```

MPI_Status – member variables:

- **Rank of sender**
- **Tag of message**
- **Length of message**

Function MPI_Probe()

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

- **Blocks until message is available to be received, but does not consume the message from the input buffer.**
- **The MPI_Status variable fields are filled in and can be used to learn the sender and the tag.**

Function MPI_Get_count

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

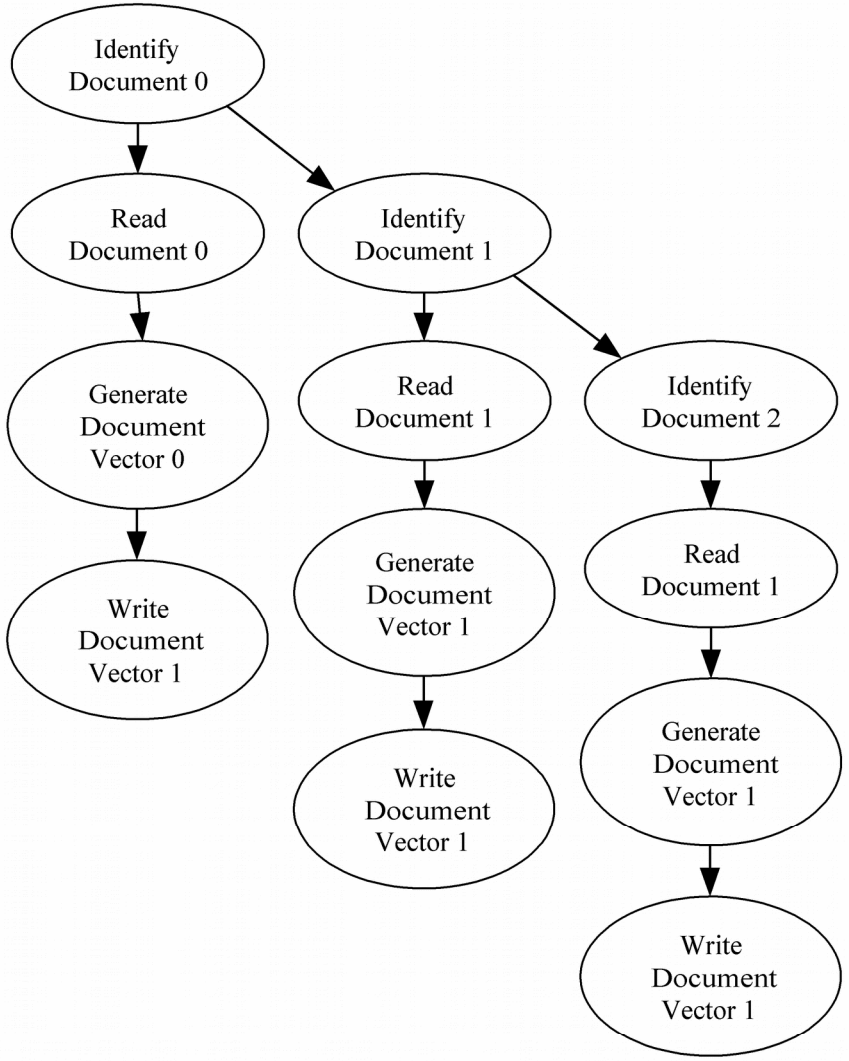
The returned count variable is the total number of datatype elements that were received in a message.

Usage of MPI_Get_count()

```
MPI_Status status;
// Receive at most MAX_NUMBERS from process zero
MPI_Recv(numbers, MAX_NUMBERS, MPI_INT, 0,0,MPI_COMM_WORLD,
                &status);
// After receiving the message, check the status to determine
// how many numbers were actually received
MPI_Get_count(&status, MPI_INT, &number_amount);

// Print off the amount of numbers, and also print additional
// information in the status object
printf("1 received %d numbers from 0. Message source = %d, tag = %d\n",
        number_amount, status.MPI_SOURCE, status.MPI_TAG);
```

Pipelining


Identify Document 0
Read Document 0
Identify Document 1
Generate Document Vector 0
Read Document 1
Identify Document 2
Write Document Vector 1
Generate Document Vector 1
Read Document 1
Write Document Vector 1
Generate Document Vector 1
Write Document Vector 1

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_count,  /* IN - Number of
        nonblocking receives to check */

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

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

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

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

Usage of MPI_Testsome()

- MPI_Testsome() is a local operation, which returns immediately
- If no operation has completed it returns out_count = 0.
- A send or receive operation is guaranteed to be eventually successful

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