CHAPTER

9

Document Classification

It is impossible to enjoy idling thoroughly unless one has plenty of work to do.
Jerome Klapka Jerome, *Idle Thoughts of an Idle Fellow*

9.1 INTRODUCTION

The World Wide Web contains millions of text documents. Many questions can be answered by retrieving the right documents, but automated search engines are needed to find the documents most likely to contain relevant information. To simplify the comparison of documents with queries or against each other, practitioners often use a vector to represent the contents of a document. Each dimension of the vector represents the "fit" between the document and a concept, which may take the form of a word or phrase.

In this chapter we develop an application that reads a dictionary of key words, locates a set of text documents, reads the documents, generates a vector for each document, and writes the document vectors. In contrast to most of the problems we have examined in previous chapters, this problem is amenable to a functional decomposition. We develop a manager/worker–style parallel program to solve this problem. In the course of developing the program and discussing enhancements to it, we add the following MPI functions to our repertoire:

- `MPI_Irecv`, to initiate a nonblocking receive
- `MPI_Isend`, to initiate a nonblocking send
- `MPI_Wait`, to wait for a nonblocking communication to complete
- `MPI_Probe`, to check for an incoming message
- `MPI_Get_count`, to find the length of a message
- `MPI_Testsome`, to return information on all completed nonblocking communications

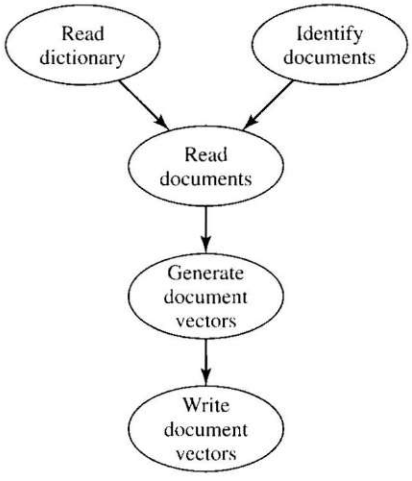


Figure 9.1 The document classification problem consists of five general tasks.

9.2 PARALLEL ALGORITHM DESIGN

Our objective is a program that reads a dictionary and searches a directory structure for plain text files (such as `.html`, `.tex`, and `.txt` files). For each of these files, the program opens the file, reads its contents, and generates a profile vector that indicates how many times the text document contains each word appearing in the dictionary. The program writes a file containing the profile vectors for each of the plain text files it has examined. A data dependence diagram for the five steps appears in Figure 9.1.

9.2.1 Partitioning and Communication

While reading the dictionary and identifying the documents may be performed concurrently, we need to break the tasks into finer pieces if we are going to exploit parallelism more fully. Let's assume that reading documents and generating the profile vectors consume the vast majority of execution time. It makes sense, then, to generate two tasks for *each document*: one to read the document file and another to generate the vector. The resulting data dependence graph appears in Figure 9.2.

This algorithm is a natural candidate for a functional decomposition. Each operation is a primitive task. A data element (i.e., a document) is associated with each task.

9.2.2 Agglomeration and Mapping

The number of tasks is not known at compile time. Tasks do not communicate with each other. The time needed to perform each task (process each document)

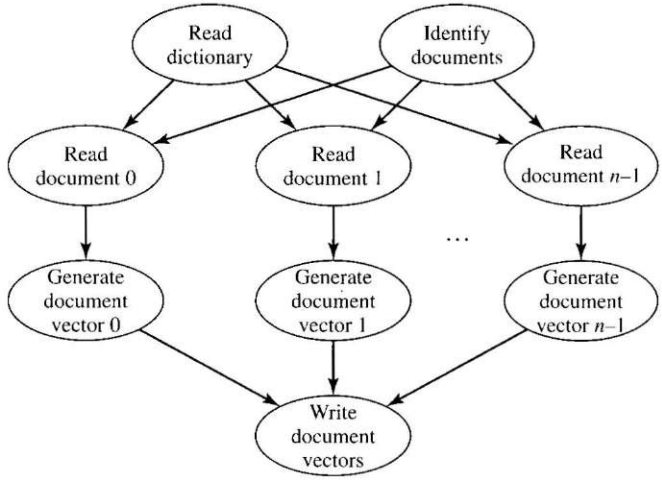


Figure 9.2 The reading and profiling of each document may occur in parallel.

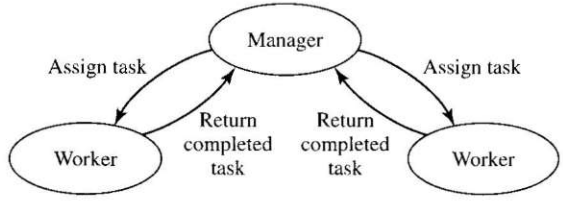


Figure 9.3 In a manager/worker–style parallel algorithm, a manager process assigns tasks to and receives results from a set of worker processes.

may vary widely, because the documents may have radically different sizes, and some documents (such as `.html` files) may be more difficult to process than others (such as `.txt` files). Given these characteristics, our mapping decision tree (Figure 3.7) suggests we should map tasks to processes at run-time.

9.2.3 Manager/Worker Paradigm

To support the run-time allocation of tasks to processes, we will construct a manager/worker–style parallel program. One process, called the **manager,** is responsible for keeping track of assigned and unassigned data. It assigns tasks to the other processes, called **workers,** and retrieves results back from them (Figure 9.3).

The advantage of allocating only a single task at a time to each worker is that it balances workloads. A worker is done when it completes a task and the

manager has no more tasks to assign. At this point, no worker has more than one task left to complete.

The disadvantage of allocating a single task at a time to each worker is that it introduces additional communication overhead into the parallel algorithm, increasing execution time and lowering speedup.

To date, all of the parallel programs we have written are in the **SPMD** (Single Program Multiple Data, sometimes pronounced "spim-dee") style. In SPMD programs every process executes the same functions (though a designated process may be responsible for file or user I/O). The manager/worker model is a clear break from the SPMD style of programming. The manager process has different responsibilities from the worker processes. In a parallel program implementing a manager/worker algorithm, there is typically a control flow split early in the program's execution that sends the manager process off executing one function and the worker processes off executing another function.

Keeping workloads balanced is essential for high efficiency, and we choose the manager/worker paradigm as the basis for our parallel algorithm design. Our first step is to decide which tasks should be done by the manager and which should be done by the workers. Identifying the documents is clearly a job for the manager, since it is the manager that will be assigning file names to the workers. Reading the dictionary should be done by the workers, since they are the processes that will be constructing the profile vectors. Given a document file name, a worker will read the file and produce the document profile vector. Finally, we'll give the manager responsibility for gathering the document vectors and writing the results file.

In the task/channel graph of Figure 9.3, you can see there is an interaction cycle between the manager and each worker. The manager provides the worker with a task. Some time later the worker returns the completed task to the manager (or simply reports that the task is done). At this point the manager may give the worker another task.

The cycle may begin with either a message from the manager to the worker or vice versa. Which should come first? In our design, we choose to have the worker initiate the dance by sending a message to the manager indicating it is ready to receive a task. We do this because we cannot be certain when the MPI processes on different processors begin execution. This way, the manager only sends tasks to workers it knows are active.

9.2.4 Manager Process

Pseudocode for the manager process appears in Figure 9.4. The manager begins by identifying the n plain text documents in the directory specified by the user. It receives from worker 0 the value of k, the number of elements in each document vector, so that it can allocate $n \times k$ matrix s for storing the vectors it receives from the workers. It initializes variables d and t showing that no documents have been assigned and no workers have been terminated, respectively.

The manager enters a loop that it repeats until it has terminated all workers. In this loop it receives a message from a worker. If the message contains a document's

```
Manager
  Local variables
    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 (p − 1 are workers)
    s — storage array containing document vectors
    t — terminated workers
    v — individual document vector

  Identify n documents in user-specified directory
  Receive dictionary size k from worker 0
  Allocate s with dimension n × k to store document vectors
  d ← 0
  t ← 0
  repeat
    Receive message from worker j
    if message contains document vector v
      s[a[j]] ← v
    else
      {Message is first request for work—do nothing}
    endif
    if d < n then
      Send name of document d to worker j
      a[j] ← d
      d ← d + 1
    else
      Send termination message to worker j
      t ← t + 1
    endif
  until t = p − 1
  Write s to output file
```

Figure 9.4 Pseudocode for the document classification manager process.

profile vector, it stores the vector in the appropriate place in s. Otherwise, the worker is simply indicating it is ready for a document. (This only happens once per worker.) If there are any documents left, the manager sends the file name to the worker, records in array a which document it assigned, and increments d, the number of documents assigned. If there are no documents left, the manager sends a termination message to the worker and increments the termination count. It repeats the loop until it has terminated all of the workers.

Exiting the loop, the manager process writes to a file the document profile vectors stored in s.

9.2.5 Function `MPI_Abort`

Recall that after the manager identifies the n plain text documents in the directory and receives k, the document vector size, from process 0, it must allocate an $n \times k$

matrix for storing the vectors. This is an operation that only the manager process performs—the worker processes are off doing other things at this point.

If the memory allocation fails, we need a simple way to terminate the execution of the MPI program. Function `MPI_Abort` gives us this power. It has this header:

```
int MPI_Abort (MPI_Comm comm, int error_code)
```

Function `MPI_Abort` makes a "best effort" attempt to abort all processes in the communicator passed as `comm`. It returns to the calling environment the value of `error_code`.

9.2.6 Worker Process

Now let's think about the worker processes. Every worker needs a copy of the dictionary. One solution is for every worker to open the dictionary file and read its contents. Another option is for one worker to open the dictionary file, read its contents, and then broadcast the dictionary to the other workers. If the broadcast bandwidth inside the parallel computer is greater than the bandwidth between the file server and the parallel computer, the second strategy is better. It is the one we adopt.

The pseudocode for the worker process is in Figure 9.5. As soon as a worker becomes active, it notifies the manager it is ready for work. (Technically, this

```
Worker
  Local variables
    f — file name
    k — dictionary size
    v — document vector

  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 k to manager
  endif
  repeat
    Receive file name f from manager
    if f indicates termination
      exit loop
    else
      Read document from file f
      Generate document vector v
      Send v to manager
    endif
  forever
```

Figure 9.5 Pseudocode for the document classification worker process.

is not true, because the worker has not yet acquired the dictionary. However, making the request early allows the time spent sending the message and receiving the first document's file name from the manager to be overlapped with dictionary setup time.) Worker 0 reads the dictionary. All workers (but not the manager) participate in the collective communication operation to broadcast the dictionary. Each worker constructs a hash table from dictionary elements. This will enable constant-time access (in the average case) to dictionary entries, speeding the profiling of the documents. Worker 0 also sends the dictionary size to the manager.

The worker process enters a **repeat . . . forever** loop. It receives a message from the manager. If the message is the name of a file containing a document, the worker reads the file, generates the document vector, sends the document vector to the manager, and iterates. If the worker receives a termination message from the manager, there are no more documents to process, and the worker ceases execution.

We can create a task/channel graph for this manager/worker design. It is illustrated in Figure 9.6 for the case when there are five processes (one manager and four workers).

We need to decide which process will be the manager. It makes no particular difference, but for the sake of simplicity, let's assign management responsibilities to the process with rank 0 in `MPI_COMM_WORLD`; processes with ranks $1, 2, \ldots, p-1$ will be the workers.

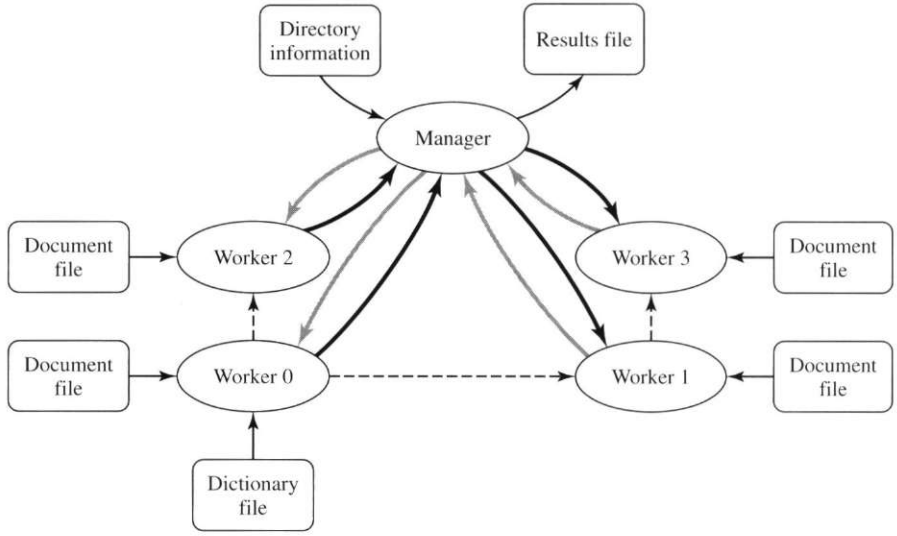


Figure 9.6 Task/channel graph for the parallel document classification algorithm. Dashed arrows represent channels used to broadcast the dictionary. Heavy gray arrows represent channels that carry document names to workers. Heavy black arrows represent channels that carry document vectors to the manager.

Note that our design assumes at least two processes will execute the program—one manager and at least one worker. Our implementation needs to check to ensure at least two MPI processes are active.

9.2.7 Creating a Workers-Only Communicator

In the parallel algorithm we have designed, the dictionary is broadcast among the workers while the manager is searching the directory structure for plain text files. Function `MPI_Bcast` is a collective communication operation, meaning it must be performed by every process in a communicator.

To support a worker-only broadcast, we must create a new communicator that includes all the workers but excludes the manager. In Chapter 8 we saw how to use `MPI_Comm_split` to split a communicator into one or more new communicators.

In this case, however, we do not want the manager process to be a member of a new communicator. We can exclude the manager process by having it pass the constant `MPI_UNDEFINED` as the value of `split_key`. The return value of `new_comm` will be `MPI_COMM_NULL`.

We can create a new, workers-only communicator with this code:

```
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);
```

9.3 NONBLOCKING COMMUNICATIONS

The work of the manager process has three phases. In the first phase the manager finds the plain text files in the directory structure specified by the user, receives the dictionary size from worker 0, and allocates the two-dimensional array that is used to store the document profile vectors. In phase 2, the manager allocates documents to workers and collects profile vectors. It writes the complete set of profile vectors to a file in phase 3.

Let's focus on phase 1. The manager must search a directory structure and receive a message from worker 0. Is there a way to overlap these two activities?

To date, we have used `MPI_Send` and `MPI_Recv` for point-to-point message-passing. These are **blocking** operations. Function `MPI_Send` does not return until either the message has been copied into a system buffer or the message has been sent. In either case, you can overwrite the message buffer as soon as the function returns. Function `MPI_Recv` does not return until the message has been received into the buffer specified by the user; you may access the message values as soon as the function returns.

Blocking sends and receives may limit the performance of a parallel program. With `MPI_Send`, there may be some reason why the system does not copy the message into a system buffer. In this case the function blocks until the message has been sent, even if you have no intention of overwriting the buffer right away.

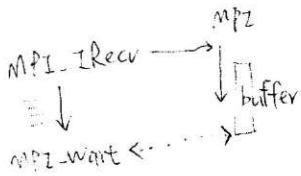
Posting a receive before a message arrives can save time, because the system can save a copy operation by transferring the contents of the incoming message directly into the destination buffer rather than a temporary system buffer. It is difficult to do this with `MPI_Recv`. If the function is called too soon, the calling process blocks until the message arrives. If the function is called too late, the incoming message has already been copied into a system buffer and must be copied again.

Fortunately, the MPI library provides **nonblocking** send and receive functions. Calls to `MPI_Isend` and `MPI_Irecv` simply post, or initiate, the appropriate communication operation. (Think of the "I" as standing for *initiate*.) The message buffer may not be accessed by the user process until it explicitly completes the communication with a call to `MPI_Wait`.

Posting a message, performing other computations or I/O operations, and then completing the message, may save time in two different ways. First, it may allow the system to eliminate message-copying by the sending and/or the receiving processes. Second, it allows dedicated communication coprocessors, if they exist on the parallel computer, to perform communication-related activities while the CPU assigned to the computation manipulates local data.

9.3.1 Manager's Communication

Getting back to our manager process, it knows at the beginning of its execution that it needs to receive the dictionary size from a worker, even though it does not actually use this value until after it has identified the document files to be processed. (Of course, the value of a nonblocking read is higher when the length of the message to be received is greater.) Let's look at the two MPI functions needed to perform a nonblocking receive.



9.3.2 Function `MPI_Irecv`

Function `MPI_Irecv` has this header:

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

The first six parameters are identical to those of `MPI_Recv`. However, since `MPI_Irecv` only initiates the receive, you cannot access `buffer` until a matching call to `MPI_Wait` has returned. The function returns, through the last parameter, a **handle** (pointer) to an `MPI_Request` object that identifies the communication operation that has been initiated.

Note that the function does not return a pointer to an `MPI_Status` object, since the receive has not yet been completed.

9.3.3 Function MPI_Wait

Here is the header for function MPI_Wait:

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

Function MPI_Wait blocks until the operation associated with pointer handle completes. In the case of a send operation, the buffer may then be assigned new values. In the case of a receive operation, the buffer may be referenced, and status points to the MPI_Status object containing information about the received message.

9.3.4 Workers' Communications

Now let's examine the needs of the worker processes for new MPI functionality. Before being assigned its first document, each worker must notify the manager process that it is active. It can initiate this send to the manager, then proceed immediately to the broadcasting of the dictionary and the construction of the hash table.

The worker also must receive file names (actually, complete path names) from the manager. There is no way of knowing in advance how long these names may be, since directory structures may be deeply nested. For this reason it would be convenient if the worker could check on an incoming message and determine its length before actually reading it.

Here are the three MPI functions that meet these needs of the workers.

9.3.5 Function MPI_Isend

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

Function MPI_Isend posts a nonblocking send operation. The first six parameters have the same meaning as in MPI_Send. The last parameter is an output parameter—a handle to an opaque MPI_Request object created by the run-time system. It identifies this communication request. The message buffer may not be reused until the matching call to MPI_Wait has returned.

9.3.6 Function MPI_Probe

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

Passed src, the rank of the message source; tag, the incoming message's tag; comm, the communicator; and status, a pointer to an MPI_Status object, function MPI_Probe **blocks** until a message matching the source and tag specifications is available to be received. It returns through the status pointer information about the source, tag, and length of the message, but it does not actually receive the message.

By passing MPI_ANY_SOURCE as the src argument, you can probe for a message from any other process. Passing MPI_ANY_TAG as the tag argument allows you to probe for any message from the process you specified in src. Using both MPI_ANY_SOURCE and MPI_ANY_TAG will allow the probe to match any sent message.

In general, it is best to keep the source and tag specifications as narrow as possible, to minimize mismatch bugs that occur when messages arrive in an unexpected order. In this case, the worker knows both the source and the tag of the message it is expecting from the manager, and there is no need for it to use these constants.

9.3.7 Function MPI_Get_count

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

Passed status, a pointer to an MPI_Status object; dtype, the data type of the message elements; and cnt, a pointer to an integer, function MPI_Get_count returns through cnt the number of elements in the message.

9.4 DOCUMENTING THE PARALLEL PROGRAM

With these new MPI functions in hand, we may now construct a parallel program to perform the document classification task. We do not include the entire program in this section; we omit the directory-searching and profile-writing functions called by the manager process and the hash-table-building and profile-generating functions called by the worker processes. Our focus is on the general structure of a manager/worker MPI program and how the new MPI functions introduced in this chapter fit into the overall design.

The program appears in Figure 9.7.

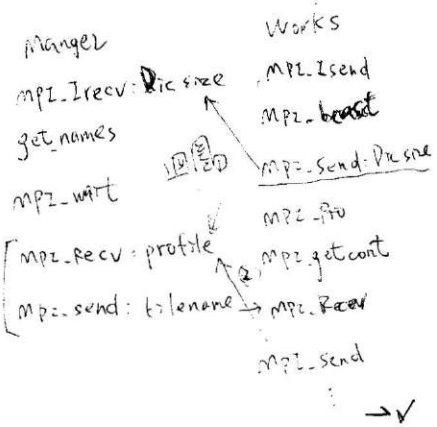
We define four constants to be used as tags for the four types of messages the processes are sending and receiving.

Using message tags helps document the code. It also allows a process to receive messages from another process in a different order than they were sent.

For example, worker 0—like all workers—sends an initial request for work to the manager. After it has read, broadcast, and processed the dictionary, worker 0 sends the dictionary size to the manager. The manager, on the other hand, needs to construct the document vector profile storage area before it begins handling requests for work from processes. For this reason it wants to receive the dictionary size message from worker 0 before it receives worker 0's initial request for work. We give these two messages different tags, enabling their out-of-order reception. We use DICT_SIZE_MSG as the tag for the message from worker 0 to the manager that contains the number of words in the dictionary file. It and all other workers inform the manager that they are active by sending the manager an empty message with the tag EMPTY_MSG.

```
/*
 *   Document Classification Program
 */

#include <mpi.h>
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
#include <ftw.h>

#define DICT_SIZE_MSG    0  /* Msg has dictionary size */
#define FILE_NAME_MSG    1  /* Msg is file name */
#define VECTOR_MSG       2  /* Msg is profile */
#define EMPTY_MSG        3  /* Msg is empty */

#define DIR_ARG          1  /* Directory argument */
#define DICT_ARG         2  /* Dictionary argument */
#define RES_ARG          3  /* Results argument */

typedef unsigned char uchar;

int main (int argc, char *argv[]) {

   int         id;           /* Process rank */
   int         p;            /* Number of processes */
   MPI_Comm    worker_comm;  /* Workers-only communicator */

   void manager (int, char **, int);
   void worker (int, char **, MPI_Comm);

   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &id);
   MPI_Comm_size (MPI_COMM_WORLD, &p);

   if (argc != 4) {
      if (!id) {
         printf ("Program needs three arguments:\n");
         printf ("%s <dir> <dict> <results>\n", argv[0]);
      }
   } else if (p < 2) {
      printf ("Program needs at least two processes\n");
   } else {
      if (!id) {
         MPI_Comm_split (MPI_COMM_WORLD, MPI_UNDEFINED,
            id, &worker_comm);
         manager (argc, argv, p);
      } else {
         MPI_Comm_split (MPI_COMM_WORLD, 0,id, &worker_comm);
         worker (argc, argv, worker_comm);
      }
   }
   MPI_Finalize();
   return 0;
}
```

Figure 9.7 Document classification program.



```
void manager (int argc, char *argv[], int p) {

   int           assign_cnt; /* Docs assigned so far */
   int          *assigned;   /* Document assignments */
   uchar        *buffer;     /* Store profile vectors here */
   int           dict_size;  /* Dictionary entries */
   int           file_cnt;   /* Plain text files found */
   char        **file_name;  /* Stores file (path) names */
   int           i;
   MPI_Request   pending;    /* Handle for recv request */
   int           src;        /* Message source process */
   MPI_Status    status;     /* Message status */
   int           tag;        /* Message tag */
   int           terminated; /* Count of terminated procs */
   uchar       **vector;     /* Profile vector repository */

   void build_2d_array (int, int, uchar ***);
   void get_names (char *, char ***, int *);
   void write_profiles (char *, int, int, char **, uchar **);

   /* Put in request to receive dictionary size */
   MPI_Irecv (&dict_size, 1, MPI_INT, MPI_ANY_SOURCE,
      DICT_SIZE_MSG, MPI_COMM_WORLD, &pending);

   /* Collect the names of the documents to be profiled */
   get_names (argv[DIR_ARG], &file_name, &file_cnt);

   /* Wait for dictionary size to be received */
   MPI_Wait (&pending, &status);

   /* Set aside buffer to catch profiles from workers */
   buffer = (uchar *)
      malloc (dict_size * sizeof(MPI_UNSIGNED_CHAR));

   /* Set aside 2-D array to hold all profiles.
      Call MPI_Abort if the allocation fails. */
   build_2d_array (file_cnt, dict_size, &vector);

   /* Respond to requests by workers. */
   terminated = 0;
   assign_cnt = 0;
   assigned = (int *) malloc (p * sizeof(int));

   do {
      /* Get profile from worker */
      MPI_Recv (buffer, dict_size, MPI_UNSIGNED_CHAR,
         MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD,
         &status);
      src = status.MPI_SOURCE;
      tag = status.MPI_TAG;
      if (tag == VECTOR_MSG) {
         for (i = 0; i < dict_size; i++)
            vector[assigned[src]][i] = buffer[i];
      }
```

Figure 9.7 (contd.) Document classification program.

```
      /* Assign more work or tell worker to stop. */
      if (assign_cnt < file_cnt) {
         MPI_Send (file_name[assign_cnt],
            strlen(file_name[assign_cnt])+1,
            MPI_CHAR, src, FILE_NAME_MSG, MPI_COMM_WORLD);
         assigned[src] = assign_cnt;
         assign_cnt++;
      } else {
         MPI_Send (NULL, 0, MPI_CHAR, src, FILE_NAME_MSG,
            MPI_COMM_WORLD);
         terminated++;
      }
   } while (terminated < (p-1));

   write_profiles (argv[RES_ARG], file_cnt, dict_size,
      file_name, vector);
}

void worker (int argc, char *argv[], MPI_Comm worker_comm)
{
   char          *buffer;     /* Words in dictionary */
   hash_el      **dict;       /* Hash table of words */
   int            dict_size;  /* Profile vector size */
   long           file_len;   /* Chars in dictionary */
   int            i;
   char          *name;       /* Name of plain text file */
   int            name_len;   /* Chars in file name */
   MPI_Request    pending;    /* Handle for MPI_Isend */
   uchar         *profile;    /* Document profile vector */
   MPI_Status     status;     /* Info about message */
   int            worker_id;  /* Rank in worker_comm */

   void build_hash_table (char *, int, hash_el ***, int *);
   void make_profile (char *, hash_el **, int, uchar *);
   void read_dictionary (char *, char **, long *);

   /* Worker gets its worker ID number */

   MPI_Comm_rank (worker_comm, &worker_id);

   /* Worker makes initial request for work */

   MPI_Isend (NULL, 0, MPI_UNSIGNED_CHAR, 0, EMPTY_MSG,
              MPI_COMM_WORLD, &pending);

   /* Read and broadcast dictionary file */

   if (!worker_id)
      read_dictionary (argv[DICT_ARG], &buffer, &file_len);
   MPI_Bcast (&file_len, 1, MPI_LONG, 0, worker_comm);
   if (worker_id) buffer = (char *) malloc (file_len);
   MPI_Bcast (buffer, file_len, MPI_CHAR, 0, worker_comm);
```

Figure 9.7 (contd.) Document classification program.

```
    /* Build hash table */

    build_hash_table (buffer, file_len, &dict, &dict_size);

    profile = (uchar *) malloc (dict_size * sizeof(uchar));

    /* Worker 0 sends msg to manager re: size of dictionary */

    if (!worker_id) MPI_Send (&dict_size, 1, MPI_INT, 0,
                              DICT_SIZE_MSG, MPI_COMM_WORLD);

    for (;;) {

       /* Find out length of file name */

       MPI_Probe (0, FILE_NAME_MSG, MPI_COMM_WORLD, &status);
       MPI_Get_count (&status, MPI_CHAR, &name_len);

       /* Drop out if no more work */

       if (!name_len) break;

       name = (char *) malloc (name_len);
       MPI_Recv (name, name_len, MPI_CHAR, 0, FILE_NAME_MSG,
                 MPI_COMM_WORLD, &status);

       make_profile (name, dict, dict_size, profile);
       free (name);

       MPI_Send (profile, dict_size, MPI_UNSIGNED_CHAR, 0,
                 VECTOR_MSG, MPI_COMM_WORLD);
    }
}
```

Figure 9.7 (contd.) Document classification program.

When the manager assigns a document to a worker, it uses message tag `FILE_NAME_MSG`. When a worker responds with the profile vector for that document, it uses message tag `VECTOR_MSG`.

We also define constants to refer to the three command-line arguments. The first argument, indexed by `DIR_ARG`, is the name of the directory that serves as the root of the directory structure to be searched for plain text files. The second, `DICT_ARG`, is the name of the file containing the dictionary. The third argument, indexed by `RES_ARG`, is the name of the output file that contains the set of document profile vectors upon successful completion of the program.

Function `main` contains code that all processes execute before the manager goes one way and the workers go another. Execution begins with the traditional calls to initialize MPI and retrieve the process rank and the process group size.

The function checks to ensure that the user supplied the correct number of arguments on the command line. If not, execution will not continue. The function also checks to ensure that there are at least two processes. Without at least one worker, no documents will be processed. If these conditions are satisfied, all

processes cooperate to create a new, workers-only communicator. After this has been done, the roles of the manager and the workers diverge. Process 0 (the manager) calls `manager` and the other processes call `worker`.

Only a single process executes function `manager`. Refer back to the pseudocode of Figure 9.4 for a refresher on how it is structured. The manager begins by posting a receive for the message containing the size of the dictionary. It then calls `get_names` to construct `file_name`, an array of strings. These strings are the names of the plain text files in the directory tree specified by the user on the command line. After the function returns, `file_cnt` is the number of files that need to be processed.

At this point the manager needs to allocate the memory that will be used to hold the document profile vectors. The number of vectors is equal to the number of documents (`file_cnt`). The length of the vectors depends on the number of dictionary entries. So the manager must wait until the receive it posted has been completed. After the message containing the dictionary size has arrived, the manager constructs the two-dimensional array holding the vectors.

Before the principal loop of the function, the manager initializes the number of terminated processes and the number of assigned documents to 0. It also allocates the array that will be used to keep track of the document currently assigned to each process.

Inside the loop, the manager receives the next message from a worker. If the message tag indicates the message contains a document profile vector, the manager stores it. If unassigned documents remain, the manager sends the name of the next unassigned document to the worker and increments the number of assigned documents. Otherwise, it sends an empty file name to the worker, indicating to the worker that it should cease, and increments the number of terminated workers. The loop continues until all of the workers have been terminated.

The manager exits the loop only after it has received all of the document profile vectors from the workers. It writes the vectors to the file the user specified on the command line.

Now let's look at function `worker`. If you need to refresh your memory of what the worker does, refer to the pseudocode in Figure 9.5. Each worker begins by finding its rank in the worker-only communicator. If the workers did not interact, this would not be necessary, but in this algorithm, worker 0 is responsible for reading the dictionary file and broadcasting it to the other workers. Hence the workers need to know their ranks.

After calling `MPI_Comm_rank`, each worker makes its initial request for work. The message tag `EMPTY_MSG` indicates to the manager that this is the worker's initial request for work, not a message containing a document profile vector.

Next, worker 0 reads the dictionary and broadcasts it to the other workers. Note that before broadcasting the dictionary worker 0 broadcasts an integer containing the size of the dictionary. That way, the other workers can allocate enough space to hold the dictionary's contents. The workers extract the words from the dictionary and put them in a hash table. This will speed the document classification

task by enabling the process to determine in constant time (on average) if a word in the document appears in the dictionary.

After extracting words from the dictionary, the workers know how large the document profile vectors will be. In this implementation the document profile vector contains an unsigned character for each dictionary entry, enabling the correlation between the document and that entry to be expressed as an integer between 0 and 255. Each worker allocates room for a profile vector. Worker 0 sends the dictionary-size message to the manager.

Now the worker enters its principal loop. It probes for a message from the manager containing the name of a plain text document file. It allocates enough room to receive the file name, then calls `MPI_Recv` to actually get the name. Given the file name and the hash table, function `make_profile` builds the document profile vector. The worker sends this vector back to the manager. When the worker receives a zero-length file name from the manager, that means there are no more documents to process, and the worker returns from the function.

9.5 ENHANCEMENTS

In this section we consider ways to improve the execution time of our parallel document classification program.

9.5.1 Assigning Groups of Documents

In some applications a preallocation of data to processes can result in an imbalanced workload. Imbalanced workloads lead to idle processors, which lowers speedup. Allocating data to processes at run-time balances the workloads. On the other hand, it introduces additional interprocessor communication overhead, which lowers speedup. Sometimes, the best design chooses a middle point between the two extremes. For example, we might construct a manager/worker algorithm in which the manager assigns k tasks at a time to workers.

9.5.2 Pipelining

Let's reconsider the task graph of Figure 9.2. If one process can retrieve the dictionary file from the file server as quickly as k tasks, then there is not much we can do with that task. We're already allowing processes to build their hash tables concurrently. We will consider this task no further.

On the other hand, we can make improvements to the document identification and results writing tasks. When we began our design, we assumed that nearly all of the time would be spent reading document files and generating the associated profile vectors. We left the tasks of identifying the plain text files and writing the results file as sequential tasks. No document files are processed until the manager has identified all of them. If the time needed to perform these tasks is not negligible, then our design will not scale well to larger numbers of processes (Amdahl's Law).

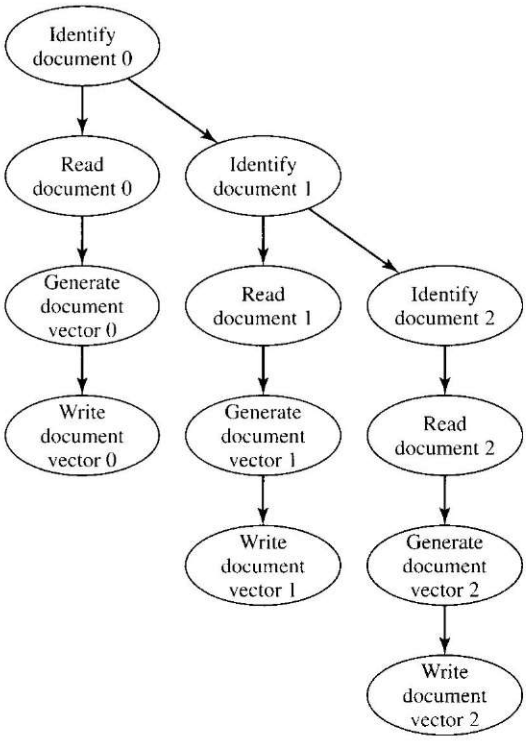


Figure 9.8 By dividing the task of identifying documents into its elemental tasks, we can expose the opportunity for pipelining the processing of the documents. We can be constructing the profile vector for document 0 while we are reading document 1 while we are identifying document 2.

Let's reconsider the task graph of Figure 9.2, ignoring the dictionary construction task. What happens when we divide the document identification task into smaller units? We end up with the new task graph shown in Figure 9.8. If we "dangle" the graph by the "Identify Document 0" node, we can see that while we are identifying document 1, we could begin reading document 0. While we are in one phase of processing document i, we can be in later phases of processing documents $i - 1$, $i - 2$, etc. This is an example of pipelining.

Pipelining can dramatically reduce the execution time of a parallel algorithm exhibiting functional parallelism. For example, Figure 9.9 illustrates how a pipelined program with one read process, two worker processes, and one write process could outperform a nonpipelined program with one manager and three workers.

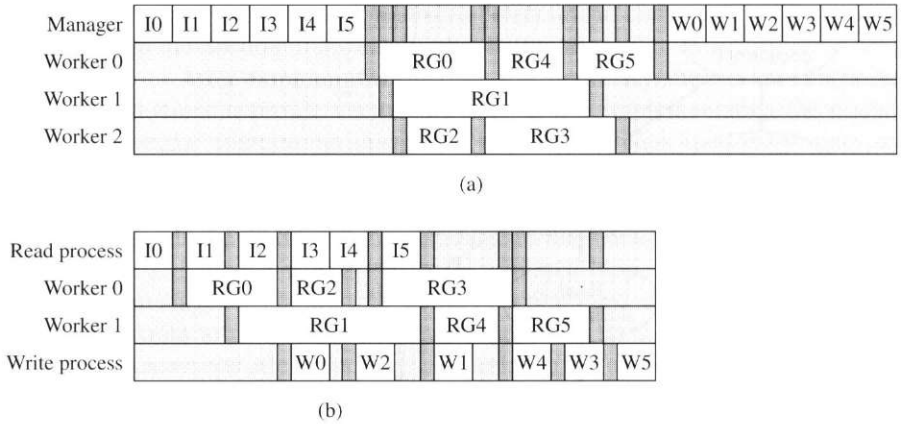


Figure 9.9 Pipelining tasks can reduce the execution time of programs with functional parallelism. In this figure I*i* refers to the task of identifying text file *i*, RG*i* refers to the task of reading file *i* and generating document profile vector *i*, and W*i* refers to the task of writing document profile vector *i* to a file. The dark gray bars represent time spent communicating. (a) A manager process identifies all text files before assigning them to workers. It collects all document vectors before writing them. This is the approach taken for the program developed in this chapter. (b) A pipelined solution. One process identifies text files, two more processes read text files and generate profile vectors, and a fourth task writes the profile vectors to a file.

The downside of implementing a manager/worker program incorporating pipelining is that it can be much more complicated. In the previous implementation the manager identified all the document file names before responding to any worker requests. Suppose we want to implement a manager that gets workers busy as soon as possible. In other words, as soon as the manager has identified at least one document and has received at least one request for work from a process, it starts sending document names to processes. That means the manager must multiplex its time between identifying documents and responding to the requests of the workers.

Here is one way we might implement the document identification/task assignment logic. Let j be the number of unassigned tasks and w be the number of workers waiting for something to do. If $j > 0$ and $w > 0$, then the manager can assign $\min(j, w)$ tasks to workers. If $j > 0$, the manager should check to see if any messages from workers have arrived. If so, the manager can receive these messages. Then we're back in the situation where $j > 0$ and $w > 0$. Otherwise, the manager should find more tasks.

9.5.3 Function `MPI_Testsome`

To implement this functionality, we need a way to check, without blocking, whether one or more expected messages have arrived. The MPI library provides four functions to do this: `MPI_Test`, `MPI_Testall`, `MPI_Testany`, and

MPI_Testsome. All of these functions require that you pass them handles to MPI_Request objects that result from calls to nonblocking receive functions. We'll describe how to use MPI_Testsome, which is the best function to use for the purpose we have described.

The manager posts a nonblocking receive to each of the worker processes. It builds an array of handles to the MPI_Request objects returned from these function calls. In order to determine if messages have arrived from any or all of the workers, the manager calls MPI_Testsome, which returns information about how many of the messages have arrived.

Function MPI_Testsome has this header:

```
int MPI_Testsome (int in_cnt, MPI_Request *handlearray,
                  int *out_cnt, int *index_array,
                  MPI_Status *status_array)
```

Passed in_cnt, the number of nonblocking receives to check, and handlearray, an array containing MPI_Request handles, function MPI_Testsome returns out_cnt, the number of completed communications. The first out_cnt entries of index_array contain the indices in handlearray of the completed communications. The first out_cnt entries of status_array contain the status records for the completed communications.

9.6 SUMMARY

The manager/worker paradigm is an effective way to think of parallel computations where it is difficult to preallocate work to processes and guarantee balanced workloads. In this chapter we considered the problem of classifying a set of plain text documents according to a user-supplied dictionary. Since document sizes can vary widely, and since some documents may be easier to process than others, the manager/worker design is appropriate.

In the process of developing this application, we introduced some additional MPI capabilities. We used function MPI_Comm_split to create a new, workers-only communicator that facilitated the broadcast of the dictionary among the workers. We discovered several places where communications could be overlapped with either computations or other I/O operations. We introduced the nonblocking communications functions MPI_Isend and MPI_Irecv and their companion completion function, MPI_Wait. We also saw how it could be beneficial for the worker processes to check the length of the path names sent by the manager before actually reading them. Functions MPI_Probe and MPI_Get_count allow this to be done.

We examined two ways to enhance the performance of the parallel program. The first enhancement is to consider a "middle ground" between preallocating all documents to tasks (which can lead to an imbalanced workload) and allocating documents one at a time to tasks (which can lead to excessive interprocessor communication). In some applications the best performance may be obtained by allocating small groups of tasks to workers.

We also considered the use of pipelining to reduce the large sequential component represented by the task of identifying all of the plain text documents. If the processing of earlier documents can be overlapped with the identification of later documents, execution time can be reduced. Implementing this enhancement requires a more complicated manager process, which must be able to divide its time between identifying documents and responding to requests from workers. In order to perform this multiplexing, the manager must be able to test for messages from workers without blocking. We examined function `MPI_Testsome`, which provides this ability.

This chapter completes our introduction to the MPI library. While the library contains well over 100 functions, the 27 functions we have described form a powerful subset that is more than adequate for a wide variety of applications.

9.7 KEY TERMS

blocking communication
handle
manager/worker paradigm
nonblocking communication
SPMD programming

9.8 BIBLIOGRAPHIC NOTES

Parallel Programming by Wilkinson and Allen has good coverage of manager/worker–style parallel programming [115]. Carriero and Gelerntner see the manager/worker paradigm (which they call the master-worker paradigm) as an embodiment of what they call "agenda parallelism," which "focuses on the list of tasks to be performed" [15].

A vector containing the frequency of individual words, while an old technique, is still the most popular document representation method [75].

9.9 EXERCISES

9.1 Assume a manager-worker scheme with a single manager and $p - 1$ workers, where $p \geq 2$. The manager assigns tasks one at a time to the workers. Assume k tasks with execution times $t_0, t_1, \ldots, t_{k-1}$ must be performed, and assume it requires 0 time to dispatch a task or return a result.

a. Under what circumstances is efficiency maximized? What is the maximum efficiency possible?

b. Under what circumstances is efficiency minimized? What is the minimum efficiency possible?

9.2 Give two reasons why the use of nonblocking sends and receives can reduce the execution time of a parallel program.

9.3 The tag `EMPTY_MSG` is not strictly necessary in the implementation of the document classification program. Describe two other ways the

manager could distinguish between a process's initial request for work and a subsequent message containing a document profile vector.

9.4 In function `worker`, explain why there is no need for the call to `MPI_Isend` to have a matching call to `MPI_Wait`.

9.5 Describe how the performance of function `manager` could be improved through the additional use of nonblocking communications.

9.6 In the document classification program developed in this chapter, one worker reads the dictionary and broadcasts its contents to the other workers. All workers then build a hash table. An alternative design would have one worker read the dictionary, build the hash table, and broadcast the hash table to the other workers. Explain the principal problem that needs to be overcome with this design, and describe a way to solve it. (Assume the hash table resolves collisions by chaining; that is, by putting all elements that hash to the same index into a linked list.)

9.7 Write a manager/worker–style program to perform matrix-vector multiplication. The manager process should read the vector from a file and distribute a copy of it to all of the workers. Next, the manager should read the matrix from a file and distribute rows of the matrix to the worker processes on demand. For each row the manager sends a worker, it should receive in return an element of the solution vector. After all of the results have been received, the manager should print the product vector to standard output.

9.8 Write a manager/worker–style parallel program that finds the smallest positive root of the equation

$$f(x) = -2 + \sin x + \sin x^2 + \sin x^3 + \cdots + \sin x^{1000}$$

This root is the unique value r between 0 and 1 such that $f(r) = 0$. The program should divide the interval [0, 1] into several subintervals and create a set of tasks, one for each subinterval. Each task computes the value of $f(x)$ at both ends of its subinterval. One task will find the subinterval where the function changes from negative to positive. The algorithm can then iterate, dividing this subinterval into pieces. When the subinterval size becomes less than 10^{-11}, the program should terminate, printing the root of the function.

9.9 Write a parallel program to display a well-known Mandelbrot set (Figure 9.10). A Mandelbrot set is a set of points in the complex plane that are quasi-stable (will increase or decrease, but will not exceed a limit) when computed by iterating a function. For this particular Mandelbrot set, we repeatedly compute

$$z_{k+1} = z_k^2 + c$$

where $z_0 = 0$ and c is a complex number representing the position of a point in the complex plane. In other words, the $k + 1$st value of the complex number z is computed from c and the kth value of z.

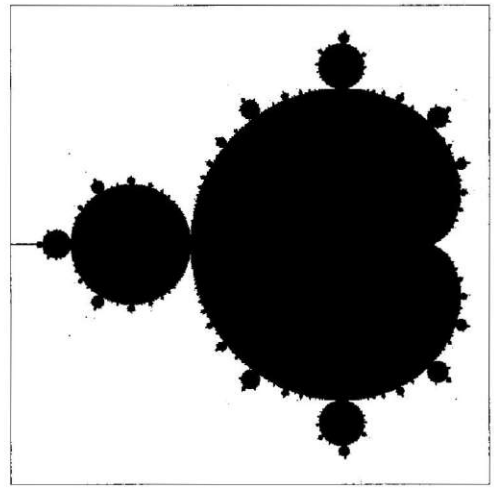


Figure 9.10 The Mandelbrot set is an example of a fractal. In this figure the lower left corner of the box represents the complex number $-1.5 - i$. The upper right corner of the box represents the complex number $0.5 + i$. Black points are in the set.

The magnitude of z is its distance from the origin; i.e., the length of the vector formed by its real and imaginary parts. If $z = a + bi$, the magnitude of z is $\sqrt{a^2 + b^2}$. If the magnitude of z ever becomes greater than or equal to 2, its subsequent values will grow without bound, and we know that c is not a point in the Mandelbrot set. If we iterate n times and find that the magnitude of z_n is still less than 2, we can conclude c is in the Mandelbrot set.

Your program should compute a Mandelbrot set for 600×600 evenly spaced points in a square region of the complex plane bounded by $-1.5 - i$ and $1 + i$. Let $n = 1{,}000$. If $z_{1000} < 2$, you should display point c as a member of the Mandelbrot set.

9.10 A **perfect number** is a positive integer whose value is equal to the sum of all its positive factors, excluding itself. The first two perfect numbers are 6 and 28:

$$6 = 1 + 2 + 3$$

$$28 = 1 + 2 + 4 + 7 + 14$$

The Greek mathematician Euclid (c. 300 BCE) showed that if $2^n - 1$ is prime, then $(2^n - 1)2^{n-1}$ is a perfect number. For example, $2^2 - 1 = 3$ is prime, so $(2^2 - 1)2^1 = 6$ is a perfect number. Write a parallel program to find the first eight perfect numbers.