Contents

1 Overview

The codebase `mandelbrot`, which can be found at https://github.com/dgsaf/mandelbrot, consists of the original code provided by Associate Professor Nigel Marks, with the following additions:

- `src/mandelbrot_static.f90`:

- `src/mandelbrot_master_worker.f90`:

- `src/mandelbrot_cyclic.f90`:

- `mandelbrot.slurm`:

- `mandelbrot-static.slurm`:

- `mandelbrot-master_worker.slurm`:

- `mandelbrot-cyclic.slurm`:

- `mandelbrot-jobs.sh`:

- `output/`:

- `bin/`:

- `pictures/`:

2 Serial

The Mandelbrot serial code, `src/mandelbrot.f90`, can be found in subsection A.1.

3   Static Decomposition

The Mandelbrot MPI static decomposition code, src/mandelbrot_static.f90, can be found in subsection A.2.

The load-balance for the 10 processes is show in Figure 1.

Load Balance for Static Decomposition
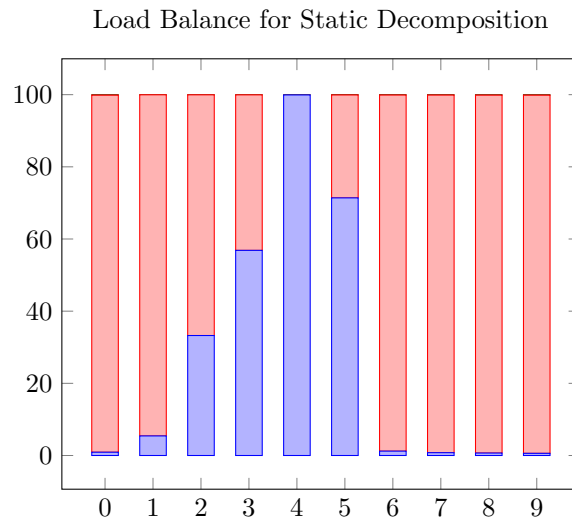

Figure 1: The load balance for the Mandelbrot MPI static decomposition scheme, with 10 processes, where $N = 8000$, maxiter = 1000. For each process, the percentage of time spent working is shown in blue, the percentage of time spent waiting in red, and the percentage of time spent communicating is shown in brown (however, this time is negligble and so is barely visible).

4   Master-Worker Scheme

The Mandelbrot MPI master-worker scheme code, `src/mandelbrot_master_worker.f90`, can be found in subsection A.3.

5   Cyclic Decomposition

The Mandelbrot MPI cyclic decomposition code, `src/mandelbrot_cyclic.f90`, can be found in subsection A.4.

A   Appendix

A.1   Serial

```fortran
program mandelbrot

  complex :: z, kappa
  integer :: green, blue, i, j, k, loop

  integer :: N, maxiter
  real , allocatable :: x(:)

  ! Timing variables.
  !!  times        An array storing time markers used to determine the following
  !!               timing variables.
  !!  time_total   Time taken overall.
  double precision :: times(1:2)
  double precision :: time_total

  ! Read command line arguments.
  call read_input(N, maxiter)

  allocate(x(0:N*N-1))

  call cpu_time(times(1))

  ! mandelbrot calculation
  do loop = 0, N*N-1
    ! i varies from 0 to N-1
    i = int(loop/N)

    ! j varies from 0 to N-1
    j = mod(loop, N)
    kappa = cmplx((4.0*(i-N/2)/N), (4.0*(j-N/2)/N))

    k = 1
    z = kappa
    do while (abs(z) <= 2 .and. k < maxiter)
      k = k+1
      z = z*z + kappa
    end do

    x(loop) = log(real(k))/log(real(maxiter))
  end do

  call cpu_time(times(2))

  ! Timing analysis.
  time_total = times(2) - times(1)

  write (*, *) &
      "timing for serial code:", NEW_LINE('a'), &
      "  total: ", time_total

  ! writing data to file
  write(*, *) "Writing mandelbrot.ppm"

  open(7, file="mandelbrot.ppm", status="unknown")

  write(7, 100) "P3", N, N, 255
```

```fortran
57
58    do loop = 0, N*N-1
59      if (x(loop) < 0.5) then
60        green = 2.0*x(loop)*255
61        write(7, 110) 255-green, green, 0
62      else
63        blue = 2.0*x(loop)*255 - 255
64        write(7, 110) 0, 255-blue, blue
65      end if
66    end do
67
68 100 format(A2, /, I4, I5, /, I3)
69 110 format(I3, /, I3, /, I3)
70
71    close(7)
72
73    ! Deallocate
74    deallocate(x)
75
76 contains
77
78    ! Read in the value of N, maxiter from command line arguments
79    subroutine read_input (N, maxiter)
80      integer , intent(out) :: N, maxiter
81      integer :: num_args
82      character(len=20) :: arg
83
84      num_args = command_argument_count()
85
86      if (num_args < 2) then
87        write (*, *) "Usage: <N> <maxiter>"
88      end if
89
90      if (num_args >= 1) then
91        call get_command_argument(1, arg)
92        read (arg, *) N
93      else
94        write (*, *) "<N> not specified. Using default value, N = 2000."
95        N = 2000
96      end if
97
98      if (num_args >= 2) then
99        call get_command_argument(2, arg)
100       read (arg, *) maxiter
101     else
102       write (*, *) &
103           "<maxiter> not specified. Using default value, maxiter = 1000."
104       maxiter = 1000
105     end if
106
107   end subroutine read_input
108
109 end program mandelbrot
```

A.2   Static Decomposition

```fortran
!> mandelbrot_static
!
!  Adaption of mandelbrot program to use MPI with a static decomposition of the
!  data.
!
!  For a NxN grid (represented in a 1D array; that is, {0, .., N*N - 1}),
!  with n_proc processes.
!
!  The default chunksize, the size of the data subset to be assigned to each
!  process, is defined to be the smallest integer such that
!  > chunksize * n_proc >= N*N.
!  This is so that even in the case where n_proc doesn't divide N*N, every
!  data point is guaranteed to be covered by a process.
!
!  Each process, p, is assigned a section of the data
!  {loop_min(p), .., loop_max(p)} such that
!  > loop_min(0) = 0
!  > loop_max(p) = loop_min(p+1) - 1, for p = 0, .., n_proc - 1
!  > loop_max(n_proc-1) = N*N - 1.
!  To ensure this, we select
!  > loop_min(p) = max(0, chunksize * p)
!  > loop_max(p) = min(N*N-1, chunksize * (p + 1) - 1)
!  We then define a process indexed chunksize array,
!  > chunksize_proc(p) = loop_max(p) - loop_min(p) + 1
!  whence, if n_proc divides N*N, we will have chunksize_proc(p) = chunksize.
program mandelbrot_static

  use mpi

  complex :: z, kappa
  integer :: green, blue, i, j, k, loop

  integer :: N, maxiter
  real , allocatable :: x(:)

  ! MPI variables.
  !  proc_id         ID of current MPI process.
  !  n_proc          Number of MPI processes.
  !  err             Stores error code for MPI calls.
  !  chunksize       The default chunksize; the maximum possible number of loop
  !                  iterations assigned to each process. Defined to be the
  !                  smallest integer such that
  !                  that chunksize * n_proc >= N*N.
  !  loop_min        An array of the lower loop-iteration bound for each
  !                  process.
  !  loop_max        An array of the upper loop-iteration bound for each
  !                  process.
  !  chunksize_proc  An array storing the size of the data subset for each
  !                  process. If n_proc divides N*N, then each element will
  !                  simply be equal to chunksize.
  !  x_proc          A work array, local to each process, which will yield the
  !                  mandelbrot data for the data subset assigned to this
  !                  process. Will have size chunksize_proc(proc_id).
  !  proc            A counter variable for looping over processes.
  integer :: proc_id, n_proc, err
  integer :: chunksize
```

```fortran
57    integer , allocatable :: loop_min(:), loop_max(:), chunksize_proc(:)
58    real , allocatable :: x_proc(:)
59    integer :: proc
60
61    ! Timing variables.
62    !  times       An array storing time markers used to determine the following
63    !              timing variables.
64    !  time_setup  Time taken for this process to setup MPI variables for
65    !              partitioning data.
66    !  time_comp   Time taken for this process to perform mandelbrot calculations
67    !              for its given data subset.
68    !  time_wait   Time this process spends waiting while other processes finish
69    !              performing their calculations.
70    !  time_comm   Time taken for this process to communicate its data subset
71    !              to the root process.
72    !  time_total  Time taken overall.
73    double precision :: times(1:5)
74    double precision :: time_setup, time_comp, time_wait, time_comm, time_total
75
76    ! Read command line arguments.
77    call read_input(N, maxiter)
78
79    allocate(x(0:N*N-1))
80
81    ! MPI initialisation.
82    call MPI_INIT(err)
83
84    times(1) = MPI_WTIME()
85
86    call MPI_COMM_RANK(MPI_COMM_WORLD, proc_id, err)
87    call MPI_COMM_SIZE(MPI_COMM_WORLD, n_proc, err)
88
89    ! Determine (default) chunk size.
90    chunksize = ceiling(real(N*N) / real(n_proc))
91
92    ! Determine loop bounds and chunk size for each process.
93    allocate(loop_min(0:n_proc-1))
94    allocate(loop_max(0:n_proc-1))
95    allocate(chunksize_proc(0:n_proc-1))
96
97    do proc = 0, n_proc - 1
98      loop_min(proc) = max(0, chunksize * proc)
99      loop_max(proc) = min(N*N-1, chunksize * (proc + 1) - 1)
100     chunksize_proc(proc) = loop_max(proc) - loop_min(proc) + 1
101   end do
102
103   ! Allocate work array for given process.
104   allocate(x_proc(0:chunksize_proc(proc_id)-1))
105
106   times(2) = MPI_WTIME()
107
108   ! Mandelbrot calculation.
109   ! Modified to loop only over a certain subset of indexes (due to utilising
110   ! static decomposition).
111   do loop = loop_min(proc_id), loop_max(proc_id)
112     ! i varies from 0 to N-1
113     i = int(loop/N)
114
```

11

```fortran
115      ! j varies from 0 to N-1
116      j = mod(loop, N)
117      kappa = cmplx((4.0*(i-N/2)/N), (4.0*(j-N/2)/N))
118
119      k = 1
120      z = kappa
121      do while (abs(z) <= 2 .and. k < maxiter)
122        k = k+1
123        z = z*z + kappa
124      end do
125
126      ! x_proc is indexed to ensure x_proc(0:chunksize_proc(proc_id)-1) are
127      ! determined.
128      x_proc(loop-loop_min(proc_id)) = log(real(k))/log(real(maxiter))
129    end do
130
131    times(3) = MPI_WTIME()
132
133    ! MPI wait for all process to finish.
134    call MPI_BARRIER(MPI_COMM_WORLD, err)
135
136    times(4) = MPI_WTIME()
137
138    ! MPI gather the work array from each process to the root process.
139    call MPI_GATHERV(x_proc, chunksize_proc(proc_id), MPI_REAL, x, &
140        chunksize_proc, loop_min, MPI_REAL, 0, MPI_COMM_WORLD, err)
141
142    times(5) = MPI_WTIME()
143
144    ! Timing analysis.
145    time_setup = times(2) - times(1)
146    time_comp = times(3) - times(2)
147    time_wait = times(4) - times(3)
148    time_comm = times(5) - times(4)
149    time_total = times(5) - times(1)
150
151    if (proc_id == 0) then
152      write (*, *) &
153          "timing for MPI static code:", NEW_LINE('a'), &
154          "  total: ", time_total
155
156      write (*, *) "time spent working/waiting/communicating:"
157    end if
158
159    call MPI_BARRIER(MPI_COMM_WORLD, err)
160
161    write (*, '(a, i1, a, f5.2, a, f5.2, a, f5.2, a)') &
162        "  ", proc_id, ": ", &
163        100.0*time_comp/time_total, " % / ", &
164        100.0*time_wait/time_total, " % / ", &
165        100.0*time_comm/time_total, " %"
166
167    ! Write timing data to an output file.
168    call write_timing_data (N, maxiter, n_proc, proc_id, &
169        time_setup, time_comp, time_wait, time_comm, time_total)
170
171    call MPI_BARRIER(MPI_COMM_WORLD, err)
172
```

```fortran
173    ! Writing data to file (only done by root process).
174    if (proc_id == 0) then
175
176      write (*, *) "Writing mandelbrot_static.ppm"
177
178      open(7, file="mandelbrot_static.ppm", status="unknown")
179
180      write(7, 100) "P3", N, N, 255
181
182      do loop = 0, N*N-1
183        if (x(loop) < 0.5) then
184          green = 2.0*x(loop)*255
185          write(7, 110) 255-green, green, 0
186        else
187          blue = 2.0*x(loop)*255 - 255
188          write(7, 110) 0, 255-blue, blue
189        end if
190      end do
191
192 100 format(A2, /, I4, I5, /, I3)
193 110 format(I3, /, I3, /, I3)
194
195      close(7)
196
197    end if
198
199    ! Deallocate
200    deallocate(loop_min)
201    deallocate(loop_max)
202    deallocate(chunksize_proc)
203    deallocate(x_proc)
204    deallocate(x)
205
206    ! MPI finalisation.
207    call MPI_FINALIZE(err)
208
209 contains
210
211    ! Read in the value of N, maxiter from command line arguments
212    subroutine read_input (N, maxiter)
213      integer , intent(out) :: N, maxiter
214      integer :: num_args
215      character(len=20) :: arg
216
217      num_args = command_argument_count()
218
219      if (num_args < 2) then
220        write (*, *) "Usage: <N> <maxiter>"
221      end if
222
223      if (num_args >= 1) then
224        call get_command_argument(1, arg)
225        read (arg, *) N
226      else
227        write (*, *) "<N> not specified. Using default value, N = 2000."
228        N = 2000
229      end if
230
```

```fortran
231      if (num_args >= 2) then
232        call get_command_argument(2, arg)
233        read (arg, *) maxiter
234      else
235        write (*, *) &
236            "<maxiter> not specified. Using default value, maxiter = 1000."
237        maxiter = 1000
238      end if
239
240    end subroutine read_input
241
242    ! Write the timing data (for the given parameters: N, maxiter, chunksize,
243    ! n_proc), for a given process.
244    !
245    ! The timing data includes the time taken spent: setting up, communicating,
246    ! performing computations, waiting, and the total time spent.
247    !
248    ! The filename is defined by (N, maxiter, n_proc, proc_id)
249    subroutine write_timing_data (N, maxiter, n_proc, proc_id, &
250        time_setup, time_comp, time_wait, time_comm, time_total)
251      integer , intent(in) :: N, maxiter, proc_id
252      double precision , intent(in) :: time_setup, time_comp, time_wait, &
253          time_comm, time_total
254      character(len=1000) :: timing_file
255      character(len=20) :: str_N, str_maxiter, str_n_proc, str_proc_id
256      integer :: file_unit
257
258      ! Construct timing filename to be of the form:
259      ! "output/timing.static.N=<N>.maxiter=<maxiter>.n_proc=<n_proc>\
260      ! .proc_id=<proc_id>.dat"
261      write (str_N, *) N
262      write (str_maxiter, *) maxiter
263      write (str_n_proc, *) n_proc
264      write (str_proc_id, *) proc_id
265
266      write (timing_file, *) &
267          "output/timing.static.", &
268          "N-", trim(adjustl(str_N)), ".", &
269          "maxiter-", trim(adjustl(str_maxiter)), ".", &
270          "n_proc-", trim(adjustl(str_n_proc)), ".", &
271          "proc_id-", trim(adjustl(str_proc_id)), ".dat"
272
273      ! Append the timing data to the data file
274      file_unit = 10 + proc_id
275
276      open (file_unit, file=trim(adjustl(timing_file)), action="write")
277
278      write (file_unit, *) time_setup, time_comp, time_wait, &
279          time_comm, time_total
280
281      close (file_unit)
282
283    end subroutine write_timing_data
284
285 end program mandelbrot_static
```

14

A.3   Master Worker Scheme

```fortran
!> mandelbrot_master_worker
!
!  Adaption of Mandelbrot program to use MPI with master_worker parallelism.
!
!  For a NxN grid (represented in a 1D array; that is, {0, .., N*N - 1}),
!  with n_proc processes, and a specified chunksize.
!
!  The data is broken up into contiguous subsets, with size chunksize, indexed
!  by an integer. Each subset defined by
!  > {loop_min(task), ..., loop_max(task)} for task = 1, ..., n_task.
!
!  The master process will distribute integers to the worker processes,
!  representing which task they are to be working on. The master process will
!  retain a ledger of which task each worker process is working on. If a worker
!  process is not working on any task, this will be recorded in the ledger as
!  task_ledger(proc) = no_task := 0.
!  Initially, all worker processes are assigned a task to work on. The master
!  thread will then wait for the workers to return the Mandelbrot data, and if
!  there are more tasks left to work on, distribute them to the idle workers.
!  When there are no more tasks left to work on, the master process will then
!  switch to collecting unfinished tasks, and telling the workers that return
!  them that there is no more to do. When all outstanding tasks have been
!  collected, the master thread finishes the master-worker scheme.
!
!  The worker processes will first receive a logical flag from the master thread
!  indicating if there is any work to do. If there is, they will then receive an
!  integer indicating which task (that is, which subset of the data) they will
!  work on. After they have performed the Mandelbrot calculation for that subset
!  of data, they will send the Mandelbrot data back to the master thread and
!  wait for its response. When the master thread accepts the workers data, it
!  will tell the worker if there is more work to do, and if there is, send it
!  another task to perform. When there are no more tasks to work on, the worker
!  will finish.
program mandelbrot_master_worker

  use mpi

  complex :: z, kappa
  integer :: green, blue, i, j, k, loop

  integer :: N, maxiter
  real , allocatable :: x(:)

  ! MPI variables.
  !   master_id
  !   proc_id          ID of current MPI process.
  !   n_proc           Number of MPI processes.
  !   err              Stores error code for MPI calls.
  !   tag              MPI tag variable.
  !   request          MPI request variable.
  !   status           MPI status variable. Used by the master process to
  !                    determine the proc_id of worker processes returning tasks.
  !   chunksize        The number of loop iterations assigned to a process for
  !                    one task.
  !   n_tasks          The number of tasks.
  !   loop_min         An array of the lower loop-iteration bounds for each
```

```fortran
 57    !                     task.
 58    !    loop_max         An array of the upper loop-iteration bounds for each
 59    !                     task.
 60    !    x_task           A work array, local to each process, which will yield the
 61    !                     mandelbrot data for the data subset assigned to this
 62    !                     worker process. The master thread will use this to copy
 63    !                     across the mandelbrot data returned by the worker
 64    !                     processes.
 65    !    task             A counter variable for looping over tasks.
 66    !    proc             A counter variable for looping over processes.
 67    !    proc_recv        The proc_id of the worker process returning a task to the
 68    !                     master process.
 69    !    task_recv        The task completed by the worker process returning a task
 70    !                     to the master process.
 71    !    task_ledger      A record (kept by the master process) of which task each
 72    !                     process is currently working on. If the worker process,
 73    !                     proc, isn't working on any task, then
 74    !                     task_ledger(proc) = no_task.
 75    !    no_task          An integer representing an idle task; that is, no task.
 76    !
 77    !    all_tasks_distributed  A flag indicating whether or not all the tasks
 78    !                     have been distributed amongst the worker process. Once the
 79    !                     master thread has determined that all tasks have been
 80    !                     distributed, it will send the worker threads this flag as
 81    !                     they come to return tasks.
 82    integer , parameter :: master_id = 0
 83    integer :: proc_id, n_proc, err, tag, request
 84    integer :: status(MPI_STATUS_SIZE)
 85    integer :: chunksize, n_tasks
 86    integer , allocatable :: loop_min(:), loop_max(:)
 87    real , allocatable :: x_task(:)
 88    integer :: proc, task
 89    integer :: proc_recv, task_recv
 90    logical :: all_tasks_distributed
 91    integer , allocatable :: task_ledger(:)
 92    integer , parameter :: no_task = 0
 93
 94    ! Timing variables.
 95    !    times        An array storing time markers used to determine the following
 96    !                 timing variables.
 97    !    time_setup   Time taken for this process to setup MPI variables for
 98    !                 partitioning data.
 99    !    time_comp    Time taken for a worker process to perform mandelbrot
100    !                 calculations for its given data subset. For the master thread,
101    !                 this is used to track the time it takes to copy returned data
102    !                 into the final data set.
103    !    time_wait    For a worker process, this measures the time spent waiting for
104    !                 the master process to receive this workers completed task.
105    !                 For the master process, this is not a suitable variable,
106    !                 since time spent waiting is hard to differentiate from time
107    !                 spent communicating.
108    !    time_comm    For a worker process, this measures the time spent
109    !                 communicating with the master thread. For the master thread,
110    !                 this measures the all time spent communicating with the
111    !                 worker process
112    !    time_total   Time taken overall.
113    double precision :: times(1:8)
114    double precision :: time_setup, time_comp, time_wait, time_comm, time_total
```

```
115
116    ! Read command line arguments.
117    call read_input(N, maxiter, chunksize)
118
119    allocate(x(0:N*N-1))
120
121    ! Initialise timing variables
122    time_setup = 0d0
123    time_comp = 0d0
124    time_wait = 0d0
125    time_comm = 0d0
126    time_total = 0d0
127
128    ! MPI initialisation.
129    call MPI_INIT(err)
130
131    times(1) = MPI_WTIME()
132
133    call MPI_COMM_RANK(MPI_COMM_WORLD, proc_id, err)
134    call MPI_COMM_SIZE(MPI_COMM_WORLD, n_proc, err)
135
136    ! Arbitrarily set tag = 0.
137    tag = 0
138
139    ! Determine number of tasks needed for given chunksize.
140    n_tasks = ceiling(real(N*N) / real(chunksize))
141
142    allocate(loop_min(1:n_tasks))
143    allocate(loop_max(1:n_tasks))
144
145    do task = 1, n_tasks
146      loop_min(task) = max(0, chunksize * (task - 1))
147      loop_max(task) = min(N*N-1, chunksize * task - 1)
148    end do
149
150    ! Default value for flag indicating all tasks have been handed out
151    all_tasks_distributed = .false.
152
153    ! Allocate work arrays.
154    allocate(x_task(0:chunksize-1))
155
156    ! Master-Worker Scheme
157    if (proc_id == master_id) then
158      ! Master process.
159
160      ! Allocate process ledger (records which task a given process is working
161      ! on).
162      allocate(task_ledger(1:n_proc-1))
163      task_ledger(:) = no_task
164
165      times(2) = MPI_WTIME()
166      time_setup = times(2) - times(1)
167
168      ! Initialise task counter.
169      task = 1
170
171      ! Distribute initial tasks.
172      do proc = 1, n_proc - 1
```

17

```fortran
173
174        call MPI_ISEND(all_tasks_distributed, 1, MPI_LOGICAL, proc, tag, &
175            MPI_COMM_WORLD, request, err)
176
177      if (all_tasks_distributed) then
178
179        task_ledger(proc) = no_task
180      else
181        call MPI_ISEND(task, 1, MPI_INTEGER, proc, tag, MPI_COMM_WORLD, &
182            request, err)
183
184        task_ledger(proc) = task
185        task = task + 1
186      end if
187
188      all_tasks_distributed = (task > n_tasks)
189    end do
190
191    times(3) = MPI_WTIME()
192    time_comm = time_comm + times(3) - times(2)
193
194    ! Loop until all tasks distributed.
195    do while (.not. all_tasks_distributed)
196      times(4) = MPI_WTIME()
197
198      ! Receive completed task from worker.
199      call MPI_RECV(x_task, chunksize, MPI_REAL, MPI_ANY_SOURCE, tag, &
200          MPI_COMM_WORLD, status, err)
201
202      proc_recv = status(MPI_SOURCE)
203      task_recv = task_ledger(proc_recv)
204
205      times(5) = MPI_WTIME()
206
207      do loop = loop_min(task_recv), loop_max(task_recv)
208        x(loop) = x_task(loop - loop_min(task_recv))
209      end do
210
211      times(6) = MPI_WTIME()
212
213      ! Tell worker there is more work to do.
214      call MPI_ISEND(all_tasks_distributed, 1, MPI_LOGICAL, proc_recv, tag, &
215          MPI_COMM_WORLD, request, err)
216
217      ! Distribute new task.
218      call MPI_ISEND(task, 1, MPI_INTEGER, proc_recv, tag, MPI_COMM_WORLD, &
219          request, err)
220
221      task_ledger(proc_recv) = task
222      task = task + 1
223
224      all_tasks_distributed = (task > n_tasks)
225
226      times(7) = MPI_WTIME()
227
228      time_comp = time_comp + times(6) - times(5)
229      time_comm = time_comm + (times(5) - times(4)) + (times(7) - times(6))
230    end do
```

```
231
232      ! Collect outstanding tasks.
233      do while (any(task_ledger /= no_task))
234        times(4) = MPI_WTIME()
235
236        ! Receive completed task from worker.
237        call MPI_RECV(x_task, chunksize, MPI_REAL, MPI_ANY_SOURCE, tag, &
238            MPI_COMM_WORLD, status, err)
239
240        proc_recv = status(MPI_SOURCE)
241        task_recv = task_ledger(proc_recv)
242
243        times(5) = MPI_WTIME()
244
245        do loop = loop_min(task_recv), loop_max(task_recv)
246          x(loop) = x_task(loop - loop_min(task_recv))
247        end do
248
249        times(6) = MPI_WTIME()
250
251        ! Tell worker there is no more work to do.
252        call MPI_ISEND(all_tasks_distributed, 1, MPI_LOGICAL, proc_recv, tag, &
253            MPI_COMM_WORLD, request, err)
254
255        ! No more work to distribute.
256        task_ledger(proc_recv) = no_task
257
258        times(7) = MPI_WTIME()
259
260        time_comp = time_comp + times(6) - times(5)
261        time_comm = time_comm + (times(5) - times(4)) + (times(7) - times(6))
262      end do
263
264      deallocate(task_ledger)
265
266      times(8) = MPI_WTIME()
267
268      time_total = times(8) - times(1)
269    else
270      ! Worker process.
271
272      times(2) = MPI_WTIME()
273      time_setup = times(2) - times(1)
274
275      ! Receive direction if there are/aren't more tasks to perform.
276      call MPI_RECV(all_tasks_distributed, 1, MPI_LOGICAL, master_id, tag, &
277          MPI_COMM_WORLD, status, err)
278
279      ! Work until no more tasks to be distributed.
280      do while (.not. all_tasks_distributed)
281        times(3) = MPI_WTIME()
282
283        ! Collect task to complete.
284        call MPI_RECV(task, 1, MPI_INTEGER, master_id, tag, MPI_COMM_WORLD, &
285            status, err)
286
287        times(4) = MPI_WTIME()
288
```

```fortran
289        ! Complete task.
290        x_task(:) = 0.0
291        call mandelbrot_calculation(N, maxiter, loop_min(task), loop_max(task), &
292            x_task)
293
294        times(5) = MPI_WTIME()
295
296        ! Return completed task.
297        call MPI_SEND(x_task, chunksize, MPI_REAL, master_id, tag, &
298            MPI_COMM_WORLD, err)
299
300        ! Receive direction if there are/aren't more tasks to perform.
301        call MPI_RECV(all_tasks_distributed, 1, MPI_LOGICAL, master_id, tag, &
302            MPI_COMM_WORLD, status, err)
303
304        times(6) = MPI_WTIME()
305
306        time_wait = time_wait + times(4) - times(3)
307        time_comp = time_comp + times(5) - times(4)
308        time_comm = time_comm + times(6) - times(5)
309     end do
310
311     times(7) = MPI_WTIME()
312
313     time_total = times(7) - times(1)
314   end if
315
316   ! Timing analysis.
317   if (proc_id == master_id) then
318     write (*, *) &
319         "timing for MPI master-worker code, chunksize:", chunksize, &
320         NEW_LINE('a'), &
321         "  total: ", time_total
322
323     write (*, *) "time spent working/waiting/communicating:"
324   end if
325
326   call MPI_BARRIER(MPI_COMM_WORLD, err)
327
328   if (proc_id /= master_id) then
329     write (*, '(a, i1, a, f5.2, a, f5.2, a, f5.2, a)') &
330         "  ", proc_id, ": ", &
331         100.0*time_comp/time_total, " % / ", &
332         100.0*time_wait/time_total, " % / ", &
333         100.0*time_comm/time_total, " %"
334   end if
335
336   ! Write timing data to an output file.
337   call write_timing_data (N, maxiter, chunksize, n_proc, proc_id, &
338       time_setup, time_comp, time_wait, time_comm, time_total)
339
340   ! Writing Mandelbrot data to file (only done by master process).
341   if (proc_id == master_id) then
342
343     write (*, *) "Writing mandelbrot_master_worker.ppm"
344
345     open(7, file="mandelbrot_master_worker.ppm", status="unknown")
346
```

```fortran
347        write(7, 100) "P3", N, N, 255
348
349        do loop = 0, N*N-1
350          if (x(loop) < 0.5) then
351            green = 2.0*x(loop)*255
352            write(7, 110) 255-green, green, 0
353          else
354            blue = 2.0*x(loop)*255 - 255
355            write(7, 110) 0, 255-blue, blue
356          end if
357        end do
358
359 100 format(A2, /, I4, I5, /, I3)
360 110 format(I3, /, I3, /, I3)
361
362    close(7)
363
364    end if
365
366    ! Deallocate
367    deallocate(loop_min)
368    deallocate(loop_max)
369    deallocate(x_task)
370    deallocate(x)
371
372    ! MPI finalisation.
373    call MPI_FINALIZE(err)
374
375 contains
376
377    ! Mandelbrot calculation.
378    subroutine mandelbrot_calculation (N, maxiter, lower_bound, upper_bound, x)
379      integer , intent(in) :: N, maxiter, lower_bound, upper_bound
380      real , intent(out) :: x(0:upper_bound - lower_bound)
381      complex :: z, kappa
382      integer :: loop, k
383
384      do loop = lower_bound, upper_bound
385        ! i varies from 0 to N-1
386        i = int(loop/N)
387
388        ! j varies from 0 to N-1
389        j = mod(loop, N)
390        kappa = cmplx((4.0*(i-N/2)/N), (4.0*(j-N/2)/N))
391
392        k = 1
393        z = kappa
394        do while (abs(z) <= 2 .and. k < maxiter)
395          k = k+1
396          z = z*z + kappa
397        end do
398
399        x(loop-lower_bound) = log(real(k))/log(real(maxiter))
400      end do
401    end subroutine mandelbrot_calculation
402
403    ! Read in the value of N, maxiter, chunksize from command line arguments.
404    subroutine read_input (N, maxiter, chunksize)
```

```fortran
405      integer , intent(out) :: N, maxiter, chunksize
406      integer :: num_args
407      character(len=20) :: arg
408
409      num_args = command_argument_count()
410
411      if (num_args < 3) then
412        write (*, *) "Usage: <N> <maxiter> <chunksize>"
413      end if
414
415      if (num_args >= 1) then
416        call get_command_argument(1, arg)
417        read (arg, *) N
418      else
419        write (*, *) "<N> not specified. Using default value, N = 2000."
420        N = 2000
421      end if
422
423      if (num_args >= 2) then
424        call get_command_argument(2, arg)
425        read (arg, *) maxiter
426      else
427        write (*, *) &
428            "<maxiter> not specified. Using default value, maxiter = 1000."
429        maxiter = 1000
430      end if
431
432      if (num_args >= 3) then
433        call get_command_argument(3, arg)
434        read (arg, *) chunksize
435      else
436        write (*, *) &
437            "<chunksize> not specified. Using default value, chunksize = 100000."
438        chunksize = 100000
439      end if
440
441    end subroutine read_input
442
443    ! Write the timing data (for the given parameters: N, maxiter, chunksize,
444    ! n_proc), for a given process.
445    !
446    ! The timing data includes the time taken spent: setting up, communicating,
447    ! performing computations, waiting, and the total time spent.
448    !
449    ! The filename is defined by (N, maxiter, n_proc, proc_id), and the chunksize
450    ! is included in the data output to allow for comparison of timing with
451    ! varying chunksize.
452    subroutine write_timing_data (N, maxiter, chunksize, n_proc, proc_id, &
453        time_setup, time_comp, time_wait, time_comm, time_total)
454      integer , intent(in) :: N, maxiter, chunksize, proc_id
455      double precision , intent(in) :: time_setup, time_comp, time_wait, &
456          time_comm, time_total
457      character(len=1000) :: timing_file
458      character(len=20) :: str_N, str_maxiter, str_n_proc, str_proc_id
459      integer :: file_unit
460
461      ! Construct timing filename to be of the form:
462      ! "output/timing.master_worker.N=<N>.maxiter=<maxiter>.n_proc=<n_proc>\
```

```fortran
463      ! .proc_id=<proc_id>.dat"
464      write (str_N, *) N
465      write (str_maxiter, *) maxiter
466      write (str_n_proc, *) n_proc
467      write (str_proc_id, *) proc_id
468
469      write (timing_file, *) &
470          "output/timing.master_worker.", &
471          "N-", trim(adjustl(str_N)), ".", &
472          "maxiter-", trim(adjustl(str_maxiter)), ".", &
473          "n_proc-", trim(adjustl(str_n_proc)), ".", &
474          "proc_id-", trim(adjustl(str_proc_id)), ".dat"
475
476      ! Append the chunksize and timing data to the data file
477      file_unit = 10 + proc_id
478
479      open (file_unit, file=trim(adjustl(timing_file)), action="write", &
480          position="append")
481
482      write (file_unit, *) chunksize, time_setup, time_comp, time_wait, &
483          time_comm, time_total
484
485      close (file_unit)
486
487   end subroutine write_timing_data
488
489 end program mandelbrot_master_worker
```

A.4   Cyclic Decomposition

```
1  !> mandelbrot_cyclic
2  !
3  !  Adaption of mandelbrot program to use MPI with a cyclic decomposition of the
4  !  data.
5  !
6  !  For a NxN grid (represented in a 1D array; that is, {0, .., N*N - 1}),
7  !  with n_proc processes.
8  !
9  !  The default chunksize, the size of the data subset to be assigned to each
10 !  process, is defined to be the smallest integer such that
11 !  > chunksize * n_proc >= N*N.
12 !  This is so that even in the case where n_proc doesn't divide N*N, every
13 !  data point is guaranteed to be covered by a process. However, we note that
14 !  the remainder of this program has not been adapted to work in cases where
15 !  n_proc doesn't divide N*N. May lead to segfault errors in such a case.
16 !
17 !  Each process, p = 0, .., n_proc-1, is assigned a section of the data
18 !  {p + (n_proc * k) , k = 0, ..., chunksize-1 }, to perform the mandelbrot
19 !  calculation on.
20 !
21 !  The results of each process are then gathered by the root process, and the
22 !  block-sequential arranged data is recovered from the cyclically arranged
23 !  data.
24 program mandelbrot_cyclic
25
26   use mpi
27
28   complex :: z, kappa
29   integer :: green, blue, i, j, k, loop
30
31   integer :: N, maxiter
32   real , allocatable :: x(:)
33
34   ! MPI variables.
35   !  proc_id         ID of current MPI process.
36   !  n_proc          Number of MPI processes.
37   !  err             Stores error code for MPI calls.
38   !  chunksize       The default chunksize; the maximum possible number of loop
39   !                  iterations assigned to each process. Defined to be the
40   !                  smallest integer such that
41   !                  that chunksize * n_proc >= N*N.
42   !                  Note that this program has not been adapted to work for
43   !                  cases where n_proc does not divide N*N.
44   !  x_proc          A work array, local to each process, which will yield the
45   !                  mandelbrot data for the data subset assigned to this
46   !                  process. Will have size chunksize.
47   !  x_cyclic        A work array, for the root process, which will gather the
48   !                  cyclic data, and from which x(:) can be recovered
49   !  proc            A counter variable for looping over processes.
50   !  l               A counter variable used for clarity when performing cyclic
51   !                  loops.
52   integer :: proc_id, n_proc, err
53   integer :: chunksize
54   real , allocatable :: x_proc(:), x_cyclic(:)
55   integer :: proc, l
56
```

```fortran
57    ! Timing variables.
58    !  times       An array storing time markers used to determine the following
59    !              timing variables.
60    !  time_setup  Time taken for this process to setup MPI variables for
61    !              partitioning data.
62    !  time_comp   Time taken for this process to perform mandelbrot calculations
63    !              for its given data subset.
64    !  time_wait   Time this process spends waiting while other processes finish
65    !              performing their calculations.
66    !  time_comm   Time taken for this process to communicate its data subset
67    !              to the root process.
68    !  time_total  Time taken overall.
69    double precision :: times(1:5)
70    double precision :: time_setup, time_comp, time_wait, time_comm, time_total
71
72    ! Read command line arguments.
73    call read_input(N, maxiter)
74
75    allocate(x(0:N*N-1))
76
77    ! MPI initialisation.
78    call MPI_INIT(err)
79
80    times(1) = MPI_WTIME()
81
82    call MPI_COMM_RANK(MPI_COMM_WORLD, proc_id, err)
83    call MPI_COMM_SIZE(MPI_COMM_WORLD, n_proc, err)
84
85    ! Determine (default) chunk size.
86    chunksize = ceiling(real(N*N) / real(n_proc))
87
88    ! Allocate work array for given process.
89    allocate(x_proc(0:chunksize-1))
90    allocate(x_cyclic(0:N*N-1))
91
92    times(2) = MPI_WTIME()
93
94    ! Mandelbrot calculation.
95    ! Modified to loop only over a certain subset of indexes (due to utilising
96    ! cyclic decomposition).
97    do l = 0, chunksize-1
98      loop = (n_proc * l) + proc_id
99
100     ! i varies from 0 to N-1
101     i = int(loop/N)
102
103     ! j varies from 0 to N-1
104     j = mod(loop, N)
105     kappa = cmplx((4.0*(i-N/2)/N), (4.0*(j-N/2)/N))
106
107     k = 1
108     z = kappa
109     do while (abs(z) <= 2 .and. k < maxiter)
110       k = k+1
111       z = z*z + kappa
112     end do
113
114     x_proc(l) = log(real(k))/log(real(maxiter))
```

```fortran
115    end do
116
117    times(3) = MPI_WTIME()
118
119    ! MPI wait for all process to finish.
120    call MPI_BARRIER(MPI_COMM_WORLD, err)
121
122    times(4) = MPI_WTIME()
123
124    ! MPI gather the work array from each process to the root process.
125    call MPI_GATHER(x_proc, chunksize, MPI_REAL, x_cyclic, chunksize, MPI_REAL, &
126        0, MPI_COMM_WORLD, err)
127
128    ! Extract the original data from the cyclically distributed data
129    if (proc_id == 0) then
130
131      do loop = 0, N*N-1
132        proc = mod(loop, n_proc)
133        l = (loop - proc) / n_proc
134
135        x(loop) = x_cyclic((chunksize*proc) + l)
136      end do
137
138    end if
139
140    times(5) = MPI_WTIME()
141
142    ! Timing analysis.
143    time_setup = times(2) - times(1)
144    time_comp = times(3) - times(2)
145    time_wait = times(4) - times(3)
146    time_comm = times(5) - times(4)
147    time_total = times(5) - times(1)
148
149    if (proc_id == 0) then
150      write (*, *) &
151          "timing for MPI cyclic code:", NEW_LINE('a'), &
152          "  total: ", time_total
153
154      write (*, *) "time spent working/waiting/communicating:"
155    end if
156
157    call MPI_BARRIER(MPI_COMM_WORLD, err)
158
159    write (*, '(a, i1, a, f5.2, a, f5.2, a, f5.2, a)') &
160        "  ", proc_id, ": ", &
161        100.0*time_comp/time_total, " % / ", &
162        100.0*time_wait/time_total, " % / ", &
163        100.0*time_comm/time_total, " %"
164
165    ! Write timing data to an output file.
166    call write_timing_data (N, maxiter, n_proc, proc_id, &
167        time_setup, time_comp, time_wait, time_comm, time_total)
168
169    call MPI_BARRIER(MPI_COMM_WORLD, err)
170
171    ! Writing data to file (only done by root process).
172    if (proc_id == 0) then
```

```fortran
173
174      write (*, *) "Writing mandelbrot_cyclic.ppm"
175
176      open(7, file="mandelbrot_cyclic.ppm", status="unknown")
177
178      write(7, 100) "P3", N, N, 255
179
180      do loop = 0, N*N-1
181        if (x(loop) < 0.5) then
182          green = 2.0*x(loop)*255
183          write(7, 110) 255-green, green, 0
184        else
185          blue = 2.0*x(loop)*255 - 255
186          write(7, 110) 0, 255-blue, blue
187        end if
188      end do
189
190 100 format(A2, /, I4, I5, /, I3)
191 110 format(I3, /, I3, /, I3)
192
193      close(7)
194
195    end if
196
197    ! Deallocate
198    deallocate(x_proc)
199    deallocate(x_cyclic)
200    deallocate(x)
201
202    ! MPI finalisation.
203    call MPI_FINALIZE(err)
204
205 contains
206
207    ! Read in the value of N, maxiter from command line arguments
208    subroutine read_input (N, maxiter)
209      integer , intent(out) :: N, maxiter
210      integer :: num_args
211      character(len=20) :: arg
212
213      num_args = command_argument_count()
214
215      if (num_args < 2) then
216        write (*, *) "Usage: <N> <maxiter>"
217      end if
218
219      if (num_args >= 1) then
220        call get_command_argument(1, arg)
221        read (arg, *) N
222      else
223        write (*, *) "<N> not specified. Using default value, N = 2000."
224        N = 2000
225      end if
226
227      if (num_args >= 2) then
228        call get_command_argument(2, arg)
229        read (arg, *) maxiter
230      else
```

```fortran
231         write (*, *) &
232             "<maxiter> not specified. Using default value, maxiter = 1000."
233         maxiter = 1000
234       end if
235
236   end subroutine read_input
237
238   ! Write the timing data (for the given parameters: N, maxiter, chunksize,
239   ! n_proc), for a given process.
240   !
241   ! The timing data includes the time taken spent: setting up, communicating,
242   ! performing computations, waiting, and the total time spent.
243   !
244   ! The filename is defined by (N, maxiter, n_proc, proc_id)
245   subroutine write_timing_data (N, maxiter, n_proc, proc_id, &
246       time_setup, time_comp, time_wait, time_comm, time_total)
247     integer , intent(in) :: N, maxiter, proc_id
248     double precision , intent(in) :: time_setup, time_comp, time_wait, &
249         time_comm, time_total
250     character(len=1000) :: timing_file
251     character(len=20) :: str_N, str_maxiter, str_n_proc, str_proc_id
252     integer :: file_unit
253
254     ! Construct timing filename to be of the form:
255     ! "output/timing.cyclic.N=<N>.maxiter=<maxiter>.n_proc=<n_proc>\
256     ! .proc_id=<proc_id>.dat"
257     write (str_N, *) N
258     write (str_maxiter, *) maxiter
259     write (str_n_proc, *) n_proc
260     write (str_proc_id, *) proc_id
261
262     write (timing_file, *) &
263         "output/timing.cyclic.", &
264         "N-", trim(adjustl(str_N)), ".", &
265         "maxiter-", trim(adjustl(str_maxiter)), ".", &
266         "n_proc-", trim(adjustl(str_n_proc)), ".", &
267         "proc_id-", trim(adjustl(str_proc_id)), ".dat"
268
269     ! Append the timing data to the data file
270     file_unit = 10 + proc_id
271
272     open (file_unit, file=trim(adjustl(timing_file)), action="write")
273
274     write (file_unit, *) time_setup, time_comp, time_wait, &
275         time_comm, time_total
276
277     close (file_unit)
278
279   end subroutine write_timing_data
280
281 end program mandelbrot_cyclic
```