Various parallel programmes using MPI in Fortran

Nilas Mandrup Hansen, Ask Hjorth Larsen

January 27, 2010

1 Speed-Up due to Pipelining

We define the speed-up as $s = T/T_{pipe}$. In the case where the is no pipelining the time it would take to do n operations would be $T = 4n+1$ (the last operation is used only to flush the last result out of the buffer; this is largely a question of definitions) while when using pipelining it is $T_{pipe} = 4 + n + 1$ meaning that the speed-up can be written as follows;

$$S = \frac{4n+1}{n+5} \tag{1}$$

where the latency, the cycles taken before results start coming out, would be 4.

2 Timing and Optimization

In this small exercise, wall-clock times for different types of loops are to be tested. The types are normally coded loops, unrolled loops, interchanged loops and split loops. Also different compiler optimizations are tested.

a, Here a normally coded loop and a partly unrolled loop are compared. The source code can be seen in appendix A in the subroutine `ex2`. Table 1 shows the results from the normally coded version and the partly unrolled version compiled with different compiler optimizations. For the version with no optimization, the

	-O0	-O1	-O2	-O3
Normally coded	3.589	2.223	2.016	0.004
Unrolled	1.716	1.223	0.856	0.049
Time reduction	−52%	−45%	−58%	+965%

Table 1: Timing Results (in seconds). Normal loop vs. Unrolled loop.

unrolled version of the loop is much faster than the normally coded version. With the O1 option on for the compiler, the unrolled version is again faster than the normal version, but less so than in the case with no optimization. This

could indicate that the compiler itself creates some kind of unrolled version when compiling. With option O2, the unrolled version is the fastest, while for the O3 option the normally coded loop becomes the fastest. Evidently the normally coded loop is simpler for the compiler to figure out, so it is capable of optimizing it more aggressively.

b, Here a triply nested loop is being timed and compared to a version where the loop nesting order is reversed. The loop in its ordinary version calls the Fortran built-in function `real` on the *outermost* loop variable, but stores this in a 3D array with leading dimension corresponding *also* to the outermost loop. Thus, in the ordinary version, it is easy to reduce the number of integer-to-real conversions, but the array is accessed contrary to its memory layout. The reversed version of the loop accesses the array in memory-contiguous order, but needs to call the `real` function many more times.

The source code can be seen in appendix A in the subroutine `ex3a`. The results are shown in table 2. The reversed version is by far the fastest version.

	-O0	-O1	-O2	-O3
Nested loop	1.851	1.791	1.819	0.148
Reversed	0.609	0.473	0.450	0.074
Time reduction	67%	−74%	−75%	−50%

Table 2: Timing Results (in seconds). Nested loop vs its reversely nested loop.

It can be seen that the reversed version becomes more efficient when compiler optimizations are used. However the O3 optimization considerably reduces the computation time for both the nested loop and the reverse.

As a final part of the question, a loop containing an `IF`-statement and its split version, without an `IF`-statement are to be compared. The source code can be seen in appendix A in the subroutine `ex3b`. The results are shown in table 3. The split version is the fastest version when only compiler option O0 or O1

	-O0	-O1	-O2	-O3
Loop with `IF`-statement	2.412	1.916	1.463	0.0067
Split version	1.753	1.238	1.551	0.267
Time reduction	27%	−35%	+6%	+3885%

Table 3: Timing Results (in seconds). Loop with an `IF`-statement vs its split version.

is used, while the loop with the `IF`-statement is the fastest if compiler option O2 and O3 are used.

3 Loop Vectorization

Here we consider the possibility of vectorizing two versions of a `DO`-loop. See exercise text. The two do loops do not perform the same operation and they do not give the same result. The second `DO`-loop can be vectorized as follows;

$$a(2:n) = b(1:n-1) + d(2:n) \quad (2)$$
$$d(2:n) = a(1:n-1) - d(2:n) \quad (3)$$

The first `DO`-loop can not be vectorized, as values in one iteration depend directly on values from the immediately preceding iteration.

For the last part of the question we should explain how the Intel compiler can vectorize a particular loop with the cyclic dependency, where the dependency goes p iterations back. Our guess is that the Intel compiler does the `n` operations in vectors of size p. If one would show the execution time versus the parameter

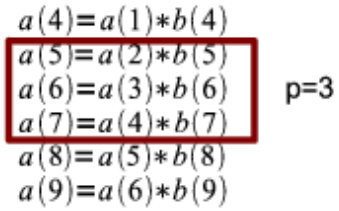

Figure 1: Principle

p, we think it would have the form;

$$t = \alpha \frac{1}{p} \quad (4)$$

4 Parallel World with MPI

Here our first MPI program saying "Hello World" is created. The source code can been seen in appendix B.

The MPI library is used by specifying the `include ''mpif.h''`. Then the parallel environment is established with the `MPI_init` command. To get the number of processors and the *rank* of each processor the commands `MPI_rank` and `MPI_size` is used. The command `MPI_Barrier` is used to ensure that all threads has done the `do` before the another cycle begins. To finalize the MPI program the command `MPI_Finalize` is used.

The results is shown below.

```
 Hello world 0 4
Hello world 1 4
Hello world 2 4
Hello world 3 4
```

5 MPI Ping-Pong

In this question a MPI ping-pong program is written. A message is sent from process 0 to process 1. When the process 1 have received the message, its sends back the message to process 0 (ping-pong). This process is timed when completed a 1000 times.

First a single message is sent having a size of 16 byte. The time its takes to send this message from process 0 to process 1 is $8.188E-3s$ (Ping) and the time its takes to send the message back from process 1 to process 0 is $7.907E-3s$ (Pong). For the record it should be mentioned that the ping-pong is run on the Bohr-cluster. The estimated bandwidth is given as (message size/transfer time);

$$\text{Bandwidth} = \frac{\text{Messagesize}}{\text{transfertime}} = \frac{16byte}{0.008s \cdot 1000} \approx 2MB/sek. \tag{5}$$

From the preceding equation an estimated bandwidth of approximately $2MB/sek.$ can be found.

Next the message size is increased incrementally up to a final message size of approximately 2MB. The results is shown in Fig. 2.

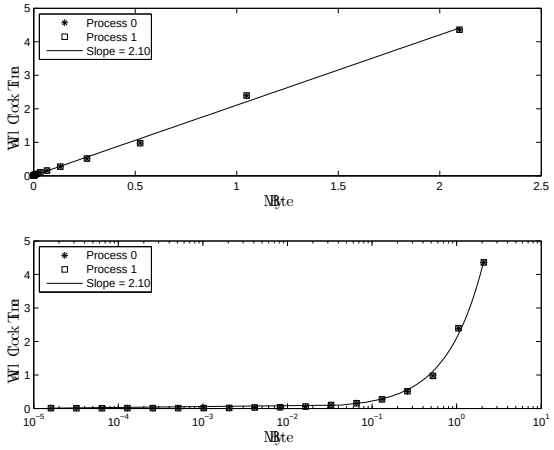

Figure 2: Timing results.

It can be seen that the wall clock time scales nearly linear with a slope of 2.1 which means that the approximate bandwidth is $2.1MB/sek.$.

The source code can be seen in appendix C.

6 Non-blocking communication

In this exercise the sum of the ranks of a set of processes enclosed by a ring is to be calculated. In order to avoid deadlocking the method **Issend-Recv-Wait** is used.

The output from a run on 8 processors produces the following output.

```
2 issendrecvwait 28
3 issendrecvwait 28
5 issendrecvwait 28
4 issendrecvwait 28
1 issendrecvwait 28
6 issendrecvwait 28
0 issendrecvwait 28
7 issendrecvwait 28
```

From the preceeding that the result is 28 from all processors as expected.

The send-receive method is now replaced with the `MPI_Sendrecv`. the result is shown in the following;

```
5 sendrecv 28
6 sendrecv 28
7 sendrecv 28
0 sendrecv 28
1 sendrecv 28
2 sendrecv 28
3 sendrecv 28
4 sendrecv 28
```

Again the result is 28. Finally the `MPI_Sendrecv_Replace` is used. The advantage is here that one do not have to copy the receive buffer to the send buffer. The resutls is shown in the following;

```
3 sendrecv_replace 28
4 sendrecv_replace 28
5 sendrecv_replace 28
6 sendrecv_replace 28
7 sendrecv_replace 28
0 sendrecv_replace 28
1 sendrecv_replace 28
2 sendrecv_replace 28
```

The source code can be seen in appendix D.

7 Collective communication

Here the sum of all teh ranks is again computed. To do this, the command `MPI_Allreduce` is used. In the present, the program is written such that is processor has its rank stored in a variable. The `MPI_Allreduce` reads this variable from each processor and sum all the values, since it is given the argument `MPI_SUM`. The results is then distributed to all processors. The output is shown below.

```
4 allreduce 28
6 allreduce 28
0 allreduce 28
2 allreduce 28
1 allreduce 28
5 allreduce 28
3 allreduce 28
7 allreduce 28
```

Result is 28.

To produce the $i!$ of $i = 1$ the `MPI_Scan` is used. The function basiccaly scans each processor for a given value and the carries that value with it to the next processor where it calculates the product using the `MPI_PROD`. The result is shown below;

```
5 720 8
7 40320 8
2 6 8
6 5040 8
1 2 8
4 120 8
0 1 8
3 24 8
```

It can be seen that processor 0 returns 1, processor 1 returns 2 and processer 2 returns 6 as expected. Finally the results from each processor is collected on a single processor which do an I/O.

```
Collected 1 2 6 24 120 720 5040 40320
```

8 Monte Carlo computation of π

In the exercise, the quantity π is to be found by the Monte Carlo method of integration. A unit circle is tightly enclosed by a square width is 2 meaning that the area of the square is 4, while the area of the unit circle is π.

The square is filled with randomly placed seeds. Some will be placed outside

the unit circle, and some will be placed within the unit circle. The ratio $\hat{\pi}$ will be given as follows;

$$\hat{\pi} = 4 \cdot \frac{n_{ni}}{n} \tag{6}$$

where $\hat{\pi} \overset{n\to\infty}{\to} \pi$ as the number of seeds is increased. To produce a set of random number from 0 to 1, the Fortran random number generator is used. A total of 2000 seeds is used for the computation.
When run on multiple processors, the command `MPI_reduce` is used to collect the numbers from each processor to a single value. An output example is shown below for a MPI run on 8 processors.

```
0 3.104
1 3.09
3 3.146
4 3.138
5 3.16
2 3.136
6 3.214
7 3.152
Total 3.1425
```

where the first column is the rank and the second column is the value π from each processor. The source code can be seen in appendix E.

9 Poisson solver using red-black Gauss-Seidel method

We consider the Poisson equation

$$\nabla^2 u(\mathbf{r}) = f(\mathbf{r}) \tag{7}$$

in the 2D square domain $\mathbf{r} \in]0;1[\times]0;1[$ with some arbitrary boundary conditions, say $u(0,y) = u(1,y) = 1$ and $u(x,0) = u(x,1) = 0$. We wish to implement a parallel Fortran routine using MPI which solves this on a real-space grid $u_{i,j}$ by repeatedly performing the operation

$$u_{i,j} \leftarrow \frac{1}{4}(u_{i-1,j} + u_{i+1,j} + u_{i,j-1} + u_{i,j+1} + h^2 f_{i,j}). \tag{8}$$

Depending on the order in which neighbouring elements are updated in this fashion, the above scheme can be applied in several different ways. Here we shall use the red-black Gauss-Seidel method.

Points are classified as "red" or "black" in a checkerboard pattern, such that each point has four neighbours of the opposite colour. First all red points are updated as per Eq. (8), requiring knowledge of only the black points. Then the

black points are updated, using the newly calculated red values. Each point is thus updated once by performing two separate *sweeps* of the array.

To do this in parallel, we assign each CPU to be responsible for a distinct subdomain, such that the subdomains are square (for simplicity) and equally large. The edge points for one CPU will then, during a given sweep, depend on values for which the neighbouring CPUs are responsible. Every CPU therefore has to send and receive information to and from each of its four neighbours in the grid, with the exception of those CPUs that are on the domain boundary.

9.1 Implementations and performance

We have implemented a main loop which performs one "red" sweep and one "black" sweep per overall iteration, where each sweep is governed by the following code:

```
call recv_buffers_nonblocking(rbuf, ranks, M, comm, reqs)
call update_local_array(U, M, M, h2, eps, color)
call copy_array_to_buffers(U, M, M, sbuf)
call send_buffers_blocking(ranks, sbuf, M, comm)
call wait_for_requests(ranks, reqs)
call copy_buffers_to_array(U, M, M, rbuf)
```

- First a non-blocking receive is started with a reception buffer `rbuf` (`M` is the size of the array which must for simplicity be square).
- Then the local array slice `U` is updated according to the usual procedure (the `color` parameter determines the checkerboard offset, `h2` is the squared grid-spacing and `eps` is the residual which is integrated dynamically).
- After the update, the array edges are copied out into a send buffer (`sbuf`) and sent to the neighbours in a blocking manner (which is far from optimal; the parallelization only helps to the extent that the processes do not finish the previous step equally fast. This will be improved below).
- Finally the receive buffer is copied back into the array.

Figure 3 shows a speedup plot for the Poisson solver.

It scales to some extent, although less than perfectly due to the communication. We have previously theorized about another reason for bad scaling, namely that the test computer is in fact a shared-memory machine, which means that a sufficiently large number of CPUs can exhaust the main memory bandwidth if they are loading large arrays (that do not fit in the caches) at the same time.

The latter we cannot do anything about, but for smaller matrices in particular, performance should improve by using a better non-blocking method. Our plan is to update only the boundary points at first, then send the boundary values in a non-blocking manner, *then* update the remaining interior points, and finally copy the by hopefully received buffers back into the array.

Improved main loop for one sweep:

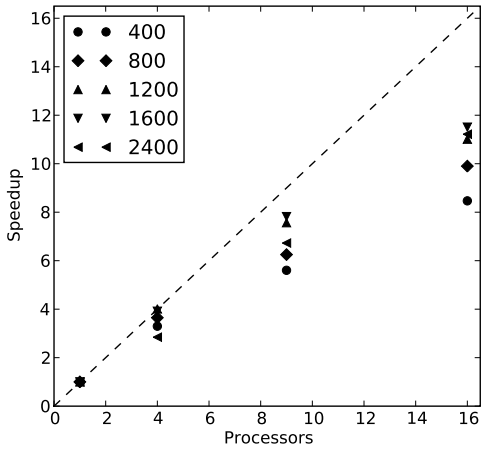

Figure 3: Speedup of first Poisson solver for different CPU counts and matrix sizes

```
call update_boundaries_and_copy_to_buffers(U, M, h2, eps, color, sbuf)
call send_buffers_nonblocking(sbuf, M, ranks, comm, reqs)
call update_interior(U, M, h2, eps, color)
call recv_buffers_blocking(rbuf, M, ranks, comm)
call wait_for_requests(ranks, reqs)
call copy_buffers_to_array(U, M, M, rbuf)
```

Figure 4 shows the speedup for the improved non-blocking scheme. It turns out that this does scale better: for 16 CPUs, the previous method had a speedup of 8 to 12 depending on matrix size, whereas the new one reaches 9.5 to 13. The tendency is more favourable for smaller matrix sizes, consistently with the theory of memory bandwidth exhaustion. We have observed that the variation in serial speed due to the extra looping over boundary points does not contribute measurably to calculation time.

10 Parallel matrix multiplication

We would like to perform the matrix multiplication $\mathbf{AB} = \mathbf{C}$ in a distributed manner using the Fox algorithm with MPI.

Let us say that we have p processes and, for simplicity, that p is a square number such that we can have a quadratic process grid of $b \times b$. We would then like to allocate the matrices $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ in a blocked manner: Each CPU has one block of $\mathbf{A}$ and $\mathbf{B}$, and is responsible for calculating one block of $\mathbf{C}$ (which means it will have to obtain data from the other blocks of $\mathbf{A}$ and $\mathbf{B}$ by communicating with the other processes). We denote the (p, q)'th block by $\mathbf{A}^{pq}$

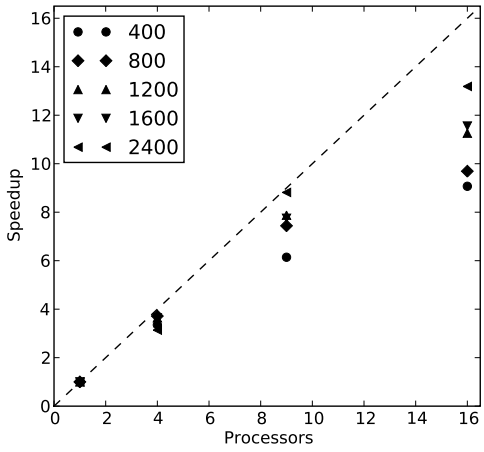

Figure 4: Speedup of improved Poisson solver. Consistently better scaling than the previous implementation.

and its (i, j)'th element by a_{ij}^{pq}. The blocks contain, in total, all elements of $\mathbf{A}$ exactly once.

Suppose we have a 2 by 2 process grid. As an example, the element c_{23} of a 4 by 4 square matrix product can be written down in the usual ("global") way, or in terms of 2 by 2 blocks of 2 by 2 elements, respectively:

$$c_{23} = a_{20}b_{03} + a_{21}b_{13} + a_{22}b_{23} + a_{23}b_{33} \tag{9}$$

$$c_{23} = c_{01}^{11} = a_{00}^{10}b_{01}^{01} + a_{01}^{10}b_{11}^{01} + a_{00}^{11}b_{01}^{01} + a_{01}^{11}b_{11}^{01}. \tag{10}$$

As an example, for 2 by 2 processes, block (1, 1) of the product is

$$\mathbf{C}^{11} = \mathbf{A}^{10}\mathbf{B}^{01} + \mathbf{A}^{11}\mathbf{B}^{11} \tag{11}$$

While process (1, 1) already has the (1, 1) slices of $\mathbf{A}$ and $\mathbf{B}$, it will need to receive the (1, 0) and (0, 1) slices of $\mathbf{A}$ and $\mathbf{B}$, respectively, to finish the calculation.

As a more complete example of Fox' algorithm, this demonstrates the multiplication of a 3 by 3 matrix distributed on 9 processors:

$$\begin{array}{lll} \mathbf{C}_{00} = \mathbf{A}_{00}\mathbf{B}_{00} & \mathbf{C}_{01} = \mathbf{A}_{00}\mathbf{B}_{01} & \mathbf{C}_{02} = \mathbf{A}_{00}\mathbf{B}_{02} \\ \mathbf{C}_{10} = \mathbf{A}_{11}\mathbf{B}_{10} & \mathbf{C}_{11} = \mathbf{A}_{11}\mathbf{B}_{11} & \mathbf{C}_{12} = \mathbf{A}_{11}\mathbf{B}_{12} \\ \mathbf{C}_{20} = \mathbf{A}_{22}\mathbf{B}_{20} & \mathbf{C}_{21} = \mathbf{A}_{22}\mathbf{B}_{21} & \mathbf{C}_{22} = \mathbf{A}_{22}\mathbf{B}_{22} \end{array} \tag{12}$$

Thus, processes $i0$, $i1$, and $i2$ all need slice ii of $\mathbf{A}$. Process ii will have to broadcast its slice to those three processes before the step is performed. Aside

from this, only the local slices are used. Borrowing the notation "+ =" from C/C++, the next step is:

$$\begin{array}{lll} \mathbf{C}_{00} += \mathbf{A}_{01}\mathbf{B}_{10} & \mathbf{C}_{01} += \mathbf{A}_{01}\mathbf{B}_{11} & \mathbf{C}_{02} += \mathbf{A}_{01}\mathbf{B}_{12} \\ \mathbf{C}_{10} += \mathbf{A}_{12}\mathbf{B}_{20} & \mathbf{C}_{11} += \mathbf{A}_{12}\mathbf{B}_{21} & \mathbf{C}_{12} += \mathbf{A}_{12}\mathbf{B}_{22} \\ \mathbf{C}_{20} += \mathbf{A}_{20}\mathbf{B}_{00} & \mathbf{C}_{21} += \mathbf{A}_{20}\mathbf{B}_{01} & \mathbf{C}_{22} += \mathbf{A}_{20}\mathbf{B}_{02} \end{array} \tag{13}$$

This time process $(i, i+1)$ (modulo b) broadcasts its slice of $\mathbf{A}$. Also each proces has exchanged its slice of $\mathbf{B}$ with another process. Specifically process (i, j) must have sent its slice to process $(i, j-1)$ (modulo b). To generalize: at each step, the next processor to the "west" in the process grid broadcasts $\mathbf{A}$ along the east-west direction. Then each process performs a block multiplication with the currect local slices. Last, the local slice of $\mathbf{B}$ is sent back "north" in the grid for the next step. For completeness, the final step in a 3 by 3 block multiplication is:

$$\begin{array}{lll} \mathbf{C}_{00} += \mathbf{A}_{02}\mathbf{B}_{20} & \mathbf{C}_{01} += \mathbf{A}_{02}\mathbf{B}_{21} & \mathbf{C}_{02} += \mathbf{A}_{02}\mathbf{B}_{22} \\ \mathbf{C}_{10} += \mathbf{A}_{10}\mathbf{B}_{00} & \mathbf{C}_{11} += \mathbf{A}_{10}\mathbf{B}_{01} & \mathbf{C}_{12} += \mathbf{A}_{10}\mathbf{B}_{02} \\ \mathbf{C}_{20} += \mathbf{A}_{21}\mathbf{B}_{10} & \mathbf{C}_{21} += \mathbf{A}_{21}\mathbf{B}_{11} & \mathbf{C}_{22} += \mathbf{A}_{21}\mathbf{B}_{12} \end{array} \tag{14}$$

10.1 Implementation

This is the interesting part of the source code for the multiplication function:

```
call copy(A, buf2, M)
do step=0,nblocks - 1
   call copy(buf2, A, M)
   root = mod(step + coords(1), nblocks)
   call MPI_Bcast(A, MM, dtype, root, rcomm, err)
   call multiply_add(A, B, C, M)
   call MPI_Sendrecv( &
      B, MM, dtype, ranks(NORTH), 0, &
      buf, MM, dtype, ranks(SOUTH), 0, &
      comm, status, err)
   call copy(buf, B, M)
enddo
```

We perform as many steps as there are blocks, which is why there is a loop. Each step consists of the following operations:

- `A`, `B` and `C` are the `M` by `M` matrices involved in the product. First `A` is copied into a backup buffer (`copy` is a small utility subroutine), whereafter it is broadcast from a particular rank on `rcomm`, a communicator for the processor row.
- The subroutine call `multiply_add` then adds the local slice product of `A` and `B` to `C`.

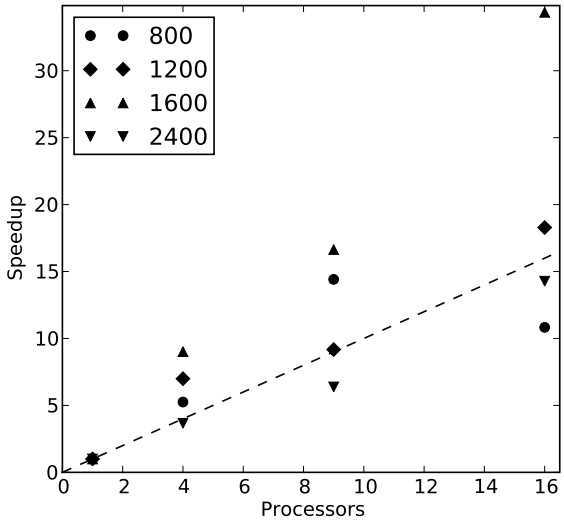

Figure 5: Speedup of Fox algorithm for various matrix sizes.

- Finally the local slice of `B` is propagated along the process grid to the "north" by calling `MPI_Sendrecv`. Another temporary buffer is used for this.

10.2 Timings

Figure 5 shows the speedup of the matrix multiplication algorithm for different matrix sizes and CPU counts. In most cases the speedup is superlinear as the matrix slices fit better into the 8MB L2 cache when using many CPUs. For the 800 by 800 matrix, performance decreases with 16 CPUs as there is little work compared to the amount of synchronization. For larger matrices the multiplication scales very well. The superlinear speedup disappears for the 2400 by 2400 matrix as the matrices never fit well into the cache.

A Source code - Timing and Optimaztion (`main.f90`)

```
1  program timing
2    implicit none
3
4    call ex3b
5  end program timing
6
7  subroutine ex3b()
8    implicit none
9
10   integer :: i, n, j, m=500
11   real(kind=8), dimension(:), allocatable :: a
12   real :: t1, t2
13   n = 100000
14
15   allocate(a(n))
16
17   do i=1, 100000
18      if (mod(i, 10) == 0) then
19         a(i) = 0.0
20      else
21         a(i) = real(i, 8)
22      endif
23   enddo
24
25
26   call cpu_time(t1)
27   do j=1, m
28      do i=1, n
29         if (mod(i, 10) == 0) then
30            a(i) = 0.0
31         else
32            a(i) = real(i, 8)
33         endif
34      enddo
35   enddo
36   call cpu_time(t2)
37   print*, t2 - t1, a(17)
38
39   call cpu_time(t1)
40   do j=1, m
41      do i=1, n
42         a(i) = real(i, 8)
43      enddo
44      do i=10, n, 10
45         a(i) = 0.0
46      enddo
47   enddo
48   call cpu_time(t2)
49   print*, t2 - t1, a(17)
50
51 end subroutine ex3b
52
53 subroutine ex3a()
54   implicit none
55
56   integer :: m, i, j, k, n, n0
57   real :: t1, t2
58   real(kind=8), dimension(:, :, :), allocatable :: f
59
60   m = 200
61
62   allocate(f(m, m, m))
63
64   ! warmup
65   do i=1,m
```

```
66        do j=1,m
67           do k=1,m
68              f(i, j, k) = real(i, 8)
69           enddo
70        enddo
71     enddo
72
73
74     call cpu_time(t1)
75
76     do i=1,m
77        do j=1,m
78           do k=1,m
79              f(i, j, k) = real(i, 8)
80           enddo
81        enddo
82     enddo
83     call cpu_time(t2)
84     print*, t2 - t1
85     print*, f(17, 17, 17)
86
87     call cpu_time(t1)
88     do k=1,m
89        do j=1,m
90           do i=1,m
91              f(i, j, k) = real(i, 8)
92           enddo
93        enddo
94     enddo
95     call cpu_time(t2)
96
97     print*, t2 - t1
98     print*, f(17, 17, 17)
99
100  end subroutine ex3a
101
102  subroutine ex2()
103    implicit none
104
105    real :: t1, t2
106    real :: t
107    integer :: it, i, n
108
109    real(kind=8), dimension(:), allocatable :: a
110
111    n = 100000
112
113    allocate(a(n))
114
115    ! warmup loop
116    do it=1, 100
117       do i=1,n
118          a(i) = real(i, 8)
119       enddo
120    enddo
121
122    call cpu_time(t1)
123
124    do it=1, 1024
125       do i=1,n
126          a(i) = real(i, 8)
127       enddo
128    enddo
129
130    call cpu_time(t2)
131
132    print*, t2 - t1
133
```

```
134     call cpu_time(t1)
135     do it=1, 1024
136        do i=1,n / 8
137           a(i) = real(i, 8)
138           a(i + 1) = real(i + 1, 8)
139           a(i + 2) = real(i + 2, 8)
140           a(i + 3) = real(i + 3, 8)
141           a(i + 4) = real(i + 4, 8)
142           a(i + 5) = real(i + 5, 8)
143           a(i + 6) = real(i + 6, 8)
144           a(i + 7) = real(i + 7, 8)
145        enddo
146     enddo
147     call cpu_time(t2)
148     print*, t2 - t1
149   end subroutine ex2
```

B Source code - Parallel World with MPI (`question4.f90`)

```
 1  program helloworld
 2    include 'mpif.h'
 3
 4    integer :: ierror
 5    integer :: rank, size,r
 6
 7    call MPI_Init(ierror)
 8    call MPI_Comm_rank(MPI_Comm_World, rank, ierror)
 9    call MPI_Comm_size(MPI_Comm_World, size, ierror)
10
11    do r=0,size
12       if (r.EQ.rank) then
13          print*,"Hello world",rank,size
14       endif
15       call MPI_Barrier(MPI_Comm_World,ierror)
16    enddo
17    call MPI_Finalize(ierror)
18
19  end program helloworld
```

C Source code - MPI Ping-Pong (mpi_pingpong.f90)

```
1  program pingpong
2    include "mpif.h"
3    implicit none
4
5    integer :: n, npings
6    integer :: count
7    integer :: err
8
9    call MPI_Init(err)
10
11   do n=2, 19
12      count = 2**n
13      npings = 1000! * 32 / count
14      if (npings.le.4) then
15         npings = 4
16      endif
17      call runpingpong(count, npings)
18   enddo
19
20   call MPI_Finalize(err)
21
22 end program pingpong
23
24 subroutine runpingpong(count, npings)
25   include "mpif.h"
26   implicit none
27
28   integer :: err ! should really check error state but can't be bothered
29   integer :: rank, size
30   integer :: iping, npings
31   integer :: dest ! target rank (0 or 1)
32   integer :: tag = 0
33   integer :: count
34   integer :: comm
35
36   !character, dimension(count) :: sendbuf, recvbuf
37
38   integer, dimension(:), allocatable :: sendbuf, recvbuf
39   integer, dimension(MPI_STATUS_SIZE) :: stat
40   double precision :: t1, t2
41
42   allocate(sendbuf(count), recvbuf(count))
43   sendbuf = 0
44   recvbuf = 0
45
46
47   !character(count) :: sendbuf ! dimension(4)?
48   !character(count) :: recvbuf
49
50   call MPI_Comm_size(MPI_COMM_WORLD, size, err)
51   call MPI_Comm_rank(MPI_COMM_WORLD, rank, err)
52
53   comm = MPI_COMM_WORLD
54
55   if(size.ne.2) then
56      print*, 'Please use size 2'
57      call MPI_Finalize(err)
58      stop
59   endif
60
61   t1 = MPI_Wtime()
62
63   if (rank.eq.0) then
64      sendbuf(1) = 0!'p'
65      sendbuf(2) = 2!'i'
```

```
66       sendbuf(3) = 5!'n'
67       sendbuf(4) = 1!'g'
68       !sendbuf = 'ping'
69       dest = 1
70       !print*, 'Rank:', rank, 'Send:', sendbuf
71       call MPI_Send(sendbuf, count, MPI_INTEGER, dest, tag, MPI_COMM_WORLD,
             err)
72    else
73       !sendbuf = 'pong'
74       sendbuf(1) = 1
75       sendbuf(2) = 42
76       sendbuf(3) = 17
77       sendbuf(4) = 37
78       dest = 0
79       !call MPI_Recv(recvbuf, count, MPI_INTEGER, dest, tag, MPI_COMM_WORLD, &
80       !     stat, err)
81    endif
82
83    do iping = 1, npings - 1
84       call MPI_Recv(recvbuf, count, MPI_INTEGER, dest, tag, comm, stat, err)
85       !print*, 'Rank:', rank, 'Recv:', recvbuf, 'Send:', sendbuf, 'Count:',
             iping
86       call MPI_Send(sendbuf, count, MPI_INTEGER, dest, tag, comm, err)
87    enddo
88
89    if (rank.eq.1) then
90       call MPI_Recv(recvbuf, count, MPI_INTEGER, dest, tag, comm, stat, err)
91       !print*, 'Rank:', rank, 'Recv:', recvbuf, 'Send:', sendbuf
92    endif
93
94    t2 = MPI_Wtime()
95
96    print*, 'Rank', rank, 'Time', t2 - t1, 'Size', count, 'Number', npings
97
98
99 end subroutine runpingpong
```

D Source code - Non-Blocking communication (ranksum.f90)

```
1  program ranksum
2    include "mpif.h"
3    implicit none
4
5    integer :: rank, size, err
6    integer :: comm
7    integer :: sum = 0
8    integer :: sendrq
9
10   integer :: src, dst
11   integer :: i
12
13   integer, dimension(1) :: sendbuf, recvbuf
14   integer, dimension(MPI_STATUS_SIZE) :: stat
15   integer, dimension(:), allocatable :: collectivebuf
16
17   call MPI_Init(err)
18   comm = MPI_COMM_WORLD
19
20   call MPI_Comm_rank(comm, rank, err)
21   call MPI_Comm_size(comm, size, err)
22
23   allocate(collectivebuf(size))
24   collectivebuf = 0
25
```

```
26    dst = mod(rank + 1, size)
27    src = modulo(rank - 1, size)
28
29    sendbuf(1) = rank
30
31    sum = 0
32    do i=1, size
33       call MPI_Issend(sendbuf, 1, MPI_INTEGER, dst, i, comm, sendrq, err)
34       call MPI_Recv(recvbuf, 1, MPI_INTEGER, src, i, comm, stat, err)
35       sum = sum + recvbuf(1)
36       sendbuf(1) = recvbuf(1)
37       call MPI_Wait(sendrq, stat, err)
38    enddo
39    if (rank.ne.recvbuf(1)) then
40       print*, 'BAD'
41    endif
42    print*, rank, 'issendrecvwait', sum
43
44    call MPI_Barrier(comm, err)
45
46    sum = 0
47    do i=1, size
48       call MPI_Sendrecv( &
49            sendbuf, 1, MPI_INTEGER, dst, i, &
50            recvbuf, 1, MPI_INTEGER, src, i, &
51            comm, stat, err)
52       sum = sum + recvbuf(1)
53       sendbuf(1) = recvbuf(1)
54    enddo
55    if (rank.ne.recvbuf(1)) then
56       print*, 'BAD'
57    endif
58    print*, rank, 'sendrecv', sum
59
60    flush(6)
61    call MPI_Barrier(comm, err)
62
63    sum = 0
64    do i=1, size
65       call MPI_Sendrecv_Replace(sendbuf, 1, MPI_INTEGER, dst, i, src, i, comm,
           &
66            stat, err)
67       sum = sum + sendbuf(1)
68    enddo
69    if (rank.ne.sendbuf(1)) then
70       print*, 'BAD'
71    endif
72    print*, rank, 'sendrecv_replace', sum
73
74    ! If we want to use blocking Recv/Send, then we'll have to get
75    ! e.g. even ranks to start with Send, while uneven ones start with Recv.
76    !enddo
77
78    sum = 0
79
80    recvbuf(1) = 0
81    call MPI_Allreduce(sendbuf, recvbuf, 1, MPI_INTEGER, MPI_SUM, comm, err)
82    sum = recvbuf(1)
83    print*, rank, 'allreduce', sum
84
85    sendbuf(1) = rank + 1
86    call MPI_Scan(sendbuf, recvbuf, 1, MPI_INTEGER, MPI_PROD, comm)
87    call MPI_Barrier(comm, err)
88
89    print*, rank, recvbuf, size
90    call MPI_Gather( &
91         recvbuf, 1, MPI_INTEGER, &
92         collectivebuf, 1, MPI_INTEGER, &
```

```
93          0, comm, err)
94    if(rank.eq.0) then
95       print*, 'Collected', collectivebuf
96    endif
97    call MPI_Barrier(comm, err)
98
99    call MPI_Finalize(err)
100   deallocate(collectivebuf)
101 end program ranksum
```

E Source code - Monto Carlo computation of π (montecarlo.f90)

```
1  program montecarlo
2    include "mpif.h"
3    implicit none
4
5    integer :: ni = 0, nx = 0, n, i, nmax = 2000
6    real :: rx, ry
7    real :: pi, tmp
8    real :: sum = 0.0
9
10   integer :: seedsize = 7
11   integer, dimension(:), allocatable :: seed
12
13   integer :: err
14   integer :: rank, size
15
16   real, dimension(1) :: sendbuf
17   real, dimension(1) :: recvbuf
18
19   call MPI_Init(err)
20
21   call MPI_Comm_rank(MPI_COMM_WORLD, rank, err)
22   call MPI_Comm_size(MPI_COMM_WORLD, size, err)
23
24   call random_seed(size=seedsize)
25   allocate(seed(seedsize))
26   do i=1, seedsize
27      seed(i) = 42 + rank * 17
28   enddo
29   call random_seed(put=seed)
30
31   do n=1, nmax
32      call random_number(rx)
33      call random_number(ry)
34      tmp = rx**2 + ry**2
35      if (tmp.le.(1.0)) then
36         ni = ni + 1
37      else
38         nx = nx + 1
39      endif
40   enddo
41
42   pi = 4.0 * real(ni) / (real(ni) + real(nx))
43   print*, rank, pi
44
45   sendbuf(1) = pi
46
47   call MPI_Reduce( &
48        sendbuf, recvbuf, 1, &
49        MPI_REAL, MPI_SUM, 0, MPI_COMM_WORLD, err)
50
51   if (rank.eq.0) then
52      print*, 'Total', recvbuf(1) / size
```

```
53     endif
54
55     call MPI_Finalize(err)
56
57   end program montecarlo
```