CM30225 Parallel Computing
Coursework Assignment 2

James Potter

January 6, 2019

Contents

1 Introduction

This report is explaining the approach, testing and analysis of a distributed program that uses the MPI library. The program uses the relaxation technique to find the solution of differential equations. To do this, we have an initial matrix, and continually average the neighbouring values for every element until they settle within a precision. This kind of problem is well suited to SIMD (single instruction, multiple data) architectures, which is taken into account in the design and implementation phase.

2 Setup

To compile the MPI code on Balena, we use the following command:

```
mpicc -O3 -o dist -std=c99 filename.c -lm
```

The -O3 flag will be discussed as it has a huge impact on the results. However, for all results in this report, the -O3 flag will be used.

To run the MPI code, we use the following command, where numprocs is the total number of processes to start:

```
mpirun -np numprocs ./dist
```

3 Approach

Before starting to program, the approach to the overall architecture had to be considered. A divide and conquer approach was the obvious choice, using the root process to split the data and send it to all the processes, to then be processed and then once settled, the results can all be consolidated and combined in the root process. The first challenge was thinking about how to split up the data. Data in a distributed system needs to be carefully considered, as the latency of sending data between nodes is much greater than between cores on the same chip. Any unnecessary waiting for communication could lower the potential speedup considerably.

My first instinct to try to mitigate the effect of this slow inter-node communication was to, upon splitting up the initial matrix, process and send the edges asynchronously, and perform the rest of the calculation while the communication could take place. This left the question of which method of splitting up the edges would make be both efficient and not overcomplicated. The way I initially thought about splitting up the matrix was in a grid. However, this has the problem of having 4 edges per process that need to be sent, while using rows allows us to only send 2 edges per process. We are trying to avoid communication as much as possible, so this is much better. This was the first architectural decision, splitting the matrix into rows, assigning each process some rows to compute, and sending the top and bottom row to the processes working on the rows above and below it with a non-blocking send. The use of these non-blocking calls is to allow other rows to be worked on while communication takes place – hopefully removing the majority of the communication overhead.

The next problem was how to communicate the settling flag between processes. Communication of settle values needed to be taken with care, as if implemented incorrectly the overhead would be significant. Settling is the biggest overhead, with the completion of computation needed in order to check and send the settled value. The solution used gets every process to send their settled value to every other process in a non-blocking send, with the waits for the send and receive of the top and bottom edges taking place after, and the waits for the settle values being after this. Having the waits in between should lower the overhead, but some solutions could lower the overhead further is discussed in the reflections.

Gathering the processed array from all the processes was the final problem, and I used MPI_Gatherv to do this. MPI_Gatherv takes values from processes, and send them all to one process (in this case the root process, of rank zero). It does this by taking an input from each process of a size, and then on the root process it uses an array of counts (the number of values sent from each rank) to receive the correct number of values, and an array of displacement values (the displacement of where these values should go in the destination array) to know where to put the values in the consolidated array[1].

4 Correctness testing

With the splitting, sending, processing, settling and gathering stages implemented, we now get a consolidated output from our distributed parallel program. It is important to perform extensive testing to ensure the program is working as intended, as many unexpected bugs can occur due to situations that may not occur every time. For example, if some communication happens in multiple processes, it can occur in different orders. This variance can cause bugs that can seem to appear out of nowhere and for no reason – it is important we find these kinds of bugs, as without the rectification of said bugs we have an incorrect program.

There are a few ways to ensure correctness, with the most obvious being to compare it against a hardcoded and hand solved matrix first. Below are a few examples.

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \xrightarrow{\text{With precision of 1}} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 0.9375 & 0.9375 & 1 \\ 1 & 0.9375 & 0.9375 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

[1] https://www.mpich.org/static/docs/v3.1/www3/MPI_Gatherv.html

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix} \xrightarrow{\text{With precision of 2}} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 0.9844 & 0.9844 & 1 \\ 1 & 0.9844 & 0.9844 & 1 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} \xrightarrow{\text{With precision of 1}} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 0.875 & 0.812 & 0.875 & 1 \\ 1 & 0.812 & 0.750 & 0.812 & 1 \\ 1 & 0.875 & 0.812 & 0.875 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} \xrightarrow{\text{With precision of 2}} \begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 0.984 & 0.977 & 0.984 & 1 \\ 1 & 0.977 & 0.969 & 0.977 & 1 \\ 1 & 0.984 & 0.977 & 0.984 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

In addition to hand solving, the Haskell relaxation code appendix A was used to calculate the stages after each loop of relaxation and compared these stages with the output of each stage from the distributed program. The use of interactive compiler GHCi allows composition and execution of functions in Haskell, which lets us test and produce intermediate stages of the relaxation algorithm to compare against. Haskell code is easy to reason about and is suitable for verification purposes. An example of the output to compare with during testing of each step is shown in the excerpts from testing appendix B.

Another way to ensure the program is working correctly, especially for larger matrices where hand solving is unreasonable, is to compare it against the output of a sequential version of the program. Below is the code used for the sequential algorithm, and the implementation was chosen to be as simple as possible, with the row averaging and settling being in one method, the same method used in the distributed program. This method is shown in

Although using the same method for the algorithm we are testing against could prove to be another source of bugs, it also allows us to test an important function stand-alone, where we can ensure its correctness through unit tests and correctness of the final output array. The hard coded correctness tests were also performed on this sequential implementation, to ensure its correctness.

```c
double* relaxSeq(double* workingArr, int dim, double precision) {
    double* avgArr = malloc(sizeof(double) * dim * dim);
    int lastRowIndex = (dim*dim)-dim;
    memcpy(&avgArr[0], &workingArr[0], sizeof(double) * dim);
    memcpy(&avgArr[lastRowIndex], &workingArr[lastRowIndex],
            sizeof(double) * dim);

    int settled = 0;
    while (!settled) {
        settled = 1;
        for (int i = 1; i < dim - 1; i++) {
            averageRow(workingArr, avgArr, dim, i, &settled, precision);
        }
        double* tmp = avgArr;
        avgArr = workingArr;
        workingArr = tmp;
    }
    return workingArr;
}
```

Listing 1: Sequential algorithm to compare against.

With this sequential algorithm confirmed of its correctness, it can now be used to test the distributed program. This testing involved running both the sequential and distributed versions with the same input array and comparing the output arrays. This was done using Linux diff, which prints out the differences from two inputs. The command is shown below and was run for many different arrays. When diffing with results from Balena, the output was changed to the same form, and then the file was diffed.

```
diff <(./sequential) <(mpirun -np 4 dist)
```

With this testing completed, the correctness of the distributed program was verified, and the analysis of the implementation could begin.

5 Speedup and efficiency

To analyse the distributed implementation of the relaxation algorithm, we will turn to Amdahl and Gustafson's laws. The results used in this analysis will all use a starting matrix with ones around the edge, and zeros in the middle, to ensure consistency between results.

Before performing this analysis, however, we must discuss the use of the -O3 compiler flag, as its use has a sizeable impact on our results. This flag enables the highest and most aggressive level of optimisation, and is most beneficial for programs that have loops that perform many floating-point calculations and process large amounts of data[2]. With the flag enabled, relaxation of a matrix with a dimension of 10,000 and a precision of 2, takes 7.524 seconds, while with the flag disabled the same computation takes 47.288 seconds. This is over six times the time for the same computation. Similar results come from adding the flag from the distributed program, although the improvements are not quite as extreme.

Although it might make sense to enable this flag if it has a positive effect, some thoughts come to mind regarding how real-world speedup could be much less than anticipated for specific applications, due to sequential applications being easier to optimise by the compiler. It is because of this that I will use the -O3 flag for all the analysis in this report, which will make the overall speedup seem lower than without it, however, the benefit of using the flag

[2] https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

is a more realistic look at the speed up and efficiency this parallel program would have in the real world.

Moving onto the analysis of the program, I ran the program with varying number of processes, to see the how the speedup and efficiency would be affected.

Number of Processes	Average time (seconds)	Speedup	Efficiency
Sequential	7.524	1	100%
1	8.281	0.908	90.9%
2	5.285	1.424	71.2%
4	3.178	2.368	59.2%
8	2.252	3.341	41.8%
16	2.208	3.408	21.3%
32	4.512	1.668	5.2%
48	3.801	1.979	4.1%
64	3.738	2.013	3.2%

Results from running the distributed program using a dimension of 10,000 and a precision of 2 with a varying number of processes.

Initially, we will focus on speedup. From the difference between the sequential and single process runs, we can see that the overhead of MPI, minus communications, gives us about 9% slowdown. This will become smaller as the problem becomes bigger (relative to the size of the problem) and does not seem like as much overhead as you would expect.

Looking at the other runs, from 1 to 16 processes everything looks normal, with the speedup increasing to 3.4 times as fast as the sequential version at 16 processes. Past 16 processes, we see a notable decrease in speedup. This shows the inter-node communication is a huge bottleneck, and that in order for the use of more than one node to be faster, we need to increase the size of the problem significantly.

The sequentiality of our parallel program consists of sending and receiving the edges from one another, initial startup, sending and receiving settle values and gathering the computation at the end. This is a large amount of sequentiality for a smaller problem, and even with an infinite number of processors, our results show Amdahl's law in action. Efficiency decreases the more processes we add, and the large amount of communication overhead from settling and sending edges becoming apparent once we go above one node, which means no matter how many processors we get, there will always be this sequential overhead.

However, Gustafson's law says the sequential part decreases relative to the size of the problem. So that our initial overhead of 9%, and the slow speed of inter-node communication, although will not disappear, can be made smaller relative to the size of the problem. Below is a table of results using a dimension of 40,000 and precision of 2.

From these results, we can see Gustafson's law in effect, with a greater speedup seen compared

Number of Processes	Average time (seconds)	Speedup	Efficiency
Sequential	110.495	1	100%
8	30.206	3.658	45.7%
16	27.545	4.011	25.1%
32	22.770	4.853	15.2%
48	20.074	5.504	11.5%
64	18.426	5.997	9.4%

Results from running the distributed program using a dimension of 40,000 and a precision of 2 with a varying number of processes.

to the 10,000 dimension array showing the sequential parts of the program becoming less of an issue the larger we make the problem. The table also shows the speedup not decrease beyond 16 cores, showing that we have overcome the communication overhead by making the problem larger, which is in line with Gustafson's law.

6 Reflection

In reflection, I think my implementation is a reasonable solution, with minimal overheads overall. If I were to improve the program, I would be more careful with the sending of settling values, sending asynchronously before the computation, and waiting for the value after the computation. This would remove the bottleneck when the computation got large enough and should be much faster.

I would also look into the possibility of giving a node a number of rows, and then have the communication be as minimal as possible, putting the edges of each nodes working array on the closest nodes available, and so forth. The trouble with trying to over-optimise any program is adding too much overhead, and it would have to be carefully implemented with that in mind.

Appendices

A Haskell parallel relaxation

```haskell
import Data.Vector (Vector)
import qualified Data.Vector.Split as V
import qualified Data.Vector as V
import Control.Monad
import Control.Parallel.Strategies
import Data.Vector.Strategies

type Elem = (Int, Double, Bool)

precision = 10.0 ^^ (-1)

initVec :: Int -> Vector Elem
initVec n = V.zipWith3 (,,) (V.enumFromTo 0 (size-1)) vec (V.replicate size True)
    where
        size         = n*n
        zeros        = V.replicate (n-2) 0.0
        midRow       = V.cons 1.0 $ zeros V.++ V.singleton 1.0
        middle       = V.concat $ replicate (n-2) midRow
        topandbottom = V.replicate n 1.0
        vec          = topandbottom V.++ middle V.++ topandbottom

constIndex :: Int -> Int -> Bool
constIndex i dim = topRow || bottomRow || leftCol || rightCol
    where
       topRow    = i <= (dim-1)
       bottomRow =  i >= (dim*dim) - dim
       leftCol   = (mod i dim) == 0
       rightCol  = (mod i dim) == (dim-1)

avgNeighbors :: Int -> Elem -> Vector Elem -> Elem
avgNeighbors dim (i, v, s) vec = (i, newV, dComp v newV)
    where
        (_, up, _)    = vec V.! (i-dim)
        (_, down, _)  = vec V.! (i+dim)
        (_, left, _)  = vec V.! (i-1)
        (_, right, _) = vec V.! (i+1)
        newV  = (up + down + left + right)/4
        dComp = (\a b -> abs (a-b) < precision)
```

```haskell
getNewElem :: Int -> Vector Elem -> Elem -> Elem
getNewElem dim vec e@(i, _, _) = if constIndex i dim then e
                                 else avgNeighbors dim e vec

relaxVec :: Int -> Vector Elem -> Vector Elem
relaxVec dim vec = if settled updatedVec then updatedVec
                   else relaxVec dim updatedVec
    where
        updatedVec = V.map (getNewElem dim vec) vec
        -- uncomment below for parallel version
        -- updatedVec = (V.map (getNewElem dim vec) vec) `using` (parVector 100)
        -- 100 is the chunk size

settled :: Vector Elem -> Bool
settled = all (\(_, _, s) -> s == True)

prettyPrint :: Int -> Vector Elem -> IO ()
prettyPrint dim vec = do
    let valVec   = V.map (\(_, v, _) -> v) vec
        splitVec = V.chunksOf dim valVec
    mapM (putStrLn.show) splitVec
    return ()

main = do
    let dim        = 30
        vec        = initVec dim
        relaxedVec = relaxVec dim vec
    prettyPrint dim relaxedVec
    return ()
```

B Excerpts from testing

Testing the steps for an array of dimension 6, precision 2, with two processes:

```
Processor with rank 0 has a working array of:
1.0000 1.0000 1.0000 1.0000 1.0000 1.0000
1.0000 0.0000 0.0000 0.0000 0.0000 1.0000
1.0000 0.0000 0.0000 0.0000 0.0000 1.0000
1.0000 0.0000 0.0000 0.0000 0.0000 1.0000
Processor with rank 1 has a working array of:
1.0000 0.0000 0.0000 0.0000 0.0000 1.0000
1.0000 0.0000 0.0000 0.0000 0.0000 1.0000
```

```
1.0000 0.0000 0.0000 0.0000 0.0000 1.0000
1.0000 1.0000 1.0000 1.0000 1.0000 1.0000
Processor with rank 0 has a working array of:
1.0000 1.0000 1.0000 1.0000 1.0000 1.0000
1.0000 0.5000 0.2500 0.2500 0.5000 1.0000
1.0000 0.2500 0.0000 0.0000 0.2500 1.0000
1.0000 0.2500 0.0000 0.0000 0.2500 1.0000
Processor with rank 1 has a working array of:
1.0000 0.2500 0.0000 0.0000 0.2500 1.0000
1.0000 0.2500 0.0000 0.0000 0.2500 1.0000
1.0000 0.5000 0.2500 0.2500 0.5000 1.0000
1.0000 1.0000 1.0000 1.0000 1.0000 1.0000
Processor with rank 0 has a working array of:
1.0000 1.0000 1.0000 1.0000 1.0000 1.0000
1.0000 0.6250 0.4375 0.4375 0.6250 1.0000
1.0000 0.4375 0.1250 0.1250 0.4375 1.0000
1.0000 0.4375 0.1250 0.1250 0.4375 1.0000
Processor with rank 1 has a working array of:
1.0000 0.4375 0.1250 0.1250 0.4375 1.0000
1.0000 0.4375 0.1250 0.1250 0.4375 1.0000
1.0000 0.6250 0.4375 0.4375 0.6250 1.0000
1.0000 1.0000 1.0000 1.0000 1.0000 1.0000
Processor with rank 0 has a working array of:
1.0000 1.0000 1.0000 1.0000 1.0000 1.0000
1.0000 0.7188 0.5469 0.5469 0.7188 1.0000
1.0000 0.5469 0.2812 0.2812 0.5469 1.0000
1.0000 0.5469 0.2812 0.2812 0.5469 1.0000
Processor with rank 1 has a working array of:
1.0000 0.5469 0.2812 0.2812 0.5469 1.0000
1.0000 0.5469 0.2812 0.2812 0.5469 1.0000
1.0000 0.7188 0.5469 0.5469 0.7188 1.0000
1.0000 1.0000 1.0000 1.0000 1.0000 1.0000
Processor with rank 0 has a working array of:
1.0000 1.0000 1.0000 1.0000 1.0000 1.0000
1.0000 0.7734 0.6367 0.6367 0.7734 1.0000
1.0000 0.6367 0.4141 0.4141 0.6367 1.0000
1.0000 0.6367 0.4141 0.4141 0.6367 1.0000
Processor with rank 1 has a working array of:
1.0000 0.6367 0.4141 0.4141 0.6367 1.0000
1.0000 0.6367 0.4141 0.4141 0.6367 1.0000
1.0000 0.7734 0.6367 0.6367 0.7734 1.0000
1.0000 1.0000 1.0000 1.0000 1.0000 1.0000
Processor with rank 0 has a working array of:
1.0000 1.0000 1.0000 1.0000 1.0000 1.0000
1.0000 0.8184 0.7061 0.7061 0.8184 1.0000
```

```
1.0000 0.7061 0.5254 0.5254 0.7061 1.0000
1.0000 0.7061 0.5254 0.5254 0.7061 1.0000
Processor with rank 1 has a working array of:
1.0000 0.7061 0.5254 0.5254 0.7061 1.0000
1.0000 0.7061 0.5254 0.5254 0.7061 1.0000
1.0000 0.8184 0.7061 0.7061 0.8184 1.0000
1.0000 1.0000 1.0000 1.0000 1.0000 1.0000
Processor with rank 0 has a working array of:
1.0000 1.0000 1.0000 1.0000 1.0000 1.0000
1.0000 0.8530 0.7625 0.7625 0.8530 1.0000
1.0000 0.7625 0.6157 0.6157 0.7625 1.0000
1.0000 0.7625 0.6157 0.6157 0.7625 1.0000
Processor with rank 1 has a working array of:
1.0000 0.7625 0.6157 0.6157 0.7625 1.0000
1.0000 0.7625 0.6157 0.6157 0.7625 1.0000
1.0000 0.8530 0.7625 0.7625 0.8530 1.0000
1.0000 1.0000 1.0000 1.0000 1.0000 1.0000
Processor with rank 0 has a working array of:
1.0000 1.0000 1.0000 1.0000 1.0000 1.0000
1.0000 0.8812 0.8078 0.8078 0.8812 1.0000
1.0000 0.8078 0.6891 0.6891 0.8078 1.0000
1.0000 0.8078 0.6891 0.6891 0.8078 1.0000
Processor with rank 1 has a working array of:
1.0000 0.8078 0.6891 0.6891 0.8078 1.0000
1.0000 0.8078 0.6891 0.6891 0.8078 1.0000
1.0000 0.8812 0.8078 0.8078 0.8812 1.0000
1.0000 1.0000 1.0000 1.0000 1.0000 1.0000
Processor with rank 0 has a working array of:
1.0000 1.0000 1.0000 1.0000 1.0000 1.0000
1.0000 0.9039 0.8445 0.8445 0.9039 1.0000
1.0000 0.8445 0.7484 0.7484 0.8445 1.0000
1.0000 0.8445 0.7484 0.7484 0.8445 1.0000
Processor with rank 1 has a working array of:
1.0000 0.8445 0.7484 0.7484 0.8445 1.0000
1.0000 0.8445 0.7484 0.7484 0.8445 1.0000
1.0000 0.9039 0.8445 0.8445 0.9039 1.0000
1.0000 1.0000 1.0000 1.0000 1.0000 1.0000
Processor with rank 0 has a working array of:
1.0000 1.0000 1.0000 1.0000 1.0000 1.0000
1.0000 0.9223 0.8742 0.8742 0.9223 1.0000
1.0000 0.8742 0.7965 0.7965 0.8742 1.0000
1.0000 0.8742 0.7965 0.7965 0.8742 1.0000
Processor with rank 1 has a working array of:
1.0000 0.8742 0.7965 0.7965 0.8742 1.0000
1.0000 0.8742 0.7965 0.7965 0.8742 1.0000
```

```
1.0000 0.9223 0.8742 0.8742 0.9223 1.0000
1.0000 1.0000 1.0000 1.0000 1.0000 1.0000
Processor with rank 0 has a working array of:
1.0000 1.0000 1.0000 1.0000 1.0000 1.0000
1.0000 0.9371 0.8982 0.8982 0.9371 1.0000
1.0000 0.8982 0.8354 0.8354 0.8982 1.0000
1.0000 0.8982 0.8354 0.8354 0.8982 1.0000
Processor with rank 1 has a working array of:
1.0000 0.8982 0.8354 0.8354 0.8982 1.0000
1.0000 0.8982 0.8354 0.8354 0.8982 1.0000
1.0000 0.9371 0.8982 0.8982 0.9371 1.0000
1.0000 1.0000 1.0000 1.0000 1.0000 1.0000
Processor with rank 0 has a working array of:
1.0000 1.0000 1.0000 1.0000 1.0000 1.0000
1.0000 0.9491 0.9177 0.9177 0.9491 1.0000
1.0000 0.9177 0.8668 0.8668 0.9177 1.0000
1.0000 0.9177 0.8668 0.8668 0.9177 1.0000
Processor with rank 1 has a working array of:
1.0000 0.9177 0.8668 0.8668 0.9177 1.0000
1.0000 0.9177 0.8668 0.8668 0.9177 1.0000
1.0000 0.9491 0.9177 0.9177 0.9491 1.0000
1.0000 1.0000 1.0000 1.0000 1.0000 1.0000
Processor with rank 0 has a working array of:
1.0000 1.0000 1.0000 1.0000 1.0000 1.0000
1.0000 0.9588 0.9334 0.9334 0.9588 1.0000
1.0000 0.9334 0.8922 0.8922 0.9334 1.0000
1.0000 0.9334 0.8922 0.8922 0.9334 1.0000
Processor with rank 1 has a working array of:
1.0000 0.9334 0.8922 0.8922 0.9334 1.0000
1.0000 0.9334 0.8922 0.8922 0.9334 1.0000
1.0000 0.9588 0.9334 0.9334 0.9588 1.0000
1.0000 1.0000 1.0000 1.0000 1.0000 1.0000
Processor with rank 0 has a working array of:
1.0000 1.0000 1.0000 1.0000 1.0000 1.0000
1.0000 0.9667 0.9461 0.9461 0.9667 1.0000
1.0000 0.9461 0.9128 0.9128 0.9461 1.0000
1.0000 0.9461 0.9128 0.9128 0.9461 1.0000
Processor with rank 1 has a working array of:
1.0000 0.9461 0.9128 0.9128 0.9461 1.0000
1.0000 0.9461 0.9128 0.9128 0.9461 1.0000
1.0000 0.9667 0.9461 0.9461 0.9667 1.0000
1.0000 1.0000 1.0000 1.0000 1.0000 1.0000
Processor with rank 0 has a working array of:
1.0000 1.0000 1.0000 1.0000 1.0000 1.0000
1.0000 0.9731 0.9564 0.9564 0.9731 1.0000
```

```
1.0000 0.9564 0.9295 0.9295 0.9564 1.0000
1.0000 0.9564 0.9295 0.9295 0.9564 1.0000
Processor with rank 1 has a working array of:
1.0000 0.9564 0.9295 0.9295 0.9564 1.0000
1.0000 0.9564 0.9295 0.9295 0.9564 1.0000
1.0000 0.9731 0.9564 0.9564 0.9731 1.0000
1.0000 1.0000 1.0000 1.0000 1.0000 1.0000
Processor with rank 0 has a working array of:
1.0000 1.0000 1.0000 1.0000 1.0000 1.0000
1.0000 0.9782 0.9647 0.9647 0.9782 1.0000
1.0000 0.9647 0.9429 0.9429 0.9647 1.0000
1.0000 0.9647 0.9429 0.9429 0.9647 1.0000
Processor with rank 1 has a working array of:
1.0000 0.9647 0.9429 0.9429 0.9647 1.0000
1.0000 0.9647 0.9429 0.9429 0.9647 1.0000
1.0000 0.9782 0.9647 0.9647 0.9782 1.0000
1.0000 1.0000 1.0000 1.0000 1.0000 1.0000
Processor with rank 0 has a working array of:
1.0000 1.0000 1.0000 1.0000 1.0000 1.0000
1.0000 0.9824 0.9715 0.9715 0.9824 1.0000
1.0000 0.9715 0.9538 0.9538 0.9715 1.0000
1.0000 0.9715 0.9538 0.9538 0.9715 1.0000
Processor with rank 1 has a working array of:
1.0000 0.9715 0.9538 0.9538 0.9715 1.0000
1.0000 0.9715 0.9538 0.9538 0.9715 1.0000
1.0000 0.9824 0.9715 0.9715 0.9824 1.0000
1.0000 1.0000 1.0000 1.0000 1.0000 1.0000
Processor with rank 0 has a working array of:
1.0000 1.0000 1.0000 1.0000 1.0000 1.0000
1.0000 0.9824 0.9715 0.9715 0.9824 1.0000
1.0000 0.9715 0.9538 0.9538 0.9715 1.0000
1.0000 0.9715 0.9538 0.9538 0.9715 1.0000
Processor with rank 1 has a working array of:
1.0000 0.9715 0.9538 0.9538 0.9715 1.0000
1.0000 0.9715 0.9538 0.9538 0.9715 1.0000
1.0000 0.9824 0.9715 0.9715 0.9824 1.0000
1.0000 1.0000 1.0000 1.0000 1.0000 1.0000
Processor with rank 0 has a working array of:
1.0000 1.0000 1.0000 1.0000 1.0000 1.0000
1.0000 0.9824 0.9715 0.9715 0.9824 1.0000
1.0000 0.9715 0.9538 0.9538 0.9715 1.0000
1.0000 0.9715 0.9538 0.9538 0.9715 1.0000
1.0000 0.9824 0.9715 0.9715 0.9824 1.0000
1.0000 1.0000 1.0000 1.0000 1.0000 1.0000
```