PARALLEL COMPUTING USING R AND A RASPBERRY PI CLUSTER PART II: SHIFTING GEARS FOR MORE SPEED!

KEITH E EMMERT AND PETER WHITE
TARLETON STATE UNIVERSITY
DEPARTMENT OF MATHEMATICS
BOX T-0470, STEPHENVILLE, TX 76402
EMMERTTARLETON.EDU
WHITETARLETON.EDU

1 Introduction

In a previous paper, we discussed the midpoint method in higher dimensions and noted some of its shortcomings. The two largest shortcomings are the need for large chunks of RAM to be allocated in order to utilize R's ability to vectorize functions in order to process the midpoint rule faster. One way to avoid large blocks of RAM allocation is to use nested for loops. This latter idea slows the processing down immensely. See Emmert and White [2024].

In this paper we will investigate the idea of distributed computing as well as parallel computing. In distributed computing, multiple cores and multiple processes across multiple computers can be utilized in order to solve a problem. In parallel computing, only one computer is utilized, but multiple cores or multiple threads can be utilized. We will use a single desktop for some benchmarks and then apply parallel processing with a single Raspberry Pi compute module as well as distributed computing to a Raspberry Pi cluster.

Once again the idea found in many calculus courses as well as higher level courses is numerical integration. We focus on the classic midpoint method. See, for example, Hass et al. [2022], Press et al. [1992], Zill [2024], DuChateau and Zachman [2002]. The ease of cluster creation using Raspberry Pi compute modules is well documented and can easily inspire students to learn about clusters and numerical analysis. See Raspberry Pi Tutorial [2024] or Rasmurtech [2024]. A quick search will illustrate the popularity of building Pi clusters and many cases exist to safely store them and many, many online tutorials or videos.

For this paper, we use using R and the futures library. The futures library allows sequential, parallel, or distributed processing architectures. For package documentation and vignettes, see Bengtsson [2024]. A tutorial can be found at Heiss [2008].

Throughout this paper, we will apply the midpoint method to the following function on $\Omega = [-1, 1]^{10}$ to estimate the volume of the paraboloid

$$f(x_1, x_2, \dots, x_{10}) = 10 - x_1^2 - x_2^2 - \cdots - x_{10}^2$$

The exact value of the integral is given by

$$\int_{-1}^{1} \cdots \int_{-1}^{1} \int_{-1}^{1} (10 - x_1^2 - x_2^2 - \cdots - x_{10}^2)\, dx_1\, dx_2 \cdots dx_{10} = \frac{20,480}{3} = 6,826.6666\cdots. \tag{1}$$

Given $N \in \mathbb{N}$, the ten-dimensional midpoint method is computed in the usual manner by defining the following values

$$\Delta x_k = \frac{1-(-1)}{N} = \frac{2}{N}$$
$$x_{k,j} = -1 + j\Delta x_k$$
$$m_{k,i_k} = \frac{x_{k,i_k} - x_{k,i_k-1}}{2}$$

where $k = 1, 2, \ldots, 10$, $j = 0, 1, 2, \ldots, N$, and $i_k = 1, 2, \ldots, N$. Of course, Δx_k represents one side of the *base* of a hyper-cube in the domain Ω, $x_{k,j}$ form the regular partition along one of the ten different segments $[-1, 1]$, and m_{k,i_k} are the midpoints along one of the ten different segments $[-1, 1]$.

The midpoint method calculated in a similar way to what would be found in a calculus book by adding the areas (*base* times *height*) of N rectangles. See Hass et al. [2022]. For higher dimensions, we have the hyper-cube volume $\Delta x_1 \Delta x_2 \cdots \Delta x_{10}$ (our *base*) and the additional *height* given by $f(m_{1,i_1}, m_{2,i_2}, \ldots, m_{10,i_{10}})$. The resulting N volumes are summed in order to estimate the value of the integral. Thus, we have

$$M_{N^{10}} = \sum_{i_{10}=1}^{N} \cdots \sum_{i_2=1}^{N} \sum_{i_1=1}^{N} f(m_{1,i_1}, m_{2,i_2}, \ldots, m_{10,i_{10}}) \Delta x_1 \Delta x_2 \cdots \Delta x_{10}. \tag{2}$$

2 An Introduction to R's futures Library

R's *futures* library allows different *plans* to be created. While possible to create a *plan* that is *sequential*, the main purpose would be utilizing *plans* that are *multiprocessing* or *cluster*. The *multiprocessing plan* allows multiple cores on the same machine to be created while the *cluster plan* allows multiple cores distributed across multiple machines. See Heiss [2008] for some examples or Bengtsson [2024] for the documentation and many vignettes on the *futures* package.

Listing 23 is an example of creating a *cluster plan* with three Raspberry Pi compute modules (or any computers) utilizing two cores. The key set up is the *plan* command shown below.

```
plan(cluster,
      workers = c(
        'localhost', '10.0.0.3', '10.0.0.4',
        'localhost', '10.0.0.3', '10.0.0.4'
        )
      )
```

We see that the *plan* is *cluster* and there are six *workers* located at the addresses *localhost*, which refers to the machine running the originating script, and two slave machines whose IP addresses have been specified (10.0.0.3 and 10.0.0.4).

The next step is to construct *futures*. A *future* is function which will begin executing, but will not execute in a sequential manner. So, several *futures* can be created which might take a long time before completion. Code will continue to execute until the *value* of the future is requested. If the *future* is already complete, then the value is returned. If not, the program halts and waits for the *future* to complete. An example of setting up a *future* is shown below. This is an excerpt from Listing 8.

```
myMidMultiCore1 <- future({
  midPoint10D(
    a, b, my10N, numberOfPartitions,
    whichPartition = 1, whichPartition2 = 0
    )
  })
```

```
myMidMultiCore2 <- future({
  midPoint10D(
    a, b, my10N, numberOfPartitions,
    whichPartition = 2, whichPartition2 = 0
    )
})

myMidMultiCore3 <- future({
  midPoint10D(
    a, b, my10N, numberOfPartitions,
    whichPartition = 3, whichPartition2 = 0
    )
  })

myMidMultiCore4 <- future({
  midPoint10D(
    a, b, my10N, numberOfPartitions,
    whichPartition = 4, whichPartition2 = 0
    )
  })

myMidMultiCore5 <- future({
  midPoint10D(
    a, b, my10N, numberOfPartitions,
    whichPartition = 5, whichPartition2 = 0
    )
  })

myMidMultiCore6 <- future({
  midPoint10D(
    a, b, my10N, numberOfPartitions,
    whichPartition = 6, whichPartition2 = 0
    )
  })

# Now we request the values of each future
myMidMultiCore <- value(myMidMultiCore1) + value(myMidMultiCore2) +
    value(myMidMultiCore3) + value(myMidMultiCore4) +
    value(myMidMultiCore5) + value(myMidMultiCore6)
```

So, six *futures* are created and stored in variables *myMidMultiCore1*, *myMidMultiCore2*, etc. These variables really point to separate instances of the function *midPoint10D* (more on this later) which is used to compute portions of the midpoint rule. Note that we have 3 Raspberry Pi compute modules (or three desktops if you wish) and 2 cores available on each computer. These 6 *futures* are split evenly among the computers and cores.

The very last statement requests the *value* of each of the 6 *futures*. Again, if one or more of these *values* is not completed, the main program can not progress until all are returned.

To create a *sequential plan*, using one machine and one core, use the command *plan(sequential)* as seen in Listing 11. For a *multisession plan*, use the command *plan(multisession, workers = 2)*, see Listing 12. This *multisession plan* utilizes one computer and two cores. Create *futures* for *sequential* or *multisession plans* in the same manner as a *cluster plan*.

A value of $N = 6$ is utilized throughout the paper to avoid RAM allocation problems. R's *microbenchmark* library is used to time all of the methods. The midpoint method is repeated 10 times and the average time

in seconds to completion is computed. Additionally, a five number summary is generated.

3 Benchmarks Using a Desktop Computer

In this paper, a desktop was utilized in order to better compare how a more standard computer would compare to a Raspberry Pi compute module. The desktop has the following characteristics at the time the data was generated. The desktop uses a AMD Ryzen 5 3600 with 6 Cores, 3593 MHz for each core, and a total of 12 Threads. There is 32 GiB RAM installed with a separate graphics card. The computer runs Windows 11 Pro Version 10.0.22631 Build 22631.

Even though there are 6 cores available and 12 threads, we restrict our work to 3 cores with 1 thread each. This is so that we can more readily compare to the Raspberry Pi compute modules used which have only 4 cores and are single threaded.

3.1 R Vectorized Code on the Desktop Computer

First, we will consider R's ability to apply a function across an entire vector. All of the midpoints are constructed in RAM using an outer join and stored in a data frame. Then the function can be applied to all midpoints at the same time and the returned results are summed. Unfortunately R has a limitation due to RAM allocation. Attempting the midpoint method with $N = 8$, R throws *Error: cannot allocate vector of size 4.0 Gb*. So, for vectorized code, $N < 7$. Again, we assume $N = 6$ to avoid RAM allocation issues, especially when spinning up multiple processes which share RAM.

Here, we perform everything at once with vectorized processing. Listing 1 is utilized for one, two, or three cores to compute function values given midpoints. The listings utilized to perform the midpoint method vary depending upon the number of cores.

The estimates and times to completion are listed in Table 1.

Table 1: Estimates using R's vectorized functions for $M_{6^{10}}$ using 1, 2, and 3 cores in a single desktop computer. The true value is $\frac{20{,}480}{3} = 6,826.666\cdots$. Ten evaluations are used to create a five number summary as well as the mean time to completion.

$M_{6^{10}}$	Comp	Core	N	Min	Q_1	Mean	Median	Q_3	Max
6921.481	1	1	6	6.450	6.559	6.656	6.598	6.628	7.240
6921.481	1	2	6	4.921	4.924	4.968	4.930	4.978	5.150
6921.481	1	3	6	4.732	4.743	4.796	4.783	4.804	4.979

3.1.1 One Core Only

The code in Listing 11 is executed. The *future* and *microbenchmark* libraries are loaded and all external R code are sourced. This sets up a sequential plan using $N = 6$ and 10 repetitions. The timing is initiated and this code calls *midPointSequential* function.

The *midPointSequential* function is found in Listing 2. Here the *sequential future* with one partition is initiated. The function *midPoint10D* is called.

The function *midPoint10D* resides in Listing 10 and calls two functions. First, the function *buildPartition* also contained in Listing 10 is executed. This creates the required regular partitions $x_{k,j} = -1 + j\Delta x_k$ for each segment $[-1, 1]$ and then computes the midpoints, $m_{k,i_k} = \dfrac{x_{k,i_k} - x_{k,i_k-1}}{2}$, needed for the method. These midpoints are returned to *midPoint10D* as a data-frame. Second, the function *my10DFun* found in Listing 1 is executed to find the heights of each of the midpoints. These results are summed. This sum is multiplied times the volume of the base, Δx^{10}, completing one midpoint rule.

Ultimately, the midpoint rule iteration is returned to the code in Listing 11. The library *microbenchmark* computes the averages of ten iterations of the midpoint rule.

3.1.2 Two Cores

The basic flow remains mostly unchanged. The code in Listing 12 is executed. The *future* and *microbenchmark* libraries are loaded and all external R code are sourced. This sets up a *multisession plan* using $N = 6$ and 10 repetitions. The timing is initiated and this code calls *midPointMulti2* function.

The *midPointMulti2* function is found in Listing 3. Here two *multisession futures* each using a different partition are initiated. Each future calls function *midPoint10D* with a different partition request. Eventually, when both futures are completed, the results are summed to form the final midpoint rule.

The other change worth noting is that the function *midPoint10D*, residing in Listing 10, calls the *buildPartition* function. The *buildPartition* function splits the midpoints along one of the segments $[-1, 1]$ into two equal sizes. Thus, depending upon the partition request, 1 or 2, the lower or upper partition is returned to *midPoint10D*. Thus, when *midPoint10D* calls the function *my10DFun* (in Listing 1), half of the midpoints are utilized.

Each of these volumes is returned to *midPointMulti2* where they are summed to complete one iteration of the midpoint method. Again, *microbenchmark* library is utilized to compute the average times based upon ten iterations of the midpoint method across two cores.

3.1.3 Three Cores

Three cores behaves in a similar manner to two cores. Listing 13 is executed, creating a *multisession plan*, which then calls the function *midPointMulit3* in Listing 4. This sets up three *futures*. The function *buildPartition* (Listing 10) splits the midpoints along one segment $[-1, 1]$ into three equal sizes. Each of the three futures uses their respective partitions to compute part of the midpoint rule. Finally, *midPointMulti3* sums these volumes. This process is executed ten times by the *microbenchmark* library.

3.2 Using Nested For Loops on the Desktop Computer

Since the Raspberry Pi compute modules can not handle the creation of the entire data frame at once, we will swap to nested for loops. Of course this will drastically slow the evaluation. However, the gain is that less RAM is required. We hope that multiple cores might be able to overcome this problem. Once again, we choose $N = 6$ so that 6^{10} midpoints are evaluated. The timing library *microbenchmark* is utilized with ten repetitions so that the average time in seconds can be computed.

The estimates and times to completion are listed in Table 2.

Table 2: Estimates using nested for loops for $M_{6^{10}}$ using 1, 2, and 3 cores in a single desktop computer. The true value is $\frac{20{,}480}{3} = 6,826.666\cdots$. Ten evaluations are used to create a five number summary as well as the mean time to completion.

$M_{6^{10}}$	Comp	Core	N	Min	Q_1	Mean	Median	Q_3	Max
6921.481	1	1	6	404.321	404.868	405.544	405.578	406.147	407.240
6921.481	1	2	6	73.596	74.013	74.251	74.150	74.288	75.622
6921.481	1	3	6	51.527	51.549	51.693	51.651	51.697	52.307

3.2.1 One Core Only

There are a few changes in the code. Listing 16 is similar to that in Listing 11 and is executed first creating a *sequential plan*. Required libraries are loaded and all external R code are sourced. The only difference is that *midPoint10DVer2.R*, found in Listing 14, is sourced rather than *midPoint10D.R*, found in Listing 10. Both provide *midPoint10D* but only *midPoint10D.R* (Listing 10) provides *buildPartition* as well.

The function *midPointSequential* (Listing 2) is called and a *future* with one partition is created.

The function *midPoint10D*, see Listing 14, is called. This contains nested for loops which evaluate the midpoints. Two vectors are created, *xmSlice* and *xmFull*. These contain the midpoints. Note that when only one partition is utilized, *xmSlice* is the same as *xmFull*. Later, *xmSlice* will contain part of the midpoints along $[-1, 1]$. Thus only two vectors of midpoints are utilized.

The function *my10DFun* found in Listing 1 evaluates one single midpoint. Eventually, all of the heights will be summed and the volume of the base, Δx^{10} is multiplied, generating the midpoint rule's estimate for the integral.

3.2.2 Two Cores

A *multisession plan* is created by running the code in Listing 17. Then *midPointMulti2*, Listing 3 is used to create two *futures*. The rest proceeds as before. We should note that in *midPoint10D*, Listing 14, *xmSlice* contains either the upper or lower half of the midpoints along $[-1, 1]$. One future uses the lower slice, the other future uses the upper slice. The slicing impacts only the outer loop. The other nine loops use the full set of midpoints found in *xmFull*. Thus the workload is split between two sets of nested for loops running in different cores. So, *xmFull* is duplicated twice, and *xmSlice* contains half of the midpoints but is duplicated. So, essentially, three vectors of midpoints (two *xmFull* and two *xmSlice*) are stored in RAM. In this example, we have $N = 6$, so the RAM usage required is based upon 18 points (three vectors of 6) rather than $6^{10} = 6,0466,176$ midpoints that a full outer join (Cartesian product) of 10 vectors of 6 elements requires.

3.2.3 Three Cores

This is similar to the two core set up. Listing 18 is sourced creating a *multisession plan* and three *futures* are created in the function *midPointMulti3* which is located in Listing 4. Each of the three futures calls *midPoint10D* (Listing 14) and *xmSlice* contains different partitions, lower, middle, and upper, of midpoints along $[-1, 1]$. Thus the workload of the nested for loops is shared across three cores. Again, only the outermost for loop uses *xmSlice*. The other loops use the complete set of midpoints in *xmFull*. This time *xmFull* is duplicated three times, and *xmSlice* contains one third of the midpoints but is also duplicated three times. So, essentially, four vectors of midpoints (three *xmFull* and three *xmSlice*) are stored in RAM. Here we need enough RAM to store 4 vectors of length $N = 6$ or 24 points.

4 Raspberry Pi Compute Module

For this paper, three Raspberry Pi Ver 3B+ (Model: Cortex-A53) compute modules were utilized. This version of Raspberry Pi contains four cores clocked at 1400 MHz each, are single threaded, utilizes shared memory of 948 MiB for CPU and 76 MiB for GPU, and are running Linux version 6.1.21-v7+, 32 Bit. Due to RAM constraints, only nested for loops will be considered while utilizing Raspberry Pi compute modules.

4.1 Single Raspberry Pi Compute Modules

First we utilize one Raspberry Pi. The results are shown in Table 3. The code is the same as the desktop situation where ten nested for loops are utilized. One, two, or three *futures* are used. One of the four cores is left untouched for the operating system.

4.2 Two Raspberry Pi Compute Modules

Now we utilize two Raspberry Pi compute modules. The results are shown in Table 4. The main change to the R code utilized is that the *plan* is no longer *multisession*, but rather *cluster*. The total number of processes depends upon the number of cores times the number of Raspberry Pi compute modules. So, for this section, we consider 2, 4, and 6 *futures*. One core on each Raspberry Pi compute module is left untouched for the operating system.

Table 3: Estimates using nested for loops for $M_{6^{10}}$ using 1, 2, and 3 cores in a single Raspberry Pi compute modile. The true value is $\frac{20,480}{3} = 6,826.666\cdots$. Ten evaluations are used to create a five number summary as well as the mean time to completion.

$M_{6^{10}}$	Comp	Core	N	Min	Q_1	Mean	Median	Q_3	Max
6921.481	1	1	6	2879.836	2889.209	2894.955	2896.058	2899.240	2910.333
6921.481	1	2	6	825.095	827.320	830.904	830.722	834.099	838.113
6921.481	1	3	6	607.252	608.996	614.480	612.942	616.289	636.018

Table 4: Estimates using nested for loops for $M_{6^{10}}$ using 1, 2, and 3 cores in a two Raspberry Pi compute modiles. The true value is $\frac{20,480}{3} = 6,826.666\cdots$. Ten evaluations are used to create a five number summary as well as the mean time to completion. Note that there are $Comp * Core$ processes run (i.e. 2, 4, 6).

$M_{6^{10}}$	Comp	Core	N	Min	Q_1	Mean	Median	Q_3	Max
6921.481	2	1	6	817.324	820.561	822.740	823.032	825.332	828.223
6921.481	2	2	6	590.525	595.004	600.165	601.776	605.411	608.458
6921.481	2	3	6	321.336	322.902	323.293	323.423	323.742	324.707

4.2.1 Two Cores

We begin by running the code found in Listing 19. This sets up the *plan* for *cluster* registering the addresses of the *workers* to be *localhost* and an IP address (10.0.0.3 for my old router) to a second Raspberry Pi compute module. The function *midPointMulti2*, Listing 3, is called and two *futures* are created, one for each Raspberry Pi. The function *midPoint10D* found in Listing 14 is called. One of two partitions (say upper or lower) is created depending upon the variable *whichPartition* and the ten nested for loops are run. Half of the midpoint rule is returned (based upon the upper or lower partition). The function *midPointMulti2* sums both parts of the midpoint rule and returns the final estimate. RAM usage is one vector of length $N = 6$ and one vector of length $\frac{N}{2} = 3$, that is, allocates enough RAM for 9 points.

4.2.2 Four Cores

We begin by running the code found in Listing 20. This sets up the *plan* for *cluster* registering the addresses of the *workers* to be *localhost* and an IP address (10.0.0.3 for my old router) to a second Raspberry Pi compute module. The main difference when registering processes is that *localhost* and the IP address (10.0.0.3 for example) each appear twice. This generates a total of four potential processes, two on each Raspberry Pi. The function *midPointMulti4*, Listing 5, is called and four *futures* are created, two for each Raspberry Pi. The function *midPoint10D* found in Listing 14 is called. One of four partitions is created depending upon the variable *whichPartition* and the ten nested for loops are run. The function *midPointMulti4* sums the four parts of the midpoint rule and returns the final estimate. RAM usage is one vector of length $N = 6$ and one vector of length $\frac{N}{4} = 3$ per core per Raspberry Pi. So, each Raspberry Pi allocates enough RAM for 18 points.

4.2.3 Six Cores

We begin by running the code found in Listing 21. This sets up the *plan* for *cluster* registering the addresses of the *workers* to be *localhost* and an IP address (10.0.0.3 for my old router) to a second Raspberry Pi compute module. Again, the main difference when registering processes is that *localhost* and the IP address (10.0.0.3 for example) each appear three times. Six potential processes, three on each Raspberry Pi have now been created. The function *midPointMulti6*, Listing 6, is called and six *futures* are created spread evenly across the two Raspberry Pi compute modules. The function *midPoint10D* found in Listing 14 is called. One of six partitions is created depending upon the variable *whichPartition* and the ten nested for loops are

run. The function *midPointMulti6* returns the sum of the six parts of the midpoint rule. RAM usage is one vector of length $N = 6$ and one vector of length $\frac{N}{6} = 1$ per core per Raspberry Pi. So, each Raspberry Pi allocates enough RAM for 21 points.

4.3 Three Raspberry Pi Compute Modules

Finally we utilize three Raspberry Pi compute modules. The results are shown in Table 5. The total number of processes depends upon the number of cores times the number of Raspberry Pi compute modules. So, for this section, we consider 3, 6, and 9 processes with one core on each Raspberry Pi compute module is left untouched for the operating system.

Table 5: Estimates using nested for loops for $M_{6^{10}}$ using 1, 2, and 3 cores in a three Raspberry Pi compute modiles. The true value is $\frac{20{,}480}{3} = 6,826.666\cdots$. Ten evaluations are used to create a five number summary as well as the mean time to completion. Note that there are $Comp * Core$ processes run (i.e. 3, 6, 9).

$M_{6^{10}}$	Comp	Core	N	Min	Q_1	Mean	Median	Q_3	Max
6921.481	3	1	6	548.842	550.179	553.788	550.658	553.205	578.533
6921.481	3	2	6	319.607	320.396	321.373	321.634	321.895	323.794
6921.481	3	3	6	218.141	219.378	220.345	220.226	221.596	221.842

4.3.1 Three Cores

The code in Listing 22 is executed. This creates a *plan* for *cluster* and registers the addresses for three *workers* as *localhost* and appropriate IP addresses (10.0.0.3 and 10.0.0.4).

The function *midPointMulti3* is called which is located in Listing 7. (Note that Listing 4 also contains a function named *midPointMulti3*. Technically this function could be used, but the newer one was written adding extra functionality which is needed for larger numbers of processes.) The function *midPointMulti3* allows two variables, *whichPartition* and *whichPartition2*, to control which partition is accessed. The second one, *whichPartition2*, is set to zero as this functionality is not needed. Different values of N, especially $N = 9$, could necessitate the use of this second partition selection variable. Three *futures* are created and *midPoint10D* is called.

The function *midPoint10D* is located in Listing 15. This function contains the nested for loops and allows *xmSlice1* and *xmSlice2* to be created. These variables contain slices of midpoints along $[-1, 1]$. The function *my10D* in Listing 1 handles the function evaluation. Eventually, part of the midpoint rule is returned.

Back in the function *midPointMulti3*, all of the parts are summed and the final value of the midpoint rule is returned. For this process, RAM usage is one vector of length $N = 6$ and one vector of length $\frac{N}{3} = 2$ for each Raspberry Pi compute module. Thus enough RAM is allocated to store 8 points.

4.3.2 Six Cores

Listing 23 is used to create a *cluster plan* and registers the addresses for six *workers* using *localhost* and appropriate IP addresses (10.0.0.3 and 10.0.0.4), twice.

The function *midPointMulti6* in Listing 8 (not the one located in Listing 6 which is used for two Raspberry Pi compute modules) is called. This function again has two partition selection variables, *whichPartition* and *whichPartition2*, but only really needs one (*whichPartition2* is set to zero). Six *futures* are created and distributed among the three Raspberry Pi compute modules. Each future calls *midPoint10D* (Listing 15) to compute compute part of the midpoint rule using nested for loops. Partitioning occurs in *midPoint10D*. The function *my10DFun* in Listing 1 handles the evaluation of each midpoint.

Each of the *futures* in *midPointMulti6* return a portion of the midpoint rule and are summed and returned. This process requires two vectors of length $N = 6$ and two vectors of length $\frac{N}{6} = 1$. Thus, each Raspberry Pi compute module must allocate enough RAM to store 14 points.

4.3.3 Nine Cores

A *cluster plan* is created by executing Listing 24. Nine *workers* are registered using *localhost* and appropriate IP addresses (10.0.0.3 and 10.0.0.4), three times.

Function *midPointMulti9* in Listing 9 is executed which creates nine *futures* and allocates partitions. Since $N = 6$ is not divisible by 9, partitioning occurs using two partition selection variables, *whichPartition* and *whichPartition2*.

Function *midPoint10D* in Listing 15 is called. Since nine partitions are required, two slices are split into three pieces each, *xmSlice1* and *xmSlice2*. The variables *whichPartition* and *whicPartition2* contain the numbers 1, 2, or 3. Thus nine possible combinations exist, one for each partition and the first two of the ten nested loops iterate over partitions whose length is one-third of N. Function *my10DFun* in Listing 1 evaluates each midpoint. A part of the midpoint rule is returned to *midPointMulti9*.

Finally, *midPointMulti9* sums the nine pieces that the *futures* return, forming the completed midpoint rule. For this process, each Raspberry Pi must track three vectors of length $N = 6$ and three vectors of length $\frac{N}{3} = 2$. Thus, enough RAM for 24 points is required.

5 Conclusion

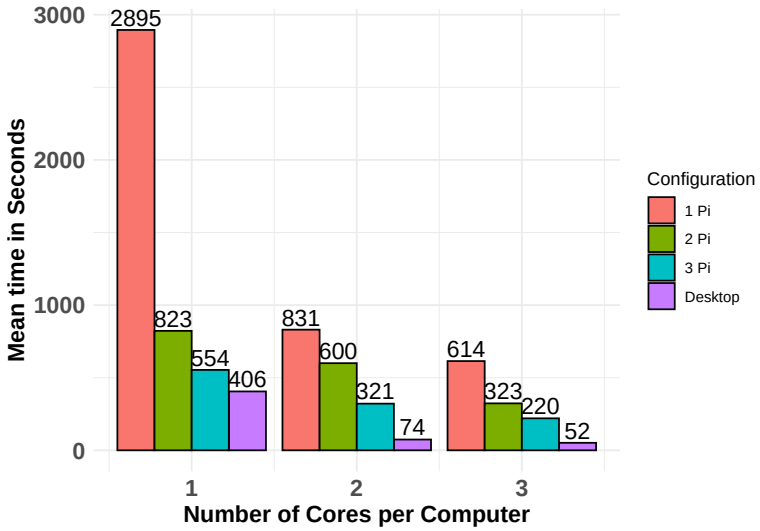

Figure 1: Comparison of nested for loops utilizing the desktop, one, two, and three Raspberry Pi compute modules. Mean completion time shown for one, two, or three cores.

Figure 1 summarizes the mean time to completion of 10 iterations using various plans and hardware architecture. It is interesting to note that moving from a single core on one machine to two cores on the same machine or one core on two machines decreases the mean run time by an order of magnitude. This is not too troubling since it is not reasonable to use the *futures* library if you are only utilizing one core on one machine. This creates unnecessary overhead. Also, we should note that either 2 Raspberry Pi compute modules using 3 cores each or 3 Raspberry Pi compute modules with 2 cores each perform better than the desktop with one core.

This might well be an interesting project for students who are interested in parallel processing to explore. It

requires low costs as Raspberry Pi compute modules are fairly cheap and any old router will work. Internet access is not required after the Raspberry Pi compute modules are set up.

Future work will explore faster methods for estimating integrals utilizing variations of Monte Carlo integration.

A Code Listings

A.1 Common Code

A.1.1 The Function

Listing 1: The function whose integral we seek to approximate.

```
# File name:  my10DFun.R
my10DFun = function(x1, x2, x3, x4, x5, x6, x7, x8, x9, x10)
{
  return(
    10 - x1*x1  - x2*x2  - x3*x3  - x4*x4  - x5*x5  -
    x6*x6  - x7*x7  - x8*x8  - x9*x9  - x10*x10
    )
}
```

A.1.2 Sequential Midpoint Rule with a Single Future

Listing 2: Perform a sequential midpoint rule with a single future

```
# File name:  midPointSequential.R
midPointSequential = function(a, b, my10N, numberOfPartitions)
{
  myMidSequential <- future({
    midPoint10D(a, b, my10N, numberOfPartitions, whichPartition = 1)
  })

  myMidResult <- value(myMidSequential)

  return(myMidResult)
}
```

A.1.3 Parallel Midpoint Rule with Two Futures

Listing 3: Perform a parallel midpoint rule with a two futures

```
# File name:  midPointMulti2.R
midPointMulti2 = function(a, b, my10N, numberOfPartitions)
{
  myMidMultiCore1 <- future({
    midPoint10D(a, b, my10N, numberOfPartitions, whichPartition = 1)
  })

  myMidMultiCore2 <- future({
    midPoint10D(a, b, my10N, numberOfPartitions, whichPartition = 2)
  })

  myMidMultiCore <- value(myMidMultiCore1) + value(myMidMultiCore2)
```

```
  return(myMidMultiCore)
}
```

A.1.4 Parallel Midpoint Rule with Three Futures

Listing 4: Perform a parallel midpoint rule with a three futures

```
# File name:  midPointMulti3.R
midPointMulti3 = function(a, b, my10N, numberOfPartitions)
{
  myMidMultiCore1 <- future({
    midPoint10D(a, b, my10N, numberOfPartitions = 3, whichPartition = 1)
  })

  myMidMultiCore2 <- future({
    midPoint10D(a, b, my10N, numberOfPartitions = 3, whichPartition = 2)
  })

  myMidMultiCore3 <- future({
    midPoint10D(a, b, my10N, numberOfPartitions = 3, whichPartition = 3)
  })

  myMidMultiCore <- value(myMidMultiCore1) +
      value(myMidMultiCore2) +
      value(myMidMultiCore3)

  return(myMidMultiCore)
}
```

A.1.5 Parallel Midpoint Rule with Four Futures

Listing 5: Perform a parallel midpoint rule with a four futures

```
# File name:  midPointMulti4.R
midPointMulti4 = function(a, b, my10N, numberOfPartitions)
{
  myMidMultiCore1 <- future({
    midPoint10D(a, b, my10N, numberOfPartitions, whichPartition = 1)
  })

  myMidMultiCore2 <- future({
    midPoint10D(a, b, my10N, numberOfPartitions, whichPartition = 2)
  })

  myMidMultiCore3 <- future({
    midPoint10D(a, b, my10N, numberOfPartitions, whichPartition = 3)
  })

  myMidMultiCore4 <- future({
    midPoint10D(a, b, my10N, numberOfPartitions, whichPartition = 4)
  })

  myMidMultiCore <- value(
    myMidMultiCore1) + value(myMidMultiCore2) +
    value(myMidMultiCore3) + value(myMidMultiCore4
```

```
    )

  return(myMidMultiCore)
}
```

A.1.6 Parallel Midpoint Rule with Six Futures and Two Raspberry Pi Compute Modules

Listing 6: Perform a parallel midpoint rule with a six futures and two Raspberry Pi compute modules.

```
# File name:  midPointMulti6.R
midPointMulti6 = function(a, b, my10N, numberOfPartitions)
{
  myMidMultiCore1 <- future({
    midPoint10D(
      a, b, my10N,
      numberOfPartitions, whichPartition = 1
      )
  })

  myMidMultiCore2 <- future({
    midPoint10D(a, b, my10N, numberOfPartitions, whichPartition = 2)
  })

  myMidMultiCore3 <- future({
    midPoint10D(a, b, my10N, numberOfPartitions, whichPartition = 3)
  })

  myMidMultiCore4 <- future({
    midPoint10D(a, b, my10N, numberOfPartitions, whichPartition = 4)
  })

  myMidMultiCore5 <- future({
    midPoint10D(a, b, my10N, numberOfPartitions, whichPartition = 5)
  })

  myMidMultiCore6 <- future({
    midPoint10D(a, b, my10N, numberOfPartitions, whichPartition = 6)
  })

  myMidMultiCore <- value(myMidMultiCore1) + value(myMidMultiCore2) +
      value(myMidMultiCore3) + value(myMidMultiCore4) +
      value(myMidMultiCore5) + value(myMidMultiCore6)

  return(myMidMultiCore)
}
```

A.1.7 Parallel Midpoint Rule with Three Futures and Three Raspberry Pi Compute Modules

Listing 7: Perform a parallel midpoint rule with a three futures and three Raspberry Pi compute modules. This is equivalent to Listing 4, but follows the style for three Raspberry Pi compute modules where two partition variables are utilized, whichPartition and whichPartition2.

```
# File name:  midPointMulti3For3Pi.R
midPointMulti3 = function(a, b, my10N, numberOfPartitions)
```

```
{
  myMidMultiCore1 <- future({
    midPoint10D(
      a, b, my10N, numberOfPartitions = 3,
      whichPartition = 1, whichPartition2 = 0
      )
  })

  myMidMultiCore2 <- future({
    midPoint10D(
      a, b, my10N, numberOfPartitions = 3,
      whichPartition = 2, whichPartition2 = 0
      )
  })

  myMidMultiCore3 <- future({
    midPoint10D(
      a, b, my10N, numberOfPartitions = 3,
      whichPartition = 3, whichPartition2 = 0
      )
  })

  myMidMultiCore <- value(myMidMultiCore1) +
      value(myMidMultiCore2) + value(myMidMultiCore3)

  return(myMidMultiCore)
}
```

A.1.8 Parallel Midpoint Rule with Six Futures and Three Raspberry Pi Compute Modules

Listing 8: Perform a parallel midpoint rule with a six futures and three Raspberry Pi compute modules.

```
# File name:  midPointMulti6For3Pi.R
midPointMulti6 = function(a, b, my10N, numberOfPartitions)
{
  myMidMultiCore1 <- future({
    midPoint10D(
      a, b, my10N, numberOfPartitions,
      whichPartition = 1, whichPartition2 = 0
      )
  })

  myMidMultiCore2 <- future({
    midPoint10D(
      a, b, my10N, numberOfPartitions,
      whichPartition = 2, whichPartition2 = 0
      )
  })

  myMidMultiCore3 <- future({
    midPoint10D(
      a, b, my10N, numberOfPartitions,
      whichPartition = 3, whichPartition2 = 0
      )
```

```
  })

  myMidMultiCore4 <- future({
    midPoint10D(
      a, b, my10N, numberOfPartitions,
      whichPartition = 4, whichPartition2 = 0
      )
  })

  myMidMultiCore5 <- future({
    midPoint10D(
      a, b, my10N, numberOfPartitions,
      whichPartition = 5, whichPartition2 = 0
      )
  })

  myMidMultiCore6 <- future({
    midPoint10D(
      a, b, my10N, numberOfPartitions,
      whichPartition = 6, whichPartition2 = 0
      )
  })

  myMidMultiCore <- value(myMidMultiCore1) + value(myMidMultiCore2) +
      value(myMidMultiCore3) + value(myMidMultiCore4) +
      value(myMidMultiCore5) + value(myMidMultiCore6)

  return(myMidMultiCore)
}
```

A.1.9 Parallel Midpoint Rule with Nine Futures and Three Raspberry Pi Compute Modules

Listing 9: Perform a parallel midpoint rule with a nine futures.

```
# File name:  midPointMulti9For3Pi.R
midPointMulti9 = function(a, b, my10N, numberOfPartitions)
{
  myMidMultiCore1 <- future({
    midPoint10D(
      a, b, my10N, numberOfPartitions = 3,
      whichPartition = 1, whichPartition2 = 1
      )
  })

  myMidMultiCore2 <- future({
    midPoint10D(
      a, b, my10N, numberOfPartitions = 3,
      whichPartition = 1, whichPartition2 = 2
      )
  })

  myMidMultiCore3 <- future({
    midPoint10D(
      a, b, my10N, numberOfPartitions = 3,
      whichPartition = 1, whichPartition2 = 3
```

```
    )
  })

  myMidMultiCore4 <- future({
    midPoint10D(
      a, b, my10N, numberOfPartitions = 3,
      whichPartition = 2, whichPartition2 = 1
      )
  })

  myMidMultiCore5 <- future({
    midPoint10D(
      a, b, my10N, numberOfPartitions = 3,
      whichPartition = 2, whichPartition2 = 2
      )
  })

  myMidMultiCore6 <- future({
    midPoint10D(
      a, b, my10N, numberOfPartitions = 3,
      whichPartition = 2, whichPartition2 = 3
      )
  })

  myMidMultiCore7 <- future({
    midPoint10D(
      a, b, my10N, numberOfPartitions = 3,
      whichPartition = 3, whichPartition2 = 1
      )
  })

  myMidMultiCore8 <- future({
    midPoint10D(
      a, b, my10N, numberOfPartitions = 3,
      whichPartition = 3, whichPartition2 = 2
      )
  })

  myMidMultiCore9 <- future({
    midPoint10D(
      a, b, my10N, numberOfPartitions = 3,
      whichPartition = 3, whichPartition2 = 3
      )
  })

  myMidMultiCore <- value(myMidMultiCore1) + value(myMidMultiCore2) +
    value(myMidMultiCore3) + value(myMidMultiCore4) +
    value(myMidMultiCore5) + value(myMidMultiCore6) +
    value(myMidMultiCore7) + value(myMidMultiCore8) +
    value(myMidMultiCore9)

  return(myMidMultiCore)
}
```

A.2 Vectorized Code Listings

A.2.1 Code to Build Partitions and Compute Midpoint using Vectorized Code

Listing 10: Build one or more partitions and compute the midpoint rule using vectors. This is a faster version, but more RAM intensive.

```
# File name: midPoint10D.R
#
# Create the needed partition of midpoints using an outer join / cross product.
# When numberOfPartitions > 0, then split xp1 into 1 or more pieces.
# This is used for parallel processes so that the data can be split to
# multiple CPUs / cores.
# whichPartition is an indication of which of the 1, 2, ... numberOfPartitions
# should be used.
buildPartition = function(a, b, my10N, numberOfPartitions, whichPartition)
{
  deltaX10D = (b-a) / my10N

  xp1 = seq(a, b,by=deltaX10D)
  xp2 = seq(a, b,by=deltaX10D)
  xp3 = seq(a, b,by=deltaX10D)
  xp4 = seq(a, b,by=deltaX10D)
  xp5 = seq(a, b,by=deltaX10D)
  xp6 = seq(a, b,by=deltaX10D)
  xp7 = seq(a, b,by=deltaX10D)
  xp8 = seq(a, b,by=deltaX10D)
  xp9 = seq(a, b,by=deltaX10D)
  xp10 = seq(a, b,by=deltaX10D)

  xm1 = (xp1[1:(length(xp1) - 1)] + xp1[2:length(xp1)])/2

  # Select a partition if positive
  if (numberOfPartitions > 0) {
    xm1 = split(
      xm1, cut(seq_along(xm1),
      numberOfPartitions, labels = FALSE)
      )[[whichPartition]]
  }

  xm2 = (xp2[1:(length(xp2) - 1)] + xp2[2:length(xp2)])/2
  xm3 = (xp3[1:(length(xp3) - 1)] + xp3[2:length(xp3)])/2
  xm4 = (xp4[1:(length(xp4) - 1)] + xp4[2:length(xp4)])/2
  xm5 = (xp5[1:(length(xp5) - 1)] + xp5[2:length(xp5)])/2
  xm6 = (xp6[1:(length(xp6) - 1)] + xp6[2:length(xp6)])/2
  xm7 = (xp7[1:(length(xp7) - 1)] + xp7[2:length(xp7)])/2
  xm8 = (xp8[1:(length(xp8) - 1)] + xp8[2:length(xp8)])/2
  xm9 = (xp9[1:(length(xp9) - 1)] + xp9[2:length(xp9)])/2
  xm10 = (xp10[1:(length(xp10) - 1)] + xp10[2:length(xp10)])/2

  # Cartesian product/full outer join
  gridPts10DF = expand.grid(
      xm1, xm2, xm3, xm4, xm5,
      xm6, xm7, xm8, xm9, xm10
      )
  colnames(gridPts10DF) = c('xm1', 'xm2', 'xm3', 'xm4', 'xm5',
```

```
                                'xm6', 'xm7', 'xm8', 'xm9', 'xm10')

  return(gridPts10DF)
}




# Perform the midpoint method. Need to know how many partitions and which
# partition should be utilized. When numberOfPartitions > 0, then the data
# in xm1 should be split into numberOfPartitions pieces. The variable
# whichPartition indicates which partition is to be used in the midpoint
# method.
midPoint10D = function(a, b, my10N, numberOfPartitions, whichPartition)
{
  deltaX10D = (b-a) / my10N

  gridPts10DF = buildPartition(
    a, b, my10N, numberOfPartitions, whichPartition
    )

  # Estimate for midpoint method using grid
  midPoint10D =
    sum(my10DFun(
                  gridPts10DF$xm1, gridPts10DF$xm2,
                  gridPts10DF$xm3, gridPts10DF$xm4,
                  gridPts10DF$xm5, gridPts10DF$xm6,
                  gridPts10DF$xm7, gridPts10DF$xm8,
                  gridPts10DF$xm9, gridPts10DF$xm10
                 )
        ) * deltaX10D^10

  return(midPoint10D)
}
```

A.2.2 Sequential Midpoint Method Using Vectorized Code

Listing 11: Code utilized to set up a sequential process for the ten-dimensional midpoint rule.

```
# File name: testMidPointRule.R
# Reality:  This is the first part of testMidPointRule.R which covers
# sequential on one core and one thread. The other parts use 2 and 3 cores,
# respectively.
#
# Load required files
source('my10DFun.R')
source('midPoint10D.R')
source('midPointSequential.R')

# Load required libraries
library(future) # Parallel Processing when desired
library(microbenchmark) # Timing

# Build a sequential process on one machine
plan(sequential)
```

```
a = -1
b = 1
my10N = 6

numRepetitions = 10 # Seek an average time to completion

mbmSequential <- microbenchmark(
  mySequentialMidResult <- midPointSequential(
      a, b, my10N, numberOfPartitions = 0
      ),
  times = numRepetitions,
  unit = 's'
  )

seqTiming = subset(
  summary(mbmSequential), select = -c(expr)
  )

seqTiming = cbind(
    Midpoint = round(mySequentialMidResult, 6),
    Computers = 1, Cores = 1, N = my10N, seqTiming
    )

midpointMethodTiming = seqTiming
```

A.2.3 Parallel Midpoint Method Using Vectorized Code with Two Cores

Listing 12: Code utilized to set up a multissesion process with two cores for the ten-dimensional midpoint rule.

```
# File name: testMidPointRule.R
# Reality:  This is the second part of testMidPointRule.R which covers
# sequential on two cores and one thread. The other parts use 1 and 3 cores,
# respectively.
#
# Load required files
source('my10DFun.R')
source('midPoint10D.R')
source('midPointMulti2.R')

# Load required libraries
library(future) # Parallel Processing when desired
library(microbenchmark) # Timing

# Build a sequential process on one machine
plan(multisession, workers = 2)

a = -1
b = 1
my10N = 6

# Build a multisession process using two cores
mbmMulti2 <- microbenchmark(
  myMultiMidResult2 <-midPointMulti2(a, b, my10N, numberOfPartitions = 2),
```

```
  times = numRepetitions,
  unit = 's'
)


multiTiming2 = subset(
  summary(mbmMulti2), select = -c(expr)
  )

multiTiming2 = cbind(
  Midpoint = round(myMultiMidResult2, 6),
  Computers = 1, Cores = 2, N = my10N, multiTiming2
  )
```

A.2.4 Parallel Midpoint Method Using Vectorized Code with Three Cores

Listing 13: Code utilized to set up a multissesion process with three cores for the ten-dimensional midpoint rule.

```
# File name: testMidPointRule.R
# Reality:  This is the first part of testMidPointRule.R which covers
# sequential on one core and one thread. The other parts use 2 and 3 cores,
# respectively.
#
# Load required files
source('my10DFun.R')
source('midPoint10D.R')
source('midPointMulti3.R')

# Load required libraries
library(future) # Parallel Processing when desired
library(microbenchmark) # Timing

# Build a multisession process on one machine
plan(multisession, workers = 3)

a = -1
b = 1
my10N = 6

numRepetitions = 10 # Seek an average time to completion

# Begin multissesion with three cores
mbmMulti3 <- microbenchmark(
  myMultiMidResult3 <- midPointMulti3(a, b, my10N, numberOfPartitions = 3),
  times = numRepetitions,
  unit = 's'
)

multiTiming3 = subset(
  summary(mbmMulti3), select = -c(expr)
  )

print('End Multissession core = 3')
```

```
multiTiming3 = cbind(
  Midpoint = round(myMultiMidResult3, 6),
  Computers = 1, Cores = 3, N = my10N, multiTiming3
  )
```

A.3 Code for Nested For Loops

A.3.1 Code to Build Partitions and Compute Midpoint using Nested For Loops

Listing 14: Utilize nested for loops to compute the midpoint rule. This, of course, is a slower version but less RAM intensive.

```
# File name: midPoint10DVer2.R
#
# When numberOfPartitions > 0, then split xp1 into 1 or more pieces.
# This is used for parallel processes so that the data can be split to
# multiple CPUs / cores.
# whichPartition is an indication of which of the 1, 2, ... numberOfPartitions
# should be used.
midPoint10D = function(a, b, my10N, numberOfPartitions, whichPartition)
{
  deltaX10D = (b-a) / my10N

  xp = seq(a, b,by=deltaX10D)

  xmSlice = (xp[1:(length(xp) - 1)] + xp[2:length(xp)])/2

  # Select a partition if positive
  if (numberOfPartitions > 0) {
    xmSlice = split(
      xmSlice,
      cut(seq_along(xmSlice), numberOfPartitions, labels = FALSE)
      )[[whichPartition]]
  }

  xmFull = (xp[1:(length(xp) - 1)] + xp[2:length(xp)])/2

  area = 0

  for (x1 in xmSlice)
  {
    for (x2 in xmFull)
    {
      for (x3 in xmFull)
      {
        for (x4 in xmFull)
        {
          for (x5 in xmFull)
          {
            for (x6 in xmFull)
            {
              for (x7 in xmFull)
              {
                for (x8 in xmFull)
                {
```

```
                    for (x9 in xmFull)
                    {
                      for (x10 in xmFull)
                      {
                        area = area + my10DFun(
                            x1, x2, x3, x4, x5,
                            x6, x7, x8, x9, x10
                            )
                      }
                    }
                  }
                }
              }
            }
          }
        }
      }

      area = area * deltaX10D^10 # Multiply by the Volume

      return(area)
}
```

A.3.2 Code to Build Partitions and Compute Midpoint using Nested For Loops Using 3 Raspberry Pi Compute Modules

Listing 15: Utilize nested for loops to compute the midpoint rule. This, of course, is a slower version but less RAM intensive. This is for 3 Raspberry Pi compute modules.

```
# File name: midPoint10DVer2.R
#
# When numberOfPartitions > 0, then split xmSlice1 and xMSlice2 into 1 or more
# pieces.
# This is used for parallel processes so that the data can be split to
# multiple CPUs / cores.
# whichPartition1 and whichPartition2 are indicators of which of the
# 1, 2, ... numberOfPartitions should be used.
midPoint10D = function(
  a, b, my10N, numberOfPartitions,
  whichPartition, whichPartition2
  )
{
  deltaX10D = (b-a) / my10N

  xp = seq(a, b,by=deltaX10D)

  xmSlice = (xp[1:(length(xp) - 1)] + xp[2:length(xp)])/2

  # Select a partition if positive
  if (numberOfPartitions > 0) {
    xmSlice1 = split(
      xmSlice, cut(seq_along(xmSlice),
      numberOfPartitions, labels = FALSE)
      )[[whichPartition]]
```

```
} else {
  xmSlice1 = xmSlice
}

if (whichPartition2 > 0) {
  xmSlice2 = split(
    xmSlice, cut(seq_along(xmSlice),
    numberOfPartitions, labels = FALSE)
    )[[whichPartition2]]
} else {
  xmSlice2 = xmSlice
}

xmFull = (xp[1:(length(xp) - 1)] + xp[2:length(xp)])/2

area = 0

for (x1 in xmSlice1)
{
  for (x2 in xmSlice2)
  {
    for (x3 in xmFull)
    {
      for (x4 in xmFull)
      {
        for (x5 in xmFull)
        {
          for (x6 in xmFull)
          {
            for (x7 in xmFull)
            {
              for (x8 in xmFull)
              {
                for (x9 in xmFull)
                {
                  for (x10 in xmFull)
                  {
                    area = area + my10DFun(
                      x1, x2, x3, x4, x5,
                      x6, x7, x8, x9, x10
                      )
                  }
                }
              }
            }
          }
        }
      }
    }
  }
}

area = area * deltaX10D^10 # Multiply by the Volume
```

```
  return(area)
}
```

A.3.3 Sequential Midpoint Method Using Nested For Loops

Listing 16: Code utilized to set up a sequential process for the ten-dimensional midpoint rule.

```
# File name: testMidPointRule.R
# Reality:  This is the first part of testMidPointRule.R which covers
# sequential on one core and one thread. The other parts use 2 and 3 cores,
# respectively.
#
# Load requred files
source('my10DFun.R')
source('midPoint10DVer2.R') # Slower but less memory required
source('midPointSequential.R')

library(future) # Parallel Processing when desired
library(microbenchmark) # Timing

# Build a sequential process on one machine
plan(sequential)

a = -1
b = 1
my10N = 6

numRepetitions = 10

mbmSequential <- microbenchmark(
  mySequentialMidResult <-
    midPointSequential(
      a, b, my10N, numberOfPartitions = 0
      ),
    times = numRepetitions,
    unit = 's'
    )

seqTiming = subset(
  summary(mbmSequential), select = -c(expr)
  )

seqTiming = cbind(
  Midpoint = round(mySequentialMidResult, 6),
  Computers = 1, Cores = 1, N = my10N, seqTiming
  )

midpointMethodTiming = seqTiming
```

A.3.4 Parallel Midpoint Method Using Nested For Loops and Two Cores

Listing 17: Code utilized to set up a parallel process for the ten-dimensional midpoint rule. Nested for loops with two cores are utilized.

```
# File name: testMidPointRule.R
```

```
# Reality:  This is the second part of testMidPointRule.R which covers
# sequential on one core and one thread. The other parts use 1 and 3 cores,
# respectively.
#
# Load requred files
source('my10DFun.R')
source('midPoint10DVer2.R') # Slower but less memory required
source('midPointMulti2.R')

library(future) # Parallel Processing when desired
library(microbenchmark) # Timing

# Build a multisession process on one machine
plan(multisession, workers = 2)

a = -1
b = 1
my10N = 6

mbmMulti2 <- microbenchmark(
  myMultiMidResult2 <- midPointMulti2(a, b, my10N, numberOfPartitions = 2),
  times = numRepetitions,
  unit = 's'
)

multiTiming2 = subset(
  summary(mbmMulti2), select = -c(expr)
  )

multiTiming2 = cbind(
  Midpoint = round(myMultiMidResult2, 6),
  Computers = 1, Cores = 2, N = my10N, multiTiming2
  )
```

A.3.5 Parallel Midpoint Method Using Nested For Loops and Three Cores

Listing 18: Code utilized to set up a parallel process for the ten-dimensional midpoint rule. Nested for loops with three cores are utilized.

```
# File name: testMidPointRule.R
# Reality:  This is the third part of testMidPointRule.R which covers
# sequential on one core and one thread. The other parts use 1 and 2 cores,
# respectively.
#
# Load requred files
source('my10DFun.R')
source('midPoint10DVer2.R') # Slower but less memory required
source('midPointMulti3.R')

library(future) # Parallel Processing when desired
library(microbenchmark) # Timing

# Build a multisession process on one machine
plan(multisession, workers = 3)
```

```
a = -1
b = 1
my10N = 6

mbmMulti3 <- microbenchmark(
  myMultiMidResult3 <- midPointMulti3(a, b, my10N, numberOfPartitions = 3),
  times = numRepetitions,
  unit = 's'
)

multiTiming3 = subset(
  summary(mbmMulti3), select = -c(expr)
  )

multiTiming3 = cbind(
  Midpoint = round(myMultiMidResult3, 6),
  Computers = 1, Cores = 3, N = my10N, multiTiming3
  )
```

A.4 Code for Clusters of Two Raspberry Pi Compute Modules Using Nested For Loops

A.4.1 Code to Build Partitions and Compute Midpoint using Nested For Loops with Two Computers and One Core

Listing 19: Utilize nested for loops to compute the midpoint rule. This, of course, is a slower version but less RAM intensive. Two Raspberry Pi compute modules and one core are used.

```
# File name: testMidPointRuleCluster.R
# Reality:  This is the first part of testMidPointRuleCluster.R which covers
# 2 Raspberry Pi compute modules with 1 core each. The other parts use 2 and 3
# cores each.
#
# Load requred files
source('my10DFun.R')
source('midPoint10DVer2.R') # Slower but less memory required
source('midPointMulti2.R')

library(future) # Parallel Processing when desired
library(microbenchmark) # Timing

# Build a  process on two machines, one core each
plan(cluster,
       workers = c('localhost', '10.0.0.3')
       )

a = -1
b = 1
my10N = 6

numRepetitions = 10

mbmCluster <- microbenchmark(
  myClusterMidResult <- midPointMulti2(a, b, my10N, numberOfPartitions = 2),
  times = numRepetitions,
```

```
  unit = 's'
  )

clusterTiming = subset(
  summary(mbmCluster), select = -c(expr)
  )

mbmMidpointMethodClusterTiming = cbind(
  Midpoint = round(myClusterMidResult, 6),
  Computers = 2, Cores = 1, N = my10N, clusterTiming
  )
```

A.4.2 Code to Build Partitions and Compute Midpoint using Nested For Loops with Two Computers and Two Cores

Listing 20: Utilize nested for loops to compute the midpoint rule. This, of course, is a slower version but less RAM intensive. Two Raspberry Pi compute modules and two cores are used.

```
# File name: testMidPointRuleCluster.R
# Reality:  This is the second part of testMidPointRuleCluster.R which covers
# 2 Raspberry Pi compute modules with 2 cores each. The other parts use 1 and 3
# cores each.
#
# Load requred files
source('my10DFun.R')
source('midPoint10DVer2.R') # Slower but less memory required
source('midPointMulti4.R')

library(future) # Parallel Processing when desired
library(microbenchmark) # Timing

# Build a  process on two machines, two cores each

plan(cluster,
      workers = c('localhost', '10.0.0.3', 'localhost', '10.0.0.3')
      )

a = -1
b = 1
my10N = 6

numRepetitions = 10

mbmCluster2Pi2Core <- microbenchmark(
  myClusterMidResult2Pi2Core <-
    midPointMulti4(a, b, my10N, numberOfPartitions = 4),
    times = numRepetitions,
    unit = 's'
  )

clusterTiming2Pi2Core = subset(
  summary(mbmCluster2Pi2Core), select = -c(expr)
  )
```

```
mbmMidpointMethodClusterTiming2Pi2Core = cbind(
  Midpoint = round(myClusterMidResult2Pi2Core, 6),
  Computers = 2, Cores = 2, N = my10N, clusterTiming2Pi2Core
  )
```

A.4.3 Code to Build Partitions and Compute Midpoint using Nested For Loops with Two Computers and Three Cores

Listing 21: Utilize nested for loops to compute the midpoint rule. This, of course, is a slower version but less RAM intensive. Two Raspberry Pi compute modules and three cores are used.

```
# File name: testMidPointRuleCluster.R
# Reality:  This is the third part of testMidPointRuleCluster.R which covers
# 2 Raspberry Pi compute modules with 3 cores each. The other parts use 1 and 2
# cores each.
#
# Load requred files
source('my10DFun.R')
source('midPoint10DVer2.R') # Slower but less memory required
source('midPointMulti6.R')

library(future) # Parallel Processing when desired
library(microbenchmark) # Timing

# Build a  process on two machines, two cores each
plan(cluster,
       workers = c(
         'localhost', '10.0.0.3',
         'localhost', '10.0.0.3',
         'localhost', '10.0.0.3'
         )
       )

a = -1
b = 1
my10N = 6

numRepetitions = 10

mbmCluster2Pi3Core <- microbenchmark(
  myClusterMidResult2Pi3Core <-
    midPointMulti6(a, b, my10N, numberOfPartitions = 6),
    times = numRepetitions,
    unit = 's'
    )

clusterTiming2Pi3Core = subset(
  summary(mbmCluster2Pi3Core), select = -c(expr)
  )

mbmMidpointMethodClusterTiming2Pi3Core = cbind(
  Midpoint = round(myClusterMidResult2Pi3Core, 6),
  Computers = 2, Cores = 3, N = my10N, clusterTiming2Pi3Core
  )
```

A.4.4 Code to Build Partitions and Compute Midpoint using Nested For Loops with Three Computers and One Core Each

Listing 22: Utilize nested for loops to compute the midpoint rule. This, of course, is a slower version but less RAM intensive. Three Raspberry Pi compute modules and one core each are used.

```
# File name: testMidPointRuleCluster3Pi.R
# Reality:  This is the first part of testMidPointRuleCluster3Pi.R which covers
# 3 Raspberry Pi compute modules with 1 core each. The other parts use 2 and 3
# cores each.
#
# Load requred files
source('my10DFun.R')
source('midPoint10DVer2For3Pi.R') # Slower for use with 3 Pi's
source('midPointMulti3For3Pi.R')

library(future) # Parallel Processing when desired
library(microbenchmark) # Timing

# Build a  process on three machines, one core each
plan(cluster,
      workers = c('localhost', '10.0.0.3', '10.0.0.4')
      )

a = -1
b = 1
my10N = 6

numRepetitions = 10

mbmCluster <- microbenchmark(
  myClusterMidResult <- midPointMulti3(a, b, my10N, numberOfPartitions = 3),
  times = numRepetitions,
  unit = 's'
  )

clusterTiming = subset(
  summary(mbmCluster), select = -c(expr)
  )

mbmMidpointMethodClusterTiming3PiDF = cbind(
  Midpoint = round(myClusterMidResult, 6),
  Computers = 3, Cores = 1, N = my10N, clusterTiming
  )
```

A.4.5 Code to Build Partitions and Compute Midpoint using Nested For Loops with Three Computers and Two Cores Each

Listing 23: Utilize nested for loops to compute the midpoint rule. This, of course, is a slower version but less RAM intensive. Three Raspberry Pi compute modules and two cores each are used.

```
# File name: testMidPointRuleCluster3Pi.R
# Reality:  This is the second part of testMidPointRuleCluster3Pi.R which covers
# 3 Raspberry Pi compute modules with 1 core each. The other parts use 1 and 3
# cores each.
```

```
#
# Load requred files
source('my10DFun.R')
source('midPoint10DVer2For3Pi.R') # Slower for use with 3 Pi's
source('midPointMulti6For3Pi.R')

library(future) # Parallel Processing when desired
library(microbenchmark) # Timing

# Build a  process on three machines, two cores each
plan(cluster,
      workers = c(
        'localhost', '10.0.0.3', '10.0.0.4',
        'localhost', '10.0.0.3', '10.0.0.4'
        )
      )

a = -1
b = 1
my10N = 6

numRepetitions = 10

mbmCluster3Pi2Core <- microbenchmark(
  myClusterMidResult3Pi2Core <-
    midPointMulti6(a, b, my10N, numberOfPartitions = 6),
  times = numRepetitions,
  unit = 's'
  )

clusterTiming3Pi2Core = subset(
  summary(mbmCluster3Pi2Core), select = -c(expr)
  )

mbmMidpointMethodClusterTiming3Pi2Core = cbind(
  Midpoint = round(myClusterMidResult3Pi2Core, 6),
  Computers = 3, Cores = 2, N = my10N, clusterTiming3Pi2Core
  )
```

A.4.6 Code to Build Partitions and Compute Midpoint using Nested For Loops with Three Computers and Three Cores Each

Listing 24: Utilize nested for loops to compute the midpoint rule. This, of course, is a slower version but less RAM intensive. Three Raspberry Pi compute modules and three cores each are used.

```
# File name: testMidPointRuleCluster3Pi.R
# Reality:  This is the third part of testMidPointRuleCluster3Pi.R which covers
# 3 Raspberry Pi compute modules with 1 core each. The other parts use 1 and 2
# cores each.
#
# Load requred files
source('my10DFun.R')
source('midPoint10DVer2For3Pi.R') # Slower for use with 3 Pi's
source('midPointMulti3For3Pi.R')
```

```
source('midPointMulti9For3Pi.R')

library(future) # Parallel Processing when desired
library(microbenchmark) # Timing

# Build a  process on three machines, three cores each
plan(cluster,
       workers = c(
         'localhost', '10.0.0.3', '10.0.0.4',
         'localhost', '10.0.0.3', '10.0.0.4',
         'localhost', '10.0.0.3', '10.0.0.4'
         )
       )

a = -1
b = 1
my10N = 6

numRepetitions = 10

mbmCluster3Pi3Core <- microbenchmark(
  myClusterMidResult3Pi3Core <-
    midPointMulti9(a, b, my10N, numberOfPartitions = 9),
  times = numRepetitions,
  unit = 's'
  )

clusterTiming3Pi3Core = subset(
  summary(mbmCluster3Pi3Core), select = -c(expr)
  )

mbmMidpointMethodClusterTiming3Pi3Core = cbind(
  Midpoint = round(myClusterMidResult3Pi3Core, 6),
  Computers = 3, Cores = 3, N = my10N, clusterTiming3Pi3Core
  )
```

References

Henrik Bengtsson. future: Unified parallel and distributed processing in r for everyone, 2024. URL https://cran.r-project.org/web/packages/future/index.html.

Paul DuChateau and David Zachman. *Applied Partial Differential Equations*. Dover Publications, Inc., 1 edition, 2002.

Keith E. Emmert and Peter White. Parallel computing using r and a raspberry pi cluster part i: The need for speed! In *Proceedings of the International Conference on Technology in Collegiate Mathematics, ICTCM 2024*, New York, NY, USA, 2024. Pearson.

Joel R. Hass, Christopher E. Heil, Maurice D. Weir, and Przemyslaw Bogacki. *Thomas' Calculus*. Pearson, 15 edition, 2022.

Andrew Heiss. Create a cheap, disposable supercomputer with r, digitalocean, and future, 2008. URL https://www.andrewheiss.com/blog/2018/07/30/disposable-supercomputer-future/.

William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brin P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Pearson, 2 edition, 1992.

Rasmurtech. Raspberry pi cluster, 2024. URL https://www.instructables.com/Raspberry-Pi-Cluster/.

Raspberry Pi Tutorial. How to build a raspberry pi cluster, 2024. URL https://www.raspberrypi.com/tutorials/cluster-raspberry-pi-tutorial/.

Dennis G. Zill. *A First Course in Differential Equations with Modeling Applications*. Cengage, 12 edition, 2024.