

# EP3 Report - PPD - UFSCar 2025.1

Student: Raphael Alessander Prado dos Santos

Professor: Ph.D Hermes Senger

May 11th, 2025

## 1 Introduction

The Laplace heat transfer equation can be solved using an iterative 2D stencil method. The absence of data dependencies between the data points (read, each matrix element) to be computed allows us to parallelize this problem, drastically improving the time needed to approximate the result for this data set.

This report presents three different methods of parallelizing this algorithm (contiguous data distribution, Round-Robin, and Demand-based data distribution) and, for each method, two different ways to handle the data update in each iteration (copy and pointer-swap) and two different ways to implement 2D arrays (2D-Matrix and linearized), totaling twelve distinct approaches to this problem.

The C language and the OpenMP library are used to implement the different parallel codes used in this report, highlighting performance results and the best optimization strategies considering data locality and workload distribution between threads.

All tests were performed in the UFSCar Cluster, using varying numbers of threads to test how well the code scales. Overall, the pointer-based algorithms show a speedup of approximately thirty-nine (46) times against the best serial Jacobi implementation while having the same memory complexity, displaying how the proposed optimizations achieved staggering improvements in execution time. This follows the trends established in the first report that adopted the PThreads library as the parallelism implementation tool. It's important to note that all the proposed data locality and pointer-swap optimizations were also implemented in the serial version.

## 2 Parallelization Strategy and Implementation Details

The implementation builds upon the sequential version by reusing key elements such as the initialization functions, stencil computation, and grid update logic. The main differences lie in four core strategies: data locality, chunk-based division, matrix linearization, and efficient grid swapping.

### 2.1 Matrix Linearization and Data Locality

To improve memory access efficiency, the 2D matrix is linearized into a 1D array. This reduces the overhead of double-pointer referencing in traditional 2D arrays. The matrix is accessed via:

$$A_{i,j} = A'[i \times \text{len} + j]$$

where `len` is the number of columns, and  $i, j$  are the row and column indices, respectively.

## 2.2 Chunk Division and Thread Assignment

The grid is divided into fixed-size chunks, each representing a matrix row. Even though there's a possibility for creating multi-row chunks (one strategy being to use powers of two as the number of chunks), this creates another layer of complexity for both the algorithms and the test samples.

The most basic implementation (Regular or R) assigns a chunk of continuous rows to each thread, so that the work is evenly divided but not perfectly, as the number of rows may not be a multiple of the number of threads. As such, the last thread computes the remainder rows. Equations 1 and 2 show how the chunks are divided for each thread

$$C_{i,start} = i * S + 1 \quad (1)$$

$$C_{i,end} = C_{i,start} + S \quad (2)$$

where  $i$  is the thread id = 0, ...,  $n-1$  for  $n$  threads,  $S$  is the chunk size ( $S = \lfloor (R - 2)/T \rfloor$ , where  $R$  is the number of rows and  $T$  is the number of threads). In this context,  $C_i$  represents the chunk assigned to a thread of id =  $i$ . Since the number of rows may not be a multiple of the number of threads, any remaining rows that would not be included using only equations 1 and 2 are added to the last chunk.

The second implementation, called Demand (D), uses a pseudo-producer-consumer approach, so that after finishing computing its current row, the thread finds the next available row. Appendix B has the implementation of the selection mechanism that the threads use to obtain the next chunk for computation. It's worth noting that, in this case, as with the Round-Robin approach, each chunk represents only one row. Even though multi-row chunks can be used, this creates the need for the optimization of the chunk size for each execution, which would not fall under the objectives of this work, while also increasing the test samples needed. Finally, the Round-Robin approach follows a straightforward implementation, where each thread computes the rows with offset indexes based on the number of threads.

The program supports two configurable parameters at runtime:

- Matrix size ( $n \times n$ ): Defines the problem size.
- Number of threads: Determines how many parallel workers are launched.

This flexibility allows the user to test how different combinations affect performance and memory usage.

## 2.3 Stencil Calculation

OpenMP differs from the PThreads library as it uses implicit parallelization, as the program is parallelized with *pragma omp* calls. As such, the programmer doesn't have full control over how parallelism is implemented, as opposed to Pthreads, MPI, and the CUDA libraries. Furthermore, OpenMP provides a large set of personalization calls that allow the programmer to achieve the same strategies that were implemented in PThreads on the EP2 report. One should note that it's possible to mimic on OpenMP how PThreads implements parallelism, doing a more hands-on

approach, where the user has more control over how the program is parallelized, but since the main feature of OpenMP is how parallelism can be achieved using simple *pragma omp* calls, this is the methodology that was used during this assignment.

Each thread computes the stencil operation on its assigned chunks and writes the results to the new grid array and annotates the error, as defined in Appendix A: Stencil Computation Code. All three implementations use the same base calls OpenMP: *#pragma omp parallel for reduction(max err)*. However, they do differ on additional commands assigned to each implementation, as a way of dividing the workload among the threads.

The regular implementation uses an additional *collapse(2)* call, in which the array is evenly divided between the threads, just as the first implementation did. In this context, the *collapse(2)* adds a layer of parallelization where there are continuous chunks assigned to a single thread.

The Demand-based and Round-Robin approaches used, respectively, *#pragma omp parallel for reduction(max err) schedule(dynamic, 1)* and *#pragma omp parallel for reduction(max err) schedule(static)* calls. This allowed the program to operate as expected, while needing no further modifications to the code. This is useful in the sense that new strategies can be tested using this base framework, with customization of parallel implementation.

Finally, the Barrier and Mutex calls are not explicitly needed here, as OpenPM handles how the variables update each value. This is most notable for the *reduction()* call, where *max* is the logic behind the reduction and *err* is the variable that suffers this operation. One can note that this follows a similar logic as MPI reduce operations, but that needs to be manually implemented in PThreads.

After calculating the stencil, changes on how the grid is updated are made in the *main()* function, which sets it apart from the PThreads implementation, as there was a need for a specific function call to do these updates.

### 3 Experimental Setup

In the EP2 assignment, a smaller matrix 5k dimension matrix was used. As this assignment builds upon the results found in that report, there was an increase in thread and dimension values for this assignment, which can be checked in the following.

#### Execution Parameters:

- Matrix sizes tested:  $10,000 \times 10,000$ .
- Thread counts: 1 (serial), 20, 40, 60, 80, 100, 120.
- Processing: a node from the UFSCar Cluster with 128 available threads.

This thread count variation was used to check how well the different algorithms scale. Even though different chunk sizes can be tested, this is beyond the scope of this report, but it's an interesting metric for future works, as the task of testing this code for variable chunk sizes would take more time than what is available for this assignment. For the sake of simplicity, each row is a chunk in this test.

## 4 Results

### 4.1 Serial Improvements

As Table 1 shows, there's a maximum speedup of 1.323 by only changing the way the data is updated when using matrix linearization and pointer-swapping, while only using Pointer-swapping renders similar results, with a 1.236 speedup.

Table 1: Execution Time, Speedup, and Efficiency

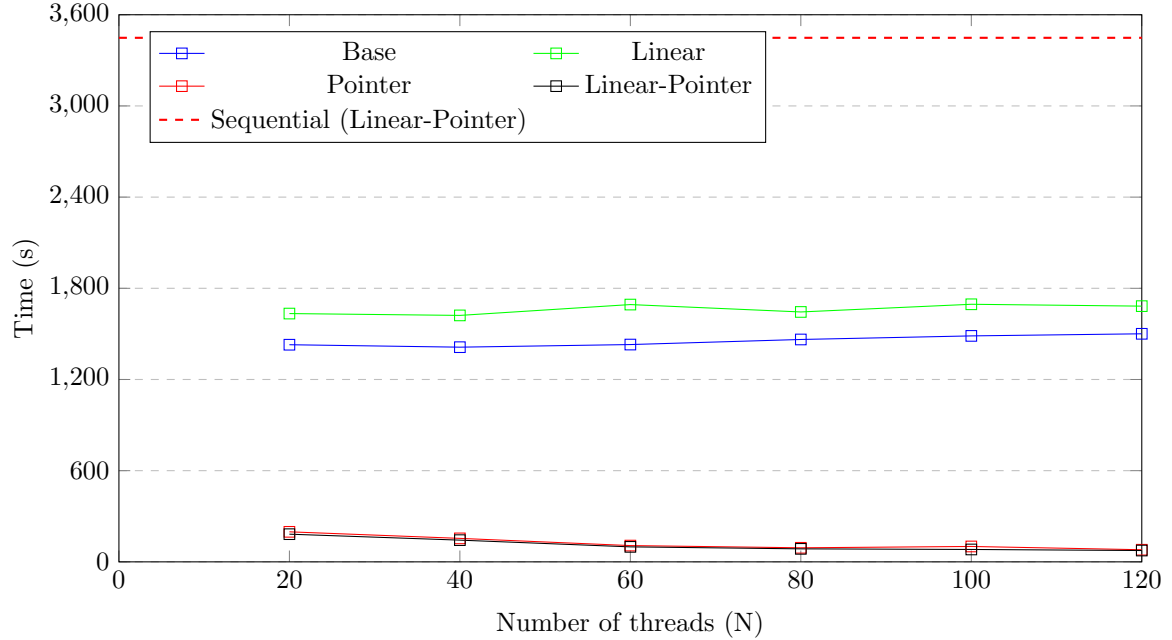
Strategy	Time (s)	Speedup*
Base	4,560.47	—
Linear	4,673.94	0.976
Pointer	3,689.89	1.236
Linear+Pointer	3,448.22	1.323

It should be noted that these results are from a 10k x 10k matrix and, as such, it's expected that this gap increases with the matrix dimensions.

### 4.2 Parallel Improvements

The entire results from the tests are available in the Appendix D table. Since these results follow the same logic as the PThreads implementation in the EP2 assignment, it's notable how figures 1, 2, and 3 show a continuous gap between the pointer and non-pointer swapping implementations.

Figure 1: Execution time for Parallel Regular Algorithm

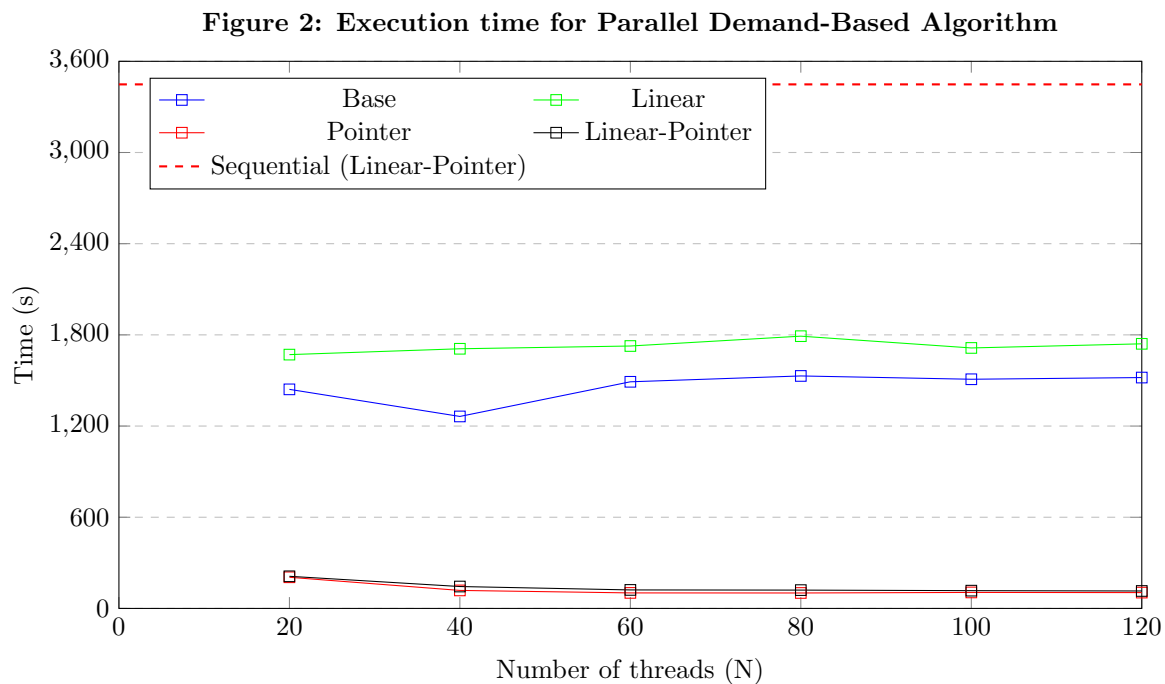


The tendency for the Pointer and Linear + Pointer variants of the Base approach to have the best results is shown in Table 2, with only a small margin of difference between them in terms of time, speedup, and efficiency. There also appears to be a linearized performance from both base and linear implementations, such that the execution plateau already seems to have been achieved. Overall, the Linear + Pointer implementations present better results than those obtained for the same strategy, but while using PThreads.

Table 2: Execution Time, Speedup, and Efficiency for the Regular approach

Strategy [Threads]	Time (s)	Speedup		Efficiency	
		Base	L+P*	Base	L+P*
Base [40]	1412.60	3.23	2.44	0.081	0.061
Linear [40]	1621.64	2.81	2.13	0.070	0.053
Pointer [120]	79.73	57.20	43.25	0.477	0.360
Linear + Pointer [120]	74.47	61.24	46.30	0.510	0.386

Figure 2 shows an initial drop in execution time for the base implementation, but followed by a stabilization tendency in execution time. Overall, all four approaches follow the same pattern as the Base algorithm, even with the thread counts, as Linear is the worst-performing implementation, followed by Base, Pointer, and Linear + Pointer, respectively.



An analysis of Table 3 shows that the Pointer-only implementation achieved the minimum execution time as well as the best efficiency results, which is different from the EP2 assignment,

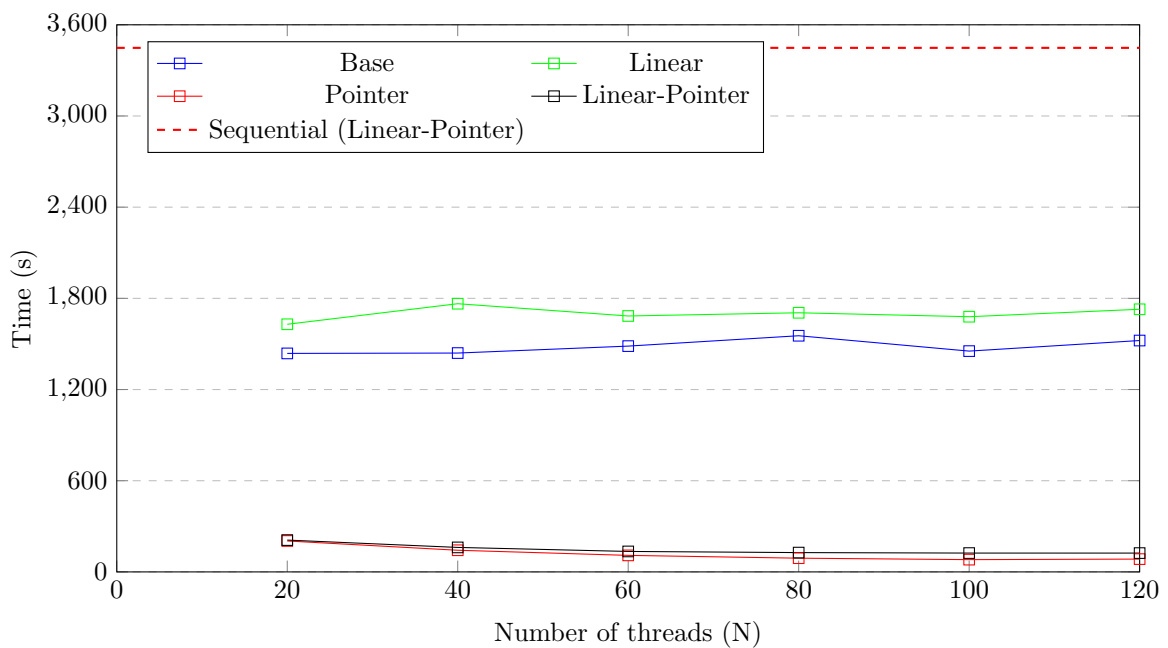
where Pointer and Linear + Pointer implementations had, respectively, the best speedup and best efficiency.

Table 3: Execution Time, Speedup, and Efficiency for the Demand-based approach

Strategy [Threads]	Time (s)	Speedup		Efficiency	
		Base	L+P*	Base	L+P*
Base [40]	1263.29	3.61	2.73	0.090	0.068
Linear [20]	1670.07	2.73	2.06	0.068	0.052
Pointer [80]	101.67	44.86	33.92	0.561	0.424
Linear + Pointer [120]	114.80	39.72	30.04	0.497	0.375

Even though there are some variations in execution time for both Base and Linear implementations, they still show the same pattern as the PThreads implementations and the other OpenMP strategies. Besides, there's a small variance for both during thread count variation. Finally, there's a clear advantage of Pointer only against Linear + Pointer, as from 40 threads onwards, a small gap begins to form between them.

Figure 3: Execution time for Parallel Round-Robin Based Algorithm



What separates Round-Robin from the other implementations is how well the Pointer-only variation performs when compared to the Linear + Pointer. While there's a small difference between the execution time for these implementations when compared to other strategies, in this case, there's over 40% increase in execution time, which results in worse efficiency and speedup for the Linear + Pointer alternative.

Table 4: Execution Time, Speedup, and Efficiency for the Round-Robin approach

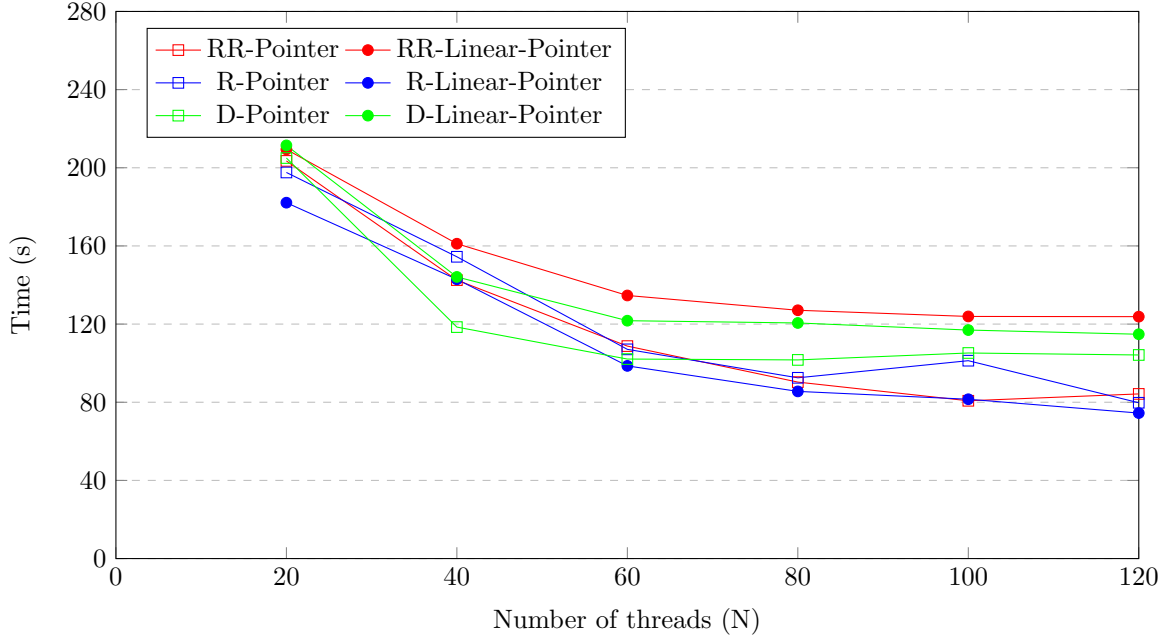
Strategy [Threads]	Time (s)	Speedup		Efficiency	
		Base	L+P*	Base	L+P*
Base [20]	1437.72	3.17	2.40	0.079	0.060
Linear [20]	1629.41	2.80	2.12	0.070	0.053
Pointer [100]	80.75	56.47	42.70	0.471	0.534
Linear + Pointer [120]	123.81	36.84	27.85	0.307	0.348

Overall, the Regular strategy achieved the best execution time (as can be noted in Figure 4), and speedup, while in terms of efficiency, the Round-Robin strategy achieved better results.

The linearized trend for execution time was noticed during these tests and confirms the trend established in the EP2 Report. Alas, it's worthwhile to point out that there's a plateau in execution time for both the Base and Linear-Only implementations. After hitting the 40 threads mark, an increasing thread count does not cause a significant increase or decrease in execution, as there's an average of 9.75% difference between the execution time for 20 and 120 threads, as can be seen in the Appendix D, with the Demand-Base approach being the exception, showing a 21.09% variation.

The same cannot be said for the Pointer and Linear + Pointer implementations, where the average increase is 116.56%. One should note that this increase does not reflect the increasing amount of threads being assigned to the problem, meaning that the efficiency does not improve with the execution time.

Figure 4: Execution time of All Parallel Algorithms using Pointer and Linear-Pointer Methods



Finally, all implementations that used the pointer-swapping technique provided better performance results, as Table 5 shows. While the Base and Linear-Only strategies showed a limitation

when increasing threads to achieve better performance, the Pointer and Linear-Pointer methods allowed the program to decrease the execution time, even though with a worse efficiency.

Table 5: Execution Time, Speedup, and Efficiency for the OpenMP implementation

Number of Threads	Strategy	Time (s)	Spedup	Efficiency
20	R-Linear + Pointer	182.13	18.933	0.947
40	D-Pointer	118.47	29.106	0.728
60	R-Linear + Pointer	98.61	34.968	0.583
80	R-Linear + Pointer	85.56	40.303	0.504
100	RR-Pointer	80.75	42.701	0.427
120	R-Linear + Pointer	74.47	46.304	0.386

If the Regular strategy, which produced the best results in all three strategies, has all of its implementations compared in terms of execution time, Speedup, and efficiency, as displayed in Appendix D, there's a 144.57% increase in Speedup, while a 60% decrease in efficiency can be calculated.

Table 6: Execution Time, Speedup, and Efficiency for the PThreads implementation

Number of Threads	Strategy	Time (s)	Spedup	Efficiency
20	R-Linear + Pointer	181.83	18.964	0.948
40	D-Pointer	117.21	29.426	0.735
60	R-Linear + Pointer	99.06	34.809	0.580
80	R-Linear + Pointer	81.04	42.552	0.532
100	R-Linear + Pointer	81.58	42.269	0.423
120	R-Linear + Pointer	75.65	45.579	0.380

A look into Table 6 will show that there's only a small difference between the results for both implementations (OpenMP and PThreads), as these results are from running the program with a 10k x 10k matrix.

## 5 Discussion

Following from the results obtained during the EP2 assignment, it becomes clear that, when having to opt between the two libraries (PThreads and OpenMP), the one with the simplest code should be preferred if the results stay the same, with simple in this context being the code that has less points of failure and that is easy to adapt to new contexts. As such, OpenMP can be appointed as the best performing architecture, as it's able to run slightly faster. However, the difference is so insignificant that either implementation can be used.

One example of OpenMP that shows how easy it's to personalize it, when compared to PThreads, is that if one wants to change the chunk size, all it has to do is change the parameters of the parallel calls. When using PThreads, however, there are more issues that arise when attempting this modification.

## 6 Conclusion

This report expands on the work done in the EP2 assignment, in the sense that the Linear + Pointer implementation is the better-performing one overall. Even in the cases where it doesn't have the best performance, as Table 5 lists the Demand and Round-Robin implementations as the best performing for 40 and 100 threads, respectively, the difference is not an issue.

The major observation that can be made for the OpenMP implementation is how well it performs, as it's an abstract parallelism tool and, as such, should have some drawbacks, like providing less performance enhancement because the underlying parallel model is not accessible to the programmer.

## 7 Appendix

### Appendix A: Thread Manager function

```
void* thread_manager(void *arg) {

    t_data *thread_info = (t_data*)arg;
    double local_error = 0.0;
    int i;

    do {
        local_error = 0.0;

        for(i = thread_info->begin; i < thread_info->end; i++) {
            local_error = max(local_error , stencil(i));
        }

        error_buffer[thread_info->id] = local_error;

        pthread_barrier_wait(&barrier);

        if(thread_info->id == 0)
            update_grid();

        pthread_barrier_wait(&barrier);
    } while ( err > CONV_THRESHOLD && iter <= ITER_MAX );
}
```

### Appendix B: Demand-Based Workload Distribution

```
i = 1;
while(true) {
    pthread_mutex_lock(&row_selec);
    i = next_row;
    next_row++;
    pthread_mutex_unlock(&row_selec);
}
```

```

        if(i < size - 1)
            local_error = max(local_error , stencil(i));
        else
            break;
    }

```

#### Appendix C: Round-Robin Workload Distribution

```

    for(i = thread_info->id+1; i < size-1; i+=n_threads) {
        local_error = max(local_error , stencil(i));
    }

```

#### Appendix D: Test Data

Metric	Demand				Regular				Round-Robin			
	Base	Linear	Pointer	L + P	Base	Linear	Pointer	L + P	Base	Linear	Pointer	L + P
Threads												
5	420.49	449.00	102.06	93.89	419.87	486.16	94.89	87.11	449.47	454.64	103.61	94.61
10	465.83	597.43	56.82	58.05	563.63	542.06	56.12	47.72	560.04	540.59	53.62	53.85
20	477.17	568.99	36.78	47.23	543.60	572.00	45.23	34.69	577.09	577.12	42.69	47.85
40	583.60	608.22	35.02	41.51	548.66	586.40	21.98	21.70	550.66	605.48	26.96	40.60
80	593.89	605.69	34.36	41.68	573.48	561.53	23.24	23.86	564.93	601.14	25.67	38.32
<i>S</i> Worst Serial	2.71	2.54	<b>33.19</b>	27.48	2.72	2.35	51.88	<b>52.57</b>	2.54	2.51	<b>44.43</b>	29.76
<i>S</i> Best Serial	2.02	1.89	<b>24.74</b>	20.48	2.02	1.75	38.67	<b>39.18</b>	1.89	1.87	<b>33.12</b>	22.18
<i>E</i> Worst Serial	0.542	0.508	<b>0.415</b>	<b>0.687</b>	0.543	0.469	1.297	<b>1.314</b>	0.507	0.502	<b>0.555</b>	0.372
<i>E</i> Best Serial	0.404	0.379	0.309	<b>0.512</b>	0.405	0.350	0.967	<b>0.980</b>	0.378	0.374	<b>0.414</b>	0.277
$\rho$ (Worst Serial)				7.66%				39.97%				5.61%
$\rho$ (Best Serial)				5.71%				29.79%				4.18%