PARALLEL GRAPH ALGORITHMS IN INTEL DPC++

Introduction

Many graph algorithms can have incredibly long runtimes. Parallelising these graph algorithms is one way we can reduce the long runtimes these algorithms take. There are several parallel languages we can consider to parallelise code. The main options to consider are DPC++, OpenCL, CUDA, or manually creating a threaded program. DPC++ is used in this project because it is based on C++ which is a higher level than OpenCL's C based approach.

Data Parallel C++ (DPC++) is a new open source cross platform language built upon the ISO C++ and Khronos Group's SYCL specifications. It is an open-source implementation of the oneAPI specification to make heterogeneous computation more accessible and widespread.

A key aspect of SYCL parallel languages such as OpenCL and DPC++, is that the code is structured as a Host Kernel model. The host kernel model consists of two distinct sections of code, the host and the kernel code. The host orchestrates the program flow, as well as any code requiring the full C++17 standard, whilst the kernel code executes computationally intensive code.

As DPC++ is a SYCL implementation, it focuses on heterogeneous computation on a wide variety of accelerator devices, including GPUs, GPGPUs, CPUs, Neural Network Accelerators and FPGAs as well. This means that a different programming model of parallel computation is required to make DPC++ programs portable between all these devices. DPC++ doesn't use threads as traditionally implemented in other parallel frameworks, rather the mapping of work to various processors/threads/cores is done in abstract.

Algorithms

All Pairs Shortest Path Algorithms

All Pairs shortest path is a subset of the shortest path problem, of which applies to all pairs of nodes inside a graph. Often these algorithms are greedy and have high time complexity, as the general operation of these algorithms requires visiting each node in the matrix and comparing the adjacent nodes. Thus, generally time complexities range in the O(V^3) to O(V^2); this high time complexity and the fact that many of these algorithms operate on matrices, make All Pairs Shortest Path a very viable target for parallelism.

Dijkstra's Algorithm

Djikstra’s Shortest Path Algorithm is one of the most well-known algorithms for obtaining the shortest path from one node to every other node within a graph. The algorithm checks for adjacent nodes to the current node and updates the path from the source node to the adjacent node if the new path is shorter than all previous paths.

Single Source Shortest Path (SSSP) algorithms are generally difficult to parallelize effectively and produce a significant speedup compared to more complex algorithms since the computational workload allocated to each processor per allocation is very small compared to the overhead. Comparatively, All Pairs Shortest Path (APSP) algorithms are usually equally easy to understand, but require much more computation, making them generally more suitable for parallelization.

The caveat of using Djikstra for APSP is that the algorithm is inherently very sequential. This means that even in the sequential implementation of this algorithm, there is no guarantee that all pairs will have their shortest paths calculated within one iteration of the algorithm. Additionally, the sequential and parallel versions of the algorithm will produce different results due to the algorithm’s sequential nature. To combat both of these issues, both parallel and sequential implementations are run to completion five times.

One method of parallelizing Djikstra is to parallelize the inner loop, which updates the distance of new nodes from the source node. This method is highly inefficient as the amount of overhead will far outweigh the amount of calculation during the execution time. The method we have chosen instead is to parallelize the SSSP operations of Djikstra’s Shortest Path Algorithm. Each thread will have its own copy of the graph to modify and write to the original graph at the end. This implementation has the lowest amount of overhead without wasting any calculations, which would occur if all five iterations of the complete APSP Dijkstra algorithm were run in parallel.

Floyd Warshall

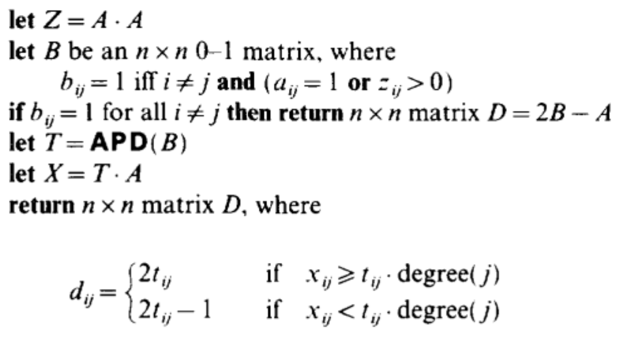
Floyd Warshall is another all pairs shortest path algorithm. The algorithm compares all possible paths between two pairs of nodes. Through this operation, all edges of the graph are checked, and estimated to the smallest weighted edge. The algorithm will continue through the entire matrix, estimating the shortest path for all pairs of nodes in the graph.

This manifests in implementation as 3 layered nested for loops, which iterate through the entire range of the matrix. This means that it has a O(V^3) and thus has potential to be easily parallelized.

Seidel's Algorithm

The final APSP algorithm that we investigated was Seidel's algorithm. Seidel’s algorithm is an algorithm made for use on undirected, unweighted graphs, unlike the previous algorithms that have been discussed, which have been weighted. Seidel's algorithm works by first computing the shortest paths between all vertices, which makes use of matrix multiplication, condition checking and other matrix operations such as matrix addition. Seidel's algorithm also makes use of recursion to operate on the input. The algorithm then makes use of a Las Vegas algorithm to reconstruct the shortest paths from this computation of the distances. Due to the use of a Las Vegas algorithm, which leads to uncertainty of implementation in DPC++, we decided to tackle the main computation of distances in the first half of the algorithm, which consists of matrix manipulations and recursion. The time complexity of this section of the algorithm, with the use of Coppersmith-Winograd fast matrix multiplication is O(n^2).

The pseudocode of the distances computation is shown in the original paper as:



```
**let** $Z = A \cdot A$
**let** $B$ be an $n \times n$ 0–1 matrix, where
    $b_{ij} = 1$ iff $i \neq j$ **and** $(a_{ij} = 1$ **or** $z_{ij} > 0)$
**if** $b_{ij} = 1$ for all $i \neq j$ **then return** $n \times n$ matrix $D = 2B - A$
**let** $T = \mathbf{APD}(B)$
**let** $X = T \cdot A$
**return** $n \times n$ matrix $D$, where
```

$$d_{ij} = \begin{cases} 2t_{ij} & \text{if } \quad x_{ij} \geqslant t_{ij} \cdot \text{degree}(j) \\ 2t_{ij} - 1 & \text{if } \quad x_{ij} < t_{ij} \cdot \text{degree}(j) \end{cases}$$

Figure 1: Seidel's Algorithm Pseudocode

Where the input matrix A is a square adjacency matrix of 0-1s. The input is subsequently multiplied to itself, and assigned to the Z Matrix. Another matrix is created, where the cells are filled with 1 or 0 based on the corresponding cells in input A and the Z matrix. This matrix is then checked if all cells apart from the diagonal cells are 1. If this is true, then the D matrix is returned with the result of 2 * B - A. If this condition is not met, then the function will recur, and the same operations previously described will be performed until the base case has been reached. Once the base case is reached, the X matrix is created with the result of the multiplication between the T and A matrix. Finally the final return matrix is created based on conditions from the matrix multiplication and the degree of input matrix.

The heavy use of matrix manipulations makes the algorithm easily parallelizable in each step of the algorithm. It also makes a good comparison between the other two algorithms as the similar data access patterns makes for a good algorithm to compare the two ASPS algorithms with.

Minimum-Spanning Tree Algorithms

A minimum-spanning tree (MST) is the path which connects all nodes on a graph, without any cycles and with the minimum possible total edge weight. That is, it is a spanning tree whose sum of edge weights is as small as possible.

Prim’s Algorithm

Prim’s algorithm is a greedy algorithm that finds a MST based on a particular starting point on a graph. The algorithm begins with a random node and then selects the lowest-weighted path to the next available node. In the event of a tie, a random path is chosen. This is repeated until there are no more available nodes.

It is a greedy algorithm because it makes the local optimal-choice in each iteration, the disadvantage of this is that it may not find the global-optimal solution. In the context of MSTs, the global-optimal solution is the MST with the shortest total path length. This is because multiple MSTs can be created depending on the starting location.

The main loop of the algorithm is not parallelizable because it is inherently sequential — You need to build the graph one node at a time. The inner loop however, is parallelizable. This is because the inner loop simply determines which, out of the available paths, has the minimum weight.

A major drawback of Prim’s algorithm is that the time taken to check for the smallest path-weight is slow for a large number of nodes; this is because, traditionally, each path-weight would have to be checked sequentially. Utilising parallel computing, a large amount of the path-weights can be ruled out in parallel as the program converges on the minimum weighted path.

Implementation

Most of our implementations of parallelizing the various algorithms use DPC++ with basic kernel implementations. As DPC++ has no traditional model of threading as we are familiar with, we opted for basic kernels over more complex ND-Range kernels or Hierarchical kernels. After consulting various resources related to DPC++ such as “Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL.”, the consensus was to opt for basic kernels. With more experience and deeper understanding of DPC++, some of these implementations will benefit from more complex kernel design and implementations. This can be seen in Figure 6, which is a flow chart on choosing a kernel design.

For testing, we generated random dense adjacency matrices, which represent a near fully connected graph. This allowed for near worst case performance in all algorithms implemented, allowing for a clear indication of speed up due to parallelism.

Dijkstra's Algorithm

The implementations of Dijkstra APSP are similar in nature, as both call the Dijkstra’s SSSP algorithm onto all nodes of the adjacency matrix. The sequential version of the algorithm calls the SSSP algorithm one after the other for each node in the matrix.

The parallel version of the algorithm is the same as this sequential algorithm, however, instead of calling the SSSP one after another, this algorithm is called in parallel, with each workgroup operating for one source node. This means that for optimal performance, the number of vertices in the matrix should be larger than the number of processors present in the system.

Floyd Warshall

The implementation can be broken down into three implementations. The sequential version of the algorithm, the naive parallelized implementation, and a block based implementation.

The sequential implementation is very straightforward. The implementation consists of a sequential FW function, which implements the three nested for loops which iterate over the entirety of the adjacency matrix. The innermost loop contains the conditional which checks whether the related edges of a node have a lower combined path-weight than what is currently set. If this is lower, then the edge is set to the combined path-weight.

The naive implementation of the algorithm makes use of a basic DPC++ kernel which is held inside a for loop. This kernel contains the same condition check as the sequential implementation. The for loop iterates over the entirety of the matrix, and then the various workgroups/threads apply the same condition check in parallel, essentially parallelising the two inner loops of the sequential implementation of the algorithm.

The block based implementation is based on other parallel implementations of the floyd-warshall algorithm, where the matrix is chunked into various sections. Other implementations make use of 2-D blocks which are then allocated to various threads. The memory access pattern of a block, or even a cell, is simply the entire block/cell row and column of the matrix, whilst other blocks are untouched. This allows for mapping of these various sections to the threads and thus allows for faster parallelism as instead of operating the outermost loop to iterate through the entire length of the matrix, only the length divided by the block size can be iterated through. However, implementation of these kernels can be difficult for beginners. As such, we opted to create a simple ND-Range kernel, which was based on previous implementations of parallel Floyd-Warshall algorithms. The implementation splits the various work items into blocks based on the number of nodes and what the intended block size should be for the implementation.

Seidel’s Algorithm

The implementation of Seidel’s algorithm is fairly straightforward. The sequential implementation of the algorithm contains multiple nested for loops operated sequentially for various sections of the algorithm. This leaves the time complexity of the implementation to be far higher than the original paper has calculated, as the calculations for Seidel takes into account fast matrix multiplication algorithms, which our implementation does not contain. The sequential implementation requires multiple matrices of the same size as the input matrix, as the algorithm requires recursion for possible operations and thus requires multiple matrices to be held in memory until a base case has been reached in the recursion process.

The parallel implementation of Seidel is similar to the sequential implementation in structure, but instead applies the matrix operations inside the accelerator device, parallelising every computation done. The same issues of recursion also exist in the parallel version, however, with the use of unified shared memory instead of buffers , memory is distributed between the two devices and can also be freed far more efficiently during runtime than the sequential version. This does, however, lead to more communication and memory operations than the sequential version of the algorithm. This design choice was made because there are several restrictions on kernel code inside DPC++. One major limitation is that recursion cannot be executed in the kernel code, mainly due to the fact that the devices that kernel code is meant to run on do not have all the functionality as a CPU host, such as not implementing the entire C++17 standard. Any code intended for the device must have the amount of memory and workgroups that should be operating at compile time, and any form of recursion breaks this condition.

Prim’s Algorithm

Prior to implementing Prim’s algorithm, we needed a way to quickly generate graphs with any amount of nodes. The method we created randomly generates a graph of any size represented as an adjacency matrix and stored on the heap as a one-dimensional representation of a two-dimensional matrix. Path-weights between nodes are randomly generated between one and the maximum distance, set as a macro, with zero representing no connection. The path-weight between a node and itself is automatically zero, along with roughly half of all paths. Randomly generating the graph worked fantastic; however, there was a major issue: the pseudo-random algorithm in the standard c library was compiler dependent meaning we wouldn’t generate the same graph across different platforms. To fix this, we instead used the platform-independent deterministic algorithm mt19937 with the constant zero as a seed.

Our implementation of Prim’s Algorithm begins by selecting a random node, which will always be the same for a given amount of nodes, due to our constant seed. This allows us to ensure there isn't any uncertainty in our results due to changing the starting location. We then store the starting node in a separate array representing the visited nodes, and continue the algorithm until all nodes have been visited. For every node currently in the MST, the algorithm searches the nodes connected to it and adds the node with the lowest path-weight, that isn’t already part of the MST, to the visited nodes array. When the algorithm is complete, the total minimum path is printed to the console, which we can use to ensure our algorithm is running correctly when we parallelise it.

Before we attempted to parallelise Prim’s Algorithm in DPC++, we made an attempt in OpenMP, as this was much simpler and more familiar to us. Originally, we only parallelised the innermost for-loop; however, as this only determines the minimum path for a single node, there was no significant speedup. We then realised that we could parallelise the middle loop, which determines the minimum path for all nodes currently in the MST, and doing so gave us a notable performance increase. From the output value, it was clear we were getting coherence issues due to a race condition; however, this was fixed (without degrading the performance) by creating local minimum variables for each thread and determining the global minimum in a critical section after the parallel for-loop.

The DPC++ implementation uses similar logic to the OpenMP version; however, as DPC++ uses an entirely different parallel programming model, the implementations are unalike. The same loop is parallelised using the parallel_for directive; however, in DPC++ it must be submitted to a queue for execution on a parallel device. To avoid the race condition of setting the minimum value, we use atomic variables instead of a critical section, as they can guarantee no overlapping memory accesses. It is also worth noting that some of the shared variables had to be allocated to memory in order to share them between devices, which doesn’t need to be done in OpenMP.

Results

Testing was done on various machines, however to keep our results consistent, we decided to use Intel DevCloud to benchmark our algorithms. DevCloud is a service which offers a range of powerful hardware to users for free, fully supporting DPC++ programming. The target platform we chose was a machine with 24 Intel Xeon processors.

Within the APSP algorithms, there were noticeable speedups moving from a sequential implementation to a parallel implementation (Figures 2 – 4). The execution times of parallel Floyd-Warshall and Djikstra both increased at a much slower rate as the number of nodes increased. Parallel Djikstra was about 40 times faster than the sequential implementation at 1515 nodes. The execution time of Seidel is ambiguous as we were not able to obtain more data for a greater number of nodes due to segmentation faults created from a lack of memory. Since Seidel is a recursive algorithm, there are too many copies of graphs which cannot be freed in a sequential implementation, which was not supported by our testbench.

As shown in Figure 5, there are significant speedups in performance of Prim’s Algorithm for both OpenMP and DPC++ versions of the parallel implementation. As the number of nodes on the tree are increased, this is common across all variants of Prim’s Algorithm. It is interesting to note that the OpenMP variant of Prim’s performed better than the DPC++ version of Prim’s, especially as the number of nodes were increased. While this may seem to suggest that OpenMP may be the language of choice for this algorithm, with DPC++, more performance can be obtained by switching the device the kernel runs on from CPU to a more computational capable device such as a GPU.

Discussion

Performance

Overall, the results which we obtained were within our expectations. When the number of nodes within the graph was small, the sequential implementations of each algorithm performed far better than the parallel implementations. At 15 nodes per graph, the number of nodes within the graph was less than the number of processor cores allocated to us by the Intel DevCloud, showing an extreme case of the amount of overhead required to allocate work to each processor core outweighing the temporal performance gain of parallel processing. Many of the graphs were able to execute within a millisecond sequentially, displaying an output of 0 milliseconds.

As the number of nodes within the graph increased, we began to see parallel implementations outpace the sequential implementations by greater and greater margins, with the parallel implementation of APSP Djikstra's Shortest Path Algorithm reducing the computation time by more than an order of magnitude. However, we did also see the effects of Amdahl's Law as the speedup did not continue its initial exponential trend and gradually tapered off.

Segmentation Faults

Many of our implementations began to run into segmentation faults when the number of nodes began to increase past a certain threshold. This unfortunately was due to the implementation themselves, as many would use simple variable allocations for various matrices, along with multiple copies of these matrices. Algorithms such as Seidels require multiple copies of the various matrices that are created from the input matrix, and thus this is unavoidable. The use of other mechanisms for storing data would be the logical step to remove this issue, however, this would increase the time spent on I/O retrieving data from the disk or RAM, which is not ideal. Ideally, sections of data will be prefetched in the necessary blocks from disk, and then used by the DPC++ application.

To improve on the current implementations, the use of ND-Ranges/Hierarchical kernels would be used, alongside the use of granular control over the implicit memory migrations that are currently occurring in the implementations. The use of memory prefetching between the device and host by using the prefetch method, as well as mem_advise for allocation of memory would be incorporated into the implementation.

Conclusion

Overall, we were able to effectively implement various graph algorithms using DPC++. The results of which show a clear increase in performance when compared to sequential implementations. Unfortunately, the issues due to memory were not able to be overcome, and thus results from APSP testing leave some room for improvement. Given more time, we would have liked to improve our implementations with a greater understanding of DPC++.

References

[1] D. Coppersmith and S. Winograd, "Matrix multiplication via arithmetic progressions," in *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, 1987, pp. 1-6.

[2] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik,* vol. 1, no. 1, pp. 269-271, 1959.

[3] R. W. Floyd, "Algorithm 97: shortest path," *Communications of the ACM,* vol. 5, no. 6, p. 345, 1962.

[4] R. C. Prim, "Shortest connection networks and some generalizations," *The Bell System Technical Journal,* vol. 36, no. 6, pp. 1389-1401, 1957.

[5] J. Reinders, B. Ashbaugh, J. Brodman, M. Kinsner, J. Pennycook, and X. Tian, *Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL*. Springer Nature, 2021.

[6] R. Seidel, "On the all-pairs-shortest-path problem in unweighted undirected graphs," *Journal of computer and system sciences,* vol. 51, no. 3, pp. 400-403, 1995.

Appendix

Table 1: Table of Contributions

Name	**Contribution**
Lenny Schwarz	- Breadth-First Search - Research and Presentation - Prim’s Algorithm - Research and Presentation - Sequential Implementation - OpenMP Parallel Implementation - DPC++ Parallel Implementation - Benchmarking with Intel DevCloud
Rodger Chen	- Initial Research - DPC++ - Research and Presentation - Prim’s Algorithm - DPC++ Parallel Implementation Research - Data Collection
Nikhil Kumar	- Initial Research - DPC++ - Research and Presentation - Floyd-Warshall - DPC++ Parallel Implementation research - Sequential Implementation - Naive Implementation - Blocked Implementation - Seidels - DPC++ Parallel Implementation research - Sequential Implementation - Naive Implementation - Makefile - Benchmarking - Benchmarking on Local Machine - Scripting for Testing - Bash - Devcloud Testing - Data Collection
Bill Yang	- Breadth-First Search - Research - Binary Search - Research and Presentation - Floyd-Warshall - Research - Djikstra’s Shortest Path - Research - DPC++ Sequential and Parallel Implementation - Benchmarking with Intel DevCloud

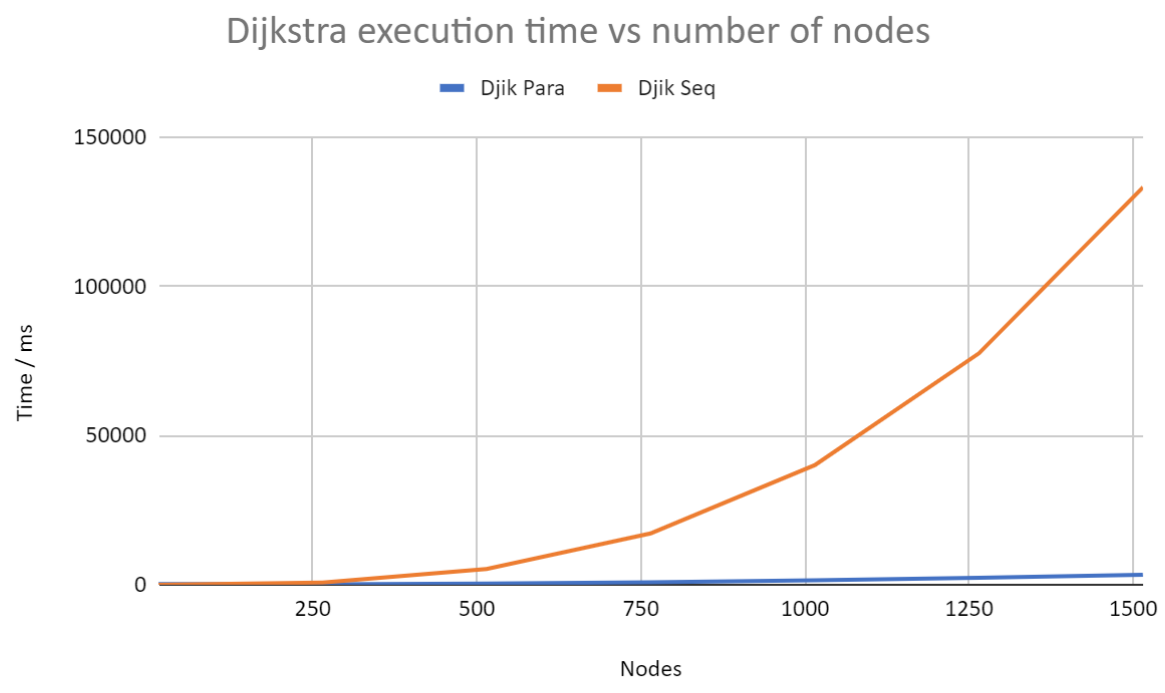


Figure 2: Dijkstra's Algorithm Execution Speed

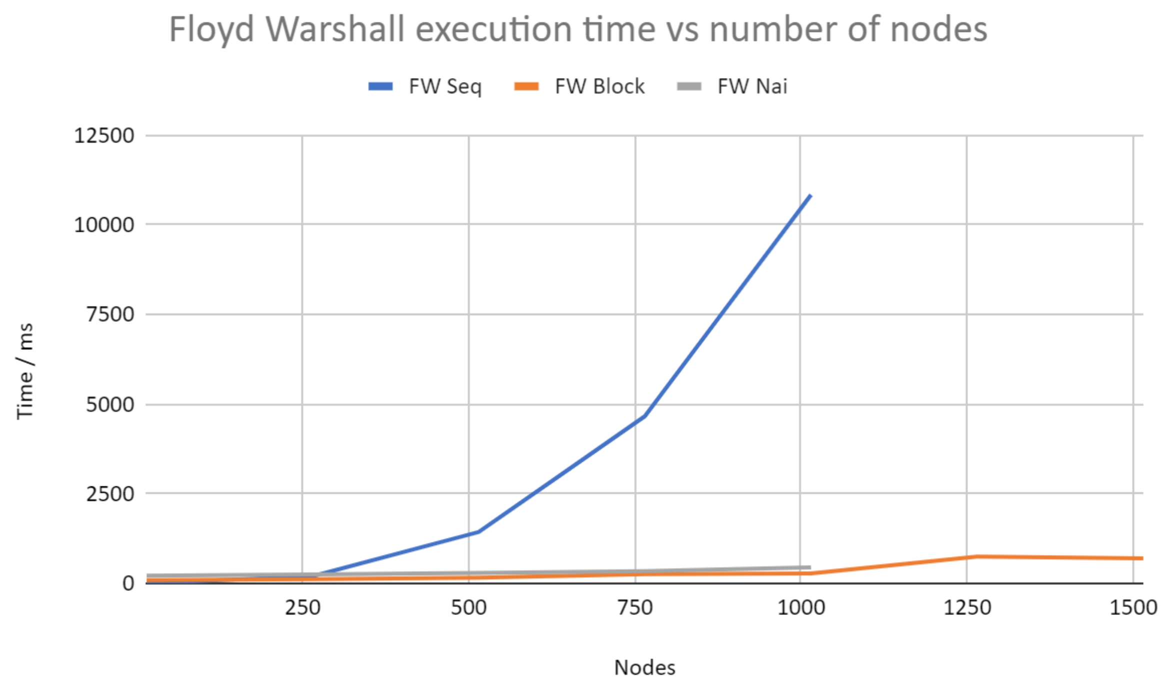


Figure 3: Floyd Warshall Execution Speed

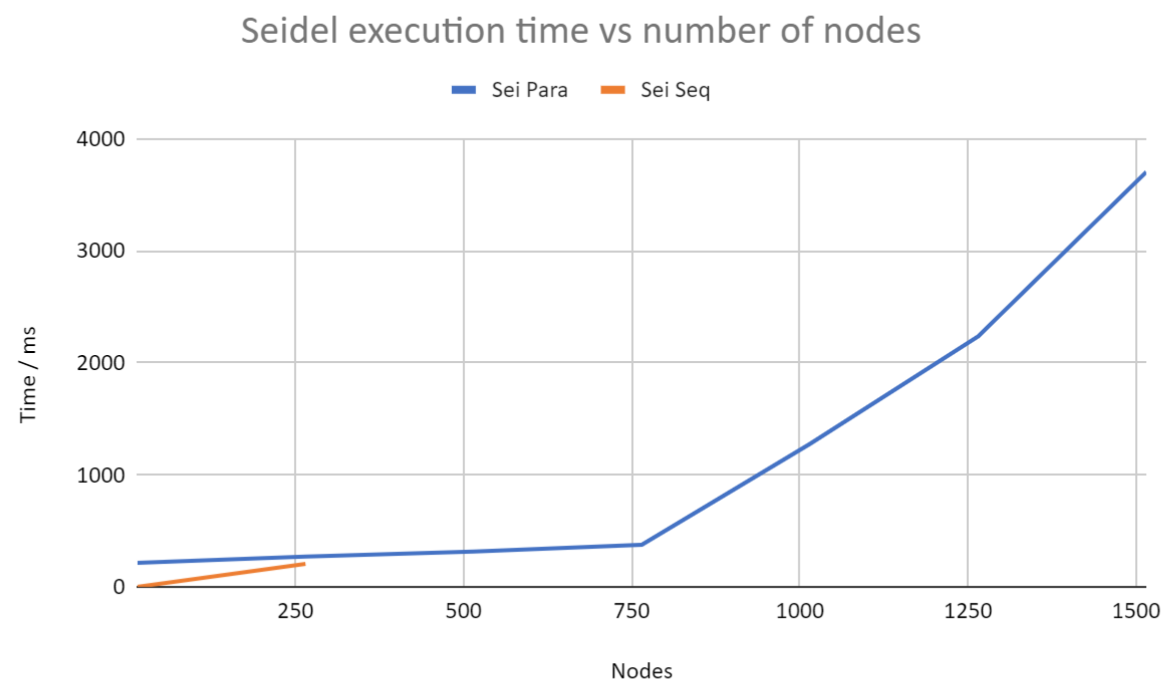


Figure 4: Seidel's Algorithm Execution Speed

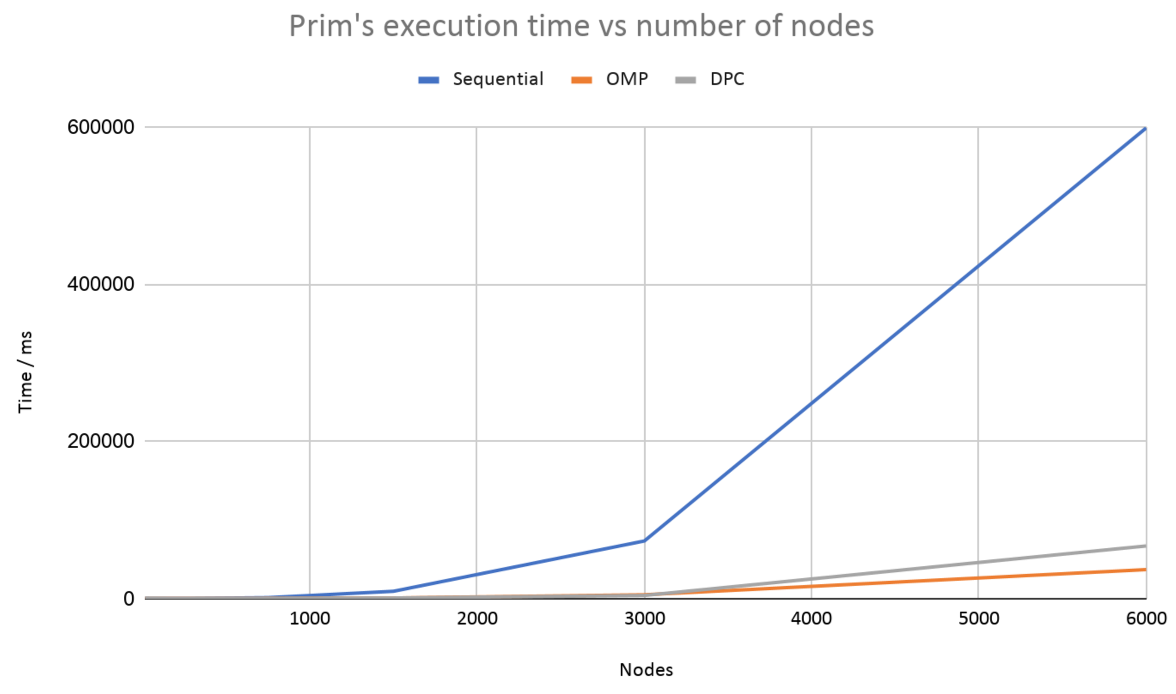


Figure 5: Prim's Algorithm Execution Speed

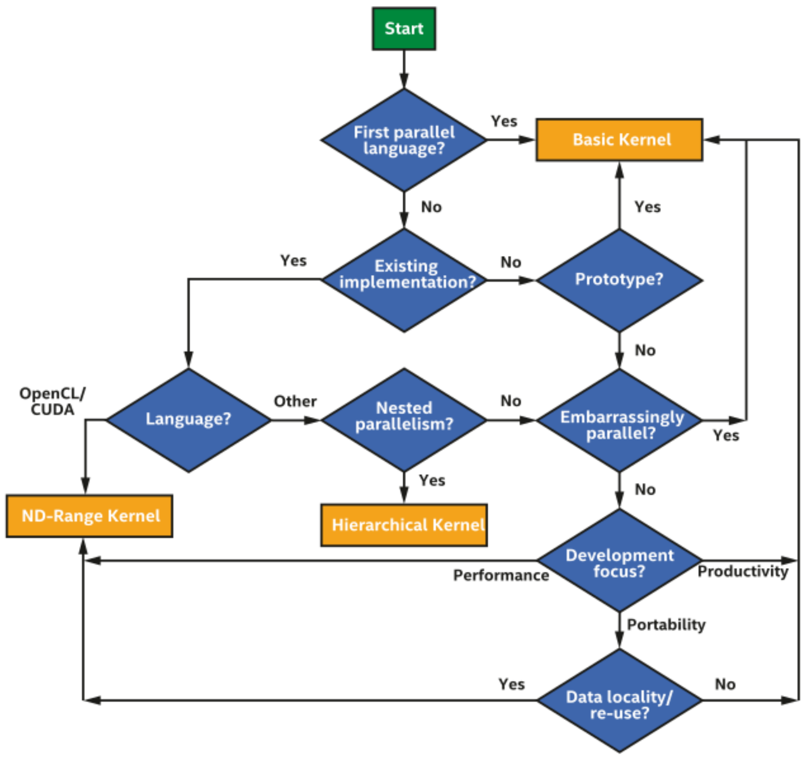


Figure 6: Flow Chart Provided by Intel on Choosing a Kernel for Implementations