View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Illinois Digital Environment for Access to Learning and Scholarship Repository

© 2018 Dong Hun Lee

A NODE-BASED APPROACH TO CHARM-FFT

BY

DONG HUN LEE

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2018

Urbana, Illinois

Adviser:

Professor Laximikant V. Kalé

Abstract

Parallel 3D Fast Fourier Transform is a communication intensive algorithm that suffers from the unignorable communication overhead. Because the interconnect communication bandwidth is a static component, adjustments to reduce or hide the necessary communication overheads are performed to obtain the optimal performance with a FFT grid in a given environment. In this thesis, an alternative method to an existing Parallel 3D FFT library was explored. The FFT library, Charm-FFT empowered by Charm++, was redesigned to utilize larger number of nodes while aiming to reduce the number of necessary communications between its components during its computations. Instead of decomposing the input FFT grid into the fine-grained objects that are distributed to the available PEs, coarser-grained decomposition method that only distributes to the available nodes was applied. As there are less number of receivers that each decomposed object communicates during the state transposition, the overall number of communication is reduced at the cost of parallelism from using the finer decomposition method. This loss of parallelism is attempted to be mitigated by applying within-node parallelism using multi-threading or accelerators. Lastly, to maintain the usability of the modified library when multiple FFT grid computations are needed with given resource, each FFT grid is assigned to a subset of the resource to compute and communicate only within its subset rather than to use all resource for each grid's computation.

Acknowledgments

I would like to thank my advisor, Professor Laximikant V. Kalé, for the opportunity and guidance in pursuing this thesis. I would also like to express my appreciation to the group members at the Parallel Programming Lab for their unconditional help, especially Raghavendra Kanakagiri who continuously provided constructive feedbacks during our countless discussions throughout my research. Lastly, being a part of PPL during my time at UIUC has truly been an invaluable experience.

Table of Contents

Chapter 1 Introduction 1

Chapter 2 Charm-FFT 4
2.1 Charm-FFT Implementation 4
2.2 Charm-FFT Performance 6

Chapter 3 Charm-NodeFFT 8
3.1 Overview 8
3.2 Modification Detail 9
3.3 Performance Analysis of Charm-NodeFFT 11

Chapter 4 Accelerator Usage for FFT Operations 21
4.1 Modifications to *Charm-NodeFFT* 22
4.2 Performance 22

Chapter 5 Multi-Instance FFT 27
5.1 Modifications to *Charm-NodeFFT* 27
5.2 Performance 28

Chapter 6 Conclusion 32

References 34

Chapter 1: Introduction

Fast Fourier Transform (FFT) is a widely used algorithm with its usage in numerous areas in scientific computing, ranging from digital signal processing to molecular dynamics and simulations. It is designed to compute discrete Fourier transform in relatively lower complexity, and numerous libraries have expanded the methods to implement FFT operations to compute multi-dimensional transforms, some in parallel. In a typical parallel three-dimensional FFT (3D-FFT) algorithm, two types of domain decompositions exist for the parallelization method on distributed-memory systems. First is a 1D domain decomposition, which is often called *slab decomposition*. In this method, the application domain is divided into 2D slabs that are distributed to the available processors. Each processor performs a sequential 2D-FFT operation on its slab. Then, a global transpose is taken place after all processors complete their FFT steps. After the transpose, sequential 1D FFTs are performed on the last dimension to complete the parallel 3D FFT [1]. Another partitioning method is to divide the application domain into 2D decomposition, which is commonly known as *pencil decomposition*. For the 2D decomposition method that computes a single 3D FFT grid, two global transpositions are needed between the three states, Z, Y, and X states. Each pencil objects per state consists of a buffer that represents multiple rows of the FFT domain with points along the dimension that it is set to perform 1D sequential FFT operation on [1].

Due to the efficiency and organization it provides to complete discrete Fourier transform, FFTs have been integrated and optimized to many different libraries, one of which is *Charm-FFT*. *Charm-FFT*[2] is a *Charm++* library that computes parallel three dimensional Faster Fourier Transform. *Charm++* is an object-based message-driven parallel programming system that decomposes an application's domain into asynchronously executing units called *chares* [3]. In *Charm++*, the user may break down a problem into a desired decomposition granularity by creating any number of *chares* that can be mapped to available processing elements (PEs). The created chares are then mapped by Charm++ Runtime System (RTS) using the user-defined mapping or block mapping to available PEs. *Charm++ chares* may communicate with another *chare* by invoking its non-blocking *entry method*. This will send a message to the receiver *chare*, scheduled by *Charm++* RTS, and can be used to send a buffer to a *chare* to be copied.

Charm-FFT [2] is a library that computes parallel three dimensional Fast Fourier Transform. It was developed by Nikhil Jain specifically for *OpenAtom* project, an application for parallel Ab-initio molecular dynamics simulations,[4]. *Charm-FFT* implements 2D-decomposition method to effectively distribute the pencils to available PEs. As stated before,

this method allows computing a single 3D-FFT grid in parallel with higher scalability.

In addition, to benefit from *Charm++'s* fully asynchronous nature, each pencil object is assigned to a *Charm++ chare*. Multiple *chares* can then be mapped to unoccupied PE. When a PE completes a pencil *chare's* FFT operation, it will send FFT'ed data to the designated receiver *chares*. While waiting for the incoming messages, a PE can compute another FFT operation for a chare that it may be holding, potentially compensating for the idle time that is produced until all messages arrive. However, this method of using 2D-decomposition with a single given FFT-grid for computation does not scale indefinitely because of the natural limitation in how small the decomposing granularity can be in a given resources and the inevitable communication overhead that exists during each transposition step. *Charm-FFT* tries to overcome this by overlapping other computational work such as multiple *Charm-FFT* instances or other computation in the same *Charm++* application. This is possible because *Charm++* allows processing messages for any *chare* in a PE to minimize the idle time, reinforced by having smaller decomposition sizes of pencil *chares* for shorter execution units.

If larger number of processors are desired to be utilized for a fixed sized FFT grid, *Charm-FFT* needs more instances to effectively use the idle time between *chares'* state transition. If the number of instances become fixed, then smaller granularity will be needed for each instance to distribute over more nodes. In such cases, each pencil *chare* will be distributed with smaller pencil size to compute, and while the FFT computation time of individual *chare* would decrease, the number of messages that it is required to send as part the transposition increases. Additionally, since the receiver *chares* would also be handling larger number of messages, the overall communication overhead for the transposition would increase. At a certain point, the communication overhead for the transposition steps will surpass the actual computation time for each pencil *chare*, unnecessarily occupying CPU time. In such case, larger granularity for pencil decompositions, at the cost of parallelism and asynchrony, may be preferred.

In this thesis, an alternative approach to *Charm-FFT's* implementation to pencil *chare* distribution was experimented in attempts to focus on using sets of available nodes over large number of available PEs. Chapter 2 describes the current *Charm-FFT* implementation and displays its characteristics that are used to handle the inclining communication and scaling problem as finer decomposition is desired. In following chapters, different approaches and modifications to *Charm-FFT* are presented. Chapter 3 modifies *Charm-FFT's* decomposition method for minimizing the number of communications between the pencil *chares*. Chapter 4 is about an additional experiment that follows Chapter 3 to seek further benefit from the modified decomposition structure. Lastly, Chapter 5 will describe the modified

Charm-FFT's method of using multiple library instances and its effectiveness in amending scalability limitation.

Chapter 2: Charm-FFT

Charm-FFT was developed in attempts to address the limited parallelism and the scaling bottleneck that parallel FFTs have. With *Charm++*, each state of FFT grid is decomposed into large number of *Charm++ chares*. These smaller objects that can be executed asynchronously allows to overlap multiple concurrent instances of FFT computations on separate FFT grids efficiently if desired[2].

As with many effective parallel FFT libraries, *Charm-FFT* also uses 2D decomposition method, which is also commonly known as "pencil decomposition", to partition each of Z, Y and X state's domain into 2D blocks, called 'pencil'. Each pencil holds rows of arrays of 1D FFT points along the dimension that it is partitioned for as shown in Figure 2.1. Each of these pencils that holds an array of 1D FFT points along its dimension are respectively called *X-Pencil*, *Y-Pencil* and *Z-Pencil* for convenience.

A typical parallel FFT that uses 2D decomposition method properly divides two dimensions of each states to form blocks of 1D pencils. But while still implementing the 2D decomposition method, *Charm-FFT's* design uses a 1D decomposition on one of the three states, X-state. This causes the X-states' 'pencils' to be formed into slab-like blocks but are considered as thicker pencils. As a result, decomposition on the X-dimension is only done along the Z-plane, effectively creating a 1D decomposition just for that state. This design was influenced by *OpenAtom* project that normally needed the FFT'ed output complex values to be in planes, bypassing an extra step to aggregate the output values.

The pencils of each state are represented as *Charm++ chares*. Each *chare* computes 1D line FFT on the buffer that it holds, independently from other pencil *chares*. After the completion of the line FFT on its buffer, the *chares* asynchronously perform the transposition steps by packing its FFT'ed buffer into messages and sending each of these messages to the corresponding *chares* that are responsible for the next phase of FFT.

2.1 CHARM-FFT IMPLEMENTATION

A *Charm-FFT* library instance is created by calling *Charm_createFFT* function in *Charm-FFT* interface from a processor. For the inputs, the size of the FFT grid to compute, the pencil decomposition of each dimension and a callback function that would be invoked when the initial internal setups are complete are expected. Based on the inputs, the newly created *Charm-FFT* library instance constructs several *Charm++ chares* that represents the dimensional pencils, *Z-Pencil*, *Y-Pencil* and *X-Pencil*. The pencil *chares* are then assigned

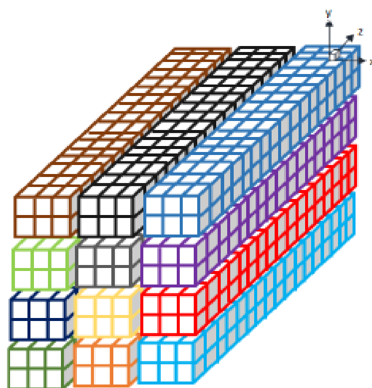


Figure 2.1: Z-state 2D decomposed into 1D pencils

to a portion on which it is expected to compute a line FFT on. Once complete, the *chares* are mapped to the available PEs by default block distribution or by the mapping scheme that the user provides as an optional argument to *Charm_createFFT* call.

During the pencil *chares'* construction, each pencil *chare* will be set to have the references to the receiver *chares*. These references are kept as the linear indices to the collections of each state's pencil *chares* and are used later during the state transition to send the segments of the FFT'ed buffer after computation. Every *chare* has expected number of incoming messages to receive before it is ready to perform the line FFT on its copied buffer. When all pencil *chares* are initialized, the library instance creation is complete. The user-input callback is then used to inform the user application that the pencil *chares* may now be accessed from each PE using *Charm++ group chares*.

From each PE, *Charm-FFT* instance can be queried to acquire the linear offsets of Z, Y or X dimensional pencil objects that are mapped to the current PE. Using these offsets as identifiers, the user may assign the allocated input and output buffers for the FFT operations on that portion of the FFT-grid. After these assignments, *Charm-FFT* instance is ready to start the FFT operation. This can be requested to start by calling *Charm_doForwardFFT* or *Charm_doBackwardFFT* from each PE. As the message to start the operation has been broadcast from the PE 0 to all pencil *chares* of the starting phase, each *chare* performs a series of line FFTs on its buffer independently. For the FFT operation, the line FFT would begin with *Z-pencils* and *X-pencils* for the inverse FFT.

When a *chare* completes its FFT operation, messages that contain the FFT'ed data are sent to the corresponding pencil *chares* of the next state. When a receiver *chare* is scheduled to process an incoming message by *Charm++ RTS*, it unpacks the message and copies the

Total Chare Count	Chares Per Dim	Decomposition	FFT (ms)	IFFT (ms)
75	25	5x5	157	127
192	64	8x8	92	78
300	100	10x10	50	41
675	225	15x15	89	45
1200	400	20x20	96	70
1875	625	25x25	160	90
2700	900	30x30	316	153

Table 2.1: Effect of increasing decomposition on performance

contained segment of FFT'ed data into its own buffer. To ensure that the segments do not overlap, the source *chare's* indices are used to determine each segment's location within the buffer memory. Each message delivered to a pencil *chare* is queued and processed sequentially, until all expected messages are received. Then, if the receiver *chare* has no more expected incoming messages, it may start the FFT operation regardless of the other pencil *chares'* transposition status.

As all output pencil *chares* complete their FFT on its buffer, a reduction is performed to ensure that all outputs are ready to be presented to the user. When the reduction completes, the user-input callback is used to inform the user that the output values are now available. The user, then, may access the FFT'ed data from each PE with the output memory buffer that was allocated and provided prior to the start of the operation.

2.2 CHARM-FFT PERFORMANCE

2.2.1 Effect of Decomposition Size on Communication

To show the effects of decomposition size on performance, a simple test with single *Charm-FFT* instance that performs series of FFT and IFFT operations is run using 300 x 300 x 300 FFT grid on 4 nodes of Knights Landing. Total of 128 cores were used with 32 core usage on each node.

Table 2.1 presents the separate timings of FFT and IFFT operations as finer decomposition is used for each dimension of the input FFT grid. As it can be seen from Table 2.1, there is a gradual reduction in operation time as finer decomposition is used until 10 x 10 decomposition, which allocates total of 300 *chares* for the pencil decomposition, 100 *chares* per state. This decomposition distributes about one pencil *chare* per state per PE. But after this point, further decrease in granularity continuously drops the overall performance. Based

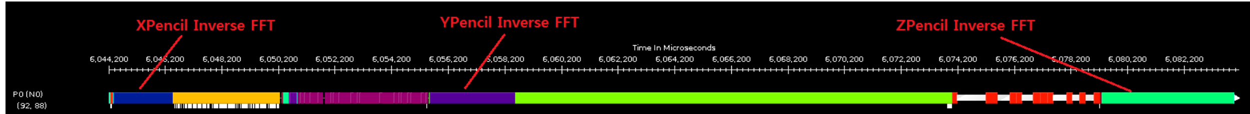


Figure 2.2: Timeline of pencil objects' activities in 10x10 decomposition

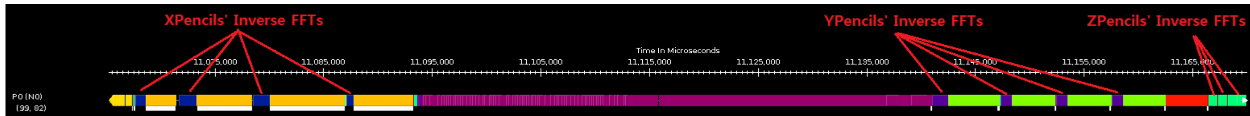


Figure 2.3: Timeline of pencil objects' activities in 20x20 decomposition

on this, it could be seen that the communication overhead from excessive grid decomposition eventually overshadows the benefits of utilizing multiple short asynchronous objects.

Figure 2.2 and 2.3 are visualized timelines of traceable *Charm++* events in PEs throughout the application. Each color-coded segment is a traced entry method execution by a pencil *chare*. These timelines have been traced and generated using *Charm++ Projections* tool[5]. Figure 2.2 is the timeline of one of the PEs performing inverse FFT operation on 300 x 300 x 300 FFT grid using the optimal 10x10 decomposition from 2.1.

In Figure 2.2, the labeled blocks are the entry methods running inverse FFT for *X-pencil chare*, *Y-pencil chare* and *Z-pencil chare* respectively. The other visible blocks are the communication steps that consist of forming IFFT'ed points into messages, sending each messages to their corresponding receivers and receiving the message to apply the points to the buffers. The white blocks before the *ZPencil Inverse FFT* are the idle time when there are no executing entry methods in a PE. As it can be observed, the amount of time that are used for communication steps surpasses the computation time of each pencil *chare*. From the total IFFT execution time for this PE, only about 25% of it was spent for the inverse FFT operations on the current PE's pencil *chares*.

Figure 2.3 presents a timeline for 20x20 decomposition, which creates 400 pencil *chares* per state, distributing about 4 *chares* per state to each PE. According to Table 2.1, 20x20 decomposition produced 71% diminish in performance compared to the optimal decomposition of 10x10. The time that was spent in inverse FFT operations with this decomposition was about 21%. While it is visible that finer-decomposition allows reducing the idle time from waiting for other pencil *chares'* messages, it can be seen that as excessive decomposition is used, that maps multiple *chares* per state in each PE, the communication overhead inflates that ultimately impacts the overall performance negatively.

Chapter 3: Charm-NodeFFT

Charm-FFT emphasizes the decomposition of FFT grid's states into fine-grained decomposition pencil objects to increase the parallelism of FFT computations by distributing each of these objects to available PEs. In addition, as these objects are formed as *Charm++ chares*, their computations can be performed asynchronously, allowing overlapping the computations of multiple FFT-grids with multiple library instance.

Normally, when increase in the number of instances or higher parallelism is desired, a larger number of *chares* must be constructed to optimally utilize available cores. Using finer granularity for the decomposition not only implies that each *chare* will handle smaller pencil size but also suggests that the number of messages generated for the communication between each *chare* during the transposition steps increase. As the number of *chare* increase, there are greater number of *chares* to coordinate with.

However, the decomposition size of each state is limited to the size of a minimum line FFT that can be performed in a PE. The communication overhead, which consists of message creation, sending, scheduling, unpacking and applying, is only partially dependent on the size of the decomposition and is mostly constant. In this case, to utilize larger number of nodes, *Charm-FFT* would need to minimize the distribution of pencil *chares* of an instance across the nodes so that the inter-node communication cost would not become the major bottleneck. If, for instance, the network bandwidth between the nodes is sufficiently low or if too many pencil *chares* are mapped to a single node while multiple *Charm-FFT* instance is concurrently running, excessive partitioning would lead to costly communication during each *chare's* transposition step, eventually causing the communication time to surpass the computation time.

Thus, modifications focusing on reducing the number of necessary inter-node communications have been made to *Charm-FFT* as an alternate approach to implement the communication intensive 3D FFT algorithm.

3.1 OVERVIEW

To reduce the communication overhead that *Charm-FFT* induces during each *chare's* transposition step, modifications that adjusts the pencil decomposition and distribution was attempted. Instead of partitioning the FFT grids into large number of pencil objects to compute per state, the modified *Charm-FFT* (*Charm-NodeFFT*) decomposes the FFT grids into less number of objects. Each pencil object in *Charm-NodeFFT* then holds a larger

pencil size than the pencil *chares* in *Charm-FFT*. The pencil *chares* in *Charm-NodeFFT* was also set to distribute to the available nodes rather than to available PE's, as it was in *Charm-FFT*.

This conversion is primarily aiming to reduce the number of required communications between the pencil *chares*. Yet there is a clear cost in parallelism with less decomposition because it undermines the benefits of over-decomposition. Thus, the loss of parallelism was compensated by multithreading each pencil *chare's* FFT computation using the local PEs available to a node. Multithreading each *chare's* FFT computation was done with *Charm++ CkLoop* library[6]. *CkLoop* library supports loop-level parallelism by allowing multithreading a task of a *chare* with PEs that are already available in Charm++ runtime.

3.2 MODIFICATION DETAIL

Similar to *Charm-FFT*, to create an instance of *Charm-NodeFFT* library, the user must call *Charm_createFFT* from a processor with expected inputs. The dimensions of the FFT grid to compute and a callback function to invoke when initialization is complete are required inputs and remain unchanged. However, instead of determining each state's pencil decomposition according to the user's needs, the library expects the total number of pencil *chares* for each state to equal the number of available nodes when the application runs. This design implies that only one pencil *chare* per state should exist per node. This also results in each pencil *chare* to have relatively larger pencil buffer compared to the pencil *chares* in *Charm-FFT*.

Internally, once the request to create a *Charm-NodeFFT* library is received, the library creates *Charm++ nodegroup chares*. In contrast to *Charm-FFT's* use of *Charm++ group chares*, which provide convenient access to the pencil *chares* in a local PE, *nodegroup chares* are used to interact with the pencil *chares* in *Charm-NodeFFT* nodes. When message is sent to a *nodegroup chare* by invoking its entry method, the message is processed by any of the available PE within the node. Because of this, incoming messages may be processed concurrently in separate PE's, avoiding the bottleneck that could occur if the arriving messages were processed sequentially by one PE.

Pencil *chares* are created based on the user input and are distributed to nodes so that each *nodegroup chare* has access to one of each X, Y, and Z pencil *chare*. As with *Charm-FFT*, each pencil will have references to the destination *chares* for the transposition and have expected number of messages before it is ready to start its FFT computation. Once all pencil *chares* have been constructed according to the user-input arguments and registered to the respective *nodegroup chares*, the library initialization is assumed complete. The user

is notified by the previously provided callback function.

After receiving the messages to the callback functions, the user may access the pencil *chares* from each *node*. Unlike the original *Charm-FFT*, where the access was achieved through each PE, the user access to pencil *chare* is achieved from each node to allocate and assign the input and output buffers to the pencil *chare* that is in the local node. Then, the request to start the FFT or inverse FFT operations can be made with *Charm_doForwardFFT* or *Charm_doBackwardFFT* from each *node*.

When the operation is commenced, the root *nodegroup chare* of *Charm-NodeFFT* instance will broadcast the start message to all pencil *chares*. This broadcast will initiate the operation on the pencil *chares* that are in the starting state of the requested FFT type (X-pencil if Inverse FFT or Z-pencil if FFT). As it did with *Charm-FFT*, the pencil *chares* in *Charm-NodeFFT* are to perform series of line FFT's in the buffer along the dimension that it is responsible for; however, because each pencil *chare* would typically hold distinctly thicker pencil compared to that of *chare* in *Charm-FFT*, sequential computation of each line of its pencil would cause significant loss of performance.

To mitigate this, the FFT operations within each *chare* have been multi-threaded using *Charm++ CkLoop* library. As each pencil *chare* begins its FFT operation, the buffer will be evenly assigned to the PEs so that each available PE in a node will be assigned to approximately similar sized segments of the pencil to compute on. Because *CkLoop* library works for shared-memory multi-threading environment and all PEs in a node will share memory because of SMP mode, each PE does not need to introduce additional space and time overhead need to copy its portion of the pencil segement; instead, each will work on the non-overlapping memory space, then perform series of 1D line FFTs over the assigned segments of the pencil buffer. After PEs in a node complete their FFT operation on their respective segments, the *nodegroup chare* prepares to create messages from its buffer to be sent to the corresponding *nodegroups* as part of the transposition step.

The original *Charm-FFT's* process of transposition consists of a sequential message packing for each sending messages, invoking the destination *chares* with the packed message and unpacking to apply the contained FFT'ed points to the local buffer when received. The receiver pencil *chare* in a PE handles the incoming messages one at a time when *Charm++ RTS* schedules it. Since the sending messages contain points that never overlap with any other points in other message of this pencil *chare*, the message creation do not need to be sequential. In *Charm-NodeFFT*, this step has been multithreaded using the functionality of *CkLoop* that allows the available PEs in the node to concurrently pack and send the messages to their corresponding receivers. Additionally, in contrast to *Charm-FFT*, the messages are not sent directly to the receiver *chares*. Instead, they are sent to the *nodegroup chares* that

the receiver *chares* are respectively placed in. This modification is derived from the feature that *nodegroup chare* provides. If the messages are directly sent to the pencil *chare*, the PE that has the pencil *chare* will process it. However, because *Charm-NodeFFT* suffers from lack of decomposition by the intended conversion, the larger pencil size leads to larger message size per communication. If one PE processed all messages sequentially, this would become the major bottleneck of the transposition step. To avoid this, the FFT'ed data are sent to the *nodegroups* by invoking their entry methods. *Nodegroup chares* receive messages and one of the available PEs in the node handles the message. As previously stated, the PEs do not mistakenly overwrite each other's segments regardless of the order of the entry method invocations, because the messages containing FFT'ed data do not overlap.

Afterward, similar to *Charm-FFT*, the *nodegroup chare* will be on hold until all expected messages are received and copied properly into its local buffer by its PEs. Then, the next phase's FFT operations will be ready to start. After all three states' FFTs across all *nodegroups* have been performed and verified by the reduction, the user-input callback function will be invoked. The user can then access the FFT'ed data from the output memory from each node.

3.3 PERFORMANCE ANALYSIS OF CHARM-NODEFFT

3.3.1 Testing for the Correctness

A simple test case has been used throughout the development of *Charm-NodeFFT* to constantly verify the efficacy of the modification and the consistency of outcome of the FFT procedure. To ensure the correctness, the initial inputs of each pencil have been saved. After performing a series of transformation, the resulting output are normalized using the product of the input dimensions and compared to the initial inputs. The errors from each pencil *chare* are aggregated with reduction to check if any reduced error occurred on each iteration of transformation.

3.3.2 Multithreading Optimization

Omitted optimization

Figure 3.1 shows the basic comparison of FFT and IFFT operations of *Charm-NodeFFT* on two nodes of Knights Landing using 300 x 300 x 300 FFT grid. As shown in the figure, inverse FFT operations show clearer gain in performance from increase in available worker

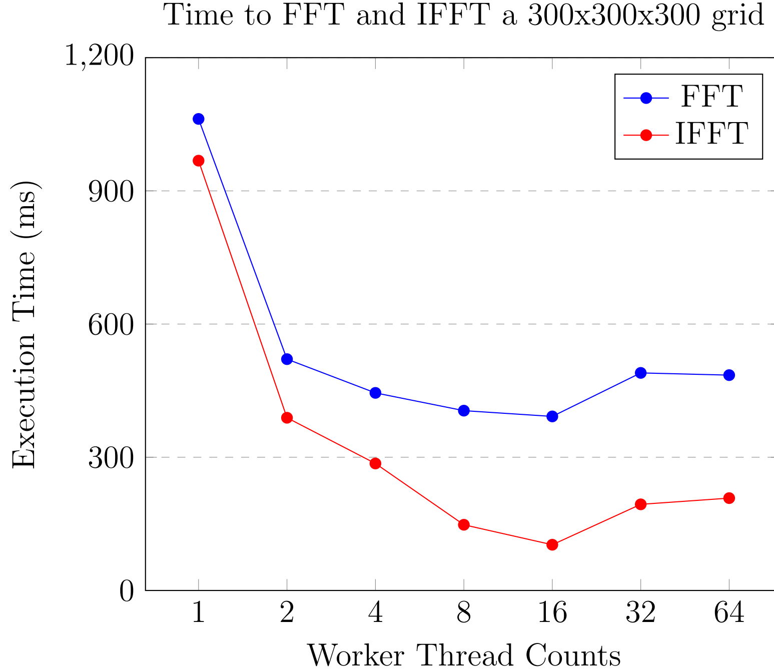


Figure 3.1: Comparison of FFT and IFFT using CkLoop in 2 nodes

threads count compared to the FFT operation. This is caused by the omitted optimization for the FFT procedure.

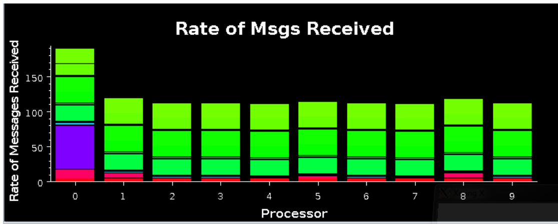
The multithreading optimization that uses *CkLoop* has been implemented in following parts of *Charm-NodeFFT*:

1. FFT operations of each node's pencil *chares*
2. Message creation and copying the FFT'ed data into each message to send
3. Message sending to each of their receiver pencil *chares*

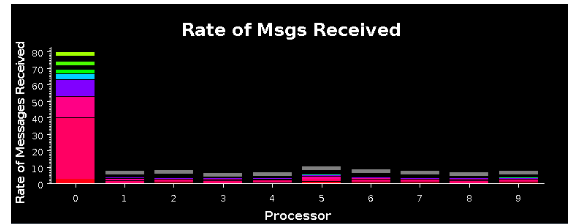
This optimization was to be implemented for all transformation stages, *Z-pencil* to *Y-pencil* to *X-pencil* and its inverse. However, *CkLoop* integration to *Z-pencil chares* post-FFT message management was not completed. Currently, this part of copying and sending to *Y-pencil chares* is only being processed sequentially by one PE using the initial Charm-FFTs code, which creates a significant bottleneck. Such lack of *CkLoop* integration hinders the ability to properly observe the effect of the intended conversions in *Charm-NodeFFT* when Z to Y transformation is involved. Thus, the generated performance results from the current *Charm-NodeFFT* are mainly from the inverse FFT results. This allows for the performance results to avoid being affected by the unoptimized bottleneck conversion from *Z-pencils* to *Y-pencils*.

Points per node	Worker threads	IFFT (ms)
150x150x300	4	286
150x150x300	8	148
150x150x300	16	122
150x150x300	32	194
150x75x300	4	212
150x75x300	8	194
150x75x300	16	109
150x75x300	32	262
75x75x300	4	208
75x75x300	8	164
75x75x300	16	67
75x75x300	32	174

Table 3.1: Inverse FFT timings based on the number of worker threads



(a) Messages received with Charm-FFT



(b) Messages received with Charm-NodeFFT

Figure 3.2: Rate of messages received by PEs during the FFT and IFFT procedure

Optimal points per worker thread

In Table 3.1, inverse FFT performance on 300 x 300 x 300 FFT grid is measured to find the optimal number of threads usage based on the number of points that each thread is given to compute. Ironically, the optimal number of threads in each node for computation is always measured at 16 threads regardless of the available node counts or distribution size. Post 16 threads, it can be observed that the performance on each node starts to drop. This trend implies that up to 16 threads in a node, there is benefit from using multithreading to parallelize the computations for the pencil *chare*. Afterward, the performance always immediate drops, indicating that the overhead from multithreading and synchronization costs starts to outweigh the computation time. Thus, 16 threads per node is used for the performance measurement.

3.3.3 Number of Communications

The number of messages that are received per second by each PE in a node were traced and visualized using *Projections*[5]. Figure 3.2 presents the messages received by a subset of a node's PEs. Figure 3.2a is generated from running 300x300x300 FFT grid using 10x10 decomposition on 128 PEs, using *Charm-FFT*. Figure 3.2b is also generated from 128 PEs, using 8 nodes total with *Charm-NodeFFT*.

From figure 3.2a, most PEs, excluding the PE 0, evenly distribute a number of messages that averages just over 100 messages received. PE 0, which handles reductions and broadcasts, naturally receive significantly larger number of messages. Figure 3.2b, on the other hand, shows significantly reduced number of overall received messages. Incoming messages for *Charm-NodeFFT* are focused to PE 0, and the PEs within a node do not need to send messages amongst themselves. The messages received by PE 0 throughout the application is largely for synchronization, and the number of messages for the transposition is below 40, which is sigficantly less than messages received by *Charm-FFT's* root PE alone. Furthermore, non-root PE in a node only receives messages during the initialization steps. As part of the design for *Charm-NodeFFT*, the messages are only sent to the PE that has *Charm++ nodegroup chare*, which then can be executed by any available PE in a node without blocking.

While the number of communication for FFT and IFFT decreased with *Charm-NodeFFT* conversion, these results strongly suggest that *Charm-NodeFFT* is greatly dependent on the inter-node network bandwidth since each message sent to the root PE of the node is significantly larger.

3.3.4 Comparison with Charm-FFT

Figure 3.3 presents the basic comparison of the inverse FFT operation between a single instance of *Charm-FFT* and *Charm-NodeFFT* using their respective optimal decomposition per given node count. Again, 300 x 300 x 300 FFT grid on Knights Landing nodes was used to measure the timings, and only inverse FFT was considered for effective performance comparison. For *Charm-FFT*, optimal time was generally produced when as many number of PEs per node was used with pencil decomposition that closely maps one pencil *chare* per PE. On the other hand, for *Charm-NodeFFT*, using wider nodes for larger pencil decomposition benefited its performance the most while only using 16 worker threads per nodes for its within-node parallelism.

As it can be observed from Figure 3.3, *Charm-FFT* shows the steady increase in performance as PE counts increase, while *Charm-NodeFFT* shows sharp reduction in execution

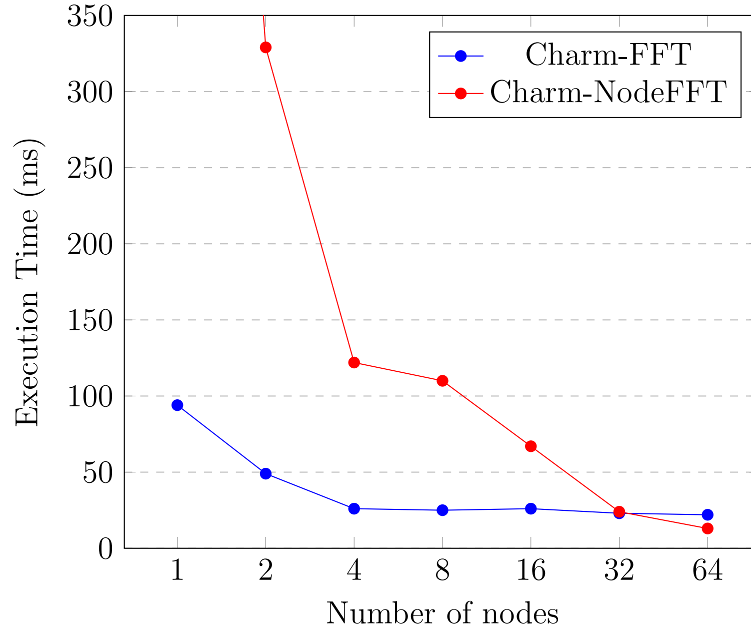


Figure 3.3: Single instance comparison based on the number of nodes, using respective optimal settings

time from 1 to 4 nodes. This is because of the lack of decomposition that *Charm-NodeFFT* has when smaller number of nodes are used, significantly hindering the performance because of the sequential operations on large number of points. As more nodes are used, smaller sized pencils are distributed across the nodes, significantly reducing the wait time of the PEs during the transposition steps. At 4 nodes usage, *Charm-FFT* has 61% speedup over *Charm-NodeFFT*. Afterwards *Charm-NodeFFT* starts to benefit from finer decomposition across the nodes while *Charm-FFT's* execution time plateaus at about 20 ms for the inverse FFT step.

However, to utilize *Charm-NodeFFT* as effectively as *Charm-FFT*, a wider node was needed. For instance, the *Charm-NodeFFT's* measurement of 64 nodes was run using 16 PEs on each node, using total of 1024 PEs. Meanwhile, with *Charm-FFT*, 1024 PEs were run on only 16 nodes, using 64 PEs each. When the same number of PEs were used, the number of nodes used for *Charm-FFT* was lower. And *Charm-NodeFFT*, when given a number of nodes, used very small portion of cores within each node.

Figure 3.4 and Figure 3.5 presents the performance comparison in two separate settings with fixed number of PEs per node. As previously stated, *Charm-FFT* generally benefits from utilizing as many PEs as available per node and decompose each grid to map each pencil

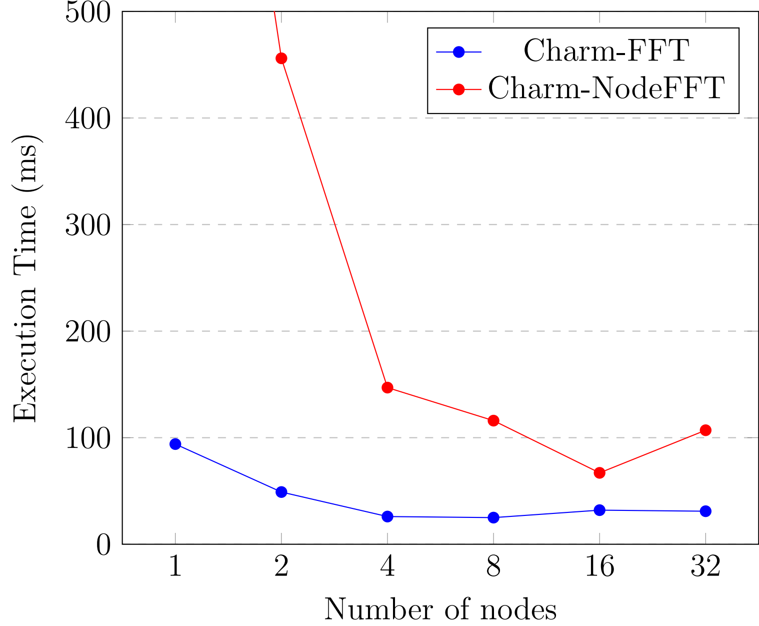


Figure 3.4: Single instance comparison based on the number of nodes; using 64 PEs per node

chare to a PE per state. To show this, Figure 3.4 was produced using 64 PEs per node to compute 300 x 300 x 300 FFT grid. As it can be seen, *Charm-FFT* reaches its optimal timing at 8 nodes, 1024 PEs and plateaus its execution time to about 25 to 30 ms. *Charm-NodeFFT*, meanwhile, does not benefit from this setting as much as *Charm-FFT* does. Because 64 PEs are available per node, each within-node parallelization using *CkLoop* will always attempt to use all 64 PEs for all *CkLoop* parts. At 32 nodes with total of 2048 PEs, it can be observed that the excessive attempt to multithread within each node eventually caused negative impact to the performance, producing 60% increase in execution time from 16 nodes. This implies that for *Charm-NodeFFT*, the overhead from multithreading in the transposition steps has surpassed the FFT computation time for each pencil *chare* after 16 nodes.

On the other hand, Figure 3.5 presents the performance comparison when only 16 PEs were used per node. This setting benefits *Charm-NodeFFT* as it was shown in Table 3.1 that 16 worker threads were the optimal number for *Charm-NodeFFT's CkLoop* usage. Unlike Figure 3.4, *Charm-FFT* shows gradual increase in performance until 64 nodes, which uses 1024 PEs total. *Charm-FFT's* execution time at 64 nodes with 16 PEs in each node closely matches its performance from Figure 3.4 at 8 nodes with 64 PEs in each node. This confirms

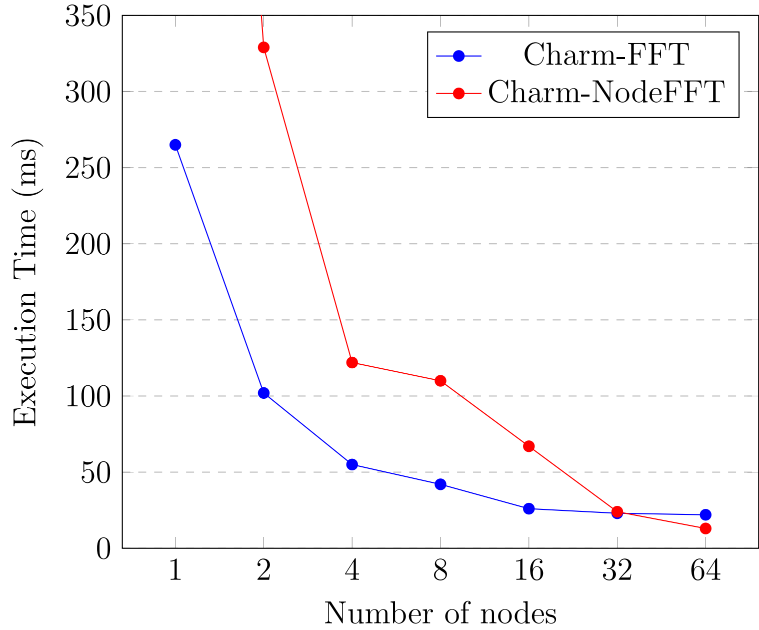


Figure 3.5: Single instance comparison based on the number of nodes; using 16 PEs per node

that for *Charm-FFT*, the performance is optimal when large number of PEs are in small number of nodes. Conversely, better performance from *Charm-NodeFFT* can be observed when only 16 PEs are used per node. At its optimal performance when using 64 nodes and total of 1024 PEs, it has 81% reduction in execution time compared to its performance from Figure 3.4 at 16 nodes with 64 PEs in each node, which also uses total of 1024 PEs. Thus for *Charm-NodeFFT*, using larger number of nodes with specific number of PEs in each node produces the best performance.

Charm-NodeFFT's irregular behavior at small number of nodes

In all performance comparisons presented in this section, *Charm-NodeFFT* consistently shows problematic performance for 1 and 2 nodes. Afterward there is a significant improvement in execution time between 2 and 4 nodes usage. This problem is always observed regardless of the number of PEs per node. For the current *Charm-NodeFFT*, this behavior is caused by how *CkLoop* is used for within-node parallelism. As previously described, *Charm-NodeFFT* involves *CkLoop* in two parts of each pencil *chare's* procedure. First is for multi-threading the FFT operation by dividing each pencil into multiple rows, and each PE will be given multiples of these non-overlapping rows to compute sequential 1D FFTs on

concurrently. The second place is just after the first *CkLoop* parts' ends. After all sequential 1D FFTs are complete in every PE that is used for *CkLoop*, each pencil *chare* needs to create messages to send and copy its FFT'ed segments to the corresponding messages. In current implementation, *CkLoop* is set to use each message as the chunk to distribute to the PEs. Thus, each PE will be creating the assigned messages, copying the segments from the node's memory and sending the filled in messages to its receivers sequentially. However, in case of 1 or 2 nodes, there are only 1 or 2 messages to send between the pencil *chares*, because there is almost no decomposition at 1 and 2 nodes by design. Since *CkLoop* is set to parallelize based on the messages, with only 1 or 2 messages per transpose steps, it is as if the entire pencil is handled by one PE, becoming the major bottleneck in the procedure. As a simple idea to mitigate this, *CkLoop* can be set to parallelize based on the segments of each message, rather than per message. However, this method can be expected to produce synchronization overheads between the PEs as it will have to be ensured that each message creation and copying is complete by all PEs working on it before the message is ready to send.

In addition to the message based multi-threading, the performance loss is also caused from the receiver *chares* when using small number of nodes. As described before, the receiver *nodegroup chare* directs one of its PEs to handle the incoming message. This allows the messages to be processed in parallel, reducing the time to process all messages. However, in case of 1 or 2 nodes, there are only 1 or 2 incoming messages. Thus only 1 or 2 PEs will be handling the incoming message, causing the similar bottleneck as the previous problem. To resolve this problem, when a message is received, it can be determined if the message is sufficiently large to use *CkLoop* to process the message by distributing the segments of it to be copied to the local buffer by each PE. This method will not introduce any new synchronization overhead as all PEs in a *nodegroup* should always be complete before it is ready to start the FFT operation.

3.3.5 Current Problem of Charm-NodeFFT

The initial purpose of reducing the number of communications was to minimize the communication overhead between the state transitions. *Charm-NodeFFT* was able to limit the necessary amount of communication cost primarily by increasing each pencil's size and secondarily by using within-node parallelism to mitigate the loss of parallelism from lack of decomposition. This approach was attempted since the processing time for the message transactions inclined faster than the actual FFT computation time as the number of available PEs increased.

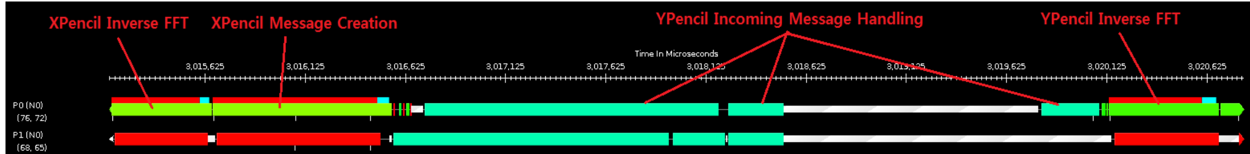


Figure 3.6: Timeline of pencil objects' activities in 2 PEs in a node

However, this conversion introduced other sources of bottleneck that led to overall performance loss. The newly identified problem that was introduced occurred mainly from the synchronization within a node for *CkLoop* operation. Within-node idle time that produced during the transposition steps between the nodes are further propagated from *CkLoop* synchronization.

During each part of *Charm-NodeFFT* that uses *CkLoop*, the work is divided among the specified number of worker threads. While the amount of time that each worker thread takes to execute the given work is approximately equal, it is not always guaranteed that all workers will start their execution at the same point, especially if there are other overlapping works in the background that are blocking the PEs. In current *Charm-NodeFFT's* implementation, when a node that is ready to compute one of its pencils, there are no background works that are running. If a pencil *chare* is ready to compute FFT, it implies that all messages are received and processed. However, because *CkLoop* will always break the FFT computation into the number of specified chunks, if there are any computation external to this library that occupies PEs, it can delay *Charm-NodeFFT's* stages that uses *CkLoop*.

Another cause of the performance loss is from the node's idle time while it is waiting for the other nodes' pencil *chares* to complete their computation and send their FFT'ed data, as shown in Figure 3.6. This timeline, generated with *Projections*[5], shows pencil *chare* events on 2 out of 16 PEs in a node, performing *X-pencil* to *Y-pencil* transition step. The first block, labeled *XPencil Inverse FFT*, is followed by *XPencil Message Creation* block which creates and sends messages to *Y-pencil chares*. Then, the colored blocks between the *XPencil Message Creation* and *YPencil Inverse FFT* are *Y-pencil chare's* entry methods that handle the incoming messages and fill in the local pencil *chare's* buffer. In this timeline, the idle period for all PEs, displayed by the white intervals between the *YPencil Incoming Message Handling* blocks, continues until the last expected message is received and processed. Only after then, *Y-pencil* operations start, and the PEs are utilized again.

This is a similar symptom that was displayed by *Charm-FFT* as well in Figure 2.2. *Charm-FFT* has the capability to utilize these idle times by mapping multiple finer-grained pencil *chares* to a PE. *Charm-NodeFFT*, however, has traded off this asynchrony with reduction in communication count. For *Charm-NodeFFT*, to fill in these idle times, multiple FFT

operations may run concurrently during each others' idle time or other computation may be overlapped as *Charm-FFT* does to continuously scale. This attempt for *Charm-NodeFFT* will be described in Chapter 5.

Chapter 4: Accelerator Usage for FFT Operations

The modifications made to *Charm-FFT* was an attempt to reduce the number of necessary communications between the *chares* during the transposition steps. *Charm-NodeFFT* achieves this by decomposing each dimension into larger pencil *chares* per state and distributing them to the available nodes.

While *Charm-NodeFFT* reduces the number of messages that each pencil *chare* must create and send, each pencil *chare* was given comparably larger pencils to compute. In *Charm-NodeFFT*, multithreading the computation step was implemented using *Charm++ CkLoop* library for serial FFT computations. However, using the multithreading for the within-node parallelization introduced another another potential bottleneck. Because all within-node operations are parallelized by multithreading, if any PE's parts of the operation lags behind, it will delay the whole node's progression, potentially further preventing the other nodes' from proceeding to the next pencil computation.

In the case when the number of available nodes is low and the FFT-grid to compute on is large, *Charm-NodeFFT* library will have to depend greatly on the number of available processing elements that *CkLoop* library can utilize to maintain the parallelism lost from the lack of node count. However, it is shown in the previous chapter that there is a limitation to how much parallelism can be expected within a node with the current implementation of *Charm-NodeFFT*. This finding suggests, if the size of the pencils that each *nodegroup chare* has is sufficiently large, it may be beneficial to offload the FFT computation of the entire pencil to an accelerator, such as GPU or Xeon Phi, if available in the local node.

Offloading the FFT computation to an accelerator in each node may not only remove the limitation bound by the multithreading but may also may open up a possibility of reducing the number of communications even further. By offloading, the option to use larger pencil size per node even further as accelerators are relied on for the within-node computation may be available.

In this chapter, *Nvidia cuFFT* library[7], a CUDA-based FFT library that performs FFT on supported GPUs, was used in place of FFTW[8] to offload FFT computations to the GPUs in each node. While *cuFFT* library supports a 3D-FFT operations on a 3D-FFT grid, for this experiment, only 1D transforms were used in the similar manner that FFTW[8] was used in *Charm-FFT* and *Charm-NodeFFT*.

4.1 MODIFICATIONS TO *CHARM-NODEFFT*

To run any computation on GPU, device memory on GPU has to be allocated in order to hold data that the user wants to run computation on. Because GPU cannot normally access the host memory which is accessible by the PEs, a separate memory for GPU has to be allocated for its accessibility. Then, data from the host memory has to be transferred to the allocated device memory. Often, a form of *cudaMemcpy* is used for this operation. Once the desired computation on GPU is complete, data in the device memory will then need to be transferred back to the host memory so that the PEs can manage the computed data.

In *Charm-NodeFFT*, this operation is performed each time when a pencil *chare* needs to run a series of 1D line FFTs on its buffer. During the initialization phase, each pencil *chare* constructs a *cufftPlan* to use when *cufft* execution call has to be made on its buffer. At this stage, the device memory on GPU to copy the buffer is also allocated. When a pencil *chare* properly fills in all segments of its buffer with incoming messages from other pencil *chare*, the buffer is then copied to the device memory. With the pre-constructed *cufftPlan* the PEs in the node are left idle until *cufft* execution completes. The device is then synchronized and the FFT'ed points in device memory is copied back to the host memory. Afterward, the message creation and packing steps are started and parallelized using *CkLoop*.

Depending on the application, the memory transfer counts for the inputs and outputs can be reduced if the computations before or after the FFT operation is still required to be performed in the device. However, to maintain the interface that *Charm-FFT* has initially designed, the user is expected to allocate memory on the host side and assign it to the *Charm-NodeFFT* even when using GPU for internal computation.

4.2 PERFORMANCE

Figure 4.1 shows the basic performance comparison of *Charm-NodeFFT* using *FFTW* [8] and *cuFFT* [7] for the 1D FFT computations. The table was produced computing 300 x 300 x 300 FFT grid with 4 worker threads per node. And Nvidia Pascal P100 GPUs was used in each node for *cuFFT*. Although it was shown from Chapter 3 that *Charm-NodeFFT* does not perform well in smaller number of nodes, only up to 4 nodes were used in this experiments to demonstrate effectiveness of the accelerator's computation power and its impact on *Charm-NodeFFT*.

As it can be observed from the figure above, when larger pencils are given to smaller number of nodes, *cuFFT* demonstrates its effectiveness. At 1 node, *cuFFT* shows 34.5% reduced execution time compared to *FFTW*. However, this reduction in execution time

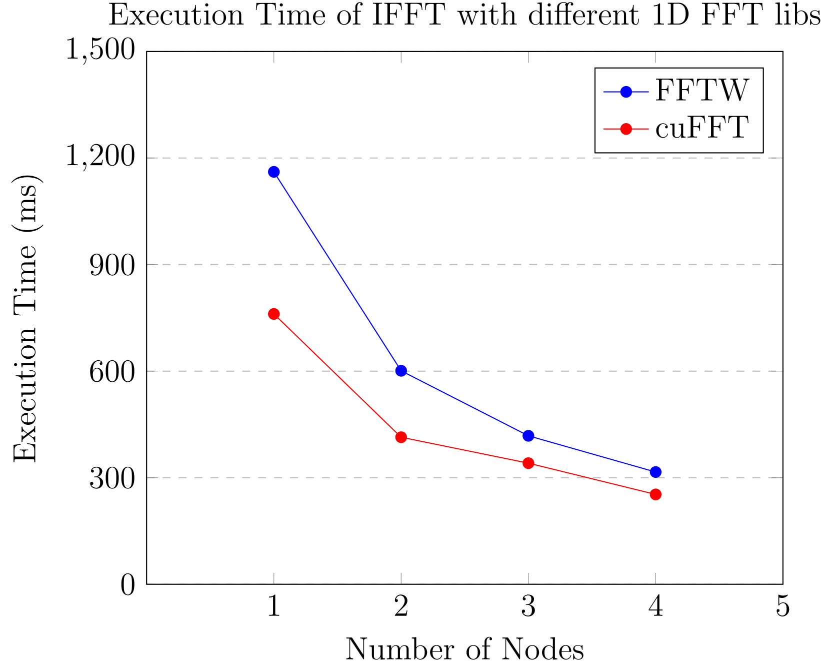


Figure 4.1: Comparison of using FFTW and cuFFT for the 1D FFT operations within Charm-NodeFFT

quickly diminishes as the pencil size per node decreases. By 4 nodes, *cuFFT* only produces 19.9% reduction in time, which will continuously decrease as the number of nodes increase.

The results show that if the whole FFT grid to compute is extremely large and the resource to run the computation is limited, it could be preferable to use accelerators like a GPU at lower number of nodes. As an example, the performance of *cuFFT* version on 2 nodes produced shorter execution time compared to the *FFTW* version on 3 nodes.

4.2.1 Data Transfer Time

Table 4.1 shows a simplified breakdown of GPU activities during the computation step of a node, profiled using *NVidia CUDA nvprof* tool [9]. As it can be seen, the combined amount of time that took to transfer the memory between the device and the host for the pencil occupies over 97% of the GPU operation. This test was performed on Nvidia Pascal P100 GPU with 300 x 300 x 300 FFT grid. It is speculated that as the input grid becomes sufficiently larger, the percentage of time spent to perform each dimension's FFT will rise. Regardless, the transfer time will be the dominating factor.

Because the usage of *cuFFT* showed performance improvement from the previous section when small number of nodes were used, it could be a useful idea to further optimize the

Operation	Total Time (ms)	Average Time (ms)	Op Count	Percentage (%)
CUDA Memcpy HtoD	174.94	29.157	6	56.05
CUDA Memcpy DtoH	128.97	21.495	4	41.32
Data Conversion	3.2862	0.822	4	1.05
D1 FFT	1.6445	0.822	2	0.53
D2 FFT	1.6243	0.812	2	0.52
D3 FFT	0.835	0.835	2	0.27

Table 4.1: Breakdown of GPU activity during the cuFFT operations

implementation in *Charm-NodeFFT* for a better result. Since the memory transfer time seems to be the main source of the time consumption for the GPU operation, a few options to optimize the memory transfer are considered.

Pinned Memory

Pinned memory denotes the page-locked memory space that *CUDA* uses when transferring data from host to device [10]. Since the user allocates pageable host memory, when the data transfer to the device is invoked, *CUDA* first copies the content of the pageable host memory to pinned memory first, which then is copied to the device memory. If the initial host memory is allocated to be the pinned memory, an unnecessary data transfer step can be avoided.

Unified Memory

Another optimization for the memory transfer is the usage of *Unified Memory* [11]. Unified memory is a memory space that is a managed memory space that designed to be accessible by both the host and the device by using prefetches to the pages and device memory as needed [11] *Charm-NodeFFT* as the frequent simultaneous access to the memory from both the host and the device is unnecessary, the unified memory may still be a sufficient optimization if continuous data transfer on demand during each pencil computation can be avoided.

Results of alternative allocations

The table 4.2 shows the comparison of Inverse FFT operation on 300 x 300 x 300 FFT grid using the default, pinned and unified memory allocation methods. In this result, the default allocation method ironically always produced the lowest execution times for all number of

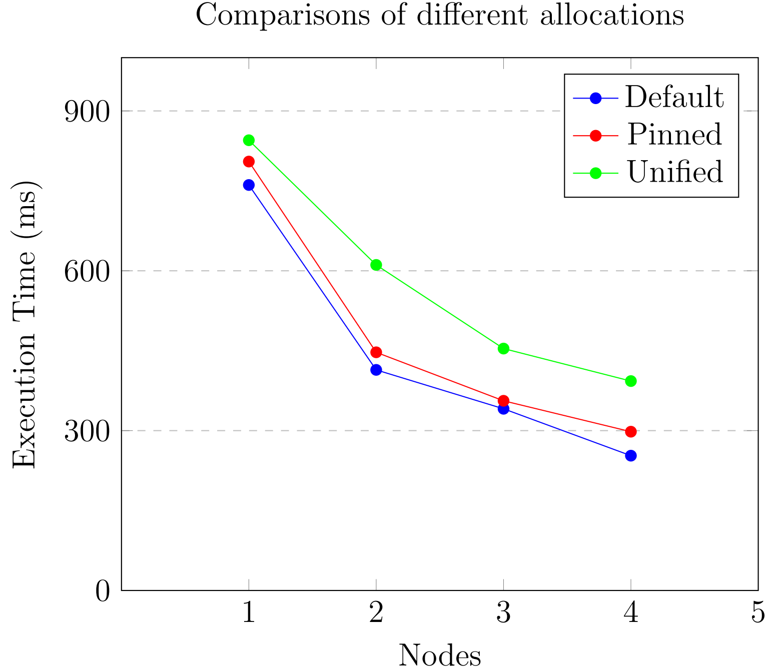


Figure 4.2: Inverse FFT times of Charm-NodeFFT when using different allocations for cuFFT

Allocation Type	FFT Time (s)	IFFT Time (s)
No GPU; FFTW	2.264516	1.98487
Default	1.274754	1.260624
Pinned	1.663768	1.448181
Unified	1.95958	1.66119

Table 4.2: Comparison of allocations when using larger grid of 500x500x500

nodes. While pinned memory allocation method stayed close to the optimal execution time that the default method produced, its performance does not necessarily seem to get better even when the node counts increase. It seems that the overhead of using the pinned memory allocation outweighs the performance benefit that it should provide. If there are occasions when the allocated memory need to be reused by GPU, using pinned memory would show its benefits more clearly. However, in *Charm-NodeFFT*, where the allocated memory is only used once per computation, the benefit of the pinned memory seems to be belittled. Unified memory, on the other hand, was noticeably slower than the other two methods. This is expected as the main benefit of the unified memory is to provide spontaneous access to the host without the need of frequent manual memory copying operations.

Table 4.2 is a simple result of computing 500 x 500 x 500 FFT grid on 4 nodes with 4 worker threads on each. Focusing on the inverse FFT time, the optimal method of using GPU in small number of nodes seems to be using the default allocation, reducing the execution time of the inverse FFT operation by 36.5% from the baseline *Charm-NodeFFT* that used CPU for its 1D FFTs.

In this chapter, accelerator was used in place of 1D FFT operation to see if the within-node parallelization that was previously relying on *Charm++ CkLoop* library can be alternated. As the result of the experiment, using GPU per node accelerates the operation if the number of nodes is sufficiently small and the pencil size that each node is handling is rather large. As more nodes become available and the finer pencil decomposition is used, the effect from using accelerator diminishes quickly, and the overhead to transfer the memory grows. While other allocation optimizations have been attempted, because of *Charm-NodeFFT's* characteristic of not using any allocated buffer more than once, the allocation optimization did not provide further benefit.

Chapter 5: Multi-Instance FFT

OpenAtom[4] was used as the standard application to test the efficacy of the modified *Charm-NodeFFT* library. Using the *OpenAtom* allows for a comparative analysis of the *Charm-NodeFFT* to be made since the *Charm-FFT* library has been developed for *OpenAtom* and is integrated in it. In addition, using *OpenAtom* could be hypothesized to be the most compatible application to the modified library, as *Charm-FFT* has already been well integrated and optimized within *OpenAtom* application. However, this is not to limit the use of *Charm-NodeFFT*, as the library can be integrated into other applications.

In a standard *OpenAtom* simulation, multiple instances are created during a lifetime of the application, and each instance holds certain size of FFT grids that are used for density function. These instances are independent of each other during their respective 3D FFT operation. The original *Charm-FFT* handles multiple *OpenAtom* instances by allowing multiple *Charm-FFT* library instances to be created and be computed independently. Furthermore, because the library instances do not need to be synchronized with each other throughout the application, *Charm-FFT* utilizes *Charm++'s* asynchronous nature by distributing each *Charm-FFT* instance's 2D decomposed pencil *chares* across the available PE's and having them run concurrently.

The *Charm-NodeFFT* library, however, is designed to utilize all available nodes for each library instance. Thus, if multiple *Charm-NodeFFT* instances are created in an application in the same manner that *Charm-FFT* does, it would simply increase the number of different instances' pencil *chares* per node, where the parallelism within a node would be bound by the number of available cores per node regardless of the actual number of nodes in the system. Instead, when a node size and the count of the instance that the simulation attempts to run with is specified, a subset of the nodes can be assigned per instance so that all necessary communication for an instance would only occur within this assigned subset of the nodes.

5.1 MODIFICATIONS TO *CHARM-NODEFFT*

To create a library instance for the standard *Charm-NodeFFT*, calling *Charm_createFFT* once with the expected inputs from any of the processor is sufficient. To create multiple library instances, the user would need to call *Charm_createFFT* for each instance that needs to be created with their respective inputs. After the library instances are created, these instances can be distinguished from another using the identifiers within the messages that are sent to the user-input callbacks. However, these instances are not informed of other *Charm-*

NodeFFT instances that may be running concurrently. This creates *nodegroup chares* and pencil *chares* as if only one instance is running in the application. In this case, each library instance creation will simply just add more library instances per node.

For each library instance to be aware of other instances that the user is expecting to create, the total number of instances in the application and the current instance's linear offset, for the identifier, are required as the additional arguments for *Charm_createFFT*. During the library instances' creations, based on the additional inputs, each instance will internally identify the available nodes for the current application by first creating the *nodegroup chares* and then assign itself to the subset of the created *nodegroup chares* as evenly as possible. While the steps for the pencil *chares* constructions remain the same, the mapping of these *chares* is altered so that the *chares* will be mapped only amongst the designated subset that their library *instance's* are assigned to. Because of this, the user will also need to adjust the arguments for pencil decomposition sizes based on the environment that the application will be running on.

Even though each instance is aware of other library instances in the application by the provided inputs, these instances compute their process independently and do not need to be synchronized once the libraries are properly initialized. Each instance will follow the procedure that the standard *Charm-NodeFFT* performs, just within the nodes that it was assigned. When an instance's operation is complete, its user-input callback function will be invoked even if other instances may still be computing. Because of this, an instance may compute multiple iterations of FFTs regardless of the status of other concurrently running instances.

5.2 PERFORMANCE

5.2.1 Test Design

The original *Charm-FFT* was initially designed to be compatible with *OpenAtom* application. *Charm-NodeFFT*, while being a modification of *Charm-FFT*, requires a different mapping of the controller *chares* in *OpenAtom* application to be able to access the input and output data using *nodegroup chares*. In addition, because *Charm-FFT* library has already been integrated into *OpenAtom* application, optimizations that specifically supports the *Charm-FFT's* input and output *chare* layouts have already been implemented. This creates difficulties in applying the same optimizations to *Charm-NodeFFT* due to the compatibility reasons. Thus, it is difficult to observe the effect of the *Charm-NodeFFT* modifications in isolated manner. Instead, a simple imitation that simulates the *OpenAtom's* density

Number of Instances	Avg FFT Time (ms)	Avg IFFT Time (ms)	Completion Time (ms)
1	53	6	61
2	50	6	60
3	52	7	66
4	53	10	70
8	58	35	101
16	82	74	165

Table 5.1: Effect of increasing the number of Charm-NodeFFT instances in 8 nodes

calculation stage that uses 3D FFT for each of its instance was created.

In a standard *OpenAtom* run, each *OpenAtom* instance contains electron density in g-space and real-space that are respectively represented as separate *chare* arrays. These *chares* are used to perform series of FFT and inverse FFT[4]. To imitate this, in the test simulation, a *chare array* is created to function as *OpenAtom* instances. Each instance *chare* will construct its own electron density space representations as *chare arrays* based on the user input FFT grid size. When all instances and their density spaces are initialized, all instance *chares* performs a FFT followed an inverse FFT individually. In the simulation, each *OpenAtom* instance will create a set of electron density space *chare* arrays called *RhoR* and *RhoG*. Then, each instance further creates a FFT library instance that will be used to compute FFTs on each of the density space *chare*. When all instances for the test are ready to begin the FFT operation, all instances start their procedures independently. Then, at the end of its procedure, it contributes to a reduction to alert that its computation is complete. When all instances have contributed to the reduction, an iteration of the simulation ends.

5.2.2 Multi-instance handling comparison

When there are multiple FFT library instances to run in an application, *Charm-NodeFFT* can be used in a similar manner that *Charm-FFT* handles multiple instances by allowing the instances to run in all nodes as available. In *Charm-NodeFFT's* implementation, this method will allow each instance to use all available nodes, and within each node, the instances' pencil *chares* will use available PEs to compute their operations. On the other hand, a subset of nodes to each instance can be assigned to perform the standard *Charm-NodeFFT* within the instances' respectively assigned nodes.

Using all nodes for each instance

Table 5.1 shows the effect of increasing the number of instances in a given number of nodes. For this test, each instance was given 100 x 100 x 100 FFT grid to compute, and each instances' *Rho chares* were constructed to best utilize the all available nodes for the test. The test was run on 8 Haswell nodes with 24 cores on each.

As shown in Table 5.1, the majority of the operation time is taken by FFT operation which contributes to a substantial performance loss due to the unoptimized multi-threading part that was described in Chapter 3. However, it can be seen that as more instances are added, the average inverse FFT time increases, showing a significant rise in the computation time if the number of cores to use per instance becomes scarce. In such case, the pencils of the instance that are ready to compute have to wait until the occupying pencil *chare* releases the resource. Thus, when larger number of instances are used, some of the instances simply act as if they are performing FFT operations after another instance is complete, unable to run their operations simultaneously.

Because of this, the method of increasing the number of nodes for *Charm-NodeFFT* does not increase the number of instances that can be running concurrently. Although having more nodes will allow reduced average FFT and inverse FFT operations, the same lack of resource per instance will occur, leading some pencil *chares* to be computed as if they are scheduled to run in sequence.

Using only a subset of nodes for each instance

Table 5.2 and Table 5.3 show the result of assigning a subset of nodes to 2 and 4 instances. Same as before, each instance was given 100 x 100 x 100 FFT grid with its pencil *chares* and *Rho chares* decomposed to best utilize its assigned subset of nodes. The average FFT time still shows its bottleneck from its unoptimized multi-threading parts even though it shows its improvement in performance as larger number of nodes are used for each instance. Also when only one node is given per instance, it is unable to effectively utilize the available PEs, as described in Chapter 3.

Number of nodes	Avg FFT Time (ms)	Avg IFFT Time (ms)	Completion Time (ms)
2	85	58	223
4	67	12	108
8	47	9	61
16	41	7	59

Table 5.2: Using a subset of nodes for 2 instances

Number of nodes	Avg FFT Time (ms)	Avg IFFT Time (ms)	Completion Time (ms)
4	105	47	182
8	89	14	112
16	76	10	94

Table 5.3: Using a subset of nodes for 4 instances

Table 5.2 shows that with 2 instances, increasing the node count from 2 to 16 reduces the overall completion time by 74%. In this case, each instance was given 1 to 8 nodes to compute on, and synchronization between these two instances was not needed. As a result, there are only 18% difference in the completion time of both instances and the average FFT and inverse FFT times combined. Comparing the completion time and the combined average times may not be an appropriate method to confirm that the instances are running concurrently. But, in this method, if the FFT or the inverse FFT operations are synchronized between the instances and are caused to be computed in sequential order, as it did with the previous method when too many instances were given for available cores in a node, overall completion times would be very inflated even if the average computation times of instances would seem low.

Table 5.3 shows the performance of using the same method with 4 instances instead of 2. Similar to 2 instances, the reduction in the completion time and the average computing time from 1 node per instance to 2 and 4 is still visible. But, the overall completion time and the inverse FFT time for 4 instances is higher than 2 instances even when the same number of nodes are given to each instance. For example, when 8 nodes are used for 2 instances, giving 4 nodes per instance, it produced 61 ms for the completion time, while 4 instances test has 59% increased completion time of 97 ms when 4 nodes are given to each of its instances as well. Even if the average operation time can be improved when each instance is provided larger subset of nodes, this steady increase in completion time is still observed as more instances are added.

In this test that was created to imitate just the FFT operations of *OpenAtom* density function, the completed instances wait for the other incomplete instances to measure the overall timing. However, in a standard application where instances are independent of each other, when the concurrently computing FFT operations are complete for some, they can move on to start the next steps without necessarily waiting for the delayed instances. So in such case, the overall completion time is less impactful than it would seem while the average FFT operation time that multiple instances can perform in parallel would be more meaningful.

Chapter 6: Conclusion

In this thesis, a different approach to increase the scalability of a communication intensive algorithm, a parallel FFT, has been explored based on an existing parallel FFT library, *Charm-FFT*. The primary method of this approach was to reduce the number of communication that often becomes costlier than the actual FFT time. This is achieved by adjusting the decomposition granularity of the pencil objects. These pencil objects are then distributed to the available nodes rather than the available PEs so that each node will be given relatively larger sized pencils to compute. The loss of parallelism was mitigated using multithreading, effectively parallelizing within-node computation without producing additional communications.

By the decomposition design of *Charm-NodeFFT*, each node was given sufficiently larger pencil. In order to further optimize the within-node FFT computation that was relying on multithreading, usage of the accelerators was attempted. In this experiment, the pencils were offloaded to GPU to measure the necessary data transfer time and the overall computation time. While the usage of GPU was capable of producing faster computation result, its benefit was retained by the data transfer time. Because of this, its benefit diminished as the number of available nodes became larger. To mitigate this, other types of device allocation have been tested, which were unnecessarily producing positiveness performance.

To follow the *Charm-FFT's* original intent of optimizing *OpenAtom* application, the design of *Charm-NodeFFT* had to be adjusted so that multiple library instances could be handled simultaneously and run concurrently. Due to its similarities to *Charm-FFT*, *Charm-NodeFFT* can be used by simply adding multiple *Charm-NodeFFT* instances per node and by stacking up multiple library instances in every node. However, this design is inferior to *Charm-FFT* as it imitates *Charm-FFT's* pencil distribution, only using multiple instances to accomplish it. Because of this, the library has been altered so that it would know the other *Charm-NodeFFT* instances that would be running concurrently and adjust the initialization accordingly so that their resources do not overlap.

Charm-NodeFFT is designed to always use all of the available nodes that it is given. Because of the lack of flexibility, the library does not attempt to use only the subset of the nodes for a smaller grid's computation. Further, it is currently unable to provide custom maps to only utilize a subset of the nodes for performance optimization. Thus, when a significantly large number of nodes are available with very small FFT grid to compute on, *Charm-NodeFFT* will naturally be inferior to the original *Charm-FFT*. In addition, because only one pencil *chare* is mapped per node, the node usage for *Charm-NodeFFT* is

quite low. While this can be mitigated by *Charm++'s* feature of asynchrony to run other background works, the design of *Charm-NodeFFT* naturally wastes a large portion of each node's available resource.

References

[1] A. Chan, P. Balaji, W. Gropp, and R. Thakur, "Communication analysis of parallel 3d fft for flat cartesian meshes on large blue gene systems," in *International Conference on High-Performance Computing*. Springer, 2008, pp. 350–364.

[2] N. Jain, "Optimization of communication intensive applications on HPC networks," Ph.D. dissertation, Dept. of Computer Science, University of Illinois, 2016.

[3] B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, E. Mikida, X. Ni, M. Robson, Y. Sun, E. Totoni, L. Wesolowski, and L. Kale, "Parallel Programming with Migratable Objects: Charm++ in Practice," ser. SC, 2014.

[4] S. Kumar, Y. Shi, E. Bohm, and L. V. Kale, "Scalable, fine grain, parallelization of the car-parrinello ab initio molecular dynamics method," UIUC, Dept. of Computer Science, Tech. Rep., 2005.

[5] L. V. Kale, G. Zheng, C. W. Lee, and S. Kumar, "Scaling applications to massively parallel machines using projections performance analysis tool," in *Future Generation Computer Systems Special Issue on: Large-Scale System Performance Modeling and Analysis*, vol. 22, no. 3, February 2006, pp. 347–358.

[6] C. Mei, "Message-driven parallel language runtime design and optimizations for multicore-based massively parallel machines," Ph.D. dissertation, Dept. of Computer Science, University of Illinois, 2012.

[7] NVIDIA, *CUFFT library Version 10*, 2018. [Online]. Available: https://developer.nvidia.com/cufft

[8] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005, special issue on "Program Generation, Optimization, and Platform Adaptation".

[9] NVIDIA, *CUDA Toolkit Documentation - nvprof*, 2018. [Online]. Available: https://docs.nvidia.com/cuda/profiler-users-guide/index.html#nvprof-overview

[10] M. Harris, *How to Optimize Data Transfers in CUDA C/C++*.

[11] M. Harris, *Unified memory in CUDA 6*.