

Improving Job Node Allocation on HECToR in Various Topological Manifestations

Larisa Stoltzfus

August 21, 2013

MSc in High Performance Computing

The University of Edinburgh

Year of Presentation: 2013

Abstract

The compactness (i.e. the spatial relationship between nodes allocated) of the placement of a job on a HPC machine such as HECToR can have a large effect on its performance (over %10)[7][1]. At the moment, HECToR has not been subjected to any optimisation in this area. Carl Albing (of Cray) has recently produced a doctoral thesis with suggestions as to how performance may be improved using job node allocation algorithms for 3D torus topologies, as employed on HECToR. This project first explored the similarities and differences between his results on another another Cray XE6 machine (the same architecture as HECToR) using a simulator. Results showed that even on the same architecture, the job size distribution can have a significant effect on how effective job placement algorithms are.

Initially two algorithms were tried out for the Cray XE6 machine comparison: first fit and FIFO. The first fit algorithm performed well for small jobs and the FIFO algorithm showed potential for large jobs. The next step of the project involved trying to bridge this gap by finding an algorithm that worked well across the whole spectrum of job sizes. The most overall improvement for small and large jobs was seen with a “closest fit” algorithm, which works by the minimizing the largest ordinal distance between an allocated group of nodes. With this algorithm, the improvement seen for small jobs was over %70, for large jobs %28 and overall %31.

The final stage of the project involved trying all the algorithms out on other topologies - including the dragonfly and 5D torus - using the same number of nodes that HECToR currently has. These simulations confirmed that the network topology also has an effect how effective job placement algorithms are, however the same two families of algorithms were the most effective across all the topologies. The closest fit algorithm performed the strongest, more so on the dragonfly network and less so for the 5D torus. Overall, the placement of jobs had a mixed effect on these more highly connected networks.

Contents

1	Introduction	1
2	Background and Literature	3
2.1	Network Affect on Performance	3
2.2	Network Topology	4
2.3	Optimizing Job Placement	4
2.3.1	Dynamic Allocation Algorithms	5
2.4	Job Placement Metrics	6
2.4.1	MIND and Improvement	6
2.5	HECToR	7
2.5.1	Specifications	7
2.5.2	Log Data	8
2.6	Previous Research on Cray XE6 Machines	9
3	Simulator Code	10
3.1	Original Code	10
3.2	Updated Code	11
3.3	Testing	12
4	Comparison of Job Node Allocation Algorithms on Cray XE6 Machines	14
4.1	Job Size Distributions	14
4.2	Preliminary Dynamic Allocation Algorithms	15
4.2.1	First Fit Results	16
4.2.2	FIFO Results	20
4.2.3	General Discussion of Similarities and Differences	24
5	New Dynamic Allocation Algorithms	26
5.1	Largest Fit	26
5.1.1	Algorithm Concept	26
5.1.2	Results	28
5.2	FifoFit	29
5.2.1	Algorithm Concept	29
5.2.2	Results	30
5.3	Varying Fit	31
5.3.1	Algorithm Concept	31
5.3.2	Results	32

5.4	Closest Fit	33
5.4.1	Algorithm Concept	33
5.4.2	Results	35
6	Further Analysis of Job Allocation Algorithms	38
6.1	Normalization of Improvement	38
6.2	Threshold Investigation	41
6.3	Averages Comparison	45
6.4	T-Test Analysis	48
6.5	Performance Timings	50
7	Results On Other Topologies	51
7.1	Code Refactor	51
7.2	Dragonfly	52
7.2.1	Best Case Scenario	53
7.2.2	Worst Case Scenario	55
7.3	5D Torus	58
7.4	Best Overall MIND Comparisons	61
7.5	Individual MIND Comparisons	63
8	Conclusions	72
9	Future Work	75

List of Tables

1	Top Three Algorithms for 3D Torus Topology	48
2	Algorithm Data Analysis and T-Test Results	49
3	Timing Comparisons for Simulation Runs Using Different Algorithms	50
4	Improvement Values for the Top Three Algorithms for Best Case Dragonfly Topology	55
5	Improvement Values for the Top Three Algorithms for Worst Case Dragonfly Topology	58
6	Improvement Values for the Top Three Algorithms for 5D Torus Topology	61
7	Top Algorithms By Topology	73
8	Comparisons Between the Topologies of Actual Change in MIND Values	74

List of Figures

1	Fragmented Job on HECToR (Size 8)	4
2	MIND Equation	6
3	3D Torus Network Topology	7
4	Inside One of HECToR's Cabinets	8
5	Original Simulation Code UML Diagram	11
6	Updated Simulation Code UML Diagram	12
7	Simulation Test Package UML Diagram	13
8	Number of Jobs versus Job Size for Jobs on HECToR	15
9	Number of Jobs versus Job Size for Jobs on NERSC	15
10	Improvement versus Job Size for Original and Fixed Fit Algorithms Run on HECToR Log Data	16
11	Functions to Allocate Nodes and Get Suitable a Chunk Size For First Fit Algorithm	18
12	Improvement versus Job Size for Fit-1 Algorithm on HECToR and NERSC Data	19
13	Improvement versus Job Size for Fit-8 Algorithm on HECToR and NERSC Data	20
14	FIFO Algorithm	21
15	Improvement versus Job Size for FIFO-2 Algorithm on HECToR and NERSC Data	22
16	Improvement versus Job Size for FIFO-32 Algorithm on HECToR and NERSC Data	23
17	Improvement versus Job Size for FIFO-128 Algorithm on HECToR and NERSC Data	24
18	Functions to Allocate Nodes and Get a Suitable Chunk Size For Largest Fit Algorithm	27
19	Improvement versus Job Size for Largest Fit Algorithm	29
20	Improvement versus Job Size for Select FifoFit Algorithms	31
21	Function to Get Gap Size in FirstFitRevised Class	32
22	Improvement versus Job Size for Varying Fit Algorithm	33
23	Functions to Get List of Closest Fit Nodes and Largest Distances Between Nodes in a List	34
24	Improvement versus Job Size for Closest Fit With Minimized Largest Distance Algorithm	36
25	Improvement versus Job Size for Closest Fit With MIND Algorithm	37
26	kAU Value versus Job Size for HECToR	39

27	Improvement versus Job Size for Best Performing Algorithms on 3D Torus	40
28	Normalized Improvement versus Job Size for Best Performing Algorithms on 3D Torus	41
29	Graph of Average Improvement Versus Gap Size For First Fit, FIFO and Largest Fit Algorithms	42
30	Graph of Low Job Size Average Improvement Versus Gap Size For First Fit, FIFO and Largest Fit Algorithms	43
31	Graph of Large Job Size Average Improvement Versus Gap Size For First Fit, FIFO and Largest Fit Algorithms	44
32	Overall Averages	46
33	Normalized Overall Averages	47
34	Refactored Simulation Code for New Topologies UML Diagram	52
35	Dragonfly Network Topology	53
36	Improvement versus Job Size for Algorithms on the Best Case Dragonfly Topology	54
37	Normalized Improvement versus Job Size for Algorithms on the Best Case Dragonfly Topology	55
38	Improvement versus Job Size for Algorithms on the Worst Case Dragonfly Topology	57
39	Normalized Improvement versus Job Size for Algorithms on the Worst Case Dragonfly Topology	58
40	5D Torus Network Topology	59
41	Improvement versus Job Size for Algorithms on the 5D Torus Topology	60
42	Normalized Improvement versus Job Size for Algorithms on the 5D Torus Topology	61
43	Average MIND Value With Closest Fit Algorithm versus Job Size	62
44	Average Δ MIND Value Between Closest Fit Algorithm and FIFO-0 versus Job Size	63
45	Minimum, Maximum and Average MIND Values versus Job Size for 3D Torus Topology Using the Original Job Allocation Algorithm	64
46	Minimum, Maximum and Average MIND Values versus Job Size for 3D Torus Topology Using Various Job Allocation Algorithms	65
47	Minimum, Maximum and Average MIND Values versus Job Size for Best Case Dragonfly Topology Using the Original Job Allocation Algorithm	66
48	Minimum, Maximum and Average MIND Values versus Job Size for Best Case Dragonfly Topology Using Various Job Allocation Algorithms	67
49	Minimum, Maximum and Average MIND Values versus Job Size for Worst Case Dragonfly Topology Using the Original Job Allocation Algorithm	68

50	Minimum, Maximum and Average MIND Values versus Job Size for Worst Case Dragonfly Topology Using Various Job Allocation Algorithms	69
51	Minimum, Maximum and Average MIND Values versus Job Size for 5D Torus Topology Using the Original Job Allocation Algorithm	70
52	Minimum, Maximum and Average MIND Values versus Job Size for 5D Torus Topology Using Various Job Allocation Algorithms	71

Acknowledgements

Big thank you to Andy Turner for supervising me and answering my many, many questions that cropped up during my work on this project and about the field as a whole. Another big thank you to Carl Albing of Cray for answering even more questions and letting me use his simulator code for this project. Apologies for ripping much of it apart and putting it back together slightly differently.

Also thank you to my employer, Ikon Science - in particular Paul Jones and Phil Wild - for letting me work so flexibly as a student. This year would have been a lot less feasible without this support.

And of course thank you to EPCC for providing this master's program. I enjoyed it immensely and already want to pursue another master's.

I am also forever indebted to my parents for their unwavering belief and support, in particular my mother for insisting that 29 was an excellent age to be going back to university.

Finally, I am so grateful to my pals Matthew and Andrea who have done so much for me this year in providing me with a home away from home.

1 Introduction

Modern supercomputers use schedulers to allocate jobs to free nodes. With no optimization in place, jobs can be placed on any available nodes. Often job placements become dispersed across the network, leading to increased distances that must be traversed for communication between nodes. Where nodes communicate frequently between one another in a job, this extra distance can cause a job to run for longer than necessary.

It is however possible to apply algorithms to minimize the distance between these allocated nodes, thus decreasing communication traffic overall - particularly for jobs with higher levels of communication. Previous research has shown that performance gains of more than %10 could be seen from doing this[7][1]. The purpose of this project is to explore such algorithms, in particular looking for ones which works well over a wide range of job sizes. Additionally, different topologies were also looked at to explore what kind of effect this had on which algorithms showed the most improvement in compactness.

This project used a simulated job scheduler to explore the effects different job scheduling algorithms had on the average distance between nodes (as an indication of possible performance increase) for the machine HECToR. First, results were corroborated with previous runs on a machine with similar architecture to HECToR. Next, new algorithms were developed, implemented and run on HECToR log data. Finally, new topologies using the same number of nodes as HECToR were mocked and the same algorithms were run to compare their effects on different network topologies.

Chapter 2 lays down the background work behind the project, including information on:

- supercomputing network performance
- network topology
- job placement optimisation
- metrics for ascertaining improvement in placement
- specifications of HECToR
- previous research on Cray XE6 machines

Chapter 3 outlines the structure of the simulator code in its original form, the upgrades made to create the results seen in this project and the tests that were created to assure correct results.

Chapter 4 presents the two job distributions seen in previous research on another Cray XE6 and on HECToR and compares the results of the two initial job allocation algorithms run on

the log data from both machines.

Chapter 5 introduces several new job allocation algorithms and presents their results using HECToR log data.

Chapter 6 analyses the results of running the various algorithms from several other angles.

Chapter 7 covers the work trying out all the algorithms on other network topologies.

Chapter 8 wraps up the project with some conclusions.

Chapter 9 proposes possible extensions to the project that could be carried out in the future.

2 Background and Literature

Supercomputers are often built from individual machines (or parts of) to form what is known as clusters. These clusters connect together smaller groups of processors, also called nodes, with some form of cabling or other connection to form a network. The number of nodes on a network can be on the order of tens, hundreds to thousands and even millions. A program submitted to be run on a supercomputer (or job) is allocated a number of these nodes as prescribed by the submitter, which defines the job size. A job is then run on groups of these separate nodes, which communicate with one another by sending data and passing messages across the network.

2.1 Network Affect on Performance

Many factors can affect performance for a job on a particular node configuration, including network latency and bandwidth. Network latency is a measurement of the delay in communication between nodes on the network and bandwidth is the speed that data can traverse across the network[5]. Once a physical system is set up though, these factors become much harder to change. Something that is easier to change, however, is what configuration of nodes a job is running on.

When jobs are placed in no particular order on nodes on the network, “fragmentation” can occur (see Figure 1) which can exacerbate existing communication bottlenecks[6]. Where this happens, any communication that goes on between nodes will have to traverse past many other nodes in the network each time data must be passed. This can be problematic for a few reasons. Firstly, the physical distance for the data to travel is needlessly increased, thus increasing the latency. Secondly, other surrounding nodes may also be in use and competing for the same network paths thus decreasing the bandwidth for each job. As such, the distances between nodes a job is run on can add up and have a noticeable affect on job performance.

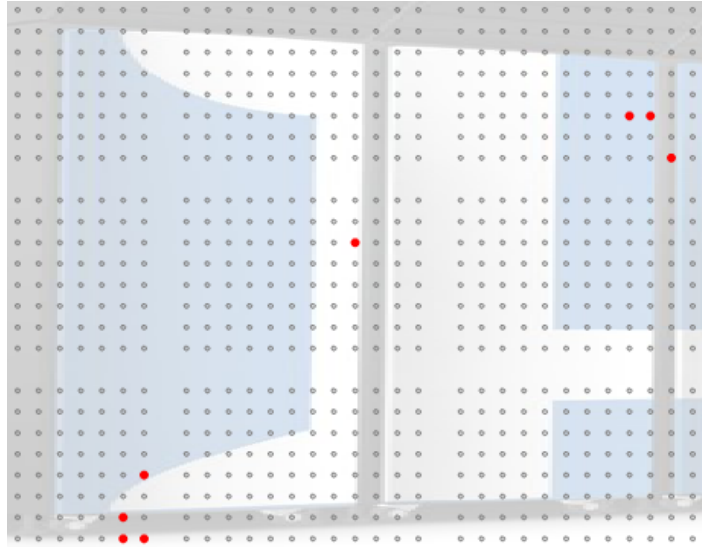


Figure 1: Fragmented Job on HECToR (Size 8) [3]

2.2 Network Topology

The network topology is the shape the network takes when connecting separate nodes together. There are many variations of topologies that can be found in supercomputing clusters, ranging from simple structures like rings and meshes, to loop-around toruses and newer ones like the dragonfly[5][15]. The differences in these shapes help determine how quickly data can traverse between nodes in the network. A simple metric for determining this is “hops,” which is the number of nodes a message must traverse through to get to the node the message is intended for. The closer together the nodes a job is allocated are, the fewer the number of hops that is required to pass information between them.

2.3 Optimizing Job Placement

Because jobs are constricted by their placement on the network topology they are running on, optimisation of job node configuration can be beneficial. This optimisation on the network can be approached in a number of ways. Two of the main types are: node ordering and node allocation algorithms[1]. Node ordering attempts to optimise the ordering of the nodes themselves in a topology. When the scheduler selects free nodes on the network for a job, it chooses from these nodes in a particular order, some of which are more optimal than others. Node allocation algorithms on the other hand select free nodes from the scheduler in nearby groups without necessarily considering the order they are in. This project focused on the latter type of optimisation.

Although the optimisation of the placement of jobs is an NP-complete problem[10], recent research has shown some promising results[6][7]. Two of these include bin-packing and dynamic allocation algorithms. Bin-packing algorithms look for consecutive available space to hold specific sized bins, whereas the dynamic algorithms focus more on looking for free nodes in the vicinity of one another. Bin-packing algorithms have been used frequently for simulating storage in areas such as warehouses or trucks. However, where consecutive free nodes of a particular size cannot be found, jobs must wait until the space is available, even where there are enough nodes free. On a supercomputer, letting nodes go idle would waste valuable resources. Additionally, these algorithms can take a long time to run - some have estimated on the order of hours[11], while others rely heavily on heuristics[8]. Dynamic allocation algorithms, on the other hand, can be much more flexible and quick. Indeed, research has shown that they can produce less fragmentation than algorithms seeking out contiguousness[6]. For these reasons, this project has further focused in this direction accordingly.

2.3.1 Dynamic Allocation Algorithms

Dynamic allocation algorithms were originally conceived for use in memory storage, where chunks of variable-sized consecutive blocks are allocated. An important distinction between these types of algorithms and bin-packing ones is that while contiguity is ideal, it is also less pertinent[10]. Allocations do not need to be fully contiguous as there is still much improvement to be seen by placements which are even a little closer together. While dynamic allocation algorithms may have been created for memory allocation, they can easily be applied to other areas where improvement in compactness can be seen. Because of their speed and flexibility, dynamic allocation algorithms are more suitable for use by a job node scheduler where time and space are more precious resources.

There are many possible kinds of dynamic allocation algorithms available[9]. Generally, they seek out a suitably large fit of a particular size by traversing through the free space available and differ in when to stop looking and also where this approach fails. For example, one type is the “first fit,” where the algorithm searches for the first available free space. Where enough consecutive free space cannot be found, the algorithm takes a recursive divide-and-conquer approach[1]. This is explained more in Section 4.2. While memory allocation algorithms are predominantly designed for one dimensional arrays, for network topologies these algorithms need to be able to find contiguity in more than one dimension. Here the role of node ordering also comes into play, which helps define what is considered “consecutive.”

2.4 Job Placement Metrics

In order to properly ascertain how “well-placed” a job is, that is to capture how compact a job node placement is, a metric must be used in order to compare the effectiveness of different algorithms and their corresponding parameters. Again, there are a number of possible options, however the three main ones are: diameter, hop-bytes and MIND[1]. Diameter measures the distance between the two furthest nodes. However, this does not take very well into account shape in the sense that two jobs, one with evenly distributed nodes and one with all nodes but one in close proximity, would both return the same value for the diameter. Hop-bytes includes the byte count of each message, which is information that is not taken into account (nor included in the logs) for this project. This is because the aim is to find allocation algorithms that work well across the spread of jobs that are run on HEC-ToR. Including this information would not help towards finding a good middle ground as hop-bytes are very application specific. The final option is MIND, which is shorthand for Mean Inter-Node Distance. MIND gives a general average of the distance between nodes and is used for analyzing the improvement seen by different algorithms in this project.

2.4.1 MIND and Improvement

Figure 2 describes the MIND equation, where the MIND value of a torus T for a job comprised of S nodes is equal to the sum between all nodes of the Manhattan Distance (d_{ij}) between two particular nodes divided by one half the number of nodes in the placement (s) times the number of nodes in the placement minus one. The equation takes into account wrap-around effects of a network topology using modular values when calculating the Manhattan distance. While the MIND value gives the best general average of a “good fit,” it is also fairly computationally expensive (on the order of $O(N^2)$). The program to compute the MIND values of the simulated placements in this project was parallelised for this reason. This metric has been used for all the algorithms and topologies that were simulated. Values reported and compared are of the percentage change in the mean MIND average between the algorithm run with no placement preference and a new algorithm. This additional metric will henceforth be used interchangeably with the “improvement” seen in an algorithm for a particular job size.

$$MIND(T, S) = \frac{\sum_{i=1}^{s-1} \sum_{j=i+1}^s d_{ij}}{\frac{1}{2}s(s-1)}$$

Figure 2: MIND Equation

2.5 HECToR

2.5.1 Specifications

The overall goal of this project was to find ways to optimise the job placement of programs running on the hardware seen in Cray XE6 supercomputers. In particular it focused on HECToR, which is the UK national supercomputer based in Edinburgh and is a Cray XE6[2]. A picture of one of the cabinets in HECToR can be seen in Figure 4. HECToR has 2816 computing nodes, each of which has 32 processing cores. These nodes are connected into a 3D torus network, as can be seen in Figure 3. The dimensions of HECToR are 15x6x16, with two nodes at each coordinate and where 64 of the nodes are used as login nodes. On HECToR, the maximum number of hops across the network is 18.

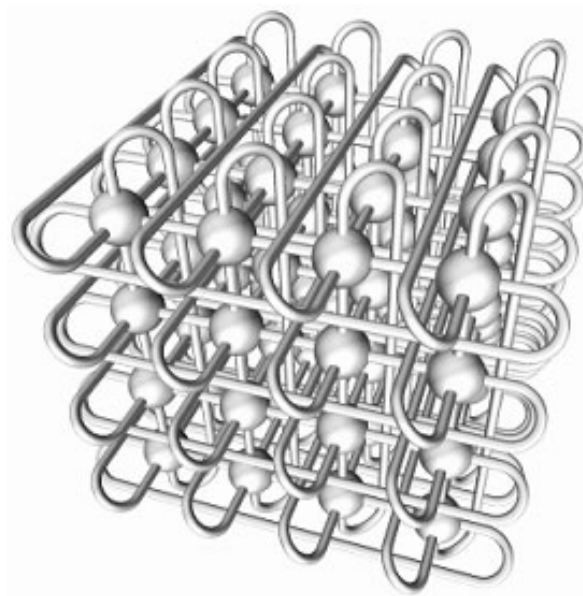


Figure 3: 3D Torus Network Topology [17]



Figure 4: Inside One of HECToR's Cabinets [4]

2.5.2 Log Data

This project used data from actual (albeit anonymized) jobs run on HECToR from the start of 2012 to the start of 2013. Although HECToR contains 2816 computing nodes, only 2048 of the nodes at a time can be requested per job, which is the highest number of a job size reported in this project. Additionally, only those job sizes that have more than thirty instances in the log file are reported and the average MIND values used to calculate the improvement are normalised by the logarithm of the job size. Throughout this paper, job sizes are often discussed in terms of “small” jobs, “large” jobs and overall. “Small” jobs are defined as 1-10 nodes and “large” jobs are between 11-2048 nodes. “Overall” includes both of these groups. This distinction is made due to the average size of 10 for the job distribution, which is explained later in Section 4.1.

2.6 Previous Research on Cray XE6 Machines

Currently there is no optimization in place on HECToR with respect to job node scheduling. As such, when jobs are placed on the network, the scheduler selects nodes on a “first free” basis[20]. Carl Albing of Cray submitted a PhD thesis on a larger survey of this field last year (building on previous work) and as part of it created a simulator to test out different scheduling algorithms[1]. He ran two different algorithms and tested them out on three different machines’ data. One of these machines was *Hopper*, the supercomputer housed at NERSC, which is also a Cray XE6 [13]. The first part of this project extended Dr. Albing’s work by determining what similarities and differences there are from running these algorithms on data from the two machines: NERSC and HECToR.

3 Simulator Code

The original simulator code was written in Java and consisted of ten classes and one external library, which was reused and upgraded for this project. The codebase also liaises with a MySQL database containing mappings for the nodes on HECToR to their coordinate locations on the network. The simulator code was refactored to accommodate several new algorithms, whereby one of the original classes was removed and an additional six new classes added. Five unit tests were also written and added under a separate testing package with an additional external library to use `TestNG` functionality[19]. Bash scripts were created to run different variations of the simulations and to collect and analyze their results. The algorithm examples included in the sections to follow (for example, Figure 11), however, are written in pseudocode for better readability. All of the real code, however, has been submitted electronically along with the dissertation.

3.1 Original Code

The simulation starts in the main function in the `ALPSSim` class, which reads in a log file one line at a time and sends each placement (P for place and F for remove) as a `Reservation` to an `ALPSEngine` object. When the engine is first created, a particular job node scheduling algorithm must be specified, which is then used to place the jobs on the `Torus` network. This network object keeps track of the current simulated node mappings on the machine as specified in an SQL database. Three allocator classes are part of the original code base: one parent class (`NodeAllocator`) and two child classes (`FirstFitAllocator` and `FifoAllocator`). The parent class is abstract and the two children correspond to the algorithms described later in Section 4.2. A complete view of the structure of the simulation code can be seen in Figure 5.

As the structure may suggest, the simulator does not precisely mimic the actual scheduler; there are several key differences. The first is - although there is capability for such a setup - a simulation always starts out with no current jobs running on the machine. This means that jobs will all find perfect fits until the simulation “gets going” (ie. fills up with enough jobs that noticeable fragmentation would ordinarily be more likely to occur). The second difference is that there is no functionality built in to account for failure. That is, only jobs in which there was originally room to be placed will actually be simulated. This can be seen in the simulator code examples where the possibility of failure is not accounted for. Finally, the time a job takes to run is not taken into account with the simulator either - jobs are simply placed and removed in the order that they show up in in the log file. However, some residual of their run-time remains in the ordering, where jobs which run for longer are

removed further down in the log file.

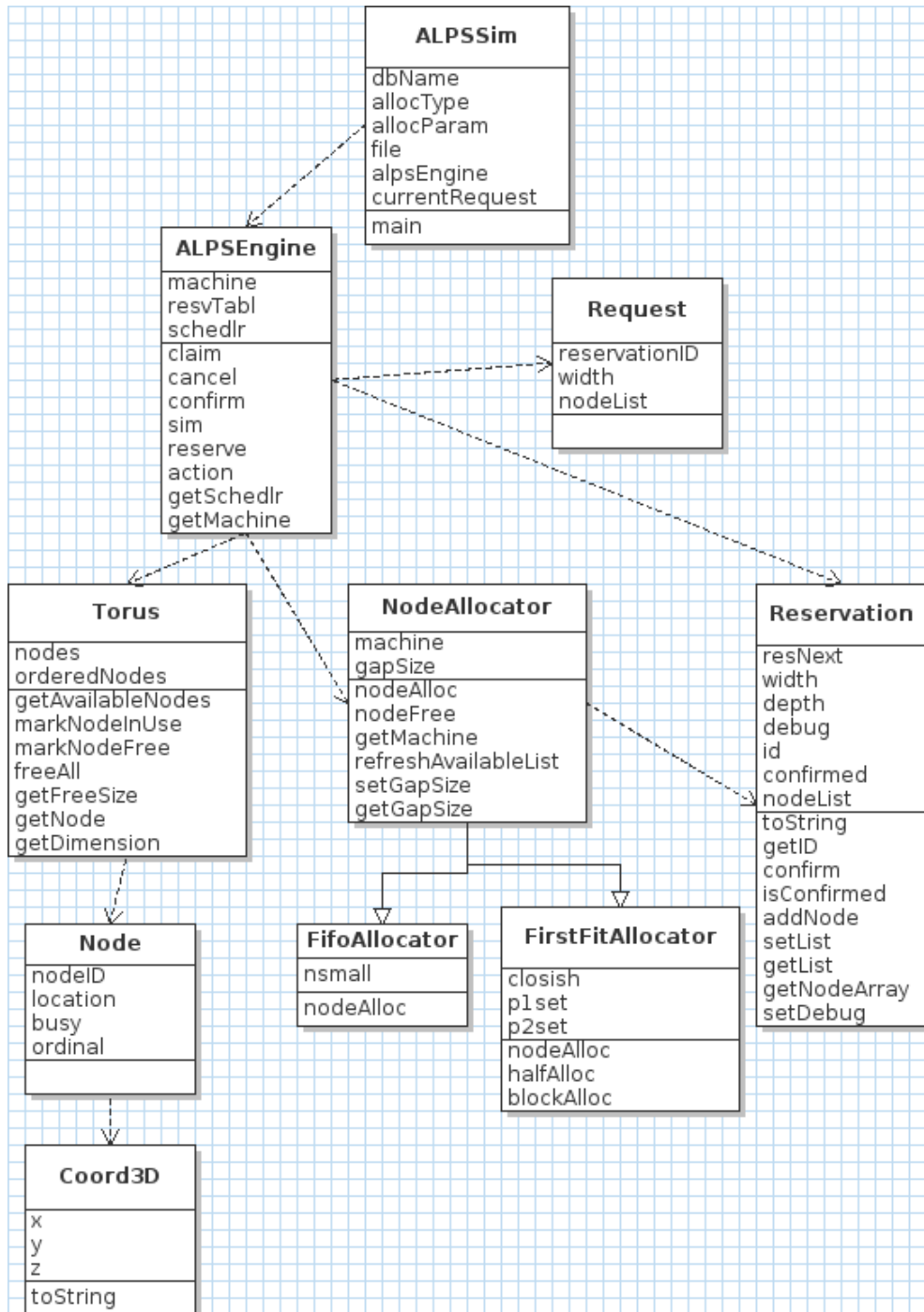


Figure 5: Original Simulation Code UML Diagram

3.2 Updated Code

The main purpose of updating the simulator code was to implement new job allocation algorithms, in particular those outlined in Section 5. This was done mainly with the addi-

tion of new child classes to inherit from the `NodeAllocator` class, however this class and the `FirstFitAllocator` class were refactored first to make adding new functionality easier. Additionally, an off-by-one error was discovered in the original algorithm in the `FirstFitAllocator` class, so this function was rewritten. Although only three new classes were added, there are actually four new algorithms. The `FirstFitAllocatorRevised` accommodates two algorithms as well as being a parent class to a third. The main changes in this new structure are shown in Figure 6.

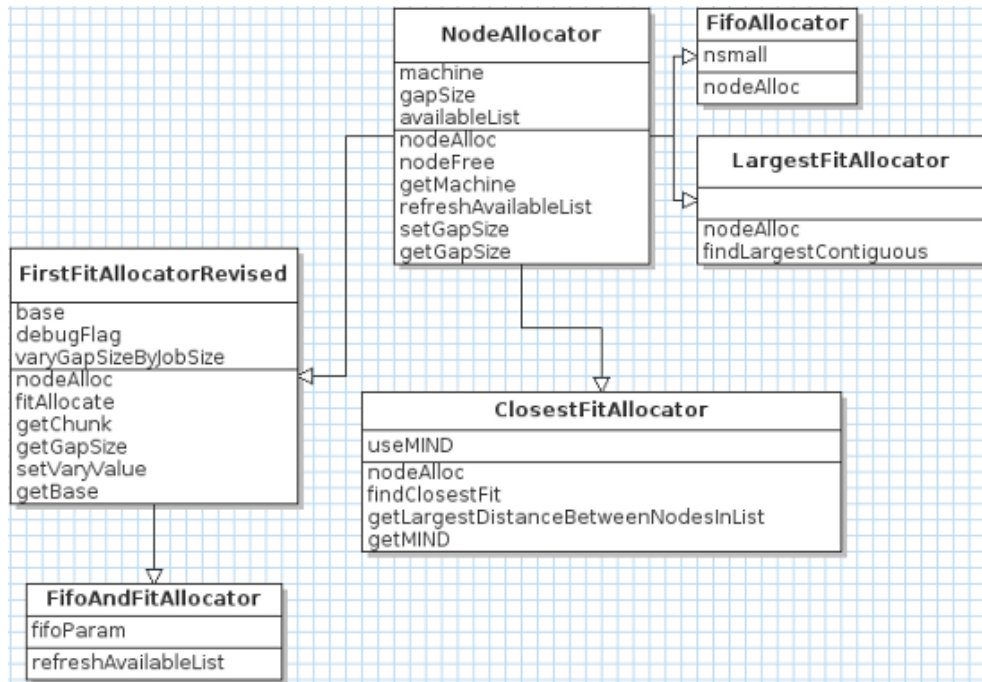


Figure 6: Updated Simulation Code UML Diagram

3.3 Testing

Finally, a new testing package was added to ensure that the algorithms were producing the correct results and would continue to do so where changes were made elsewhere in the codebase. The package includes unit tests that inherit functionality from the open source `TestNG` library. At the top of the hierarchy there is an abstract parent, `CommonTest`, which sets up the database connection for use by the child classes and cleans up after each of their tests are run. Then four other tests were created to test the functionality of the four new classes added. Another class, called `NodeDebugHarness`, containing static functions was also added to allow for shorter available nodelists to be used as well as printing out available nodes for debugging purposes. The new testing package structure can be seen in Figure 7.

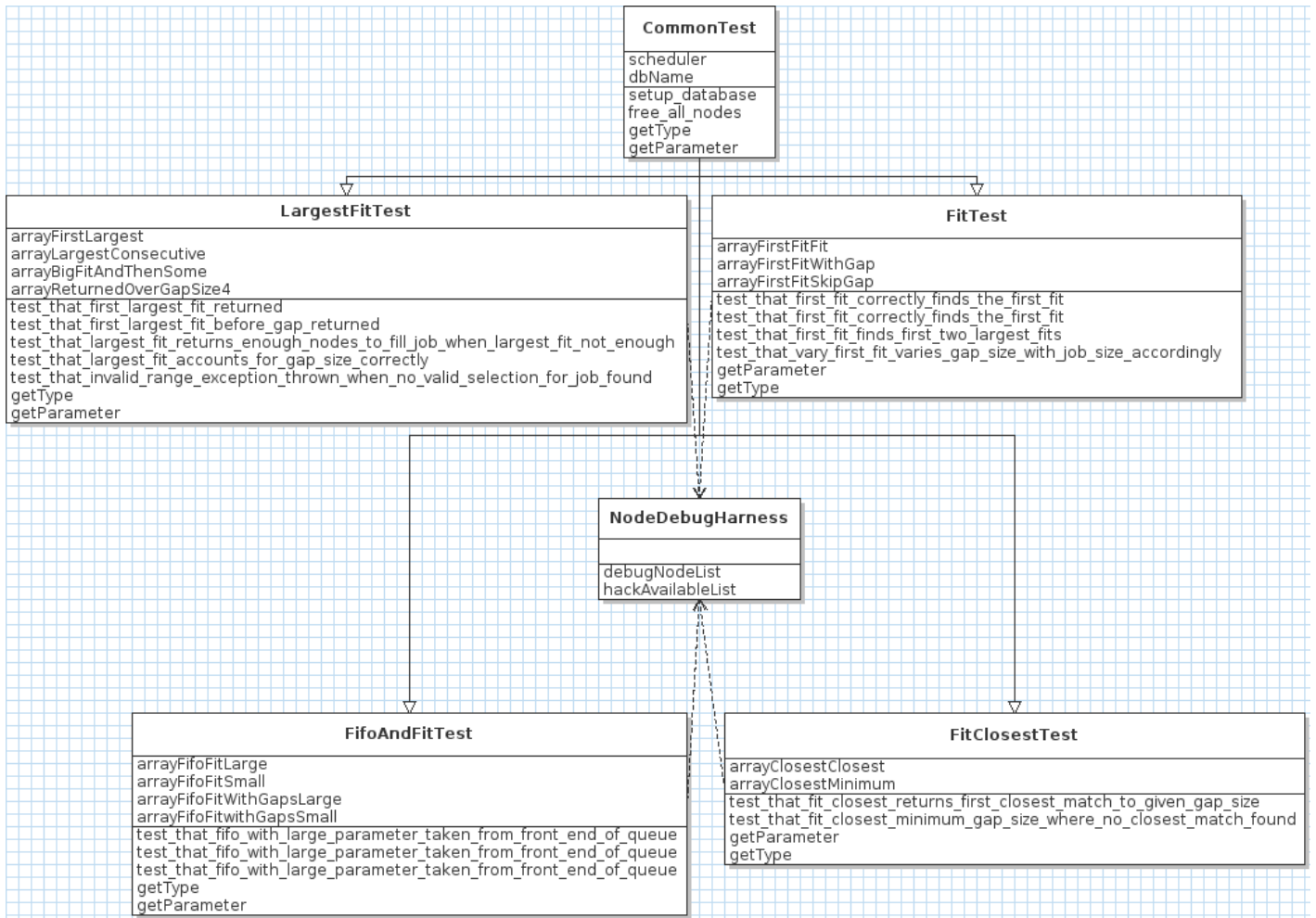


Figure 7: Simulation Test Package UML Diagram

4 Comparison of Job Node Allocation Algorithms on Cray XE6 Machines

The first part of this project involved reproducing the research done by Carl Albing on job node allocations for the NERSC machine, another Cray XE6. The purpose of doing this was two-fold: to confirm his results and to see what role, if any, the job size distribution had on job node allocation algorithms. First, the differences between the jobs run on both machines are explored. Then the preliminary algorithms are introduced and explained. Finally, the results and conclusions from the comparison of the two is explored.

4.1 Job Size Distributions

As can be seen in Figures 8 and 9, the job sizes of the two Cray XE6 machines (HECToR and NERSC) follow somewhat different distributions. Both distributions peak early and have long, small tails. However, HECToR has a second peak at 9-16 nodes, where there are nearly as many jobs run at 9-16 nodes as at 1 node. On the NERSC machine, no other job size range contain close to as many jobs as 1 node. The mean job size for each the two distributions confirms this difference: the mean job size of NERSC is 3 and for HECToR is ~ 10 .

While both HECToR and NERSC are Cray XE6 machines, they have a different number of processors per node. As the x-axes on the graphs in Figures 8 and 9 show, each node on the NERSC machine contains twenty-four processors while each node on the HECToR machine contains thirty-two. This should be kept in mind when viewing subsequent comparisons of improvement seen on the two machines as they will not show an exact match. However, rebinning both values proved awkward and showing both on a consolidated graph seemed confusing, so they are simply graphed on the same axis for clarity. Furthermore, there are roughly four times as many jobs in the logs used from HECToR as there are for NERSC (probably due to the fact that the data logs from the NERSC machine cover only 43 days, while the HECToR logs span a whole year). This difference can be seen along the y-axes in the same figures.

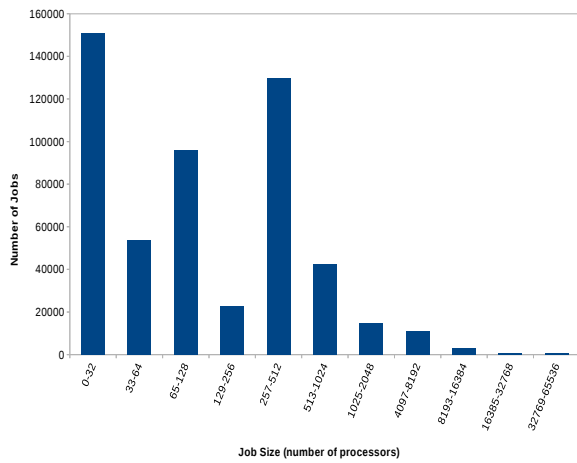


Figure 8: Number of Jobs versus Job Size for Jobs on HECToR

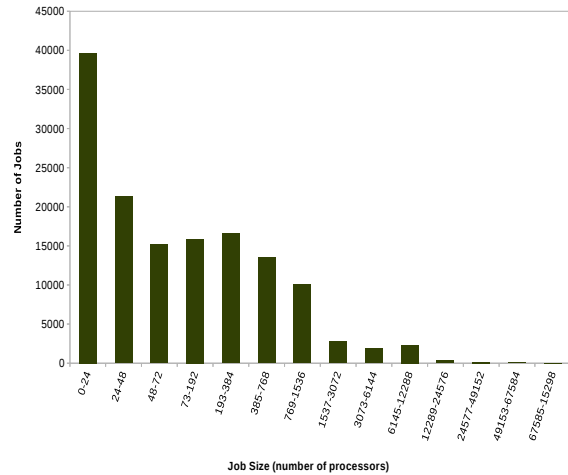


Figure 9: Number of Jobs versus Job Size for Jobs on NERSC

4.2 Preliminary Dynamic Allocation Algorithms

Two dynamic allocation algorithms were originally implemented and run on the NERSC data: `First Fit` and `FIFO[1]`. The `First Fit` algorithm takes in an acceptable gap size as a parameter and iterates over the free nodes on the network searching for a consecutive chunk of available nodes the size of the current job (where consecutive is defined by the gap size parameter). Where a consecutive chunk of free nodes cannot be found, the algorithm recursively divides and re-runs over two evenly divided smaller chunks until enough nodes have been allocated to the job. The `FIFO` algorithm takes in an input parameter specifying a “large” job size and returns the first free nodes it finds from one end of the free node list if the job size is “large” and from the other end if it is small. The nodes returned are not necessarily consecutive.

The following results for these preliminary (and subsequent) algorithms show improvement between jobs run on a particular algorithm configuration versus the jobs run on the `FIFO-0` algorithm. The new algorithm results are compared to the `FIFO-0` algorithm and not the original mapping because there were already jobs occupying nodes when the logs were collected, as explained in Section 3.1. In contrast, the `FIFO-0` algorithm is run with the same initial configuration as the new algorithms and simply selects a mapping from the first available nodes it finds, as the job scheduler of HECToR does. Finally, an off-by-one error was discovered in the original `First Fit` algorithm, so these results were created

using the corrected algorithm. The differences between the original algorithm and the updated one can be seen below in Figure 10. The updated algorithm simply returns slightly better results.

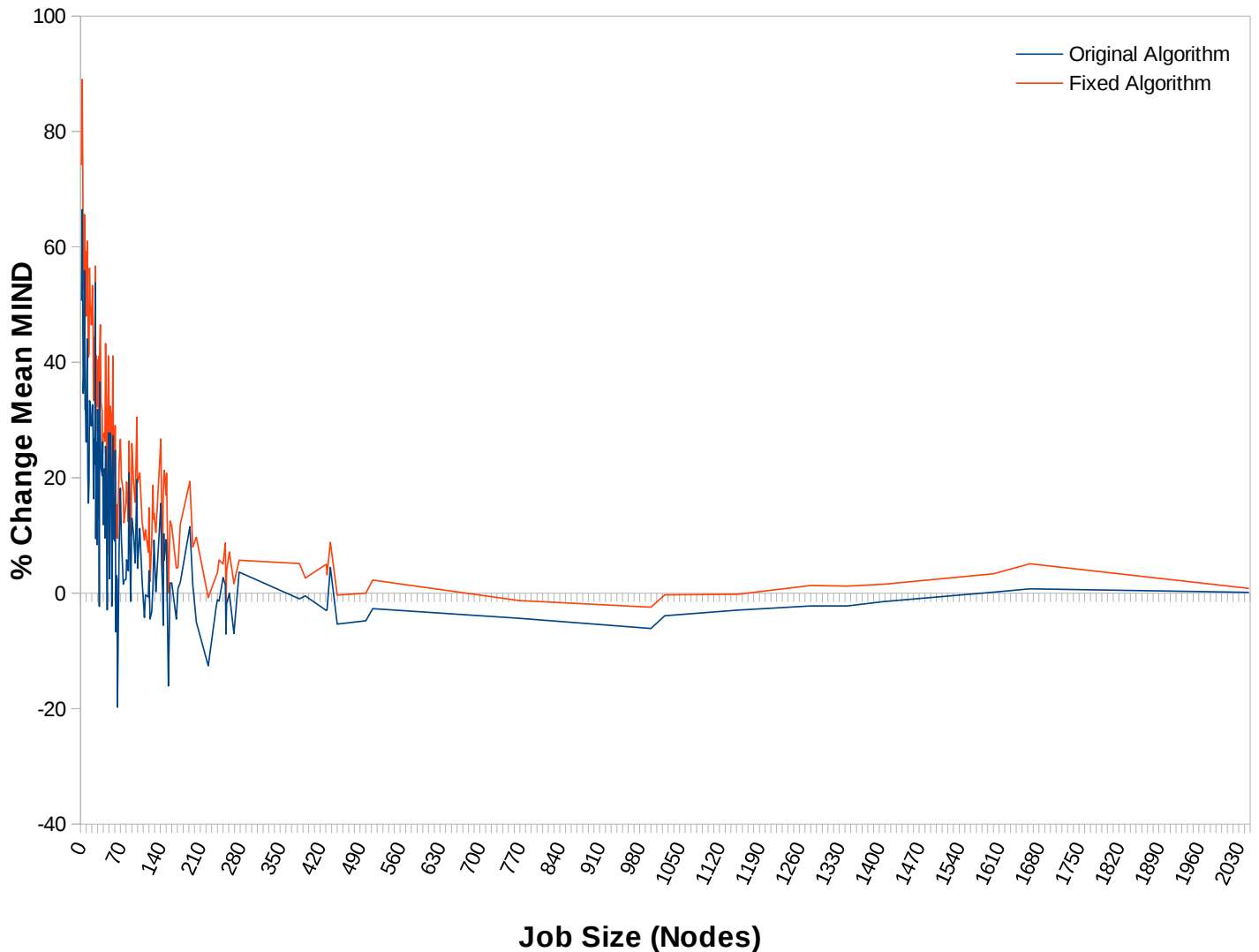


Figure 10: Improvement versus Job Size for Original and Fixed Fit Algorithms Run on HECToR Log Data

4.2.1 First Fit Results

The simulator ran the `First Fit` algorithm on the NERSC and HECToR data sets using gap sizes of 1, 2, 4 and 8. The results for the improvement using gap sizes of 1 and 8 are shown in Figures 12 and 13 respectively. In comparison to the NERSC data, the `First Fit` algorithm performed similarly (although slightly better) for smaller job sizes and generally better for larger job sizes for the HECToR data. The larger the gap size, the more positive the results were overall for both data sets. This improvement can be

seen clearly between Figures 12 and 13 for gap sizes 1 and 8. The HECToR log data also responded more positively to the algorithm overall. While both data sets oscillate between positive and negative improvement in $\text{Fit}-1$, the magnitude of the negative values for the HECToR data set is smaller than those for the NERSC. The magnitude of the positive values for the HECToR data is also notably greater for larger job sizes and for both smaller and larger gap sizes. Overall, the $\text{Fit}-8$ algorithm produced the best results, although between the four algorithm runs the average improvement only varied by $\sim\%1.5$.

First Fit Algorithm

```
function fitAllocate

    initialize returnNodes to empty

    call getChunk

    if available chunk is not empty
        add all nodes in chunk to returnNodes
        mark these nodes as in use
    else
        split the jobSize in half
        call fitAllocate again with first half
        add all nodes returned to returnNodes
        mark all nodes returned as in use
        call fitAllocate again with second half
        add all nodes returned to returnNodes
        mark all nodes returned as in use

    return returnNodes

function getChunk

    refresh available nodes on network
    initialize returnNodes list to empty

    while there are available nodes left to traverse
        if the returnNodes list is empty
            add the current node
        else if the returnNodes list is greater than or equal to the job size
            break out of the while loop
        else
            if the space between the current node and the previous node is less
                than or equal to the gap size
                add the current node to returnNodes
            else
                clear the returnNodes list and add the current node to the
                list

    if the returnNodes list is less than the job size
        return null

    return the returnNodes list
```

Figure 11: Functions to Allocate Nodes and Get Suitable a Chunk Size For First Fit Algorithm

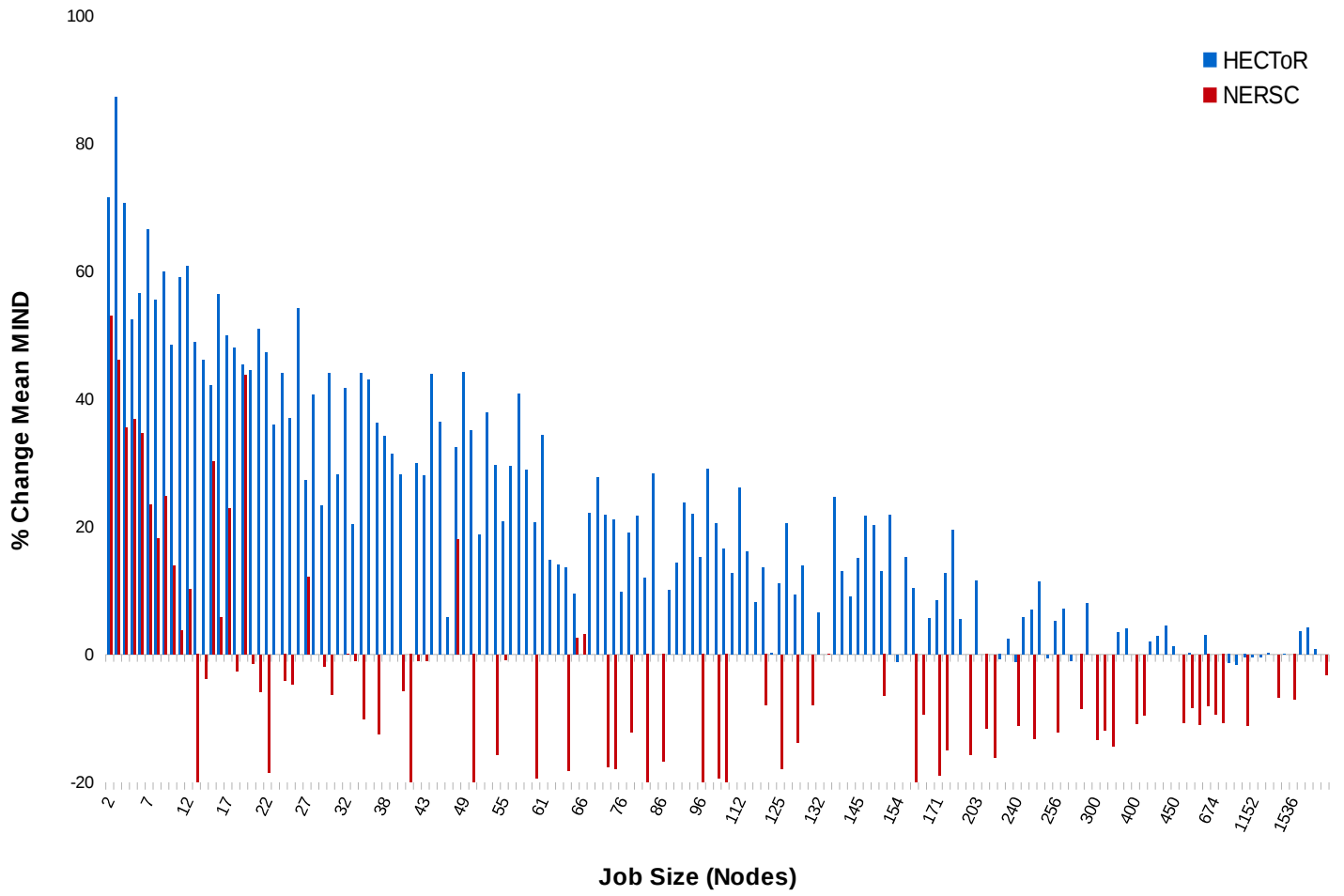


Figure 12: Improvement versus Job Size for Fit-1 Algorithm on HECToR and NERSC Data^a

^aData for all graphs has been submitted electronically

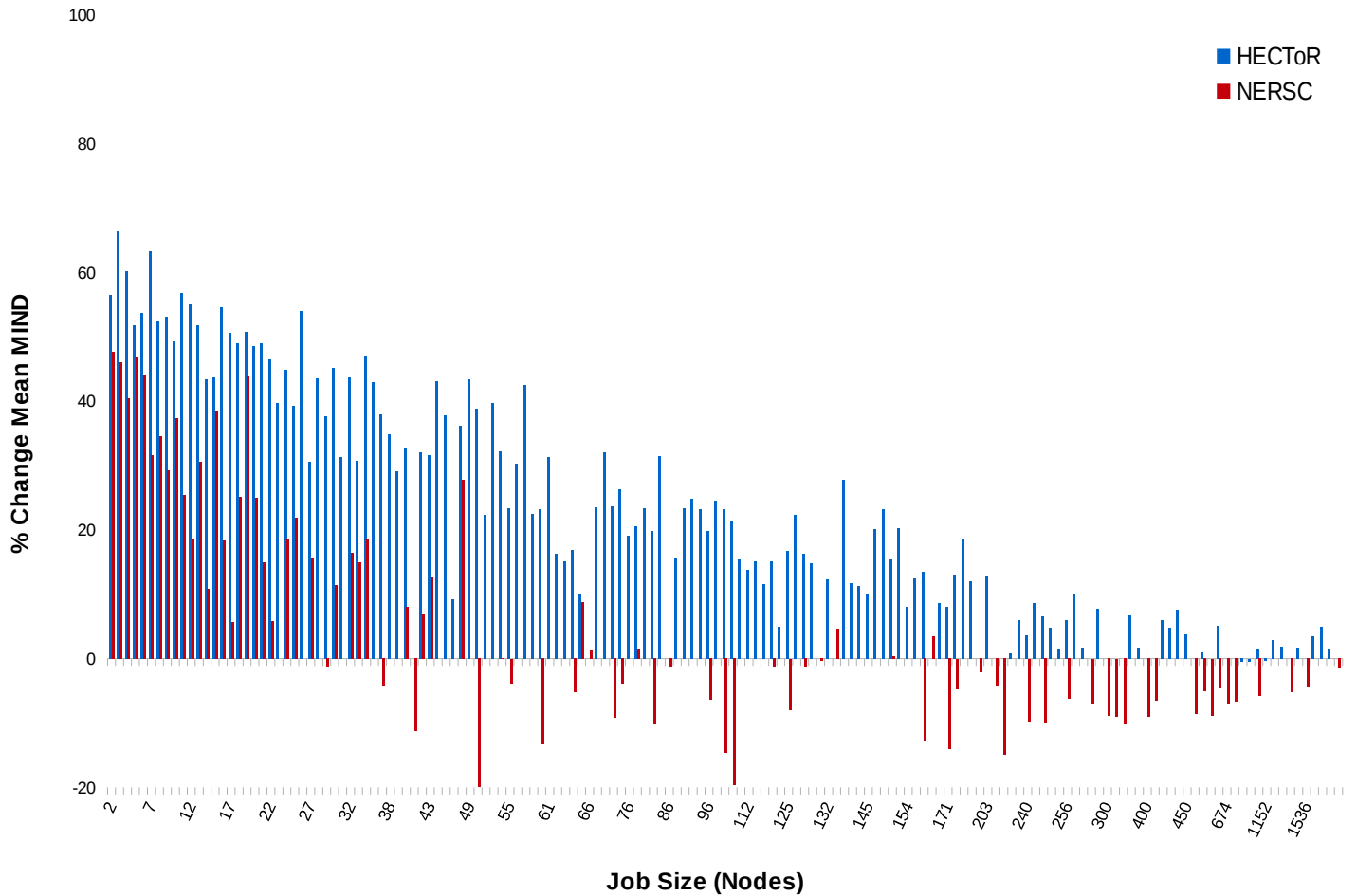


Figure 13: Improvement versus Job Size for Fit-8 Algorithm on HECToR and NERSC Data

4.2.2 FIFO Results

The simulator then ran the FIFO algorithm on both data sets using “large” parameter values of 2, 4, 8, 16, 32, 64 and 128. The results using parameter values of 2, 32 and 128 are shown in Figures 15, 16 and 17 respectively. Overall the NERSC data set responded more positively to the FIFO algorithm for smaller job sizes, while the HECToR data set responded more positively for larger job sizes. However, the magnitude of the positive response to the algorithm was much less than for the First Fit algorithm. As the size of the parameter value increased, the more positively both data sets responded for larger job sizes and the poorer the response was for smaller job sizes. In fact, the difference between mixed and positive responses can be seen at the point where $job\ size = parameter\ size$. Additionally, as can be seen most clearly in Figure 17, the percent improvement for small job sizes oscillates between positive and negative magnitudes at greater magnitudes for the HECToR data set while still responding mostly positively for the NERSC data set. Overall, the FIFO-32 algorithm produced the best results, however, unlike the Fit algorithm, the

difference between the average improvement for the worst and best runs was more varied (by $\sim 11\%$).

FIFO Algorithm

```
initialize returnNodes list to empty

get available list of nodes

if the job size is smaller than the ``large`` parameter
    reverse the list of available nodes

while there are nodes in the available nodes list
    add current node
    if the size of returnNodes is greater than or equal to the job size
        break out of the while loop

return the returnNodes list
```

Figure 14: FIFO Algorithm^a

^aThis is Carl Albing's algorithm

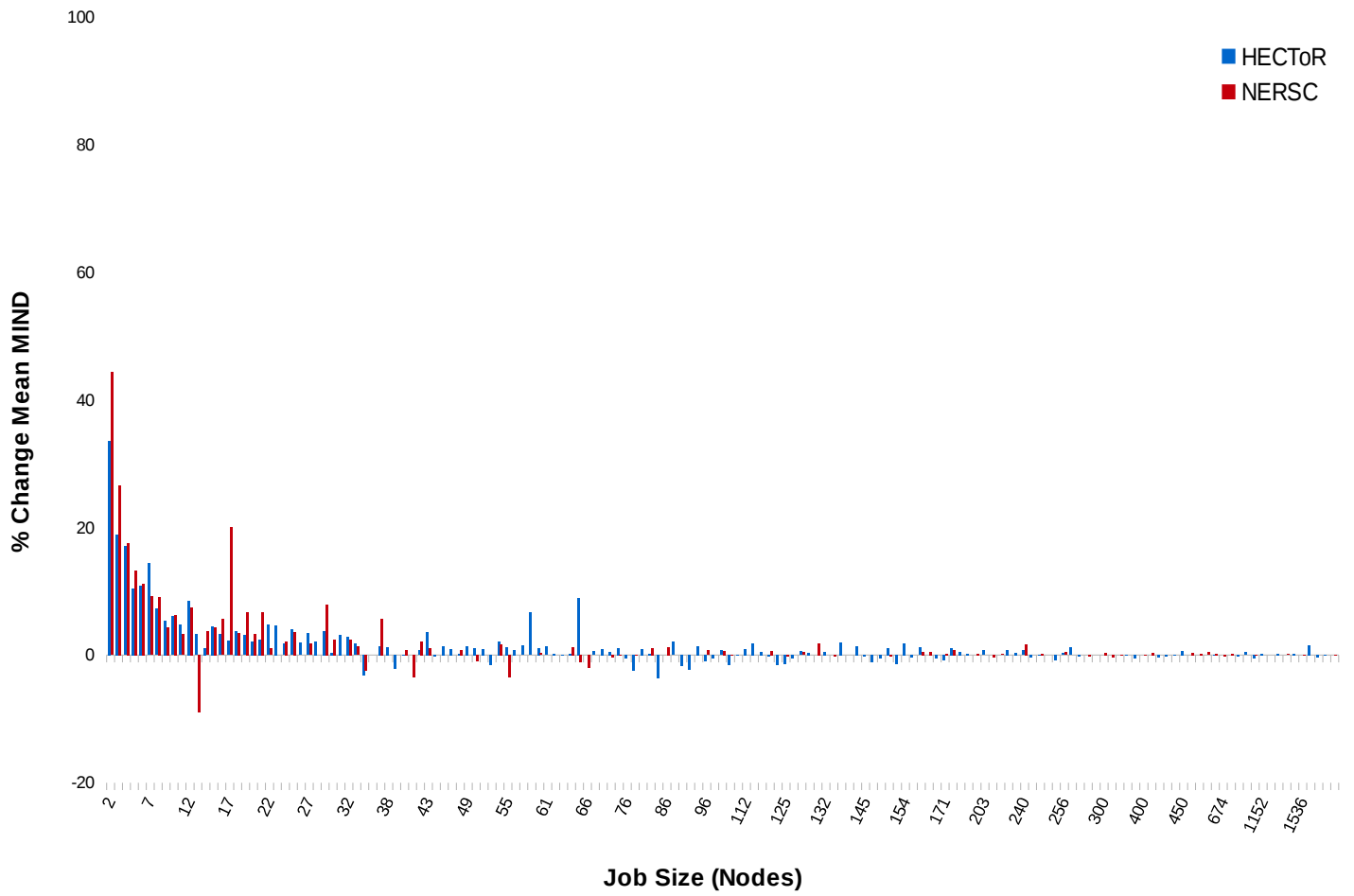


Figure 15: Improvement versus Job Size for FIFO-2 Algorithm on HECTOR and NERSC Data

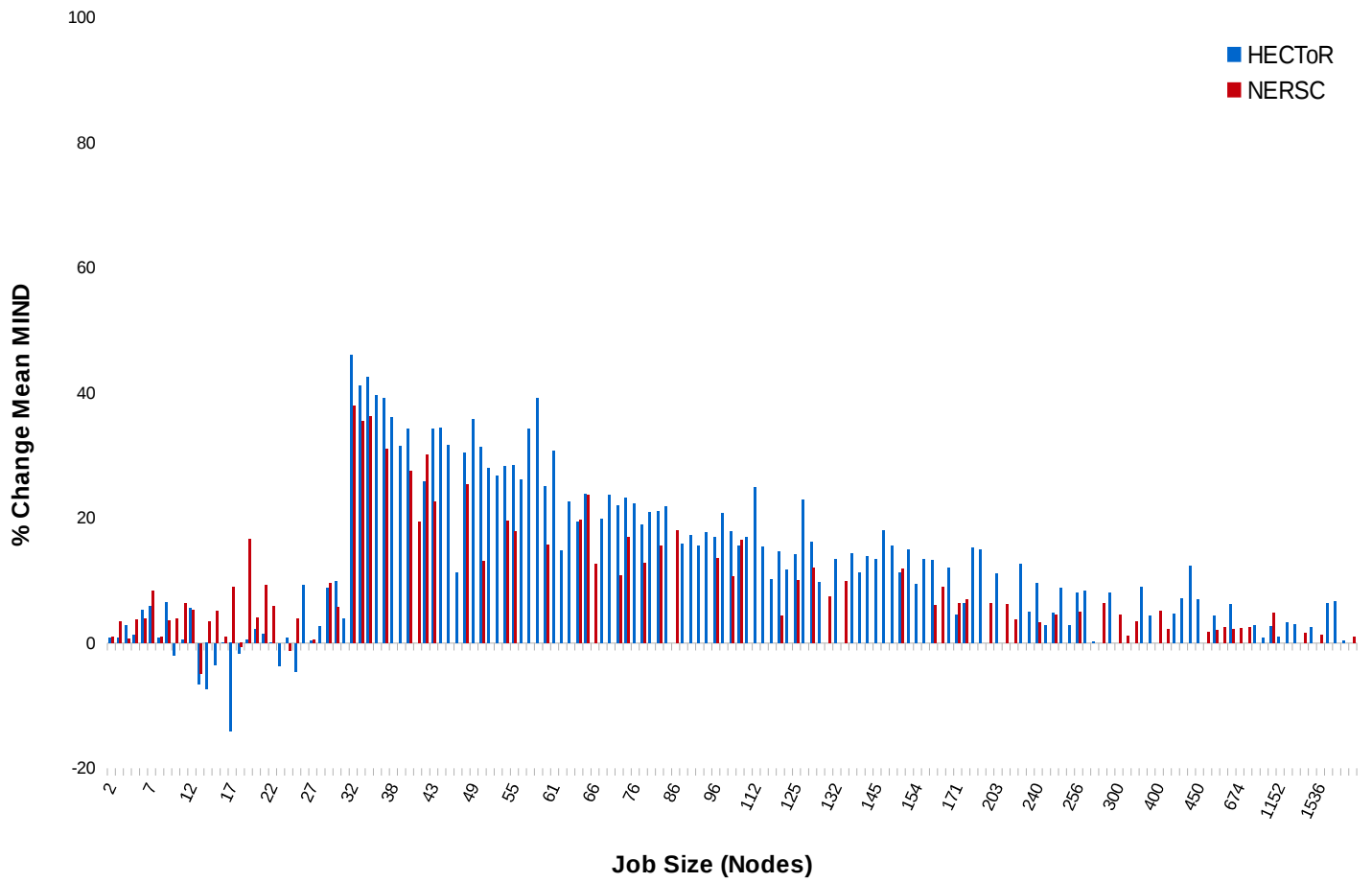


Figure 16: Improvement versus Job Size for FIFO-32 Algorithm on HECToR and NERSC Data

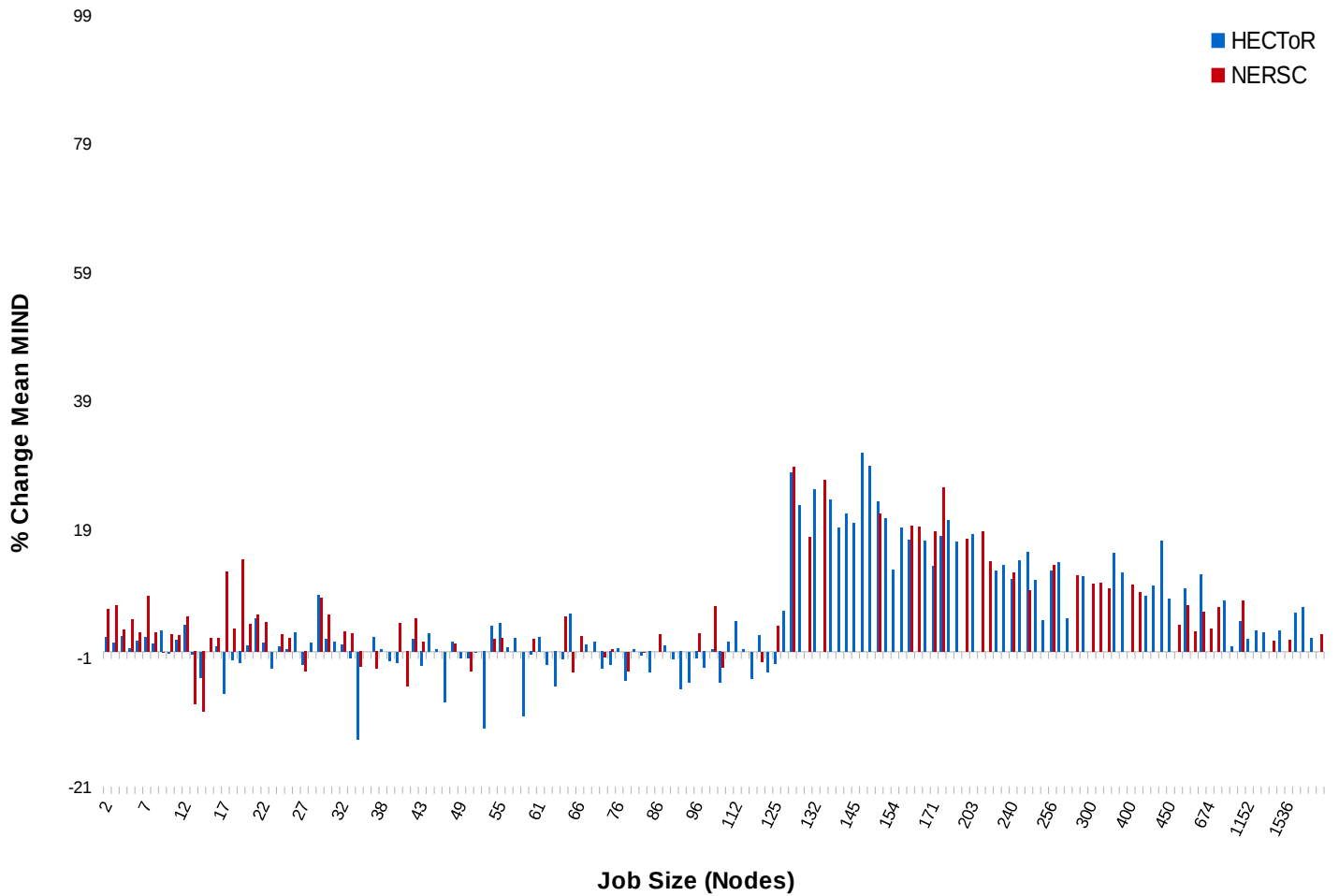


Figure 17: Improvement versus Job Size for FIFO-128 Algorithm on HECToR and NERSC Data

4.2.3 General Discussion of Similarities and Differences

The original results from running the two algorithms on the NERSC data set showed that the FIFO algorithm worked better for larger job sizes, while the First Fit algorithm worked better for smaller ones. The First Fit algorithm with a gap size parameter of 4 showed the best improvement results overall[1]. Generally, the results from running the HECToR data set through the algorithms confirmed this behaviour as well. However for the HECToR data, the First Fit algorithm with a gap of 8 produced the highest percent improvement on average across all jobs.

Although for the most part the two data sets responded similarly to the algorithms, a number of noticeable differences also cropped up. In particular, the NERSC data set was more sensitive to the First Fit algorithm and overall did not respond as well to the algorithm as the HECToR data set. Conversely, the HECToR data set was more sensitive to the larger parameter valued FIFO algorithms, oscillating more markedly between positive

and negative percentage change mean MIND values at lower job sizes. Both of these features may be due to the differences in the distribution of job sizes found in the two data sets. The various job sizes in the HECToR data set are more evenly spread out across a wider range, whereas the job sizes in the NERSC data set peak early and taper off more quickly, resulting in a smaller number of larger sized jobs. The average job size of the distributions, as described in Section 4.1, confirms this given that the value for the NERSC data set is much lower than for the HECToR data set. Because relatively there are a larger number of small job sizes in the NERSC distribution, it may always be more difficult to find contiguous chunks for larger job sizes unless the smaller jobs are placed closer together. This would explain why the FIFO algorithm works better for the smaller jobs sizes with larger parameters for the NERSC data and worse for the HECToR data set. Because the HECToR data set has a wider range of commonly used job sizes, the FIFO algorithm might show some improvement, but not in any one specific area. Additionally, the HECToR data set may respond better to the First Fit algorithm overall because there is more room for improvement with fewer small jobs and more larger job sizes which might have normally become much more fragmented.

There are other factors that may explain the differences as well. While the two data sets were run on similar architectures, the scheduling algorithms may be different on these two machines. The machine at NERSC may have different maximum times which jobs are allowed to be run so may have a different turnover altogether. This would show up in the logs as jobs being placed and removed more quickly than they might be in the HECToR logs. It is also possible that the scheduler is set up to wait until contiguous chunks of free nodes are available to run large jobs. Finally, it should be taken into consideration that the nodes do not have the same number of processors, as described in Section 4.1, so indeed the graphs would never be an exact match.

The main conclusion to be drawn from this preliminary comparison is that job node scheduling algorithms have different effects depending on job size distributions. In particular, this holds where the architecture and topologies are virtually the same. Therefore, optimisations using such algorithms would need to be done on a per-machine basis. Here, having a simulation could be quite useful because results can be obtained and decisions made about topologies and scheduling before the machine is even built using historic data indicative of the job distribution of the service.

5 New Dynamic Allocation Algorithms

Although the two preliminary algorithms separately showed some promise for large and small-sized jobs, neither of them worked well overall for both groups. Four new algorithms were designed and implemented to run over the HECToR data set to try to bridge this gap. These include:

- a “largest fit” algorithm
- a `First Fit` algorithm varying gap size by job size
- a combination of the `FIFO` and `First Fit` algorithms
- a “closest” fit algorithm

Some other variations of these algorithms were also tried and their results are documented where applicable. As original data for the NERSC set was not available for comparison, the following graphs only show the percent improvement per job in comparison to the `FIFO-0` algorithm run on the HECToR data.

5.1 Largest Fit

5.1.1 Algorithm Concept

The `Largest Fit` algorithm loops over the list of free nodes on the network looking for a contiguous chunk that will fit the job size. An input parameter of acceptable gap size must also be specified, which defines contiguousness. Where an available chunk cannot be found of the job size, the algorithm selects the largest free chunk available and then continues the search. Similar to the `First Fit` algorithm, the algorithm iterates again seeking out the “largest fit” for the remainder (ie. *job size - largest size*) until the complete number of nodes has been found. The algorithm can be seen in Figure 18.

Largest Fit Algorithm

```
function allocateNodes

    initialize returnNodes list to empty
    initialize remainingNeeded to job size

    while the size of remainingNeeded is greater than zero

        refresh the list of available free nodes in the network
        call getLargestContiguous to get largest contiguous chunk of free nodes
            with the parameter remainingNeeded
        add returned nodes to returnNodes list
        mark all returned nodes as in use
        subtract off number of returned nodes from remainingNeeded

function getLargestContiguous

    initialize nodeList to empty
    initialize maxList to empty

    if the number of available nodes is greater than the remainingNeeded size

        while there are available nodes left

            if the nodeList list is empty
                add the current node
            else if the nodeList is greater than or equal to the
                remainingNeeded size
                return the nodeList list
            else
                if the space between the current node and the previous node
                    is less than or equal to the gap size
                    add the current node to nodeList
                else
                    if the nodeList list is greater than the maxList
                        set the maxList equal to the nodeList list
                    clear the nodeList list

            if the nodeList list is greater than the maxList
                set the maxList equal to the nodeList list

    else if the number of available nodes is equal to the remainingNeeded size
        return all the available nodes

    return the maxList
```

Figure 18: Functions to Allocate Nodes and Get a Suitable Chunk Size For Largest Fit Algorithm

5.1.2 Results

The simulator ran the Largest Fit algorithm on the HECToR log data using gap size values of: 1, 2, 4 and 8. Of the four algorithms run, the Largest Fit algorithm produced the second highest results overall, for large job sizes and for small job sizes. For overall average and large job size average, the Largest Fit with a gap size of eight performed best, where the difference between the Largest Fit algorithm and the highest value (from the Closest Fit algorithm, which performed the best in all the categories) was $\sim 2\%$ improvement for the overall average and $\sim 1\%$ improvement for the large job sizes. For the small job size average, the Largest Fit with a gap size of one performed best, where the difference between the results from the Largest Fit algorithm and the best performing algorithm (again the Closest Fit algorithm) was $\sim 2\%$ improvement. Between the smallest and largest gaps, the difference was only $\sim 1\%$ improvement for the large and overall averages. However for the small job average, the difference between the averages between gap sizes of one and eight was almost 10% improvement in favor of the gap size of one. These differences can be seen in Figure 19.

This behaviour is similar to the results seen with the First Fit algorithm, although the Largest Fit algorithm produced better results overall. Between the smallest and largest gaps, the difference for both algorithms was only $\sim 1-2\%$ improvement for the large and overall averages. For the small job average, the difference between the averages between gap sizes of one and eight was $\sim 7-10\%$ improvement in favor of the gap size of one. This makes sense given that most small jobs are less than 8 nodes in size, so using a similar gap size would not be likely to show much improvement. However, for average and larger than average jobs, a gap size of eight matters less and is more flexible to allow for a compact placement. For all job sizes though, the Largest Fit algorithm performed markedly better in improvement in comparison to the First Fit algorithm. This is probably because the Largest Fit algorithm optimizes the amount of the chunk that is contiguous more than the First Fit ever would. However, both only seek out a contiguous chunk on the first iteration and on subsequent tries looks for smaller contiguous chunk(s) that can be anywhere on the network. Inevitably though, the Largest Fit algorithm is more likely to require fewer iterations as it looks for the largest chunk available. Indeed, the First Fit algorithm runs ~ 2 seconds slower than the Largest Fit one as can be seen in Table 3.

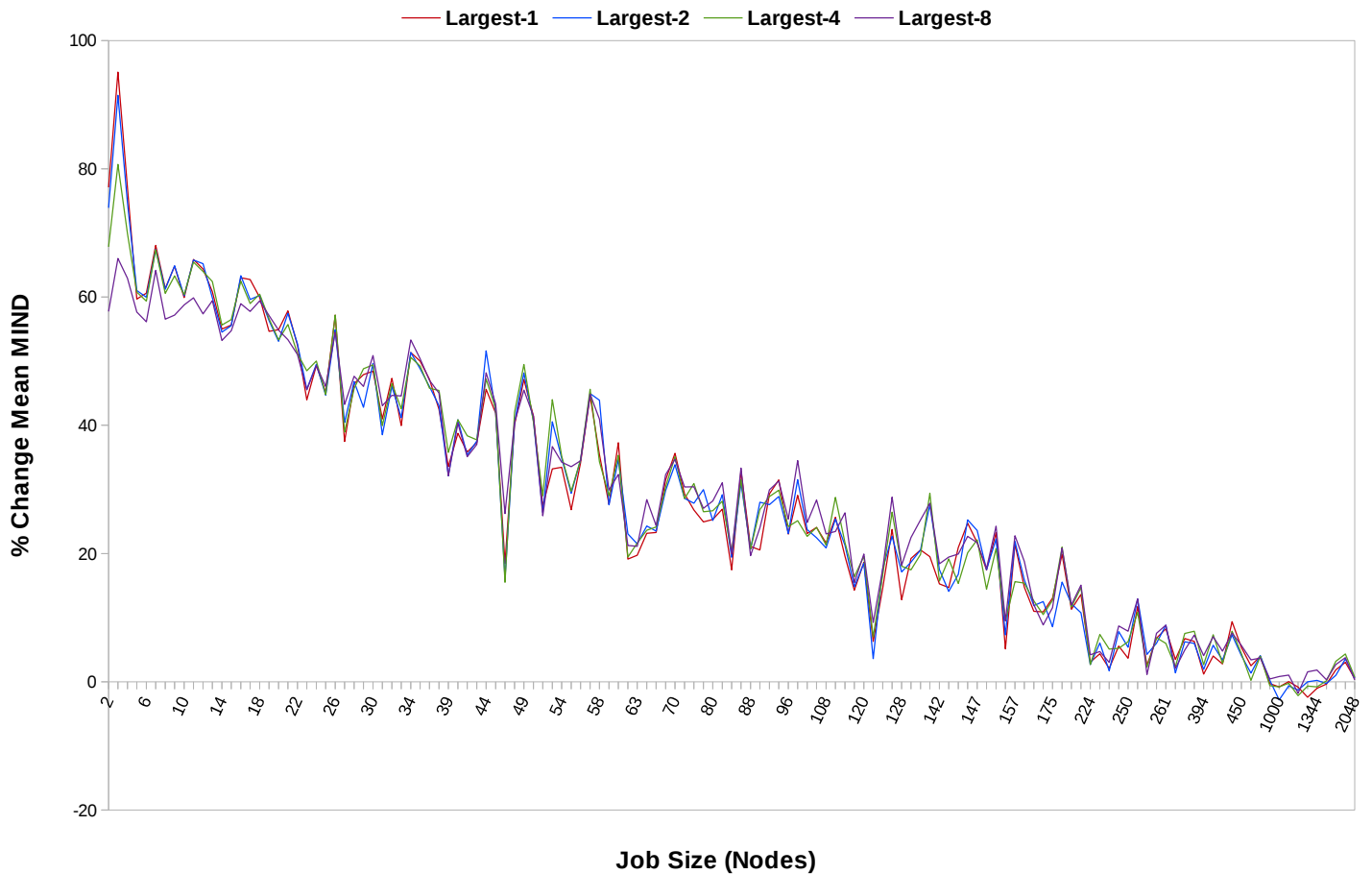


Figure 19: Improvement versus Job Size for Largest Fit Algorithm

5.2 FifoFit

5.2.1 Algorithm Concept

The `FifoFit` algorithm combines the two algorithms `FIFO` and `First Fit`. It selects free nodes using the `First Fit` algorithm from the beginning of the free nodes list if the job size is “large” and from the opposite end if the job size is small. Here two parameters are involved: one for varying the fit gap size and one for determining what constitutes a “large” value for the `FIFO` algorithm. Code-wise, the algorithm itself simply inherits from the same class as the `First Fit` algorithm and overrides the function that obtains the free node list with one that follows the same procedure as the `FIFO` algorithm. As such, snippets of the code are not included here.

5.2.2 Results

The simulator ran the `FifoFit` algorithm on the HECToR data set using “large” parameter values (for the `FIFO` algorithm) of [2, 4, 8, 16, 32, 64, 128] and gap sizes (for the `First Fit` algorithm) of [1, 2, 4, 8], resulting in twenty-eight different combinations of parameters. The improvement seen from these various runs were mixed, although the difference in the overall average improvement only differed by $\sim 4\%$ improvement between the poorest performing algorithm (`FifoFit8-1`) and highest performing one (`FifoFit64-8`). All algorithms saw improvement overall compared with the `FIFO` algorithm and some saw improvement over the `First Fit` algorithm. Most saw improvement over the `Varying Fit` algorithm as well (in particular, all those with a `Fit` gap size of eight). `FifoFit64-8` performed quite well overall and for high average values, coming in just after the `Closest Fit` and `Largest Fit` algorithms. For the small job averages, the `FifoFit32-1` and `FifoFit4-1` performed only $\sim 0.5\%$ improvement worse than the `Largest Fit` with a `Fit` gap size of four and better than the `Largest Fit` with a `Fit` gap size of eight. The `FIFO` algorithm with the best improvement overall was with a gap size of thirty-two and the `FIFO` algorithm with the most improvement for small jobs was with a gap size of four. As explained in Section 5.1.2, the `First Fit` algorithm with the most improvement for small jobs was with a `Fit` gap size of one.

As can be extrapolated from the examples above, overall the results seen in the combination of the two algorithms corroborate with the results seen separately. That is, the parameters that worked well for particular job types for each individual algorithm worked better in combination. As well as this, those parameters that worked well for small or large job sizes still worked well in combination. Generally, runs with a `Fit` gap size of eight performed well overall and for large jobs, and those with a `Fit` gap size of one performed well for small jobs (as seen for both on their own). Relatedly, runs with higher `FIFO` parameter values worked better for larger job sizes and overall, while the smaller parameter values worked better for smaller job sizes. See Figure 32 for the complete picture.

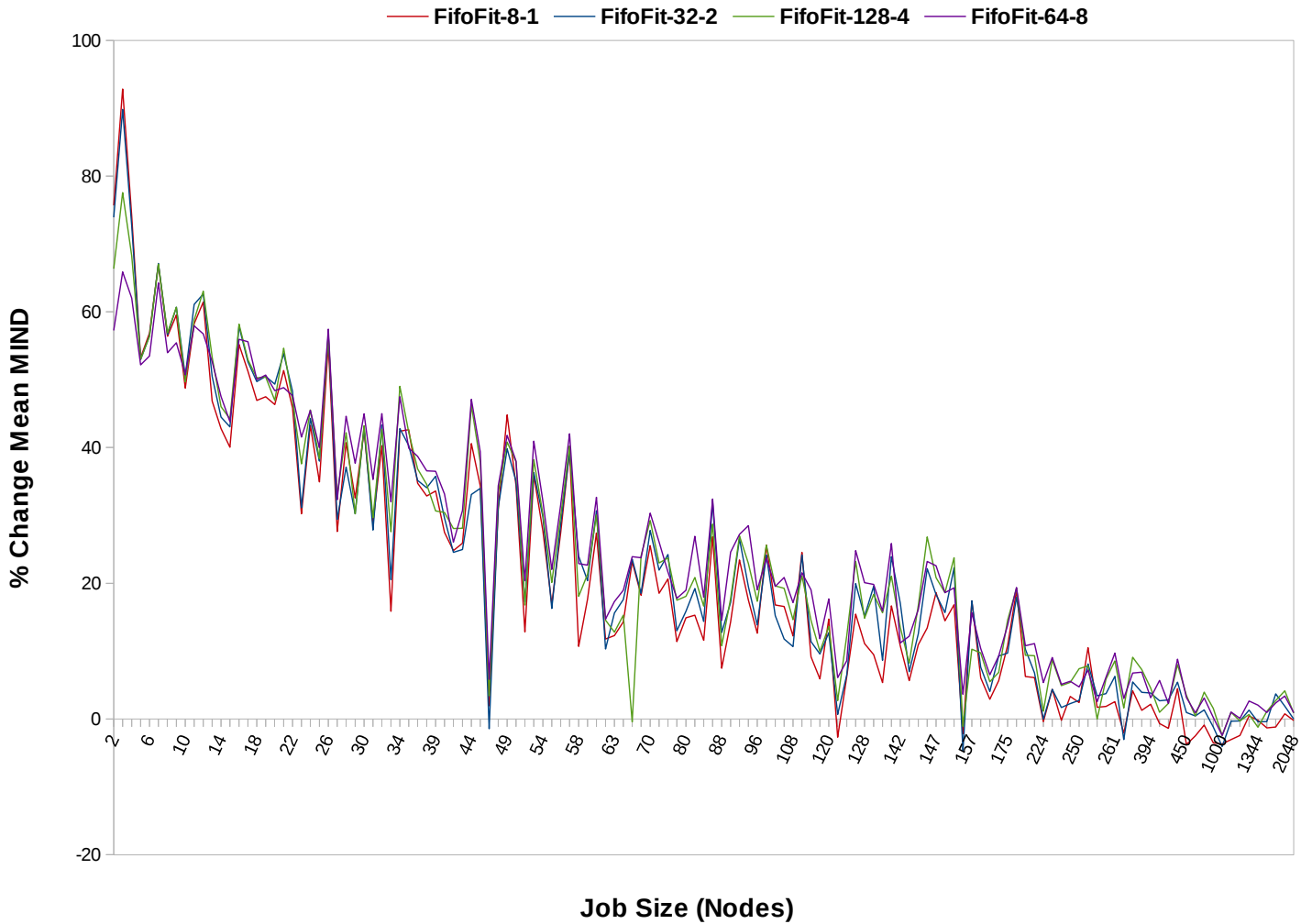


Figure 20: Improvement versus Job Size for Select FifoFit Algorithms

5.3 Varying Fit

5.3.1 Algorithm Concept

The `Varying Fit` algorithm varies the acceptable gap size for a contiguous chunk of free nodes by job size. A log function was used in order to gradually increase the gap size from small jobs up to large ones. Originally, an input parameter for gap size was used as well, however testing out several variations of the algorithm with such a parameter showed that this did not provide any additionally useful means of tuning. Furthermore, the results followed the same trend as the original `First Fit` algorithm. In multiplying or dividing by the gap size in various incarnations, the resulting values did not change by more than %1 in improvement. While this is also true for the `Largest Fit` algorithm, in this algorithm the variable parameter seemed to detract from the nature of the algorithm itself so in the end was not used. The algorithm effectively uses the same method as the `First Fit`

algorithm, while overriding the value of the gap size as shown in the code in Figure 21. The factor of two is used to keep very small jobs (size 2-3) from having gaps in them and the gaps range from a size of 1 (for job sizes of 2-3) up to 8 (for jobs over 1000 nodes).

Varying Fit Algorithm

```
function getGapSize
    if varying gap size by job size
        return the integer value of the natural logarithm of the job size
            multiplied by 2
    else
        return the normal gap size
```

Figure 21: Function to Get Gap Size in FirstFitRevised Class

5.3.2 Results

The `Varying` algorithm was the worst performing algorithm of the four new algorithms, although it still showed more consistent positive improvement than the `FIFO` algorithm and results fell between the best and worst performing `First Fit` algorithms. It seems a little surprising that it does not perform better than any of the static `First Fit` algorithms. However, perhaps it is a case of tailoring the algorithm to work well with a particular job distribution. Given that this task would likely require further tweaking where other algorithms showed more promise, this particular path was not pursued.

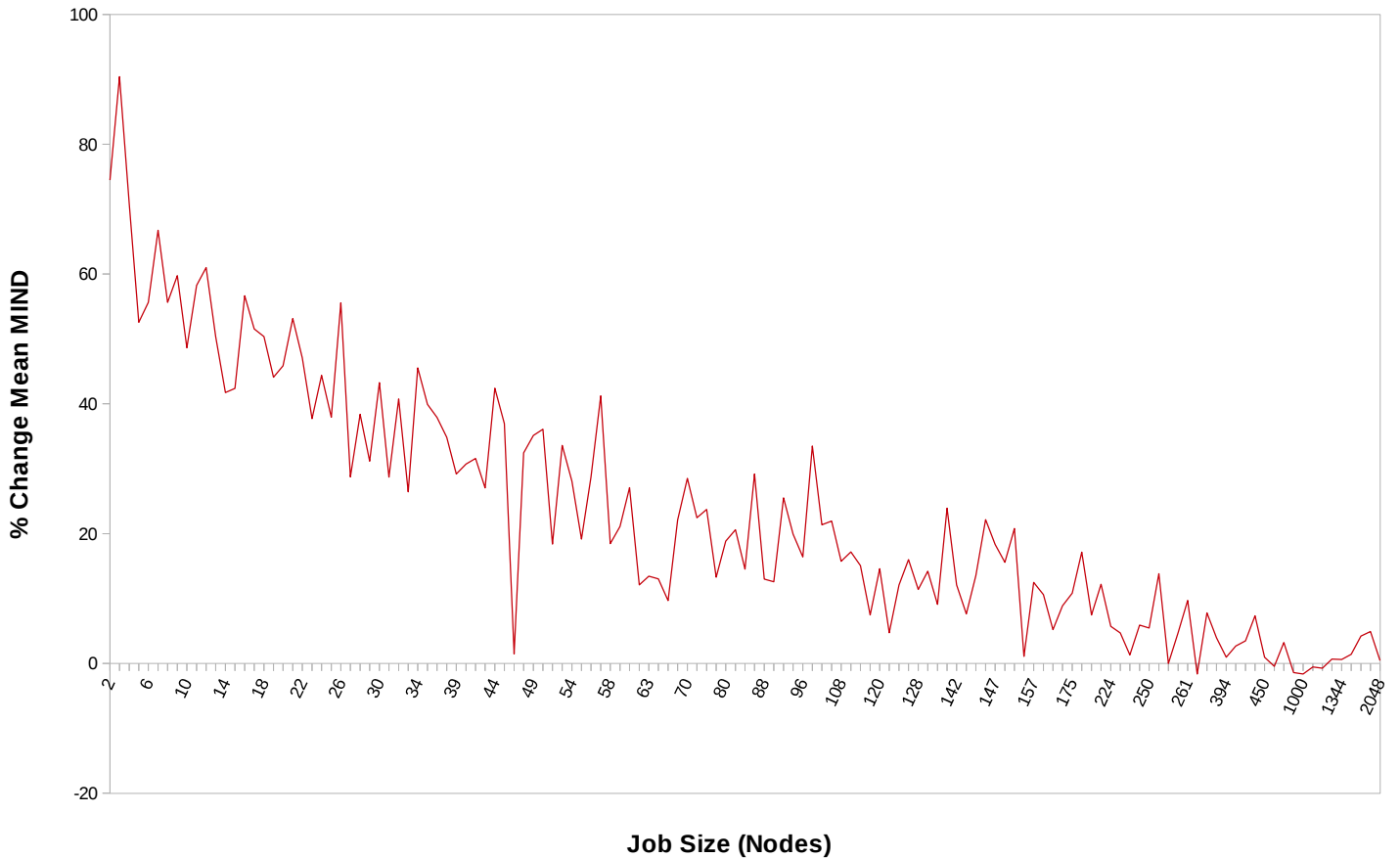


Figure 22: Improvement versus Job Size for Varying Fit Algorithm

5.4 Closest Fit

5.4.1 Algorithm Concept

The `Closest Fit` algorithm returns the subset of free nodes of a particular job size with the minimal largest distance between consecutive nodes in the available list. Because of its reliance on “consecutive” nodes, the success of this algorithm is heavily coupled with the job node ordering of a machine. It works by looping over all free nodes in the list and compares the largest ordinal distance between the nodes within each possible available grouping of a job of size N and returns the one with the smallest value. The algorithm breaks out though at the first available grouping found to be acceptable as determined by an input parameter for gap size. The only reason to test out multiple input parameters here would be to potentially speed the algorithm up (as it would break out sooner with larger gaps being acceptable). However, given that this was the second fastest running algorithm (see Table 3), this did not seem necessary. It was tested though to confirm that the improvement also decreases as larger tunable parameter values are used. The algorithm can be seen in Figure 23.

A variation on this algorithm was tried using the MIND value instead of the minimal largest distance. While the improvement values were higher using this variation, the simulator took on the order of hours to run. This kind of timescale would be unacceptable for a job scheduler on a supercomputer. Optimising the algorithm (or even parallelising it) was outside the scope of this project. However, because the results were somewhat interesting, the values found can be seen in Figure 25 (and in subsequent graphs including all the algorithms).

Closest Fit Algorithm

```
function findClosestFit

    initialize minimumList to empty
    initialize minimumDistance to 99999

    get list of available nodes

    if size of available nodes list is greater than the job size

        while there are still available nodes left in the list

            if the job size is greater than one
                get sublist of size of job size from available nodes
                call getLargestDistanceBetweenNodesInList for sublist
                if the largest distance between nodes in the sublist is
                    less than the minimumDistance
                        set the minimumList to be the sublist
                        set the minimumDistance to be the largest distance
                            between nodes in the sublist
                if the minimum distance is less than or equal to the gap
                    size
                        return the minimumList
            else
                return the next available node
        else if available nodes list size is equal to the job size
            return the list of available nodes

    return the minimumList

function getLargestDistanceBetweenNodesInList

    initialize largest value to -1

    while there are nodes in the list minus the last node

        get the distance between the current node and the next node
        if this distance is greater than the largest value
            set the largest value to the distance

    return the largest value
```

Figure 23: Functions to Get List of Closest Fit Nodes and Largest Distances Between Nodes in a List

5.4.2 Results

The `Closest Fit` algorithm produced the most improvement out of all the algorithms run on the HECToR data. For jobs of size three, the improvement is almost %100 (the actual value is %96.7). Still, at the higher end of the job sizes the algorithm stops performing as well, just as all the other algorithms. This is because there simply is not as much room for improvement for larger jobs, which are placed over most of the nodes of the network. However, the `Closest Fit` results when using MIND values instead of the minimized largest distance ones look even more promising. As can be seen in Figure 25, the results almost looks like a step function and show a noticeable improvement almost all the way up to jobs run on 1000 nodes. This is, of course, the best that can be done here as we are measuring the improvement in the MIND. Perhaps surprisingly though, while this variation of the algorithm tops the improvement values for the overall and large job size values, it only comes in third for the small job sizes (although the difference is only a couple of percentage points). This may be because improving the MIND values for the larger jobs limits the consecutive placements for the smaller jobs and so more fragmentation occurs at the lower level.

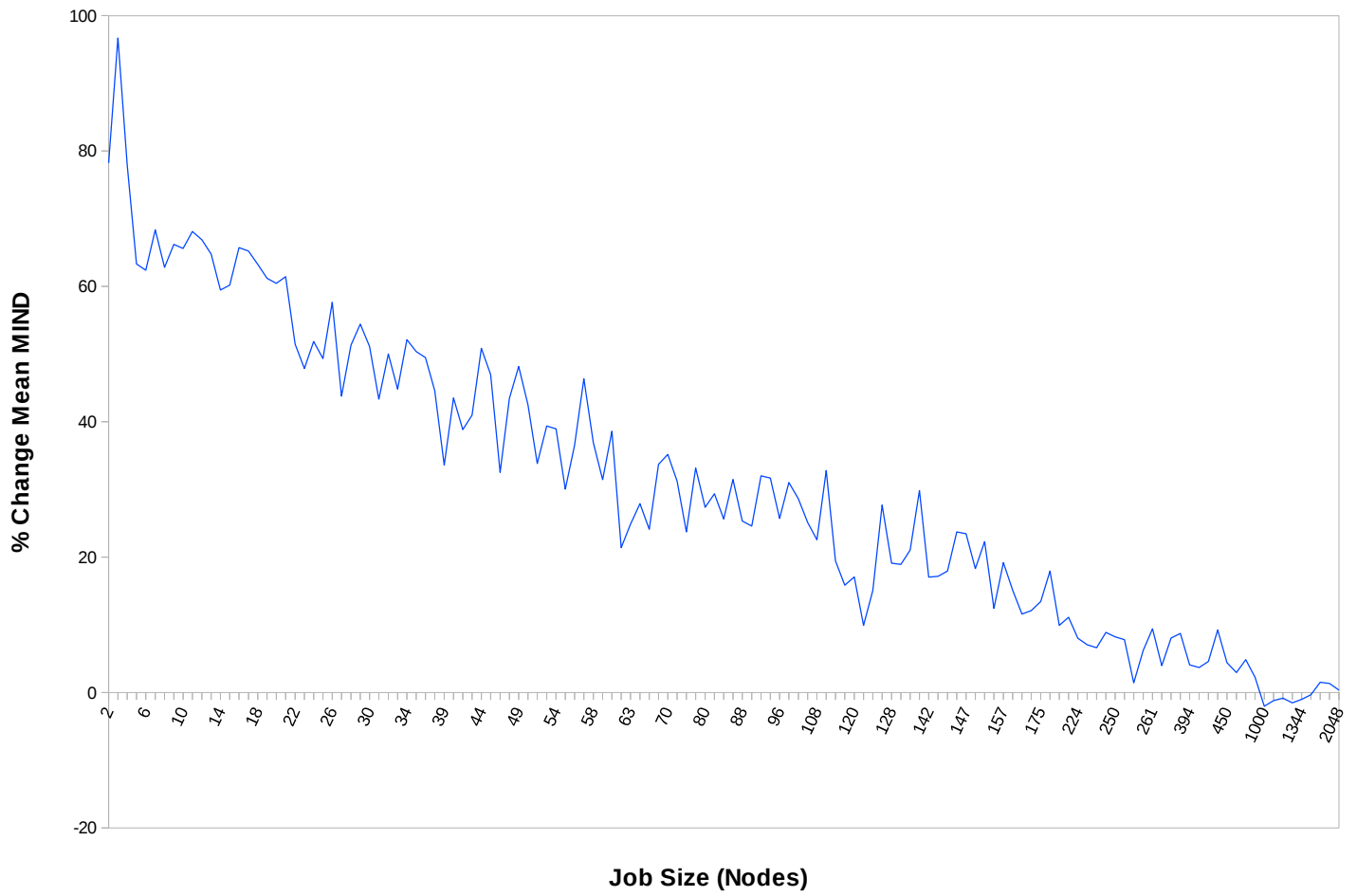


Figure 24: Improvement versus Job Size for Closest Fit With Minimized Largest Distance Algorithm

6 Further Analysis of Job Allocation Algorithms

So far the only results which have been presented have been Improvement versus Job Size graphs for the different algorithms, however there are many other aspects of the data that the following section explores. The first part of this section discusses the effect of normalizing improvement values by CPU usage to shed more light on how much the improvement actually matters per job size. The next part describes the effect of running each of the single parameter algorithms with higher and higher gap sizes in order to find their breaking point. Overall averages for different job types were then calculated for all of the algorithms. To ensure correctness, the data from each of the algorithms was run through a Student's pairwise T-test analysis. The final part of the analysis concludes with a look at the performance of the various algorithms.

6.1 Normalization of Improvement

Along with the number of jobs sizes run on HECToR, the “computing resource” (kAU) value for each job size was also available[14]. These values indicate which jobs are consuming the most amount of CPU time. A binned distribution of the kAU values per job size range for HECToR can be seen in Figure 26. “Fitness factors,” from the normalized values of each bin of this distribution, were created and used as weights on the original improvement graphs (Improvement versus Job Size) in order to see if this unearthed more clear similarities/differences between the algorithms. The assumption here is that the jobs which consume the most time are also communicating the most, which while not always the case is still more likely to be true (however, this is heavily dependent on the type of algorithm being run). Where this is true, then smaller jobs will consume less time because they have less communication overhead. So ideally, when applying these algorithms it is hoped to see more improvement at the larger end of the job size range than smaller ones, but this becomes more difficult as the job size approaches size of the network. Thus by normalizing the graphs in this manner, it was hoped that a less distorted view of the improvement can be seen.

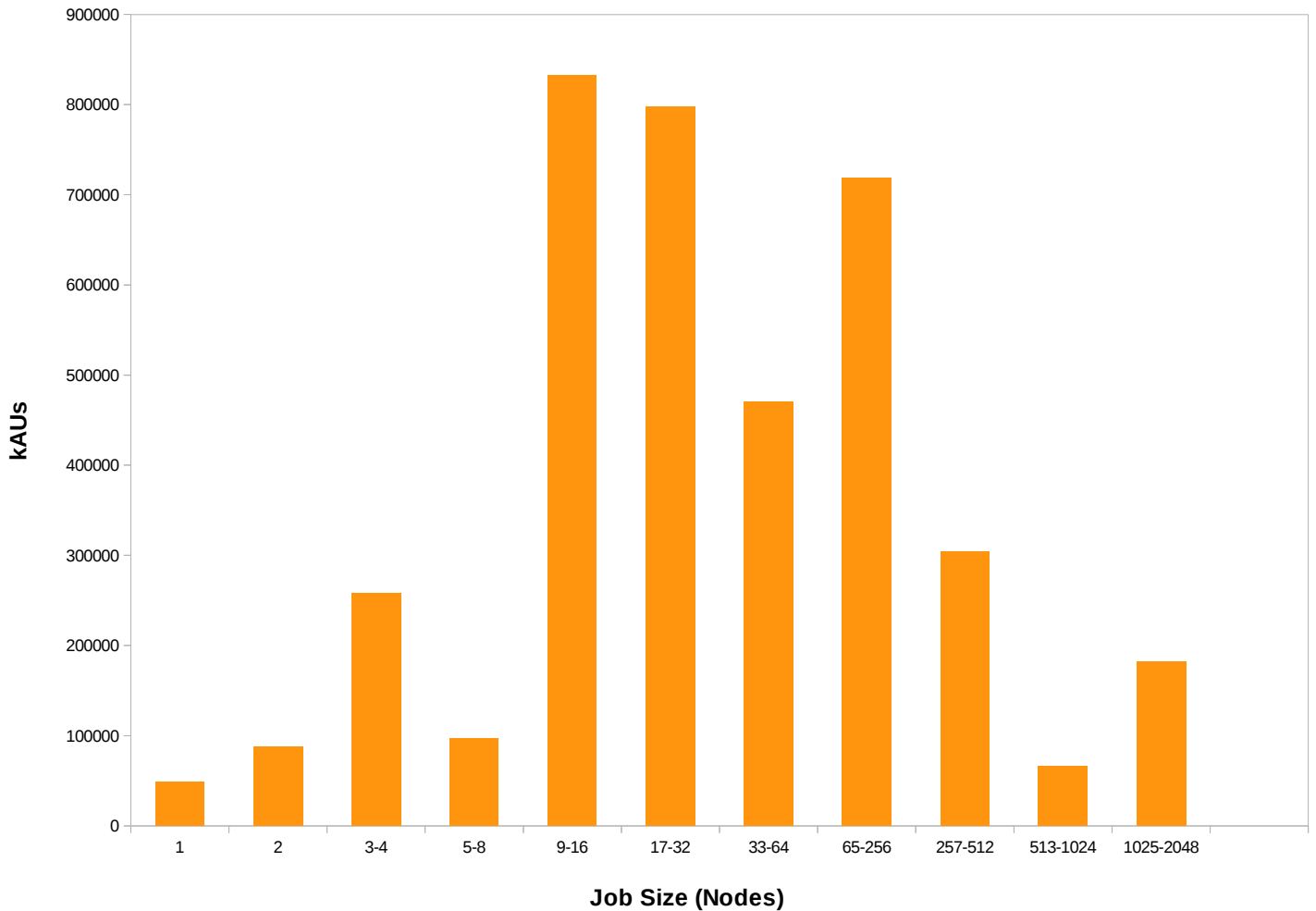


Figure 26: kAU Value versus Job Size for HECToR

The graphs in Figures 27 and 28 show the original and normalized improvement values for each of the best performing (where applicable) algorithms tested. For the normalized version of the graph, it is clear that all the improvement seen at the smaller job end of the graph in Figure 27 (for all but the `FIFO` algorithm) mean less than they seem to. The most improvement seems to actually be happening between job sizes of 9 to ~ 30 nodes. Then up until ~ 175 , the improvement seems to oscillate around $\sim 5\%$, after which it dwindles down to about 0 near 1000 nodes and then there is very little to no improvement seen after this. The top performing algorithms are the same for the normalized case as the original case, however the differences in the magnitudes of percentage improvement are much smaller. So overall the `Closest Fit` still shows the most improvement at $\sim 5\%$ overall. However, as can be seen more clearly between Figures 32 and 33, the differences between the magnitude of the overall improvement is much smaller both between algorithms and between the different job sizes.

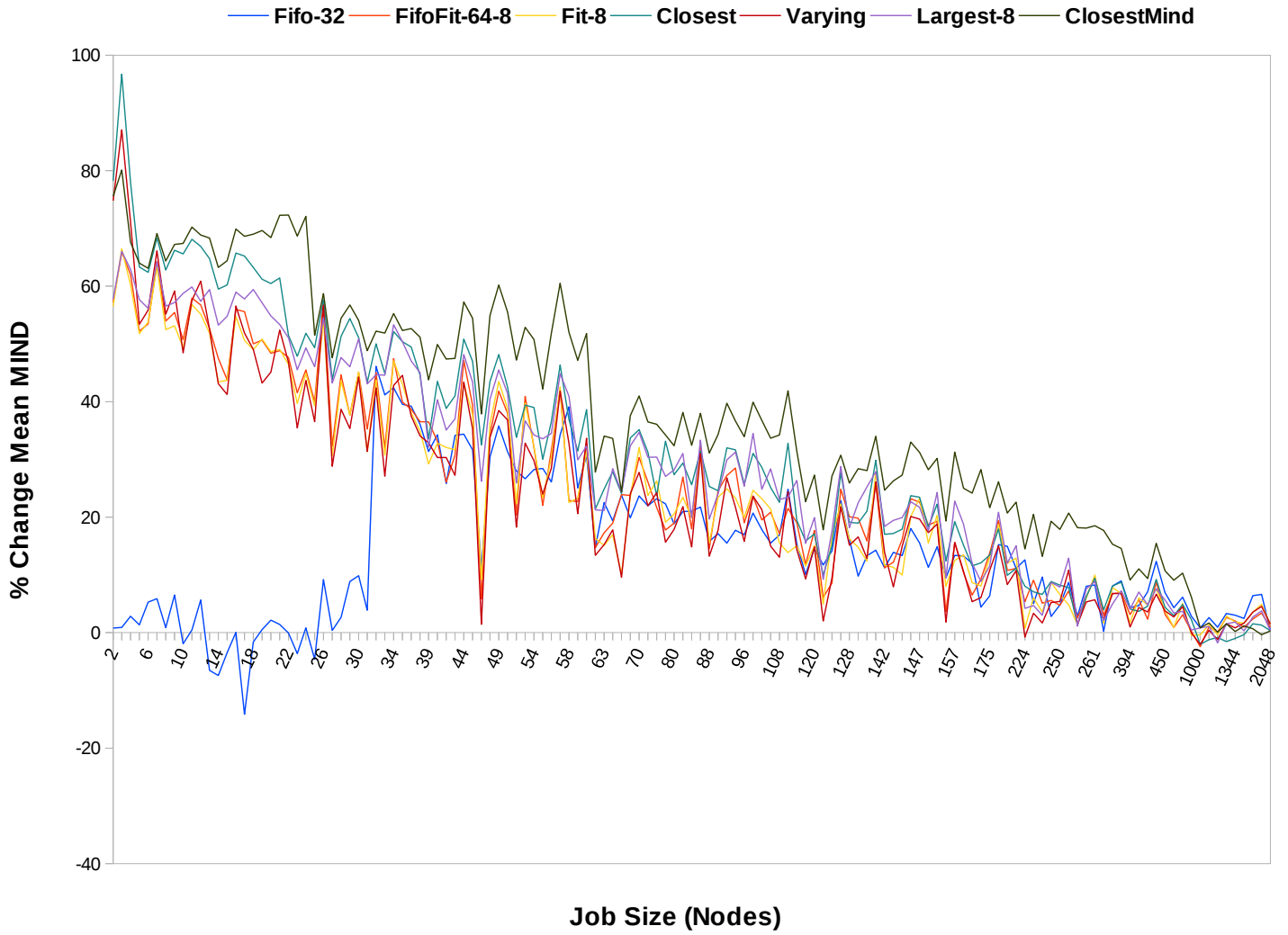


Figure 27: Improvement versus Job Size for Best Performing Algorithms on 3D Torus

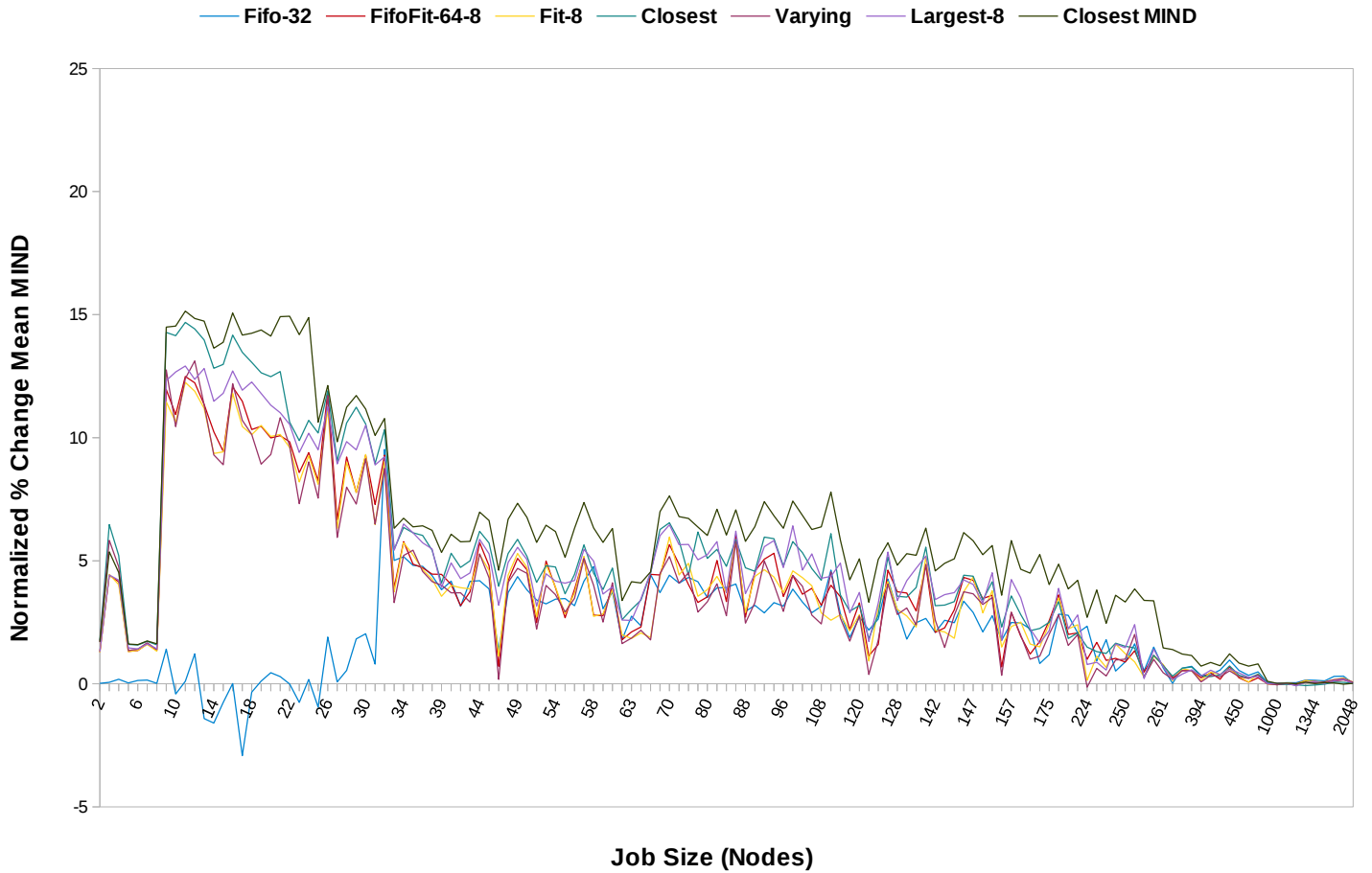


Figure 28: Normalized Improvement versus Job Size for Best Performing Algorithms on 3D Torus

6.2 Threshold Investigation

Three algorithms (First Fit, FIFO and Largest Fit) were run with varying gap sizes up to 2050 to determine at what gap threshold these algorithms break apart. Figures 29, 30 and 31 show these results. The three separate graphs are included to show differences between the overall, small job size (1-10 nodes) and large job size (11-2048 nodes) average improvements respectively. Only these three algorithms were run as they require a variable gap size parameter. FifoFit was not included as it is merely comprised of the FIFO and First Fit algorithms.

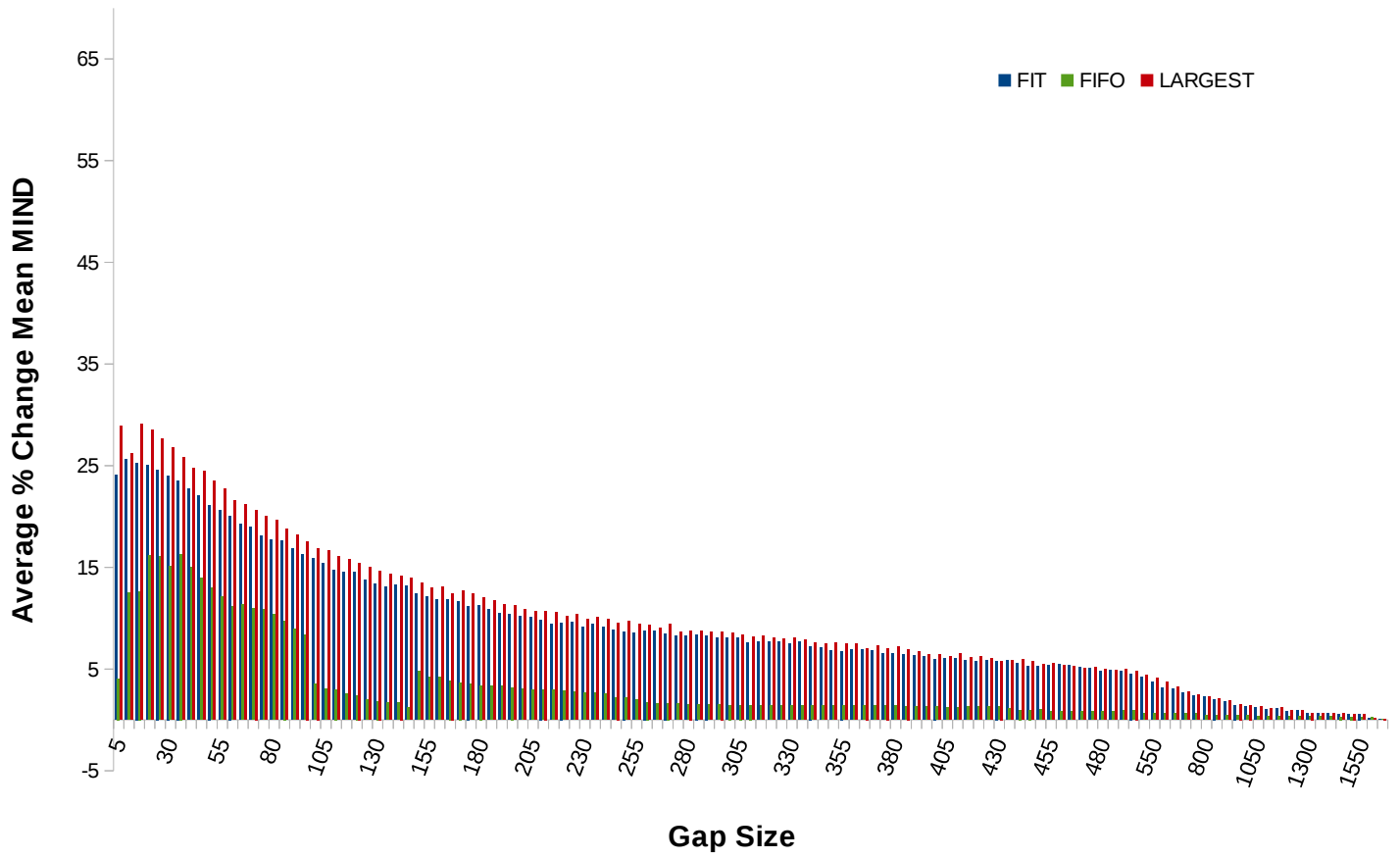


Figure 29: Graph of Average Improvement Versus Gap Size For First Fit, FIFO and Largest Fit Algorithms

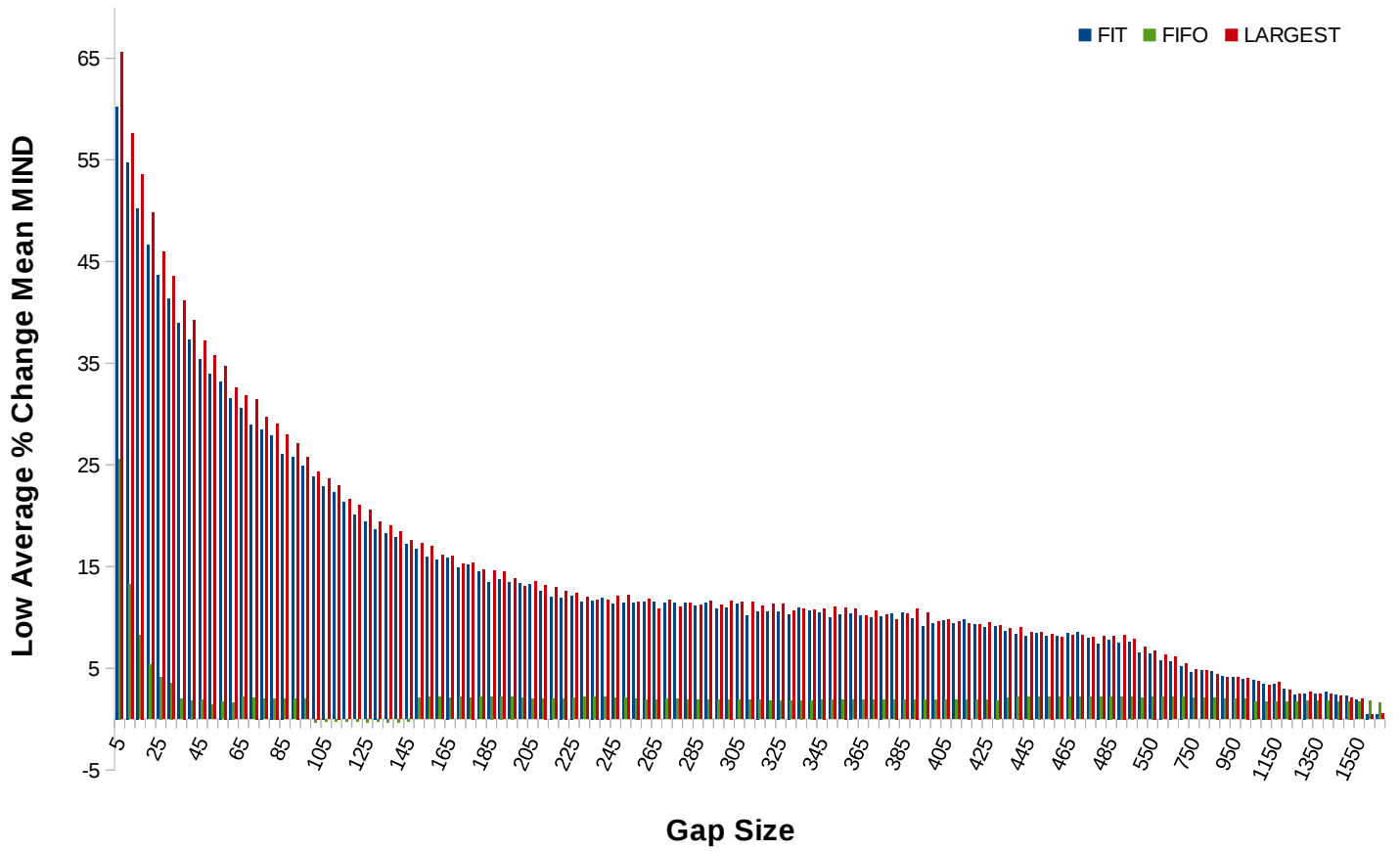


Figure 30: Graph of Low Job Size Average Improvement Versus Gap Size For First Fit, FIFO and Largest Fit Algorithms

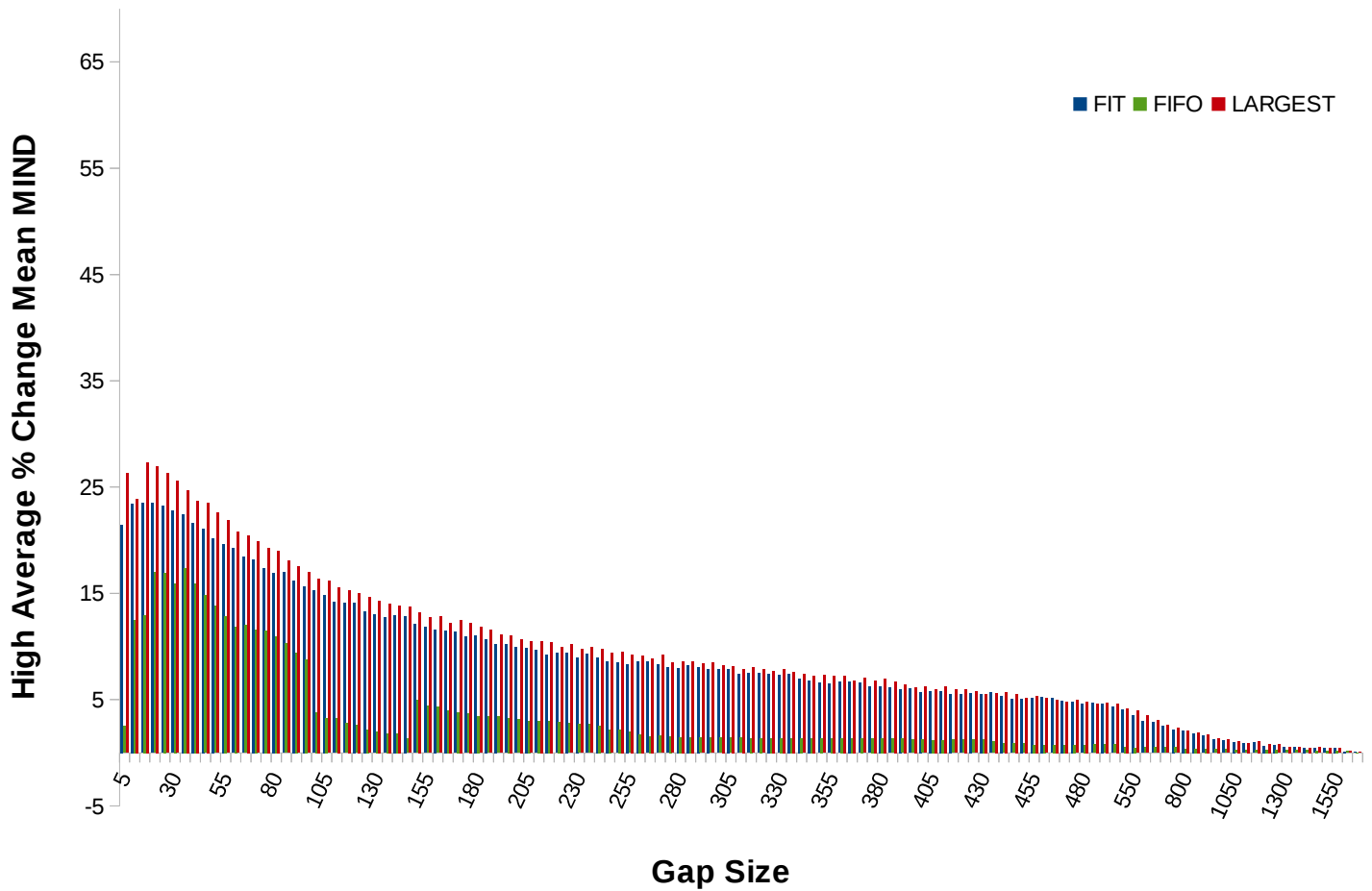


Figure 31: Graph of Large Job Size Average Improvement Versus Gap Size For First Fit, FIFO and Largest Fit Algorithms

Across all three graphs, the Largest Fit and First Fit algorithms follow the same movement from the smallest to largest gap sizes, whereas the effect on the FIFO algorithm is quite different. In the overall average and large average graphs, the shape for the Largest Fit and First Fit peaks around 15 nodes and slowly diminishes with very little going on after a gap size of 1550. Given that the average job size is just over 10 nodes, this value makes sense. The FIFO algorithm follows the same shape in these two graphs as well, however it jumps up and down quite a bit more and peters out a lot more quickly (at around a gap size of ~ 500). This is probably because there are far fewer numbers of jobs after this size, so overall not a lot of difference will be made using such large gaps. For the low average graph, the Largest Fit and First Fit algorithms peak much higher at the smallest gap size (5) and then tail off at a much steeper slope, but again petering out around 1550 nodes. Interestingly, for all three graphs, while the Largest Fit values start out slightly higher than the First Fit values, after around a gap size of 450 they start to converge. The FIFO algorithm also peaks at the smallest job size, but tails off even more quickly and then short of a gap of oscillating around zero improvement between 100 to 150

nodes remains steadily around %1-2 improvement until the very end even past 2050 nodes. While this seems wrong, there are actually a number of jobs greater than 2048 in the log file, but they are not registered in any of the graphs here because they are test runs (and indeed there are fewer than thirty of them, see Section 2.5.1). So having these larger jobs come from the opposite end of the list must still help some of the smaller jobs be placed more compactly.

6.3 Averages Comparison

The average improvement for small jobs, large jobs and all jobs for all of the algorithms are shown in Figures 32 and 33 for original values and normalized values respectively. The most striking difference between the original and normalized graphs is that, similar to what was seen in Section 6.1, when the original values are multiplied by the fitness factors the small jobs have a much smaller impact overall. Additionally, the differences flatten out overall between different parameter values for the algorithms in the normalized graph for the low values. However, the opposite is true for the overall and larger job averages. Here, the differences seem to become magnified. This makes sense considering how much more kAU there is for jobs between 9-256 nodes. The improvement in improvement can be seen here much more clearly between increased gap sizes for the `FifoFit`, `Largest Fit` and `First Fit` algorithms. For the latter two, it can be more clearly seen that the improvement goes up along with gap size for the larger jobs and overall, while it goes down for smaller jobs.

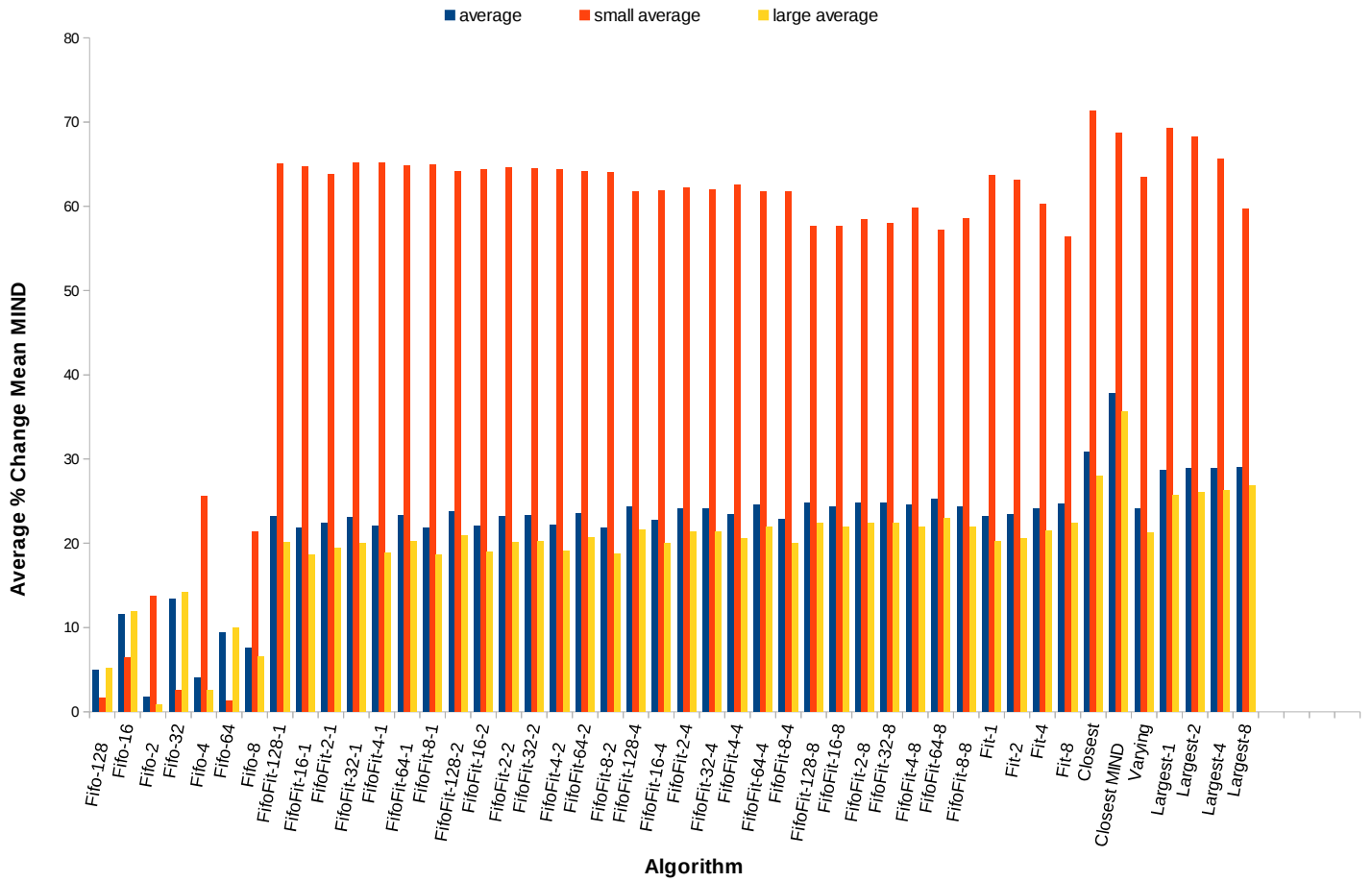


Figure 32: Overall Averages

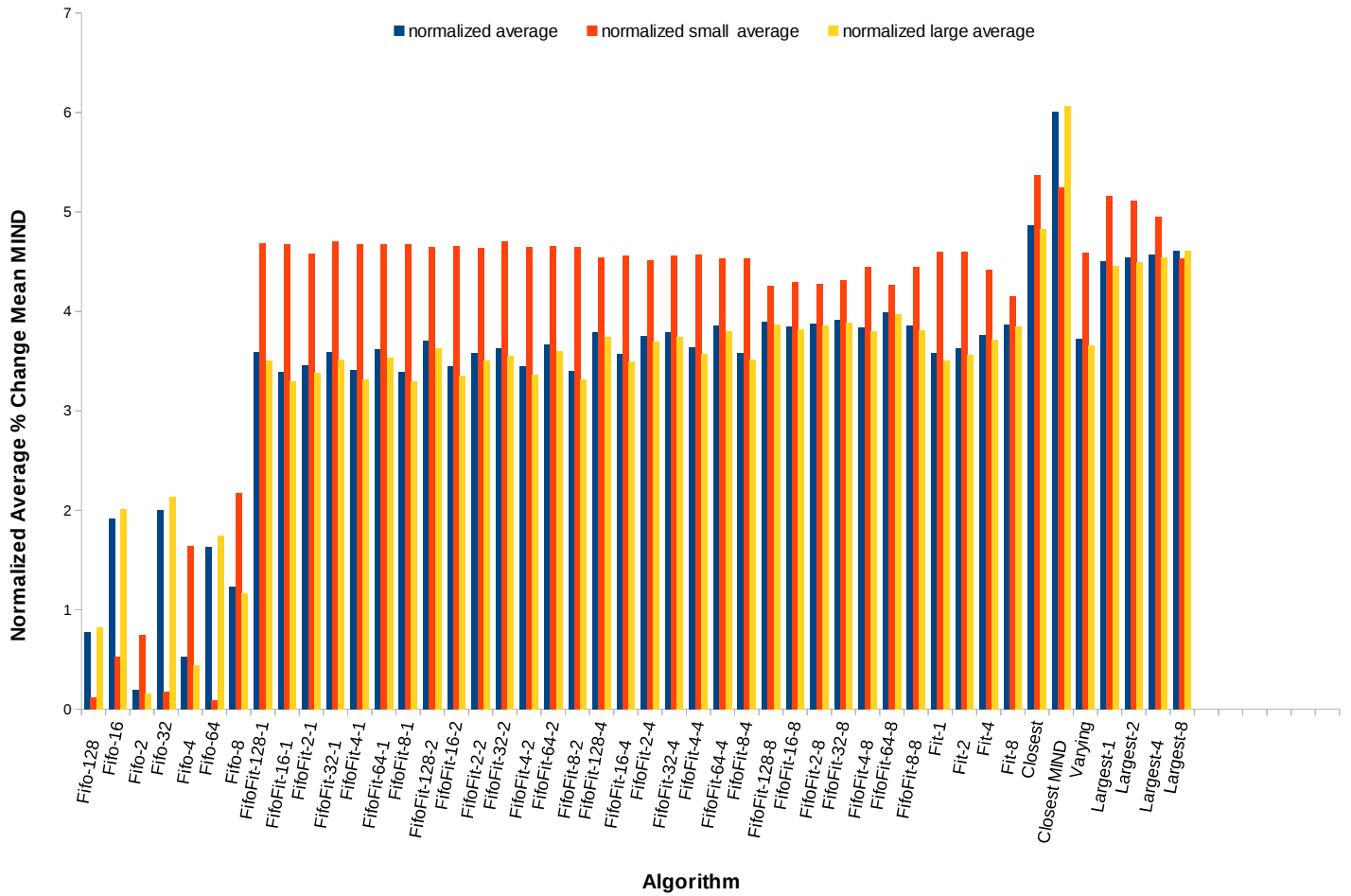


Figure 33: Normalized Overall Averages

Table 1 shows the top three algorithms in order of the percentage increase in improvement for small jobs, large jobs and overall. The `Closest Fit` showed the most improvement in all three job types and the `Largest Fit` showed the next highest improvement for both the second and third places of all three job types. However, the parameters for the `Largest Fit` in second and third places was different for the small job sizes than for the large job sizes and overall. As explained previously in Section 5.1.2, smaller gap sizes show more improvement for small job size placements and larger gap sizes show more improvement for the large job size placements for the `Largest Fit` algorithm.

Small Job Sizes		Large Job Sizes		Overall	
Algorithm	Improvement	Algorithm	Improvement	Algorithm	Improvement
Closest	71.28	Closest	27.95	Closest	30.88
Largest-1	69.29	Largest-8	26.81	Largest-8	29.037
Largest-2	68.30	Largest-4	26.28	Largest-4	28.94

Table 1: Top Three Algorithms for 3D Torus Topology

6.4 T-Test Analysis

The results were run through a paired Student's T-test algorithm to determine if the results were significant (as well as to compare with the NERSC results to see if they were conclusive). The T-test shows if the improvements are valid improvements overall by looking at individual changes for each placement[18]. An assumption is made that the algorithms will all show improvement overall greater than 0. Therefore the null hypothesis constitutes values less than or equal to zero. So to be valid, the T-test value needs to be above 0.05, which all of the algorithms are. As the average of the differences gets bigger, then the value produced by the T-test gets bigger, so better performing algorithms will have higher T-test values. Values reported in Table 2 use the average values of improvement, normalised by job size.

algorithm	mean	standard deviation	T-Test	valid
Fifo-128	1.07417	1.84998	30.8259	yes
Fifo-16	0.995274	1.66964	144.802	yes
Fifo-2	0.903076	1.54181	155.466	yes
Fifo-32	1.05037	1.78554	85.8939	yes
Fifo-4	0.810734	1.28785	253.376	yes
Fifo-64	1.07721	1.84333	32.1558	yes
Fifo-8	0.898599	1.50908	223.756	yes
FifoFit128-1	0.365051	0.68575	600.815	yes
FifoFit16-1	0.369218	0.70032	595.72	yes
FifoFit2-1	0.373645	0.731024	588.798	yes
FifoFit32-1	0.358394	0.690245	605.615	yes
FifoFit4-1	0.354944	0.698356	598.983	yes
FifoFit64-1	0.360961	0.690005	603.695	yes
FifoFit8-1	0.368196	0.696206	594.769	yes
FifoFit128-2	0.378245	0.693257	597.013	yes

FifoFit2-16	0.376985	0.690734	595.169	yes
FifoFit2-2	0.359857	0.717607	597.948	yes
FifoFit32-2	0.374336	0.689903	601.515	yes
FifoFit4-2	0.367948	0.700379	597.034	yes
FifoFit64-2	0.377472	0.693244	596.952	yes
FifoFit8-2	0.384564	0.706944	589.143	yes
FifoFit128-4	0.419477	0.711402	583.878	yes
FifoFit16-4	0.421898	0.715753	579.648	yes
FifoFit2-4	0.401975	0.718043	584.773	yes
FifoFit32-4	0.416397	0.709642	585.787	yes
FifoFit4-4	0.403487	0.733471	588.398	yes
FifoFit64-4	0.416651	0.707613	586.27	yes
FifoFit8-4	0.423429	0.709189	580.923	yes
FifoFit128-8	0.477916	0.769164	555.535	yes
FifoFit16-8	0.477117	0.766963	558.95	yes
FifoFit2-8	0.454383	0.750176	563.166	yes
FifoFit32-8	0.473175	0.764466	561.305	yes
FifoFit4-8	0.450678	0.767411	570.912	yes
FifoFit64-8	0.480437	0.771976	557.122	yes
FifoFit8-8	0.475643	0.763179	560.501	yes
Fit-1	0.38064	0.71578	589.07	yes
Fit-2	0.394499	0.719194	584.014	yes
Fit-4	0.438778	0.748051	570.009	yes
Fit-8	0.491245	0.787496	545.11	yes
Closest	0.312544	0.464702	679.43	yes
ClosestMind	0.343521	0.428882	681.091	yes
Varying	1.01559	1.8242	160.781	yes
Largest-1	0.329519	0.51473	655.87	yes
Largest-2	0.348658	0.520518	649.271	yes
Largest-4	0.389974	0.580331	630.609	yes
Largest-8	0.465111	0.681025	586.953	yes

Table 2: Algorithm Data Analysis and T-Test Results

6.5 Performance Timings

The simulation was run five times for each algorithm and the results of the average of those times are plotted in Table 3. The timings were taken over the whole simulation run for the entire year-long log data to account for differences in timings for large and small jobs as well as to get more reliable timings overall. The `FIFO` algorithm performed the fastest, which is no surprise given that it has only one potentially computationally intensive section for each placement (which is when a list requires reversal). However, given a valid gap size, this reversal only occurs when a job size is smaller than the “large” parameter, which is not always the case. The `Closest Fit` algorithm (using minimized maximum distance) comes in second place in terms of time, which makes sense given that only iterates over the whole network once and breaks out once it has found a suitable fit. The `Varying Fit` algorithm, which uses logarithms (which are expensive calculations), takes the longest of the feasible algorithms. Of course, the `Closest Fit` (using the `MIND` calculation) takes on the order of hours, which makes sense given that it calls an $O(N^2)$ algorithm for each potential fitting in the network. The `First Fit`, `Largest Fit` and `FifoFit` have very close times and they all use similar recursive approaches. The `Largest Fit` approach is slightly more optimal than the `First Fit` approach as it optimises for larger chunks and uses a single recursion instead of a divide-and-conquer one. As expected, the `FifoFit` runs slightly longer than the `First Fit` given the possibility of having to reverse the list before traversing it.

Algorithm	Overall Time for Simulation Run (s)
First Fit	89.9086
FIFO	69.3062
Largest Fit	88.0878
Varying Fit	287.6006
FifoFit	92.5408
Closest Fit (<i>Minimal Largest Consecutive Distance</i>)	81.7104
Closest Fit (<i>MIND</i>)	7843.491

Table 3: Timing Comparisons for Simulation Runs Using Different Algorithms

The long-term goal of using more compact placements is to improve performance of jobs run on supercomputers. If a particular scheduling algorithm took too long, this would undo the benefits of using it in the first place. Additionally, the algorithm will be used each time a job is placed, so this overhead could add up. For these reasons the performance of the algorithms themselves is as important as their effects.

7 Results On Other Topologies

The results covered so far have only covered one specific topology: the 3D torus. There are many other possible network configurations, two of which are covered in this section. To produce these results, the simulator codebase was first updated to accommodate other topologies. As real data for these topologies was not available (and comparisons with such would have been less enlightening as other variables - such as the job distribution - may have encroached), the next step required re-mapping the original number of nodes on HECToR to fit new network coordinates. This was done for a dragonfly topology and 5D Torus. The results of both of these incarnations are discussed further below.

7.1 Code Refactor

The simulator code was originally only written to run on a 3D Torus network, so several changes to the class structure were required to simulate the new topologies. First, the `Torus` class was refactored to inherit from an abstract `Topology3D` class and a parent abstract `Topology` class was subsequently created above this. A new `Dragonfly` class then extends from the `Topology3D` class and a new `Torus5D` class now inherits from the `Topology` class, as it has 5 coordinates instead of 3. Additionally, an abstract parent `Coordinate` class was created to handle the varying number of dimensions. The `Coord3D` and new `Coord5D` classes now inherit from this class. Figure 34 shows this new class structure hierarchy. A new (mocked) coordinate mapping of node ID (also used as the ordinal value) to coordinate value on the network was also created and put in the database for each new topology. Where multiple nodes can be found at the same coordinate, these values will overlap. Finally, a new unit test was created and the whole unit test suite was run to ensure that the refactor did not also break pre-existing functionality.

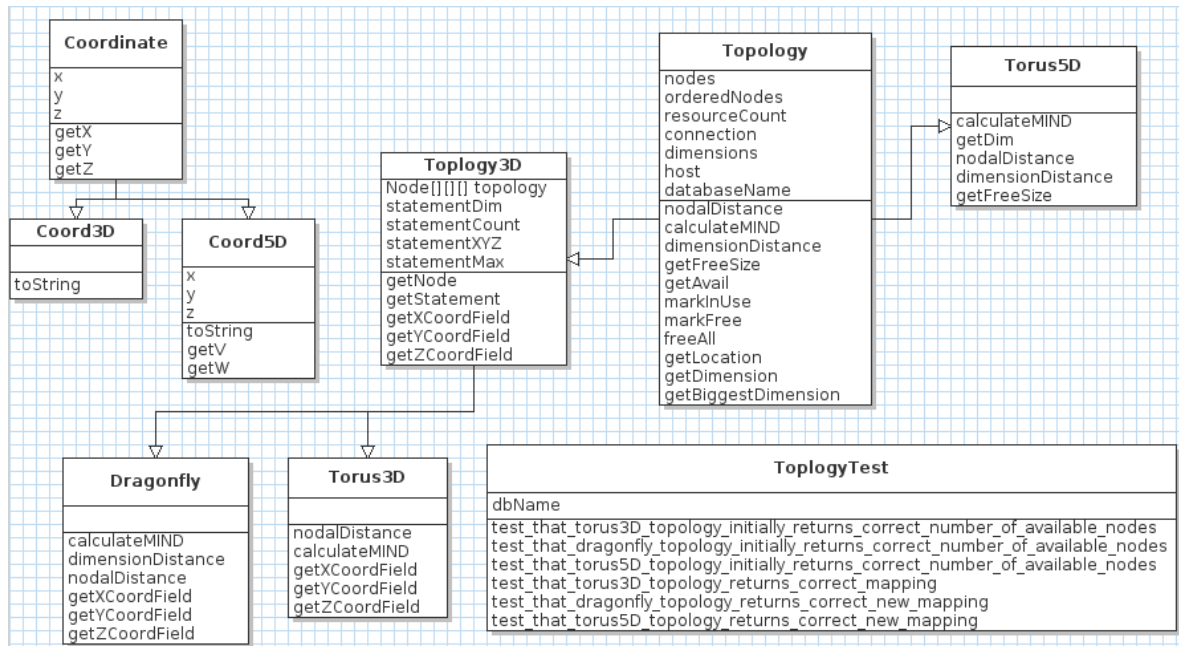


Figure 34: Refactored Simulation Code for New Topologies UML Diagram

7.2 Dragonfly

Cray has developed a new dragonfly network topology to be used on their Cray XE30 machines. This is particularly interesting to explore as the successor to HECToR service (ARCHER) will be a Cray XE30 running the same job distribution. The topology consists of 3 levels: blades, chassis and groups. Four nodes are found on a blade, sixteen blades connect to one chassis and six chassis exist in a group. Each chassis has an all-to-all mapping between blades and each group has an all-to-all mapping between chassis. A picture of a possible group is shown in Figure 35. To use the current HECToR network of 2816 nodes thus requires 8 groups, albeit not all of the last group was used. The dimensions of the new network are 16 x 6 x 8.

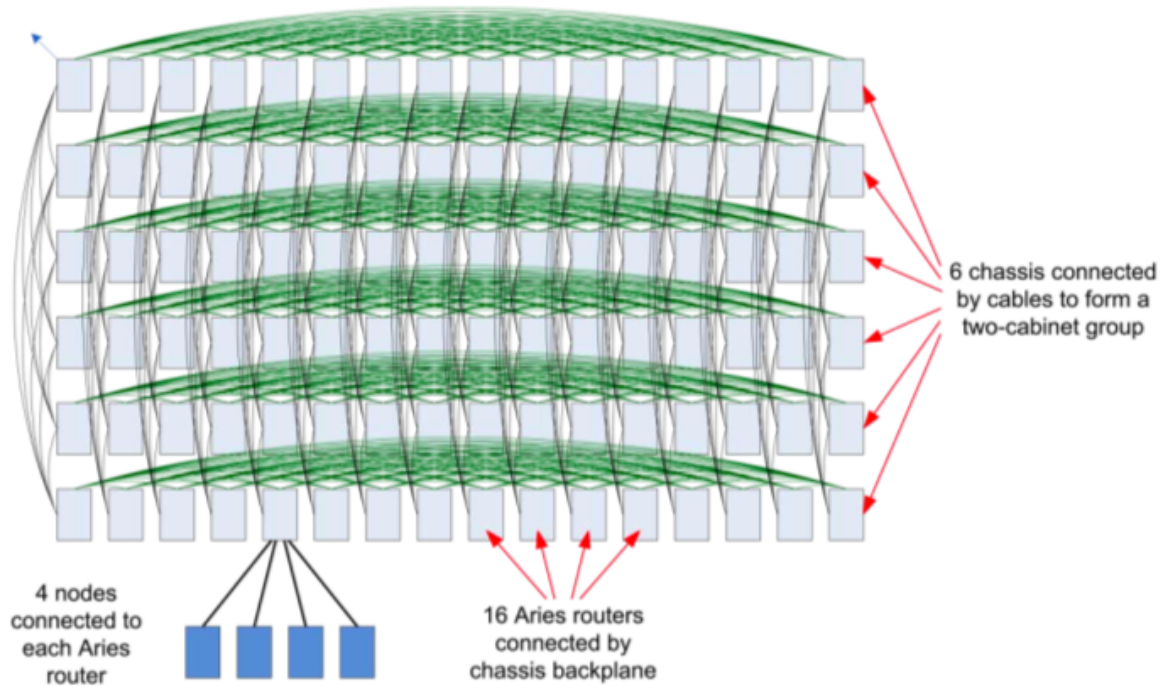


Figure 35: Dragonfly Network Topology[15]

One of the main draws of this new network topology is its dynamic nature. Messages can be bounced off of other blades intra-chassis and off of other blades inter-chassis within a group for a maximum of 4 hops per group. To simulate this network precisely would require much tweaking of the current simulation and more information about system overloading than was available. As such, only the best case and worst case scenarios for the physical connectivity of the system are simulated.

7.2.1 Best Case Scenario

The best case scenario restricts the maximum number of hops between any two nodes in the network to be three. Physically, this equates to an all-to-all connection between the groups. That is, there is a physical mapping of each node in a group to its corresponding node on every other group. Additionally, it assumes that the router never makes more than one hop between blades or chassis. Although this is a possibility, it is unlikely to always be the case.

Results from running this scenario showed that the network topology itself also makes a difference in the magnitude of improvement that can be seen from a job placement algorithm as well as the limit of improvement for the range of job sizes. Interestingly, no improvement was seen in the best case dragonfly simulation after the size of the job exceeds ~ 450 nodes for any of the algorithms. This is just above the number of nodes found in a group (384).

After 384 nodes, across almost all of the algorithms, the percent improvement drops by several points. This is probably because the jump from intra-group to inter-group is the last stage in which the hop value actually increases for this particular topology. So where a job is large enough to span multiple groups, there is much less likelihood of seeing more compactness from different types of spanning and after a point (450 nodes) there is no chance of it. As in the 3D Torus, the `Closest Fit` algorithm showed the most improvement in all categories for the best case dragonfly network (see Table 4 for more details). Comparably, the overall improvement was actually smaller than for the 3D Torus, as the maximum possible number of hops in this network is much smaller (in this scenario, the maximum number of hops is only three versus the eighteen in the 3D Torus). In this regard, the MIND comparisons are not quite equal and the percent increase seems like much more than it is.



Figure 36: Improvement versus Job Size for Algorithms on the Best Case Dragonfly Topology

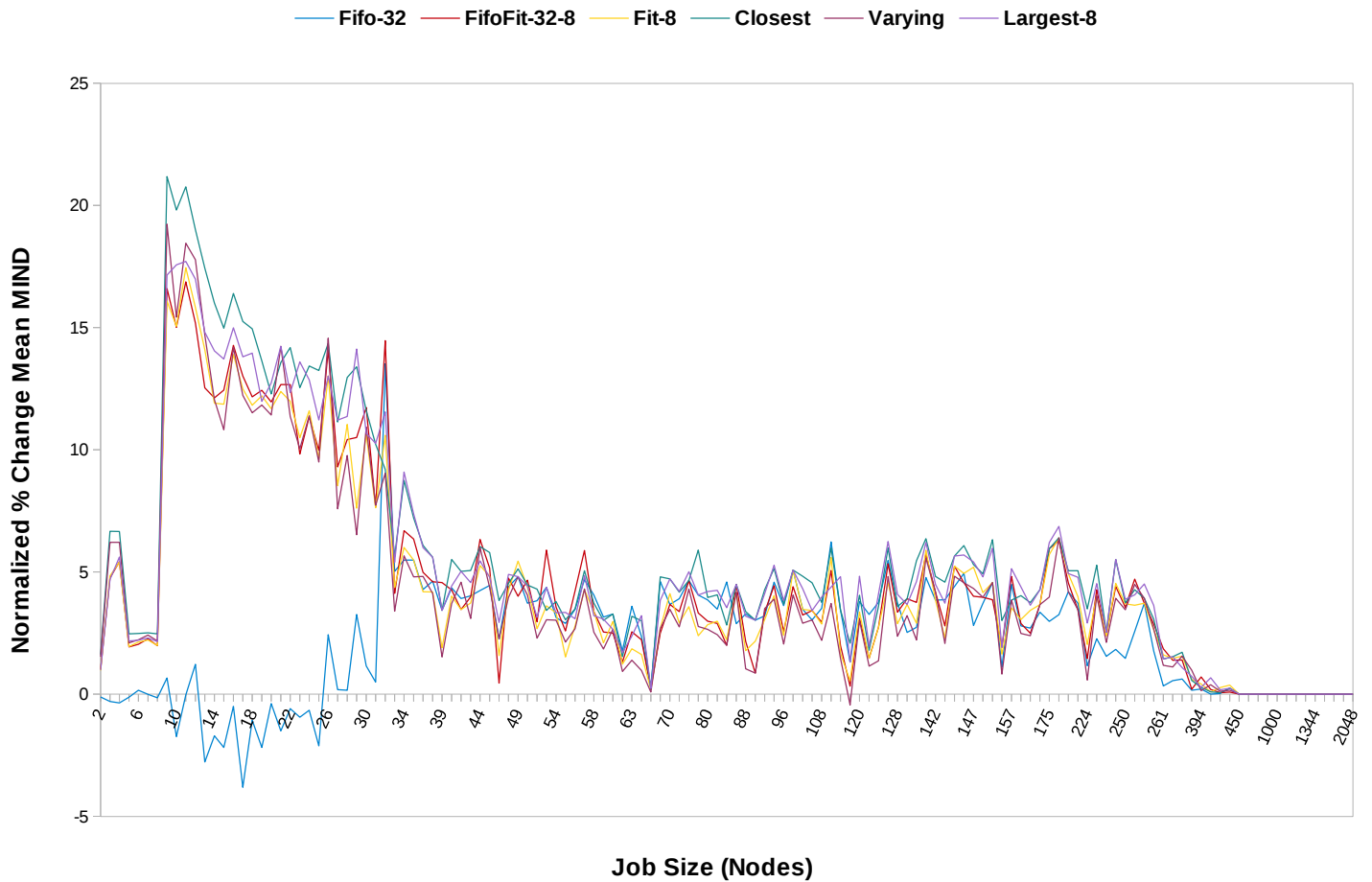


Figure 37: Normalized Improvement versus Job Size for Algorithms on the Best Case Dragonfly Topology

Small Job Sizes		Large Job Sizes		Overall	
Algorithm	Improvement	Algorithm	Improvement	Algorithm	Improvement
Closest	94.99	Closest	30.87	Closest	35.21
Largest-1	92.41	Largest-8	29.55	Largest-1	33.28
Largest-2	92.20	Largest-1	28.99	Largest-2	33.17

Table 4: Improvement Values for the Top Three Algorithms for Best Case Dragonfly Topology

7.2.2 Worst Case Scenario

As the worst case scenario is strictly a physical distinction, the number of hops can range from 0-5. The assumption made here is that groups are connected only from one node to one other node in a different group. For this simulation, the connection was simplified to always be at the node coordinate $(0,0,z)$, where z is the group node number. That is, only one cable

connects one group to another at the 0th base and 0th chassis. In reality, due to the unique routing of the dragonfly topology, the number of hops can be up to 7.

Interestingly, results showed that slightly more improvement could be seen for this setup than the for the best case scenario for large jobs and overall. However, the magnitude of the improvement was not significantly larger. Additionally, the improvement for smaller jobs was conversely slightly worse than for the best case scenario. Keeping in mind the differences in the number of maximum hops in these two networks (5 versus 3), the improvement is actually probably less overall. These results are likely due to a couple of different factors.

One reason for this behaviour could be that given the bottleneck of only one node connection between groups, there is a much greater possibility of worse placements in comparison to the best case scenario. So while the allocation algorithms have on the whole found more better placements, they have also probably found a number of worse placements which drags down the average. This is much more apparent for larger jobs at the higher end of the spectrum, where the improvement weaved much more frequently (and with higher magnitudes) between having positive and negative values. These negative values start showing up right around jobs with 450 or more nodes, where the best case dragonfly configuration started seeing no improvement. It seems slightly strange that given this single connection restriction that larger job sizes and overall this setup responded more favorably than the best case scenario, but this is probably because there are a larger number of possible hops (ie. 5 instead of 3). Where enough large jobs have improved more markedly, this could make up for the fact that many of the very large jobs responded so poorly.

It is also possible that this response can be partially attributed to the difference in the nature of what a “gap” (or even to be “ordinal”) means in a 3D torus versus a dragonfly. On a 3D torus, hops between nodes are equal for all but those that share the same coordinate, whereas on a dragonfly network there are levels of hop jumps that two nodes can differ on (ie. via blades, chassis and groups). In particular, in this scenario, because none of the algorithms are aware of where the one connection between groups is they are just as likely to put place jobs further away from that coordinate in a particular group than near it. If the job is smaller than a group, then this matters less (except when the network is quite saturated), but larger jobs will be more affected by this bottleneck. Thus, while the nature of a gap on this network is different, the algorithms have not been updated to reflect this, so do not show as much improvement as they might have potential to. Again, because the number of maximum hops in this network (5) is much smaller than in the 3D torus, the improvement seen is not comparable.



Figure 38: Improvement versus Job Size for Algorithms on the Worst Case Dragonfly Topology

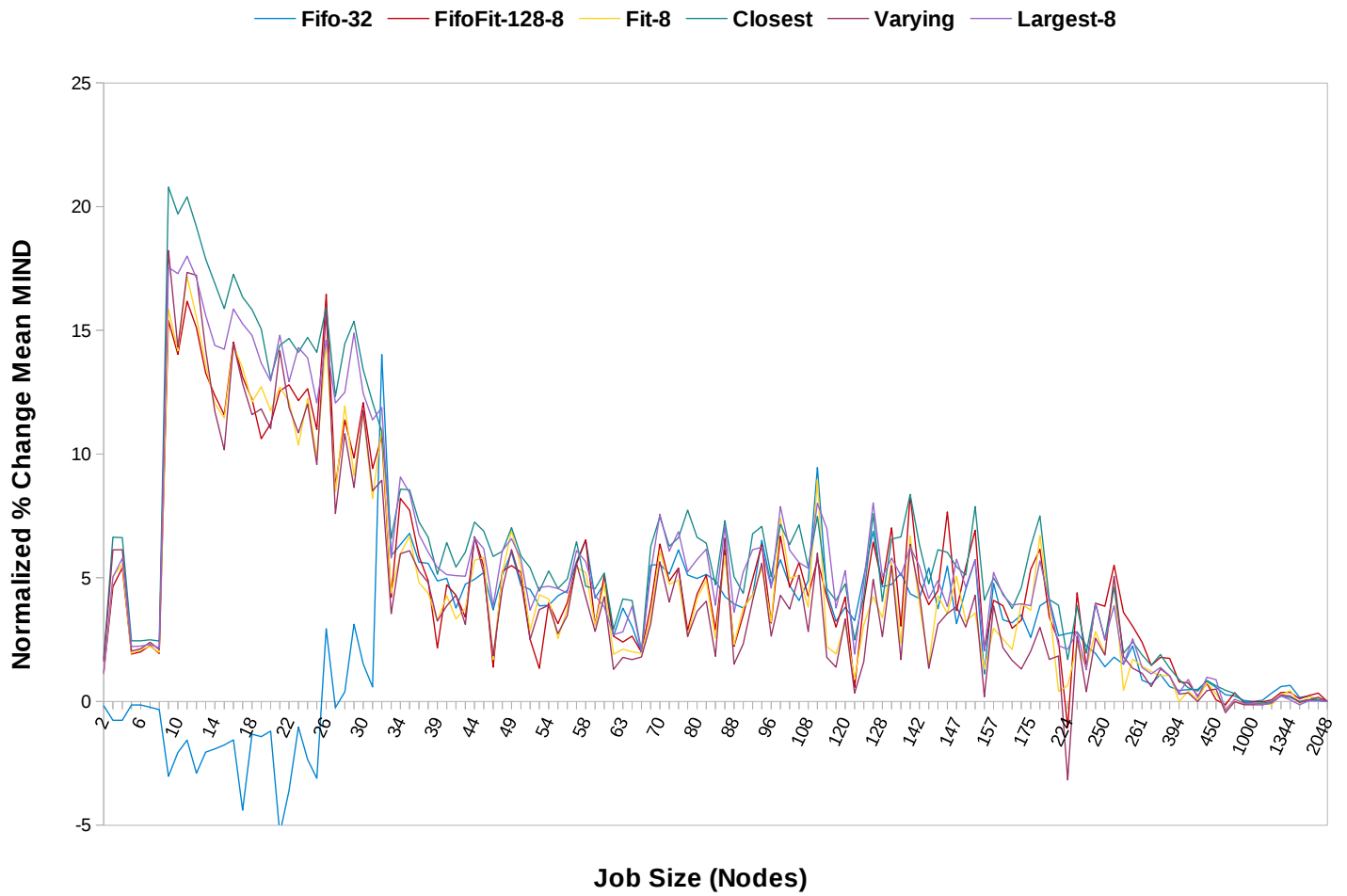


Figure 39: Normalized Improvement versus Job Size for Algorithms on the Worst Case Dragonfly Topology

Small Job Sizes		Large Job Sizes		Overall	
Algorithm	Improvement	Algorithm	Improvement	Algorithm	Improvement
Closest	94.66	Closest	36.67	Closest	40.59
Largest-2	91.54	Largest-8	33.25	Largest-8	36.48
Largest-1	91.39	Largest-1	32.02	Largest-1	36.04

Table 5: Improvement Values for the Top Three Algorithms for Worst Case Dragonfly Topology

7.3 5D Torus

IBM has also implemented a new topology on their Blue-Gen/Q machine. This topology is a 5D-torus shape, which is similar to the 3D-torus except there are two extra dimensions[16]. Figure 40 attempts to convey this setup. To mock a 5D Torus with 2816 nodes means that

instead of running on a $15 \times 6 \times 16$ grid, the network dimensions would be $11 \times 8 \times 4 \times 4 \times 2$. All of the dimensions have a wrap-around, so modular addition is again used when calculating the Manhattan Distance. The maximum number of hops on a 5D torus is 14.

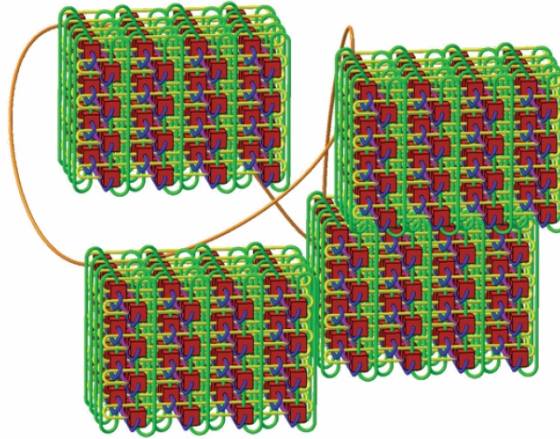


Figure 40: 5D Torus Network Topology [16]

Running the algorithms on the 5D torus simulation produced results that were similar to the 3D torus, albeit with different values of improvement. The top three algorithms for all the job types were the same, although only a fraction of the improvement was seen. Given the similarities in these two networks and that the maximum hops is somewhat smaller for the 5D torus, this makes sense. Similar to the worst case scenario on the dragonfly network, for very large jobs some of the improvement was negative. Perhaps there is a similar bottleneck going on on this network, whereby because there the largest dimension is 5x the smallest dimension, at some point the large job sizes simply are far less likely to find a better fit without some “awareness” in the algorithm of the best direction to place in. On topologies such as this in particular, node ordering could potentially prove beneficial in increasing the effectiveness of some of these algorithms

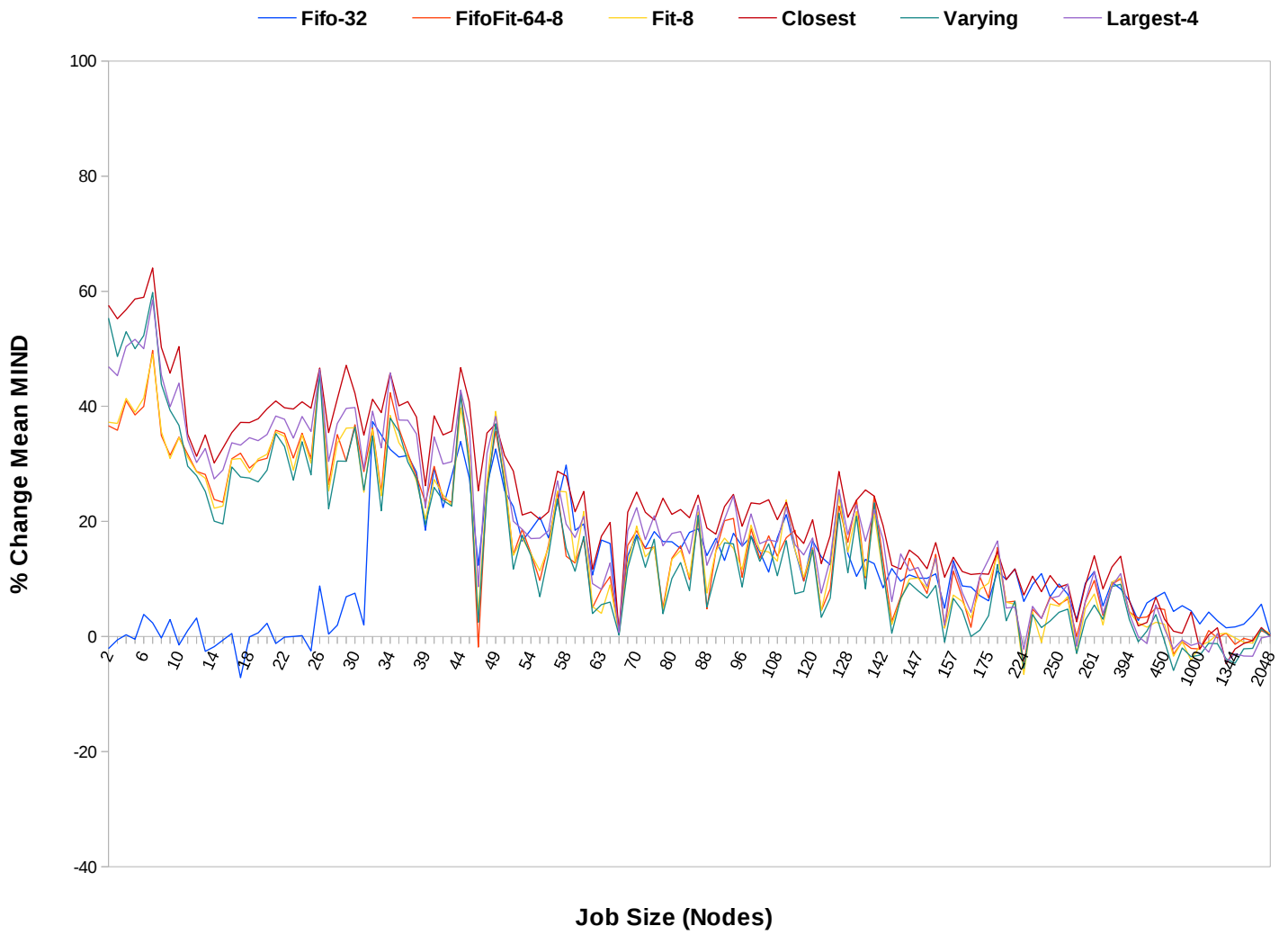


Figure 41: Improvement versus Job Size for Algorithms on the 5D Torus Topology

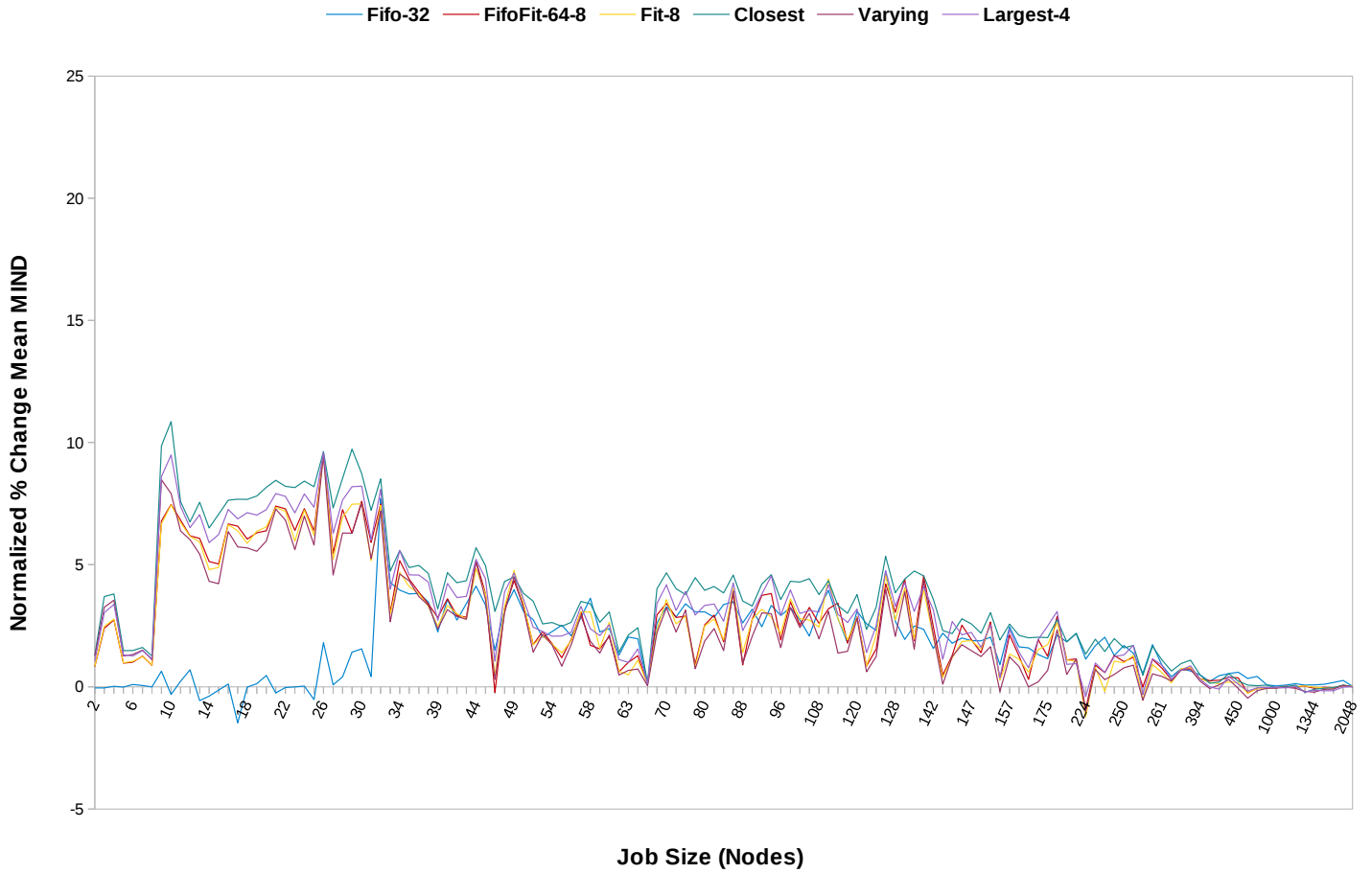


Figure 42: Normalized Improvement versus Job Size for Algorithms on the 5D Torus Topology

Small Job Sizes		Large Job Sizes		Overall	
Algorithm	Improvement	Algorithm	Improvement	Algorithm	Improvement
Closest	55.28	Closest	21.17	Closest	23.48
Largest-1	54.02	Largest-8	18.85	Largest-8	20.37
Largest-2	52.67	Largest-4	17.81	Largest-2	19.98

Table 6: Improvement Values for the Top Three Algorithms for 5D Torus Topology

7.4 Best Overall MIND Comparisons

The following graphs in Figures 43 and 44 attempt to convey the best results on the different topologies on a more equal level. Figure 43 shows the average MIND value per job size as calculated by the `Closest Fit` algorithm for each of the topologies. As can be seen, generally the larger the maximum number of hops there are in the network, the higher

the average is as the job size increases. However, interestingly, the 5D torus seems to show an equal and sometimes larger average for small to mid-sized jobs. Figure 44 then shows the actual difference between in average MIND value between the `Closest Fit` algorithm and the `FIFO-0` algorithm for each of the job sizes. Here it is much more apparent that the results for the 3D torus actually show the most improvement. At its peak, there is a decrease on average of almost 3.5 hops for a number of jobs. The next highest value is on the 5D torus with a maximum decrease of around 2 hops. As expected, as the job size increases, this value decreases (even into the negative range for the 5D torus and the worst case dragonfly) for all of the topologies.

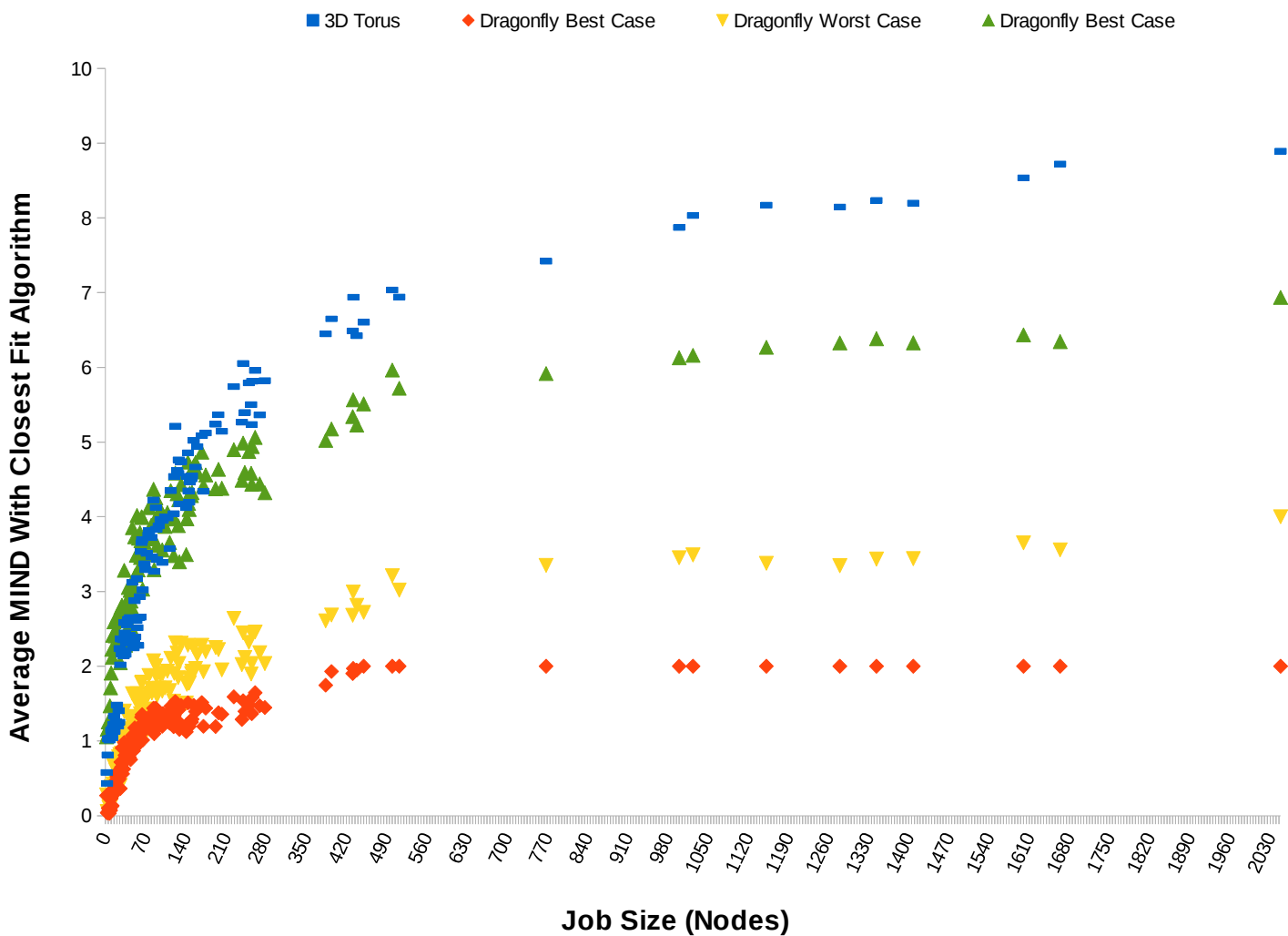


Figure 43: Average MIND Value With Closest Fit Algorithm versus Job Size

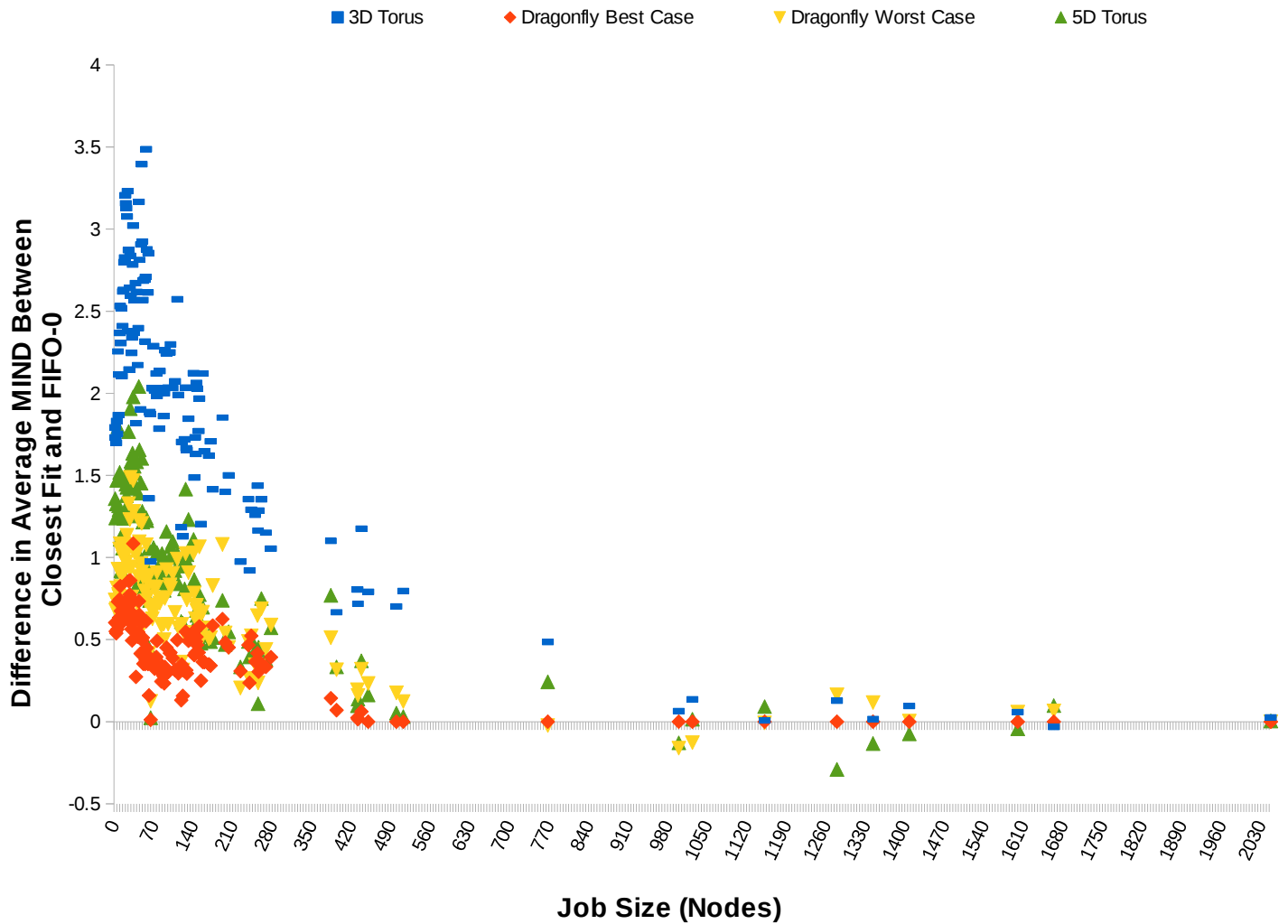


Figure 44: Average Δ MIND Value Between Closest Fit Algorithm and FIFO-0 versus Job Size

7.5 Individual MIND Comparisons

This final section shows the range of possible MIND values for the best performing (where applicable) of each of the algorithms for each of the topologies. The graphs show how the minimum, maximum and average MIND value changes as the job size increases. The first (and largest) graphs shows the original placements and the subsequent smaller graphs show the different placement values for the best performing algorithms. Interestingly, the three values for all of the algorithms tend to converge towards the highest job end. Even for the 3D torus, which has the highest range of possible MIND values of the topologies looked at, at the highest end of the job size spectrum the MIND values only vary between a minimum of 8 and a maximum of 9. For the best case dragonfly network, these values converge completely at 2 past job sizes of 450, where results previously showed no improvement could be seen. This is because as jobs grow in size, the number of possible placements decreases.

The y-axes in the graphs below for the different topologies span from the minimum to the maximum number of hops on the topology. So for the 3D torus, the y-axis spans from 0-18. Although the dragonfly network showed the most improvement, this is probably because there are fewer hops to traverse. Where the difference between the one hop and the next number of hops is %25 on the dragonfly, on the 3D torus it is %5. Looking at the graphs that follow, there is much more of a spread of the averages on the dragonfly graphs in comparison to the toruses, which would indicate that actually there is less consistent improvement overall for the dragonfly topology.

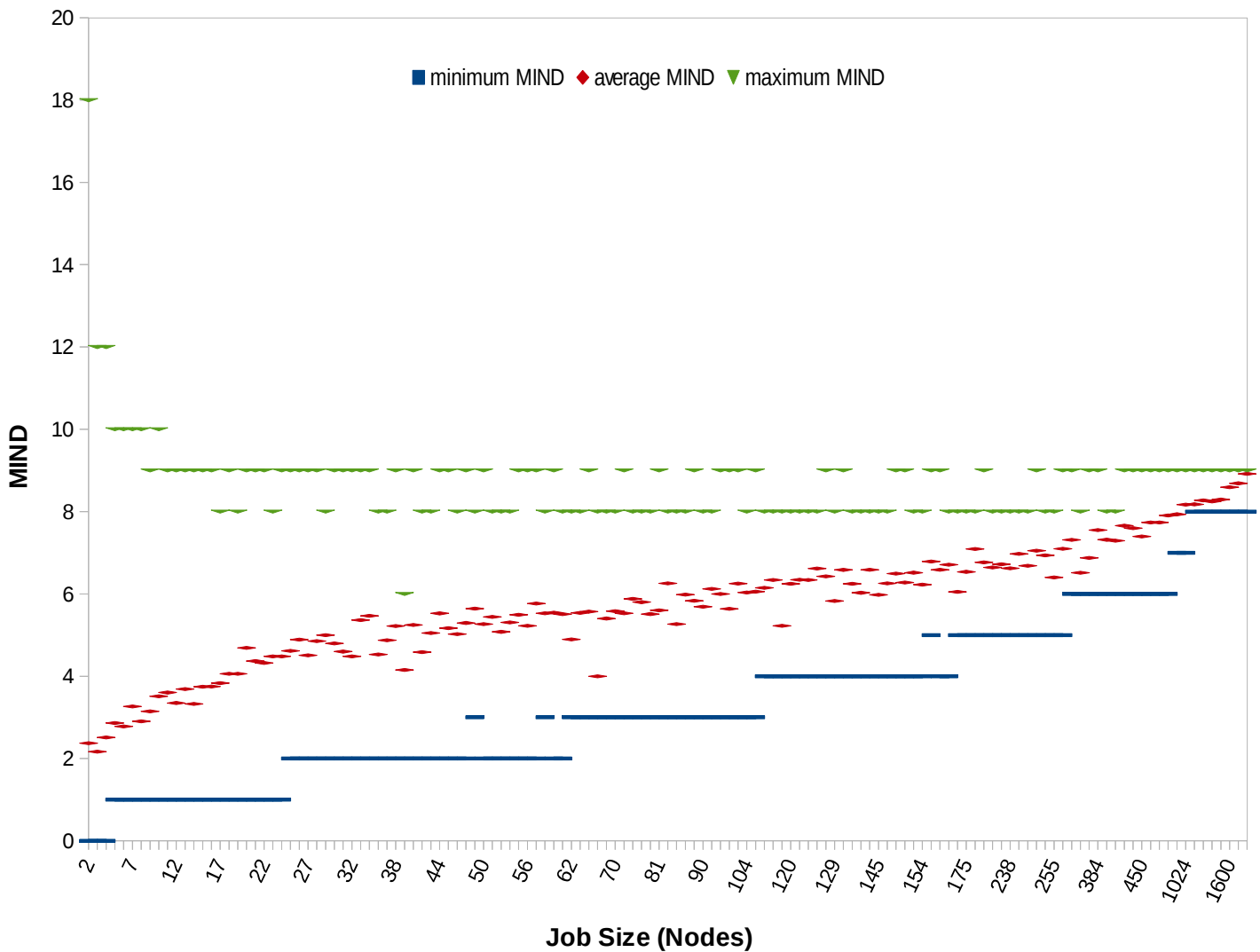


Figure 45: Minimum, Maximum and Average MIND Values versus Job Size for 3D Torus Topology Using the Original Job Allocation Algorithm

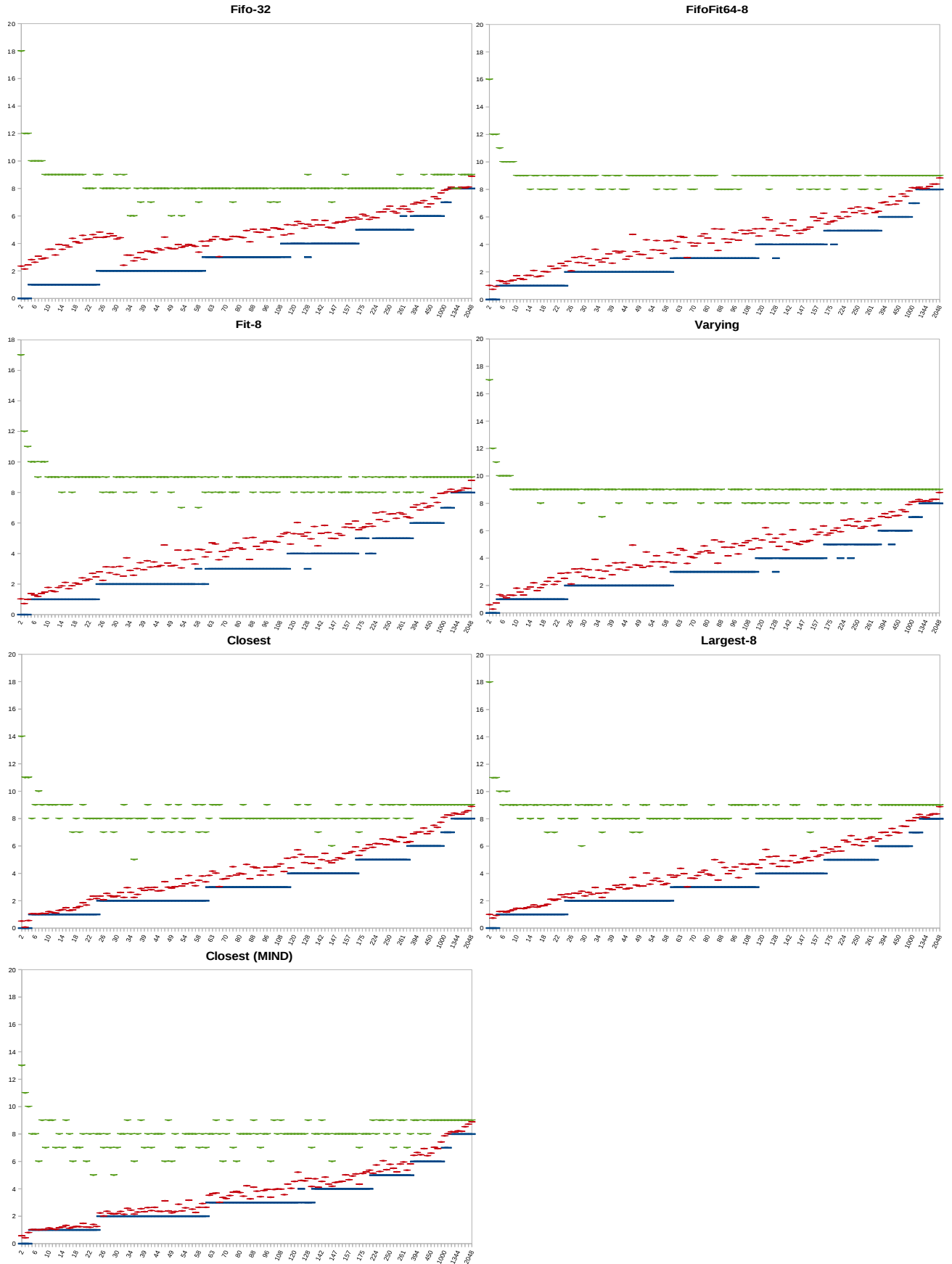


Figure 46: Minimum, Maximum and Average MIND Values versus Job Size for 3D Torus Topology Using Various Job Allocation Algorithms

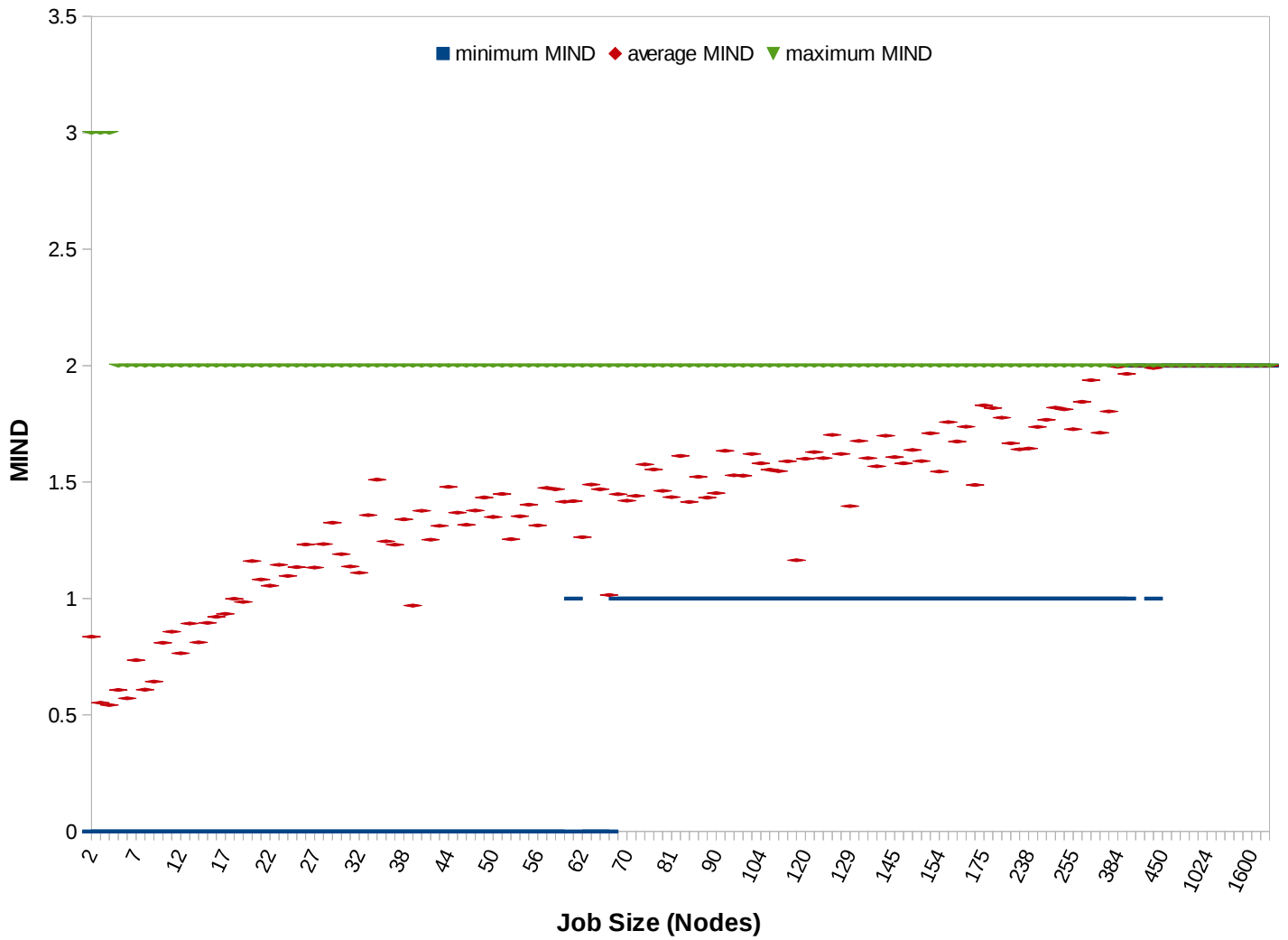


Figure 47: Minimum, Maximum and Average MIND Values versus Job Size for Best Case Dragonfly Topology Using the Original Job Allocation Algorithm

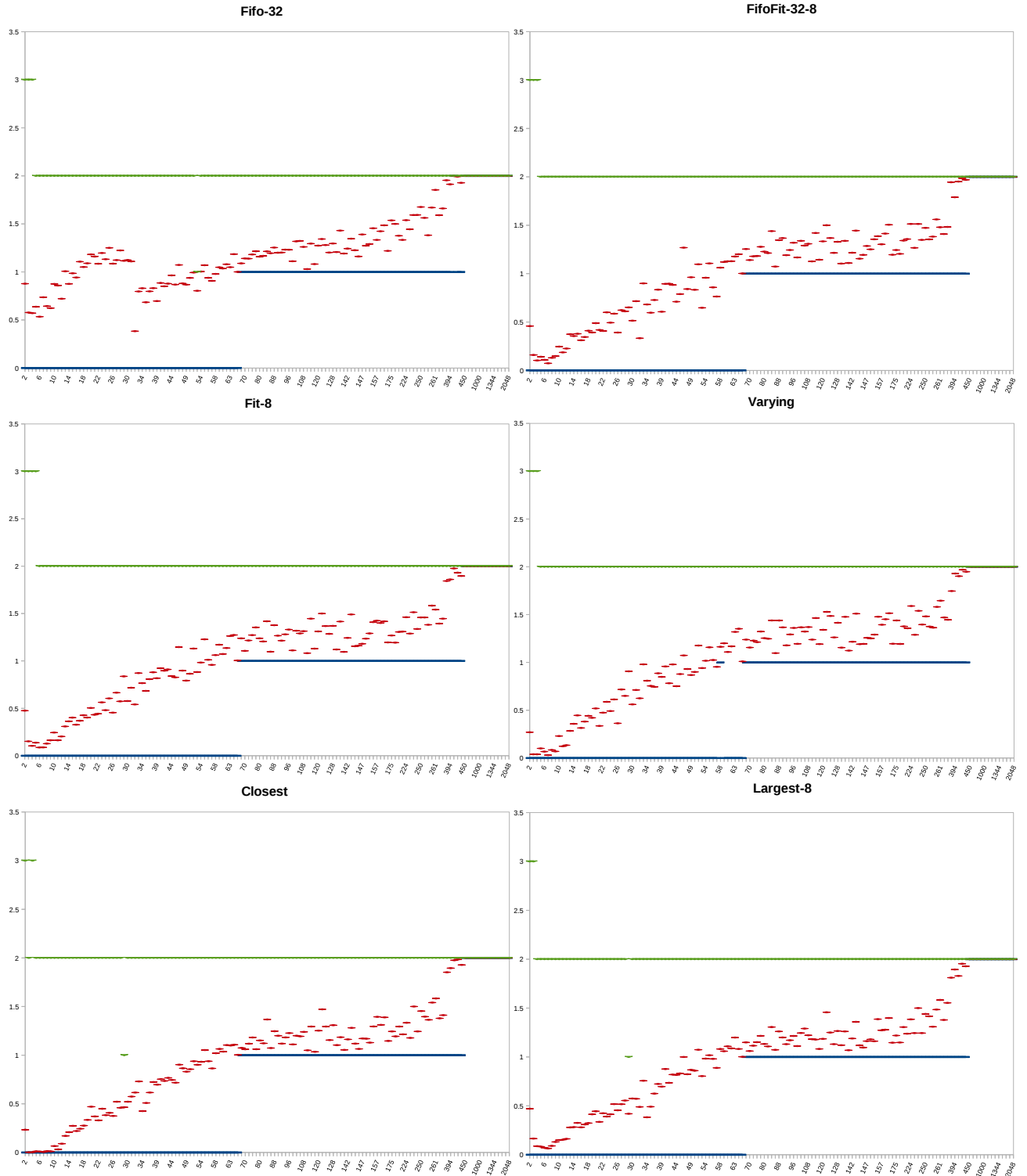


Figure 48: Minimum, Maximum and Average MIND Values versus Job Size for Best Case Dragonfly Topology Using Various Job Allocation Algorithms

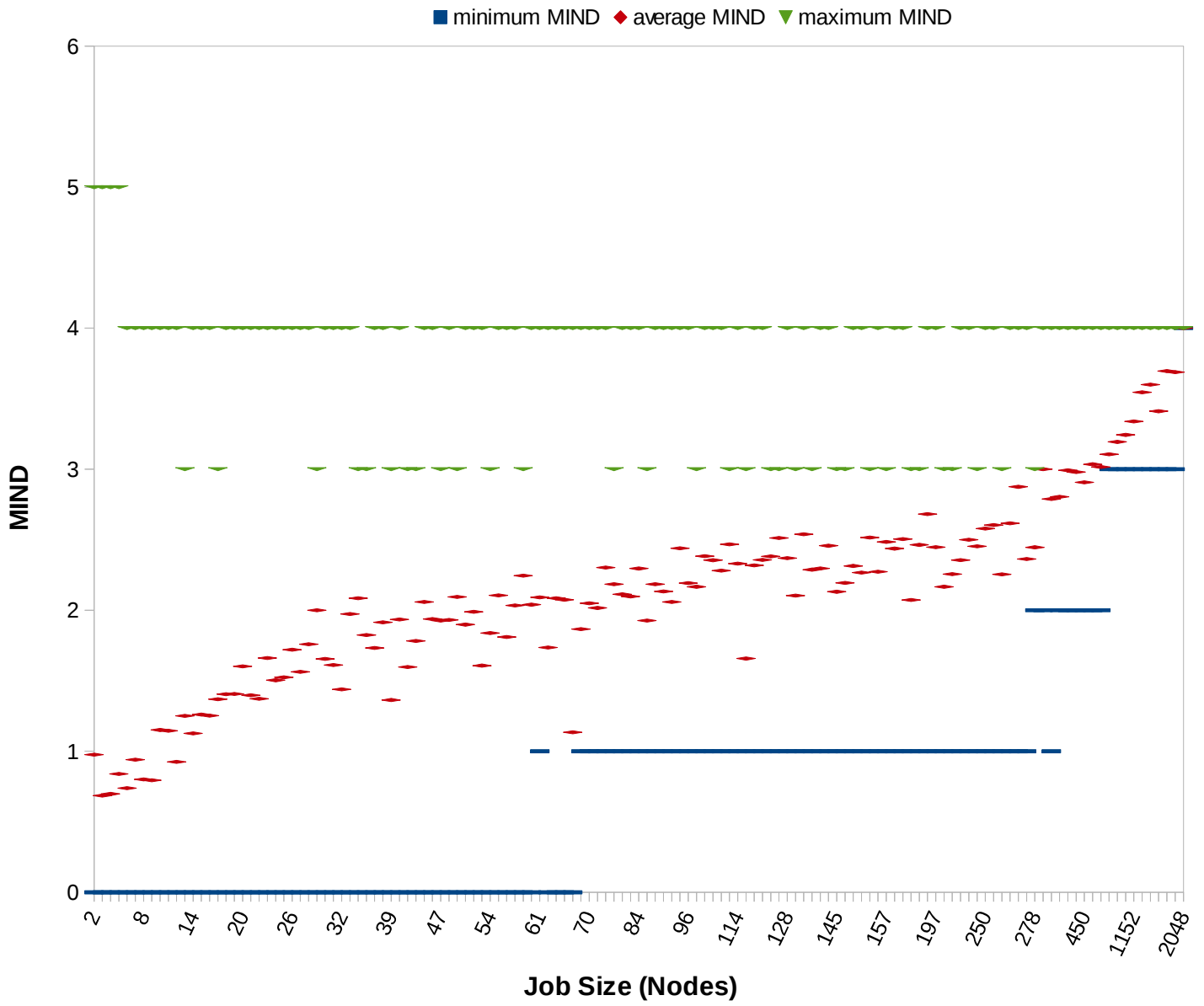


Figure 49: Minimum, Maximum and Average MIND Values versus Job Size for Worst Case Dragonfly Topology Using the Original Job Allocation Algorithm

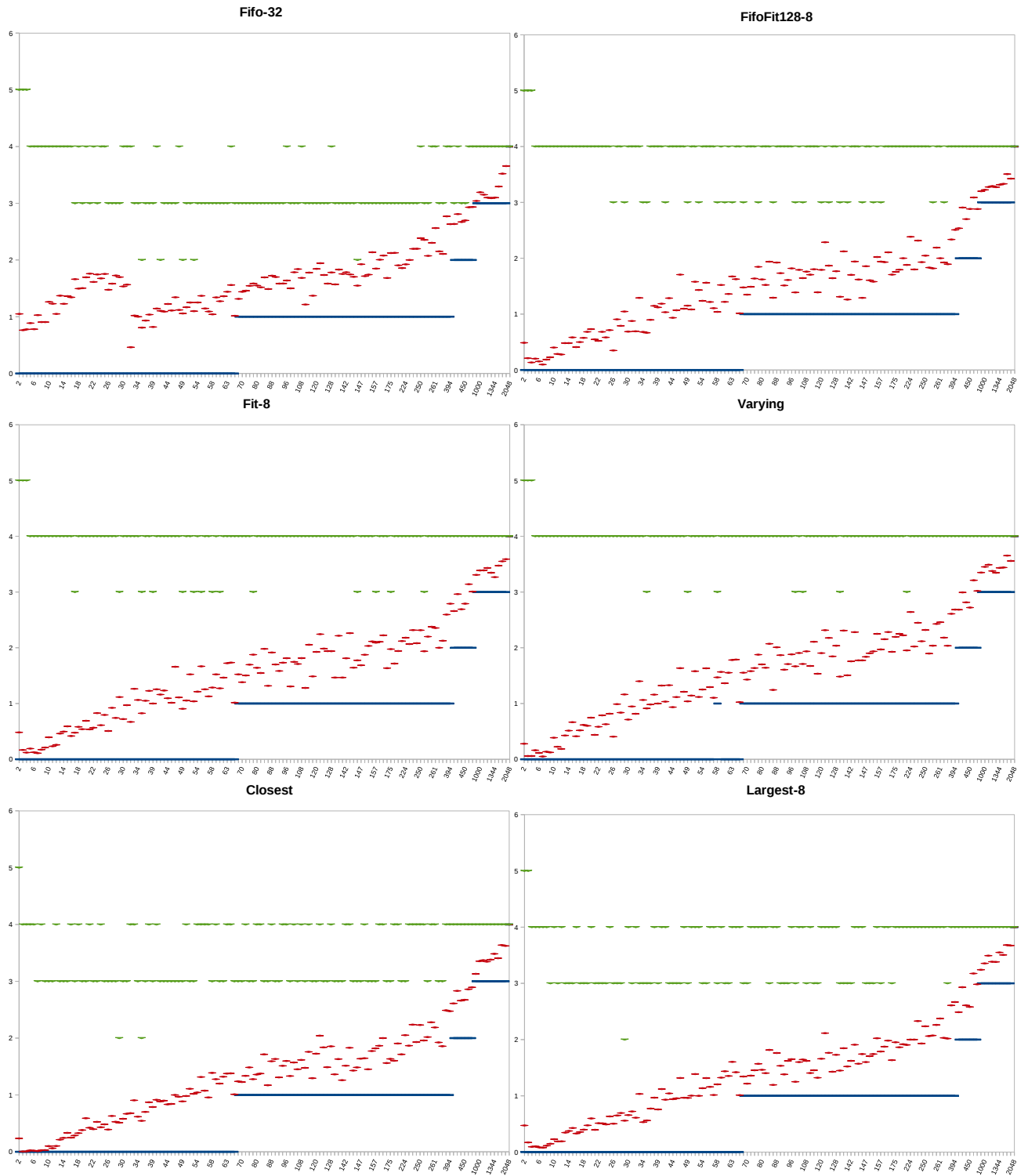


Figure 50: Minimum, Maximum and Average MIND Values versus Job Size for Worst Case Dragonfly Topology Using Various Job Allocation Algorithms

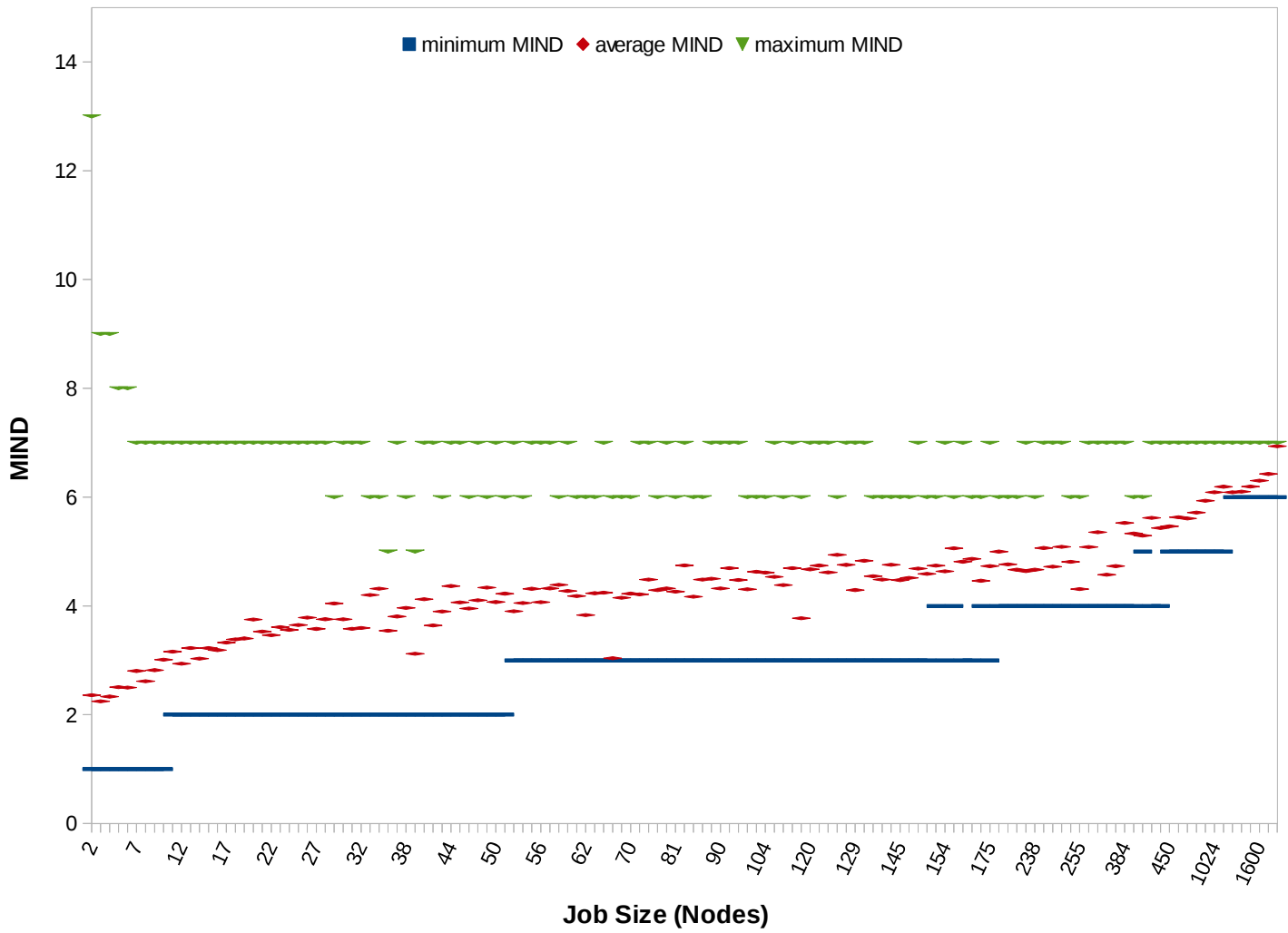


Figure 51: Minimum, Maximum and Average MIND Values versus Job Size for 5D Torus Topology Using the Original Job Allocation Algorithm

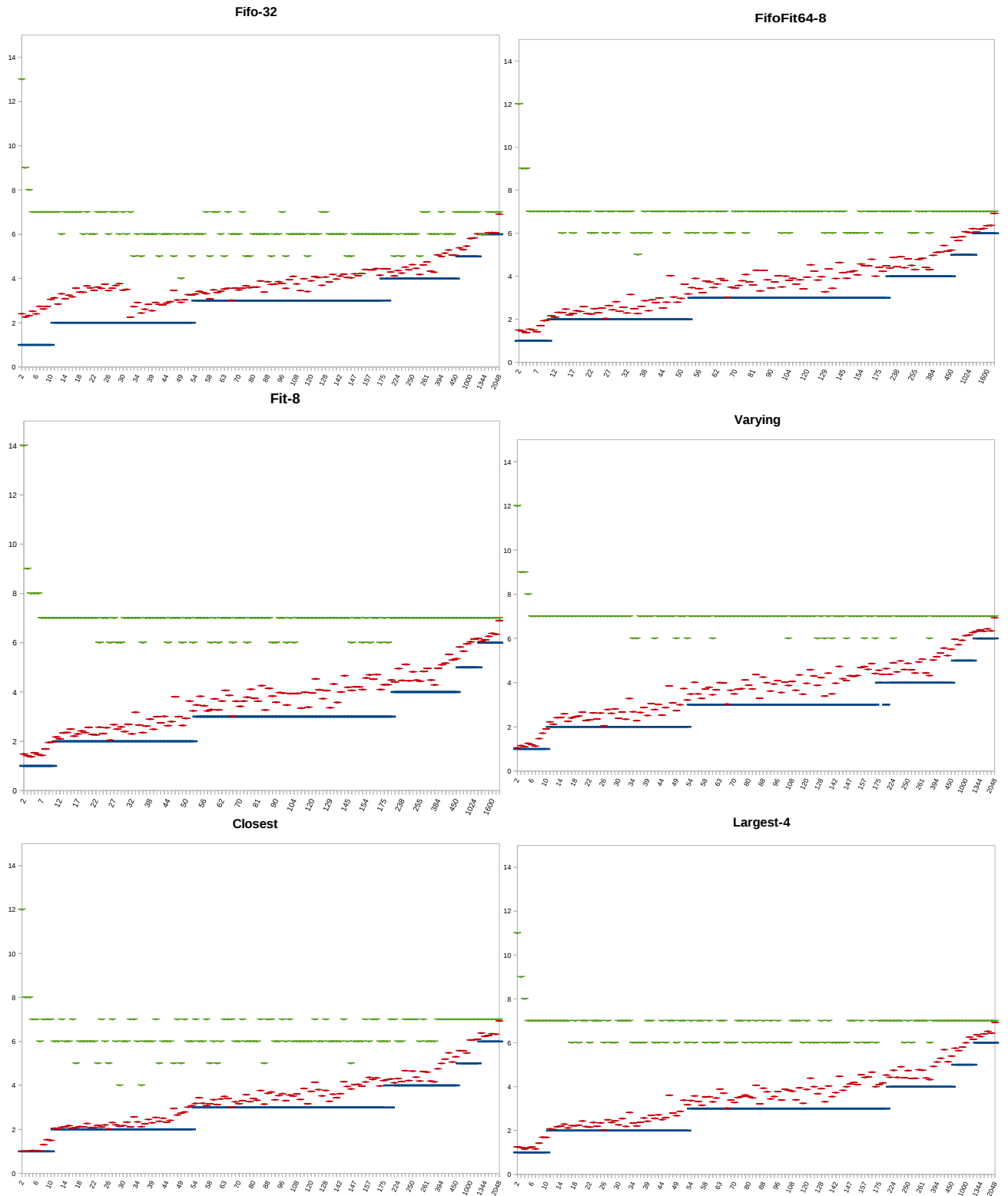


Figure 52: Minimum, Maximum and Average MIND Values versus Job Size for 5D Torus Topology Using Various Job Allocation Algorithms

8 Conclusions

This project had three main goals to tackle and succeeded in producing results for all of them. The first aim was to compare the results of running the `First Fit` and `FIFO` algorithms on HECToR data to what was found on the NERSC machine. The second part was to create some new job allocation algorithms and try them out on the HECToR data and see if there was any improvement over the two original algorithms, in particular looking for algorithms which would perform well for both small and large jobs. Finally, new mock topologies were to be created with the same number of nodes as HECToR has to see what kind of effect all the algorithms would have on them. Overall, the results seemed positive in favor of using job allocation algorithms to increase compactness - and in turn performance - for jobs run on supercomputing clusters. However, the results were also somewhat varied between different machines and topologies and showed some complications as well as benefits, particularly for highly-connected topologies.

From looking at the improvement values of the NERSC data in comparison to HECToR's data, it is clear that even on a machine with the same architecture (albeit the NERSC machine had 24 cores per node versus the 32 cores per node on HECToR), the job distribution makes a substantial difference in how well job allocation algorithms perform. Overall the HECToR data performed better than the NERSC for both algorithms. The main idea to take away from this comparison is that job node scheduling optimisations should be done on a per machine basis. The simulator conveniently allows this to be done without the actual operation of a production machine.

All of the algorithms run showed some improvement on the 3D torus, albeit some more than others. None of the algorithms performed as well for large jobs as for small jobs, although this is likely because it is impossible to see as much improvement at the higher end (this is particularly apparent in the MIND comparison graphs in Section 7.5). The most promising algorithm, that is the `Closest Fit` using minimized maximum consecutive distance, did not perform all that much worse than the algorithm that actually minimized the MIND values between nodes. The overall improvement seen was $\sim 30\%$. To simulate an entire year's worth of placements for this algorithm took less than a minute and a half, which for use on a day-to-day basis does not seem unreasonable. Additionally, one of the biggest things to take away from this work is that it does not take much to see a little improvement. It took tremendous effort (ie. high values of gap sizes) to break all but the `FIFO` algorithm. So even where the two preliminary algorithms did not perform as well as the subsequent ones, an average of even a few percent improvement overall still might be worth implementing if improvement in performance could be seen regularly (particularly considering the pains many programmers go through to optimise their own codes). It is interesting to note as well

that for many of the algorithms run on the 3D torus, improvement can still be seen at the very high end of the job size scale, even at the very highest job size reported. This is clearly not because the highest job found a much better fit, but as a side effect of all of the other smaller jobs finding better fits.

Running the HECToR log data on the different topologies also showed some interesting results. Even on the best case scenario for the dragonfly, a highly connected network, improvements of over %35 could be seen overall, although this was only up to the threshold of 450 nodes. For the worst case scenario on the dragonfly network, the highest improvement was slightly higher at ~%40 although this improvement reversed for very large job sizes. For the 5D torus, results were similar (but smaller in magnitude - upwards of ~%21 overall) to the 3D torus overall and for large job sizes, although responded similarly to the algorithms too. This ability to model different topologies using a real workload could help make more informed decisions when procuring HPC machines.

Topology	Small Job Sizes		Large Job Sizes		Overall	
	Algorithm	Improvement	Algorithm	Improvement	Algorithm	Improvement
3D Torus	Closest	71.28	Closest	27.95	Closest	30.88
Best Case Dragonfly	Closest	94.99	Closest	30.87	Closest	35.21
Worst Case Dragonfly	Closest	94.66	Closest	36.67	Closest	40.59
5D Torus	Closest	55.28	Closest	21.17	Closest	23.48

Table 7: Top Algorithms By Topology

As the percentage of increase in improvement values are not exactly comparable across the topologies, the values of the average change in MIND value (between the best allocation algorithm and no allocation algorithm) for each topology are shown in Table 8. Here it is clear that the 3D torus shows the most improvement across the entire job distribution with a decrease on average of 1.89 hops, followed by the 5D torus and then the worst and best case dragonfly topologies. Although the change is not large for the best case dragonfly network (with the smallest average decrease) the results are still significant. An improvement of ~0.4 hops in compactness out of a maximum of 3 hops can be seen, which could still very well translate into performance increases overall across the board.

Topology	Average Δ MIND Between Closest Fit Algorithm and FIFO-0
3D Torus	1.89
Dragonfly Best Case	0.424
Dragonfly Worst Case	0.702
5D Torus	0.901

Table 8: Comparisons Between the Topologies of Actual Change in MIND Values

In conclusion, results from this project have shown that the effectiveness of the compactness of job placement algorithms are dependent on:

- job size distributions of a machine
- a particular job size (ie. above or below average)
- a machine's topology

For particular algorithms (in particular, the `Closest Fit`) and topologies, there is also suggestion that these algorithms are dependent on job node ordering as well. However, further investigation there is outside the scope of this project. Overall, job node allocation algorithms show promise for varying machines and topologies for improving performance overall of a distribution of jobs being run.

9 Future Work

A continuation of this project could go in a number of directions. The first step would be to actually try out some of these algorithms on a real job scheduler as opposed to the simulator and see if the results corroborate. From there, actual performance timings could be run to see if any of the algorithms could feasibly be used regularly, as a slow allocation algorithm could undo all the benefit of a compact placement. Then, performance tests of benchmarks could be run to determine the relationship between actual performance and compactness.

Another idea would be to try to optimise some of the algorithms themselves. In particular, the `Closest Fit` with `MIND` calculation algorithm showed great potential, but with plenty of room for improvement in timing. The timing could be decreased by finding a cheap way to parallelise it (ie. using a streaming processor). Alternatively, it might be possible to find a more efficient algorithm for the calculation or even a different but similar metric. Additionally, other algorithms, variations or combinations could also be tried to see if any outperform that one. One thing that was not expounded upon much in this paper is the issue of node ordering. The `Closest Fit` algorithm in particular might benefit from a more optimal node ordering on any of the topologies. Finally, the issue of the effect of gap sizes on the dragonfly network could be explored more to see if there were a more effective dynamic use of this type of parameter. As well as this, particularly in the case of the worst case dragonfly network (and also possibly the 5D torus), a more “aware” algorithm would likely show better improvement than the static ones explored here. That is, it would be interesting to develop an algorithm with an awareness of where the bottlenecks in placement were, for example where the single connection across groups in the worst case dragonfly were located.

With regards to long-term planning, the simulator could also be modified to look at the effects of different job placement algorithms for a “straw-man” exascale machine. Job sizes could be scaled up by orders of magnitude of the tens of thousands and their placements on different topologies could be simulated. The performance of different placement algorithms becomes increasingly important as the number of nodes increases to millions (and beyond), so this would be worth investigating too. This is an aspect of exascale which has not been explored and might benefit from further investigation.

Glossary

dragonfly	network topology with all-to-all mapping across three levels of node connections and dynamic routing
FIFO	First In First Out - when run with “0,” this indicates no optimisation used
fragmentation	where jobs on the network are placed in disparate locations, as opposed to on one contiguous chunk of nodes)
HECToR	High-End Computing Terascale Resource, the Cray XE6 UK national supercomputer
hop	metric of number of jumps required to get from one node to another node on the network
improvement	percentage change MIND average between new algorithm and FIFO-0 algorithm
kAU	measure of CPU time used by a job
large job size	ranges from 10-2048 nodes
MIND	Mean Inter-Node Distance
NERSC	National Energy Research Scientific Computing Center in California, housing another Cray XE6 machine
normalized improvement	improvement value multiplied by normalized kAU values to show which jobs are seeing more improvement with regard to how much compute time they are using
small job size	ranges from 1-10 nodes
torus	network topology with wrap around dimensions (3 and 5 for this project)

References

- [1] Albing, Carl (2012). *Improving Node Allocation for Application Placement in High-Performance Computing Systems*. PhD Thesis, University of Reading, UK.
- [2] UoE HPCX Ltd. *HECToR User Guide*. Document. Available at: <http://www.hector.ac.uk/support/documentation/userguide/>. Last accessed: 6th March 2013.
- [3] SAFE for HECToR. *Hector Job Status*. Website. Available at: https://www.hector.ac.uk/safe/static/HECToR_stat_p2b.html. Last accessed: 16th August 2013.
- [4] HECToR: UK National Supercomputing Service. *HECToR » Gallery*. Website. Available at: <http://www.hector.ac.uk/about-us/gallery/>. Last accessed: 16th August 2013.
- [5] Zargham, Mehdi (1996). *Computer Architecture Single and Parallel Systems*. Prentice Hall International Editions.
- [6] Wanqian Liu, V. et al (1994). *Non-contiguous processor allocation algorithms for distributed memory multicomputers*. Proceedings of the 1994 conference on Supercomputing, IEEE Computer Society Press.
- [7] Weisser, Deborah et al. *Optimizing Job Placement on the Cray XT3 CUG Proceedings 2006*, Pittsburgh Supercomputing Center.
- [8] Qiao, Wenjian and Lionel, M. *Efficient Processor Allocation for 3D Tori* Michigan State University.
- [9] Wilson, P. et al (1995). *Dynamic storage allocation: A Survey and Critical Review*. Lecture Notes in Computer Science.
- [10] Knuth, Donald (1973). *The Art of Computer Programming, Volume 1* Addison-Wesley, Massachusetts.
- [11] S. Martello, D. Pisinger, and D. Vigo (2000). *The three-dimensional bin packing problem*. Operations Research.
- [12] Christopher Rose (1992). *Mean Internodal Distance in Regular and Random Multihop Networks*. IEEE Transactions on Communications, Vol 40, no 8.
- [13] NERSC. *NERSC's Hopper Cray XE6 System*. Website. Available at: <http://www.nersc.gov/systems/hopper-cray-xe6/>. Last accessed: 10th August 2013.
- [14] HECToR Annual Reports. Document. Available at: <http://www.hector.ac.uk/about-us/reports/annual/>. Last accessed: 19th March 2013.

- [15] Alverson, B. et. al (2012). *Cray XC Series Network*. Pamphlet, Cray Inc.
- [16] Wisniewski, Robert (2012). *BlueGene/Q: Architecture, CoDesign; Path to Exascale*. Presentation. Available at: <http://projects.csail.mit.edu/caos/2012-01-25-caos-bgq-v1-ed.pdf>. Last accessed: 10th August 2013.
- [17] Fujitsu. *Japan's Next-Generation Supercomputer Configuration is Decided*. Document. Available at: <http://www.fujitsu.com/global/news/pr/archives/month/2009/20090717-01.html>. Last accessed: 10th August 2013.
- [18] Press, W. et al. *Numerical Recipes*. Cambridge: Cambridge University Press, 2007.
- [19] TestNG. *TestNG*. Website. Available at: <http://testng.org/doc/index.html>. Last accessed: 10th August 2013.
- [20] Edwards, Tom. Cray at EPCC. Personal Communication. August 2013.