Table of Contents

Abstract

Divisible Load Theory (DLT) is an effective tool for blueprinting data-intensive computational problems; it is a mathematical framework that is applied using Divisible Load Scheduling (DLS) algorithms, but has yet to be proved optimal in all cases. A heuristic algorithm known as ITERLP rapidly produces near-optimal divisible load schedules with result collection. Though the results are near-optimal, there are some underlying issues that did not get addressed. These issues include: scheduling divisible loads on cluster networks that share identical network bandwidth speeds, and neglecting the cost of latency. ITERLP performs well when processor nodes have varying network bandwidth; however, it may pose a problem when each processor node shares the same bandwidth speed but vary in computation speed. The second problem is that ITERLP does not factor in the cost of latency; by taking the solution time calculated using ITERLP and factoring in latency, it is shown that the time is adversely worsened in all cases. In this thesis, a few modifications are made to the ITERLP algorithm, followed by numerous tests to further investigate more realistic cases. These modifications involve factoring in latency as well as trying different sorting algorithms for communication, computation and latency parameters. It is shown when latency is factored in that ITERLP2 produces near-optimal times and outperforms ITERLP in all realistic cases. It is also shown that some sorting algorithms perform better than others in certain cases; however, no sorting algorithm is best suited for all cases.

Chapter 1

Introduction

1.1 Background

A divisible load is a computable entity that can be arbitrarily partitioned into an infinite number of independent load fractions. Once partitioned, load fractions may then be allocated to a set of computer processor nodes to be computed and then collected back. A few applications that fulfill the divisibility property include: simulations, matrix computations, database searching, Kalman filtering, as well as dataset, image, and signal processing [1, 2, 3, 4, 5]. Loads that are randomly or arbitrarily partitioned do not always achieve the optimal solution and may, by consequence, result in a worse time than that of a single processor node alone; so to properly partition a divisible load, there needs to be a Divisible Load Schedule (DLS) to ensure that tasks can be run as efficiently as possible in parallel on the given resources. Divisible Load Theory (DLT) is an effective tool for blueprinting data-intensive computational problems; it is a mathematical framework that is applied using Divisible Load Scheduling (DLS) algorithms, but has yet to be proved optimal in all cases [1, 4, 5].

DLT has been becoming more popular in recent years through the use of volunteer computing; users may volunteer their PC's computational power to help form a grid network used to perform numerous calculations in parallel [4]. SETI (Search for Extraterrestrial Intelligence) has a volunteer computing network named SETI@home that is used for processing enormous amounts of signal data. Initially, SETI did not possess the resources to build or buy a computer powerful enough to process the signal data, so they compromised the fact that there was no convenient access to a supercomputer and established a volunteer computing network to allow users to enlist their home computers to do the job [6].

On the other side of the parallel computing spectrum, UNB Fredericton, as well as eight other Atlantic Canadian universities are members of the Atlantic Computational Excellence Network (ACEnet); this network provides multiple clusters for the purpose of High Performance Computing (HPC) and research [7]. Cluster networks are generally used as a resource of computer processors designed for parallel processing. A reasonable assumption is that bandwidth and latency are constant between all processors in a given cluster, however, computational power of each processor may vary.

The simplest network model used in DLT is the star network model with one-port communication; it has a master processor in the center with the load to be distributed, and a set of worker processors connected via the points. One-port communication enforces that only one transmission is allowed at a given time between the master and any of the associated worker processors [1]. This ensures that no overlapping of communication occurs in either the allocation of load, or the collection of results; however, when two or more worker processors have already received their allocated loads, they may compute them independently in parallel. Another simplification applied is the one-round approach; although the multi-round approach allows for pipelining multiple sets of smaller messages, the cost of latency would be too great [8, 9]. The one-round approach ensures that a divisible load is split up only once and that latency occurs only twice for each load fraction, once for allocation, and once for result collection.

All nodes Finish Simultaneously (AFS) is a policy that assumes no idle times occur and all processor nodes participate in the optimal solution [1, 4, 5, 8, 9, 10]. The AFS policy cannot be applied because it is unrealistic and oversimplified; though it would be simple to assume that all processor nodes finish simultaneously, it is more important for them to finish at various time slices to ensure that the idle time between communication of results is minimized, and that collection can actually occur. A few papers that do assume the AFS policy use it to help simplify describing DLT and conducting proofs [9, 10, 11].

Existing algorithms for scheduling divisible loads with result collection in this model include FIFO (First In First Out), LIFO (Last In First Out), and OPT (Optimal Solution). Each algorithm uses a linear program for a given allocation and collection order to solve a minimization problem with several constraints. The solution to the linear program gives all load fractions for each processor node considered, as well as the optimal time [1, 4, 5]. The FIFO algorithm uses the same order for both allocation and collection, whereas the LIFO algorithm has the collection in reverse order of the allocation. The OPT algorithm finds the optimal solution by permuting all possible combinations of allocation and collection for m processor nodes. By solving a linear program for each permutation,

the optimal load fractions and times are found. In most cases the optimal solution is not always attainable in practice, knowing that for m processor nodes, there would be a possible $(m!)^2$ linear programs to be solved [1, 5, 8]. So to alleviate the time taken to find the optimal solution, heuristic algorithms have been proposed [1, 4, 5, 8, 9].

A recent DLS heuristic algorithm known as ITERLP was proposed in [1], April, 2008. It rapidly produces near-optimal divisible load schedules including result collection. Many simulations were performed to compare ITERLP, FIFO, and LIFO to the optimal solution. It was found that ITERLP invariably produces nearest-optimal performance no matter the level of heterogeneity of the system, the amount of processor nodes, or the size of the allocated and collected data [1]. Though the results are near-optimal, there are some underlying issues that did not get addressed. These issues include: scheduling divisible loads on cluster networks that share identical network bandwidth speeds, and neglecting the cost of latency. ITERLP performs well when processor nodes have varying network bandwidth; however, it may pose a problem when each processor node shares the same bandwidth speed but vary in computation speed. The second problem is that ITERLP does not factor in the cost of latency; by taking the solution time calculated using ITERLP and factoring in latency, it is shown that the time is adversely worsened in all cases. Another recent heuristic algorithm known as SPORT was proposed in [5], August, 2008.

1.2 Heuristic Algorithm

A heuristic algorithm known as ITERLP was used in [1] to quickly calculate near-optimal divisible load schedules with result collection. ITERLP is heuristic in manner since it often leads to a near-optimal solution and computes much more quickly than OPT. Although the solution from ITERLP is not strictly optimal, it occasionally finds the optimal solution. To compare the performance of ITERLP to OPT, ITERLP only requires solving up to m^2 linear programs in a given iteration, as opposed to the OPT which solves $(m!)^2$, when given m processor nodes. Although ITERLP is able to produce near-optimal divisible load schedules, there are some underlying issues that did not get addressed. These issues include: limitations of sorting processor nodes only by communication parameters, and neglecting the cost of latency. ITERLP performs well when processor nodes vary in communication parameters; however, it may pose a problem with cluster networks that have processor nodes sharing identical communication parameters. ITERLP does not factor in the cost of latency because divisible loads are assumed to be sufficiently large enough to ignore it in [1, 4]. Experimental results from taking the optimal time calculated by ITERLP

then factoring in the cost due to latency will show that the solution is adversely worsened and not always near-optimal.

1.3 Motivation and Goals

The last paragraph from [1] mentions future work to create an algorithm with similar performance, but with better cost characteristics than ITERLP. This thesis investigates improving the ITERLP algorithm by adding latency parameters as well as trying different methods for sorting processor nodes in a new heuristic algorithm named ITERLP2. The limitations of ITERLP are shown when latency is factored in, and when communication parameters are the same for all processor nodes. Multiple scenerios with cluster experiments will show different cases where some sorting algorithms are best suited. Finally, results from experimentation are expected to show that ITERLP2 outperforms ITERLP in all realistic cases.

1.4 Approach

The ITERLP algorithm was implemented using C++ and a linear program solver (COIN-Clp) [12]. The algorithm was modified to accomodate latency using two different methods, one that adds latency into the equations to be solved in the linear program, and another that factors in latency after the linear programs have been solved. Rather than sorting only by communication parameters, other sorting algorithms for computation, communication then computation, and latency were implemented. Experiments with the new algorithm ITERLP2 were applied using different configurations and orderings for communication, computation, and latency parameters to show that the goals are met.

Chapter 2

Theory

2.1 System Model and Problem Definition

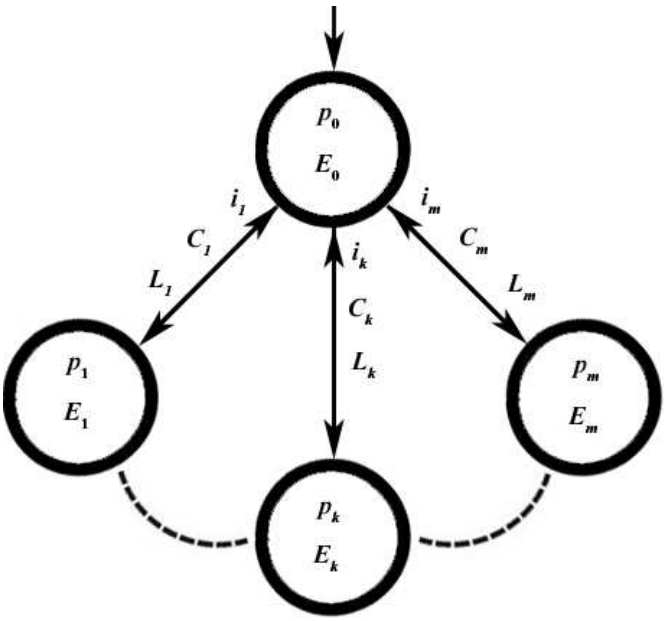


Figure 2.1: Heterogeneous star network $\mathcal{H}$.

Consider the divisible load $\mathcal{J}$ to be allocated, computed and collected on a heterogeneous star network $\mathcal{H} = (\mathcal{P}, \mathcal{I}, \mathcal{E}, \mathcal{C}, \mathcal{L})$ illustrated in Fig. 2.1. The set of m + *1* processors is denoted by $\mathcal{P} = \{p_0, ..., p_m\}$, and the set of m network links denoted by $\mathcal{I} = \{i_1, ..., i_m\}$; the links stem from the master processor p_0 to the worker processors $p_1, ..., p_m$. The set of computational parameters for each of the worker processors is $\mathcal{E} = \{E_1, ..., E_m\}$, and the set of communicational parameters for each of the network links is $\mathcal{C} = \{C_1, ..., C_m\}$. The new parameter for the latency on each of

the network links is $\mathcal{L} = \{L_1, ..., L_m\}$. The value for C_k is the reciprocal of the bandwidth speed of network link i_k, and E_k is the reciprocal of the processor speed of processor p_k. The values for E_k and C_k are measured in time units per unit load; E_k represents the time to process a unit load, and C_k represents the time to transmit a unit load from p_0 to p_k over network link i_k. The value for L_k measures the latency (total time taken in time units to initiate a connection) between processor p_0 to p_k over network link i_k. For non-negativity, it is assumed that $\forall k \in \{1, ..., m\}$ that $E_k > 0, C_k > 0$, and $l_k \geq 0$. This new system model is based upon the model from [1].

The values of $\mathcal{C}$ and $\mathcal{E}$ are already known ahead of time by the master processor p_0. Using these known values, processor p_0 calculates and divides up the entire load $\mathcal{J}$ into a set of load fractions $\alpha_1, ..., \alpha_m$; once divided, processor p_0 allocates the load fractions $\alpha_1, ..., \alpha_m$ to the respective worker processors $p_1, ..., p_m$ to be computed.

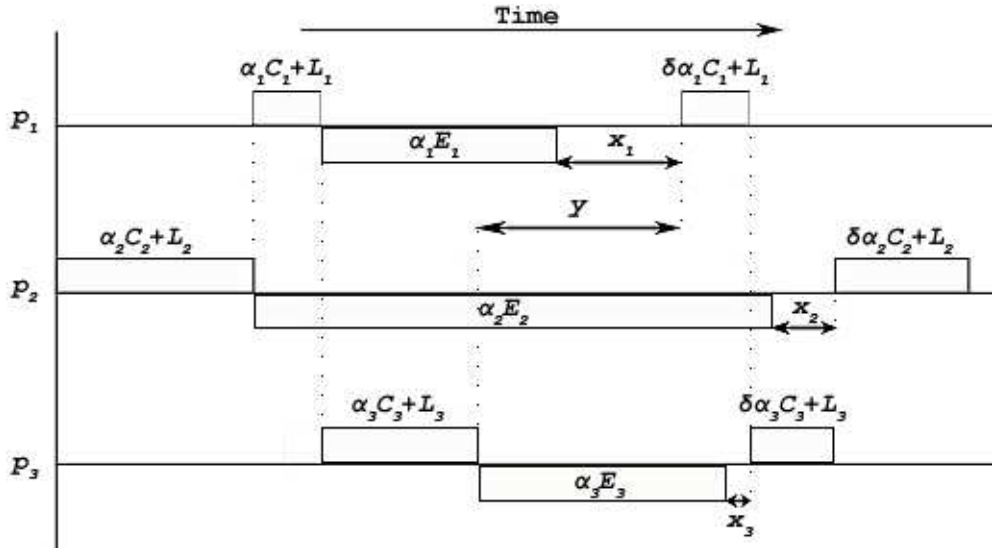


Figure 2.2: Divisible load schedule for $m = 3$, $\sigma_a = \{2, 1, 3\}$, $\sigma_c = \{1, 3, 2\}$.

Each of the worker processors goes through three discrete phases: allocation, computation, and collection. To begin, the allocation phase involves the master processor p_0 sending load fractions to a worker processor; the computation phase commences when the entire load fraction has been allocated to a worker processor. After having finished computing, the collection phase has worker processors send results back. Whether it is the allocation or collection phase, the master processor p_0 can only communicate to one processor at a time, and result collection can occur only after the entire load has been allocated; however, any number of worker processors that have already received load may compute in parallel [1]. The magnitude of data in the results to be returned is considered proportional to the size of the load initially allocated. This assumption is only made when dealing

with image and video processing, matrix manipulation, or any computation that incorporates the use of linear transformations [1, 2, 4, 5]. So given a load fraction α_k, the associated returned result is equal to $\delta\alpha_k$, where $0 \leq \delta \leq 1$. The constant value for δ is application dependent and all processors share that value for a given load $\mathcal{J}$. Given a load fraction α_k, the time to transmit from p_0 to p_k is equal to $\alpha_k C_k$ plus the cost of latency L_k, the computation time on worker processor p_k is equal to $\alpha_k C_k$, and finally the time to transmit the results back from p_k to p_0 is equal to $\delta\alpha_k C_k$ plus the cost of latency L_k. For a given load fraction α_k to go through all three phases: allocation, computation, and collection, there is the associated cost of latency on both the allocation and collection phases, due to the master processor p_0 only being able to establish one communication at a time between itself and any given worker processor.

Fig. 2.2 illustrates a simple example of a divisible load schedule with 3 processors showing all 3 phases; the x values represent the idle time between when a processor p_k finishes computing, to when it begins transferring the result, and the y value represents idle time between the point when the last load fraction finished allocating to when the first result begins transferring.

Allocation and collection order are denoted by the permuted sets σ_a and σ_c respectively. $\sigma_a[k]$ and $\sigma_a[k]$ respresent the processor number at index $k \in \{1, ..., m\}$. To find the index of a given processor, $\sigma_a(l)$ and $\sigma_c(l)$ are two functions that are given a processor number and return the index. Fig. 2.2 shows an example load schedule with an order of allocation $\sigma_a = \{2, 1, 3\}$ and order of collection $\sigma_c = \{1, 3, 2\}$.

The ITERLP uses a linear cost model for finding a solution using only the parameters for communication and processing speed with respect to load fractions; however, for ITERLP2 to accomodate the cost of latency, the linear model must factor in a fixed cost for the time to initiate a connection, thus forming an affine cost model. The linear program below is a modified version from [1, 4, 5] that accomodates the cost of latency using an affine cost model.

Given a divisible load $\mathcal{J}$ and a heterogeneous star network $\mathcal{H} = (\mathcal{P}, \mathcal{I}, \mathcal{E}, \mathcal{C}, \mathcal{L})$, the allocation and collection orders (σ_a,σ_c) are tried by the linear program below and solved to find the optimal load fractions $\alpha = \{\alpha_1, ..., \alpha_m\}$, and associated time T. Divisible load scheduling on a heterogeneous star network is characterized by the linear program as below:

MINIMIZE $\zeta = 0\alpha_1 + ... + 0\alpha_m + T$

SUBJECT TO:

$$\left[\sum_{j=1}^{\sigma_a(k)} \alpha_{\sigma_a[j]} C_{\sigma_a[j]} + L_{\sigma_a[j]}\right] + \alpha_k E_k + \left[\sum_{j=\sigma_c(k)}^{m} \delta\alpha_{\sigma_c[j]} C_{\sigma_c[j]} + L_{\sigma_c[j]}\right] \leq T \qquad k = 1, ..., m \quad (2.1)$$

$$\left[\sum_{j=1}^{m} \alpha_{\sigma_a[j]} C_{\sigma_a[j]} + L_{\sigma_a[j]}\right] + \left[\sum_{j=1}^{m} \delta\alpha_{\sigma_c[j]} C_{\sigma_c[j]} + L_{\sigma_c[j]}\right] \leq T \quad (2.2)$$

$$\sum_{j=1}^{m} \alpha_j = \mathcal{J} \quad (2.3)$$

$$T \geq 0, \quad \alpha_k \geq 0 \quad k = 1, ..., m \quad (2.4)$$

The LHS (left hand side) of constraint 2.1 is comprised of the total time spent in transmitting all load fractions to processors that will receive load before processor p_k, the computational time for processor p_k, and the total time spent in transmitting all tasks back from processors that finish computing after processor p_k. The no-overlap model is satisfied if the processing time T is greater than or equal to the time given on the LHS for all m processors. The one-port model is enforced by constraint 2.2 with the LHS consisting of the lower bound of communicational time and latency cost for both allocation and collection. The LHS of constraint 2.3 ensures that the entire load is distributed amongst the processors. Non-negativity of objective variables is ensured by constraint 2.4 [1].

2.2 ITERLP Algorithm

The heuristic algorithm ITERLP finds a solution by iteratively solving linear programs. To begin, the processors are first sorted by communication parameters C_k. Starting with the first two processors, all four permutations of σ_a and σ_c are passed into the linear program, defined by constraints 2.1 through 2.4, and solved. After solving for the first two processors, the next processor is added to the optimal sequence and the linear program is solved again; additional processors are arranged in any position in the optimal sequence of σ_a and σ_c from the last iteration. When solving the linear program, ITERLP ignores latency and keeps track of times before latency is added; however, when ITERLP is finished computing, the solution time is adjusted for latency. The algorithm requires at most k^2 linear programs solved in a given iteration with k processors.

Using the summation below, the total complexity of ITERLP, in the worst case, is calculated to be $O(m^3)$ for m processor nodes. OPT permutes all possible combinations of allocation and collection for k processors, solving $(k!)^2$ linear programs. ITERLP will halt execution will halt and not proceed to the next iteration if, in any given iteration, processor k is allocated zero load.

$$\sum_{k=2}^{m} k^2 = \frac{1}{6}k(k+1)(2k+1) - 1 = O(m^3) \tag{2.5}$$

2.3 ITERLP2 Algorithm

Similarly to the heuristic algorithm ITERLP, ITERLP2 finds a solution by iteratively solving linear programs. The complexity of ITERLP2 is the same as ITERLP, $O(m^3)$. To begin, processors are first shuffled in random order to ensure that parameters are sorted fairly. ITERLP2 uses three new sorting configurations as well as sorting by increasing value of C_k (increasing time needed to transfer a load fraction); these new sort methods are used for showing the limitation of ITERLP sorting only by C_k and for comparing to see where each is best suited. The new methods include: sorting by increasing value of C_k then increasing value of E_k (increasing time needed to compute), sorting by increasing value of E_k, and sorting by increasing value of L_k (increasing time needed to establish a connection). After solving for the first two processors, the next processor is added to the optimal sequence and the linear program is solved again; additional processors are arranged in any position in the optimal sequence of σ_a and σ_c from the last iteration. There are two separate solution times tracked, one representing the best time for the current iteration, and another for the best time overall. ITERLP2 has two different configurations for latency; the first, ITERLP2A, uses latency in the constraint equations of the linear program prior to solving, tracking times after the fact and the second, ITERLP2B, ignores latency when solving the linear program, but after each linear program is solved, latency is factored in and the adjusted times are tracked. It might be interesting to see whether one approach performs better than the other.

After the addition of latency, it is shown in results from Chapter 4 that, in some cases, ITERLP2 may produce better solution times with fewer processors. For example, ITERLP2 finds the solution time for four processors, and then a fifth processor with much higher latency cost is added; since the fifth processor has such high latency, it is given a very small fraction of the load and consequentially produces a solution time worse than with four processors. So no matter the load fraction size given to a processor, the magnitude of latency is a fixed cost that is applied regardless of there being zero

load.

For example, given the optimal sequences $\sigma_a^1 = \{2, 1\}$ and $\sigma_c^1 = \{1, 2\}$, after the first iteration, the third processor would be added to the sequences in the following manner: $\Sigma_a^2 = \{\{3, 2, 1\}, \{2, 3, 1\}, \{2, 1, 3\}\}$ and $\Sigma_c^2 = \{\{3, 1, 2\}, \{1, 3, 2\}, \{1, 2, 3\}\}$. In the second iteration, Σ_a^2 and Σ_c^2 represent all possible sequences for allocation and collection respectively.

Table 2.1: Results for 3 processors sorting by communication parameter $\mathcal{C}$ with $\mathcal{C} = \{100, 125, 150\}, \mathcal{E} = \{1000, 700, 850\}, \mathcal{L} = \{10, 7, 9\}, \delta = 0.5$, rounded to 3 decimal places

Algorithm	σ_a	σ_c	α	T
OPT	$\{1, 2, 3\}$	$\{1, 3, 2\}$	$\{0.430, 0.263, 0.307\}$	436.033
ITERLP	$\{1, 2, 3\}$	$\{2, 1, 3\}$	$\{0.330, 0.371, 0.299\}$	444.258
ITERLP2A	$\{1, 2, 3\}$	$\{1, 3, 2\}$	$\{0.430, 0.263, 0.307\}$	436.033
ITERLP2B	$\{1, 2, 3\}$	$\{1, 2, 3\}$	$\{0.308, 0.392, 0.299\}$	437.559

Tables 2.1 and 2.2 show the results for iterations 2 and 3 respectively of an example with the following parameters: $\mathcal{C} = \{100, 125, 150, 175\}$, $\mathcal{E} = \{1000, 700, 850\}$, $\mathcal{L} = \{10, 7, 9\}$, and $\delta = 0.5$. ITERLP2A finds the optimal sequence in both iterations, and the remaining algorithms are near-optimal. Though it appears that ITERLP is performing near-optimal, adding more processors results in a solution time that deviates further and further from the optimal time; this is confirmed in the experimental results in Chapter. 4.

Table 2.2: Results for 4 processors sorting by communication parameter $\mathcal{C}$ with $\mathcal{C} = \{100, 125, 150, 175\}, \mathcal{E} = \{1000, 700, 850, 500\}, \mathcal{L} = \{10, 7, 9, 8\}, \delta = 0.5$, rounded to 3 decimal places

Algorithm	σ_a	σ_c	α	T
OPT	$\{1, 2, 3, 4\}$	$\{1, 3, 2, 4\}$	$\{0.217, 0.308, 0.184, 0.292\}$	352.196
ITERLP	$\{1, 2, 4, 3\}$	$\{4, 2, 1, 3\}$	$\{0.257, 0.290, 0.187, 0.266\}$	368.724
ITERLP2A	$\{1, 2, 3, 4\}$	$\{1, 3, 2, 4\}$	$\{0.217, 0.308, 0.184, 0.292\}$	352.196
ITERLP2B	$\{1, 2, 3, 4\}$	$\{1, 2, 3, 4\}$	$\{0.219, 0.278, 0.212, 0.291\}$	354.228

Chapter 3

Experiments

3.1 Simulations with random parameters

Prior to experimentation, a few preliminary tests were conducted to find a good balance for the system parameters. The values chosen for $\mathcal{C}$ and $\mathcal{E}$ in Tables. 3.1 and 3.2 are based on the values chosen from experiments in [1] with the addition of the latency $\mathcal{L}$ values; the values in the ranges are randomly chosen, but are also based on actual values for current processors and networks to handle arbitrarily large divisible loads.

Table 3.1: Parameters for simulation using 5 processors with $\delta = 0.5$

Case	$\mathcal{C}$ values	$\mathcal{E}$ values	$\mathcal{L}$ values
1	$[1, 100]$	$[100, 1000]$	$[0.01, 10]$
2	$[1, 100]$	$[100, 1000]$	$[0.1, 10]$
3	$[1, 100]$	$[100, 1000]$	$[1, 10]$
4	$[1, 100]$	$[100, 1000]$	$[1, 100]$
5	$[1, 100]$	$[1000, 10000]$	$[0.01, 10]$
6	$[1, 100]$	$[1000, 10000]$	$[0.1, 10]$
7	$[1, 100]$	$[1000, 10000]$	$[1, 10]$
8	$[1, 100]$	$[1000, 10000]$	$[1, 100]$
9	$[10, 1000]$	$[100, 1000]$	$[0.01, 10]$
10	$[10, 1000]$	$[100, 1000]$	$[0.1, 10]$
11	$[10, 1000]$	$[100, 1000]$	$[1, 10]$
12	$[10, 1000]$	$[100, 1000]$	$[1, 100]$
13	$[1, 100]$	$[1, 1000]$	$[0.01, 10]$
14	$[1, 100]$	$[1, 1000]$	$[0.1, 10]$
15	$[1, 100]$	$[1, 1000]$	$[1, 10]$
16	$[1, 100]$	$[1, 1000]$	$[1, 100]$

The experiments from Table. 3.1 include the optimal solution OPT for comparing to ITERLP and ITERLP2 with 5 processors; however, the experiments from Table. 3.2 do not include the optimal solution, due to the time requirements to solve $(16!)^2 = 4.4 \cdot 10^{26}$ and $(32!)^2 = 6.9 \cdot 10^{70}$ linear programs for 16 and 32 processors respectively. Each case includes 100 simulation runs, with each run being done 4 times to cover all sorting algorithms (by $\mathcal{C}$, by $\mathcal{E}$, by $\mathcal{C}$ then $\mathcal{E}$, and by $\mathcal{L}$) for both ITERLP2 A and B; ITERLP is only used once per run, sorting by $\mathcal{C}$. It is expected that results will reveal that ITERLP2 A and B outperform ITERLP in all cases.

Table 3.2: Parameters for simulation using 16 and 32 processors with $\delta = 0.5$

Case	# Processors	$\mathcal{C}$ values	$\mathcal{E}$ values	$\mathcal{L}$ values
17	16	$[1, 100]$	$[100, 1000]$	$[0.01, 10]$
18	16	$[1, 100]$	$[100, 1000]$	$[1, 10]$
19	16	$[1, 100]$	$[1000, 10000]$	$[0.01, 10]$
20	16	$[1, 100]$	$[1000, 10000]$	$[1, 10]$
21	16	$[10, 1000]$	$[100, 1000]$	$[0.01, 10]$
22	16	$[10, 1000]$	$[100, 1000]$	$[1, 10]$
23	16	$[1, 100]$	$[1, 1000]$	$[0.01, 10]$
24	16	$[1, 100]$	$[1, 1000]$	$[1, 10]$
25	32	$[1, 100]$	$[100, 1000]$	$[0.01, 10]$
26	32	$[1, 100]$	$[100, 1000]$	$[1, 10]$
27	32	$[1, 100]$	$[1000, 10000]$	$[0.01, 10]$
28	32	$[1, 100]$	$[1000, 10000]$	$[1, 10]$
29	32	$[10, 1000]$	$[100, 1000]$	$[0.01, 10]$
30	32	$[10, 1000]$	$[100, 1000]$	$[1, 10]$
31	32	$[1, 100]$	$[1, 1000]$	$[0.01, 10]$
32	32	$[1, 100]$	$[1, 1000]$	$[1, 10]$

3.2 Clusters

Experiments with clusters are done with 2, 4, and 8 clusters. Each cluster $\mathcal{U}_k$ has a range of processor nodes that start from 2 to a possible 32 processor nodes per cluster. The total number of processor nodes is restricted to a maximum of 64 to allow experiments to complete under realistic time constraints. Values for $\mathcal{C}$, $\mathcal{E}$, and $\mathcal{L}$ defined to be the same for all processor nodes in a given cluster; the goal is to simulate realistic grids of clusters for load schedules to be calculated. Table 3.3 lists 5 different cases with a total of 16 subcases amongst them. Each case should pose a problem for ITERLP, due to the fact that it just sorts by $\mathcal{C}$, and ignores latency; results are expected to show that ITERLP2 A and B outperform ITERLP due to its limitations.

The following conjectures can be made about the five cases. Case 1 will cause ITERLP to choose

Table 3.3: Cluster experiments with $\delta = 0.5$

Case	# Clusters $\mathcal{U}_k$	# Nodes/Cluster	$\mathcal{U}_1$	$\mathcal{U}_2$	$\mathcal{U}_3$	$\mathcal{U}_4$	$\mathcal{U}_5$	$\mathcal{U}_6$	$\mathcal{U}_7$	$\mathcal{U}_8$
1	2	4, 8	$\mathcal{C} = 150$ $\mathcal{E} = 300$ $\mathcal{L} = 2$	$\mathcal{C} = 100$ $\mathcal{E} = 1000$ $\mathcal{L} = 25$						
2	2	8, 16	$\mathcal{C} = 150$ $\mathcal{E} = 800$ $\mathcal{L} = 5$	$\mathcal{C} = 100$ $\mathcal{E} = 2000$ $\mathcal{L} = 5$						
3	4	4, 8	$\mathcal{C} = 100$ $\mathcal{E} = 800$ $\mathcal{L} = 5$	$\mathcal{C} = 100$ $\mathcal{E} = 2000$ $\mathcal{L} = 15$	$\mathcal{C} = 100$ $\mathcal{E} = 1500$ $\mathcal{L} = 10$	$\mathcal{C} = 100$ $\mathcal{E} = 3000$ $\mathcal{L} = 20$				
4	4	8, 16	$\mathcal{C} = 150$ $\mathcal{E} = 300$ $\mathcal{L} = 5$	$\mathcal{C} = 100$ $\mathcal{E} = 3000$ $\mathcal{L} = 5$	$\mathcal{C} = 150$ $\mathcal{E} = 1000$ $\mathcal{L} = 5$	$\mathcal{C} = 100$ $\mathcal{E} = 2000$ $\mathcal{L} = 5$				
5	8	4, 8	$\mathcal{C} = 110$ $\mathcal{E} = 400$ $\mathcal{L} = 5$	$\mathcal{C} = 100$ $\mathcal{E} = 500$ $\mathcal{L} = 5$	$\mathcal{C} = 110$ $\mathcal{E} = 450$ $\mathcal{L} = 5$	$\mathcal{C} = 100$ $\mathcal{E} = 1000$ $\mathcal{L} = 5$	$\mathcal{C} = 110$ $\mathcal{E} = 1000$ $\mathcal{L} = 2$	$\mathcal{C} = 100$ $\mathcal{E} = 500$ $\mathcal{L} = 10$	$\mathcal{C} = 110$ $\mathcal{E} = 500$ $\mathcal{L} = 2$	$\mathcal{C} = 100$ $\mathcal{E} = 1000$ $\mathcal{L} = 10$

a very distant cluster with better bandwidth, although the slower bandwidth cluster is much closer and has faster processors. Case 2 will make ITERLP choose a cluster with better bandwidth, but with slower processors, in the same network. Case 3 gives an example that will show the limitations of sorting only by communication parameters where bandwidth is the same for all processor nodes. Case 4 is just an extended version of case 2 with two additional clusters. Case 5 gives a wide range of parameters and should give some varying results.

Chapter 4

Results

4.1 Simulations with random parameters

Simulations with ranges of randomly generated system parameters were conducted for $m = 5, 16, 32$, and $\delta = 0.5$. The following are algorithms used when simulating with $m = 5$: OPT, ITERLP, ITERLP2A, and ITERLP2B. The ITERLP2 algorithms were sorted in four different configurations (by $\mathcal{C}$, by $\mathcal{C}$ then $\mathcal{E}$, by $\mathcal{E}$, and by $\mathcal{L}$). When $m = 16$ or $m = 32$, the time taken to calculate OPT would be unrealistic, so only the ITERLP, ITERLP2A, and ITERLP2B were used. There were 32 cases in total with 100 simulation runs for each configuration of algorithm and sorting order.

Average percent deviation for each algorithm $T_{VARIANT}$ is represented by $\Delta T_{VARIANT}$ in the calculation as below:

$$\Delta T_{VARIANT} = \frac{Mean(T_{VARIANT}) - Mean(T_{OPT})}{Mean(T_{OPT})} * 100\% \tag{4.1}$$

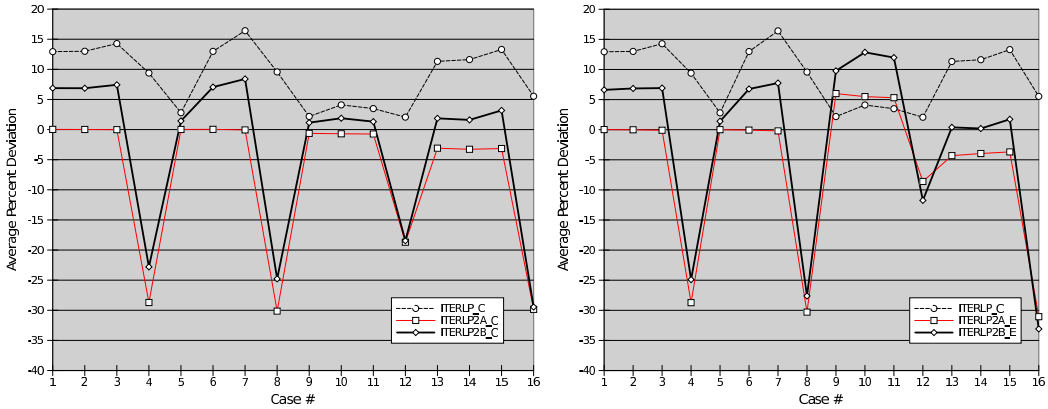


Figure 4.1: Simulation results.

The results for the first 16 cases are depicted in Fig. 4.1. The graph shows the average percent deviation $\Delta T_{VARIANT}$ that the variant algorithms ITERLP, ITERLP2A, and ITERLP2B make from OPT. ITERLP is compared to both ITERLP2 algorithms sorting by C, and by E in the left and right graphs respectively. Cases 4, 8, 12, and 16 all show anomalous points where both ITERLP2 algorithms produce solution times lower than OPT; this was due to the values for $\mathcal{L}$ being too high in relation to $\mathcal{C}$ and $\mathcal{E}$. The problem was with OPT permuting all possible combinations of 5 processors; some of them were being allocated zero load, but still assigned the cost of latency. This made ITERLP2A and ITERLP2B produce much better schedules than OPT, only because they were able to terminate early with fewer processors when zero load was allocated. Figure. 4.2 on page 18 illustrates a simulation run from case 8; it shows that OPT schedules load to 2 processors, but incurs the burden of latency from 5 processors, regardless of the fact that 3 of them are allocated zero load.

Another anomaly can be seen in cases 9, 10, and 11. Since these cases have similar results, one graph is shown for case 11 in Fig. 4.6. It appears that ITERLP is outperforming both of the ITERLP2 algorithms that sort by $\mathcal{E}$, or by $\mathcal{L}$. These cases show that when values for communication parameters are too high in relation to the computational parameters, that sorting by $\mathcal{E}$, or by $\mathcal{L}$ is not suitable. Though the results in this case are near-optimal for both ITERLP2 algorithms when sorting by $\mathcal{C}$ or by $\mathcal{C}$ then $\mathcal{E}$, in realistic cases, ITERLP should not be outperforming both ITERLP2 algorithms that sort by $\mathcal{E}$, or by $\mathcal{L}$. The reason is that ITERLP2 algorithms are designed to find the best solution times with latency cost factored in, and adjusting for latency with ITERLP is expected to fail.

The remainder of the results in cases 1, 2, 3, 5, 6, 7, 13, 14, and 15 show the limitations that

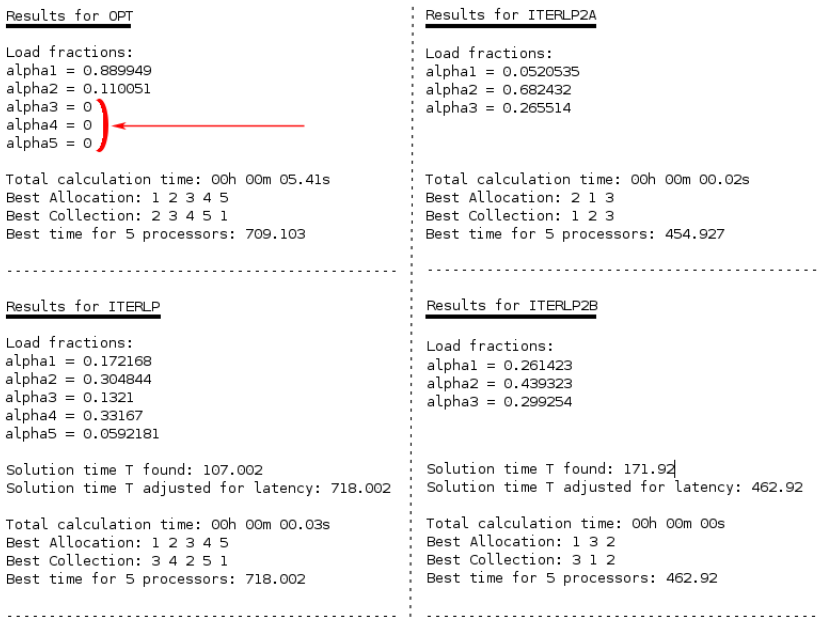
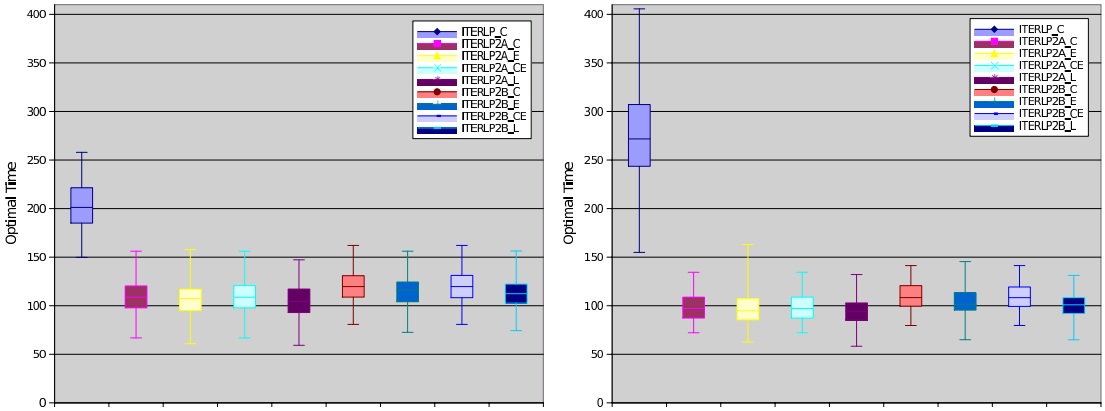


Figure 4.2: Anomalous Results

ITERLP has when latency is ignored. They also show that ITERLP2A and ITERLP2B both outperform ITERLP in all realistic cases. Overall ITERLP2A performed best with the nearest-optimal times.

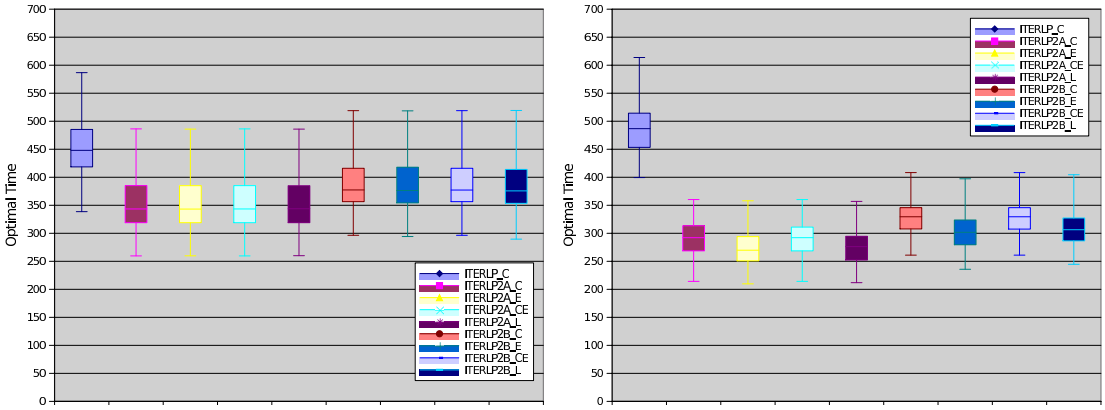
All even-numbered cases from cases 17 through 32 are shown in Figures 4.3, 4.4, 4.5, and 4.7. For trending purposes, each figure has two graphs that show the effects of increasing the number of processor nodes from 16 to 32. Each graph uses a boxplot; each box has a whisker coming from the top representing the max value, a whisker coming from the bottom representing the min value, and a line passing left-to-right through the box representing the mean value.

The performance of ITERLP in Figures 4.3, 4.4, and 4.7 appears to be following the same decremental trend when the number of processor nodes is increased from 16 to 32. The results from case 26 in Fig. 4.3 show that ITERLP produced a median optimal time that is almost 3 times that of the worst optimal times that both ITERLP2 algorithms have produced. It is unquestionable that ITERLP fails to perform near-optimal and is, by far, outperformed by ITERLP2A and ITERLP2B in these cases. Both ITERLP2 algorithms perform well in all six graphs, but though the performance does improve, the solution time is not reduced by half when the number of processor nodes is doubled to 32. No matter what sorting method the ITERLP2 algorithms use, performance stays pretty consistent with median values that vary no more than 30 time units.



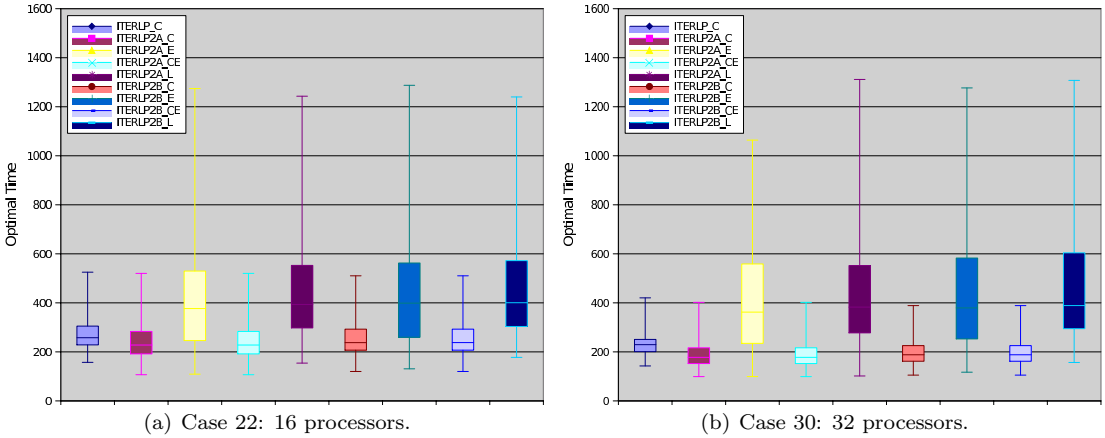
(a) Case 18: 16 processors. (b) Case 26: 32 processors.

Figure 4.3: Simulation results.



(a) Case 20: 16 processors. (b) Case 28: 32 processors.

Figure 4.4: Simulation results.



(a) Case 22: 16 processors. (b) Case 30: 32 processors.

Figure 4.5: Simulation results.

Fig. 4.5 shows results for cases 24 and 32 which follow a similar trend to what is seen in cases 9, 10, and 11; they also share the same parameters as case 11 for $\mathcal{C}$, $\mathcal{E}$, and $\mathcal{L}$. Results for case 11 are shown in Fig. 4.6 for comparing to Fig. 4.5. The results for both ITERLP2 algorithms show again that when values for communication parameters are too high in relation to the computational parameters, that sorting by $\mathcal{E}$, or by $\mathcal{L}$ is not suitable. Again it is unrealistic that ITERLP have better performance than ITERLP2A and ITERLP2B sorted by $\mathcal{C}$ or by $\mathcal{C}$ then $\mathcal{E}$.

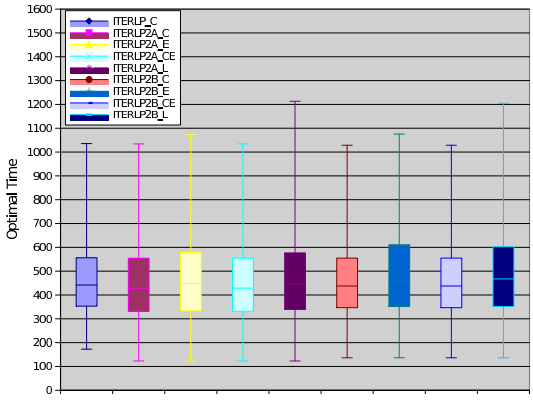
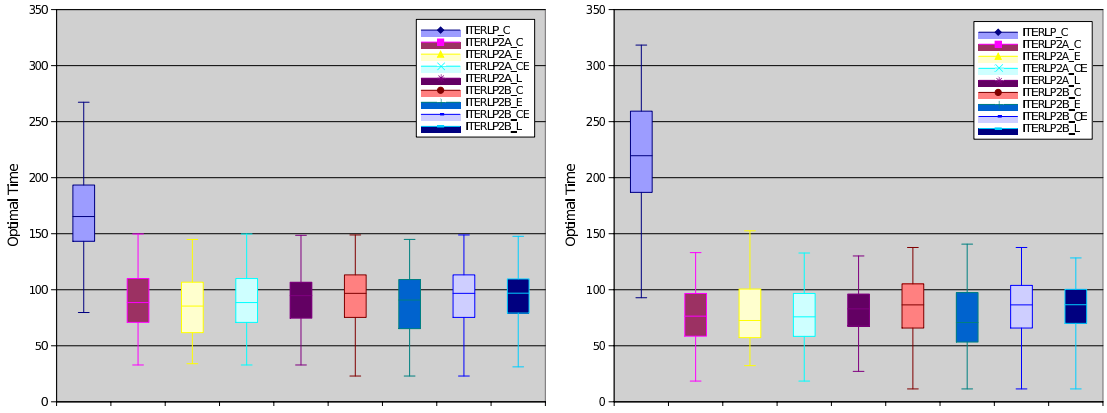


Figure 4.6: Case 11 results.



(a) Case 24: 16 processors.

(b) Case 32: 32 processors.

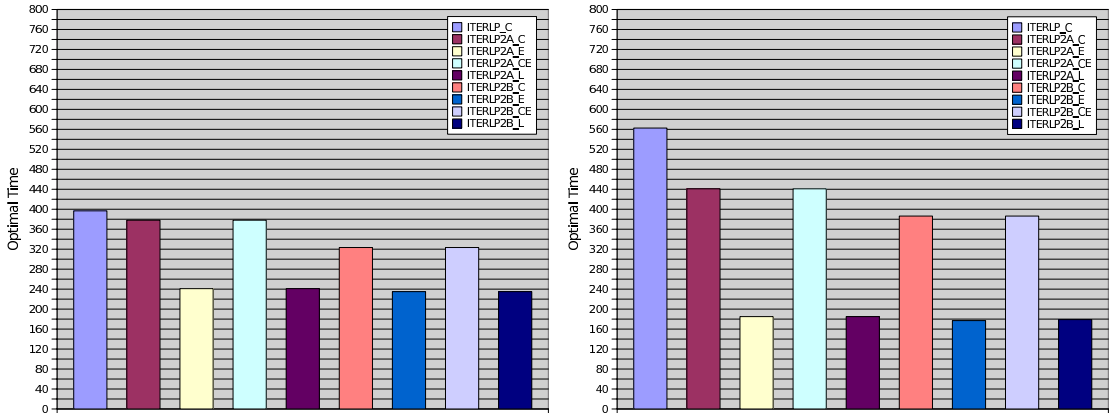
Figure 4.7: Simulation results.

Though there were a few anomalies with certain system parameters, the results definitely show that with realistic ranges of random values and the cost of latency being factored in, that ITERLP does not perform near-optimal; it is shown that ITERLP has limitations when latency is ignored. Simulations show again that both ITERLP2 algorithms outperform ITERLP in all realistic cases; as well ITERLP2A performed best with the nearest-optimal times overall.

4.2 Clusters

The following results include all algorithms except OPT. Experiments were conducted for 2, 4, and 8 clusters with $m = 4, 8, 16, 32$ processor nodes in each cluster with up to a possible 64

processors per experiment. The ITERLP2 algorithms were sorted in four different configurations (by $\mathcal{C}$, by $\mathcal{C}$ then $\mathcal{E}$, by $\mathcal{E}$, and by $\mathcal{L}$). Figures 4.8, 4.9, 4.10, 4.11, and 4.12 show some results from all five cluster experiments; there are two graphs per figure, with one cluster simulation run in each.

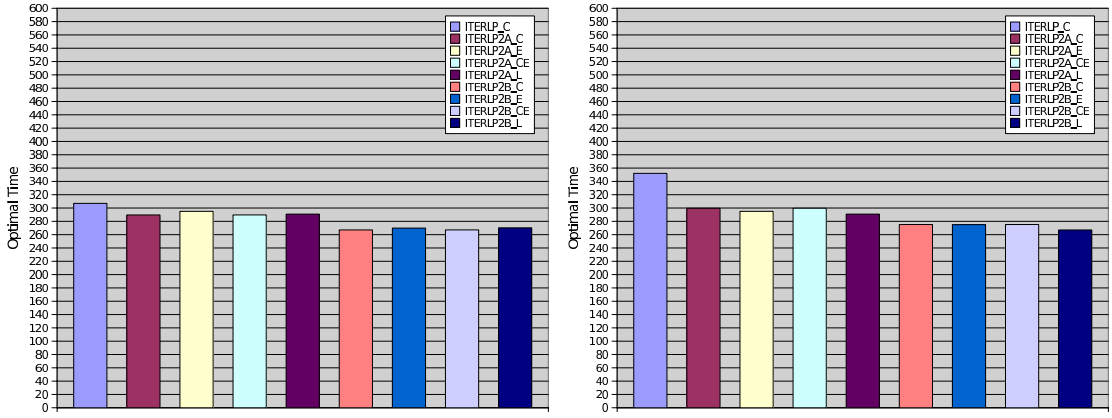


(a) 8 processors. (b) 16 processors.

Figure 4.8: Case 1 cluster results.

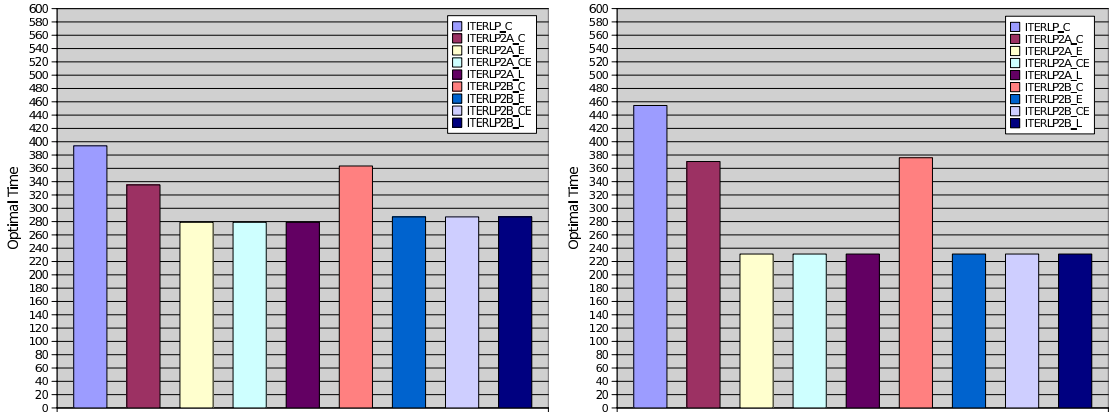
Results for case 1 are quite interesting, because they show that ITERLP, and both ITERLP2 algoriths that sorting by $\mathcal{C}$, or by $\mathcal{C}$ then $\mathcal{E}$ are progressively worsened when the number of processors is increased from 8 to 16. The optimal time of ITERLP appears to get much worse, whereas, both ITERLP2 algorithms seem to only worsen slightly. Despite the poor solution times when sorting by $\mathcal{C}$, or by $\mathcal{C}$ then $\mathcal{E}$, the results are the complete reverse for both ITERLP2 algorithms, when sorting by $\mathcal{E}$, or by $\mathcal{L}$; the results not only improve when going from 8 to 16 processors, but also outperform ITERLP by three times. It is shown in Fig. 4.8 with 16 processors that with either ITERLP2 algorithm, the worst results for $\mathcal{E}$, or by $\mathcal{L}$ are in the 180 range for optimal time, as opposed to ITERLP with an optimal time of around 560. The parameters used in this cluster experiment definitely favour sorting by $\mathcal{E}$, or by $\mathcal{L}$.

Case 2 produced very monotonous results in Fig. 4.9 for all algorithms; however, results do show that ITERLP is outperformed by ITERLP2A and ITERLP2B, but not by much. Another thing to note is that with any of the algorithms going from 16 to 32 processors, when sorting by $\mathcal{C}$, or by $\mathcal{C}$ then $\mathcal{E}$ the results are worsened and when sorting by $\mathcal{E}$, or by $\mathcal{L}$, there is no change. The reason that the results are worsened with 32 processors is that there were, on average, 16 processors chosen for the solution times; this was the worst choice due to the computational parameters being much higher than that of the other 16 processors that were not chosen. Overall for this case, whether ITERLP2A or ITERLP2B is used, any sort order is sufficient and ITERLP is still outperformed by both ITERLP2 algorithms.



(a) 16 processors. (b) 32 processors.

Figure 4.9: Case 2 cluster results.

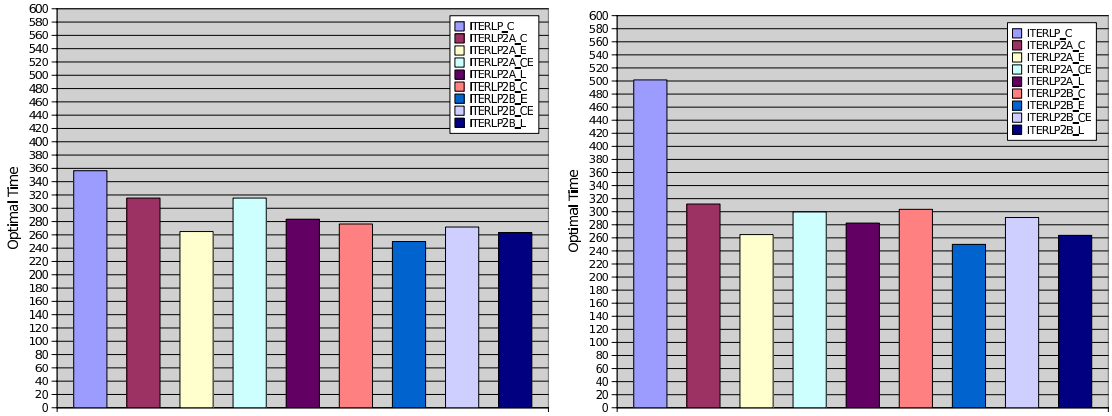


(a) 16 processors. (b) 32 processors.

Figure 4.10: Case 3 cluster results.

Results from case 3 are among the most interesting cluster experiments. This experiment was assumed to show the limitation of sorting only by $\mathcal{C}$ with ITERLP. This assumption was correct, and it definitely shows that with either ITERLP2 algorithm, that sorting by $\mathcal{C}$ is also the worst choice. Based on the parameters chosen initially, it was obvious that when sorting by $\mathcal{E}$, by $\mathcal{C}$ then $\mathcal{E}$, or by $\mathcal{L}$ that results should be the same and nearest-optimal. The results also show that the performance of ITERLP goes from being mediocre to more than twice as worse, when going from 16 to 32 processor respectively. It is shown from the results in Fig. 4.9 that ITERLP is once again outperformed by both ITERLP2 algorithms.

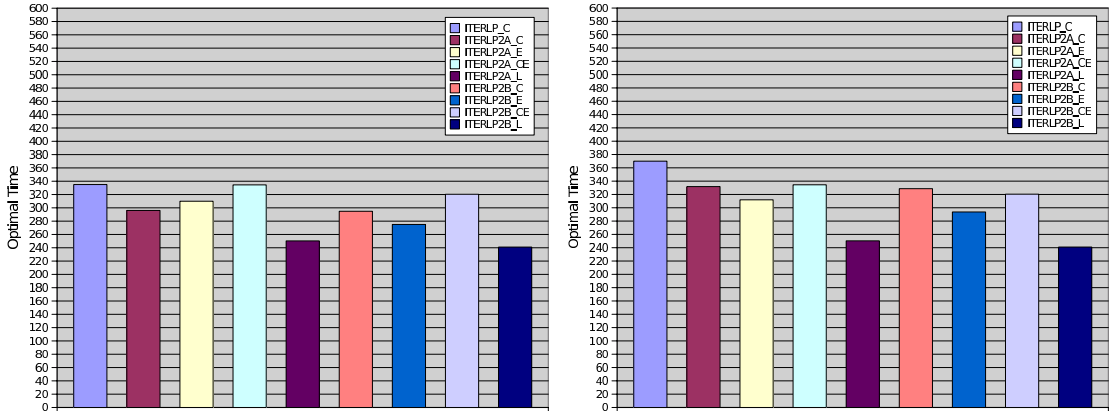
Cases 4 and 5 show similar results with some interesting irregularities. When going from 32 to 64 processors, some optimal times had no change, others had worsened, and the rest had improved only slightly. Figures 4.11 and 4.12 show that sorting by $\mathcal{E}$ or by $\mathcal{L}$ are the best choices for ordering



(a) 32 processors. (b) 64 processors.

Figure 4.11: Case 4 cluster results.

with both ITERLP2 algorithms, regardless of the times being worse with 64 processors. ITERLP is especially worsened when going from 32 to 64 processors in Fig. 4.11 with and optimal time that is double that of ITERLP2B sorting by $\mathcal{E}$. Though the results are worsened in some cases for ITERLP2A and ITERLP2B, ITERLP is still outperformed; as well, ITERLP2B appears to produce the nearest-optimal times.



(a) 32 processors. (b) 64 processors.

Figure 4.12: Case 5 cluster results.

It was unexpected in the cluster results that ITERLP2B would produce better times that ITERLP2A in all cases, since in the random simulations, ITERLP2A produced the nearest-optimal times.

Chapter 5

Conclusion

In this thesis a modified version of the heuristic algorithm ITERLP, named ITERLP2, was created to investigate other sorting methods and latency cost. Random simulations and cluster experiments have shown that, in realistic cases, ITERLP fails to produce near-optimal times when latency is introduced; however, ITERLP2A and ITERLP2B consistently produce near-optimal performance irregardless of the level of heterogeneity, the number of processor nodes, or the magnitude of the result set. It has been shown where each of the sorting methods is best suited in cluster experiments, but no method is best suited for all cases. ITERLP2A performs best in the random simulations, whereas ITERLP2B performs best with cluster experiments. Finally, results from experimentation have shown that ITERLP2 outperforms ITERLP in all realistic cases.

5.1 Contributions

To contribute to DLT, it has been shown that ITERLP has limitations when sorting only by communication parameters and neglecting the cost of latency. As well, a newly modified and improved heuristic algorithm, ITERLP2, has been created to be more realistic by factoring in the cost of latency and considering other sorting methods. It has been shown that ITERLP2 outperforms ITERLP in all realistic cases.

5.2 Future Work

When the value of m becomes very large, ITERLP2 does not effectively schedule divisible loads and could take a great deal of time to complete. To help get around this limitation, one can produce

an application of DLS involving clusters of clusters whereby each cluster can be considered as an equivalent processor [1, 9, 11]. A load schedule can be created for this by considering clusters of clusters as a multilevel-processor tree that can be recursively solved [1]. This is done by initially solving with the set of equivalent processors in the first level of the tree. After each cluster in the first level has had a load fraction calculated for it, each child processor/cluster below them will have the parent load fraction split up yet again. This process repeats until the leaf nodes have been reached in all branches.

Bibliography

[1] O. Beaumont, A. Ghatpande, H. Nakazato, H. Watanabe. *Divisible Load Scheduling with Result Collection on Heterogeneous Systems.* (Heterogeneous Computation Workshop IPDPS, pp. 1-8, 2008).

[2] S. K. Chan, V. Bharadwaj, D. Ghose. *Large matrix-vector products on distributed bus networks with communication delays using the divisible load paradigm: performance analysis and simulation.* (Math. Comput. Simul. 58, 1 (Dec. 2001), 71-92).

[3] S. Bataineh, T. Hsiung, T.G. Robertazzi. *Closed Form Solutions for Bus and Tree Networks of Processors Load Sharing a Divisible Job* (IEEE Trans. Comput., vol. 43, no. 10, pp. 1184-1196, 1994).

[4] A. Ghatpande, H. Nakazato, O. Beaumont, H. Watanabe. *Analysis of Divisible Load Scheduling with Result Collection on Heterogenous Systems* (IEICE Transactions on Communications 2008, vol. E91-B, no. 7, pp. 2234-2243, 2008).

[5] A. Ghatpande, H. Nakazato, O. Beaumont, H. Watanabe. *SPORT: An Algorithm for Divisible Load Scheduling with Result Collection on Heterogeneous Systems* (IEICE Transactions on Communications 2008, vol. E91-B, no. 8, pp. 2571-2588, 2008).

[6] SETI@home. *SETI@home: Search for Extraterrestrial Intelligence at Home* (*http://setiathome.ssl.berkeley.edu*, August, 2008).

[7] ACEnet wiki. *ACEnet website* (*http://wiki.ace-net.ca*, May, 2008).

[8] O. Beaumont, L. Marchal, Y. Robert. *Scheduling divisible loads with return messages on heterogeneous master-worker platforms* (High performance computing. International conference No12, Goa , INDE (18/12/2005) 2005, vol. 3769, pp. 498-507).

[9] O. Beaumont, H. Casanova, A. Legrand, Y. Robert, Y. Yang. *Scheduling Divisible Loads on Star and Tree Networks: Results and Open Problems* (IEEE Transactions on Parallel and Distributed Systems ,vol. 16, no. 3, pp. 207-218, March, 2005).

[10] T. G. Robertazzi. *Ten Reasons to Use Divisible Load Theory.* (Computer, vol. 36, no. 5, pp. 63-68, May, 2003).

[11] D. Yu, T.G. Robertazzi. *Divisible Load Scheduling for Grid Computing* (Proc. of the IASTED International Conference on Parallel and Distributed Computing and Systems, Los Angeles, Nov, 2003).

[12] COIN-Clp. *COmputational INfrastructure for Operations Research* (*http://www.coin-or.org*, May, 2008).