A distributed system for track discovery

by
Matthew Cleveland

A Dissertation Submitted to the Faculty of the

Department of Computer Science

In Partial Fulfillment of the Requirements
For the Degree of

Masters of Science

In the Graduate College

The University of Arizona

2 0 1 1

Table of Contents

Table of Contents—*Continued*

List of Figures

List of Algorithms

Abstract

Existing data fitting algorithms for track discovery are accurate and field-proven. As data sets increase in size, however, memory and computational restraints demand more robust solutions than are currently available. In this paper we present a set of algorithms for parallel data fitting. These algorithms make use of approximation algorithms, intelligent caching, and modeling to facilitate the efficient parallelization of the model fitting problem, with applications in track discovery.

Chapter 1

Introduction

Track discovery is a procedure that involves fitting object data to models of an object's movement over time. Fitting data points to a model is a research domain with established work, and track discovery is one application domain that has benefited from this work. Existing algorithms for track discovery do not efficiently handle large data sets, and their corresponding heavy computational loads. This thesis presents a set of algorithms for parallel track discovery capable of handling large data sets and their computational workloads.

Current sequential data fitting algorithms for track discovery are not designed to handle the large data sets that are now commonplace [ITA+08]. The algorithms pertinent to the context of this paper are those that utilize variable tree approaches to perform track discovery [Kub05]. These variable tree algorithms have unacceptable performance characteristics on large data sets.

This research focuses on the variable trees algorithms due to their pervasive use in the field. These algorithms suffer from a design that makes naive parallelization impractical. In order to achieve acceptable performance via parallelization, a restructuring of the variable trees algorithms is required. A thorough analysis of existing algorithms shows how the variable trees algorithms can be restructured in a way that enables its efficient parallelization.

This paper presents a set of algorithms and modeling techniques designed to efficiently perform parallel track discovery while utilizing the foundation provided by the variable trees algorithms. Two primary algorithms are discussed: a load balancing algorithm and a memory caching algorithm. These algorithms parallelize track discovery by identifying the basic algorithmic sections of the restructured variable tree algorithms and intelligently dividing processing between a master processor and worker processors.

Our load balancing algorithm uses solution space sampling to ensure that an efficient work distribution is realized. Modeling techniques enable the accurate prediction of worker processor computational load. These models were developed to enable the load balancing algorithm to analyze work loads and assign them scalar costs.

Memory caching enables worker processors to efficiently manage their workload. Data sets that exceed total memory size are expected. As these data move in and out of memory, the memory caching algorithms designed for this system ensure that the minimum penalty for those caching events is incurred.

We show that the restructured variable trees algorithms and the new parallel algorithms converge to create an efficient parallel track discovery system: load balancing

is shown to reduce execution time, memory caching is shown to minimize I/O events, and the system is shown to be scalable as nodes are added.

Track discovery is used in many application domains. This work was inspired by the use of track discovery in Astronomy can be used to define the movement of celestial bodies over time. By tracking object movement, orbits can be identified, and objects such as asteroids can be discovered.

This thesis evolved out of work being done at the Large Synoptic Survey Telescope (LSST) project. LSST is an Earth-based telescope whose construction is slated to complete in 2015. Once operational, the telescope expects to fulfill a variety of scientific goals, one of which is asteroid identification. The LSST will survey the entire visible night sky, giving a synoptic, or full, view of the heavens as they change over time.

Chapter 2

Track discovery background

2.1 Terminology

A *detection* is an object observed in space such as a star, asteroid, or some noise point. *Detections* can be grouped together into sets called *tracklets*. These *tracklets* define a *detection*'s path through space and time. When a collection of *tracklets* can define a *detection*'s path through space and time we have a *track*. A collection of *tracklets* from a common region of space and common time, however, is called a *sector*. A *sector* differs from a *track* in that a *sector* is a set of *tracklets* from a similar region of space at the same time, whereas a *track* is a set of *tracklets* from different times and different spaces. A *track* defines a *detection*'s movement over large spans of time and space and could define the orbit of celestial bodies, such as asteroids.

Sectors are grouped into *work items*. *Sectors* are useful as they allow the grouping of *tracklets* by region of space, which increases the granularity of *work item* data. A *work item* contains two endpoint *sectors* and a set of support *sectors*. The *sectors* that comprise a *work item* are chosen deliberately as they have been determined to contain data that represent regions of space that could describe *detections* at different times in space.

Each *work item* requires processing by the *track extraction* algorithm, which is the process of comparing a group of *sectors* and determining if any of their constituent tracklets create a roughly-quadratic line of motion, or path through space and time. If *track extraction* finds a collection of *tracklets* that could define a *detection*'s path over space and time, a *track* is generated.

The *tree walk* algorithm is a recursive algorithm that examines groups of *sectors* to determine if they meet the minimum criteria for *track* candidacy. This is also called to the variable tree walk algorithm in other related papers, such as [Mye08] and [Kub05].

A *task pool* is a set of *work items*. The *distribution algorithm* is the logic that examines the *task pool* and its associated meta-data and assigns *work items* to worker nodes.

The pseudocode in algorithm 1 describes *track extraction*, the processing of a *work item*.

Algorithm 1 Track extraction

Input: Two endpoint sectors S
Input: Set of support sectors R
Output: A set of tracks T

```
for each s ∈ S do
  /* s and s' are pairs of chunks of space-time */
  for each s' ∈ S and s' ≠ s do
    for each t ∈ s do
      /* t and t' is a pair of tracklets within s and s' */
      for each t' ∈ s' do
        if t and t' are compatible then
          determine if t, t', and compatible R describe a track, add the track to
          T
        end if
      end for
    end for
  end for
end for
```

2.2 Track extraction

Figure 2.1 illustrates a work item and the items that comprise it. A work item contains a set of sectors that may contain tracklets constituting a full track when linked together. To generate a track from a work item, two tracklets from different times are used as endpoints. The tracklet from the earlier time is the first endpoint and the tracklet from the later time is the second endpoint. Between these two tracklets must exist one or more tracklets from times greater than the first tracklet and less than the second tracklet. These intermediate tracklets are contained in support points. Track extraction requires one first endpoint, one second endpoint, and at least one support point. This set of endpoints is now evaluated to see if it represents any tracks.

2.3 Computational complexity of track extraction

Track extraction is a stage in the track discovery procedure that is significantly time-consuming and, to our knowledge, has no parallel algorithms available. This section discusses the sources of the computational complexity of track extraction, how current algorithms handle those complexities, and how parallelization can enable the dispersion of the load with the goal of mitigating the long computation time and vast memory requirements.

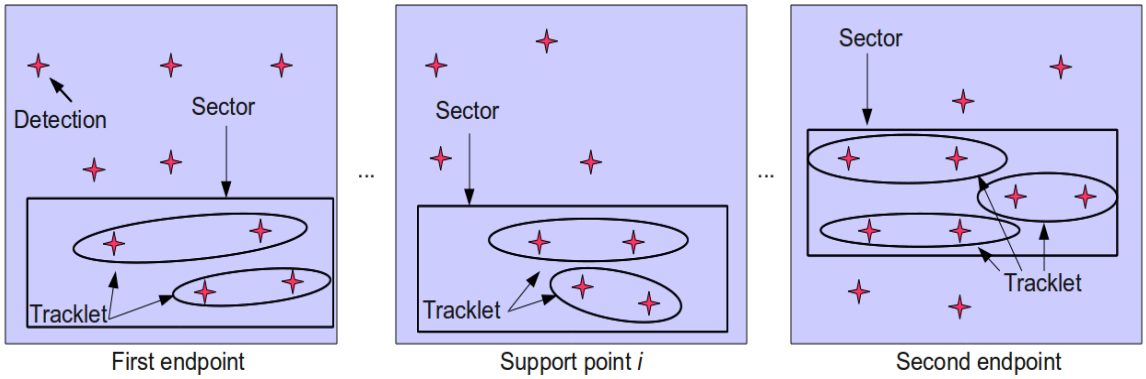

FIGURE 2.1. A work item is comprised of sectors and each sector contains tracklets. There are two endpoint sectors in a work item and $n-2$ support points, where n is the total number of sectors in the work item.

To illustrate why track extraction is so demanding of memory and computational resources, consider the following procedure required to generate all tracks from an input data set of tracklets. Each work item contains n sectors and each sector contains m tracklets. As shown in the track extraction pseudocode, all tracks from each sector must be compared with all the tracks of the remaining sectors. For each pair of sectors there are $\mathcal{O}(m^2)$ tracklet comparisons.

Each sector must be compared with $\mathcal{O}(n)$ other sectors, resulting in $\mathcal{O}(m^3)$ tracklet comparisons. There are $\mathcal{O}(n)$ sectors which must be compared with $\mathcal{O}(n)$ other sectors, resulting in $\mathcal{O}(m^4)$ total tracklet comparisons per work item.

For each of the $\mathcal{O}(m^4)$ comparisons a significant amount of double-precision computations is required, as object trajectories are calculated and compared. Due to the computational complexity and math-heavy operations, track extraction is time consuming and memory intensive.

Chapter 3

Sequential track discovery

Analysis of the current sequential track discovery algorithms reveal that two primary phases constitute the track discovery procedure. The first is a recursive traversal of the trees, the second is the analysis of potential track linkages.

3.1 Recursive tree walk

In the tree walk algorithm, binary trees are used to hold data representing tracklets. For a given data set there are n trees, where n is the number of distinct times at which data was observed. The goal of the tree walk is to traverse the trees and arrive at a point where between 3 and n leaf nodes from unique trees have been selected (3 is the minimum number, as it describes a set with two endpoint nodes and a single, intermediate, support node.) These leaf nodes represent areas of space in which an object could feasibly exist at the times represented by the tree in which they are stored (more information can be found in Kubica's paper [Kub05]). To be clear, a leaf node is a collection of tracklets and is synonymous with term *sector*. This set of leaf nodes is a work item and contains many potential candidate tracks.

The algorithm starts by selecting two trees' root nodes as endpoint nodes. All trees whose time falls between the two chosen endpoint nodes have their root nodes included as potential support nodes. The recursion starts by checking if all nodes in the set are leaf nodes. If not, the two endpoint nodes are examined to determine which is "widest", or contains data representing the largest amount space. The recursive algorithm is then called twice, once with the widest endpoint's left child as a new endpoint node, and once with the widest endpoint's right child as a new endpoint node. This recursion continues until the two endpoints are leaf nodes. At this point the set of endpoint tree nodes and support nodes are considered a work item and are ready for track extraction.

The pseudocode in algorithm 2 [Kub05] illustrates the logic of the recursive tree walk algorithm. This algorithm is performed for each unique pair of trees.

3.2 Track extraction

The track extraction phase in the sequential track discovery algorithms operates according to algorithm described in 2.2 and 2.3. Comparing all items from the one set to all objects in the set of another is an expensive process, and through careful

Algorithm 2 Recursive tree walk

function: RecursiveTreeWalk(e_1, e_2, S)
Input: First endpoint e_1 and second endpoint e_2
Input: A set of support tree nodes S
Output: A list Z of tracks

$S' \leftarrow \{\}; Scurr \leftarrow S$
if tree nodes in $e1$ and $e2$ are compatible: **then**
  while $s \in Scurr$ **do**
    if s is compatible with M: **then**
      if s is not a leaf node: **then**
        Add s's left and right children to the end of $Scurr$.
      else
        Add s to S'
      end if
    end if
  end while
  if we have enough valid support points: **then**
    if both e_1 and e_2 are leaves: **then**
      Test all combinations of points owned by the endpoint nodes, using the support nodes
      Add valid sets to Z.
    else
      Let m^* be the endpoint node that owns the most points.
      e_1' = m^*'s left child
      e_2' = m^*'s right child
      RecursiveTreeWalk(e_1', e_2, S')
      RecursiveTreeWalk(e_1, e_2', S')
    end if
  end if
end if

analysis it was determined that this stage of processing was the best granularity at which to divide processing in a parallel approach. This is further discussed in the following section.

As this set of algorithms became unmanageable in terms of computation time, parallelization was pursued. The resulting set of distributed algorithms comprise what we term Distributed Link Tracklets (DLT). These algorithms were designed to perform the variable trees algorithms task while leveraging multiple processors in an intelligent manner. In the following sections DLT is presented and described in detail.

Chapter 4

Parallel track discovery

Distributed Link Tracklets (DLT) is an extension of the existing variable trees algorithms with some key additions and modifications. Justifications for the design and a detailed description of the DLT algorithms are discussed in the subsequent subsections.

Parallel track discovery requires a reworking of the sequential algorithms to facilitate proper load balancing and memory management. Towards achieving this end, a thorough analysis was performed on the sequential track discovery algorithms. The main algorithms can be divided into two logical components: a tree walk process and a tracklet extraction process. The tree walk process, discussed in section 3.1, is where the tracklet sets are generated. The track extraction process is where these tracklet sets are analyzed for the existence of object orbits.

The latter phase of the sequential algorithm, the track extraction, dominates the execution time of the track discovery stage. Unfortunately, tracklet set comparisons occur as the base case of the recursive tree walk process, so the isolation of this process required major algorithm restructuring.

Ultimately, the goal of DLT is to distribute the track extraction workload while minimizing total cost. The master generates x work item sets of size n, $W_k; 0 \leq k < x$. For each W_k we partition the work items into per-processor sets of work items, $P_i^k; 0 \leq i < p$, where p is the number of processors. To be clear, a partition P_i^k is a work item distribution where all work items are assigned to some worker processor, and each processor receives at least one work item. Each P_i^k consists of p work item sets $w_i; 0 \leq i < p$, consisting of work items w_{ij} such that all w_{ij} are assigned to exactly one processor and $\forall w_i; |w_i| > 0$.

Describe the cost of processing a work item as $f(w_{ij})$. Therefore processor i's cost of processing a work item set is

$$C_i = \sum_{j=0}^{|w_{ij}|} f(w_{ij})$$

With these values established we can define the cost of a work item set partition to be

$$costOf(P_i^k) = max(\forall i;\ i \leq 0 < p; C_i)\ .$$

The optimal partition of W_k is the partition with the minimum cost over all possible partitions for P_i^k, define this quantity as G_k

$$G_k = \underset{i}{\operatorname{argmin}}(costOf(P_i^k))$$

The goal, therefore, is to find the optimal partition for all W_k

$$\sum_{k=0}^{x} G_k$$

4.1 Parallel algorithm structure

As track extraction is the most time consuming step of the overall track discovery procedure, it was the algorithm whose parallelization provided the most benefit to overall execution time. This step is encountered at the base of the recursive tree walk, however, which posed a problem. Distributing work items to workers each time the tree walk reached a terminal recursion would incur too much overhead, and, additionally, would prevent work load analysis for load balancing. Therefore, instead of performing track extraction as the tree walk generated work items, the work items were instead accumulated until a sufficient number had been generated for analysis and distribution.

This set of work items, say of size n, can be divided among the m workers in m^n different ways, which is far too large a set to consider many distributions. Since a work item describes a set of tracklets, it is possible that multiple work items may have similar constituent tracklets. Noting that work items may have similar tracklet composition allows us to leverage this overlap to intelligently assign task pools to the workers, keeping in mind that the size of the data to be distributed is likely larger than the memory available to a worker. With these memory restrictions in mind, we must design algorithms that can operate using restricted subsets of a processor's memory.

Worker processors, each with their own cache of sectors in memory, greatly benefit from having sectors assigned to it in such a way that cache hits are maximized. For every cache miss, sector data must be retrieved from auxiliary storage at great computational expense. By considering memory implications, intelligently distributing work items to workers can yield significant performance improvement. Algorithms for parallel track discovery, therefore, should leverage the knowledge of task pool contents to distribute work items with the intent of minimizing computational expense with regard to caching.

Distributing work items intelligently is not an easy task. With such a vast number of possible task pool assignments, it is prohibitively expensive to evaluate each assignment to determine which would best facilitate the aforementioned cache considerations. Naive task pool assignments, while easy to generate, ultimately result in poorer performance. The following section describes a mechanism for finding intelligent task pool assignments. We show that the processing required to determine

intelligent task pool assignments results in overall performance gains in run time and cache hits.

4.1.1 Simulated annealing

For tasks with large search spaces, such as task pool assignment, approximation algorithms are often used. The simulated annealing approximation algorithm complements this problem well, because simulated annealing algorithms examine diverse subsets of large solution spaces and provide safeguards against selecting solutions that are local optimums instead of global ones. In this section we describe simulated annealing as it is used in parallel track discovery.

First, a solution is chosen as the starting point. In DLT this solution is a set of work item sets, or a task pool. The number of work item sets in the task pool is equal to the number of workers. According to simulated annealing, this solution must be analyzed and assigned a scalar cost. This cost is calculated according to the execution model discussed in section 4.2.

Next, a permutation of the task pool set is generated, at random, and is assigned a cost using the same execution model. This process continues until the simulated annealing algorithm completes.

As task pools are encountered, they are accepted as the "current" solution, with a specified probability. The fact that higher-cost task pools may be accepted is a key to the simulated annealing algorithm. This feature is what deters the algorithm from settling on a local optimum. The probability with which a task pool is accepted is a function of temperature, which, in the case of DLT, is the number of simulated annealing iterations completed. As the temperature approaches zero, or the number of iterations approaches an empirically-chosen limit in the case of DLT, the probability of a higher cost task pool chosen as "current" decreases exponentially. At the end of the algorithm, the best encountered solution is used to define the task pool.

The simulated annealing algorithm is performed on the master node while waiting for the worker nodes to complete the processing of their work item sets. Immediately after assigning the task pool, the master spawns a thread that waits for signals from the workers indicating that they have completed processing. The main thread of the master then generates the next task pool and performs simulated annealing continuously on it while waiting for the signal thread to indicate that all workers have finished processing their current task pool and are ready for the next. This scheme provides the benefit of simulated annealing along with ensuring that the master processor is never idle.

The pseudocode in algorithm 3 describes how the simulated annealing algorithm works.

Algorithm 3 Simulated annealing

Input: A set of work items W_i
Output: A work item set distribution *bestSolution*

current ← naive distribution of W_i
cost ← costOf(*current*)
while workers still working **do**
  trial ← random permutation of *current*
  cost ← costOf(*trial*)
  if *cost* < *bestCost* **then**
    bestCost ← *cost*
    bestSolution ← *trial*
  end if
  if F(cost, bestCost, T) == 1 **then**
    current ← *trial*
  end if
  $T \leftarrow T - 1$
end while

4.1.2 Optimal cache replacement

The data in a worker's task pool will likely exceed the capacity of the worker's memory. Therefore, the worker has a fixed-size cache of sectors that it keeps in memory that are available for use during track extraction. When a track extraction references a sector that is not currently in the cache, the sector must be read from auxiliary storage, and the cache must be updated accordingly. If the cache is full then some sector must be evicted from the cache in order to make room for the newly-encountered sector. This is standard cache behavior, but our scenario is unique in that we have prior knowledge of cache access patterns for the entire future of this task pool. A worker can analyze its task pool and see when each sector will be required for processing. This fact enables a worker to employ intelligent caching algorithms when managing its cache.

The optimal cache replacement (OCR) strategy was developed by Belady [Bel66]. In the event of a cache eviction, according to OCR, the cached item that will be used farthest in the future should be the one chosen for replacement. By always replacing the node in the cache according to this scheme, the minimal number of cache misses can be realized. The benefits DLT enjoys due to the OCR algorithm are discussed in the results section (Section 5).

4.1.3 Holistic view of DLT

DLT maintains a similar structure to its sequential counterpart but has some key conceptual differences. The pseudocode in algorithm 4 and algorithm 5 describe the overall processing sequence of the DLT algorithm.

4.2 Models

Modeling potential solutions was an essential step in the understanding of this problem. The model was used in both defining the distributed structure of the sequential algorithms and for evaluating costs in the simulated annealing algorithm.

Evaluating the cost of a task pool is a crucial role of the simulated annealing algorithm. The cost assigned to a task pool identifies the expected computational load for that task pool. The models we define in this section describe the cost function used by the simulated annealing algorithm to assign task pool cost.

Two component models make up the full execution model of a solution to the track extraction algorithm. These two models are the computational model and the I/O (or memory) model. These models are described in detail in the subsequent sections.

4.2.1 Notation

Here we define some common variables used in the following sections.

Define set S as a set of nodes, here, leaf nodes, each of which represents a sector. Define set W as the set of all subsets of S whose data represent a possible linkage, potentially resulting in a track. Each of these potential tracks requires processing. Each processor, p_i, will receive some subset of W, w_i. The processing it will perform is the track extraction work of the algorithm. Each w_i, being a subset of W, will be a set of work items. w_i's work items can be denoted individually as w_{ij}.
Now define n_i as the cardinality of set w_i:

$$n_i = |w_i|$$

Define l_i as the total number of leaf nodes, or *sectors*, assigned to processor p_i:

$$l_i = \sum_{j=0}^{n_i} |w_{ij}|$$

Each processor is responsible for processing n_i sets. Each of these sets, w_{ij}, is a set of size $|w_{ij}|$. During track extraction, each of these nodes, or sectors, in w_{ij} will be compared with each of the remaining nodes, or sectors, in w_{ij}. The processing time required for each set depends on the number of "compatible" tracklets in the

Algorithm 4 DLT master processor

function: RecursiveTreeWalk(e_1, e_2, S)
Input: First endpoint e_1 and second endpoint e_2
Input: A set of support tree nodes S
Input: A set of work items W
Output: A list Z of tracks

$S' \leftarrow \{\}; Scurr \leftarrow S$
if tree nodes in $e1$ and $e2$ are compatible: **then**
  while each $s \in Scurr$ **do**
    if s is compatible with e_1 and e_2: **then**
      if s is not a leaf node: **then**
        Add s's left and right child to the end of $Scurr$.
      else
        Add s to S'
      end if
    end if
  end while
  if we have enough valid support points: **then**
    if all of e_1 and e_2 are leaves: **then**
      create work item, add to W
      if $|W| \geq$ threshold **then**
        spawn simulated annealing thread
        distribute W when simulated annealing thread completes
      end if
    else
      Let m^* be the endpoint node that owns the most points.
      e_1'= m^*'s left child
      e_2'= m^*'s right child
      RecursiveTreeWalk(e_1', e_2, S')
      RecursiveTreeWalk(e_1, e_2', S')
    end if
  end if
end if

Algorithm 5 DLT worker processor

Input: A set of current work items W
Output: A set of tracks, T

for each w in W **do**
  load first, p, and second, q, endpoint nodes into cache
  load all support nodes, S, into node cache, or as many as cache can hold
  for each s_i in S **do**
    if s_i is not in the cache **then**
      if cache is full **then**
        identify cached node which will be used farthest in the future, c
        load s_i into cache slot occupied by c
      else
        load s_i into next empty cache slot
      end if
    end if
    if s_i is a valid support point for p and q **then**
      add s_i's data to the track comparison data structure, D
    end if
  end for
  if p, q, and D are a potential track **then**
    add track to T
  end if
end for

endpoint sectors. Define this quantity as c_{ij}. Tracklets are "compatible" if an object could feasibly exist in the space defined by each tracklet at the tracklet's time, where feasibility is based on data-specific object movement threshold variables such as acceleration and velocity.

4.2.2 Computational model

The computational model defines the cost of a solution in terms of computation. Through empirical analysis we were able to accurately define the computational requirements of a solution with great precision. During track extraction a work item is evaluated. It was found that the amount of time required for track extraction on a given work item could be modeled as a function of the total number of sectors and the number of tracklets contained in the two endpoint sectors that were feasible endpoints for a track, or were compatible.

The following formula defines this relationship more formally.

$$p(w_i) = k \sum_{j=0}^{n_i} c_{ij} \times |w_{ij}|$$

Note the addition of k. This is defined as the constant value associating real CPU time with the time units produced by the model.

4.2.3 I/O model

The I/O model measures the number of required storage accesses for a processor. This is the final model considered for the execution model. Large numbers of work items are assigned to the processors, and each work item is, in turn, comprised of large numbers of sectors. For a single track extraction computation, a processor needs to have all of the work item's data in memory.

For each successive work item a processor performs track extraction on, it must load that work item's tree node data into memory. However, if the current work item and the previous work item have tree node data in common, then the number of storage accesses can be reduced. So, for a given work item, the cost, in terms of storage accesses, is the total number of sectors in the work item minus the number of sectors already loaded into memory by the previous work item. The only exception to this is the first work item processed, as this work item must assume the processor's memory contains no useful data.

Therefore, for processor p_i, the I/O time can be modeled as:

$$m(w_i) = L \times (|w_{i0}| + (\sum_{j=1}^{n_i} |w_{ij}| - (|w_{ij}| \cap |w_{ij-1}|) - |S|)))$$

$$\forall\ k; k \leq 1 \leq j; S_k = w_{ik} \cap w_{ik-1}, S = S_k \cap S_{k-1}$$

where L is the constant value representing the cost to load a sector's data from storage.

We will now establish the worst-case memory model for processor p_i. The worst case occurs when p_i is assigned a work item set, w_i, in such a way that all work items, w_{ij}, are comprised of entirely unique tree nodes.
We define set P and u_i as follows:

$$P = \bigcup_{j=0}^{n_i} w_{ij}$$

$$u_i = |P|$$

u_i is the total number of **unique** leaf nodes, or *sectors*, from the subset w_i on processor p_i.

Therefore, the worst case is one where:

$$u_i = l_i \ ,$$

that is, the number of unique tree nodes is equal to the total number of tree nodes for processor p_i.

With intelligent work item distribution, however, we can do much better. In regards to memory, then, the goal of work item distribution algorithm is to distribute task pools, w_i, while minimizing u_i. The lower bound on u_i, and thus the best-case distribution in terms of memory requirements, is

$$max(\forall \ j \in |w_{ij}|) \ .$$

4.2.4 Execution Model

The execution model naturally follows from the previous models. Each work item will need to be computed, communicated, and will perform I/O according to the defined models. As such, the full cost to perform track extraction on a work item is the sum of the models.

$$f(w_i) = p(w_i) + m(w_i)$$

4.3 Goal

With the distributed algorithms and models defined, we now return to the goal of DLT: distributing work items while minimizing time. Each algorithm has been shown to best address this goal. Simulated annealing discovers an intelligent task pool distribution, with respect to the execution model. Optimal cache replacement ensures that the workers minimize their I/O costs. The execution model enables us to define

how all these algorithms converge to work towards the goal of minimizing with respect to

$$min(\sum_{k=0}^{x}(max(\forall\ i; i \leq 0 < p; C_i)_k))$$

where p is the number of processors and x is the number of distributions.

In the following sections we will demonstrate that these algorithms meet the goals described here.

Chapter 5

Analysis and Results

The algorithms of DLT provide a scalable mechanism for increased workloads. In the following sections we define how performance is measured and present and analyze the results according to these metrics.

5.1 Execution environment

The data presented in the subsequent sections was gathered from DLT executions performed on a shared academic research cluster. The cluster is comprised of ten machines, each with access only to its own local memory. Cluster machines are equipped with 2GB of memory and have dual-core 64-bit Intel processors capable of 3.0GHz per processor.

5.2 Metrics

There are two key algorithms whose execution has the largest impact on total execution time: simulated annealing and optimal cache replacement.

Simulated annealing has the main goal of assigning task pools in such a way that total execution time, according to the execution model, is minimized. To best demonstrate the benefits of this algorithm, total execution times can be compared in the scenarios when simulated annealing is performed and when simulated annealing is not performed. In the latter case a naive, cyclical, distribution is used.

Optimal cache replacement minimizes the number of I/O operations performed as the result of cache misses. These I/O operations are much costlier than memory accesses and will result in increased execution times. To best demonstrate the benefits of this algorithm, total execution time is compared in the scenarios where optimal cache replacement is employed and where it is not. In the latter case, a naive cache replacement strategy is employed where nodes are chosen at random for cache replacement.

Finally, scalability is crucial for parallel systems. To show that DLT is scalable we will demonstrate that as worker processors are added run times are decreased proportionally.

5.3 Results

5.3.1 Simulated Annealing

The graph in Figure 5.1 shows total execution time with the simulated annealing algorithm disabled and enabled. The key to this graph is the gap in processing times between executions which utilize simulated annealing and those that do not. As the task pool size increases, the gap between execution times grows. This behavior is due to the granularity at which simulated annealing is able to balance the load. With larger task pool sizes there are fewer distributions and therefore less situations where workers are waiting for the others to complete their processing. With smaller task pool sizes the load imbalance that results from the absence of simulated annealing is a larger factor. This imbalance causes longer execution times because there are more task pools and correspondingly more instances where workers are waiting for others to finish.

Simulated annealing evenly spreads the load so that even when task pool sizes are low, the load is evenly balanced, and worker idle time is reduced.

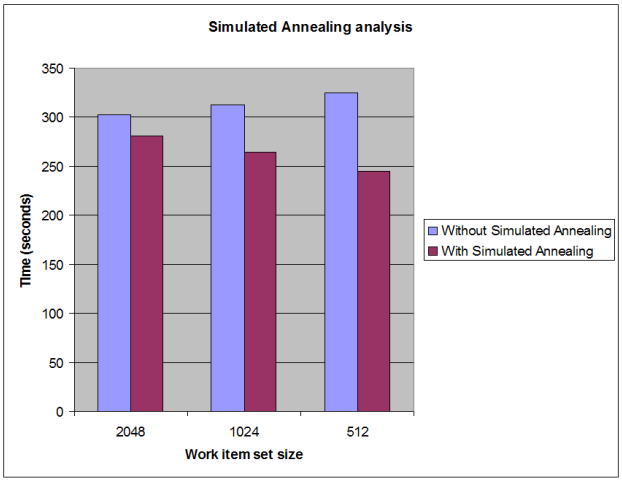

FIGURE 5.1. Simulated Annealing reduces overall execution time by intelligently assigning work to processors

5.3.2 Optimal cache replacement

The graph in Figure 5.2 shows total cache misses with optimal cache replacement algorithm disabled and enabled. As cache sizes increase, the number of total cache misses decreases. Regardless of cache size, optimal cache replacement significantly reduces cache misses over naive replacement strategies.

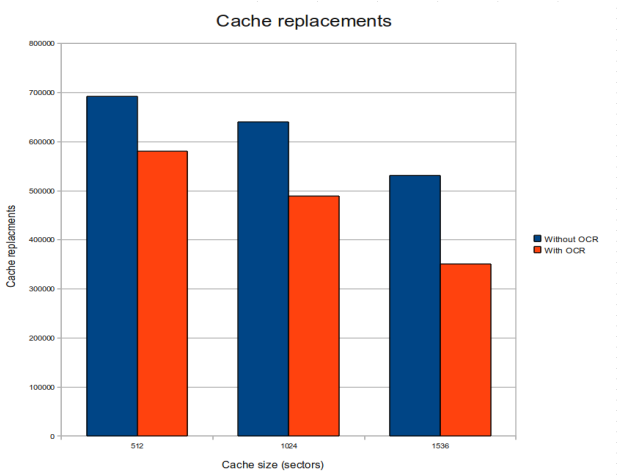

FIGURE 5.2. Optimal Cache Replacement

5.3.3 Scalability

The graph in Figure 5.3 shows total execution time when 1, 2, 4, and 8 worker processors are utilized. As evidenced by its logarithmic trend, as worker processors are added total execution time decreases. This trend shows how DLT is designed to scale linearly and efficiently utilize large numbers of processors to reduce total execution time.

The following figure shows total execution time as the sum of time taken by the master, which are the bottom dark segments of the bars, and the time taken by the workers, which are the top lighter segments of the bars. The amount of time spent in the master is roughly constant regardless of the number of workers. The amount of time the workers spend processing is the item of focus in this graph and is where the exponential time savings are realized.

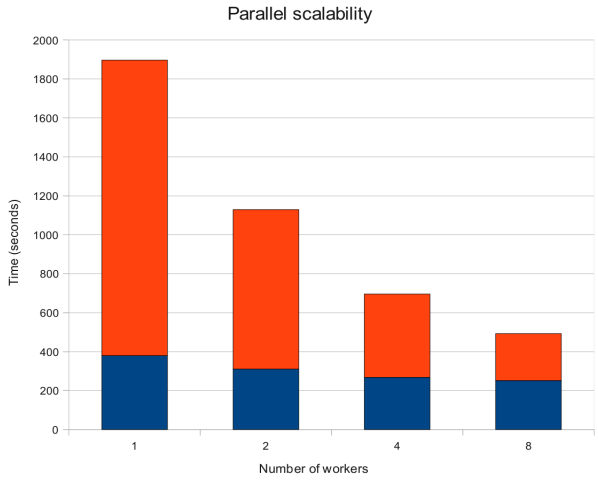

FIGURE 5.3. Parallel Scalability

Chapter 6

Conclusion

In this thesis we show that the algorithms designed are well-suited for distributed track discovery systems. Simulated annealing effectively reduces run time by balancing the processing load among the worker processors. By performing simulated annealing in its own thread that executes while the master is waiting on the workers to complete their processing, the overhead for this algorithm is effectively eliminated.

Optimal cache replacement significantly reduces the number of cache misses on the worker processors. This lowered cache miss rate enables worker processors with limited memory resources to operate on work items of arbitrary size while minimizing the penalty incurred by the auxiliary storage accesses associated with cache misses.

Finally, DLT's performance scales proportionally to the number of processors utilized. This scalability enables DLT to effectively utilize processor pools of arbitrary size to decrease overall execution times.

Chapter 7

Future Work

Exciting challenges have arisen over the course of this work but were outside the scope of the project.

One assumption that is made by DLT is that any given work item will fit in memory. If this assumption doesn't hold then the execution models will have to be re-examined. The ripple effect of such a change will likely present challenges to the caching scheme, as well.

Another interesting topic that was researched briefly during the course of this thesis was the study of the construction of the trees used by the tree walk algorithm. Execution time varies depending on the number of tracklets held in the leaf nodes of the trees. However, picking a leaf node size for a tree is not easy, as an optimal leaf node size varies depending on the data. The possibility of having a collection of trees, each with a different leaf node size, means that, again, the execution models may have to be revisited to take into account such variations.

The simulated annealing algorithm currently uses a static, empirically chosen, value for its temperature variable. The temperature could be initialized dynamically based on the input data, or other factor. Parameters and algorithms for this procedure are areas of future study. Additionally, the relationship between temperature and the probability factor of accepting a solution is static. This relationship could be reconsidered and the temperature could affect the probability factor in a dynamic, data-specific manner. Determining this relationship and which factors are significant is a future realm of study for this work.

Chapter 8

Acknowledgments

Dave has been incredibly patient and understanding throughout this work. I truly appreciate his input on the design of our system and his good-natured approach to the whole project. Jon has been a tremendous help throughout the entire project. From design tips to procedural advice to his great knowledge of the astronomical data which we processed, without Jon this project would have suffered greatly.

Also I must thank Dr. Barnard and Dr. Efrat for introducing me to Jon in the first place. Without their reference this project would not be where it is today.

Thanks are in order to Jeffrey Kantor for allowing me to "intern" (provide free labor, in his words) for LSST one summer and thus provide an opportunity to work with this exciting project.

Finally, thanks to my wife Jenna. Her well of encouragement and understanding is truly bottomless and really helped this project reach fruition.

References

[Bel66] Laszlo Belday. A study of replacment algorithms for a virtual-storage computer. *IBM Systems Journal*, 5:78–101, 1966.

[ITA+08] Z. Ivezic, J.A. Tyson, R. Allsman, J. Andrew, R. Angel, and for the LSST Collaboration. Lsst: from science drivers to reference design and anticipated data products, 2008.

[Kub05] Jeremy Kubica. *Efficient Discovery of Spatial Associations and Structure with Application to Asteroid Tracking*. PhD thesis, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, December 2005.

[Mye08] Jonathan Myers. Methods for solar system object searching in 'deep stacks'. Master's thesis, The University of Arizona, 2008.