Subset Removal On Massive Data with Dash

Jonathan Myers
Large Synoptic Survey Telescope
933 N. Cherry Ave
Tucson, AZ 85721 USA
jmyers@lsst.org

Robert Sinkovits
San Diego Computing Center
9500 Gilman Dr., MC0505
La Jolla, CA 92092, USA
sinkovit@sdsc.edu

Mahidhar Tatineni
San Diego Supercomputing Center
9500 Gilman Dr, MC0505
La Jolla, CA 92092, USA
mahidhar@sdsc.edu

ABSTRACT
Ongoing efforts by the Large Synoptic Survey Telescope (LSST) involve the study of asteroid search algorithms and their performance on both real and simulated data. Images of the night sky reveal large numbers of events caused by the reflection of sunlight from asteroids. Detections from consecutive nights can then be grouped together into tracks that potentially represent small portions of the asteroids' sky-plane motion. The analysis of these tracks is extremely time consuming and there is strong interest in the development of techniques that can eliminate unnecessary tracks, thereby rendering the problem more manageable. One such approach is to collectively examine sets of tracks and discard those that are subsets of others. Our implementation of a subset removal algorithm has proven to be fast and accurate on modest sized collections of tracks, but unfortunately has extremely large memory requirements for realistic data sets and cannot effectively use conventional high performance computing resources. We report our experience running the subset removal algorithm on the TeraGrid Appro Dash system, which uses the vSMP software developed by ScaleMP to aggregate memory from across multiple compute nodes to provide access to a large, logical shared memory space. Our results show that Dash is ideally suited for this algorithm and has performance comparable to or superior to that obtained on specialized, heavily demanded, large-memory systems such as the SGI Altix UV.

1. MOTIVATION
1.1 Asteroid Search Simulation
In order to better understand the behavior of known asteroid search and discovery systems and algorithms, the Large Synoptic Survey Telescope (LSST) Data Management team have been applying those systems and algorithms to synthetic astronomical observations conducted with a simulated version of the LSST. For asteroid search purposes, these synthetic observations are generated using a realistic model of the solar system [3], a proposed survey cadence from the telescope [1], and realistic limits on the data-gathering abilities of the LSST [6].

The algorithms currently under investigation are based on generating sky-plane tracks [5]. A track is essentially a set of astronomical detections on the sky which follow a path in the sky consistent with some model of asteroid motion. Initial phases of processing use a greatly simplified model of asteroid motion in order to find initial sets of tracks with relatively low computational cost [4], and as a result the sets of tracks generated are very large, containing many tracks which incorrectly link sets of detections. Later in processing, more precise, but also more computationally costly, models of asteroid motion are used to filter out these incorrectly linked tracks and derive more precise approximations of underlying motion [2]. At the end of processing, tracks should have highly precise associated orbital paths [7] [9].

In order to better understand the behavior of these algorithms and their relationship to LSST's observational cadence (the schedule of observations of the sky) and imaging systems, we have been attempting to characterize the sets of tracks generated by the various stages of processing. Using this knowledge, we hope to adjust our models, filters, and algorithms to find less computationally costly methods of generating and processing these asteroid tracks.

1.2 Subset Tracks and their Identification
It is a known issue that given certain patterns in source data, some algorithms generate tracks which are subsets of other tracks; that is, they link together a set of detections already linked by another, higher-cardinality track. This can create an artificial and unnecessary inflation of the set of tracks, leading to needlessly increased downstream cost. Unfortunately, the prevalence of these subset tracks is not easily predicted. This makes exhaustive subset removal a necessary step in advancing our understanding.

2. SUBSET REMOVAL DESCRIPTION AND ALGORITHM
The abstract problem of subset removal is fairly straightforward. Given a set of tracks, we wish to find and remove those items which are subsets of sets already present in the collection. That is, given an n-element set of tracks $C$ containing tracks $t_0, t_1, \ldots, t_n$,

$$\text{noSubsets}(C) = \{ t_i : t_i \in C, \forall t_j, C, t_j \neq t_i : t_i \not\subset t_j \}$$
A naïve approach to subset removal would involve the direct pairwise comparison of all tracks. While this might be practical for small data sets, the quadratic scaling in time with problem size makes this infeasible for large collections. In our data, we can take advantage of the fact that the number of detections is small compared to the number of tracks (in practice, most detections are typically found in 100 to 1000 tracks) and formulate an algorithm that is better suited to this problem.

Let $D$ and $T$ be the sets of detections $\{d_1, d_2, \ldots, d_m\}$ and tracks $\{t_1, t_2, \ldots, t_n\}$, respectively. We first create a map $M$ that associates each element of $C$ with a subset of $D$ such that $M[t]$ is the set of detections belonging to track $t$. From this map we can then generate a reverse map $R$, where $R[d]$ is the set of tracks in which detection $d$ is found. To determine if a track $t$ is subsumed by any other tracks, we evaluate the intersection over $R[d_i]$ for $d_i \in M[t]$. The resulting set contains all of the tracks, including $t$ itself, that are super-tracks of $t$.

As a concrete example, consider track $t_1$ from Figure 3. Using the procedure described above, we can show that $t_1$ is a sub-track of $t_3$ and should therefore be deleted from the collection.

$$M[t_1] = \{d_1, d_2, d_3\}$$

$R[d_1] \cap R[d_2] \cap R[d_3] = \{t_1, t_2, t_3\} \cap \{t_1, t_3, t_4\} \cap \{t_1, t_3\} = \{t_1, t_3\}$

Replacing the direct pairwise comparison of tracks with the algorithm just described involves a time-space trade off. The tracks are always accessed sequentially and the map $M$ can be held in a simple list without incurring performance penalties. The total number of items to be stored is slightly greater than the sum of the sizes of the tracks. When running the naïve algorithm, it would be advantageous to have enough memory to accommodate the entire data structure, but it is not essential given the regular patterns of data access.

Since the improved algorithm requires frequent calculations of set intersection, it is imperative that we be capable of performing this calculation very quickly. It is possible to calculate set intersection in $O(n + m)$ time given two $n$– and $m$–element sorted data structures. In order to ensure all entries are sorted and allow this quick set intersection calculation, each entry in $R$ holds a dynamically sorted set structure based on red-black trees. This imposes a linear overhead on the storage costs of each element in $R$. Further, the improved algorithm requires us to randomly access elements of $R$. Rather than using a simple list, the reverse map is also indexed via a red-black tree, thereby reducing the search time from $O(n)$ to $O(\log n)$. This imposes another linear overhead in our memory cost for the indexing of $R$. Since accesses to data are frequent, it is crucial that $R$’s index and its entries fit comfortably in memory; frequent accesses to remote storage will cripple performance.

for $t_i \in C$ do
  for $d_j \in t_i$ do
    $R[d_j] = R[d_j] \cup t_i$
  end for
end for

Figure 1: Pseudocode for the creation of the reverse map

for $t_i \in C$ do
  candidates = $C$
  for $d_j \in t_i$ do
    candidates = candidates $\cap R[d_j]$
  end for
  if $|\text{candidates}| > 1$ then
    $t_i$ is a subset of some other $t_j \in C$; discard it
  else
    keep $t_i$
  end if
end for

Figure 2: Pseudocode for the improved subset removal algorithm

<table>
<thead>
<tr>
<th>Track</th>
<th>Component Detections</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$</td>
<td>$d_1$, $d_2$</td>
</tr>
<tr>
<td>$t_2$</td>
<td>$d_1$, $d_4$</td>
</tr>
<tr>
<td>$t_3$</td>
<td>$d_1$, $d_2$, $d_3$</td>
</tr>
</tbody>
</table>

(a) Contents of $M$

<table>
<thead>
<tr>
<th>Detection</th>
<th>Parent Tracks</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d_1$</td>
<td>$t_1$, $t_2$, $t_3$</td>
</tr>
<tr>
<td>$d_2$</td>
<td>$t_1$, $t_3$</td>
</tr>
<tr>
<td>$d_3$</td>
<td>$t_3$</td>
</tr>
<tr>
<td>$d_4$</td>
<td>$t_2$</td>
</tr>
</tbody>
</table>

(b) Contents of $R$

Figure 3: Some examples of $M$ and $R$ for a trivial data set.

3. DASH AS A CANDIDATE MACHINE FOR RUNNING SUBSET REMOVAL

3.1 The Need For a Large-Memory Machine

We discovered that for many of relevant sets of tracks, the size of the reverse-map was prohibitively large, exhausting memory on our available machines and causing intolerably slow swapping. In order to deal with this problem, we were faced with several options. The most apparent two options were to find a machine with a larger available memory, or to attempt to develop a distributed-memory version of the algorithm. With no distributed-memory version of this (or other) subset removal algorithm known to us, the latter would be extremely costly, requiring research, development and potentially challenging software implementation.

Conventional large-memory machines featuring uniform memory access (UMA) were one feasible option provided to us by TeraGrid. But such machines are uncommon and in high demand; this meant potentially long waits to get access to, and results from, such a machine, a significant inconvenience in our ongoing experiment cycle.
This leads us to investigate a novel non-uniform memory access (ccNUMA) Teragrid resource known as Dash.

### 3.2 About Dash
Dash is an early prototype of the Appro Gordon system that will be deployed at SDSC in late 2011 [8]. One feature of Dash that makes it particularly well suited for memory intensive applications is that it uses the vSMP software developed by ScaleMP to aggregate memory from across multiple compute nodes to provide access to a large, logical shared memory space. Each compute node contains two quad-core 2.4 GHz Intel Nehalem processors (E5530) and 48 GB of DDR3-800MHz memory. Benchmarks on Dash were run using a single processor on a 16-node vSMP partition with a total memory of 768 GB. The actual memory available to applications is approximately 650 GB since 118 GB are used by the vSMP software for data caching. Non-local memory accesses trigger the migration of a complete page (4KB) of data from the neighboring board. As a consequence, applications that take into account of the NUMA architecture and employ regular patterns of data access achieve the best performance.

### 4. PERFORMANCE
Though our subset removal software was originally written for a conventional architecture, Dash proved an excellent resource for running this algorithm, demonstrating excellent scaling over data sets of varying sizes. Somewhat surprisingly, it also demonstrated similar (or even better) performance than comparable UMA machines.

Determining whether a given track \( t \) is a subset track requires \(|t|\) lookups in \( R \) and \(|t| - 1\) set intersection calculations. Thus, to predict the expected computation time of a data set \( C \), we chose to count \( \sum_{t \in C} |t| \). We shall refer to this value as the **net track size** of \( C \). This does not account for some other factors, such as the cost of the individual set intersection calculations, which cannot be known a priori, but it has proven to be a reasonable metric for predicting the cost of processing a data set.

### 5. DISCUSSION
As previously mentioned, fast set intersection calculation requires inputs sets be kept in sorted order. In our implementation of the subset removal algorithm, the sorted order of each entry in \( R \) is maintained by dynamically balancing trees. The use of tree structures imposes linear, but not insignificant, storage overhead. During the course of this work, we realized that there might be more memory-efficient implementations of our algorithm. An alternative would be to store each entry as a flat structure (such as an array or vector) when building \( R \), then make a second pass over \( R \) to sort these entries. After this sorting, the fast set intersection can still be used, but the storage overhead of the red-black trees could be avoided. Currently, we have not been able to test the potential reduction in storage overhead possible through this change, nor the potential increase in cost associated with this second pass over \( R \).

Regardless of whether these potential optimizations could have reduced our memory requirements, the move to Dash allowed LSST to rapidly complete its subset removal experiments without the need to develop new algorithms or software. This set of experiments yielded a deeper understanding of the relationship between the telescope’s cadence, the algorithms used, and their computational costs. We also identified the real prevalence of subset tracks, allowing us to create realistic cost estimates with and without subset removal included in our system, and to precisely measure the potential benefits of adopting algorithms that do not generate subset tracks.

This work confirmed that Dash is an excellent tool for running our subset removal algorithm. While the utility of this algorithm has been demonstrated in the context of asteroid surveying, it should be useful for any project that requires the identification of subset entries in large collections.
of sets. We also showed that Dash gave performance comparable to that obtained on expensive, heavily subscribed hardware such as the SGI Altix UV, thereby providing a very cost-effective alternative.

6. ACKNOWLEDGMENTS
The authors thank Shawn Strande and Arun Jagatheesan, Jeff Kantor, Lynne Jones and Tim Axelrod for their role in initiating this collaboration. This work was supported in part by NSF Grant OCI #0910847 entitled “Gordon: A Data Intensive Supercomputer”, and by the non-profit LSST Corporation, which is funded in part by the NSF, Department of Energy, private donors and a variety of organizations.

7. REFERENCES