Parallel & Distributed Performance of a Depth Estimation Algorithm

B. R. Calder

***Abstract*—Expansion of dataset sizes and increasing complexity of processing algorithms have led to consideration of parallel and distributed implementations. The rationale for distributing the computational load may be to thin-provision computational resources, to accelerate data processing rate, or to efficiently reuse already available but otherwise idle computational resources. Whatever the rationale, an efficient solution of this type brings with it questions of data distribution, job partitioning, reliability, and robustness.**

This paper addresses the first two of these questions in the context of a local cluster-computing environment. Using the CHRT depth estimator, it considers active and passive data distribution and their effect on data throughput, focusing mainly on the compromises required to maintain minimal communications requirements between nodes. As metric, the algorithm considers the overall computation time for a given dataset (i.e., the time lag that a user would experience), and shows that although there are significant speedups to be had by relatively simple modifications to the algorithm, there are limitations to the parallelism that can be achieved efficiently, and a balance between inter-node parallelism (i.e., multiple nodes running in parallel) and intra-node parallelism (i.e., multiple threads within one node) for most efficient utilization of available resources.

***Index Terms*—Distributed Processing, Parallel Processing, CHRT, Data Scheduling, Spatially-Aware Data Distribution, Bathymetric Estimation**

I. Introduction

ONE consistent trend in hydrographic data capture within the last decade has been a consistent increase in the density and volume of data being captured in the field. While this can significantly improve the quality of the representation of the seafloor that can be achieved, it comes at a price in terms of data storage, manipulation and archival. Although the performance of computers, discs and networks have improved, and there are now a number of algorithms that can assist in processing the data, the total processing time for larger datasets is still significant. In fact, the use of more algorithmic approaches to hydrographic data processing has meant that much of the work is now done by algorithms, but in practice this can mean that the user is free only to watch the progress bar advance between inspection cycles. In addition to reducing the computing lag, faster processing methods have additional benefits. For example, if the processing is sufficiently fast that a significant portion (ideally, all) of the data can be reprocessed in a human-scale time (say, on the order of a second or so), then the data inspection task could be made interactive, with the results of modifications to the data being reported in real-time. This would significantly improve the efficiency of the data inspection task, since the user would not have to maintain a mental picture of the data during compute cycles. Faster compute cycles would also allow the user to experiment with 'what if' scenarios, changing, e.g., the patch-test solution to investigate motion sensor anomalies. Or, it could be possible to use the output of a computational algorithm as the objective function in an optimization scheme to automatically adjust some of the solution parameters to give the best possible solution. Clearly, there is a need to improve the performance of the algorithms in use.

Setting aside rebuilding the algorithm, there are a number of potential opportunities for improving the performance by running it in parallel. Most processors currently in use have more than one processing core, and even low-end desktop machines typically have four. If the algorithm in question can be adapted to run segments in multiple processes, or threads, then it is possible to improve the overall performance to some extent. It is also now common to have multiple machines on a sufficiently fast network to consider distributing a large scale processing task across multiple machines and thereby improve the overall performance (taking into account the extra communications costs that this entails). If most of the machines are idle for much of the time (e.g., because they are used for intermittent tasks, or because they are primarily IO bound) then these machines need not even be dedicated to the processing task, which can utilize their otherwise unused compute cycles. A dedicated cluster of machines to carry out processing is not impossible, however, since the material cost of such clusters has dropped substantively, particularly for blade server clusters (i.e., a collection of identical computers without cases mounted in a single chassis with network connectivity and hard discs). Such clusters also help with robustness, since it is easy to hot-swap computers and to maintain a standard disc image that can be used to restore them to a well-known state in case of any corruption. Finally, the rise of cloud computing, such as Amazon EC2[1], Rackspace[2], Google Compute Engine[3] and Microsoft Azure[4] among others, mean that for shore-based processing centers, it may be possible to deploy computational tasks over an essentially unlimited compute resource, given sufficiently fast data ingestion processes.

The question, of course, is how to organize both the computational structure of any given algorithm to take advantage of the available computational resources, and how

B. Calder is with the Center for Coastal and Ocean Mapping and NOAA-UNH Joint Hydrographic Center, University of New Hampshire. Address: Chase Ocean Engineering Lab, 24 Colovos Road, Durham NH 03824. E-Mail: brc@ccom.unh.edu. Phone: +1-603-862-0526.

[1]http://aws.amazon.com/ec2
[2]http://www.rackspace.com
[3]https://cloud.google.com/products/compute-engine
[4]http://www.windowsazure.com

to best distribute the data and computational effort to avoid bottlenecks and inefficiencies. Parallel versions of algorithms are intrinsically more complex than serial, and must pay attention to problems of fair access to resources, deadlocks (i.e., where two or more sub-tasks need two or more resources to continue, but each is allocated only part of the requirement) and shared correctness (in the sense that multiple sub-tasks accessing the same resource without care can interfere with each other and generate incorrect results, although both appear to complete correctly). Considering the problem purely on the basis of efficiency, the primary difficulty is the elimination, or reduction, of bottlenecks. Any depth estimation algorithm contains a mixture of computational elements and data access. Often, the processing resources of a single machine are under-utilized, which can be improved if the task is split into multiple segments and run in parallel; if this parallelization increases the required data throughput to the stage that either the disc, the primary bus, or the local cache are exhausted, however, the performance may in fact be reduced. When distributed over multiple machines, these problems grow in scope and in addition the problem of where to place data become important. That is, it is better to think of the machines in the cluster as discs with computers attached, rather than as computers with discs attached. Then, the focus of the algorithm design becomes ensuring that the data is clustered appropriately so that all of the data required for a sub-task is available at one location, and then how to distribute the sub-tasks so as to minimize data motion between machines.

In the face of the difficulties entailed by distributing a depth estimation algorithm within a single machine, or across multiple machines, this paper investigates some of the problems that are likely to be encountered, and how they might be resolved. It proposes two variants of the CHRT algorithm [1], one for use within a single machine, and one for distribution across multiple machines. It then considers the limitations of running within one machine, the data distribution problem (where the requirements for placement are not fully known until the first phase of the computation is carried out), aspects of the efficiency of the data distribution process (such as how long it takes to compute an optimal distribution, the equitability of the achievable distributions of data, etc.) and the potential efficiency improvements, at the user level, given the proposed solutions to all of these problems. The primary goal is to assess the potential for gain in such an implementation of the algorithm, and gain some insight into the relative importance of the various issues that will be faced in the full implementation of the distributed version of CHRT.

This paper is therefore organized as follows. Section II considers the basic problems of the design of the algorithm, including an overview of the serial version of the algorithm, which is used as a base. It also considers the design issues involved in ensuring data availability at multiple machines in a cluster, and in determining how to distribute the data across a given set of machines. Section III examines the performance of the single-machine version of the algorithm, highlighting the bottlenecks that become apparent and their implications for scalability, while section IV considers the time complexity and distribution efficiency for the computation to split up the primary task into sub-tasks, and to distribute the data across the cluster as the algorithm commences. Section V examines the effective efficiency of the proposed distributed algorithm for various different methods of data distribution, using a model of the computational time that takes into account pre-fetching of data and computation/IO overlap. Finally, section VI considers the implications of the results reported, and in particular what they mean for a composite algorithm, and section VII provides a summary of the primary conclusions.

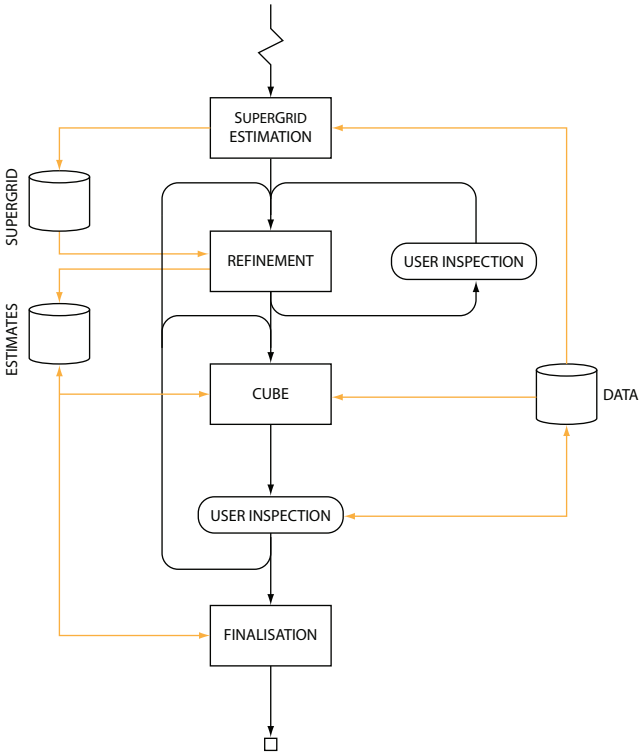

Fig. 1. Data flow for the CHRT algorithm, with additional feedback loops to allow for operator interaction. This flow assumes that all of the data are available for processing, but similar flows can be constructed for the case where data capture is episodic, and real-time.

II. Design Issues

A. Segmenting the Algorithm

The CHRT algorithm [1], used for these experiments, is a hierarchical, data-adaptive, extension of the CUBE algorithm [2] for processing high-density, high-resolution multi-beam echosounder data. The core concept of the algorithm is to use a first-pass over the raw data to estimate the data density, and hence the resolution supportable by the data, and then construct a piece-wise regular grid on which to compute the CUBE algorithm in the second-pass over the raw data. The algorithm estimates the data density on a coarse grid (typically on the order of 30 m spacing, called the 'SuperGrid'), and then refines each cell of this grid with an embedded fine-scale grid once the estimation resolution is determined. The overall data-flow for the algorithm is shown in Figure 1.

The advantage of this construction is that each SuperGrid cell is independent of the others, and therefore does not need to communicate with them about data. It is therefore trivial to

split up the computation of the CHRT algorithm on a SuperGrid cell edge, since each segment of the overall grid can be considered essentially independent of the others. In theory, therefore, the task should be very highly parallelizable, since there is no communications overhead. Of course, in order to avoid edge effects, source survey lines that impinge on the edges of a region of the whole grid must still be available to the computation, and therefore each sub-grid must be provided with the data around it, as well as the data inside it, and there is some overhead in the sub-grid computations that reduces the effective speedup that can be achieved. In principle, however, so long as the overall grid can be partitioned into non-overlapping sub-grids, the algorithm can be parallelized simply by running each segment of the partition on a separate thread, or a separate processor.

The first phase of the algorithm, to determine the data density, is relatively straightforward. Within each SuperGrid cell, the algorithm constructs an estimate of the area of the cell used by all source survey lines that pass through it, and counts the number of soundings that impinge on it. It is relatively simple to combine partial estimates within each SuperGrid cell, and therefore the first phase can be split up purely on the basis of the size of the source lines, with no concern for spatial context. The data distribution goal for a distributed version of the algorithm is therefore to spread the data onto the computation nodes such that each node has approximately the same number of soundings, and therefore the same amount of work. This ensures, Figure 2, that the computation takes about the same time on each node, and therefore that all of the nodes will finish computation at the same time, with no wasted effort.

The second phase, however, construct CUBE estimates of depth at the fine-scale embedded grids, and it is not as simple to merge partial results. It is therefore essential that each SuperGrid cell is computed by only one sub-grid task. To ensure that this is the case, the whole computation is split into segments, each of which should have approximately the same number of soundings. Consider the diagram in Figure 3. As a by-product of the first phase of the computation, the algorithm obtains a count of the number of soundings in each SuperGrid cell, and therefore for any north-south or east-west split of the grid, it is trivial to compute the number of soundings which it contains. Given the number of nodes, P, onto which the second phase of the computation is to be split, the question becomes how to cut the whole grid into P segments, each of N/P soundings (approximately). There are any number of possible solutions to this problem, but to keep the complexity of the decision-making process in check, the algorithm only considers partitions that are aligned either north-south or east-west through the area. For P processors, and assuming an initial north-south split, there are $P-1$ possible split points, corresponding to N/P to the west of the split point, and $N(P-1)/P$ to the east, $2N/P$ to the west and $N(P-2)/P$ to the east, and so on until the last possibility of $N(P-1)/P$ to the west and N/P to the east. (For an initial east-west split, the same options are possible, with 'north' and 'south' substituted.) After the first split, each segment can then be further split either east-west or north-south, and therefore the

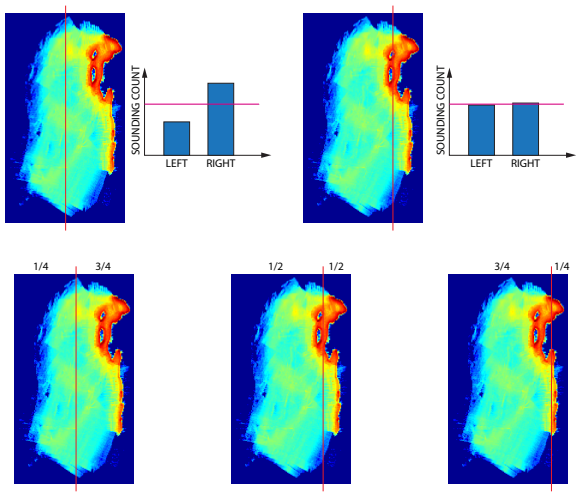

Fig. 3. Logical structure of the partitioning process. The goal is to split the overall computational task into a fixed number of sub-tasks (here, four) such that the number of soundings in each sub-task is approximately equal (i.e., as in the top right, rather than the top left). To keep things simple, the splits are made either north-south or east-west, but even this can lead to significant complexity (bottom panel) for higher numbers of sub-tasks.

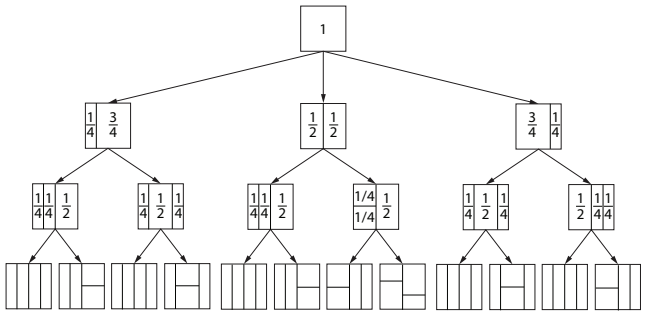

Fig. 4. Tree-structure diagram for all possible splits of a grid into four segments, starting with a north-south split. Note that many of the splits are equivalent, which can be used to advantage to reduce the amount of time required to find the optimal split.

total set of all possible splits forms a strongly structured tree, Figure 4.

The number of possible splits quickly increases with the number of nodes, and therefore it quickly becomes impossible to enumerate all of the potential splits and their relative merit (measured by how close the sounding counts are to the nominal N/P). The redundancy evident in Figure 4 means that not all of the potential splits have to be considered, however, and it is possible to construct a branch-and-bound algorithm [3] to evaluate a minimal sub-set of possibilities, which considerably improves the run-time of the algorithm.

A secondary consequence of the requirement that any SuperGrid cell only be computed by one sub-task is that it is impossible to perfectly partition the data used in the computation. The requirements for data around the edges of the sub-grid under consideration mean that one source survey line may be required for two different sub-grids, and consequently that it will most likely be required at more than one processor node. This implies that the algorithm must have the capability to obtain the data for any survey source line at any node, that the lines must have a consistent representation

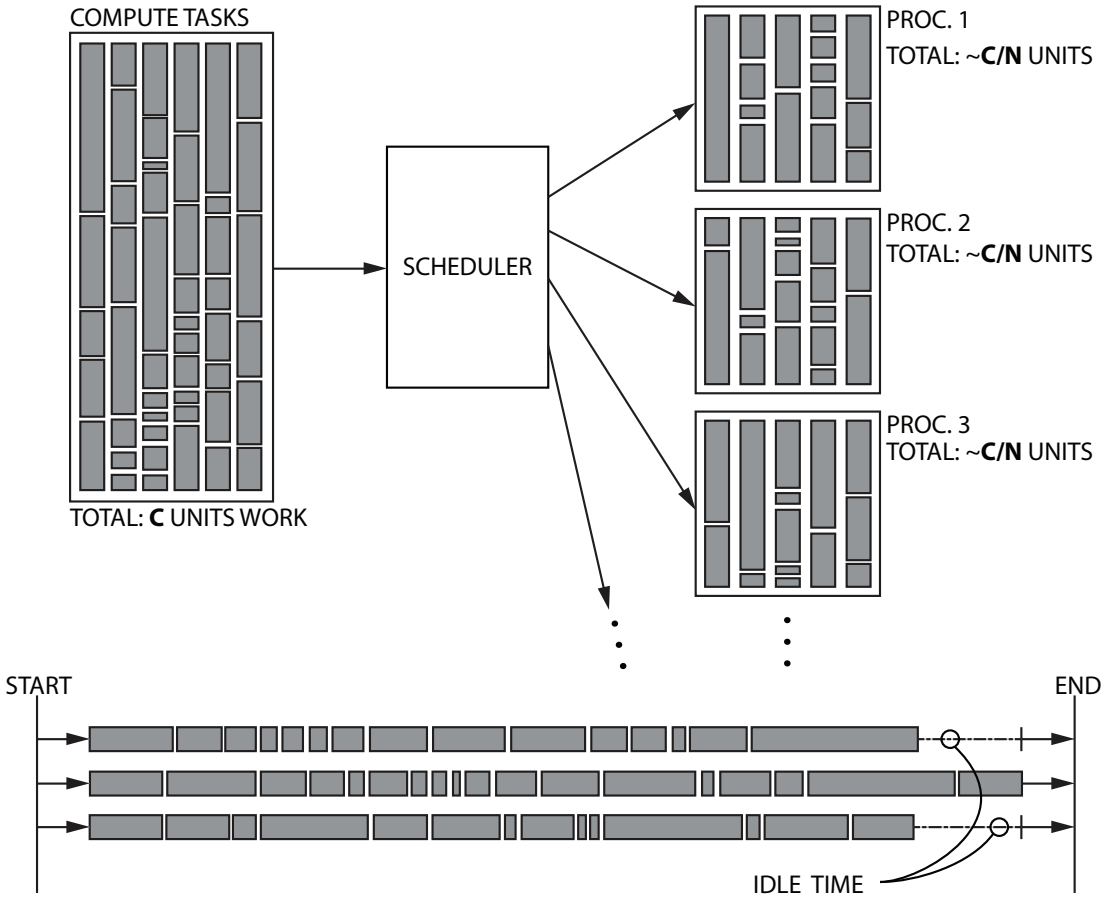

Fig. 2. First phase scheduling of lines to nodes. Since the first phase of the algorithm does not require any spatial context, and can combine partial results at the SuperGrid cell level, no special precautions on distribution are required except to keep the distributions relatively even so that there is no idle time in the computation (bottom panel).

across all of the nodes in the cluster, and that it would be a good idea to distribute the data lines for the first phase of the algorithm, and the sub-grids for the second phase so as to minimize the data movement required to allow the second phase to execute. These requirements are considered in the next two sub-sections.

B. Ensuring Data Availability

The simplest solution to ensure that each node had access to all survey source lines would be simply to have all of the lines sourced from a single data store, typically a Storage Array Network (SAN). This would have limited scalability, however, since the network bandwidth available would eventually form a bottleneck. The alternative adopted here is strongly influenced by the Hadoop[5] model of a distributed file system [4], although significantly simplified for the current application, Figure 5. (Note that Hadoop itself is not an option due to some source library dependencies in third-party software.)

In this model, one of the nodes in the cluster is considered 'special' and runs a variant of the software that maintains a database of which survey source line is available on each node, and provides consistent naming services so that each source line is available with the same name on each node (this name is a unique identifier, not necessarily the reference name of the line; the lead node maintains a lookup table for translation). The lead node also provides the ultimate source of data from the external data store, and provides the user interface. In the context of the CHRT algorithm, full name lookup is not a required service, since the lead node knows at each time exactly which survey source lines are going to be distributed to each processor node and where they are located; each assignment of a survey source line to a processor node can therefore be augmented with all of the information required for the processor node to find the source line from one or more of its peers (due to the consistent naming scheme). This ensures that each processor node can access data from one of its peers when required, greatly increasing the network bandwidth through the adoption of a mesh rather than bus network topology.

In practice, it is often the case that the lead node will know all of the survey source lines that will be processed by a particular sub-grid task before they are assigned (e.g., if the lines for a dataset are presented as a batch). In this case, the lead node can pass out this information to the processor nodes early, allowing them to pre-fetch the required data in the background as the primary computation proceeds. Depending on network congestion, line order and size, this may be able to avoid some pipeline stalls that would otherwise occur in the

[5]http://hadoop.apache.org/

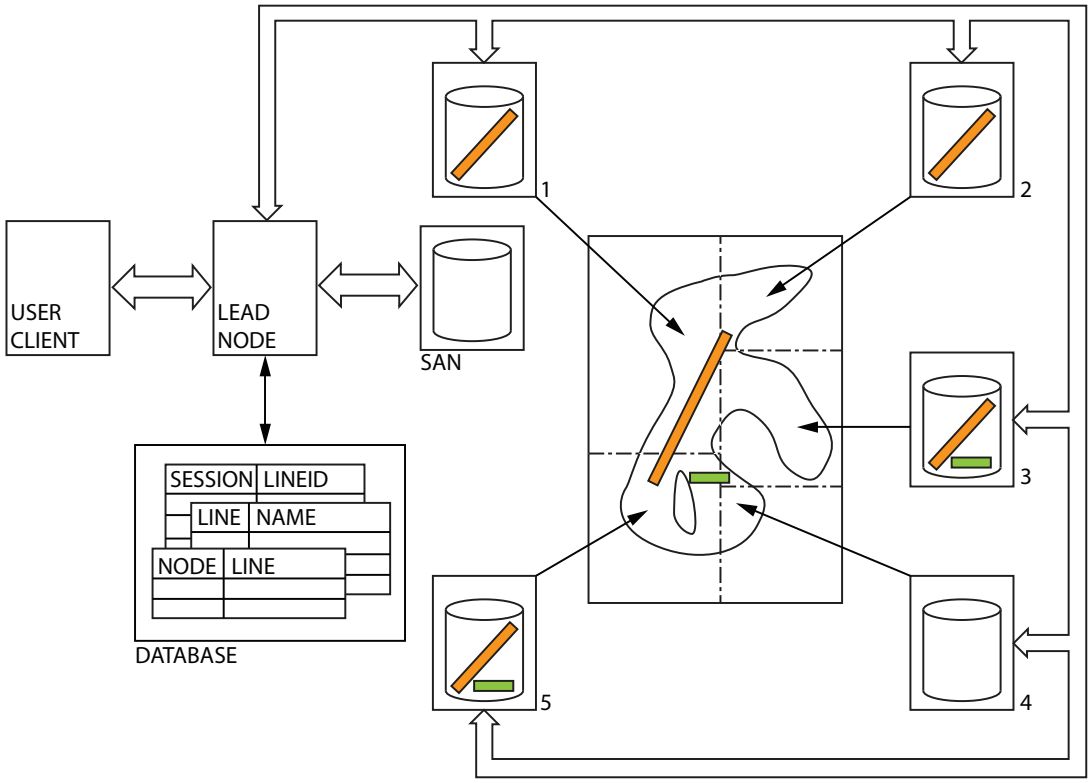

Fig. 5. Structure of the data distribution requirements for CHRT. The lead node maintains a database of which survey line is available on each node, and provides consistent naming for all data; the ultimate source of survey data is a large external store which is reflected into the cluster by the lead node. Each processor node is supplied with a copy of all of the lines required for its computation (orange and green rectangles), but can fetch them from peer processor nodes, rather than having to obtain them from the external store (in this example, a SAN).

computation, and therefore improve the overall performance.

C. Distributing Data

The logical model of processors here is discs with computers attached: the processing follows the data, so that all processing works on data that is local to the CPU. The algorithm, therefore, needs to be able to determine how to distribute the data to the nodes, and then how to distribute the sub-grid processing tasks so as to maximize data affinity. Ideally, the distribution should cluster data that is likely to be used by the same sub-grid processing task; unfortunately, however, the data has to be distributed in the first phase of the algorithm, and the survey source lines required for each sub-grid task are not know until after the first phase has completed. Consequently, the algorithm is by necessity approximate.

Distribution of data or tasks to multiple processors is a very well studied problem in computer science, and numerous variants of scheduling algorithms exist depending on the constraints of the problem (e.g., limited capacity on any one processor, maximum number of processors that can be committed, sequence of processing steps that must be achieved, etc.) [5], [6]. The scheduling problem, however, is NP-hard, meaning that it is very difficult to derive an algorithm that determines the optimal schedule in a reasonable (meaning polynomial) time. Many approximate algorithms therefore exist, such as the Longest Processing Time algorithm [7] which in this context would sort the survey lines by number of soundings (which is known *a priori*) and then assign them in order to the processing node with the minimum of currently committed soundings to process. A more useful variant is MULTIFIT [5], [8], which sorts the survey lines as before, but then assigns them to the processing nodes so as to minimize the maximum processing done at any node. Which scheduling algorithm to use in this case is not clear, particularly in attempting to construct clusters of source lines on a processor node, and forms the subject of the experiments reported in section IV-B.

Assignment of the sub-grid tasks to the processor nodes is a very similar problem, since each sub-grid should ideally be assigned to the processing nodes such that the data migration required to compute its solution is minimized. For even a few sub-grid tasks, the number of possible permutations of assignments grows very quickly, and therefore the simplest solution is to use a sub-optimal 'greedy' algorithm which makes the assignments so as to minimize at each stage the expected run-time of any sub-grid on any processor node, eliminating both sub-grid and node from further consideration afterwards. Since the lead node knows which source lines are available at every processor node, and the set of lines required to compute each sub-grid task, it can readily estimate the expected run-time for the algorithm, assuming that the pre-fetch algorithm described previously will be running, and will attempt to fetch all available lines required as quickly as possible. Ignoring network delays, the algorithm can compute a matrix of runtimes for each sub-grid task and processor node, and therefore the assignment problem becomes simply a matter

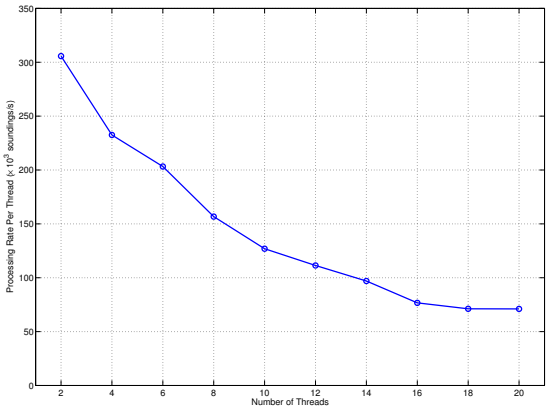

Fig. 6. Data processing rate per thread committed in thousands of soundings per second. The processing rate *per thread* decreases as more threads are added due to resource contention within the processor.

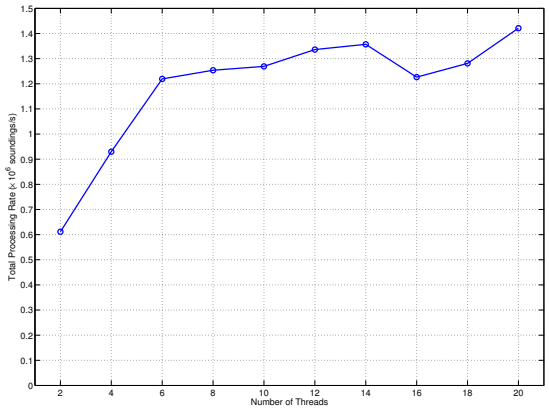

Fig. 7. Total data processing rate (over all threads) in millions of soundings per second. The total processing rate increases steadily for small numbers of threads, but then slows down and eventually stops increasing after approximately six threads are used.

to picking, at each step, the minimum element of the matrix, and keeping track of the assignments.

III. Multi-Threaded Performance

To investigate the performance of the algorithm when running in multiple threads on a single platform, a small test dataset was subject to processing on increasing numbers of threads. The data was a conventional NOAA survey, consisting of approximately 9.26×10^6 soundings from the NOAA Ship FAIRWEATHER, collected in 2009 using a mixture of Reson 8101, 8125 and 8111 multibeam echosounders. The raw data were converted into a binary input format for the research reference implementation of CHRT, and were then processed on a iMac (2.93GHz Intel Core i7 processor, 8GB main memory, 7200 rpm 1TB Western Digital hard disc, running MacOS 10.7.5). The first phase of the algorithm was run in serial mode for simplicity, and the second phase in parallel. The number of threads for processing was increased from 2 to 20, and the time taken to process the data was recorded in each case. (Note that the processor has only four cores with two hyper-threading units per core; software checks that avoid adding more threads than cores were disabled in this case.)

The mean processing rate per thread for the second phase of the algorithm is shown in Figure 6, while the total processing rate for the computation (i.e., the aggregate over all threads) is shown in Figure 7. It is clear that the mean processing rate per thread decreases continuously as more threads are added, primarily because of increasing competition between the threads for the resources of the processor (disc access, memory bandwidth, cache space and eventually core residency). The overall processing rate does still increase in general, however, although the pay-back per thread after approximately six threads are committed is relatively slight.

These results suggest that while it is possible to improve the performance on a single processor by a relatively modest amount (the best-case speedup here is approximately 3.6), the contention for resources severely limits the performance improvement. Distributing the computation over multiple processor nodes, where this contention is more limited, is more likely to improve the performance. In a distributed system, the contention is typically for network bandwidth to access data from a remote location. With the implementation as described here, however, the mesh-topology of the data access network, where each processor node can obtain data from its peers, means that this is unlikely to be a significant concern. Note that these results do still confirm that the performance of the algorithm can be improved by multi-threading. In practice, it is likely that a hybrid model will be most effective. For example, the algorithm might distribute the sub-tasks to to multiple processor nodes but run a small number of threads on each sub-task within the node, leaving some space to allow for the pre-fetch task, and the OS.

IV. Partitioning Efficiency

A. Time Complexity and Efficiency of Partitioning

The performance of the algorithm in the second phase is only part of the computational cost. Determining how to split up the overall task into sub-grid tasks can be computationally demanding, as outlined in section II-A and Figure 3. In order to test this, the time taken to determine the optimal split in the experiment of section III was recorded as a function of the number of segments in the partition. The results, Figure 8, show that the computational cost increases rapidly beyond a certain threshold, as the number of possible splits that have to be considered grows enormously. (Note that the cost does not appear to increase for small numbers of segments because the time is dominated by the time to extract the sounding counts, and then deal with the partition results after they are computed.)

These results further emphasize the diminishing returns that can be obtained by further and further segmentation of the overall task: when the sub-grid task count increases beyond 14 in this example, the time to compute where to partition the sub-grid tasks is higher than the time to do the computation. This

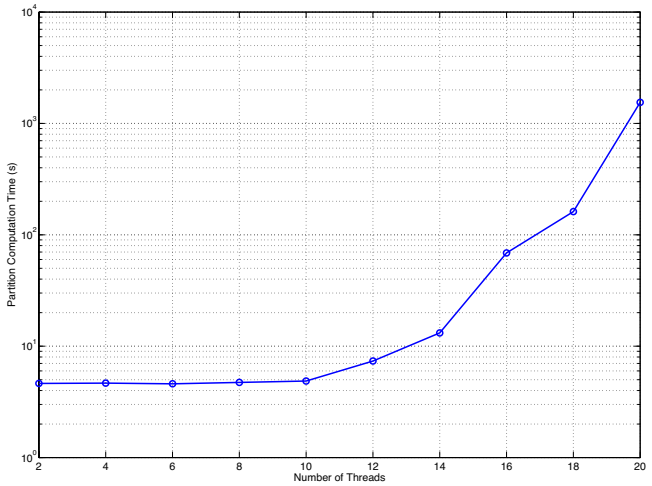

Fig. 8. Computational cost of determining the optimal partition of the overall task into sub-grid tasks. Note the logarithmic scale on the time (vertical) axis: the computational cost increases dramatically for larger numbers of segments in the partition.

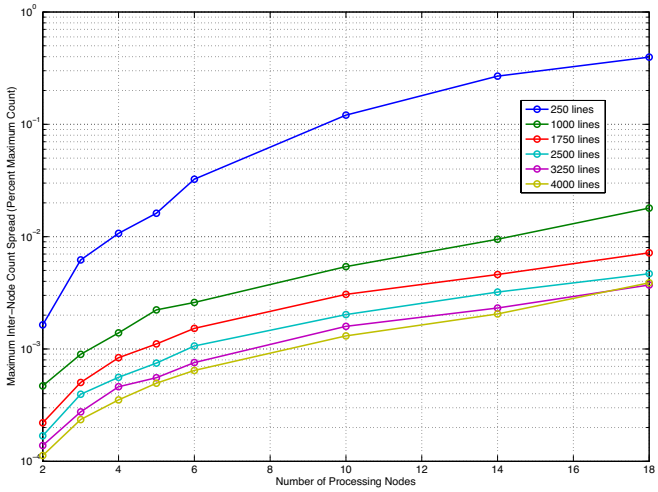

Fig. 9. Estimate of mean difference between maximum and minimum workload per node, an estimate of how evenly data has been scheduled for the first phase of the algorithm. These results are for the MULTIFIT algorithm, and are estimated over a Monte Carlo run of 100 samples for each combination of processor node count and source line count. Note the logarithmic scale on the vertical axis.

suggests that there are a limited number of processors that are useful for a given size of task, after which working out what to do with them takes longer than the actual computation. This implies, of course, that the current algorithm has to be used to optimally partition the overall task, and it is possible that a sub-optimal partition that was quicker to compute might be more effective. Note that there is no evidence of a significant growth of 'edge effect' increases in processing (i.e., where dealing with extra soundings around the edges of smaller sub-grids causes extra work) as a function of sub-grid task count, suggesting that this is not a factor in the decision on how many sub-grid tasks to allocate.

B. Data Distribution

As illustrated in Figure 2, determining an even distribution of work to the processing nodes is essential if they are to complete their computation at approximately the same time, and therefore avoid wasted compute cycles while one or more of the processors idles while the others complete. In order to test the ability of the algorithm to achieve this, an experiment was conducted using the MULTIFIT algorithm (as defined in [8]), measuring the performance of the assignment of survey lines to nodes through the difference between the node with the largest assigned workload and that with the smallest (and through the time taken to compute the schedule). Workload was approximated by the number of soundings assigned to the node, on the assumption that the processing in the first phase of the algorithm is approximately proportional to the sounding count. The experiment repeated the measurement for different numbers of processing nodes, and different numbers of lines, running the experiment 100 times for each pair. The same dataset as described above was used, uniformly sampling with replacement to synthesize datasets of variable numbers of lines.

The maximum/minimum workload difference is shown in Figure 9 as a percentage of the maximum load. Clearly, the more lines there are to be scheduled, the better the algorithm is at evening out the workload balance, primarily because the minimum scheduling quantum is a single line: with small numbers of lines, this can lead to large differences simply due to which line is placed last. (MULTIFIT assigns larger lines first in order to avoid this as much as possible.) There is also an obvious trend with the number of processor nodes, which can be explained by the same observation: given more processor nodes for the same source line count, a single line quantum can have a much larger effect. Note, however, that the unevenness is very small: the worst case is approximately 0.4% of the maximum.

The time to compute the schedules is shown in Figure 10. Here the trend is less obvious, except that the computation time is a stronger function of the number of lines being assigned than it is of the number of processor nodes over which they are assigned. The computational requirements are also low given the likely processing times for the size of problem considered here, so that the time taken to compute how to distribute the source lines is unlikely to be a significant percentage of the total computation time.

These results suggest that it is relatively inexpensive to compute a potentially sub-optimal, but very good, schedule for the distribution of the lines for the first phase of the algorithm. The relative insensitivity to the number of processor nodes also suggests that this computation time is not a major factor in selecting the number of processor nodes to use for a particular overall task.

V. RUN-TIME PERFORMANCE

In order to investigate the potential performance of the second phase of the algorithm given the data distribution used for the first phase, a simulation model of the computation was constructed. From the experiment in section III, the list of survey source lines that were required in each segment of the

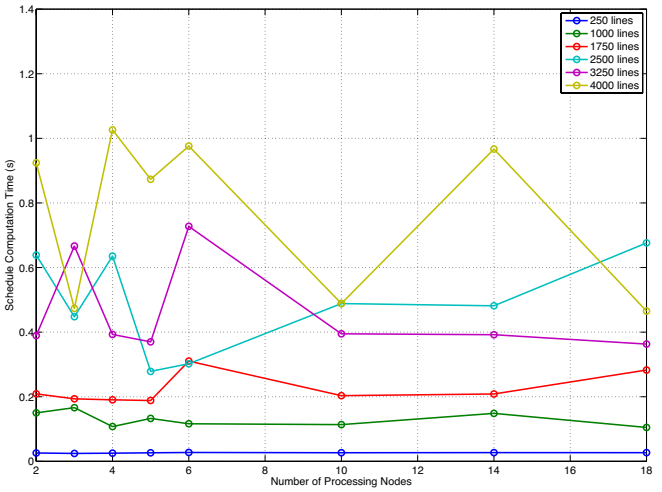

Fig. 10. Time to compute the schedule as a function of processor node count and source line count. The time required is generally small compared to the computational time, and a much stronger function of the number of source lines than the number of processor nodes.

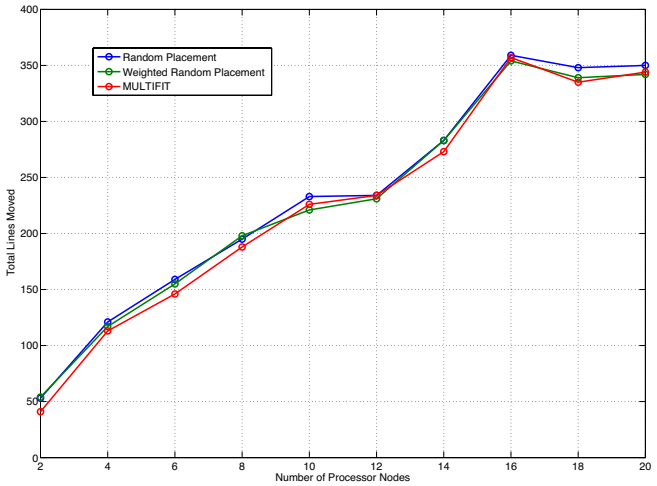

Fig. 11. Total number of source lines moved on all nodes during the second phase of the algorithm. Better distributions of source lines would be expected to reduce the number of line movements, but there is little to choose between the three algorithms used in this experiment.

partition was extracted, along with the number of soundings from each line that were used in the computation within the segment (this can be different for the same source line used in different segments). The survey source lines were then assigned to variable numbers of processor nodes in the range 2–20 using three different algorithms, and the sub-grid tasks were then assigned to the processor nodes to best match the available lines using the greedy algorithm outlined in section II-C. Finally, the runtime for the second phase of the algorithm on each processor node was estimated assuming that the data processing rate and data transfer rates were constant (i.e., ignoring network congestion) as computed from measurements of a typical processing system and SAN data transfer rate, and that the source lines required were pre-fetched as quickly as possible once the computation started. The estimate assumes that source lines that are already assigned to the processor node are available immediately the processing starts, but that those that must be transferred are transferred in order, and become available over time. The processing of source lines at each processor node must also complete in the assigned order, and therefore if the required source line is not available, the algorithm will stall until it is transferred. The number of lines that were required to be fetched on all processor nodes, the total transfer time required, stall time and wall clock computation time were all recorded.

Three algorithms were used. First, as a control, a completely random assignment of survey source lines to nodes was used. Second, a weighted random assignment was used, so that as each source line was considered (in descending order of estimated work), it was assigned according to a probability density designed to even out the accumulated work associated with all of the processor nodes (i.e., processor nodes with less work are more likely to receive a new source line). This can be considered as a stochastic variant of the LPT algorithm [7]. Finally, the MULTIFIT algorithm was implemented, which assigns survey lines (in descending order of estimated work) to the lowest numbered processor node that has capacity to accept it, minimizing the maximum work assigned for all processor nodes.

One way to estimate the efficiency of the distribution is to track the number of source lines which must be transferred in order to support the second phase of the algorithm. The results, Figure 11, show that there is little to choose between the three algorithms: all of them require more source lines to be redistributed as the number of processor nodes grows, simply because each segment of the work partitioned is smaller (spatially) and therefore they share more source lines. (Note that the average number of source lines moved per node does decrease with increasing numbers of processor nodes.)

Analysis of the worst-case time spent transferring data between nodes shows that the MULTIFIT algorithm has a small advantage over the other algorithms, Figure 12, due primarily to its more efficient placement of source lines, which the sub-grid task placement algorithm can exploit. The difference, however, is relatively small, and somewhat masked in the overall performance by the pre-fetch algorithm.

This masking effect is reflected in the worst-case stall times observed, Figure 13. Here, it is clear that the stall times generally increase with the number of processor nodes (although the correlation is more likely with the number of source lines to be transferred), with the exception of the anomalies about 8–10 processor nodes. This is thought to be due to a particularly advantageous partition of the overall task into sub-grid tasks.

Finally, the wall-clock time for the second phase of the processing is shown in Figure 14. As might be expected given the other results, there is very little difference given the placement algorithms, all of which have an anomalous reversal of performance at 12–14 processor nodes. This is believed to be due to a particularly bad configuration for the sub-grid tasks, but further investigation is required. It is more illustrative to consider the same data as a speedup, Figure 15, which shows

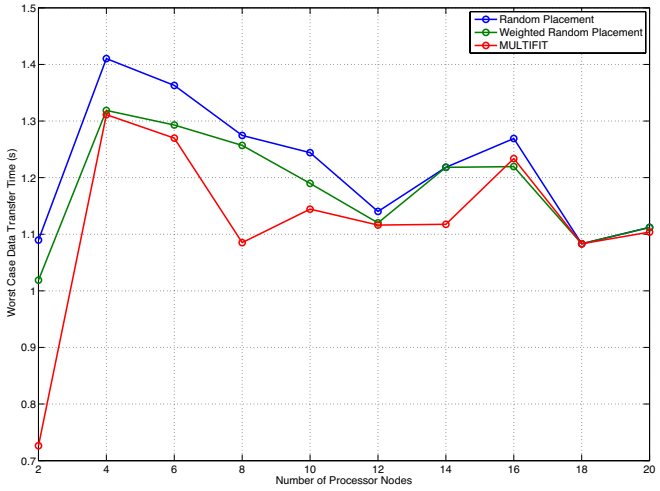

Fig. 12. Worst-case data transfer between nodes during the second phase of the algorithm. The MULTIFIT algorithm has a slight advantage in the worst-case, particularly when smaller numbers of processor nodes are concerned.

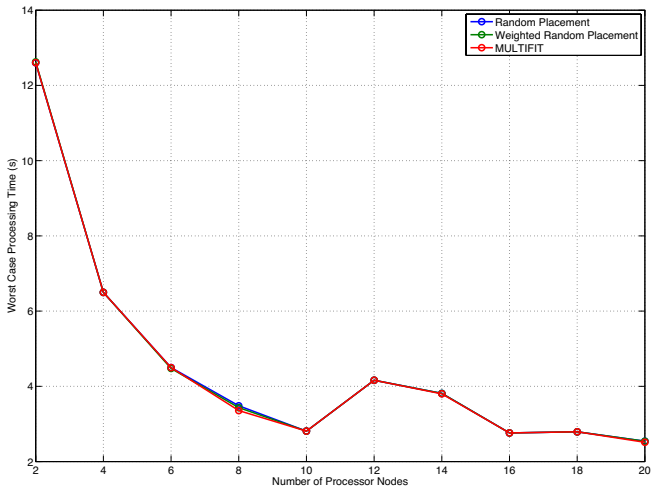

Fig. 14. Worst-case wall-clock time (i.e., time for the slowest node to complete) during the second phase of the algorithm. The differences between the algorithms are very slight; the anomaly about 12–14 processor nodes is believed to be due to a bad configuration of the sub-grid tasks.

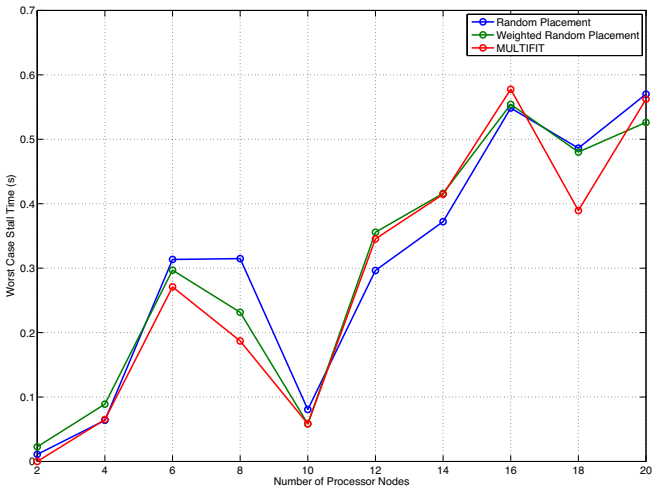

Fig. 13. Worst-case processor node stall time (i.e., total time the sub-grid task spends idle waiting for data) during the second phase of the algorithm. The differences between the algorithms are likely insignificant, although the MULTIFIT algorithm has a small advantage for smaller processor node counts.

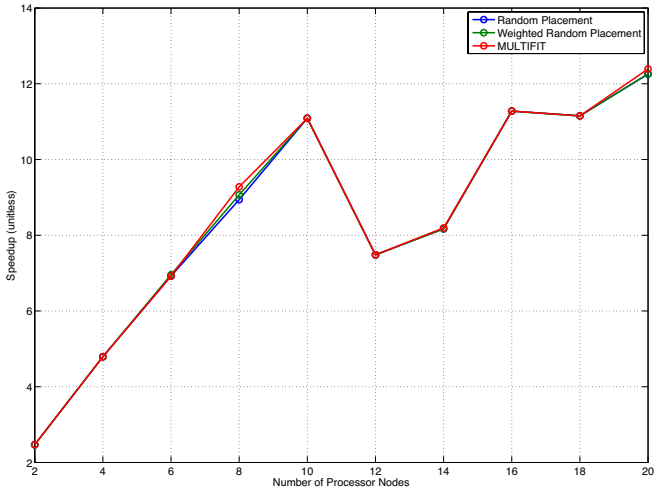

Fig. 15. Speedup for the second phase of the algorithm as a function of data placement algorithm and number of processor nodes. The differences between the algorithms are small, giving a maximum speedup of just over 12 times with 20 processor nodes committed.

almost linear increase in performance up to 10 processor nodes, and then a slow tail off (ignoring the anomalous results for 12–14 processor nodes), with maximum speedup on the order of 12 times at 20 processor nodes committed to the computation. This corresponds to an aggregate speed of 3.68×10^6 soundings/s, although this is only for a single thread of execution within each processor node. If the results of the previous section could be replicated, we might expect a performance more on the order of 13.3×10^6 soundings/s. (Note that the results of Figure 15 show an apparent super-linear speedup for low numbers of processor nodes; this is not expected, and may be due to a timing anomaly in the serial version of the algorithm used as a baseline.)

These results appear to suggest that the method used to place the source lines on the processing nodes prior to the first phase of the algorithm does not strongly control the performance of the second phase of the algorithm. That is, the processor nodes appear to transfer approximately the same amount of data, have roughly approximate worst-case data transfer, stall and wall-clock times, leading to similar performance metrics. Note, however, that none of the algorithms currently tested take into account spatial coherency between the source lines when deciding where to place them, and therefore do not necessarily optimize for placement of the sub-grid tasks. This is taken up further in section VI, below.

VI. DISCUSSION

The experiments conducted here were intended to investigate the likely performance of a parallel and distributed version of the CHRT algorithm as a means to adjusting the design before final implementation. Among other things, they show

fairly clearly that there are significant performance gains to be wrung from operating in parallel, although it appears that there is more to be gained from the distributed version than the parallel version. Neither version, however, appears to achieve the nominal linear speedup that would be preferred.

Some of the results are counter-intuitive. As more processor nodes are added to the computation, and the sub-grid tasks thereby cover smaller areas, it seems logical that the edge processing required should increase (since there are more edges). The evidence is, however, that the difference is so small as to be negligible. Similarly, the details of the distribution of data during the first phase of the algorithm might be expected to have a significant effect on the performance of the second phase, which relies on having all of the right data in the right place at the right time. The evidence is, however, that even the random reference placement method performs more or less equivalently to the carefully balanced (and more computationally expensive) MULTIFIT algorithm. Partly, this is because none of the algorithms take advantage of the known spatial correlations between the source lines, and therefore that the distribution, while balanced in terms of the workload per processor node for the first phase, is essentially spatially random. For this reason, it is tempting to say that all of the algorithms are equally bad, rather than equally good.

Solving the scheduling problem where there are constraints between the objects being scheduled (in this case the source lines) is more difficult. For example, the ideal situation would be for all of the source lines for a sub-grid task to be available on a single processor node, which might be accomplished by placing lines with preference according to their distance apart in space. Until the first source line is associated with a processing node, however, the 'cost' involved in placing a subsequent source line (that might be required for the second phase computation that also uses the first source line) on another processor node cannot be computed. It may be possible to extend the results of Woodside and Monforton [8] to fit this problem, however. Here, the MULTIFIT algorithm is extended to include communications costs associated with pairs of tasks; that is, once you place a task on a node, if it communicates with another task not on the same node, there is an additional cost (in the referenced model, this is to simulate the cost of communication between the nodes using a shared bus). The situation with the distributed CHRT algorithm is similar, although the direct costs come from concerns about even distribution of data during the first phase, and the communications costs come from concerns about data movement in the second phase. A practical difficulty is that the specific clustering of source lines required to support any given sub-grid task is not known until the first phase is complete, so that placement will be at best approximate. One plausible model for this is to consider the communications costs between source lines to be probabilistic, in the sense that any two source lines will be required on the same node only if they are close together in space, and the closer they are, the more likely it is. Assessing a (subjective) probability that one source line will require another to support the computation of a sub-grid task may allow the algorithm to operate on the basis of expected rather than deterministic costs, although this is a subject of much further research.

The observation that the computational cost of determining the optimal sub-grid split of the overall task increases dramatically with the number of processor nodes has a number of implications for the design of the algorithm. First, it suggests that arbitrarily large clusters of processor nodes may be counter productive: it could take much longer to work out how to use them than the computation itself. Of course, this is likely to depend strongly on the size of the problem being computed, which has not been considered here. This in turn suggests that it might be useful not to have a fixed number of processor nodes into which any given task should be molded, but to allow the algorithm to determine how many processor nodes would make sense for a given task, and adjust accordingly. This would require more flexibility in a cluster monitor to implement, but may have efficiency gains sufficient to be worth the cost.

Second, limitations on the number of processor nodes that can be effectively assigned in a reasonable amount of time suggests that a hybrid scheme might be more appropriate. Consider, for example, that instead of assigning a sub-grid task to a single processor node, it is assigned to a group of them. Each group of processor nodes might act as a mini-cluster, splitting up the sub-grid task into smaller jobs, and then executing them. If the mini-cluster of processor nodes shared a common disc sub-system (e.g., multiple processors on a single blade server with common disc), then this should not generate more significant traffic, but would limit the size of the partitioning problem at both levels, allowing them to be computed more readily. Multi-threading within the processor nodes could then be used to improve the local performance by sub-dividing the jobs again. The sub-division into jobs, and then into threads, is likely to be sub-optimal; the question is whether it can improve the overall throughput nevertheless.

In general, the work here has focussed on the performance of a single version of the algorithm, but in practice it is likely that there should be multiple versions that are adapted according to the implementation environment. This has the advantage that the solution can scale to the user requirements, but would require algorithms that would be able to adapt to their computational environment. The interactions between the various components of the algorithm, the implementation of the computational structure and the data being processed can be subtle, but very impactful. Making the algorithms auto-tune for their environment given these interactions is likely to prove challenging.

VII. CONCLUSIONS

It is clear from the results shown here that there is great potential for improved performance of the CHRT algorithm by suitable use of parallel and distributed processing techniques. The subtleties of obtaining the best performance from such systems, however, are exacting. The results here suggest that it may not be productive to have very large clusters of processor nodes associated with any one task, which implies that the overall performance improvement might be limited (although still substantial), and that it is likely that an efficient

implementation will need to blend inter-node and intra-node parallelism to provide the highest overall performance. The scheme considered here would be suitable for implementation within a single processor (with limited speedup), within a local cluster (or on multiple loosely coupled machines on a local network, to lesser effect), or potentially within a cloud service, given a sufficiently rapid method for data ingestion into the instance. The algorithm's performance is predicated strongly on the interaction with the particular hardware in use, however, and therefore performance tuning—and possibly auto-tuning—will be a significant concern in the future.

ACKNOWLEDGEMENTS

This work is supported by NOAA grant NA10NOS4000073.

REFERENCES

[1] B. R. Calder and G. Rice, "Design and implementation of an extensible variable resolution bathymetric estimator," in *Proc. US Hydro. Conf.* Hydro. Soc. Am., April 2011.

[2] B. R. Calder and L. A. Mayer, "Automatic processing of high-rate, high-density multibeam echosounder data," *Geochem., Geophys. and Geosystems (G3) DID 10.1029/2002GC000486*, vol. 4, no. 6, 2003.

[3] S. Martello and P. Toth, *Knapsack Problems*, ser. Wiley-Interscience Series in Discrete Mathematics. Baffins Lane, Chichester, England: John Wiley and Sons Inc, 1990.

[4] T. White, *Hadoop: The Definitive Guide*, 2nd ed. Sebastopol, CA: O'Reilly Media Inc., 2011.

[5] E. G. Coffman, M. R. Garey, and D. S. Johnson, "An application of bin-packing to multiprocessor scheduling," *SIAM J. Comput.*, vol. 7, pp. 1–17, 1978.

[6] D. S. Hochbaum and D. B. Shmoys, "Using dual approximation algorithms for scheduling problems: Theoretical and practical results," *J. Assoc. Comp. Mach.*, vol. 34, no. 1, pp. 144–162, January 1987.

[7] R. L. Graham, "Bounds on multiprocessing timing anomalies," *SIAM J. App. Math.*, vol. 17, no. 2, pp. 416–429, 1969.

[8] C. M. Woodside and G. G. Monforton, "Fast allocation of processes in distributed and parallel systems," *IEEE Trans. Parallel and Dist. Sys.*, vol. 4, no. 2, pp. 164–174, February 1993.