Research White Paper (FINAL)

For

ProLogic, Inc.

Task Order 1

of

Commercial GIS Extension for Visualization of Large Unstructured Datasets

Feb 4, 2005

Prepared for:

Chetan Desai

Principal Investigator

ProLogic, Inc.

Prepared by:

Institute for Scientific Research, Inc.

320 Adams Street

P.O. Box 2720

Fairmont, WV 26555-2720

(304) 368-9300

www.isr.us

TABLE OF CONTENTS

1. Introduction

While there has been a steady increase in the resolution and size of LIDAR datasets, there has been an attendant increase in demand for full-resolution interactive processing and visualization of this data. The requirements of this processing create a computational bottleneck, suggesting the application of High Performance Computing (HPC) resources to improving performance.

This white paper focuses on the issues surrounding the efficient use of HPC resources to alleviate the computational bottleneck, particularly on what kind of hardware and with which parallel libraries to implement the processing. The discussion is divided into the following sections.

- Section 2: HPC Overview—presents a general discussion of HPC, an overview of the options available to the procurer of such a system, and the criteria associated with choosing an appropriate architecture and an appropriate algorithm for a given problem domain.
- Section 3: Discussion of the Application—investigates the specific domain of interest—high-performance LIDAR data processing—in light of the design criteria set forth in the preceding section.
- Section 4: Summary and Recommendations—Sections 2 and 3 yield a recommendation for the appropriate parallel computing system, data processing algorithms, and approaches to implemention in the LIDAR domain. The specific recommendations—namely a .NET middleare implementation using the United Devices MP Grid API—are enumerated here.
- Section 5: Common Interface Design and Development—This section discusses the details of the design and deployment of the actual distributed infrastructure to support remote access to the Windows network of workstations and the Linux cluster.
- Section 6: Benchmarks and Performance Assessment—the HPC application software is developed and deployed on the recommended architecture; a series of tests is designed to demonstrate the performance of the software on the selected HPC system. Raw performance, Speedup, and Scalability are determined and reported.
- Section 7: Conclusions and Lessons Learned.
- Appendices: Numerical Benchmark results and the specific problem of varying work unit number for a fixed problem size.

2. HPC Overview

This section contains a general discussion of HPC, an overview of the options available to the procurer of such a system, and the criteria associated with choosing an appropriate architecture and an appropriate algorithm for a given problem domain.

2.1 *HPC Architecture*

There is a diverse set of definitions and descriptions for HPC architectures in the literature, but very little real consensus. However, whether a computer is sequential or parallel it operates by executing instructions on data. Thus, classifying parallel computer architectures for developing computer applications can be distilled to describing two characteristics:

- The instruction stream, or algorithm, executed by the computer
- The data stream, or input, which is manipulated by the instruction stream

According to Flynn's taxonomy [Quinn], the nature of these streams is the crux of choosing an appropriate architecture for a specific HPC application. This widely-used terminology is summarized as follows:

- SISD—single instruction, single data stream. This classification identifies conventional serial von Neumann architecture, as found in a desktop PC. This architecture is bound, by definition and implementation, to processing a single instruction on a single datum at a time.
- MISD—multiple instruction, single data stream. This description is often regarded as theoretical and unimplemented. The implication is multiple instructions being executed on a single piece of data, but this description is sometimes applied to conventional Symmetric Multiprocessor (SMP) machines, which comprise multiple processors sharing a single memory bus.
- SIMD—single instruction, multiple data streams, involves multiple processors simultaneously executing the same instruction on heterogeneous data. HPC machines that incorporate this architecture include: the "Connection Machine" CM-2, which contains 64,000 very simple processors which all simultaneously execute identical instructions on partitioned data, or the MasPar MP-2.
- MIMD—multiple instruction, multiple data, describes multiple processors autonomously executing diverse instructions on diverse data. This describes distributed-memory parallel machines such as the IBM SP3 and CRAY T3E.

Although Flynn's classification is the most often used, there are three weaknesses. First, the MISD classification is somewhat ambiguous and academic. Second, there is not a sufficient nomenclature for characterizing pipelined vector processors, such as the CRAY J90 or the more modern NEC vector computers used in the Earth Simulator—the world's fastest supercomputer. Finally, the distinction between MIMD and SIMD is somewhat pedantic in light of the modern predominance of distributed-memory machines which use arrays of networked autonomous processors, distributed memory, and message-passing software libraries. An enhancement is often made to the Flynn's taxonomy by including the SPMD—single program, multiple data stream—classification.

An SPMD machine is physically a MIMD machine, because it consists of a set of processors each of which contains its own distinct instruction stream and program. However, an SPMD system executes the same program on every processor rather than requiring some global management of program streams and system state. In addition, the SPMD classification is more robust than the SIMD label, in that program state of any individual processor is not advanced by a central controller in lock-step with the rest of the processors. At any given time, each

processing element in the SPMD system can be in the process of executing a different instruction in a different routine than its cohorts. In a sense, the SPMD label represents an approach to parallel architectures and parallel development rather than a single architecture, and is highly represented by many of the modern systems discussed in this paper. In contrast, the MPMD model has also been implemented, whereby each processor in a distributed computation can execute a separate and distinct program.[1]

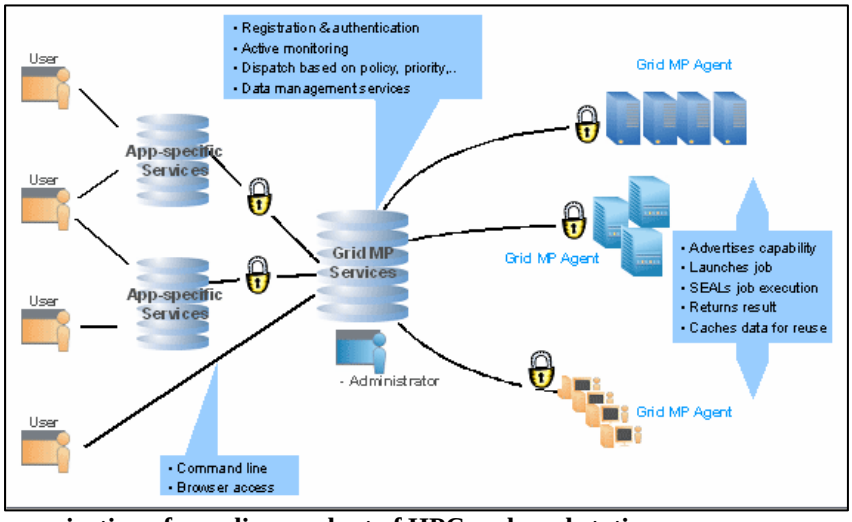
2.2 *GRID Architecture*

Whereas a Beowulf-class cluster computer is defined as a set of closely-coupled COTS PC's residing on a single network, the GRID concept refers to groups of clusters, supercomputers, or individual workstations, geographically dispersed, loosely coupled, and interconnected across heterogeneous networks.

The ISR team has applied United Devices' Grid MP framework for the implementation of the LIDAR computational algorithms on a Network of Workstations (Figure 1 [below]). The Grid MP system uses XMP and RPC technology (similar to .Net's Web Services) to aggregate Linux, Windows, AIX, Solaris, HP-UX, IRIX, and OS X machines.

As part of this investigation, the second software demo demonstrated the implementation of a .NET infrastructure which accommodates the essential functionality required of the LIDAR processing back-end. The design (described in Section 5 and in the Appendices) accommodates the integration of the United Devices Grid MP software, while at the same time maintaining a consistent software interface/API for both NOW access and cluster access.

Figure 1. The United Devices Grid MP framework facilitates creation of a network of workstations (NOW),



or the intercommunication of any dispersed set of HPC and workstation resources.

[1] The most widespread implementation of the SPMD model is implemented by the Message Passing Interface (MPI) specification. The Parallel Virtual Machine (PVM) standard, implemented by Oak Ridge National Laboratory, exemplifies the MPMD model. The PVM model is useful where a computation has been divided into separate heterogeneous subprograms that can execute and intercommunicate simultaneously, while MPI is most prevalent in modern HPC clusters.

Implementation of the Grid MP framework follows 4 steps:

Step One: Grid MP Services and Agents are installed to lay the foundation for the virtualized infrastructure. Administrators define and install policies that govern the use of this infrastructure across multiple applications and users.

Step Two: Application Services are created and installed for each application -- these Services preserve the current user interface for end users.

Step Three: Users interact with the application as they have always done. Transparent to the user, these jobs are now executed on the virtual infrastructure created by Grid MP.

Grid MP dispatches jobs based on resource constraints set by the user as well as application and business priorities and policies governing the use of all resources.

Application code and data are transferred to the resource. The appropriate execution environment is created at the selected resource level.

Grid MP monitors job execution and guarantees completion.

Step Four: Once execution completes, results are collected by Grid MP Services and passed back to the end user.

The net result of this approach is that the heterogeneous set of computational resources utilized behind the scenes appears as a single cohesive computational resource from the user perspective, yielding a high level of transparency from the perspective of the end user on his workstation.

2.3 *Parallel Algorithm Development*

When approaching the issue of how to implement computational geometry algorithms for LIDAR processing on an HPC system, there are a number of parallel programming paradigms from which to choose. This choice is typically governed by the available HPC hardware, as described above. However, even for a given hardware architecture, there is often a number of conflicting options when deciding how best to program the system to accomplish the task at hand. Thus, it is a challenge to determine the most appropriate approach to parallelizing a given algorithm. Insight into this task can be achieved by characterizing several aspects of a given algorithm: task granularity; data- versus instruction parallelism, algorithm design, and programming paradigms. This section describes the software development issues associated with designing parallel programs to function on HPC systems. [Buyya1] [Quinn]

2.3.1 Parallelization and Performance

Designing a parallel algorithm involves partitioning a problem into independently computable pieces that can be evaluated simultaneously. The portions of the program that operate in parallel tend to decrease the apparent running time relative to an equivalent nonparallel (serial) program (apparent in the sense that the total computational time is roughly equivalent, but diminished "wall clock" time is experienced), yielding "Speedup." If $T(N)$ is the time required for a program to run on N processors, then speedup, $S(N)$, is computed as a simple ratio of these two running times, as follows:

$$S(N)=\frac{T(1)}{T(N)}$$

In the idealized case, this equation would be expected to yield a linear speedup with respect to *N*; *N* processors execute the same program in *1/N* the time taken for the serial program. In reality, the portions of a program that cannot be made to operate in parallel tend to cause a serial bottleneck in the computation. All programs contain serial portions that cannot be parallelized (*e.g.*, reading input data, broadcasting the data from a master process to other processors in the system, or accumulating results on termination). This idea is formulated as Amdahl's Law, which recasts the speedup calculation in terms of a serial time, T_s, and a parallel time, T_p. The serial time is independent of *N*,

$$S(N)=\frac{T(1)}{T(N)}=\frac{T_s+T_P}{T_s+T_P/N}$$

Examining the algorithm of interest in terms of these performance measures often provides a basis for parallelizing that algorithm. Determining the location of the serial and parallel portions of the algorithm will directly affect the speedup of the final program, by directly affecting T_s and T_p.

For example, a Computational Fluid Dynamics (CFD) algorithm might require many intermediate accumulations of data at a single processor for purposes of reporting intermediate results. This is a serialization which increases the value of T_s because that individual processor must sequentially accept and process the incoming data. As T_s increases, the theoretically achievable speedup decreases asymptotically to 1.0 (no speedup at all for any number of processors). Meanwhile, a Monte Carlo simulation of protein-folding would not report results until the terminus of computation, and thus there is no such serialization introduced (although invariably there is serialization elsewhere). Relative to the above CFD algorithm the Monte Carlo algorithm would have a small value of T_s / T_p and thus a larger possible value for speedup. Assessing the algorithm in this way, independently of the HPC system on which it will be deployed, yields insight into the initial approach to parallelizing a code. These observations will be revisited for the specific deployed system in the Performance Section of this paper.

2.3.2 Efficiency

Given a set of Speedup measurements, the efficiency compares the performance against the ideal case—typically "linear speedup." Linear speedup represents the conventional "best possible" speedup attainable: if an algorithm runs in *x* seconds on one processor, the best one would typically anticipate is *x*/2 seconds on two processors (in wall-clock terms). "Superlinear" speedup yields efficiencies beyond ideal, for various reasons, but a typical parallel implementation yields efficiencies which decrease as problem size increases. The work described herein yielded results across the whole range of potential efficiencies. These are displayed and discussed in Section 6.

2.3.3 Code Granularity

When a basic determination has been made of which portions of an algorithm are parallel or serial, the granularity of the algorithm should be assessed. Granularity is a measure of the time a parallel program spends communicating data between processes, with respect to the time spent computing results. The granularity of an algorithm is often the factor which determines the

optimal hardware which should be used for a given problem domain, in terms of processor speed and communications hardware capability. There are several levels of granularity in parallel applications:

- Very Coarse-grained ("Embarrassingly Parallel") applications– the case in which computation time greatly exceeds communication time. Examples include the popular SETI@home project on the World Wide Web and MONTE Carlo Simulation. Both of these simulations distribute completely independent computations to processors, and accumulate results at the terminus of computation with no additional communication between processors. These applications operate equally well on an HPC cluster or on a network of workstations.
- Coarse-grained applications– in this case, computation time typically exceeds communication time, but the communication time begins to contribute significantly to total execution time. Examples here include image processing, where each pixel can be processed independently, and many numerical optimization algorithms, which include not only very coarse-grained distributions of computations but also relatively finely-grained intercommunication between processing elements in order to maintain knowledge of global system state.
- Medium-grained applications–in these applications computation time and communication time would be roughly comparable. The aforementioned CFD example requires a high amount of communications between processors that are adjacent in the CFD mesh, because of data dependency between processing elements at every iteration during the simulation; in other words, the results in one processor are dependent on the computational results in the adjacent processors. In this case, the architectural requirements include not only powerful computational capabilities in each processing element, but also robust high-speed communications hardware, as found in an SMP machine or in the distributed-memory CRAY, which contains a high-performance multi-stage crossbar network. As the performance of commodity networking hardware has improved, COTS computational clusters have made inroads into computations at this level of granularity.
- Fine-grained applications—"communication time much greater than computation time " generally thought of as parallelism that is intrinsic to a program, and thus requires little or no programmer intervention to accomplish. This is often referred to as instruction-level parallelism. For example:
    - Loop unrolling is requested by the programmer as an optimization, usually controlled by the compiler, and parallalelized as a set of threads that can run concurrently. This implicit parallelization is accomplished using directives to the compiler during compilation, and requires no alteration of the original program by the programmer.
    - Source code directives placed in the code by a programmer can direct a parallelizing compiler towards areas of code that can be parallelized. In an SMP machine this might be a loop that iterates 1000 times, and is partitioned into 250 operations across 4 processors; each of these processors accesses its data from the same shared-memory bus.

2.3.4 Data-parallelism versus instruction-parallelism

The granularity of an algorithm will often determine how that algorithm meshes with the HPC architecture. With respect to Flynn's taxonomy, that architecture is described by the way it is optimized to handle data streams and instruction streams. Algorithm granularity may be examined in this manner as well, by making a similar distinction. Parallel algorithms fall into a spectrum between the following two descriptions:

- Instruction-parallel algorithms—are parallelizable by executing multiple instructions at once at a very fine-grained level. These algorithms execute most efficiently on "MI-oriented" hardware: vector, pipelining, or shared-memory architectures.
- Data-parallel algorithms—are parallelizable by partitioning data prior to computation and distributing it to distinct processing elements for parallel execution. These algorithms are intrinsically "MD" in the Flynn system, and are best served by execution on distributed-memory machines.

2.3.5 Algorithm design

As mentioned above, there is a certain amount of flexibility in deciding how best to implement an algorithm on a given HPC resource. For example, consider a Beowulf-class[2] computational cluster. MPI (Message Passing Interface) and PVM (Parallel Virtual Machine) are two parallel libraries typically used to implement MIMD-parallel algorithms on such a distributed-memory parallel system; however, the Linda toolkit implements the SMP (symmetric multi-processor) shared-memory paradigm on the same distributed-memory system. Similarly, Cray's "shmem" library implements the shared-memory paradigm on the distributed-memory CRAY T3D/E. Further, modern parallel clusters can mix these paradigms; *e.g.*, many modern cluster computers are actually (distributed memory) clusters of (shared memory) SMP machines and thus can use MPI and OpenMP simultaneously within the same parallel program [Cappello]. These options allow considerable flexibility when implementing parallel algorithms. Once the architecture and attributes of the algorithm have been identified, the rest of the parallelization process proceeds by addressing the following:

- Partitioning—also known as domain decomposition—a candidate algorithm is examined to assess its parallelizability, and identify which portions of the program may be divided into parallel
- Communication—the partitioned algorithm is recast in terms of information flow and control among processing elements in the computation. This aspect is often the crux of designing efficient parallel programs, in that a computation-bound serial program can easily become a communications-bound parallel program.
- Agglomeration—this aspect represents a synthesis of partitioning and communication analysis. Here, the software design is examined in light of the available HPC resources and modified to improve performance. Typically, this takes the form of maximizing the size of messages passed from processor to processor. For example, for even the fastest

[2]"Beowulf" describes an approach to HPC system design pioneered by NASA's Goddard Space Flight Center, in which a supercomputer is built as a cluster of low-cost commodity off-the-shelf (COTS) personal computers, interconnected with a local area network.

networking fabric, such as the double crossbar found on the IBM SP3, it is generally desirable to agglomerate messages within a local processor before broadcasting those messages together as a monolithic whole. This increases system performance (decreases total time spent in communications with respect to computation). Generally, more coarsely-grained algorithms typically are associated with very rare, relatively large messages.

- Mapping—is the physical process of determining how the programmatic elements identified above map to the processing elements in an HPC system.

The text of [Buyya2] is replete with examples of production-level parallel software developed using the above four steps, including several grand challenge problems. This approach encapsulates the elements integral to the development of any parallel program and will be applied to the domain of interest in the following sections.

2.4 *Cost*

A recent trend has emerged in the scientific computing arena towards simultaneously maximizing application performance and minimizing total system cost. This has been accomplished by introduction of the Beowulf-class computational cluster, comprising commodity off-the-shelf (COTS) processing elements and networking fabric. Whereas ownership of supercomputing resources was previously the purview of large government labs and universities, the low cost of today's commodity clusters has democratized access to HPC resources. Now, massively parallel HPC systems can be deployed at smaller colleges and institutions, at a fraction of the cost of comparable "dedicated" systems.

2.5 *Summary*

This section briefly introduced the central issues associated with specifying and deploying a high performance computing system. These issues include system architecture, design of the parallel algorithm, network communications fabric, and system cost. In the next section, these criteria are applied to the specific domain of LIDAR data processing. The objective is to design a system capitalizing on parallel HPC resources to allow users to analyze LIDAR data at its fullest resolution. The following sections describe the specific algorithms and HPC infrastructure implemented to that end. Those sections present performance figures for the system which draw from this brief treatment of general parallel algorithm development.

3. Discussion of Application

The SBIR Phase I research [Desai] focused on determining the technical feasibility of applying HPC resources to the analysis of full-resolution LIDAR data sets. Successfully achieving this goal has established the groundwork for Phase II development of a detailed architecture to support extension of ArcGIS's 3D Analyst software. This research found the following:

- A desktop PC is sufficient for visualization of large unstructured datasets (LUD)
- An HPC system is desirable for supporting computationally intensive tasks such as rasterization and triangulation

In this section, this latter finding is elaborated and formalized with respect to the criteria described in Section 2—HPC Overview. Figure 2, from the text, depicts the architecture investigated in Phase I.

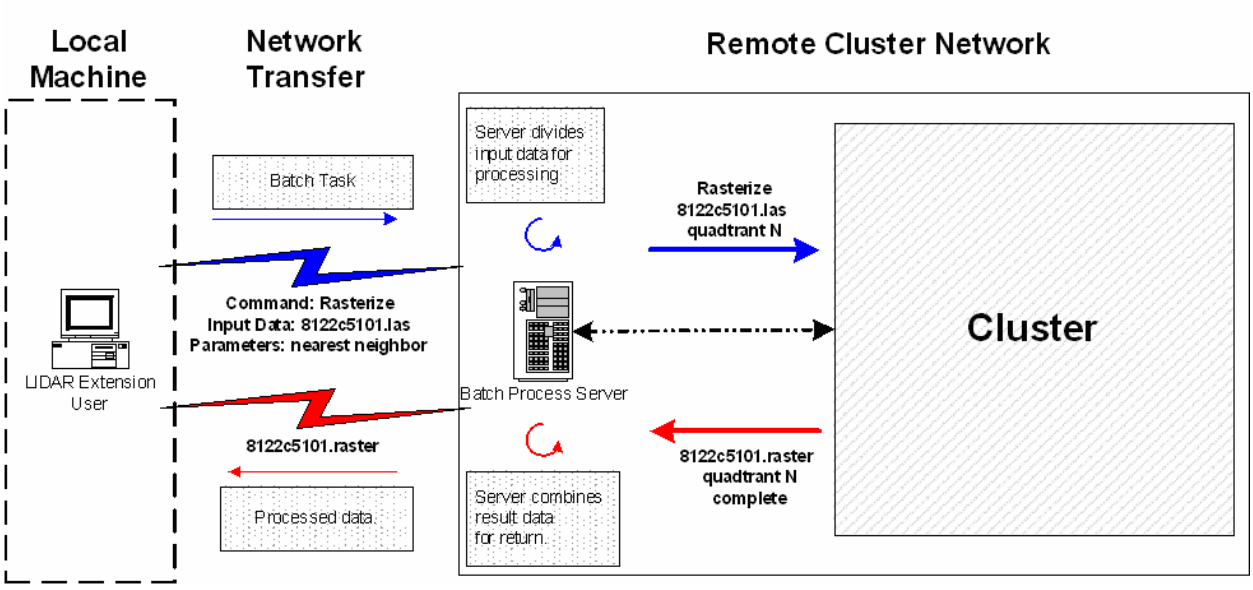


Figure 2. Utilization of an HPC to perform rasterization (Figure 6.16 from [Desai])

3.1 *LIDAR data processing*

A variety of models and algorithms were investigated in the Phase I research, and will be developed in the Phase II development of the LIDAR data processing extensions. This section summarizes these algorithms, laying the groundwork for specifying the type of HPC system most suited to their implementation.

3.1.1 Paralellizable models used in the LIDAR extension

Raster generation—the Raster model takes unstructured LIDAR data and converts it into a high-resolution structured grid. For this model, the 2D Shepard algorithm is used [Shepard].

- Partitioning—in Shepard's algorithm, each point in the structured grid comprises a distance-weighted average of the 15 "nearest neighbors" to that point from the raw LIDAR data. The computation may be partitioned by computing each point in parallel (using globally available raw data).

- Communication—interprocess communication and control are minimal. Initially each processor must read in the entire data set using file I/O and each processor autonomously determines which part of the structured grid for which it must compute points.
- Agglomeration—the granularity of the computational tasks, along with the above partitioning and communication approach, facilitates maximizing message size. The final results of each computation can be agglomerated and transmitted en masse at the end of the processing step. This ensures the largest possible message sizes, and streamlines the communications time.
- Mapping—because this is a data-parallel algorithm, and because the chosen approach ensures no data dependencies between individual computations of the distance-weighted mean values, mapping of tasks to processors involves dividing amongst the available processing elements the total number of structured raster points to be computed.

To summarize, the 2D Shepard algorithm for computing the rasterized data from raw data points takes the form of a balanced, data-parallel mapping to the available processors of the raster points to be computed. This yields a maximally efficient division of work and communication.

The computational complexity of this computation is $O(\frac{m \cdot n}{N})$, where m=the total number of datapoints in the original unstructured dataset, n=the total number of points in the structured raster grid, and N=the total number of processors involved in the computation.

As an example, referring to Figure 3 and Figure 4, the implementation of the 2D Shepard algorithm yields marked speedup on ISR's Black Diamond computer cluster (full performance metrics for all computational aspects of the system are given in Section 6).

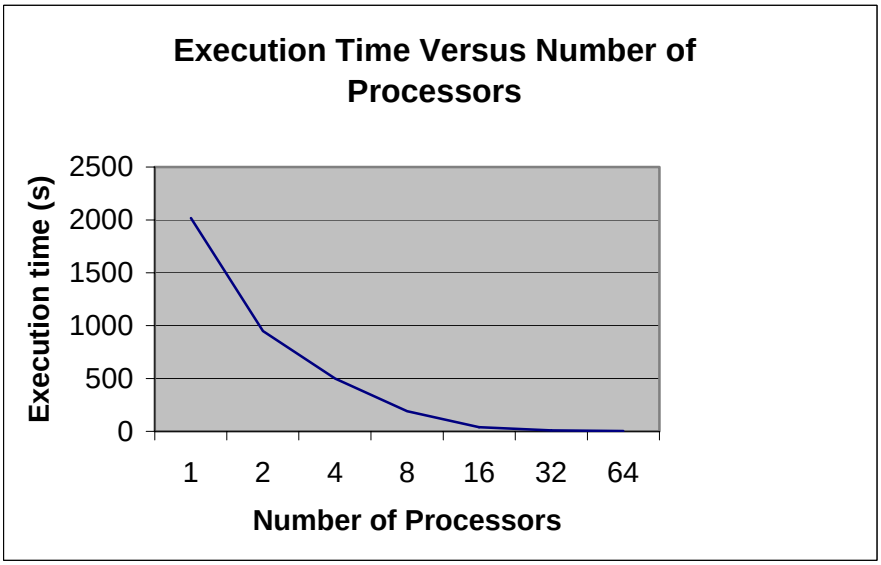


Figure 3. Observed execution time for rasterization processing of the Gauley River dataset.

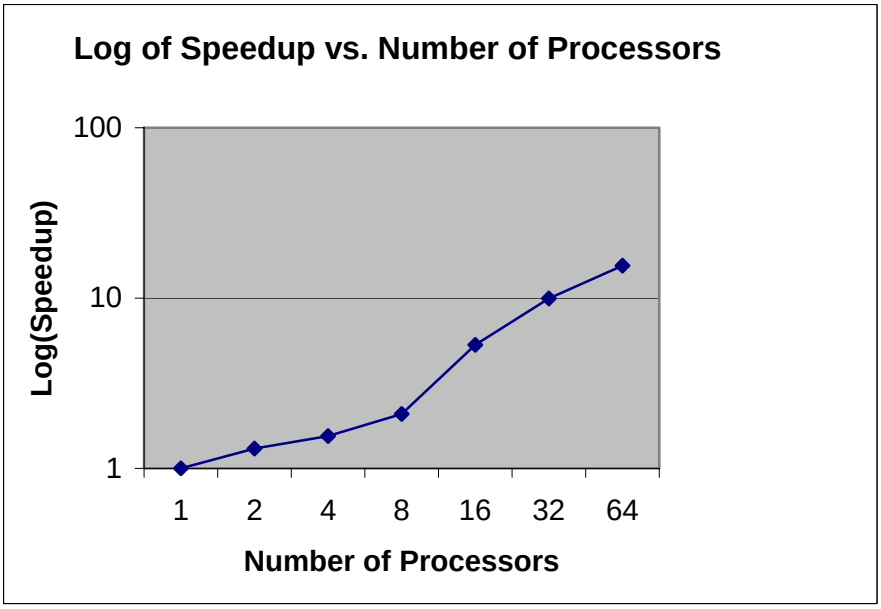


Figure 4. Observed Speedup (log-log) for rasterization processing of the Gauley River dataset demonstrates linear growth. The initial low slope at processors 1-8 is likely attributable to paging/swapping of the large amount of data in 2GB of RAM, whereas more processors contain less data per processor and thus execute fewer swaps.

TIN generation—the Triangulated Irregular Networks (TIN) model uses contiguous overlapping triangles to represent a 3-dimensional surface. For this model, the Delaunay triangulation algorithm was used to generate high-resolution TIN datasets [Delaunay].

The Delaunay Algorithm is widely used in computational geometry to compute the simplicial—or unstructured—mesh from a set of points, in this case the raw LIDAR input data. This algorithm is implemented as part of VTK, and VTK was used in the Phase I work. The VTK algorithm is a serial algorithm and thus the parallelizability of Delaunay triangulation was investigated in terms of the criteria established in Section 2.

Triangulation is an integral part of simulation in many applications, such as finite element methods, visualization, gaming, crystallography, and GIS. Thus, there is a broad literature exploring the improvement of this algorithm by extension to HPC platforms. Whereas the parallel Shepard's algorithm is a relatively straightforward (coarse-grained) decomposition of the problem, approaches to parallel Delaunay triangulation are complicated by data dependencies between partitioned processes. This requires code to be written to handle relaying of border information between processors and introduces communications overhead. Meanwhile, the parallelizability of this algorithm hinges on computational power at individual nodes. The tradeoffs between computational power at the processor level and network communications fabric relegate most of the parallel algorithms to the realm of "medium-grained parallel."

The algorithm outlined in [Blelloch] is the most promising for three reasons. First, the algorithm uses a "divide-and-conquer" approach, which is well suited to implementation on a commodity cluster. Second, unlike many other treatments in the literature, the algorithm is detailed in the paper (making it reproducible), and has been implemented and tested with practical problems

(millions of points). Finally, the authors provide extensive performance figures and analyses for several systems. These figures can be used to validate an implementation of the algorithm.

In summary, the Blelloch algorithm in terms of parallel implementation distills to the following:

- Partitioning—the Delaunay algorithm is medium- to fine-grained parallel: subregions of the dataset of interest may have a serial Delaunay performed upon them. The ISR team proposes using a kd-tree partitioning approach to evenly subdivide the dataset into balanced subregions for the approach selected [Marner]. In general, there is substantial communication and agglomeration overhead associated with this partitioning, due to the need to re-connect the regions along Delaunay edges. The Blelloch algorithm identifies the Delaunay edges between the regions during partitioning. These edges are used in the initial triangulation of both regions. This removes the necessity to generate new triangles to complete the joining of two regions and makes possible a coarse-grained algorithm.
- Communication—initial communication is very similar to that used in the Shepard algorithm, in that each processor initially needs access to all available data. The division point and the Delaunay edges are communicated at each stage of the subdivision.
- Agglomeration— Upon termination, once the independent serial Delaunay triangulations are performed on contiguous subregions, there is an accumulation step required to build a single TIN data structure.

Figure 5 and Figure 6 depict a kd-tree partition of 2D points (only fifty are shown). In Figure 5, the entire dataset has been partitioned along the median of the x coordinates (vertical median line.) The left-hand side has likewise been partitioned along the median y coordinate for that half (horizontal median line.) The upper half of the left partition has been completely partitioned along the medians (alternating x and y coordinates.) Figure 6 shows the complete partition after six steps.

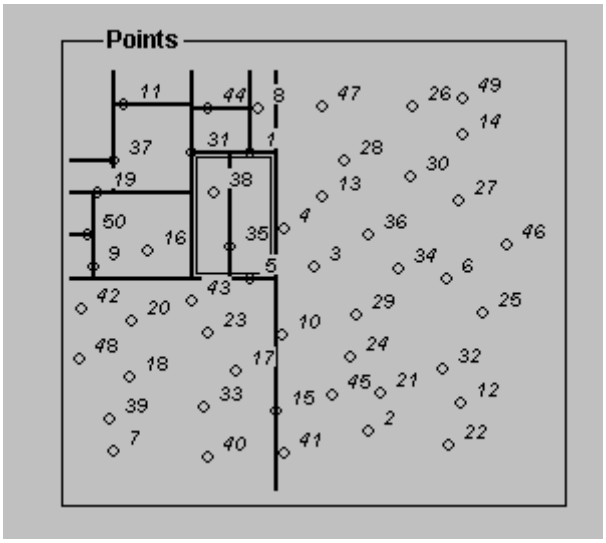


Figure 5. An intermediate stage of kd-tree processing for 50 points.

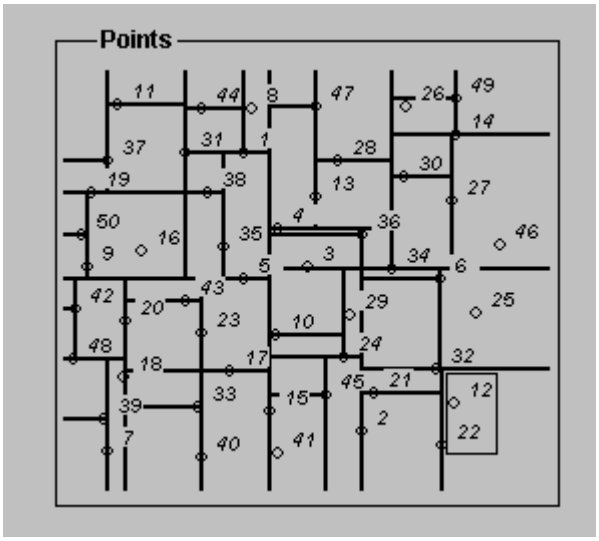


Figure 6. The completed kd-tree partition, using the dataset from Figure .

Each point is partitioned such that nearest-neighboring points are in contiguous regions. For purposes of generating a TIN dataset using Delaunay triangulation, the total number of contiguous regions is set equal to the number of processors involved in the computation. Each region may be triangulated separately, and the regions sewn together in a distributed fashion along the borders of contiguous regions.

In terms of the final algorithm used, it turns out the the kd-tree approach is too computationally expensive relative to a simpler partitioning into strips, as is done for the Shepard rasterization algorithm. This approach to partitioning is particularly suited to very large, high-density LIDAR datasets, in that simple strips contain many points, diminishing the likelihood of introducing artifacts into the triangulation: fewer points per strip would likely cause a 4-processor triangulation to yield significantly different results than a 40-processor triangulation.

This simplified approach still relies on the Blelloch algorithm for the agglomeration (“stitching”) step of the computation, in that it is important to cut the strips along Delaunay edges, which are edges of well-formed Delaunay triangles. In contrast, the Shepard algorithm computational strips are rectangles.

4. Summary and Recommendations

This section summarizes the discussions of hardware and software considerations, and provides a recommendation for a high performance computing system suitable for creating high performance LIDAR data processing algorithms.

4.1 *Summary of LIDAR Algorithms*

Table 1 distills the information presented on the Shepard and Blelloch algorithms discussed in Section 3. The two algorithms contrast in their algorithmic implementation architectural requirements, but both can be accommodated by a distributed memory parallel machine with high-speed communications fabric, i.e. ISR's cluster. While there are performance penalties for the lower speed interconnects of a network of workstations, the partitioning in both algorithms allows the major processing steps to proceed without communication.

Table 1. Summary of Criteria for selection of a High Performance Computing System for LIDAR processing.		
	Rasterization (Shepard)	*Triangulation (Delaunay)*
Algorithm	• partitioning of the algorithm yields independent computational tasks • inter-process communication is minimal	• partitioning of the algorithm yields independent computational tasks, which must be integrated at border regions • inter-process communication is required to handle borders between partitioned sub-regions
Architecture	• application is computationally intensive • rare, large volumes of data transmission • Decomposition is relatively coarse-grained (bandwidth)	• application is computationally intensive • large volues of data transmission upon termination of algorithm • Decomposition is relatively coarse-grained (bandwidth)
Performance	• Data-bound • Communication is centralized at the server where “stitching” occurs	• Data-bound • Communication can become constraint during partitioning and assembly (“stitching”) stages.
Computational Requirements	• Distributed memory • High computational performance	• Distributed memory • High computational performance
Recommendation	• Computational cluster or Network of Workstations	• Computational cluster or Network of Workstations

4.2 *Summary of Hardware Choice*

Research on implementation of representative LIDAR processing algorithms confirms that the distributed memory architecture of a PC cluster or a Network of Workstations (NOW) is an effective high performance computing platform for support of LIDAR data processing. In summary:

- LIDAR Processing software—Representative LIDAR algorithms (Raster and TIN generation) were analyzed on the basis of the following attributes (discussed in Table 1.
  - Partitioning, Communication, Agglomeration
  - Data-parallelism versus instruction-parallelism
  - Performance and computational requirements

- Commodity Distributed Memory Architecture—the tradeoffs implied by this choice are as follows:

Pros	Cons
Inexpensive	Lower Communication Performance
Adaptable to clusters, NOWs, and grids	Specialized Algorithms must be developed
Scalable	Heavy File System access is required in this domain

4.3 *Recommendation*

The research described in this paper supports the use of cluster computers and NOWs for the processing of LIDAR data. The desire for fusing these disparate resources into a single computational entity has yielded the following two design artifacts:

- detailed documentation of Requirements and high-level design for the HPC aspect of the GIS/HPC interface. The design integrates access to both HPC platforms (Cluster and NOW)
- Reference implementation of a common desktop interface to LAS datasets and implementation of raster and TIN generation both on a NOW and on a cluster

4.4 *Approach to Creating HPC Algorithms*

Applying the preceding general discussions to the specific requirements and recommendations yields an approach to designing HPC LIDAR-processing algorithms for deployment on both a Linux Cluster and a Windows Network of Workstations

The first step in creating HPC Raster and Delaunay applications for a Linux cluster and a Windows NOW was to first develop a high-level architecture using a very coarse-grained parallel approach that would work on both systems. This was done by utilizing the SIMD architecture which requires each node in either a cluster or a NOW to run the same instructions on a different slice of the input Lidar dataset. Here, either the full dataset can be used on each node, or data-splitting algorithms can be run in a preprocessing step to minimize the network overhead of transferring the same large dataset to every node. We opted to use the full dataset on each node because the overhead in determining a proper split in the data for the Raster and Delaunay algorithms outweighed the time that would be saved in transferring smaller pieces of data over a high-speed network. In sum, the very course-grained approach simply requires that the proper parameters are given to the instruction set on a given node so that it knows what portion of the input data to process.

After ensuring that a very coarse-grained parallel approach could be applied to both the Raster and Delaunay algorithms, the next task was writing the algorithms so that the same code base could be used in both a Linux and Windows environment. Using standard C++ libraries and portable open source code, Cygwin was used to make Windows executables. The only difference in the code stemmed from the architecture of the systems. Having a dedicated cluster on the Linux side allowed us to use the industry message passing interface, MPI, to initiate parallel jobs, and NFS was used to share data from the cluster's file server to each of the cluster nodes. Since a very coarse grained approach was used, there was no communication between cluster nodes once the job was started and only the main program of the HPC applications contained MPI API calls.

While MPI can also be used in Windows, the various nodes of the NOW are not dedicated since any node in the system can potentially terminate a running job since the node is owned by standard user. MPI is not a wise choice since it depends on communication with all nodes to initiate and finalize a job, and there is no guarantee that the nodes will be available for the entirety of the job. In addition, since there is no built in check pointing for MPI, all data processed is lost or requires administrative intervention to retrieve when a master process in MPI hangs after losing contact with a process from a failed node.

The best alternative in launching a very coarse-grained application on a Windows NOW was to use United Devices (UD). The inherent lack of stability in a NOW is overcome by UD's Grid MP Server which controls the MP agents running on every NOW node. If a node goes down, the UD server knows it and resubmits the job to another NOW node. Thus, once the Raster and Delaunay algorithms were written on the Linux side, all that needed to be done to run on a Windows environment was to launch the program without MPI calls. This was a simple matter of passing the size of the NOW and the work unit index of the node to the main program. This is an intrinsic property of the MPI initialization on the cluster, but for the NOW, the node needed to know what slice of the data to process.

The next section describes the design and development of the reference implementation.

5. Common Interface Design and Development

Given the need for a centralized interface mediating access to disparate HPC resources, the formal list of requirements (provided with system documentation) was generated. To meet these requirements, an *n*-tiered high level design was produced. This section discusses this design. Additional detail describing the required software interfaces/API is found in the system documentation.

5.1 *High Level Design*

Figure 7 depicts a general high level design for the required system attributes. The high level design for this system comprises three tiers:

- A front end, which is the ArcGIS software and extensions

- A back end, which is either a Windows-based Network of Workstations (NOW) or a Beowulf-class LINUX computation cluster
- A Virtual HPC resource server, which is a façade mediating exchange of requests and information between the ARC/GIS extension client and backend HPC resources

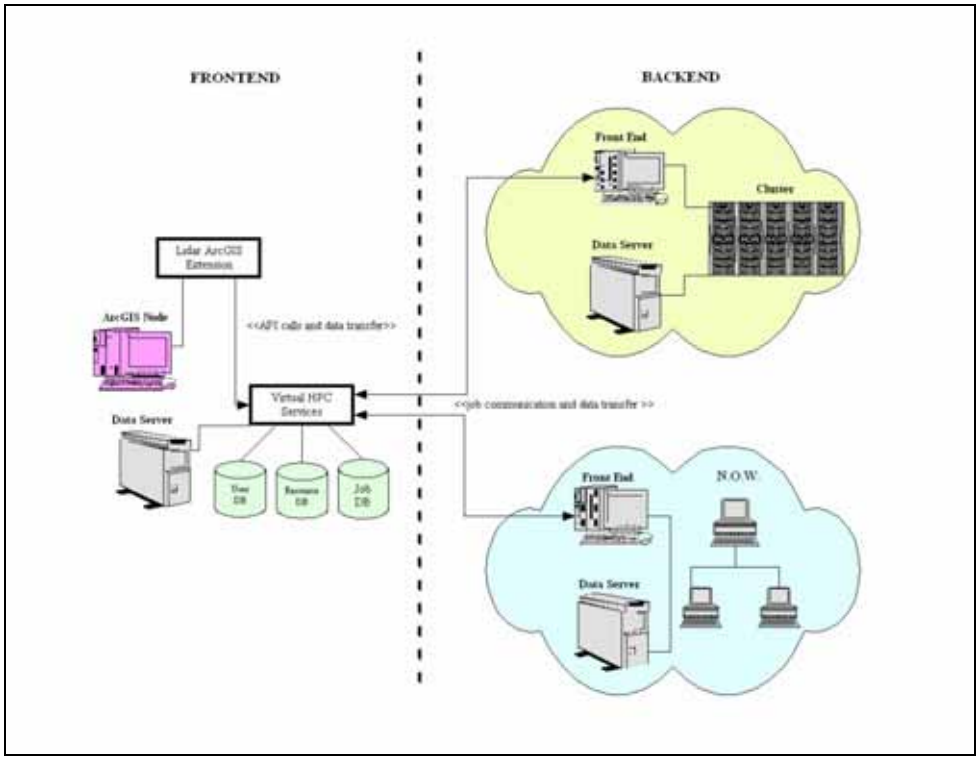


Figure 7. A Generalized High level design to support the requirements for the required system.

While initial prototypes for the system were rapidly implemented using the .NET framework, the COTS package, United Devices Grid MP was used for final deployment. The Grid MP framework arranges the individual system components as depicted in Figure 8, which depicts a single "MP Agent." The MP Agent resides on each node in a parallel system, and brokers communications with a central Grid MP server. In turn, this server brokers communications with a front-end client residing on a workstation. Thus, Figure 7 is transformed to Figure 9 for the United Devices framework.

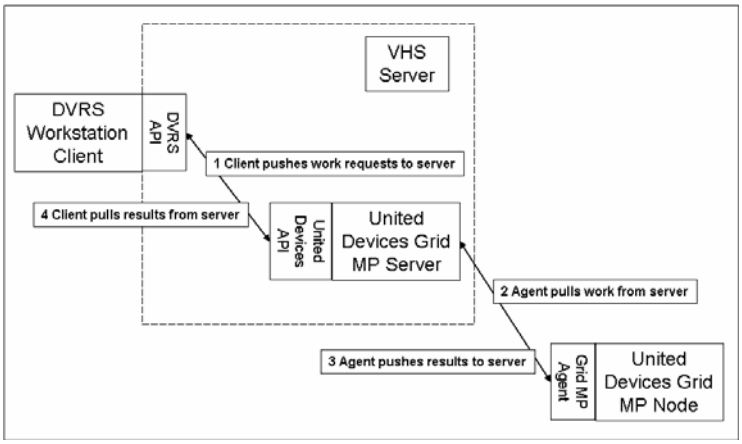


Figure 8. The United Devices configuration for a single Grid MP Node. The DVRS workstation client, implementing the API designed in Phase II, communicates with the Grid MP Server (VHS Server). The Grid MP Server mediates work requests and delivery of results between the Grid MP Node and the client workstation.

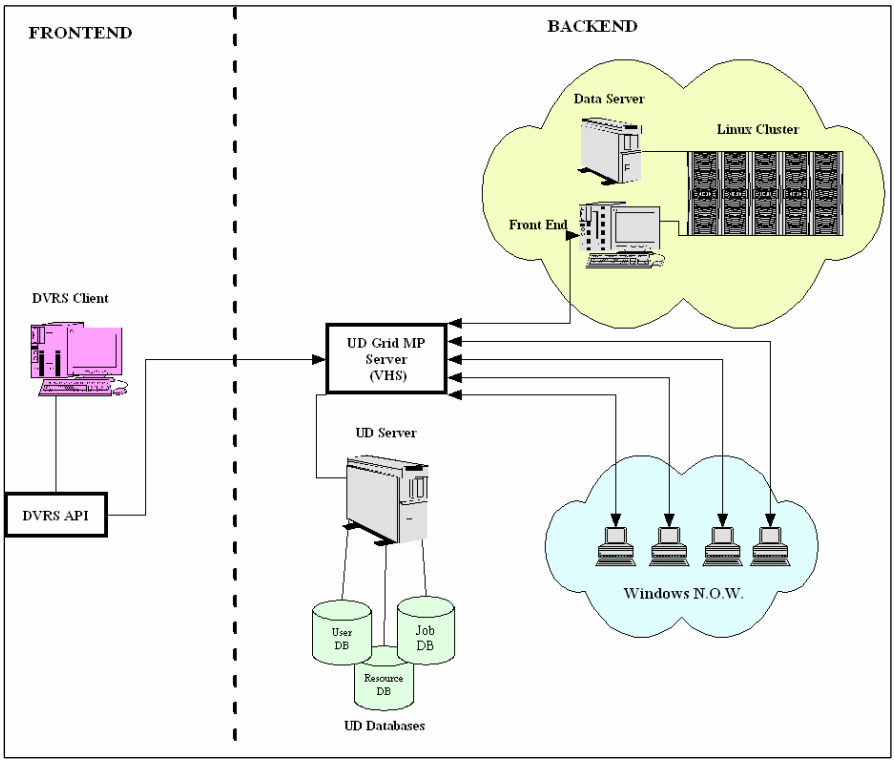


Figure 9. The generalized High Level Design adapted for use with the COTS United Devices Grid MP package.

This design accommodates three features required of this system:

- Transparency—the underlying computational resources, which comprise multiple heterogeneous computing devices, are abstract from the perspective of the ArcGIS client user. Both NOW computation and cluster computation are accessed through a single software entity.
- Openness—use of .NET web services and the COTS United Devices software facilitates the interchange of data and job requests between the heterogeneous systems.
- Scalability—the modularity of the design allows addition of new algorithms or additional HPC resources.

From a practical standpoint, the reference implementation demonstrates these three desirable features and lays the groundwork for integration with the United Devices Grid MP software. In turn the United Devices software will add robust data partitioning, data delivery, user authentication, and transmission of results back to the user. The façade presented by the VHS services will remain, and will continue to marshal the requests defined in the interfaces section. The United Devices software will connect to the VHS backend transparently to the ArcGIS extension client.

5.2 *Implementation—.NET middleware*

The initial design of the HPC software services requires infrastructure to support extensibility. A .NET Web Services-oriented implementation provides this and fits well with the eventual integration of Grid MP into the system. The initial software demonstration of this architecture comprises essential interfaces and services for deploying raster processing jobs on both a NOW and on a cluster. The design also demonstrates the aforementioned extensibility, in that the GRID MP software—which provides the bulk of the data-handling, user management, and security functionality—will “plug and play” with the .NET design. Conversely, the development mode for raster algorithms described in Section 2-4 provides plug and play attachment of new algorithms to the system.

Core front-end user interfaces and basic operations such as job requests are implemented using .NET. The production-level implementation intends to use .NET user interfaces and application programming interfaces (APIs), integrated with the Grid MP software.

5.3 *Implementation—Grid MP*

In terms of the functionality required by the HPC LIDAR data processing software, Grid MP possesses many useful attributes, all of which complete the functionality missing from the initial .NET reference implementation.

There are three categories of usage in the Grid MP realm, which combine to meet the software requirements of the system:

Application User—Uploads executables and data to the Grid MP platform; creates jobs and submits them to run on the MP Agent machines; monitors and downloads results.

The architecture of Grid MP lends itself to the tiered design described in Section 5 and in the appendices. Most functionality from the user perspective is accessed via command-line scripts and programs, which can be wrapped and abstracted by an appropriate Web Services interface. For example, essential commands include the following:

`mpsub`—this is a batch submission utility for submitting an executable and data to a set of NOW nodes in a single unit of work. It is intended for use with computational requirements that may change frequently. Application users run mpsub by submitting all the programs for the application and its input data to the Grid MP platform as a "job."

`ud_mpirun`—enables MPICH-based parallel jobs to run on the Grid MP platform. The interface enables any arbitrary MPICH executable to run on supported operating system platforms. Input is from the command line or from a configuration file. This function command bridges the feature gap between clusters and NOWs in that development for both can occur using a single message-passing library. MPICH was used in the original development on the Black Diamond cluster and is universally accepted as the standard programming library for parallel and distributed programming on distributed-memory systems.

`mpresult`—retrieves results produced by `mpsub` batch job `ud_mpirun` submission. `mpresult` also enables the viewing of all running jobs, stopping jobs, and deletion of jobs.

Application Developer—Ports, develops, and deploys applications for the Grid MP platform; creates application services that perform application preprocessing and postprocessing for use by the application user.

The most useful software provided here is the API-level MP Grid Services Interface (MGSI), which is a programmatic interface that provides third-party applications with a way to access the MP services and databases. The MGSI defines a large number of functions for controlling the system, available in XML-RPC, SOAP, and HTTP-based interfaces. These interfaces facilitate the integration of the MP Grid software with the specific system design for LIDAR extensions.

System Administrator—Installs, configures, and maintains the Grid MP platform software; manages user security, service performance, and device usage

Most of the advanced functionality required of the system design is made available through the Grid MP administrative software. The Grid MP Administrative features allow control of the following subsystems, each of which has an analog in the high-level design for the final system:

- Configuration management: security, username/password control, user privileges and roles
- Platform management: organization and control of node usage, status
- Workload management: data and application control, job and performance monitors
- Databases: Server information, Application information, User information
- MP Agent control: the agent is a distributed lightweight program which manages job processing on distributed nodes.
- Service management: starts and stops dispatch, file, poll and architecture ("Realm") services, centrally ensures the integrity of all running services.

Additional useful services provided by the Grid MP platform include the following:

- Poll services: collects periodic status reports from MP Agents and communicates commands to the MP Agent from other services (e.g. login, reset, abort, snooze, shutdown); gathers load information; system heartbeat monitoring
- Dispatch services: schedules work units to devices based on device capabilities and availability; receives result status from those devices; manages idle devices
- File services: SSL file downloading, results uploading, file maintenance, MGSI usage

Again, these features are consistent with the system design, and provide a COTS solution to the infrastructure required by the LIDAR extension concept. Thus, the choice of Grid MP is justified on all technical fronts.

5.4 *Communications fabric and OS choice*

Because the design decisions are governed by the choice of United Devices Grid MP, the network communications fabric and OS choice are made based on Grid MP requirements. For the Windows-based NOW and Linux-based cluster, this results in the following choices/constraints:

- Linux: existing high speed network, existing MPI software
- NOW: existing Ethernet network, software ported to Windows executable environment
- External access: HTTP page//CSharp Windows form (cross platform)

These are appropriate choices independent of Grid MP, in that no special networking protocols or hardware are required for deployment.

6. Benchmarks and Performance Assessment

This section explores the performance of the two components of the distributed system developed for this project. For both the NOW and cluster architectures, the following runs are made, along with the metrics enumerated described below:

- Rasterization code: parallel runs for a representative range of nodes on 2 different cell-size resolutions. The datasets used include five representative .LAS files.
- Triangulation code: parallel runs for a representative range of nodes, using the same five .LAS input files used for rasterization.

The following test metrics are compiled and presented:

- *Raw execution times*—Displays actual run times for the selected range of algorithm parameters and input data. Gives an indication of the raw performance of the system as the number of parallel nodes is varied.
- *Speedup*—As described in Section 2.3, the speedup gives a dimensionless indication of the performance of a system as the number of parallel nodes is varied. In contrast to the

raw execution times, speedups can be compared independently of actual wall-clock run time for an algorithm.

- *Efficiency*—Also described in Section 2.3, the efficiency providces a comparison to the ideal case for the empirical speedups.

The Figures on the ensuing pages contain plots of the performance metrics. Detailed numerical results are enumerated in the Appendix.

6.1 *Performance metrics for Raster Algorithm running on NOW and Cluster*

There are several high-level observations that pertain to all of the results in this section.

1. There are characteristics of the HPC LIDAR algorithms and data that manifest in every performance chart:

   Recall the discussion of Amdahl's law in Section 2. The execution time of each algorithm comprises a serial and a parallel execution time, T_s and T_p.

   For the triangulation and rasterization algorithms, these Time components are divided between the parallel computations and the serial agglomeration (or "stitching" steps), as depicted in Figure 10. The stitching step of each algorithm requires global information agglomerated from the individual computational nodes. This effect ultimately affects scalability (or the effect on performance of adding additional processors) and is discussed with each plot.

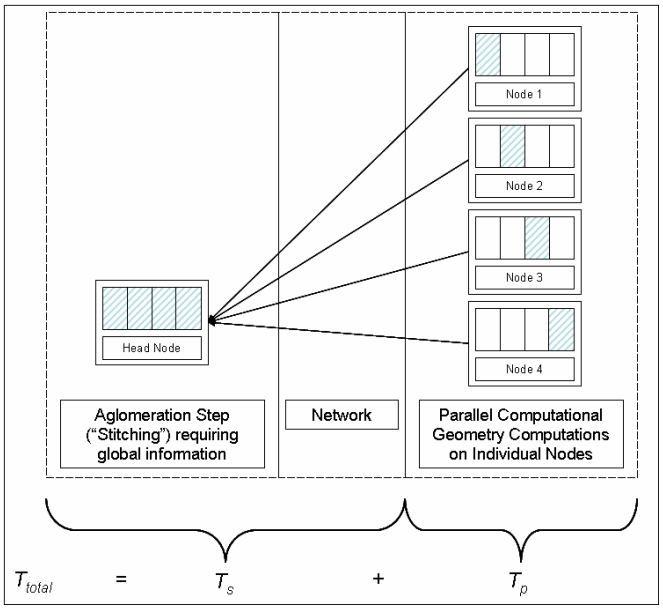


Figure 10. The types of algorithms used in LIDAR processing comprise both a parallel component, T_p, and a serial component, T_s.

2. In comparing the performance of algorithms on the Network of Workstations, it is important to note two things:

   - The nodes in the Network of Workstations used in the Phase II tests are quite heterogeneous in terms of RAM available (ranging from 256 MB – 2 GB), processing power (ranging from a single-processor at 800MHz to a dual-Athlon server-class unit). Indeed, heterogeneity is generally the case for networks of workstations, with the exception of captive server farms (often referred to as "clusters," though not the same as the Beowulf-class cluster used in the work described here). Thus, the results reported here should be indicative of results expected in a full-blown production deployment of the system.

   - Tests were constrained to less than 10 available network workstations. Because four nodes were available for testing at all times, and the additional nodes were only sporadically accessible, all results are reported for the range of 1-4 nodes. While this is enough to capture trends in performance (i.e., the general shape of Figure 11 for the Network of Workstations is consistent with the cluster results of Figure 13 on 1-64 processors), it is likely that this is an inadequate number of nodes to demonstrate scalability to large numbers of processors on the network.

3. When examining the results for the HPC algorithms running on the Linux cluster, it is important to stress that both algorithms were originally developed on the cluster and then ported to the Network of Workstations.

   On the cluster, the algorithms were designed using conventional parallel development approaches: MPI message passing, high speed network interconnectivity, a coarse-grained "master-slave" programming paradigm. Deploying these same algorithms to the NOW was useful in terms of code portability (the same core computations are used on both systems), but the code is optimized more closely to the homogeneous cluster and infrastructure than to the NOW.

   In addition, the way in which the United Devices Grid MP software is used differs between the cluster and the NOW. On the cluster, a single Grid MP agent brokers the MPI job requests from the client and MPI execution on the cluster, in a sense preserving the performance of the algorithms. On the other hand, the NOW implementation of the Grid MP infrastructure includes a Grid MP Agent residing on *each* node on the networkr, granting control to the Grid MP software and removing control from the algorithms themselves. This is a conventional approach to configuring a NOW, but can have a bearing on any perceived performance differences, especially in terms of raw execution times.

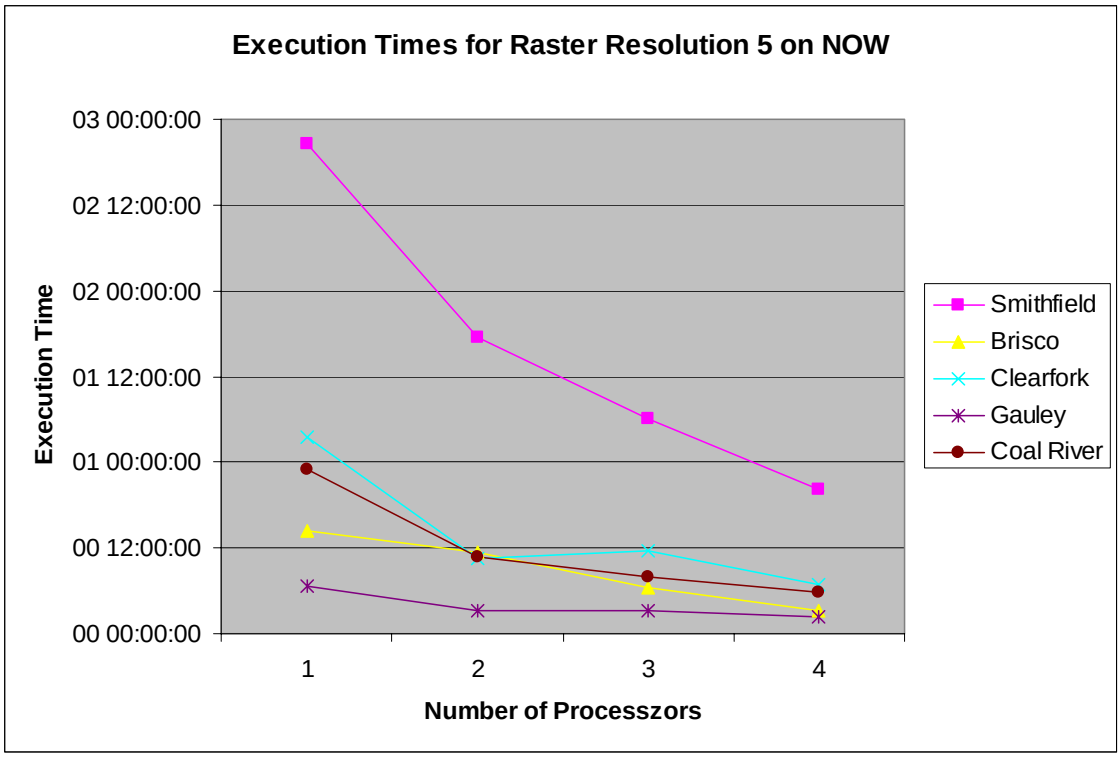


Figure 11. Raw Execution Times for Raster Resolution 5 on NOW

Figure 11 depicts the raw execution times for a raster at resolution (cell size) 5 on the NOW. As processors are added to the system, the execution times decrease asymptotically. The failure of the execution time to decrease between 2 and 3 processors for the Clearfork dataset is attributed to lack of load balancing on the heterogeneous system. Since not every processor is doing the same amount of work (irregularly shaped input datasets) and not every processor has the same computational power, adding a processor to the network doesn't guarantee improved results. Adding another processor did in fact resolve this discrepancy and continue the trend towards decreasing raw execution time.

In order to make a fairer comparison between the cluster and the NOW, *N* processors were given *N* units of work. In practice, UD Grid MP is capable of delivering *M* units of work to *N* processors, where $M>N$. Since this creates more potential work to dole out to individual processors as the processors complete their computations, this enhances load balancing. In production-mode processing of very large datasets, it will be desirable to divide up the work as finely as possible, to allow the system to load-balance in this manner.

Similar results are seen for Figure 12, raw execution times for resolution 10 on the NOW. Here the Smithfield datset failed to exhibit improved results from 2 processors to 3, but the trend continued with addition of a fourth processor.

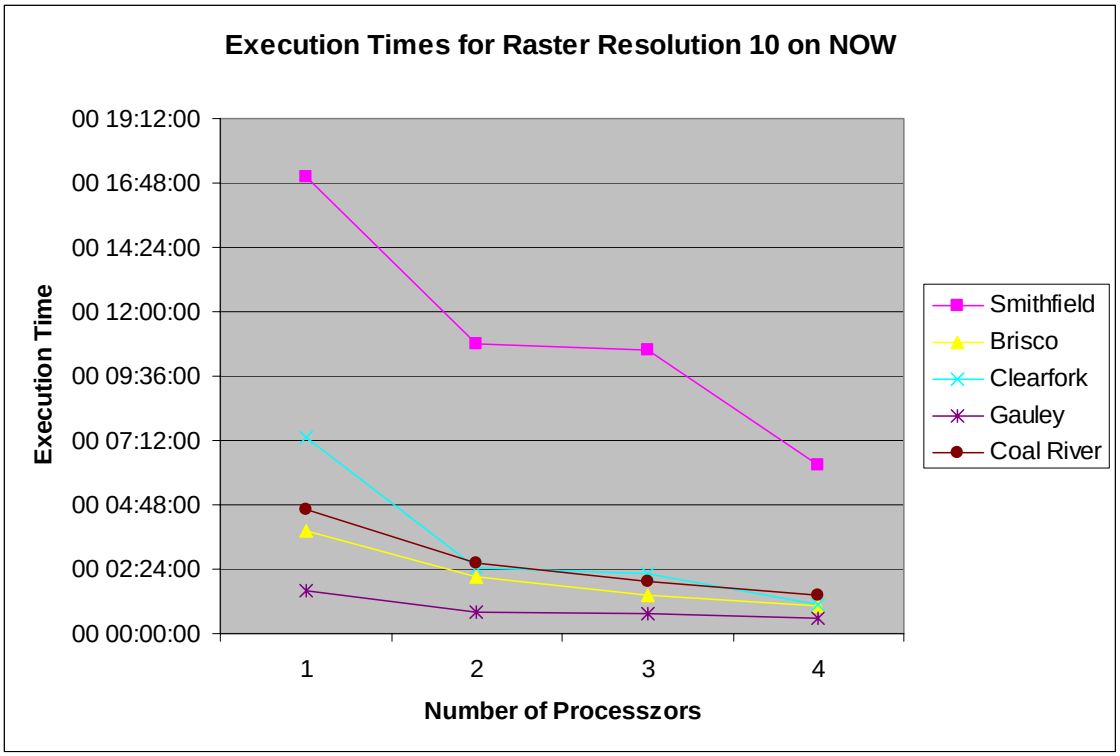


Figure 12. Raw Execution Times for Raster Resolution 10 on NOW

The Speedup plots (Fig. 13 and 14) for the raster computation, show the actual performance increase relative to an ideal speedup. The ideal speedup is typically not exceeded in the theoretical sense, but it is clear that on the NOW, the trend is towards undershooting the ideal and then besting the ideal as more processors are added. This is attributed to the heterogeneous RAM available to each processor. As processors are added to the system, there is a decrease in the amount of RAM required for any individual processor to compute $1/N$ of the global result. This results in less use of virtual memory/memory paging. Virtual memory's use of the hard disk for temporary storage of RAM out-of-core often slows down memory access by 1-2 orders of magnitude (i.e. RAM access is 100 times as fast as disk access). Thus, there is an apparent exceeding of the ideal speedup, because processors can complete computations entirely in RAM without resorting to the hard drive as is the case for fewer processors.

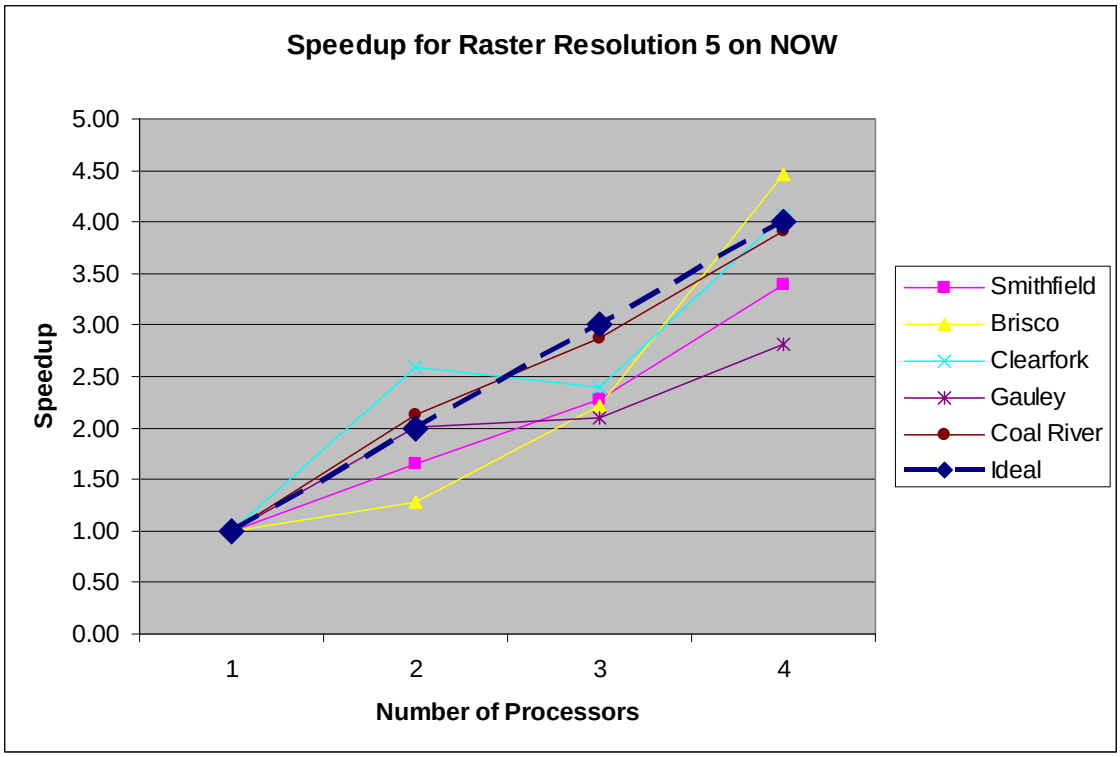


Figure 13. Speedup for Raster Resolution 5 on NOW

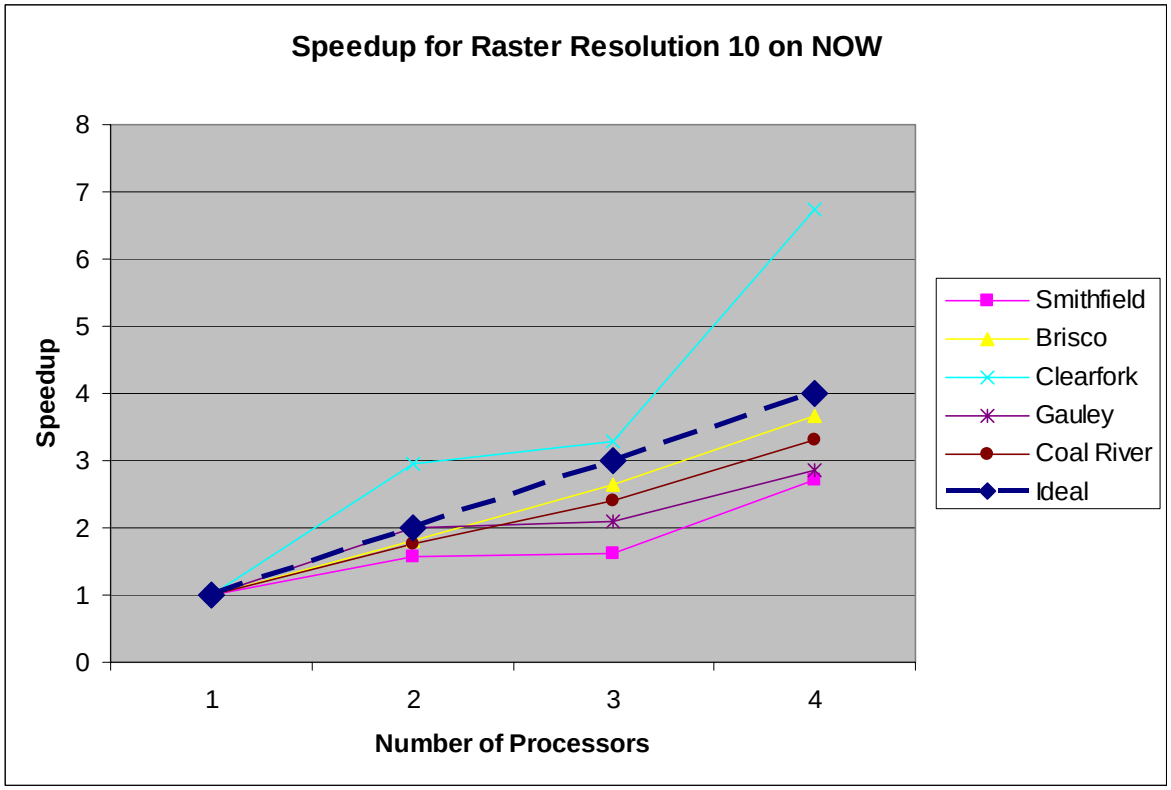


Figure 14. Speedup for Raster Resolution 10 on NOW

The cluster results in Figures 17-20 are similar to the NOW results. The plots appear much smoother, because the runs were made over many more processors (1-64 versus 1-4), but the general trends are similar: significant decreases in run time, and apparent exceeding of ideal performance due to computations completing in RAM versus virtual memory as processors are added.

A further manifestation of the memory issue is apparent in the efficiency plots in Figure 21 and 22 for the rasterization on the cluster. The apparent efficiency is higher than ideal ("100%").

One conclusion that might be drawn from this is that it would make sense to keep adding nodes to the system to achieve infinite speedup and infinite efficiency. Of course, there is an inflection point after which adding nodes will cause a decrease performance. This occurs as the serial portions of the computation (T_s) begin to dominate the parallel portions (T_P). Recall that the parallel portion of a computation can be improved by adding nodes, but the serial portion cannot. For example, Figure 15 shows a plot of speedup for a theoretical computation in which the serial time is 25% of the total time on a single processor (ignoring RAM and network issues). Plotting $T=T_1/(T_P + T_P)=1.0/(0.25+ T_P)$ yields the tapered result seen in the figure, as compared to "ideal." Clearly there is an performance asymptote that the serialized portion of the code forces onto the system. Furthermore, introducing the real-world aspects of network latency and congestion into this model results in diminishing

returns at the point where network communications of input data and results occurs. In fact, tests on the Windows network show that at a work unit count of greater than 64 distributed across 4 processors, network congestion and serial "stitching" begin to dominate the computations, and the speedup actually begins to decrease. Raw data for this is found in Appendix B, with a plot in Figure 16.

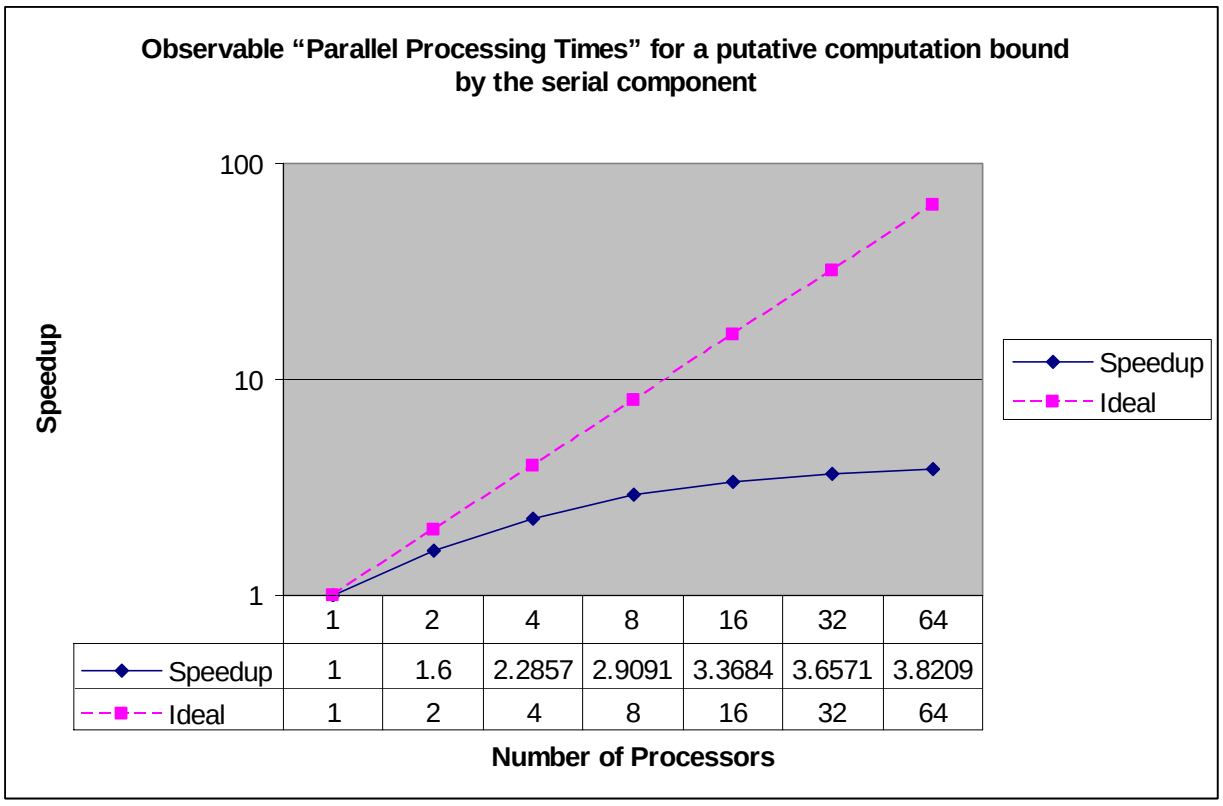


Figure 15. A significant serial component in a parallel system can force diminishing returns on speedup as more processors are introduced.

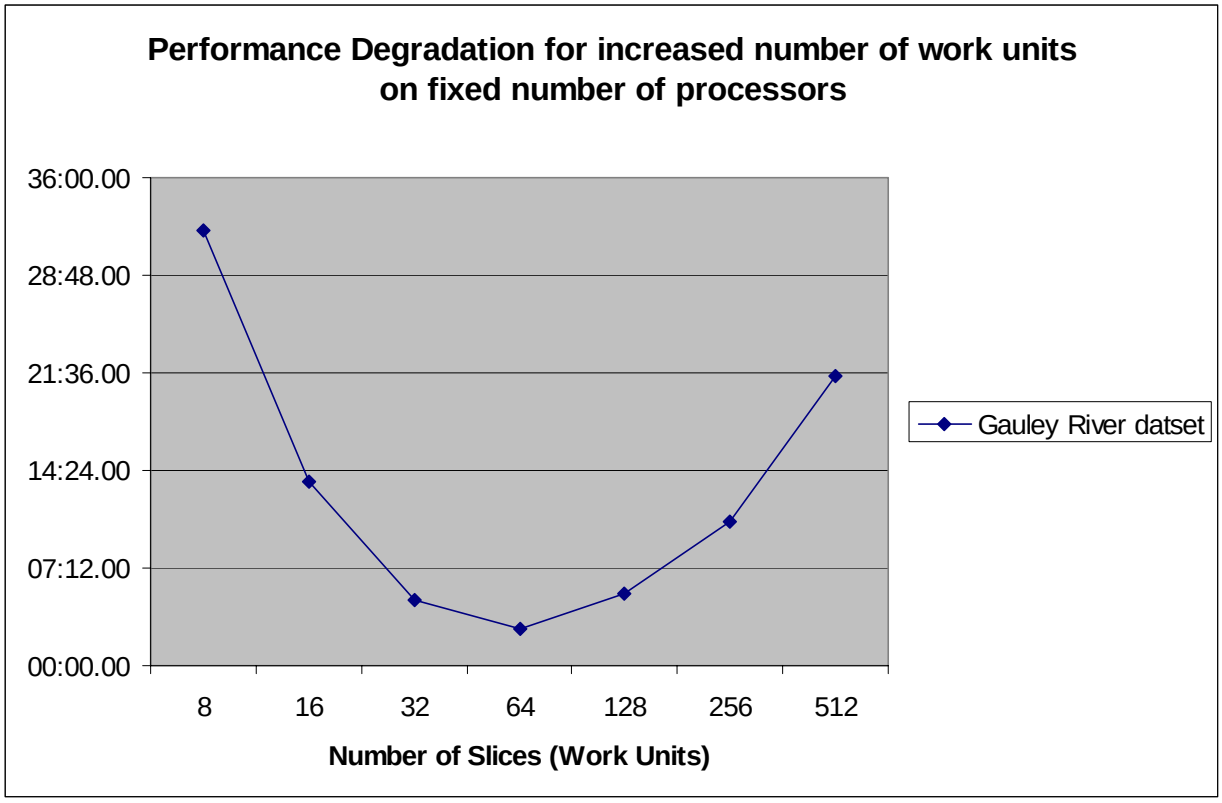


Figure 16. For high numbers of work units, as might be assigned for load balancing, network congestion and serial "stitching" begin to dominate the computations, and the speedup actually begins to decrease.

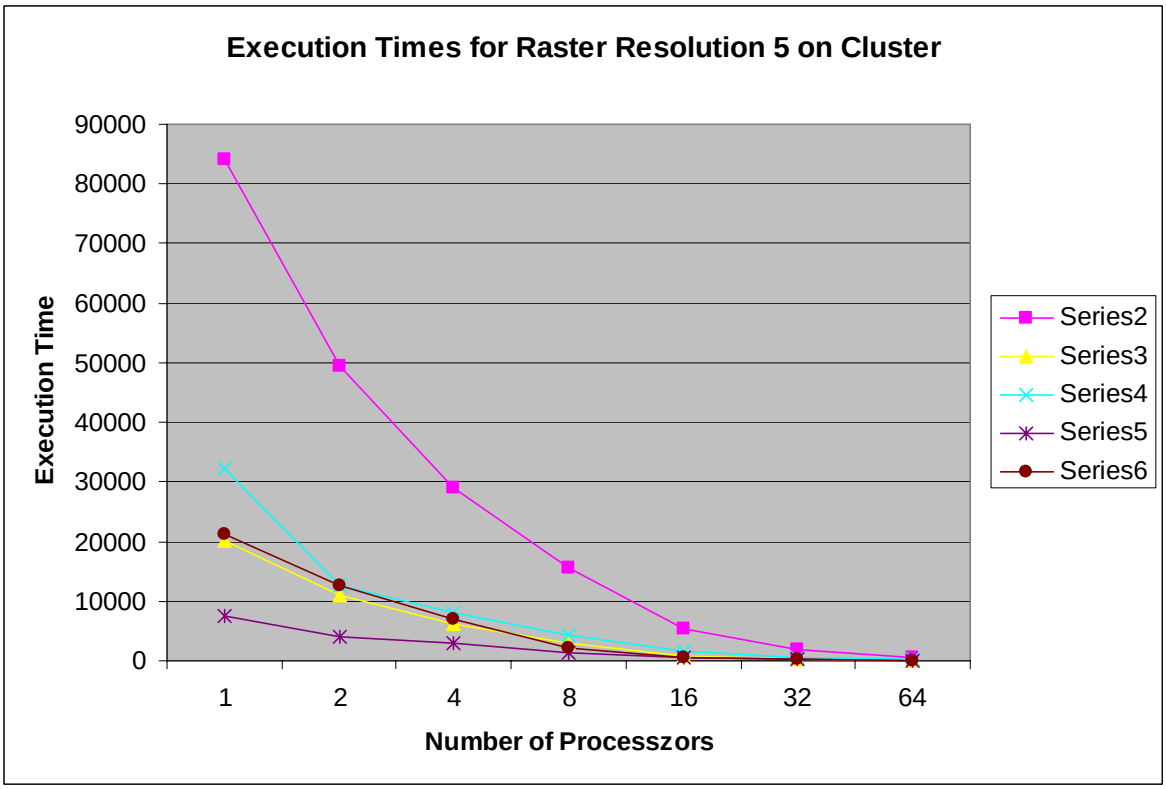


Figure 17. Raw Execution Times for Raster Resolution 10 on Cluster

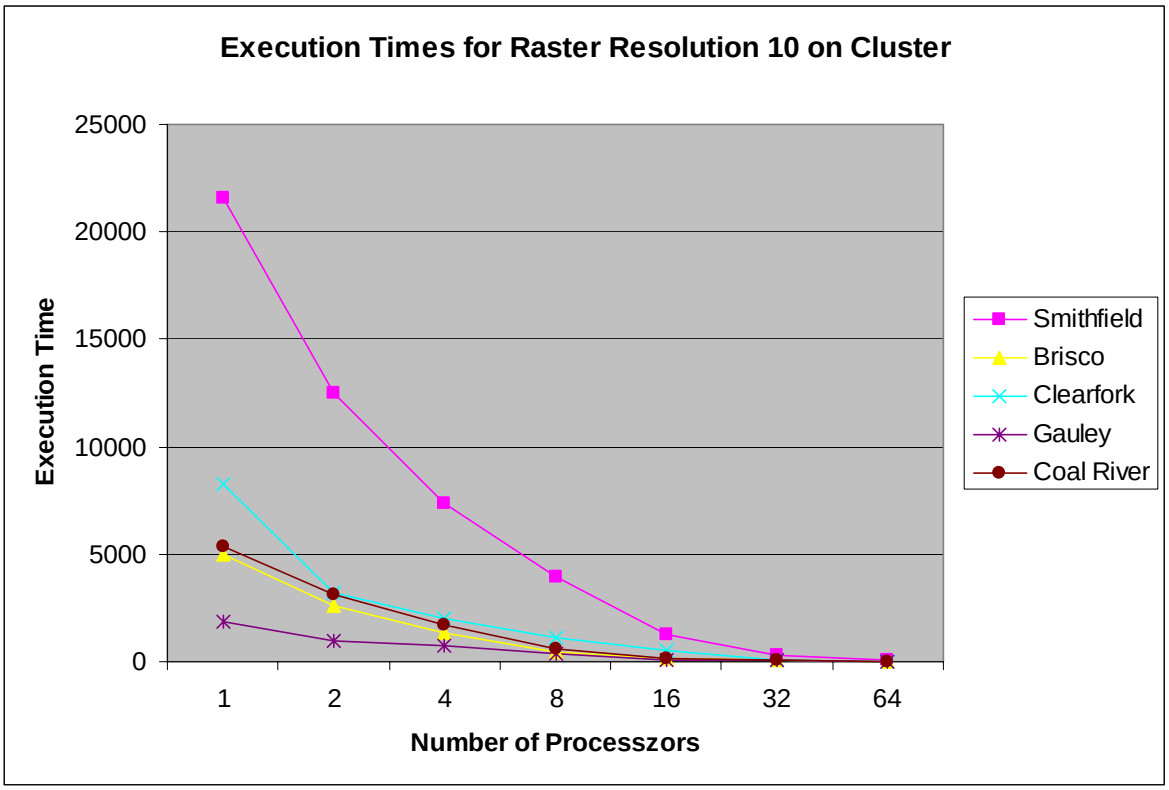


Figure 18. Raw Execution Times for Raster Resolution 5 on Cluster

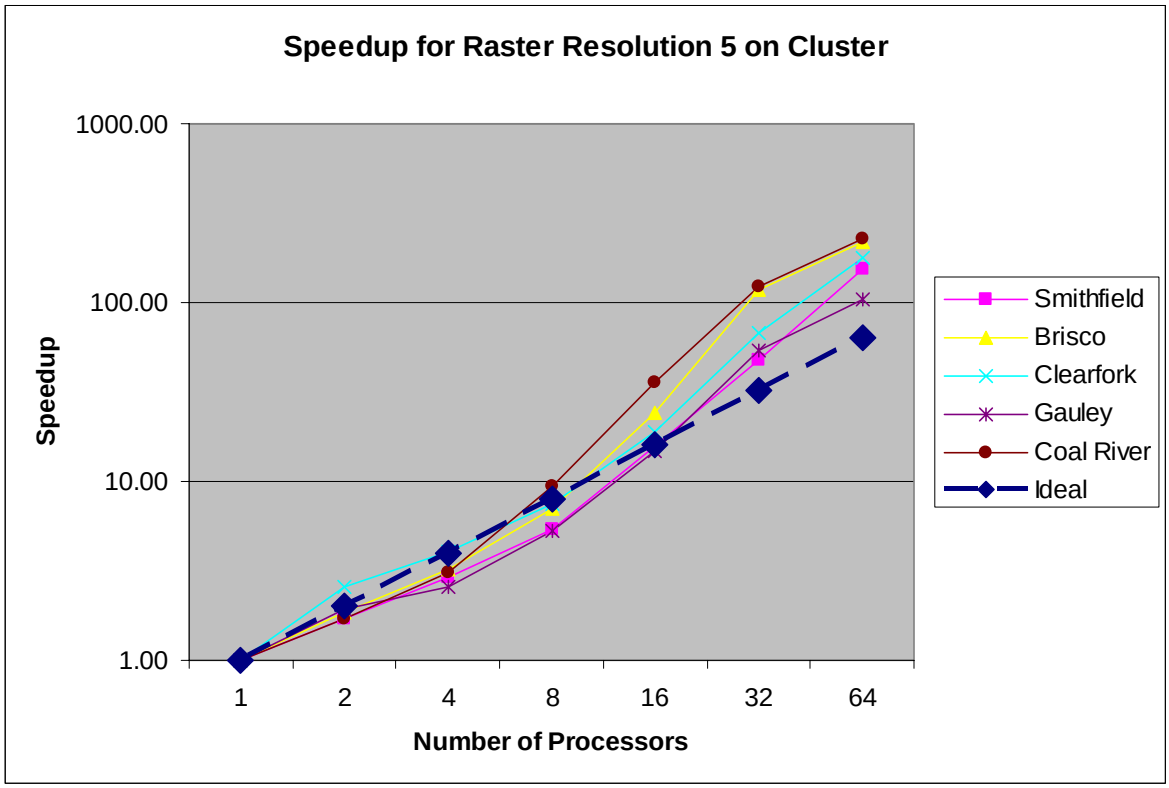


Figure 19. Speedup for Raster Resolution 5 on Cluster

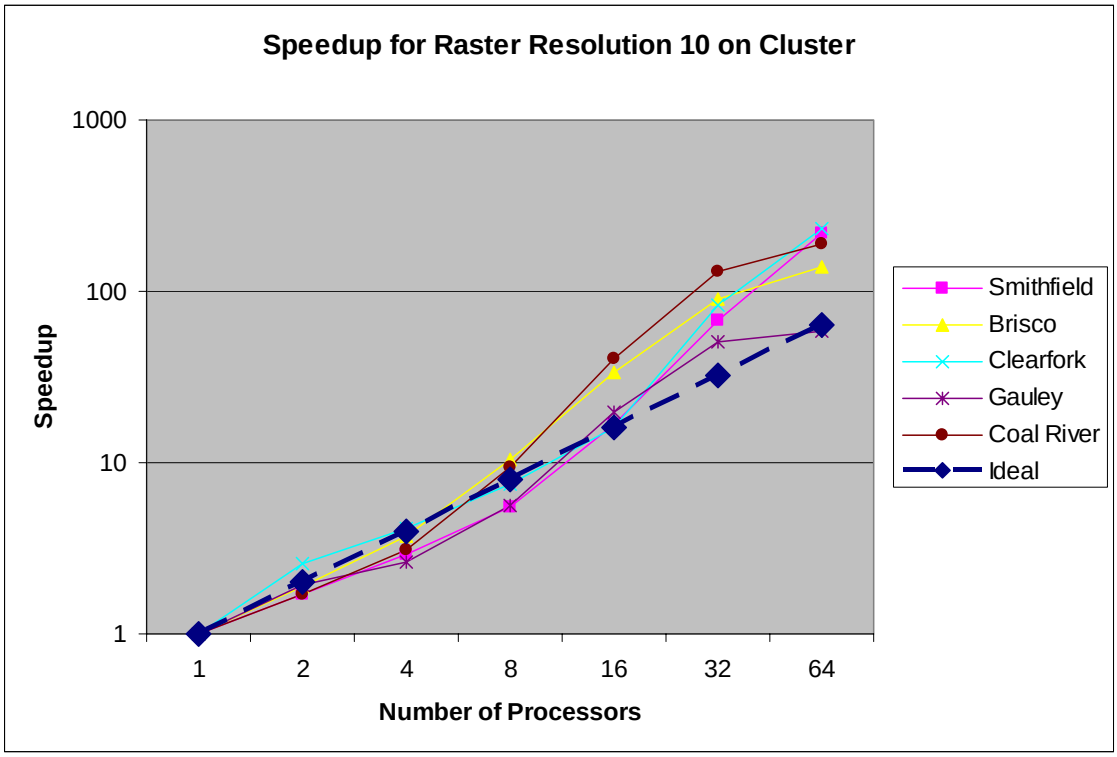


Figure 20. Speedup for Raster Resolution 10 on Cluster

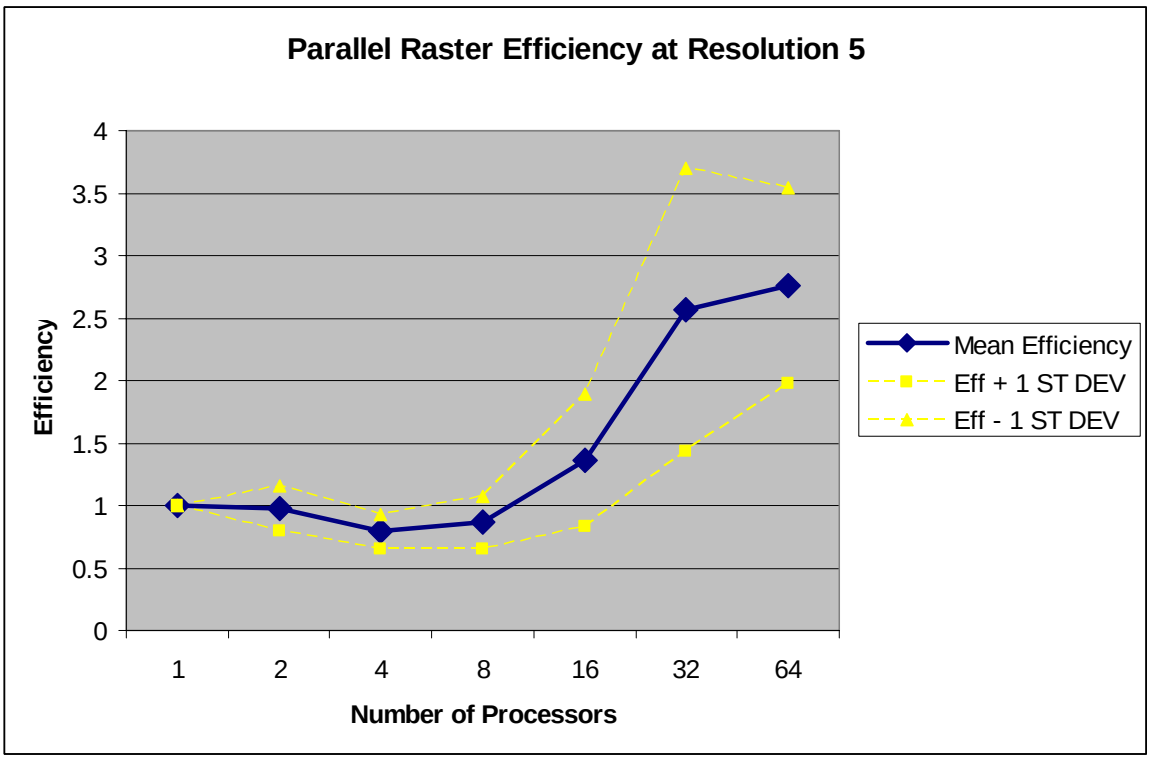


Figure 21. Parallel Raster Efficiency at Resolution 5

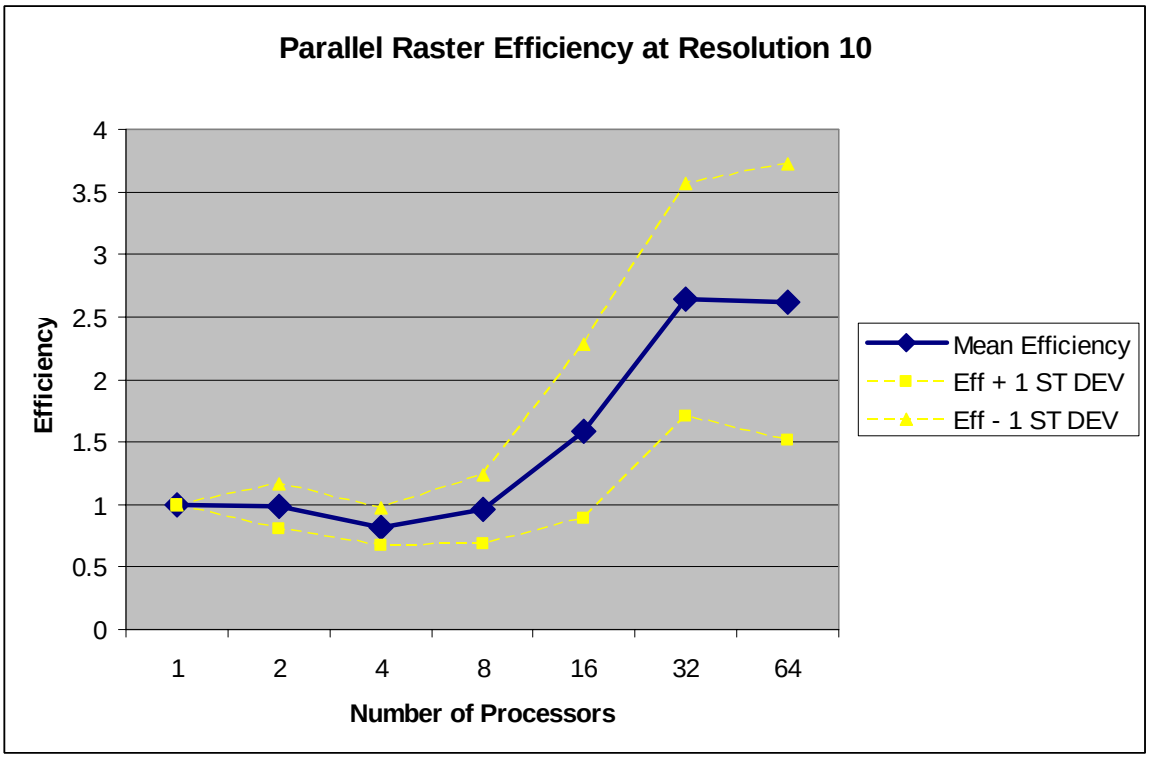


Figure 22. Parallel Raster Efficiency at Resolution 10

6.2 *Performance Metrics for Delaunay Triangulation Algorithm running on NOW and Cluster*

The parallel Delaunay triangulation is coarse-grained parallel and thus the Delaunay performance metrics show similar trends to the triangulation metrics. Note how the Speedup dramatically exceeds the Ideal after 3 processors. Again this is attributable to computations completing in RAM which previously required virtual memory utilization on fewer processors. The Delaunay computation is more RAM-intensive than the triangulation. This is the source of the exaggerated overshoot.

Interestingly, on the cluster, where the range of processors is 1-64, the full gamut of performance can be observed. The speedup actually overshoots the ideal early for the Coal River dataset, but then all datasets exhibit asymptotic behavior due to serialization after 32 nodes, with Coal River dipping back below ideal. This is reflected in the efficiency plot (Fig. 27) which depicts diminishing returns of approximately 50% efficiency at 64 processors.

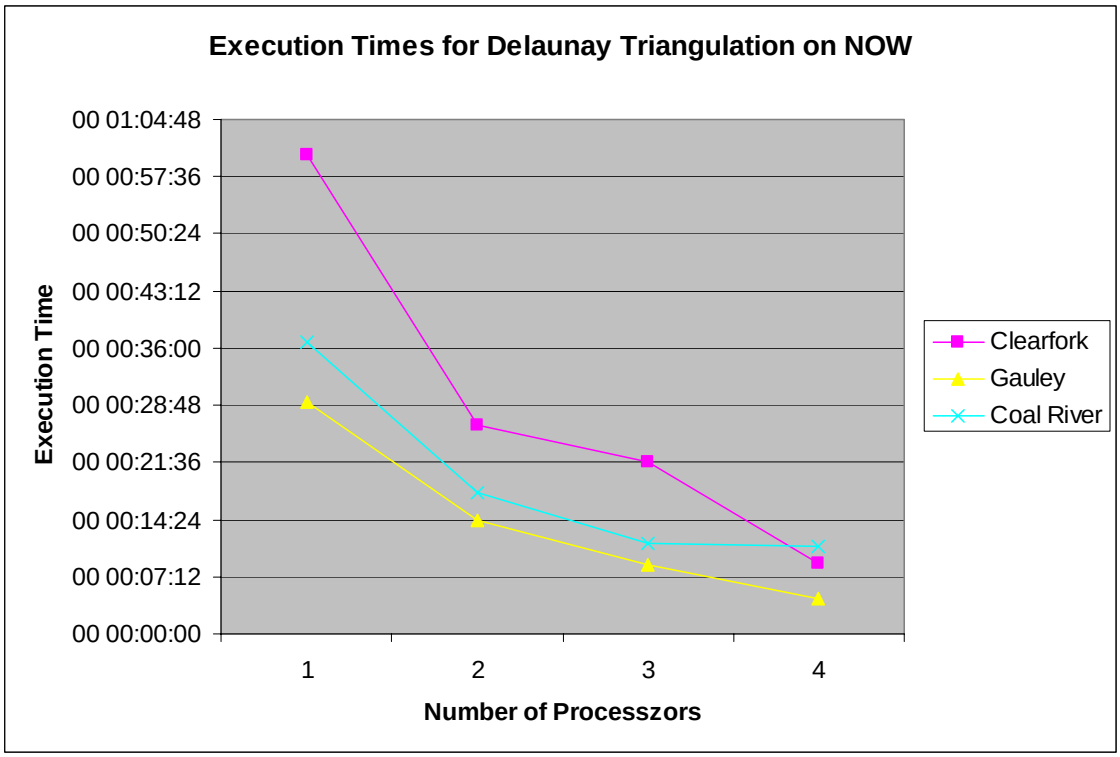


Figure 23. Execution Times for Delaunay Triangulation on NOW

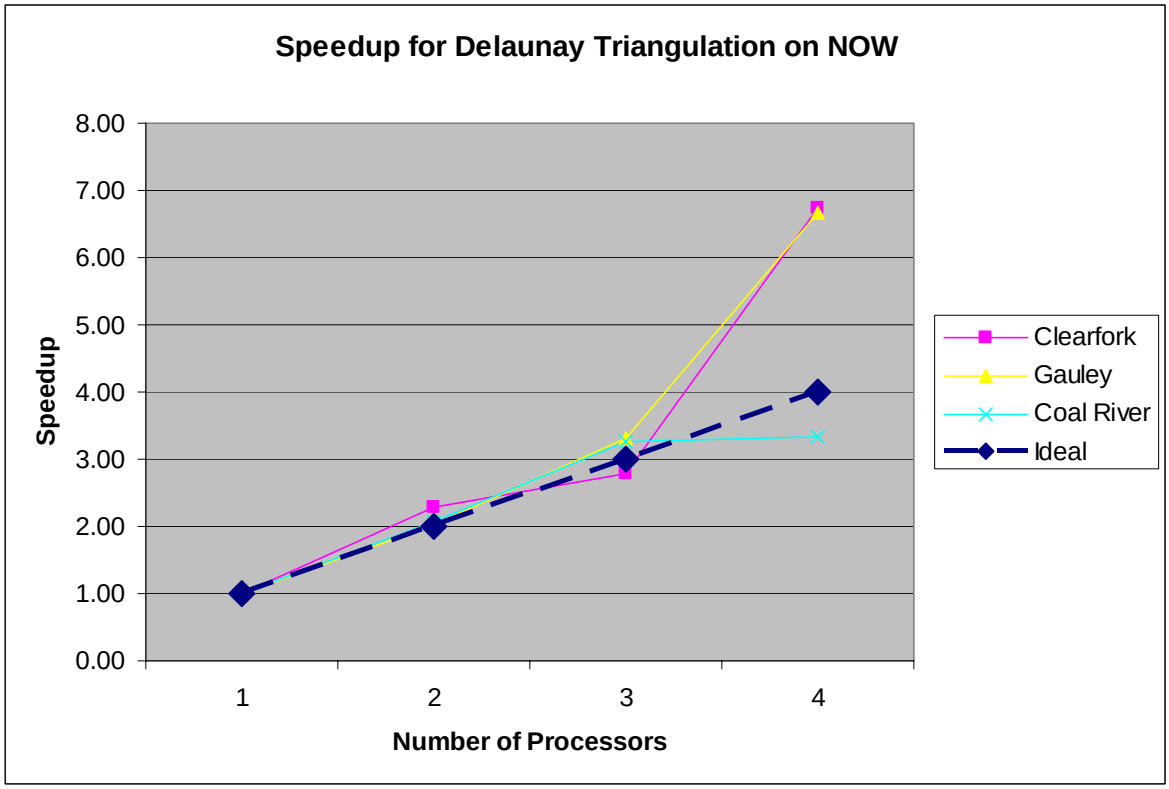


Figure 24. Speedup for Delaunay Triangulation on NOW

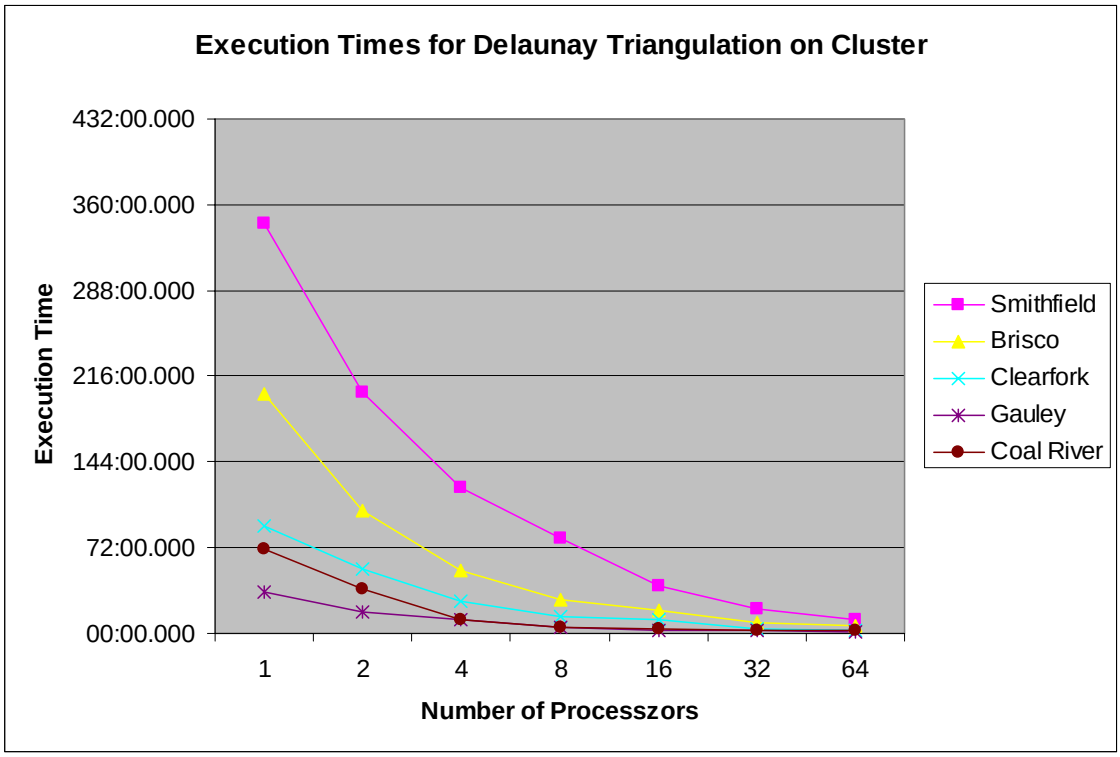


Figure 25. Execution Times for Delaunay Triangulation on Cluster

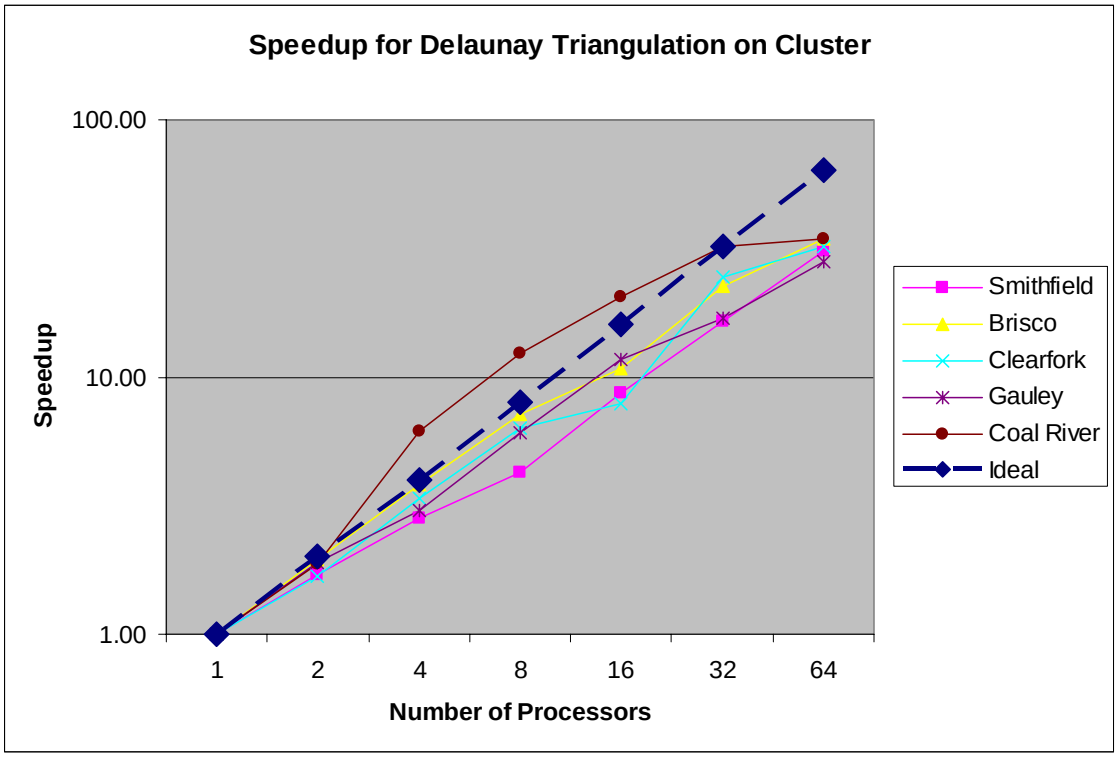


Figure 26. Speedup for Delaunay Triangulation on Cluster

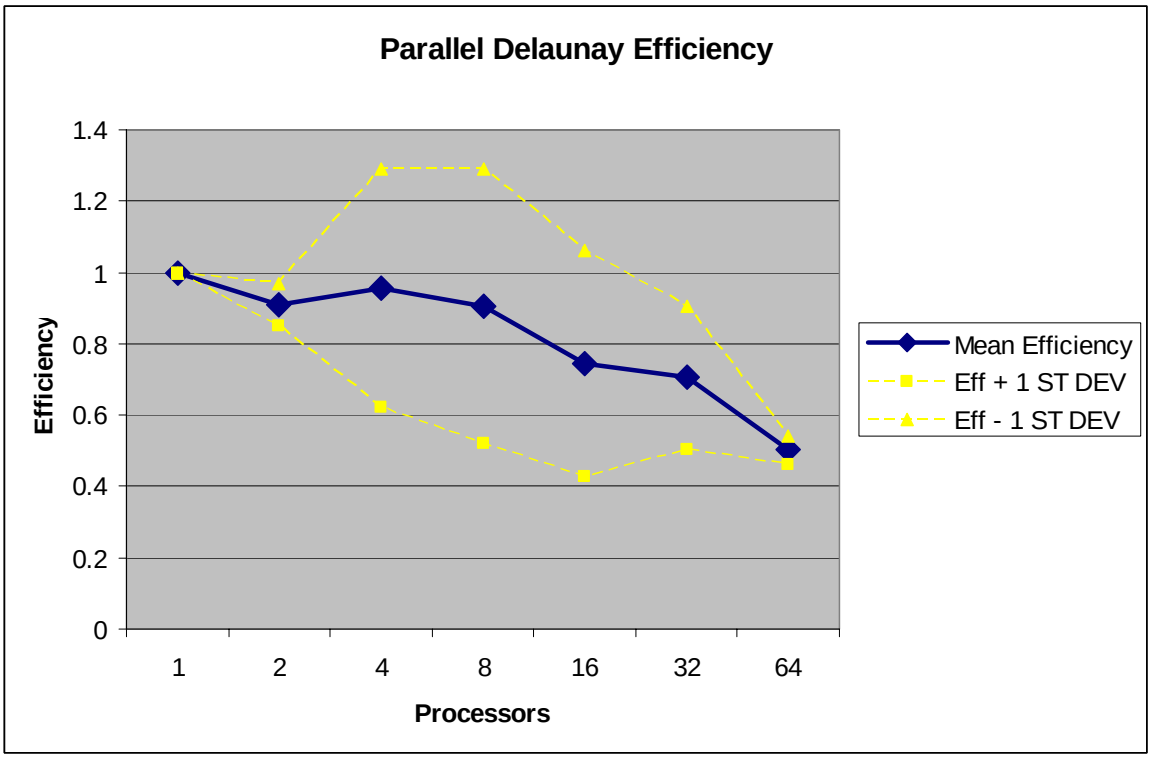


Figure 27. Efficiency for Delaunay Triangulation on Cluster

7. Conclusions and Lessons Learned

This section presents some of the high level conclusions and "lessons learned" from the implementation of a distributed HPC LIDAR-processing system.

- HPC algorithms on dedicated networked resources enable the processing of very large LIDAR datsets. Indeed, datasets which could not be evaluated on one or even two processors (due to insufficient RAM) were successfully evaluated at higher levels of parallelization. While performance begins to degrade for high numbers of processing elements, future testing is required to test scalability and for locating the "sweet spot" where diminishing returns become an issue.

- Serialization of network data transfer is a big issue. To a lesser degree the stitching algorithms affect parallel performance as well. Overall perceived performance (total "wall clock time" for an algorithm to execute) must take all these aspects into consideration. The extreme example of this is observed during off-site use of the system, where computation times take on the order of a few minutes when many processors are used. In contrast, the network bandwidth and time requirements for transmitting results require on the order of 15 minutes or more simply to download results. Thus, a critical element of using such a system is the network configuration. It is advisable to use a captive local on-site network for computations using the system described in this paper.
- Designing this system led to development of an approach to LIDAR processing algorithms that engenders straightforward development of new algorithms for use within the HPC infrastructure. Following a prescribed template, wherein parallel directives (MPI and NOW-specific domain decomposition) and actual algorithmic code are separated into modules, yields very quick turnaround in cross-platform deployment. For example, when the final version of the Delaunay algorithm was completed on the cluster, it was a matter of minutes before it was deployed to the Windows network.
- Conversely to the prior observation, most development time is used in deploying the infrastructure itself rather than the algorithms.
- Use of the COTS United Devices Grid MP software greatly facilitated the IT aspects of creating a network infrastructure to support access to HPC algorithms, such as multiple users, network security, communications between heterogeneous systems, and an established API. Conversely, custom-designed .NET Web Services accomplished the same functionality, at the expense of additional time to develop.
- The parallel algorithms can be enhanced: developing a binary stitching methodology will yield O(log n) performance rather than O(n) performance and also obviate the need for global assembly of results data prior to stitching. Further, full implementations of the kd-tree algorithm discussed in this paper will yield improved performance for the Delaunay algorithm.
- Scalability is a concern. While the Network of Workstations test bed was not sufficient to demonstrate this, the cluster showed diminishing returns with respect to speedup and efficiency after approximately 32 processors.

References

[Desai] Desai, C., SBIR 2002 Phase I: Commercial GIS extension for visualization of large unstructured geospatial data, 2003.

Shepard

[Gordon] W. J. Gordon and J. A. Wixom. *Shepard's method of metric interpolation to bivariate and multivariate interpolation*. Mathematics of Computation, 32(141):253--264, 1978

[Shepard] Shepard, D. A two-dimensional interpolation function for irregularly-spaced data, Proc. 23rd National Conference ACM, ACM, 517-524, 1968

[NCGIA] http://www.ncgia.ucsb.edu/pubs/spherekit/inverse.html

Delaunay

[Blelloch] Blelloch, G.E., et al., "Design and Implementation of a Practical Parallel Delaunay Algorithm," Algorithmica, 24:243-269, 1999

Original Delaunay paper of interest (not available electronically):

[Delaunay] Delaunay, B., 1934. Sur la sphère vide. Bulletin of the Academy of Sciences of the U.S.S.R. Classe des Sciences Mathématiques et Naturelle, Series 7 (6), 793-800.

Kd-tree

[Marner] http://www.rolemaker.dk/nonRoleMaker/uni/algogem/kdtree.htm

Basel algorithm classification

[Burkhart] Burkhart, H. Korn, et al., BACS: Basel Algorithm Classification Scheme", Technical report 93-3, Department for Computer Science, University Basel, Switzerland, March 1993

Architecture//clusters

[Baker] Mark Baker and Rajkumar Buyya, Cluster Computing: The Commodity Supercomputing. *Software - Practice and Experience*, Vol. 1(1) Jan 1999

[Buyya1] Buyya, Rajkumar, High Performance Cluster Computing, Volume 1, Architectures and Systems, Prentice Hall, NJ, 1999.

(Flynn's Taxonomy)

[Duncan] Duncan, Ralph, A Survey of Parallel Computer Architectures", IEEE Computer. February 1990, pp. 5-16

Parallel Algorithms

[Quinn] Quinn, M. J., Parallel Computing. McGraw-Hill, Inc., 1994

[Foster] Foster, I., "Designing and Building Parallel Programs," Addison-Wesley, 1995, http://www.mcs.anl.gov/dbpp

[Buyya2] Buyya, Rajkumar, High Performance Cluster Computing, Volume 2, Programming and Applications, Prentice Hall, NJ, 1999.

MPI & Performance

[Gropp] Gropp, W., et al., "Using MPI: Portable Parallel Programming with the Message-Passing Interface", MIT Press, Cambridge, 1999

Mixed mode // programming

[Cappello] Cappello, F., and Etiemble, E., MPI versus MPI + OpenMP on IBM SP for the NAS Benchmarks, Supercomputing 2000

Appendix A – Numerical Benchmark results

Execution Times for Raster on the Windows NOW at Resolution 5 and 10, 5 sample data sets, 1-4 nodes

	Execution Time (in days hours:minutes:seconds)				
#proc	Smithfield	Brisco	Clearfork	Gauley	Coal River
NOW Rasterization Resolution 5					
1.00	02 20:30:22	00 14:30:25	01 03:35:32	06:36:58	00 22:59:55
2.00	01 17:30:40	00 11:17:11	00 10:37:52	03:18:18	00 10:48:12
3.00	01 06:10:20	00 06:32:03	00 11:30:24	03:09:27	00 08:01:11
4.00	00 20:10:56	00 03:15:00	00 06:47:14	02:20:49	00 05:52:45
NOW Rasterization Resolution 10					
1.00	00 17:02:45	00 03:50:00	00 07:18:17	01:37:01	00 04:38:07
2.00	00 10:48:26	00 02:06:56	00 02:28:20	00:48:18	00 02:37:28
3.00	00 10:33:54	00 01:27:16	00 02:13:04	00:46:10	00 01:55:07
4.00	00 06:16:13	00 01:02:38	00 01:05:00	00:33:52	00 01:24:01

Execution Times for Raster on the Cluster at Resolution 5 and 10, 5 sample data sets, 1-64 nodes

	Execution Time (in seconds)				
#proc	Smithfield	Brisco	Clearfork	Gauley	Coal River
Cluster, Rasterization Resolution 5					
1	84057.40	20106.88	32341.29	7650.71	21202.93
2	49514.85	10950.66	12611.43	3925.67	12512.45
4	29080.98	6245.29	7980.58	2959.32	6886.66
8	15653.34	2844.49	4310.49	1433.22	2263.89
16	5333.35	832.7	1707.52	520.78	598.17
32	1774.23	170.03	480.14	142.86	170.95
64	543.79	92.46	181.74	73.08	92.53
Cluster, Rasterization Resolution 10					
1	21559.84	5009.52	8229.1	1877.39	5329.89
2	12504.51	2625.72	3201.65	966.08	3140.99
4	7383.72	1348.77	2001.47	716.55	1728.53
8	3943.98	478.81	1088.7	336.68	570.02
16	1292.6	147.56	509.64	95.8	132.01
32	316.13	55.22	99.94	37.24	40.93
64	98.98	36.25	35.31	31.73	28.21

Execution Times for Delaunay Triangulation on the Windows NOW, 5 sample data sets, 1-4 nodes

	Execution Time in days hours:minutes:seconds				
#proc	Smithfield	Brisco	Clearfork	Gauley	Coal River
NOW Delaunay Triangulation					
1.00	JOB UNSUCCESSFUL— NOT ENOUGH COMPUTATIONAL RESOURCES (RAM)		00 01:00:23	00:29:09	00 00:36:50
2.00			00 00:26:18	00:14:20	00 00:17:49
3.00			00 00:21:44	00:08:47	00 00:11:19
4.00			00 00:08:58	00:04:22	00 00:11:02

Execution Times for Delaunay Triangulation on the Linux cluster, 5 sample data sets, 1-64 nodes

	Execution Times in minutes:seconds				
#proc	Smithfield	Brisco	Clearfork	Gauley	Coal River
Cluster Dalaunay Triangulation					
1	344:38.804	201:36.401	90:14.375	34:43.260	70:25.205
2	202:02.165	103:29.703	53:50.587	18:22.201	37:32.204
4	122:32.218	52:58.606	26:47.162	11:27.617	11:29.956
8	80:31.391	28:15.283	14:10.404	05:45.274	05:40.543
16	39:58.714	18:46.234	11:27.074	02:57.935	03:25.413
32	20:57.606	08:57.137	03:40.774	02:02.573	02:11.100
64	11:06.707	05:52.400	02:47.216	01:14.02	02:01.865

Appendix B – Varying work unit number for a fixed problem size

NOW Results with set now size, changing number of slices (work units) with set dataset GauleyRiver_BE.las.

Slices	File Upload	Wall Time for equal num processors	Wall Time for 4 CPU's	Download Time	CPU Time
1					
2					
4					
8	00:30.00	16:02.00	32:04.00	00:04.00	3726
16	00:35.00	03:23.00	13:32.00	00:03.00	1780
32	00:40.00	00:36.00	04:48.00	00:03.00	826
64	50:00.00	00:10.00	02:40.00	00:03.00	793
128	01:00.00	00:10.00	05:20.00	00:07.00	1112
256	03:00.00	00:10.00	10:40.00	00:00.00	1966
512	06:00.00	00:10.00	21:20.00	00:00.00	4028