



LAWRENCE  
LIVERMORE  
NATIONAL  
LABORATORY

# Exploiting Data Parallelism in the Image Content Engine

W. M. Miller, J. E. Garlick, G. F. Weinert  
G. M. Abdulla

March 16, 2006

SPIE Defense and Security Symposium  
Kissimmee, FL, United States  
April 17, 2006 through April 21, 2006

## **Disclaimer**

---

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

This work was performed under the auspices of the U.S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under Contract No. W-7405-Eng-48.

# Exploiting Data Parallelism in the Image Content Engine

W. Marcus Miller, Jim E. Garlick, George F. Weinert, Ghaleb M. Abdulla  
Lawrence Livermore National Laboratory, 7000 East Ave, Livermore, CA USA 94551-9970

## ABSTRACT

The *Image Content Engine (ICE)* is a framework of software and underlying mathematical and physical models that enable scientists and analysts to extract features from Terabytes of imagery and search the extracted features for content relevant to their problem domain. The ICE team has developed a set of tools for feature extraction and analysis of image data, primarily based on the image content. The scale and volume of imagery that must be searched presents a formidable computation and data bandwidth challenge, and a search of moderate to large scale imagery quickly becomes intractable without exploiting high degrees of data parallelism in the feature extraction engine. In this paper we describe the software and hardware architecture developed to build a data parallel processing engine for ICE. We discuss our highly tunable parallel process and job scheduling subsystem, remote procedure invocation, parallel I/O strategy, and our experience in running ICE on a 16 node, 32 processing element (CPU) Linux Cluster. We present performance and benchmark results, and describe how we obtain excellent speedup for the imagery searches in our test-bed prototype.

**Keywords:** Image Processing, Parallel Processing, Performance Analysis, High Performance Computing

## 1. INTRODUCTION

The national security, intelligence, and scientific communities are facing a crisis in their inability to analyze massive volumes of remotely sensed imagery. Nearly every aspect of national security and intelligence progress is tied to the acquisition and interpretation of images. With growth in sensor and computing technologies, the capability to generate images is expanding exponentially, but the ability to extract understanding from images is not. There have been major investments over the last decade in new and more capable imaging sensors:

- A new generation of reconnaissance satellites will expand the capacity to collect images by orders of magnitude.
- New platforms, such as UAV's, support collection of real-time video, which can potentially expand the volume of imagery by orders of magnitude.
- New classes of imaging sensors, such as hyperspectral cameras and new synthetic aperture radars, provide data that is much more difficult for human analysts to interpret.
- New imaging systems like the Large Synoptic Survey Telescope (LSST) will produce massive data streams that must be processed in real-time and large-scale databases that will reach multiple Petabytes ( $10^{15}$ ) in size.

Our inability to interpret and analyze ever increasing volumes of imagery data is quickly becoming the Achilles Heel of image processing and analysis. Adding more human analysts will not solve the problem, the tools currently available do not provide sufficient assistance to allow analyst to assimilate massive volumes of image data in a time critical fashion. As the images increase in size and complexity, the tools needed to find patterns and relationships must extend to new types of images, analysis of spectral and spatial temporal patterns, and multiple disparate sources of information.

The *Image Content Engine or ICE*, is a Strategic Initiative at LLNL to develop a framework of software and underlying mathematical and physical models that enable scientists and analysts to search massive volumes of complex imagery for content relevant to their missions and applications. The objective is to develop tools for real-time analysis of streams of images and to search very large database images based on image content. The top-level architecture of this system is illustrated in the Figure 1.1 below.

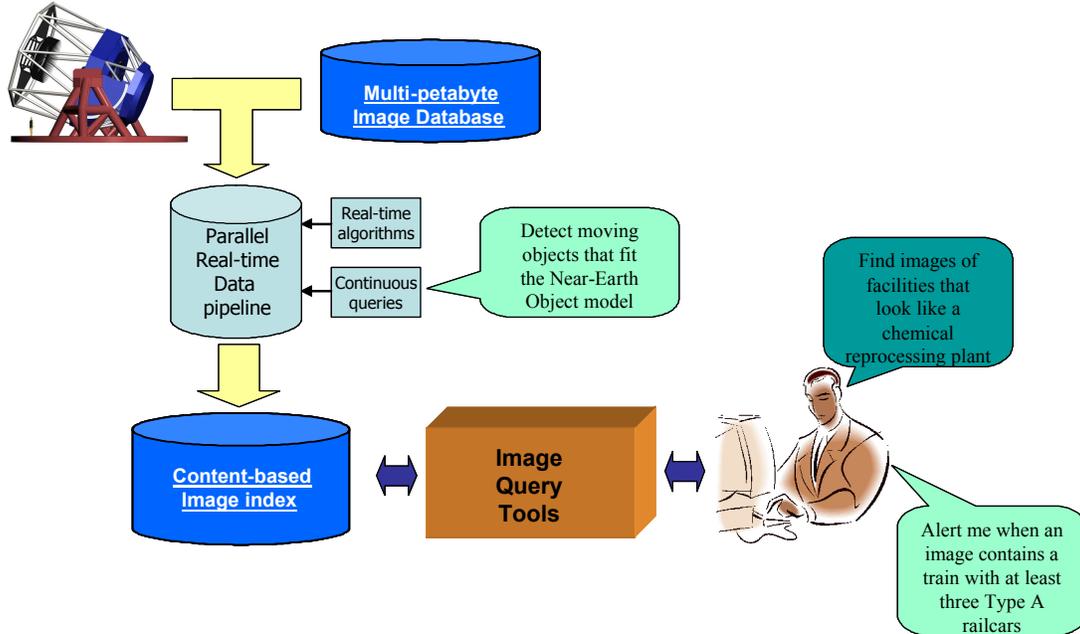


Figure 1.1: Top-level architecture of ICE system.

The basic approach in ICE is to build a hierarchical model of image content that describes image features, objects, and their spatial relationships. This set of image models forms an index of the image database or stream that can be efficiently searched by a scientist or analyst for content relevant to their query.

This paper is organized as follows: in section 2, we describe the ICE processing pipeline, in section 3 we discuss our parallel processing architecture and implementation, in section 4 we present performance results, and in section 5 we offer some thoughts on the next generation ICE pipeline and future implementations.

## 2. THE ICE PROCESSING PIPELINE

The ICE processing pipeline is a collection of software components, each of which performs a single image processing task on a block of image data. Blocks are fixed size, rectangular regions that have been extracted from a larger image. Blocks cover the entire image, and may overlap with one another in order to avoid missing information that may be only partially contained at their edges. The amount of overlap depends on the size of the object under search (larger objects require more overlap). After processing a single block, the ICE components may output a combination of processed image blocks or data describing features of the image block. Some processing components operate on only the data extracted from an image block, producing additional extracted data.

For example, a block processing component can de-speckle an image block or clip and quantize the image data. This sort of processing is useful for later processing stages. An important type of data extraction performed by the pipeline is Gradient Direction Matching (GDM), an algorithm developed at LLNL. In GDM, the edges of a 3D object model are projected into the image block, at some number of different rotations about the Z-axis (typically 63), and a degree of match is calculated for each pixel location (for each orientation) in the image block.

The GDM algorithm uses geo-positioning information contained in the image file in order to project the 3D object model into image space. This file format is typically a National Imagery Transmission Format<sup>1</sup> (NITF) file. This projection takes care of any scaling that may be required. The 3D model of the object describes the object being

searched for in engineering units (i.e., meters), fixing the size of the search object. GDM will match the object at various rotations, but not different sizes.

After calculating the degree of match for each orientation at each pixel, the orientation with the best match at each pixel is recorded along with the degree of similarity to the model at that orientation. This is repeated for each pixel, providing a similarity value and best orientation at each pixel in an image block (a.k.a., a match surface). Maxima on the match surface are then reduced to a list of pixel locations, similarity values, and model orientations. It is these maxima that are stored for later use and used for cueing an analyst's attention to a particular pixel coordinate within the image.

Much of the work done to date on ICE has concentrated on the GDM algorithm, and the rest of this paper describes results of feature extraction using the GDM algorithm. Although the GDM algorithm outlined above calculates a value at each pixel location, the calculation is not carried out directly. The algorithm may be reformulated in the Fourier domain, and implemented much more efficiently than the direct method. As implemented in ICE, the GDM algorithm is heavily dependent on an efficient implementation of the 2D FFT. Therefore the dimensions of the image block have been fixed to a convenient power of 2. In this case, 512, and each block is fixed by the implementation at 512x512 pixels.

In some cases, the search object may be a significant fraction of 512 pixels wide in the image. In those cases, in order to avoid excessive overlap in the image blocks and to reduce the total processing time, image blocks larger than 512x512 are used and then scaled down to 512x512 blocks before the GDM algorithm is run. For example, an 811x811 block may have been chosen and then scaled by approximately 0.63 to produce a 512x512 block suitable for processing by the GDM algorithm, thus reducing the total number of blocks processed as well as the total amount of overlap (further reducing the total number of blocks needed).

In the original, Phase I implementation of ICE, each image block was processed sequentially on a single processor. Since most of the processing for a given block is completely independent of the processing done on other blocks, ICE presents an ideal candidate for parallelization.

### **3. PARALLEL PROCESSING IN ICE**

During the first phase of the ICE project, we developed analysis tools based on segmenting images into multiple blocks, and then sequentially searching each block for relevant content. The processing time for the initial phase I implementation was extremely protracted however. Soon after the initial prototype implementation was running, the team began developing a processing pipeline based on exploiting the significant underlying data parallelism in the ICE imagery processing. The goal of this phase II work was to obtain at least a ten fold reduction in processing times over the phase I processing pipeline. For execution of the parallel ICE pipeline code, we assembled a 16 node, dual 1.7GHz Intel Xeon, 32 bit CPU cluster (32 total processors). Each cluster node supports 2GB of DRAM memory (small by 2006 standards), with a Gigabit Ethernet (1000Mb/s) interconnect. Linux Fedora Code 3 was used as the base operating system because of the plethora of both open source and in-house (LLNL) tools available for cluster based computing. Disk storage for the cluster was provided by four network attached BlueArc fileservers storage appliances that export NFS (v3) mounts to all the cluster nodes, providing a common file system to all the cluster nodes. Total storage available for the prototype pipeline processing was 1.6TB. Figure 3.1 depicts the hardware topology used in the parallel prototype systems. This hardware topology is readily supported by classic distributed memory message passing protocols such as Parallel Virtual Machine (PVM)<sup>2</sup>, and the Message Passing Interface (MPI)<sup>3</sup> protocols.

The implementation of the phase I version of the ICE pipeline was based on a hybrid source language approach. C/C++ codes are used to implement fundamental processing algorithms (FFT, etc.) and IDL<sup>4</sup> (RSI Interactive Data Language) is used as a scripting control and “glue” code for the pipeline. The actual block processing algorithms, including GDM, were implemented as C++ libraries that were packaged as executables and executed as separate processes by the IDL code. This approach was taken for both speed and portability and the ICE code currently runs on three distinct platforms: Windows, MacOS, and Linux.

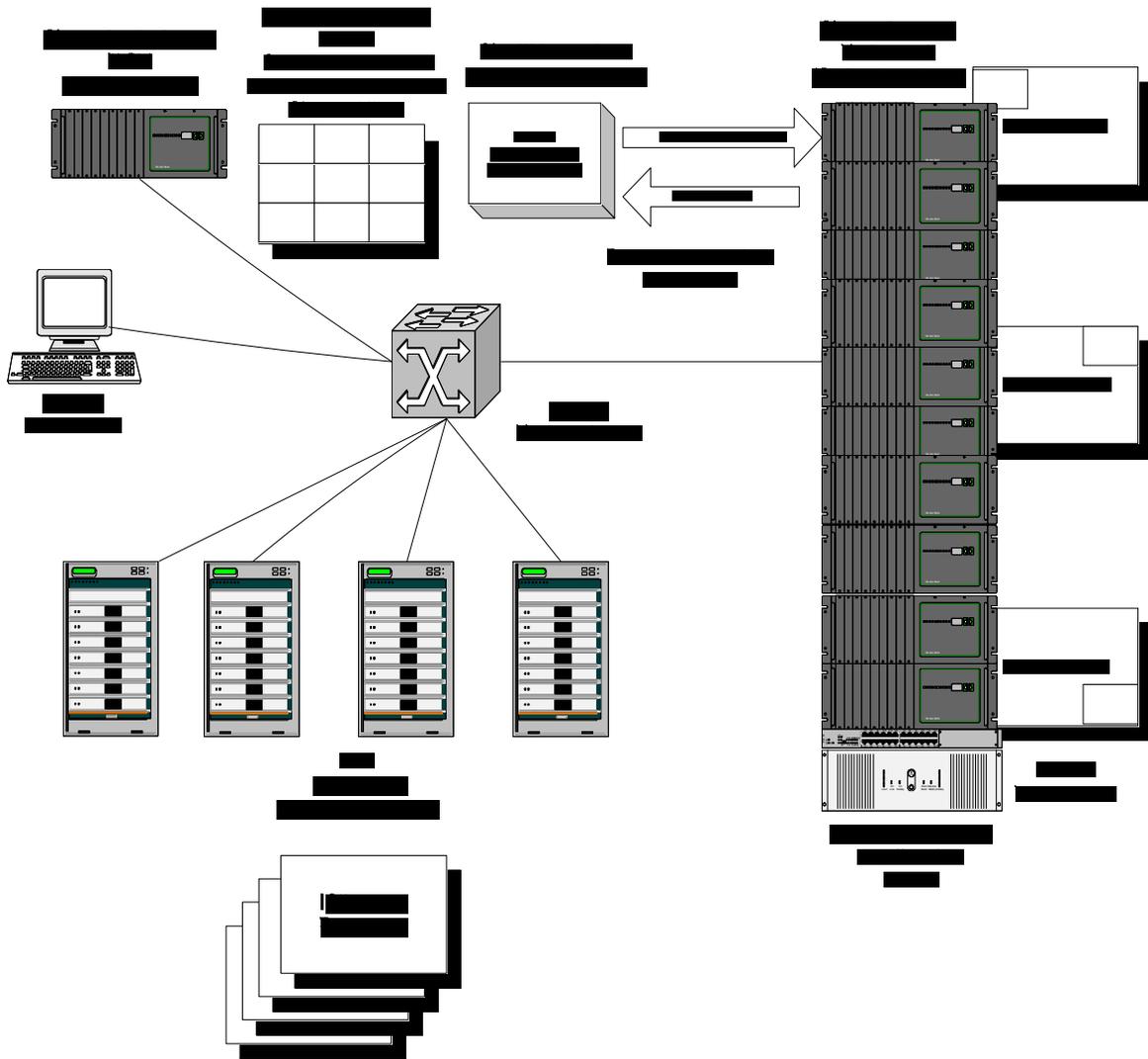


Figure 3.1: ICE Pipeline Hardware

The natural partition for a data parallel implementation in the phase I ICE code is the parallel processing of image blocks, i.e. reduce all image blocks concurrently, and then wait on a semaphore synchronization barrier until all blocks are complete. The distribution of blocks for processing by the GDM is done in IDL code, which unfortunately provides no native support for parallel execution<sup>5</sup>.

It became clear early in the software analysis for phase II development that traditional distributed memory programming tools would not be useful in the context of a parallel ICE prototype. This was primarily driven by the desire to maintain a common code base between the phase I, sequential code and the phase II, parallel code, and the considerable effort required to redevelop much of the code using a specific message passing interface. Second, the underlying image processing readily supports a highly data-parallel, independent processing model, and requires very little inter-process communication between processing image blocks. The use of a fine grain message passing protocol in this context would add overhead, and contribute little to our performance goals. There were two basic alternatives for exploiting parallelism in the image processing, process all search objects in parallel (object based parallelism, one complete image per processing unit), or process all images in parallel (image based parallelism, spread image processing across multiple processors). We decided to process images in parallel, because this yields the highest throughput on object detection per image, and the lowest latency in generating detection results.

Figure 3.2 illustrates the simplified processing loop pseudo code for submitting blocks in the Phase I implementation. The actual code is implemented in IDL and includes some preprocessing and post processing steps. The order of block submission and completion is strictly ordered, and synchronous, i.e. block  $k$  completes before block  $k+1$  is submitted.

Figure 3.2: Serial Block Processing Loop
<pre> ; ; Pseudo-code structure of serial version of Image Block ; Processing loop. ; ... <b>for</b> block=1,N <b>do begin</b>     ...     ice_pipeline(block) ; This call blocks until completed     ... <b>end</b> ... </pre>

In the parallel version of the code, we implement block processing using two asynchronous split phased loops. The first loop submits blocks to the cluster for processing, and the second processes the completed results. Both loops are non-blocking and may process blocks out of order. In the second loop we only wait on results in the event none are available or all the CPU resources are busy. This logic serves as a simple, but effective throttling mechanism to prevent the cluster from becoming saturated with block submissions. Figure 3.3 illustrates the simplified block processing for the parallel code. Although the two loops are implemented in IDL, they are none the less explicitly executed in parallel because the `ice_pipeline_submit()` and `ice_pipeline_wait()` APIs are non-blocking, and execute asynchronous to the loop bodies. These two API calls were implemented in C++ using the IDL Dynamic Link Library (DLL) extension for IDL function calls in the Unix environment. To manage the remote execution of block processing jobs on the cluster we use the LLNL developed, open source, Simple Linux Utility for Resource Management (SLURM)<sup>6</sup>. SLURM allows jobs (processes) to be executed on cluster nodes in a variety of modes, including a single job executing on multiple nodes, multiple jobs on a single node, or multiple jobs running on multiple nodes. SLURM maintains a pool of cluster resources (memory, CPU, disk space, etc.) and a queue of executing jobs, and supports blocking, non-blocking and batch execution modes. The two primary SLURM utilities are SRUN, used for submitting new jobs, and SQUEUE, used to retrieve information on executing jobs.

Figure 3.3: Parallel Block Processing Loop
<pre> ; Pseudo-code structure of parallel version of Image Block ; Processing loop. ; maxCPUs=32 completed=0 block=1 <b>while</b> (completed &lt; N) <b>do begin</b> ; submit a new block, non-blocking call.     <b>if</b> block &lt;= N <b>begin</b>         ...         ice_pipeline_submit(block)         block++         ...     <b>end</b> ; get block results when available     <b>if</b> completed &lt; block <b>do begin</b>         waitflag = (block eq 1) <b>or</b>                     (block - completed eq maxCPUs) ; return next completed block, non-blocking call if ; waitflag is not set (0). The code will wait for </pre>



(RPC) and distributed computing across heterogeneous platforms, including CORBA<sup>8</sup>, SOAP<sup>9</sup>, and ZeroC ICE<sup>10</sup>. Our original design plan was to make use of one of these facilities. Since we need to make remote procedure calls from IDL to IDL, we needed an RPC mechanism that would support IDL marshalling and un-marshalling of function and procedure parameters. To our knowledge, none of the existing distributed computing tools support this. As a consequence, we developed our own specialized RPC mechanism for calling remote IDL procedures. Since RSI IDL is implemented using a run time interpreter (as opposed to direct compilation to the target machine code), making a remote procedure call in IDL requires one to execute the interpreter on the remote CPU, that in turn executes the IDL procedure. Figure 3.4 is an illustration of the simplified flow control used in the parallel ICE pipeline. The diagram shows there are multiple copies of the processing pipeline executing at any point in time.

A small caveat in the RPC processing is that IDL procedure call semantics are “pass-by-value-result”, hence the marshalling code must copy call parameters to the remote procedure, and then copy all parameters/results from the remote procedure to the caller once the remote call is completed. This does introduce a small performance penalty in the RPC processing. Refer to Table 3.1 for the list of our RPC primitives and IDL interface to SLURM.

RPC Primitives for IDL	Description
ice_pipeline_submit, args...	RPC call to ice_pipeline(). Calls ice_pipeline_marshal() to copy args. Returns immediately.
s = ice_pipeline_wait(args...)	Calls psched_wait(1,0). If no outstanding ice_pipeline() calls have completed, return -1 immediately. Otherwise, calls ice_pipeline_unmarshal() to copy file to args, cleans up, and returns ice_pipeline()'s return value.
s = ice_pipeline_marshal(file, retVal, args...)	Marshal ice_pipeline() args to file. Opens file and calls a series of put_vars for the args. Always returns 0.
s = ice_pipeline_unmarshal(file, retVal, args...)	Marshal ice_pipeline() args from file. Opens file, calls a series of get_vars for the args. Always returns 0.
SLURM Interface to IDL	Description
dir = psched_mkdir()	Set up working directory for next psched_submit() cmd. Returns working directory path or NULL on failure.
jobID = psched_submit(cmd, jobname)	Place cmd in the slurm queue under the name jobname. Non-blocking. Returns jobID or -1 on failure.
dir = psched_wait(nonBlockFlag, jobID)	Wait for jobID to complete (jobID=0 implies any job). Returns job's working dir, or NULL on failure or if nonBlockFlag and job is not complete.

Table 3.1: RPC and SLURM API Primitives for ICE

#### 4. PERFORMANCE RESULTS

Our primary performance goal in Phase II of the ICE pipeline development was to achieve at least a factor of ten Speedup in object detections over the Phase I pipeline code. We were able to exceed this goal with a combination of technologies, including:

- Parallelizing the code so that it would have the greatest performance impact - the processing of thousands of image blocks within the inner loop of the pipeline.
- Implementing the parallel code so there are minimal dependencies between concurrently processing blocks, yielding code that is highly scalable.
- Instrumented the parallel code so that the submission of image blocks is asynchronous to the completion of image blocks, and a single synchronization barrier is used at the end of each image that is processed. This allows us to dynamically throttle the workload on the cluster, and control overall resource utilization.
- File I/O is distributed across multiple, independent storage channels and storage devices. In the ICE prototype we used four independent 1Gb/s storage appliances and multiple Firewire 800 (IEEE-1394b) attached storage units.
- Our job allocation and resource allocation broker (SLURM) supports single CPU allocation of jobs on multiple CPU node configurations, allowing us to allocate all available cluster CPUs concurrently.
- Our RPC parameter passing mechanism is distributed. The calling parameters for each image block are copied by a remote startup stub prior to invoking the remote procedure. The same technique is used to copy the results back to the caller. This mechanism makes our RPC highly scalable.

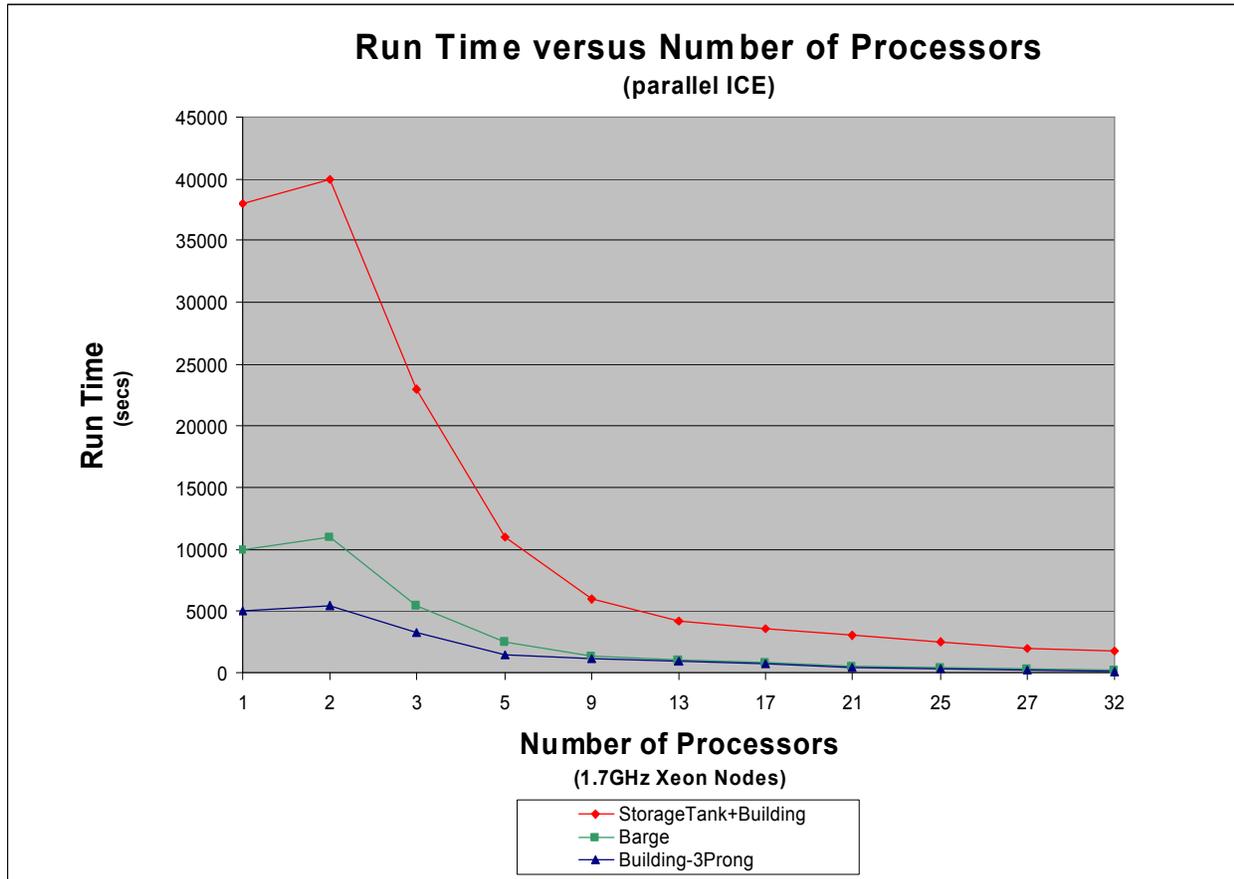
We successfully executed all the prototype codes developed in the Phase I pipeline using the parallel code implementation. We only injected a minor change to the original pipeline setup, allowing us to specify the total number of processor resources available for allocation, and the ability to disable rendering of processed image block thumbnails since this produced a bottleneck in the parallel processing. See Table 4.1 for a subset of the search models and parameters we used in benchmarking the parallel pipeline code. The images searched in these benchmarks were stored in National Imagery Transmission Format (NITF). The NITF file layout is not particularly well suited for random I/O processing or the extraction of random image blocks, so we pay a small penalty in file I/O overhead in reading each block of pixels using the NITF format. These search models range from relatively large images, medium pixel resolution, to medium size images, and high pixel resolution.

Search Model	Description	Image Blocks Processed	Image Size (Bytes/Pixels)	Pixels per Block/Mean GSD
Storage Tank + Building	Broad area search of two different objects, both relatively large	1640 blocks – Storage Tank 2162 blocks – Building	1.5GB/ 27552 x 27004	811x811 / 30.6"
Barge	Broad area search looking for small Barges	1062 blocks	452MB/ 27552 x 8140	512x512 / 24.1"
Three Prong Building	Localized search looking for a distinctive building type	529 blocks	536MB/ 16384 x 16384	875x897 / 23.6"

Table 4.1: Search Model Parameters

Graph 4.1 plots the run time (wall clock elapsed time) for processing each of the search models relative to the number of processors used in the run. An interesting observation is that the run times actually increases when two processors are used. The reason for this is straight forward, in the single processor case there are no parallel processing overheads - more complex inner loop processing, marshalling of parameters, RPC calls, and the end of loop barrier. Using two processors, there simply is not enough concurrency to amortize the overheads associated with running the parallel code. The situation looks dramatically different with three or more processors, and the run times asymptotically approach

their Amdahl's Limit<sup>11</sup>. In our case, much of the inner loop block processing control code, end of loop synchronization barrier, and some file I/O are executed sequentially, and this does impact the overall Speedup.

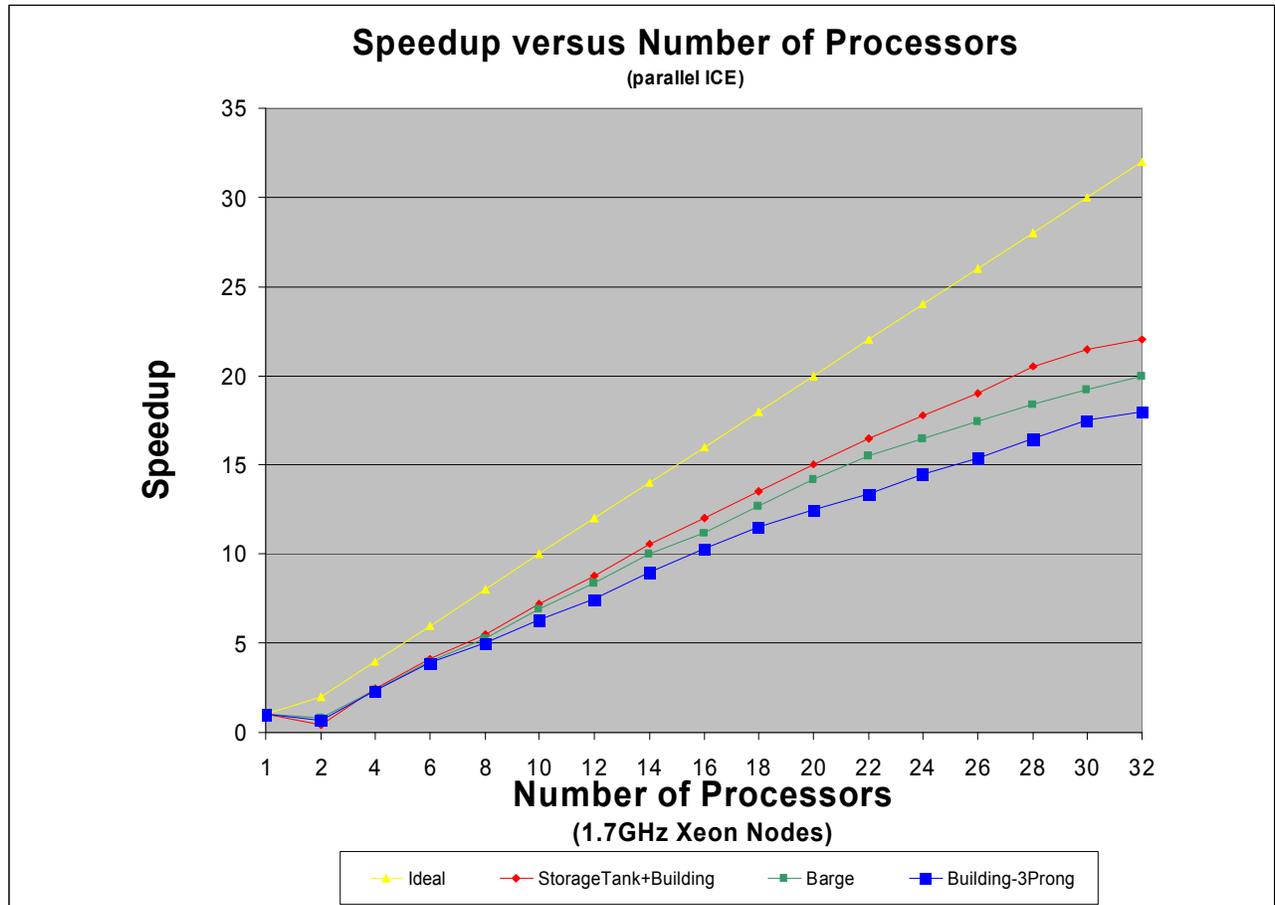


Graph 4.1: Runtime versus Processor Count

Graph 4.2 is a plot comparing the speedup of the ICE pipeline execution times that were measured for the benchmark models versus the number of processors, for 1 to 32 CPUs. Speedup is the relative measure of performance gains using multiple resources over the performance on a single resource. If  $S$  represents the Speedup,  $T_1$  the execution time on a single resource, and  $T_p$  the execution time on  $p$  resources, then

$$S = \frac{T_1}{T_p}, \text{ and in the ideal case, } T_p = \frac{T_1}{p}, \text{ hence } S = p$$

The diagonal yellow line in the plot represents this ideal, or optimal Speedup that would be obtained if the code exhibited “perfect” scaling properties. In all the benchmarks we were able to exceed our performance objective of a ten fold reduction in execution times, and in most cases we achieved close to a 20x reduction in pipeline run times using 32 processors on a 16 node cluster.

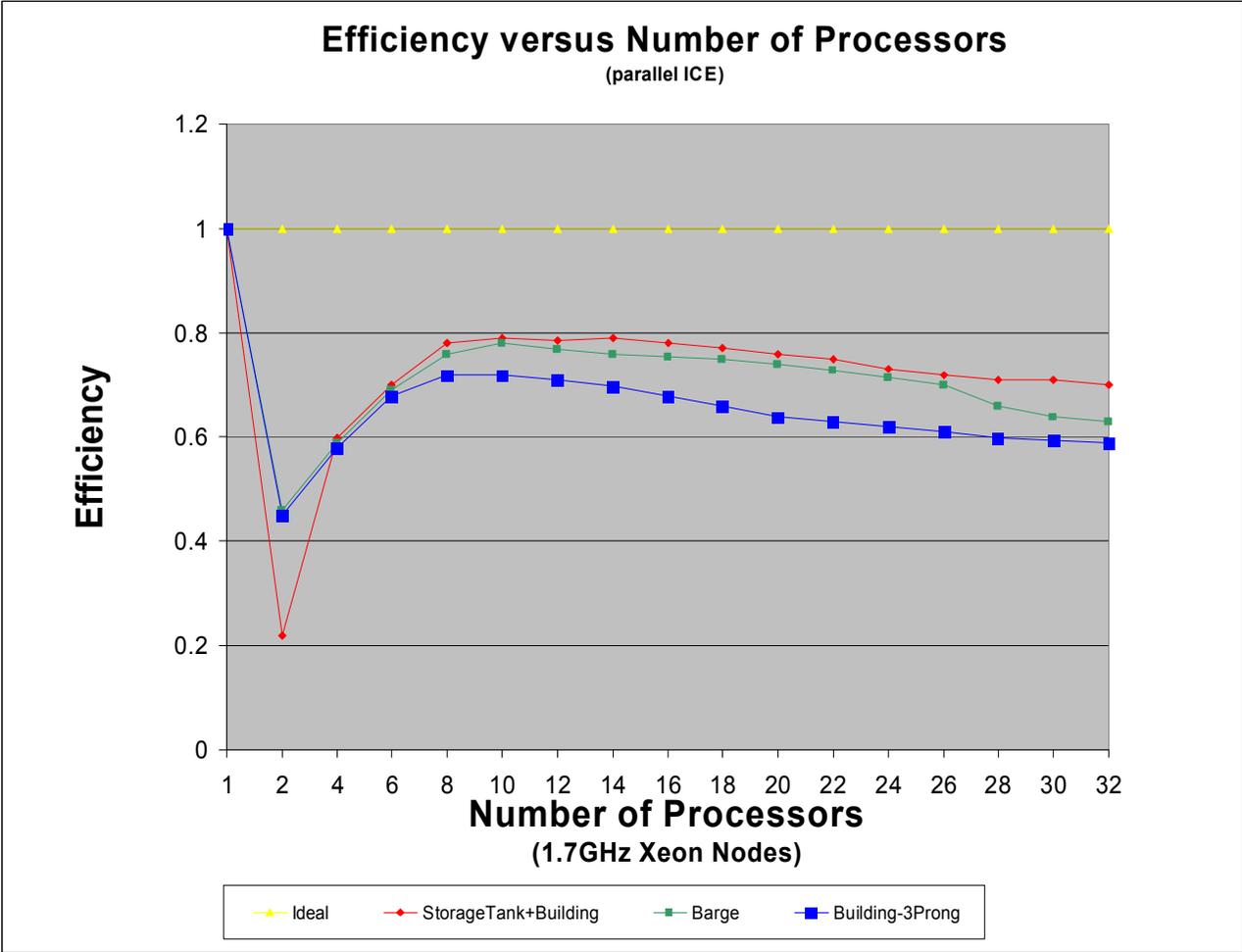


Graph 4.2: Speedup versus Processor Count

Graph 4.3 plots the efficiency of the parallel execution relative to the number of processors employed. Efficiency is a measure of how near to the ideal Speedup a given implementation comes. It is the ratio of the measured or actual Speedup to the ideal Speedup,

$$E = \frac{S}{p}, \text{ so in the optimal case, } E = 1$$

The horizontal yellow line in Graph 4.3 corresponds to the optimal Efficiency. We note that there is significant degradation in Efficiency for 2 to 8 processors in our parallel execution. Again, this is primarily caused by the relatively high overhead associated with running the pipeline in parallel relative to the number of processors participating in the computation. On average Efficiency of the ICE pipeline peaked at 10-14 processors (75-80% of ideal). There are a number of factors that prevent us from obtaining higher Efficiency, including network communication latency, distributed file system latencies, I/O contention on writing temporary and final object search results, RPC and SLURM job startup overheads, and imperfect scalability of the inner loop image block processing. We also note that resource constraints including memory pressure, thread and process over subscription, and disk allocation and storage space were not contributing factors.



Graph 4.3: Efficiency versus Processor Count

Efficiency is slightly degraded when running on processors counts above 18 CPUs. We believe the primary cause of this to be lack of I/O scalability for both the cluster communication network and the disk storage network. The cluster interconnect is 1000Mb/s Ethernet, and the storage network is based on a combination of Firewire 800 (800Mb/s) devices, 1000Mb/s Ethernet storage appliances, and node local EIDE storage. In the case of the cluster interconnect, both RPC traffic, and disk I/O for the NFS mounted file systems are carried over this network. Because we are using an 8 to 1 ratio of processors to storage units, the pipeline processing creates some hot spot contention on the network, and the disks themselves where multiple concurrent writes to the same physical storage units occur. For a fixed number of storage units, the situation is exacerbated as more processors are added to the computing pool.

To summarize our performance results, Table 4.2 compares the parallel and serial run times along with the measured Speedup using a single pipeline control processor and a 31 processor compute pool.

Search Model	Speedup on 31 processors	Serial Run Time (hr:min:sec)	Parallel Run Times (min:sec)
Storage Tank and Building	21.87	10:46:04	29:32
Barge	19.75	02:48:13	08:31
Three Prong Building	17.74	01:28:43	05:00

Table 4.2: Performance Summary

## 5. FUTURE DIRECTIONS

The first attempt to develop a parallel version of the ICE pipeline resulted in speedup and performance improvements on the order of 20x over the phase I sequential prototype running on a 32 processor Linux cluster. To reach the next level of performance improvement (100x), we anticipate a number of Hardware and Software changes will be required:

- Hardware – to obtain speedup in the 100x range will require a considerably larger cluster or grid, at least on the order of 128 processors. We need to investigate the benefit of moving to a 64 bit platform, running 64 bit code in combination with a 64 bit (LARGE FILE I/O) file system.
- File I/O – to scale to 100x, our code will need to be even more scalable than the current version. One of the primary bottlenecks in the current code is the disk and file I/O. We currently are using an 8:1 ratio of processor to disk storage. We will need to decrease this to something like 4:1 to reduce disk and file hot spot contention.
- Code Scalability – the current code will not scale well beyond 64 processors, and the overall code efficiency will fall to below 50%. The reason for this is primarily due to two architectural features. The first is the inner block processing loop. This IDL loop continuously calls two asynchronous remote procedures. This single loop is the primary serialization point in the parallel code, and is the fundamental obstacle in obtaining further scalability. Future versions of the code will need to employ a multi-threaded approach in scheduling image blocks. We may need to move away from the IDL code base for this processing, as the overhead associated with the IDL interpreter also trends to serialize this part of the code. Secondly, we will need to re-implement our RPC mechanism. The current version uses the distributed file system to pass parameters and return results. This places an additional burden on file system I/O that is already over subscribed with computational I/O. We will investigate using a commercial product that has been optimized for high performance computing using remote procedure call that is network based only.
- Exploiting further parallelism in the ICE pipeline – the existing pipeline has been designed and instrumented to run in parallel at the image block level. We can also support additional levels of parallel processing by running each object search in parallel (objects are not detected in parallel using the current code). This will also require higher levels of concurrency in disk and file I/O.

We have obtained good performance improvements with the current pipeline prototype, and have analyzed what must be done to reach the next performance milestone in pipeline processing. A new pipeline prototype based on these changes should allow us to easily deploy code that scales to 100-200 processing elements, while maintaining code efficiency that is commensurate with the current levels. To obtain scale-up to the 1000 processor level will require a complete re-architecture of the ICE pipeline code, possibly exploiting data level parallelism, thread level parallelism, and fine grain message based (MPI) parallelism.

## REFERENCES

---

<sup>1</sup> A description of many common graphics file formats can be found here: <http://www.faqs.org/faqs/graphics/fileformats-faq/part3/section-91.html>

<sup>2</sup> For an overview of, PVM see: <http://www.csm.ornl.gov/pvm/>

<sup>3</sup> For an overview of MPI, see: <http://www-unix.mcs.anl.gov/mpi/>

<sup>4</sup> Details of the RSI IDL product can be found at: <http://www.rsinc.com/>

<sup>5</sup> RSI IDL 6.2 supports the use of threads via an implicit thread pool. This pool of threads is not explicitly available to IDL application developers however.

---

<sup>6</sup> An overview and download of the LLNL Open Source resource manager can be found at:  
<http://www.llnl.gov/linux/slurm/>

<sup>7</sup> For a tutorial on P-Threads, see: <http://www.llnl.gov/computing/tutorials/pthreads/>

<sup>8</sup> For an overview of Corba, see: <http://www.cs.wustl.edu/~schmidt/corba-overview.html>

<sup>9</sup> For an overview of SOAP, see: <http://www.w3schools.com/soap/default.asp>

<sup>10</sup> Papers on Zero-C ICE can be found here: <http://www.zeroc.com/>

<sup>11</sup> Amhdal's Law states that the serial portion of any executing code limits the overall speedup available for parallel execution. If  $S$  is the speedup, and  $f$  is the fraction of the code that is performed sequentially, and  $p$  is the degree of parallelism used in the computation, then  $S = 1 / (f + (1-f)/p)$ . As  $p$  approaches infinity,  $S$  approaches  $1/f$ . Hence for even small values of  $f$ , the speedup can be dramatically impacted for large numbers of processors ( $p$ ).