Gigapixel-size Real-time Interactive Image Processing with Parallel Computers

Donald R. Jones, Elizabeth R. Jurrus, Brian D. Moon, and Kenneth A. Perrine
Pacific Northwest National Laboratory*

Abstract

The Parallel Computational Environment for Imaging Science, PiCEIS, is an image processing package designed for efficient execution on massively parallel computers. Through effective use of the aggregate resources of such computers, PiCEIS enables much larger and more accurate production processing using existing off the shelf hardware. Goals of PiCEIS are to decrease the difficulty of writing scalable parallel programs, reduce the time to add new functionalities, and provide for real-time interactive image processing. In part this is accomplished by the PiCEIS architecture, its ability to easily add additional modules, and the use of a shared-memory programming model based upon one-sided access to distributed shared memory. In this paper, we briefly describe the PiCEIS architecture and our shared memory programming tools and examine some typical techniques and algorithms. Initial image-processing performance testing is encouraging—for very large image files, processing time is less than 10 seconds.

1. Introduction

With the advent of high-resolution and multi-spectra cameras from sources such as satellites, nuclear magnetic resonance, and the Transmission Electron Microscope (TEM), large images are now pervasive. Images from satellite image companies such as IKONOS, Digital Imagery, Space Digital Imaging, and SPOT are commercially available. With the availability of low-cost commercial parallel computers, it is possible to process many large images quickly and effectively. However, some of the fundamental problems and requirements for improving processing of satellite images remain unchanged since the 1960s. For example, image size is determined by spatial and spectral resolution, radiometric resolution, network and memory bandwidth constraints, and display output requirements [8]. Also, similar issues concerning bandwidth for moving and processing massive amounts of pixels exist.

As more advanced satellites are put into place[†], timely post-processing beyond workstations and small shared-memory computers becomes imperative. Although high-performance serial processing is becoming pervasive with availability of software [1, 7, 17] and the relatively low cost of high-end processors, such as Pentium-4, Athlon, and Itanium-2, these systems are not sufficient to process very large images. In addition, specialized "custom" parallel image processing hardware solutions which can handle large images are unable to be used for multiple applications without significant hardware and software development costs. Parallel computers consisting of clusters of off-the-shelf hardware may be more cost-effective, but the development of software is still in its infancy for large images that are not perfectly parallel.

We describe an efficient and portable parallel programming methodology and associated implementations for processing images using commercial distributed shared-memory massively parallel computers: an IBM SP and a Linux cluster. We apply a scalable parallel computing approach that is based on a portable library, using shared memory techniques, for in-core and out-of-core computations, visualization, and parallel I/O. During processing and visualization, the image is fully distributed to the remote processing units in contrast to many of the current master-slave models in image processing.

In general, we apply domain-decomposition methodologies and software used in solving partial differential equations, computational chemistry [6], and parallel rendering. This permits an easy interface for processing tiles of images with user-specified routines as well as using existing libraries of serial software such as OpenCV [11], Python Imaging Library [15], and Java toolkits. In addition, for a specific processor, optimized low-level libraries supplied by the processor vendors, such as Intel's custom image processing library or IBM's ESSL library, are used.

Data rearrangement and movement can be time-consuming to program and cause reliability concerns regarding message passing. This can be avoided by using

* Address: W. R. Wiley Environmental Molecular Sciences Laboratory, PNNL, P.O. Box 999, Richland, WA 99352. For further information, e-mail Donald Jones at dr.jones@pnl.gov.

† OrbView-4, from Orbital Imaging Corporation, with 200+ bands each with 8 to 11 bits per pixel with a spectral range of 450-2500 nm at 8 m spatial resolution. Although this satellite did not survive launch it sets a point for future developments.

the Global Arrays shared memory model [10] that has a direct interface to parallel dense and sparse linear and eigensystem solvers (ScaLAPACK [3], PeIGS [4], PETSc [2]), global minimization software (TOM), and integral transformations (e.g. FFTs). Complex global communication patterns can be implemented easily in Global Arrays. Users can also add their customized image-processing routines without knowledge of data distribution, memory hierarchies, or programming models. Load-balancing schemes can also be easily adapted to different scenarios because the complexity of message passing is moved to the library level and is completely handled by Global Arrays. Parallel input and output and out-of-core processing that is coupled with the in-core parallel computation is also available. By applying these methodologies, we have seen a significant acceleration of image processing. Tasks that previously would have taken several minutes to hours to process can now run in a few seconds to near real-time. Depending on the image processing routines and the size of the image, applications often can run interactively. As processing takes place, the user sees updated images. Users can display the output either to their monitors or to the IBM Scalable Graphics Engine attached to the communication switch.

One advantage of moving the complicated indexing task to Global Arrays is that clusters of heterogeneous nodes, with different amounts of memory and processing power, can be used efficiently by allocating different sizes of data to each node. No special algorithm or code is required. We can perform dynamic load balancing by tracking the computational time over key loops over a number of iterations.

Experimental results are presented to demonstrate the parallel acceleration obtained on the Pacific Northwest National Laboratory (PNNL), Molecular Science Computational Facility [9] (MSCF) computation resources. The images can be displayed on either an X11 Display or an IBM Scalable Graphics Engine with four network adapters connected to a high-resolution IBM T221 monitor with 3840 by 2400 pixels.

2. Scalability issues

PiCEIS has a modular design using many concepts from object-oriented programming (OOP). The advantage of modular design is that it allows orderly and logical access to data, independent of why and when a given module requires the data. In addition, it permits considerable flexibility in the manipulation and distribution of data on shared memory, distributed memory, and massively parallel hardware architectures, which is needed in a shared-memory approach to parallel computation.

One of PiCEIS's design goals is to enable effective use of all of the resources of massively parallel architectures, including CPUs, memory, disk, and network. With this goal, distributing the data across all of the nodes becomes necessary. To facilitate portability while avoiding low-level message passing to manage distributed data, PiCEIS uses a shared-memory programming model that explicitly recognizes the shared-memory characteristics of modern sequential and parallel computers. Just as a workstation has various levels of memory (e.g., registers, multiple levels of cache, main memory, and disk) with varying sizes and access speed, remote memory may be regarded and used as another level in the memory hierarchy. The programmer must be aware of this extra level of memory access when designing parallel algorithms in any code to obtain efficient and scalable code. A critical aspect of this shared-memory model is one-sided access to shared data. The ability of enabling processes to access any shared data at any time without the explicit involvement of other process greatly simplifies the writing of parallel programs and greatly increases scalability through increased asynchronous execution.

The Global Arrays tools provides much of the underlying support for the parallelism in PiCEIS. These tools include the Memory Allocator for access to local memory; Global Arrays to provide portable globally addressable shared-memory programming on distributed shared-memory computers; the Aggregate Remote Memory Copy Interface (ARMCI) to provide general-purpose, portable, and efficient remote memory copy operations (one-sided communication) optimized for non-contiguous (strided, scatter/gather, I/O vector) data transfer; and the Parallel I/O (ParIO) and Disk Resident Array (DRA) tool to extend the shared memory model to disk. Global Arrays and ARMCI interoperate with message passing interface (MPI) [16] for message passing and can be considered an extension of MPI-2 [5].

Global Arrays supports the shared-memory model by allowing nodes to share arrays between processes as if the memory is physically shared while providing separate mechanisms to access shared data (Fig. 1). It provides the programmer with simple routines to access and manipulate data in the shared memory using one-sided communication that allows for the overlap of computation and communication. However, the programmer must still be aware that access to shared data will be slower than access to local data. Algorithm optimization should be performed with this knowledge. ARMCI is implemented in a platform-specific manner using any and all available mechanisms to achieve the best possible performance for one-sided communication for noncontiguous data transfers. It uses hybrid communication protocols such as active messages, threads, local memory copies, and remote memory copies to minimize the amount of communication between processors (Fig. 2).

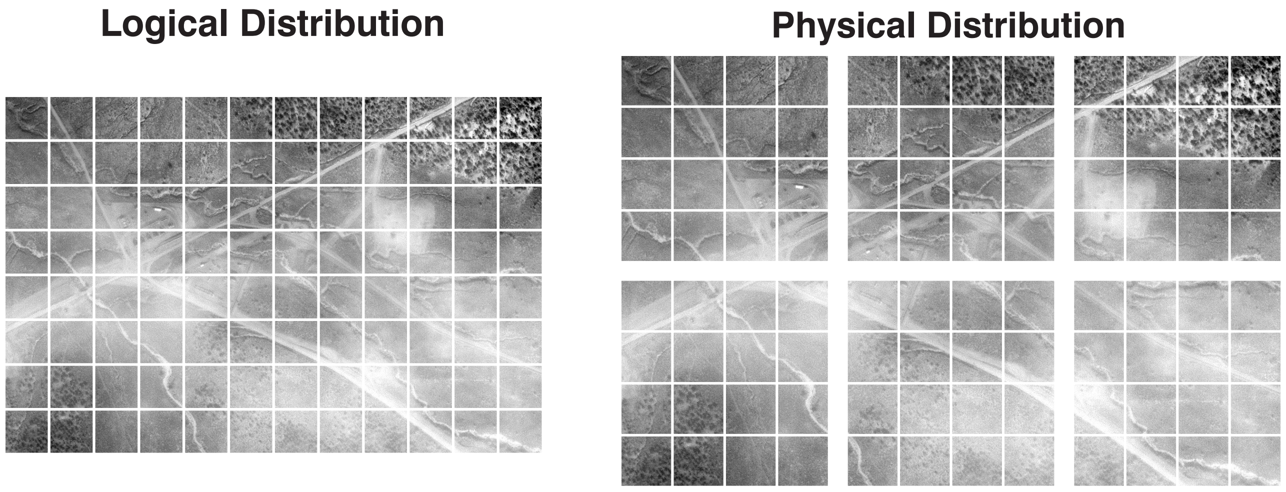

Figure 1: Global Arrays presents a global logical view of memory and the actual physical layout.

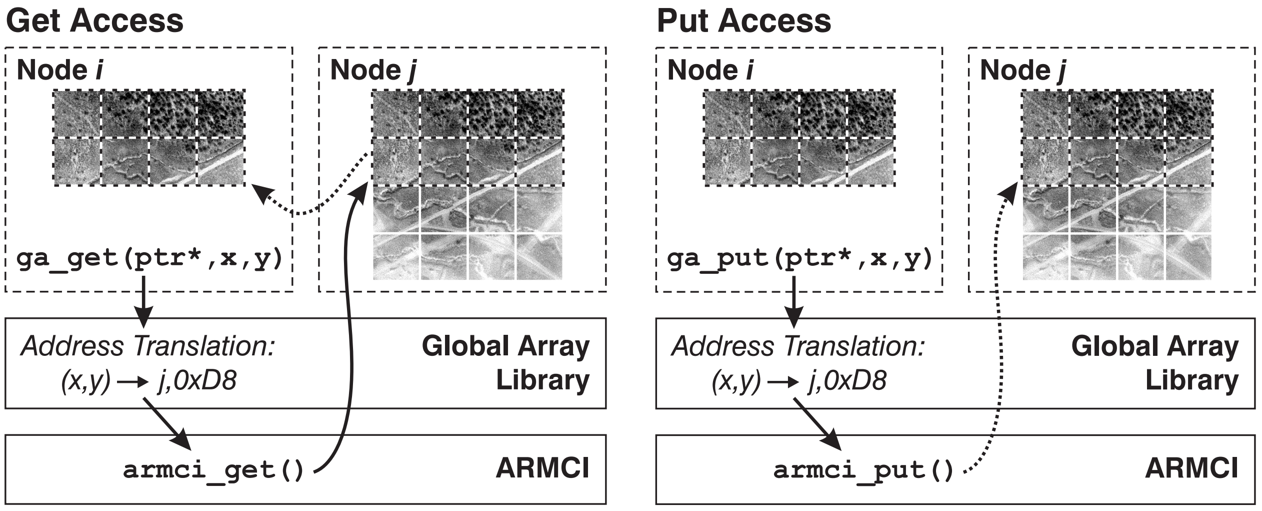

Figure 2: Global Arrays provides remote memory access on distributed shared memory computers.

The ParIO and DRA libraries allow the programmer to effectively use the shared-memory model to create files that are either local to the CPU or distributed among file systems. This allows the programmer to perform parallel I/O in the most efficient manner for a particular algorithm or particular hardware. The DRA extends the shared memory programming model to include disk and uses optimized parallel asynchronous I/O. The DRA library encapsulates the details of data layout, addressing, and I/O transfer in disk arrays objects. DRA resembles Global Arrays except that data resides on the disk instead of random access memory.

3. Design and implementation

Images are processed using a medium grain to a coarse grain computer architecture where the cost of inter-node communication is relatively high, compared to the cost of threading or to the peak performance of the processor. Clusters of workstations, such as the HP Linux Itanium 2, 128 node cluster are an example of this class of computer. A large class of image-processing tasks requires computation on a given portion of the image with associated output that is spatially localized. For example, an output image is computed by processing a window of pixels of the input image. Hence if the "bordering" pixels of each window are addressed appropriately, the output pixels can be computed in a perfectly parallel fashion. The difference between a fine grain and a coarse grain parallel decomposition is in the size of the block of the input image that is being distributed among the processor and the workload. The approach PiCEIS uses for analysis is to divide an image into tiles with extended overlap regions and have each computational node perform computations on a subset of the tiles. For PiCEIS, this depends on the particular architecture characterized by the latency of communication between the CPUs and the nodes versus flop performance.

PiCEIS is implemented using objected-oriented C++, allowing for componentization and encapsulation of the more complicated computational aspects. The architecture is divided into several distinct components: file input and output, computation, and display (Fig. 3). The computational components use encapsulation, separating the analysis from the communication. For local processing, each computational component is implemented as a serial algorithm. The communication layer and data distributions are handled at a higher level so developers can easily integrate new algorithms into the architecture. As a result, PiCEIS contains a variety of serial algorithms, some native to PiCEIS and some from other libraries, which can interact with each other in parallel.

For example, PiCEIS integrates the Canny algorithm as a computational component from OpenCV, Intel's Open Source Computer Vision Library. To correspond with the interface, PiCEIS initializes the OpenCV image structure, IplImage, and populates the structure with a chunk of image data. Other parameters, such as the low and high threshold values, are read in from text files containing key-value parameter pairs. Because the division of the data is done before calling any of the computational functions, PiCEIS only has to call the OpenCV library function. OpenCV handles the computation and the resulting computed data structure is then extracted from the OpenCV data structure. This is passed back into the Global Array objects. By enclosing the Canny algorithm in PiCEIS's computational component, the ability to integrate and encapsulate different imaging operations is demonstrated. PiCEIS gains access to the many functions available in the OpenCV library while still allowing for computations on large datasets across many processors.

PiCEIS uses the shared-memory, Global Arrays programming model, where the storage of a large image may be physically distributed among the processors or nodes of processors (Fig. 4). Once the data is read into a Global Arrays data structure, Global Arrays handles all the low-level data indexing and addressing among all the processors in a parallel fashion. Thus, from a programmer's point of view, the image is accessible as if it resided on a single processor, block indices.

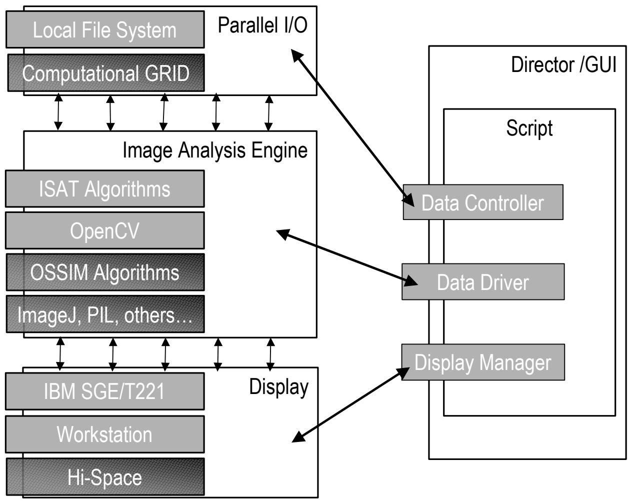

Figure 3: Modular design of PiCEIS. The three major components are driven by a script file read at run time to determine which analysis algorithm is going to be displayed, what device the display will use, and how the data will be read and written from disk.

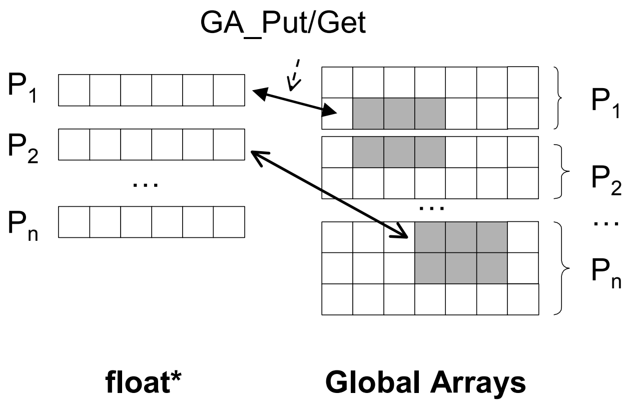

Figure 4: Each processor uses GA_Put and GA_Get methods to access pointers to the data in memory from the Global Arrays data structure.

The Global Arrays toolkit provides an efficient and portable globally addressable, shared-memory-like programming interface for distributed shared-memory computers. Each process in a MIMD parallel program can asynchronously access logical blocks of physically distributed dense multi-dimensional arrays, without need for explicit calls to message passing. Unlike other shared-memory environments, the Global Arrays model exposes the programmer to the shared memory and hierarchical hardware characteristics of the high-performance computers and acknowledges that access to a remote portion of the shared data is slower than to the local portion. The locality information for the shared data is available and direct access to the local portions of shared data is provided.

Global Arrays have been designed to complement rather than substitute the message-passing programming model. The programmer is free to use both the shared-memory and message-passing paradigms in the same program, and to take advantage of existing message-passing software libraries. Global Arrays is compatible with MPI.

The following pseudo-code shows how data is accessed and analyzed using Global Arrays and serial analysis algorithms.

```
void runAnalysis(GlobalData* inputDataObj,
                 GlobalData* outputDataObj,
                 ComputeFunction* compute)
{
   // Get the indices for the data
   long indices[ndim];

   // loop over all the data
   while (blocking_function(indices))
   {
      // Read data from Global Arrays
      float* input_data = dataObj->get(indices);

      // Analyze the data
      float* output_data =
         compute->analyze(input_data, indices);

      // Write data to back to Global Arrays
      OutputDataObj->put(indices, input_data);
   }
)
```

The serial algorithms and data-access methods are wrapped with an abstract C++ interface allowing for a simplified modular architecture. Each serial algorithm is derived from ComputeFunction, which has a single method for analysis. This object needs only the indices that the algorithm is operating on and a pointer to the data to perform the computation. The data access implementation is abstracted from a class called GlobalData. This allows for a convenient object-oriented implementation of Global Arrays. It also allows for other possible data access implementations. For example, when data is too large to fit into memory, DRA is needed to store data out-of-core, on disk (Fig. 5). Access is the same as the Global Array implementation shown above. The methods accessing data using Global Arrays and DRA are abstracted from the class GlobalData and initialized early in the program. Through the use of these two abstractions, the type of data access method and algorithm can be determined at run time using a series of parameter files.

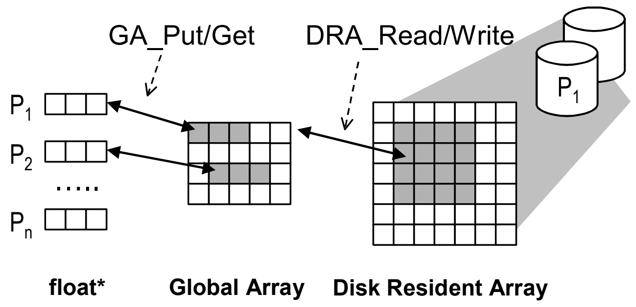

Figure 5: Data access using DRA to store data on disk.

3.1 Input/output

The parallel input method is straightforward. The name of the file to be processed and its location are read from a file accessible to all of the processors. Each processor p stores n/p parts of an image with a total of n parts. If a parallel file system is available (e.g., IBM GPFS, Linux PVFS, SGI XFS, etc.) each processor can open a given file separately as read-only and move the file pointers to its part of the file and read and store the data locally. The algorithms implemented are block-based and data can be redistributed dynamically, as needed. This data movement and realignment is usually difficult to write using straight message passing.

The image is stored as a cube of floating-point numbers in which each pixel position has a corresponding vector of spectral information. This is stored in a blocked, distributed format to preserve image continuity and gain performance in the parallel environment.

The output image can be stored as files in a number of ways: as a collection of individual files labeled with their processor number as a name extension, as one large file, or as a DRA file, which can be read and manipulated using parallel I/O methods [10].

3.2 Visualization and display output

One of the key problems of processing large, distributed data is the onerous compute-intensive and time-consuming task of gathering all of the distributed pieces to a central location and rendering for display. The output mechanism for PiCEIS couples the parallel processing with parallel rendering to take advantage of parallel frame buffer and compositing hardware. The IBM SP at PNNL is equipped with an IBM Scalable Graphics Engine (SGE) [14]. The SGE is a scalable high-performance graphics frame buffer that is directly connected to the GigE network communication switch fabric by 4 links (16 is the maximum number). These

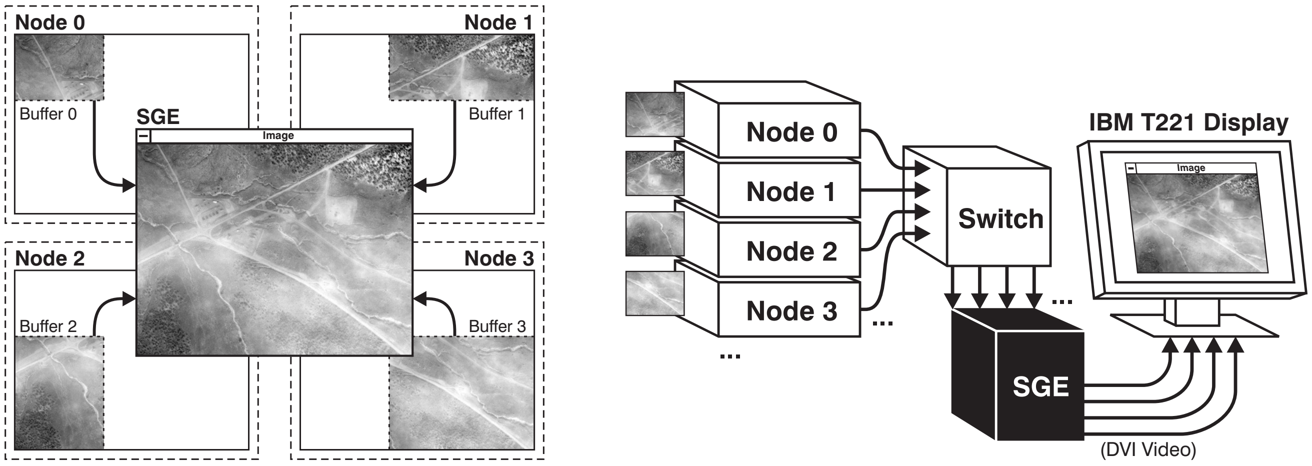

Figure 6: Display interconnect and distribution of images using the IBM SGE.

parallel switch links provide very high bandwidth from the compute nodes to the frame buffer. Each link is theoretically capable of a bandwidth of 45 mega-pixels/sec.; 24-bit color, full-motion, parallel image output using the SGE has been benchmarked [12].

The PiCEIS manager communicates with the software that synchronizes with the SGE, allowing each completed output image to be displayed (Fig. 6). Interaction with the data is achievable through various interface tools [13].

4. Performance and experimental results

PiCEIS demonstrates scalable performance on two platforms at the MSCF [9]. Figure 7 shows PiCEIS executing on an IBM SP2. The IBM SP2 test machine is a distributed shared-memory computer with 26, 4-CPU nodes. Each CPU is an IBM Power 3 processor executing at 375 MHz. Each node has 3 GB of memory. In this case, the input file is a Landsat 300 Mbyte image. Figure 8 shows PiCEIS executing on a new HP Linux cluster with a 500 Mbyte Landsat image. This machine is a 128-node/256 Itanium-2 processors (McKinley processors) with 1 Tflop peak theoretical performance and 1.5 Tbytes of RAM. On both machines, the image is converted to 8-byte floating-point format, and then two algorithms, Texture and Modified Radon Transform, are performed. Scalability is shown in both implementations. The bump in the HP timings is due to a new, evolving system architecture.

5. Conclusion

This paper addresses a number of issues associated with parallel scaling (to large numbers of processors) in

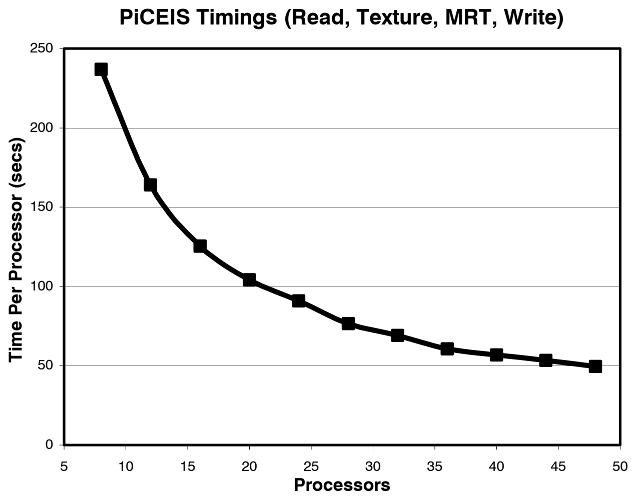

Figure 7: Timings from an IBM SP2.

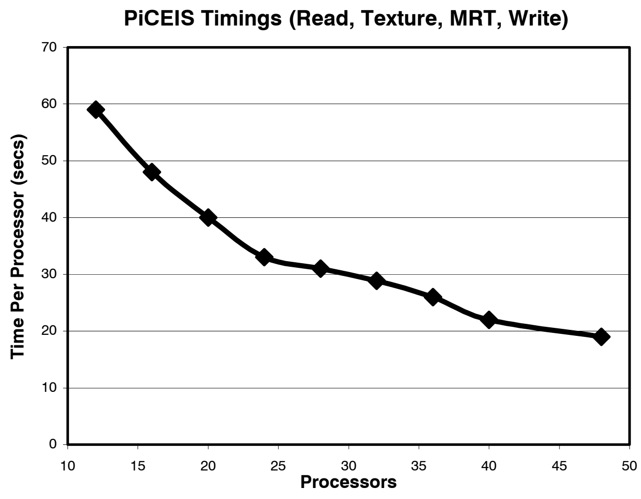

Figure 8: Timings from an HP Linux cluster.

image processing. A key contribution to the scalability is the use of the Global Arrays tools to enable the shared-memory programming model with one-sided communication and asynchronous parallel I/O. These tools enable effective overlap of computation and I/O communications. Use of an object-oriented framework provides developers with an easier framework to develop scalable software for massively parallel computers. The performance numbers indicate we are approaching our goal of real-time image processing of very large image files. The goal is to be able to process large image files (greater than 75 megapixels) under 10 seconds. The initial results are very encouraging. We feel we can achieve this goal by continuing to optimize the current process and working on the I/O subsystem.

6. Future directions

Currently, PiCEIS uses a unique set of script files to read the input and define the program execution flow. These files are read in using a simple loop. Each file contains commands and parameters for a task module in the PiCEIS framework. If we were to use the Python scripting language, this process could be translated into a series of Python objects, allowing more flexible and dynamic control of the program execution. In particular, the user would have more control over the path of data flow and could define the flow path interactively and more maturely. Many of the current commands that Python would call would be implemented at a high level, allowing for the Python control to remain separate from interprocessor communication or processor-specific function calls. This would increase the use and flexibility of PiCEIS, while keeping the scalability and ease of use of new algorithm integration.

Other future directions include analysis and viewing of 3-dimensional data. Although PiCEIS is implemented to handle 3-dimension data, it is currently designed for optimal performance with 2-dimensional image data. Incorporating more complicated 3-dimensional data would extend the current capabilities of PiCEIS to other areas of research.

Acknowledgements

Development of Global Arrays has been recently funded by U.S. Department of Energy DOE-2000 ACTS program and DOE/MICS base program on Scalable Programming Models.

This research was funded by the Pacific Northwest National Laboratory's LDRD Program Computational Sciences and Engineering Initiative. The computation was performed in the Molecular Sciences Computing Facility at the William R. Wiley Environmental Molecular Sciences Laboratory, PNNL. The MSCF is funded by the Office of Biological and Environmental Research in the U.S. Department of Energy. Pacific Northwest National Laboratory is operated by Battelle Memorial Institute for the U.S. Department of Energy under contract DE-AC06-76RL01830.

References

[1] W. E. Alexander, D. S. Reeves, and C. S. Gloster Jr., "Parallel processing with the block image data parallel architecture," *IBM J. Res. Dev.* 44:5, 2000.
[2] S. Balay, W. D. Gropp, L. C. McInnes and B. F. Smith, "PETSc users manual," ANL-95/11-Revision 2.1.3, Argonne National Laboratory, 2002.
[3] L. S. Blackford, J. Choi, A. Cleary, et. al., *ScaLAPACK User's Guide*, SIAM, Philadelphia, 1997.
[4] G. Fann, "PeIGS V.3 user's manual," Pacific Northwest National Laboratory, 2001.
[5] W. Gropp, S. Huss-Lederman, A. Lumsdaine, et. al., *MPI: The Complete Reference: Volume 2, The MPI-2 Extensions*, 2nd ed., MIT Press, 1998.
[6] High Performance Computational Chemistry Group, *NWChem, A Computational Chemistry Package for Parallel Computers, Version 4.1* (2002), Pacific Northwest National Laboratory, Richland, Washington 99352, USA. http://www.emsl.pnl.gov/pub/docs/nwchem/
[7] Khoral Khoros, http://www.khoral.com
[8] J. Lindgren, "Getting a grip on large images," *Imaging Notes* 16:6, 2001
[9] Molecular Science Computing Facility, http://mscf.emsl .pnl.gov
[10] J. Nieplocha, R. J. Harrison, and R. J. Littlefield, "Global Arrays: a nonuniform memory access programming model for high-performance computers," *The Journal of Supercomputing* 10, pp. 197-220, 1996.
[11] OpenCV, http://www.intel.com/research/mrl/research/opencv/
[12] K. A. Perrine, and D. R. Jones, "Parallel graphics and interactivity with the Scaleable Graphics Engine," *Proc. IEEE Supercomputing*, 2001.
[13] K. A. Perrine, and D. R. Jones, "Interactive Imaging Science on Parallel Computer: Getting Immediate Results," *Proc. IPDPS* 2003.
[14] K. A. Perrine, D. R. Jones, P. Hochschild, and R. A. Swetz, "An interactive parallel visualization framework for distributed data," *Visualization and Data Analysis*, R. F. Erbacher, et. al. eds., Vol. 4665, pp. 196-206, SPIE, 2002.
[15] The Python Imaging Library (PIL), http://www.pythonware .com/products/pil/
[16] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI: The Complete Reference: Volume 1, the MPI Core*, 2nd ed., MIT Press, 1998. (See also http://www-unix.mcs.anl.gov/mpi/).
[17] J. M. Squyers, A. Lumsdaine, and R. L. Stevenson, "A toolkit for image processing," preprint 1997, http://www.osl.iu .edu/research/pipt/