and compilation.

8. References

1. R. Jagannathan and C. Dodd. *GLU Programmer's Guide (version 0.9).* Computer Science Laboratory, SRI International, Menlo Park, CA, 1994. Technical Report SRI-CSL-94-06.
2. D. Le Gall. MPEG: A video compression standard for multimedia applications. *Communications of the ACM*, 34(4):46–58, April 1991.
3. K. L. Gong. *Berkeley MPEG-1 Video Encoder: User's Guide.* Computer Science Division, University of California, Berkeley, CA, 1993.
4. S. R. Deans. *The Radon Transform and Some of its Applications.* Wiley, 1983.
5. B. P. Medoff, W. R. Brody, and A. Macovski. The use of a priori information in image reconstruction from limited data. In *Proceedings of ICASSP*, pages 131–134, 1983.
6. S. G. Azevedo. *Model-based Computed Tomography for Nondestructive Evaluation.* PhD thesis, Lawerence Livermore National Laboratories, UCRL-LR-106884, 1991.
7. E. Shieh, K. W. Current, P. J. Hurst, and I. Agi. High-speed computation of the radon transform and backprojection using an expandable multiprocessor architecture. *IEEE Transactions on Circuits and Systems for Video Technology*, 2(4):347–360, 1992.
8. E. B. Hinkle, J. L. C. Sanz, A. K. Jain, and D. Petkovic. P3E: New life for projection-based image processing. *Journal of Parallel and Distributed Computing*, 4(1):45–78, 1987.

cal filesystem. This degradation would be even worse if our implementation did not seek to minimize the data transfer across the network. To minimize the data-transfer time for data-intensive applications, the processor configuration should be flexible and tuned to the application.

Another possible improvement to GLU for multidimensional signal processing applications would be to use an established graphical user interface. Many algorithms for these applications are being coded today with tools such as Simulab and Ptolemy. These tools provide a convenient mechanism for assembling and verifying complex algorithms from a set of basic functions defined by Matlab and C++ code, respectively. One advantage of using these existing tools over a GLU-specific tool is that many users are already familiar with their operation, so the learning curve for the user is low. A second advantage is that communication between the blocks is well defined. A final advantage is that these tools are readily extensible, with various hooks for different downstream tools.

Another possible improvement to the GLU implementation is the handling of global variables. There exists a great deal of code that uses global variables for interprocess communication. If this code is to be easily parallelizable with GLU, the treatment of the global variables must be handled in an unambiguous manner. In the applications discussed here, most of the global variables were used as control signals, coming down from the top to the lower-level procedures. In this case, it would be extremely useful to treat these variables as global "constants" and make sure that their values are consistent with their values in the generator. Alternatively, it would be useful to have a migration tool that helped identify the global variables and their use in a sequential program.

6. Conclusion

We described parallel implementations of two multidimensional signal processing applications: MPEG encoding and computed tomography. The inherent parallelism of signal processing algorithms makes them easily convertible for parallel programming platforms, such as GLU. We also presented the methodology used to convert these existing applications into a form usable in GLU. We observed that the best approach was to perform parallelization at the highest level possible, which minimizes the recoding and the implementation time. We described the benefits of GLU for signal processing applications, which are similar to many other computing tasks, and gave suggestions for possible improvements.

7. Acknowledgments

Special thanks to R. Jagannathan and C. Dodd for their help with the GLU coding

programming environment would greatly enhance its usefulness.

5.1. Benefits of GLU for Signal Processing Applications

The GLU functional syntax and its notions of causality and multidimensional space provide a good platform for implementation of signal processing applications in software. GLU also provides a convenient mechanism for parallelizing existing code.

The main benefit of using GLU over hand coding for parallelization of existing code is the decrease in complexity. The decrease in complexity results from having the interprocedure communication mapping, scheduling, and load balancing taken care of by the GLU compiler. Most applications implemented in software would not be written for parallel execution simply because of the time and effort involved in writing the communication software. Because GLU handles all the interprocess communication and parallelism management, it allows the designer to concentrate on the application instead of the coding.

Another benefit of GLU is that existing code can be converted to GLU procedures with little effort if there are few or no global variables in the main routine and it is written in a modular design. Further, the software approach used in designing the parallel code with GLU can aid in the design of hardware. As an added benefit, transparent parallelism discovery and management is possible with GLU coding, since the user does not directly perform the mapping and scheduling. This transparent discovery mechanism could also aid in hardware design by feeding the software mapping to the user. Since the GLU compiler is available on many different architectures, the GLU code is portable and is not tied to a specific architecture or operating system. Finally, GLU's fault tolerance also ensures that the calculations are complete.

5.2. Suggestions for Improvement

Currently, the GLU implementation uses a single generator that handles all the interprocess communication and multiple workers that handle the parallel procedures assigned to them by the generator. While this may be a model that is easy to implement, it does have some drawbacks.

Fine-grain computations of signal processing applications generally require local communication. The generator/worker model makes interworker communication cumbersome since all the data must go through the generator. One can argue that since these parallel processes run on a set of machines that communicate over a serial link—that is over the Ethernet—"local" communication takes only half the time of going through the generator. This argument may generally hold for a network of workstations, but it is not true in all architectures. Figure 3 shows that using remote workers and transferring data across a network degrades the performance of the MPEG encoder compared to using local workers in a multiprocessor with a lo-

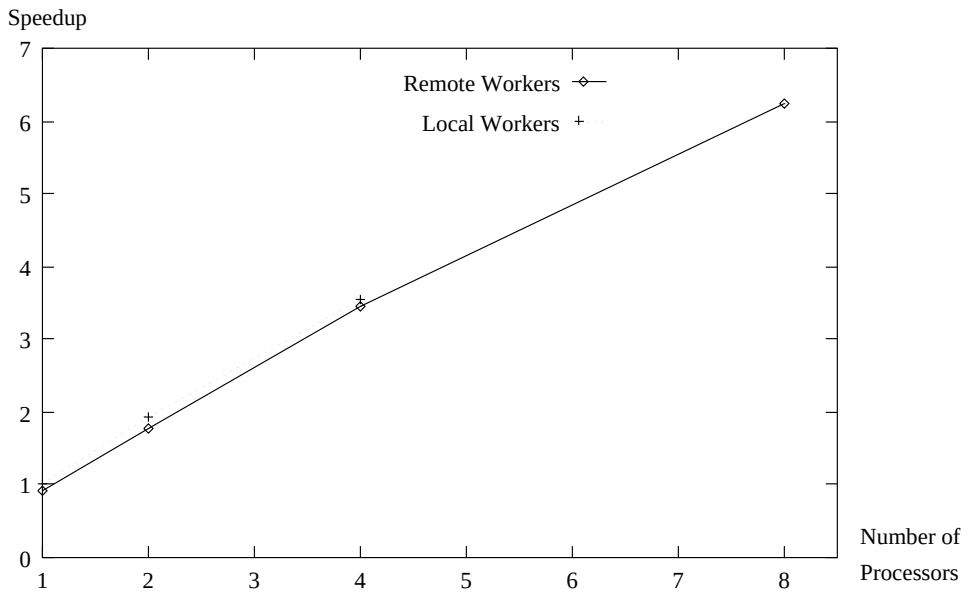

Fig. 6. Processing speed increase as a function of the number of processors for CT example.

and the starting angle as inputs, and returns a partially reconstructed image. N of these partial reconstructed images are produced and passed to `AddImage` to be combined into a single image. The starting angle for each worker is incremented by π/N until all the angles are covered. The GLU code for this example is simpler than the MPEG code because the sequence of calculations is unimportant.

The CT code required relatively larger modification to the original code than the existing MPEG code. The original MPEG code was written to perform the forward transform, the filtering and the backprojection sequentially. Efficient parallel implementation of this algorithm requires balancing the processing load among the workers and reducing the data transfer. The code was rewritten to distribute the forward transform and backprojection calculations among the workers.

Figure 6 shows the speedup of CT implemented with GLU. The speedup is nearly a linear function of the number of processors. The speedup starts to tail off when the number of processors is increased to eight because the increase in network traffic. The backprojection and the forward transform take the largest amount of computation time in CT reconstruction, and the backprojection requires the most interworker I/O during the final addition. A great deal of this I/O is eliminated by performing partial backprojections in each worker, before the the final backprojection is calculated.

5. Evaluation of GLU for Multidimensional Signal Processing

The current implementation of GLU is beneficial as a programming tool for multi-dimensional signal processing applications. Extensions in GLU's implementation and

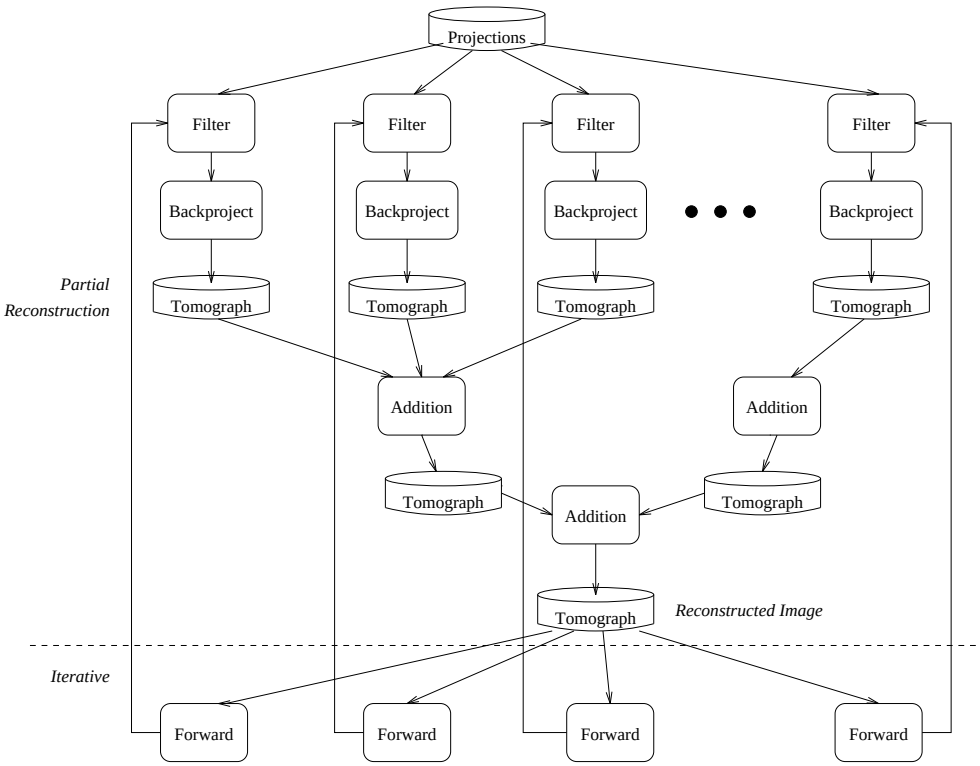

Fig. 5. Software implementation of CT reconstruction.

in polar coordinates. The generator sends a unique subset of the projections to each worker. The data independence of each projection allows each worker to filter its subset of projections in parallel. The worker then generates a partially reconstructed image by backprojecting its set of filtered projections. Since the backprojection is associative, the workers simply add the partially reconstructed images together to fully reconstruct the image. This addition operation is performed by the generator as the partially reconstructed images are calculated by the workers.

Iterative reconstruction algorithms require that the forward Radon transform is calculated in addition to the filtering and backprojection already described. The workers calculate a set of projections from the image.

The GLU function `GluRadon` used to implement the MPEG encoder is:

```
GluRadon ( imageIn, globals, theta0, N) =
  linear_tree.c ( AddImage,
                  ProcessProjections(imageIn, globals, theta), N)
    where
       dimension c;
       theta = theta0 fby.c theta + (M_PI/N);
    end;
```

The function `ProcessProjections` takes the input image, the global constants,

global variables by explicitly passing the whole set as arguments to the procedure `processGOP` on all the workers. The second major modification replaced the existing GOP generation routines with a substantially simplified version. The new code replaced socket calls and various frame ordering routines of the original version with more compact GLU code. Since the GLU system handles interprocessor communication and frame ordering (specified implicitly by the GLU code), these operations don't have to be coded explicitly. These modifications greatly simplified the code, decreasing the length of the GOP generation function by half.

4. Computed Tomography

Computed tomography is another application that requires substantial computing power. X-ray attenuation data measured by a set of sensors in a CT scanner form the Radon transform of a slice through a three-dimensional (3-D) object. The Radon transform of the object can be thought of as the set of shadows of the object, illuminated over π radians by X-radiation. The strips of the set of shadows defined by the plane that intersects the Radon transform are called the *projections* of the slice (or tomograph) through the object. The inverse Radon transform, commonly implemented as a *filtered-backprojection*, reconstructs a tomograph from the CT projection data.[4] Each tomograph of the object can be reconstructed independently. A set of tomographs reconstructed perpendicular to the plane of intersection generates a 3-D data set that represents the average volumetric density throughout the object. A useful property of the filtered-backprojection reconstruction scheme is that the processing of each projection is independent of all other projections, thus simplifying parallel computation.

CT reconstruction from large data sets using filtered-projection techniques takes on the order of minutes to complete on a single processor computer. Reconstruction of large or dense objects often requires iterative techniques that map the data back and forth between image space and Radon space, requiring the implementation of the forward Radon transform as well.[5,6] If 3-D reconstruction is required, the computation time can easily take hours to complete. Several specialized hardware architectures have been proposed to reduce the reconstruction time.[7,8] We use a software approach.

The CT code performs three functions—projection filtering, backprojection, and the forward Radon transform. These three operations can be used in a variety of ways. Single-pass reconstruction requires filtering and backprojection. Iterative reconstruction schemes require all three operations. Or, as in our case, the forward transform is used to calculate the projections from a test image, so that various reconstruction algorithms and finite-precision effects can be simulated.

Figure 5 shows the architecture of the CT implementation. During the recoding of the CT reconstruction code, we broke up calculations among the workers to minimize the data I/O. For single-pass reconstruction the projections are saved as an image

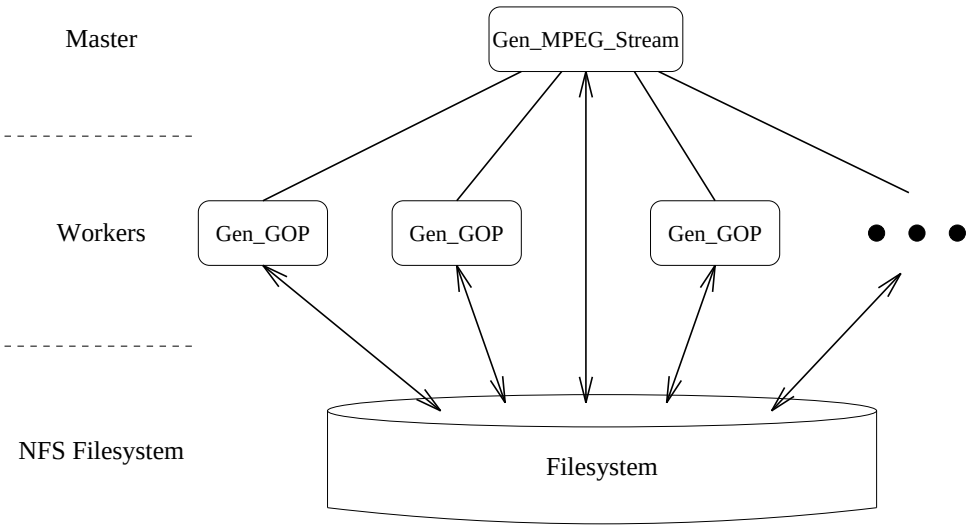

Fig. 4. Software implementation of the MPEG encoder.

of the functions are performed in the generator. Figure 4 shows the mapping of the functions to generators and workers.

The MPEG encoder can be parallelized at many levels, from the pixel level up to the GOP level. Our implementation breaks up the calculation at the level of the GOP for a number of reasons.

The first reason was that the code already existed that produced a single GOP in the sequence from the input video sequence, and code existed that combined a number of GOPs to form a MPEG sequence. While this code had to be modified to work under GLU, the existence of the code greatly simplified the assembly of the resulting GLU MPEG code.

The second reason was that a GOP has at most a dependency of one frame between it and frames in adjacent GOPs. If the frame pattern ends in a P- or I-frame the GOP has no dependency on frames in adjacent GOPs. This lack of dependency minimized bookkeeping and the read time of the function `processGOP` to a single read per frame. It also reduced the amount of data the workers would output since the GOP produced by `processGOP` is a compressed bit stream. Other parallel encoders try coding MPEG sequences by using frames as their basic unit,[3] resulting in complicated bookkeeping and increased data transfer between processors during interframe compression.

The third reason was that the length and the frame pattern of the GOP, and therefore the amount of work for the function `processGOP`, is user-definable at the beginning of the encoding sequence. Therefore the computation time of each process can be changed if the latency of the bit stream is important.

The existing MPEG code was modified in two major ways to implement the GLU version. The first major modification grouped the 200 or so global variables and structures one structure. This grouping reduced the ambiguity of the values of the

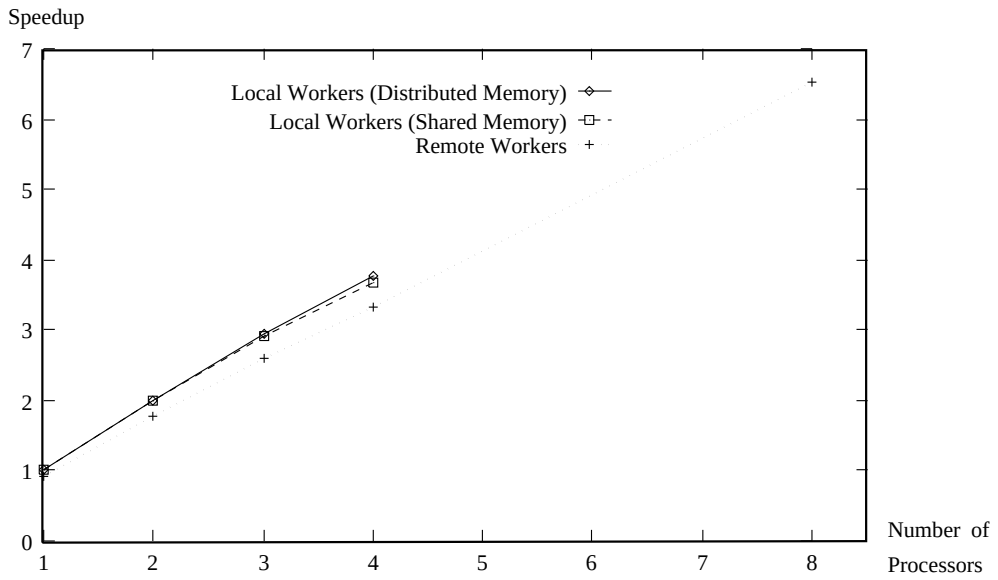

Fig. 3. Processing speed increase as a function of the number of processors for MPEG video encoder.

encoding speeds (30 FPS) with half as many processors. The major factor that keeps these data rates from being reached is the communications bandwidth. To obtain real-time operation for this image size, the interprocessor communication must have enough bandwidth to sustain a rate of 38 Mb/s. Larger images would require a larger bandwidth.

The GLU function `GenMPEGStream` is used to implement the MPEG encoder:

```
GenMPEGStream(numFrames, ofp, outputFileName, globals, huffTable) =
  linear_tree.gop(same, N, numGOPs+1),GOPStoMPEG(GOPNumber, GOPLength, N)
    where
     index gop;
     numGOPs = numFrames / framePatternLength;
     framePatternLength = GetPatternLength(globals);
     N = 0 fby.gop GOPStoMPEG(GOPNumber, GOPLength,  N, outputFileName);
     frameNumber = 0 fby.gop frameNumber + framePatternLength;
     GOPLength = processGOP(InputFileNames, GOPOutputFileName,
                            ofp, GOPNumber, globals, huffTable);
     InputFileNames = GetInputFileNames(frameNumber, globals);
     GOPOutputFileName =
        GetGOPOutputFileName(GOPNumber, outputFileName, globals);
     GOPNumber = GetGOPNumber(frameNumber, globals);
     same(a,b) = a+b-a;
end;
```

This implementation of `GenMPEGStream` forces the ordering of the GOPs sent to `GOPStoMPEG` to be sequential. The generation of the GOPs takes place in the function `processGOP`. In our implementation the workers perform this function and the rest

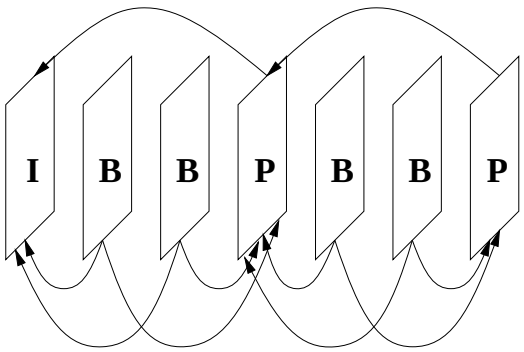

Fig. 2. Pictorial representation of a group of pictures (GOP) showing frame dependency.

A number of these blocks strung together form a *slice* (of a frame) that acts as a resynchronization unit. These slices combine to form a *picture*. Pictures are grouped to form a random access unit so that the video can be viewed either forward or backward, as illustrated in Figure 2. This group of pictures (GOP) has little or no dependence on the pictures in adjacent GOPs.

The GOP consists of intra-coded (I), forward predicted (P), and bilinearly predicted (B) frame. An I-frame is independently coded and thus has no frame dependency. A GOP must begin with an I-frame. A P-frame coding depends on the previous I- or P-frame. The B-frame coding depends on both the immediately preceding and the immediately following I- or P-frame. If the designated frame pattern ends with a B-frame then the current GOP depends on the I-frame of the subsequent GOP, otherwise it is independent.

The compression using spatial and motion techniques yields a bit stream that is compressed even further by the encoder using a variable-length (Huffman) coding. This resulting bit stream becomes the MPEG sequence with the addition of the necessary sequence and GOP headers.

In the implementation of the GLU code, a C procedure was written that calculated and wrote a GOP bit stream to a file. This procedure is run in parallel, so that all the workers calculate a GOP at the same time. As the GOPs are written to the files, the generator concatenates these GOP bit streams in serial order to form the MPEG bit stream. This coarse-grain parallelization of the MPEG encoder gives very good parallelization results. Figure 3 shows the speedup of the encoder as a function of the number of processors. This graph clearly shows that we get linear speedup and nearly 7 times the speed using eight remote processors.

A single SPARCstation 2 [a]achieves 0.381 FPS encoding rate on a 360 x 288 image, and eight remote workers achieve 2.48 FPS. Therefore, achieving real-time encoding rates (30 FPS) based on the speedup factors shown in Figure 3 requires a minimum of 84 local workers or 97 remote workers. A SPARCstation 20 can achieve real-time

[a]All product names mentioned in this paper are the trademarks of their respective holders.

found that the most natural way to break up these signal processing applications is to look for a "for-loop" at the highest level possible that will allow independent processing, known as single-program multiple-data (SPMD) parallelism, where the identical program is executed on independent sets of input data. This approach has the advantages that it saves recoding and that other techniques would not improve performance. We chose this implementation model for both the MPEG encoder and the CT reconstruction code.

2.1. Conversion Methodology

The methodology we used to convert existing applications is:

- Break the computation at the largest possible granularity level to reduce generator-worker communication traffic.
- Minimize the changes to the original program structure to reduce coding time.
- Pass large data sets between the generator and workers through files to reduce bookkeeping and simplify memory management.
- Combine all global variables into a single structure in the generator. Pass this structure to all the workers to eliminate any ambiguities.

3. MPEG Encoding

The MPEG encoder compresses a color video sequence using a combination of compression schemes, and produces a serial bit stream that can be sent over a communications channel. At the end of this channel, a MPEG decoder this bit stream and reproduce a facsimile of the original video sequence. The MPEG encoder is by far the most computationally intensive part of this scheme. Currently, MPEG bit streams of reasonable image sizes can be decoded in real time, typically 30 frames per second (FPS), by software running on modern workstations. However, a sequential software encoder running on the same machine may take roughly two orders of magnitude more time to generate the same sequence.

MPEG encoding of a video sequence requires many compression steps. Each frame in the video sequence is compressed using a combination of block-based motion compensation to take advantage of interframe temporal redundancy, and discrete cosine transform (DCT)-based compression to take advantage of intraframe spatial redundancy. [2] Each frame (picture) in the sequence is broken into 8 x 8 pixel *blocks* for interframe DCT compression and 16 x 16 pixel blocks for intraframe motion compensation.

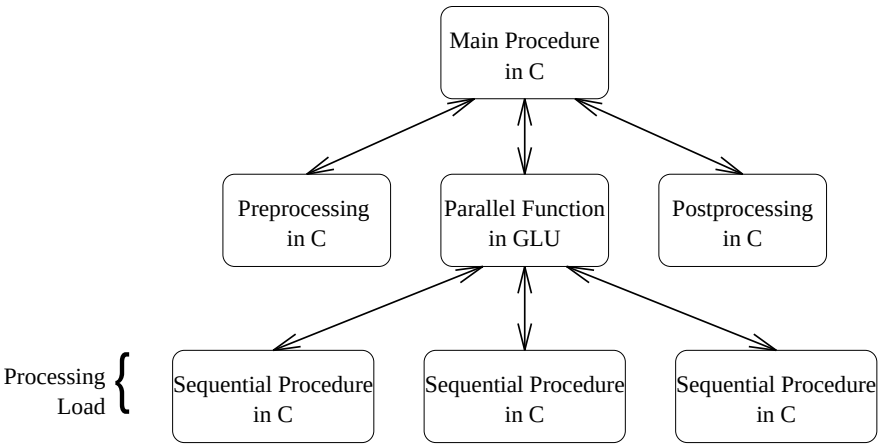

Fig. 1. Software architecture for MPEG video encoding and CT.

the GLU system.

The MPEG encoding and CT applications were implemented from a combination of rewritten existing C code and GLU code. We modified existing code so that it could be used with the GLU compiler to produce the generator and worker binaries. Figure 1 shows the software architecture for both applications.

At the top level, we have the main procedure in C, which calls preprocessing C routines that parse the arguments, allocate memory, and so forth. Then the main procedure calls a GLU function that in turn calls the C procedures to be run in parallel. The sequential procedures in C called by the GLU function bear the bulk of the processing effort.

Parallelizing existing code with a parallelization platform, such as GLU, often forces the designer to transfer large or variable-length data structures from one process to another through files. The other processes use the name of the file as a pointer, and access the data by a file read.

Often, the transfer of data between processes directly using GLU is more complicated than transferring the same data through files. Direct transfer requires precise knowledge of the data structure, but since specialized file read and file write routines often come with the signal processing application or are available in system libraries, data transfer through files requires a simple function call with little knowledge of the exact data structure. In addition, the transfer of these data structures through files allows the data structure definitions and the lower-level functions to remain unchanged so long as the interface of the GLU functions is modular.

Large data structures present another problem. The data must pass through the GLU generator to get from one process to another. This pass-through may require more time than a write and read from the file system. The file write has the added advantage of preserving intermediate results.

The motivation for parallelizing existing code is to get linear speedup. We have

GLU FOR MULTIDIMENSIONAL SIGNAL PROCESSING

Iskender Agi
Computer Science Laboratory, SRI International, 333 Ravenswood Avenue
Menlo Park, California 94025, USA
E-mail: iskender@csl.sri.com
WWW: http://www.csl.sri.com/iskender

ABSTRACT

Multidimensional signal processing applications have a great deal of inherent parallelism. Two examples of these applications—MPEG encoding and computed tomography—can be implemented in parallel. With the described methodology these applications were converted into a form usable in GLU and implemented on a network of workstations. Improvements to the GLU programming environment can benefit other inherently parallel applications.

1. Introduction

Multidimensional signal processing applications have a great deal of inherent parallelism. Many spatial signal processing applications are also noncausal and shift-invariant. These applications use various mathematical transformations performed on well-defined data and have well-structured local and global communication. These properties of multidimensional signal processing applications allow most of these transformations to be calculated concurrently.

We describe the parallel implementation of two multidimensional signal processing applications using GLU.[1] We explain the methodology used to convert these two applications from existing sequential code, comment on the benefits of using GLU for these applications, and offer some suggestions on how to improve the programming environment. The two applications that we consider are the Moving Picture Experts Group (MPEG) video encoding, and computed tomography (CT).

2. Software Architecture

GLU is a high-level system for constructing parallel software using existing sequential code and portably executing this software on diverse parallel computers ranging from heterogeneous workstation clusters to shared-memory multiprocessors to massively parallel processors.

A GLU application is a very high-level dataflow graph whose nodes denote sequential components with well-defined interfaces and whose edges denote data dependencies. The programmer is only responsible for constructing the dataflow graph, defining the sequential components, and selecting a virtual parallel application architecture. All other aspects of parallel programming such as mapping, scheduling, load balancing, communication, and synchronization are handled automatically by