A Test Case for the Parallel Programming Support Environment: Parallelizing the Analysis of Satellite Imagery Data

David V. Judge
W.G. Rudd

OACIS
&
Department of Computer Science
Oregon State University
Corvallis, Oregon 97331

89-80-2

A Test Case for the Parallel Programming Support Environment: Parallelizing the Analysis of Satellite Imagery Data

David V. Judge & W.G. Rudd
Oregon Advanced Computing Institute (OACIS) &
Department of Computer Science
Oregon State University
Corvallis, OR 97331-3902

Abstract

Within a decade, the amount of raw data acquired each day by satellite-based instrumentation will exceed 1 terabyte (10^{12} bytes). Virtually all quantitative data acquired about the Earth is useful [10]. Processing the raw data into useful information will be an enormous task. Much of the current data analysis software is FORTRAN dusty-deck code. To handle the data volume expected, not only will much of the currently used code need to be transformed or redesigned into parallel code, but many new parallel programs will need to be written from scratch. Although parallel computers are already in existence, the task of writing parallel software has proved to be exceedingly difficult [1]. The Parallel Programming Support Environment (PPSE) is intended to assist the parallel programer in the process of designing a parallel program. The primary focus of this work is to test the tools of PPSE on an actual mid-sized (4000 lines of code) FORTRAN program and report on the experience. The program (AVHTST) is part of a series of FORTRAN data analysis programs which automate the process of determining cloud properties from satellite imagery data. Although AVHTST has been in use for several years, it has been identified as being in need of parallelization in the literature [5]. The objective of this research is to concurrently examine viable methods of producing a useful parallel program while testing the utility of the PPSE tools. The product of this research is a prototype parallel version of AVHTST, a description of the usage of the PPSE tools, a list of possible improvements and extensions to the tools, and a plan for integrating the tools.

1. Introduction

Parallelism is a natural notion. The optimal solution to nearly all problems can be achieved through concurrent work. The strategy of attacking problems in parallel has been effectively employed throughout human history. But computer programs are different [1]. Although a large proportion of all existing computer programs solve problems in a strictly sequential manner, the optimal computable solution (with regard to time and accuracy) to these problems can be achieved through concurrent solution steps.

Whether parallel programming requires a greater skill level than sequential programming is unclear. However, different skills are needed. Sequential programmers frequently are not required to understand the entire problem being solved. Rather, they concentrate on solving many sub-problems and append the solutions together into a working program. Parallel programs must be carefully designed. In the current state of the art, a parallel programmer must thoroughly understand the problem and the relationships of its parts, the solution strategy, and the resources (language and machine) on which the solution will be implemented. For most people, the complexity and frustration of parallel programming outweigh the benefits; very few researchers outside of computer science are writing parallel programs and very little parallel production work is actually done [14].

A large number of sequentially coded solutions to difficult problems could be redesigned to run on existing parallel hardware if the frustration and complexity of the parallelization task could be alleviated. For many researchers, parallel solution strategies would allow timely results to extremely large problems opening up whole new avenues of research. Atmospheric Scientists, for example, find satellite gathered data to be extremely useful for studying and modeling certain phenomena. But while a large amount of incoming high resolution data will swamp sequential processing routines, and lower resolution data won't provide the desired experimental accuracy, the scientists are limited in the spatial scope of their research. In order to process global amounts of high resolution data, parallel or distributed processing techniques are necessary.

As an example, the Spatial Coherence Method [5,6] consists of passing data through several sequential FORTRAN programs in order to objectively derive cloud properties from satellite imagery data. The first pass program, responsible for data analysis and reduction, is called AVHTST. With large input sets, the runtime of AVHTST drastically increases and dominates the total processing time. A typical scene of data (a single data set) consists of 5 256*256 arrays of pixel values (the values correspond to measured radiances at 5 distinct wavelengths). Each scene typically covers an actual $(1000\ \text{km})^2$ region (see figures 1 and 2). The current sequential version of AVHTST takes 10 minutes to process all 5 channels for 1 scene of data on a typical minicomputer. A *global data set* would consist of approximately 600 scenes of data and would take at least 6000 minutes (100+ hours) to process on a minicomputer. Although global cloud studies could be extremely useful to people studying the Earth's radiation budget (or to climate modelers) the usefulness of acquiring the processed data is outweighed by the necessary computation time. The computer time (and expense) can be better spent on small amounts of data; as a result the Spatial Coherence Method is typically used to study small spatial regions.

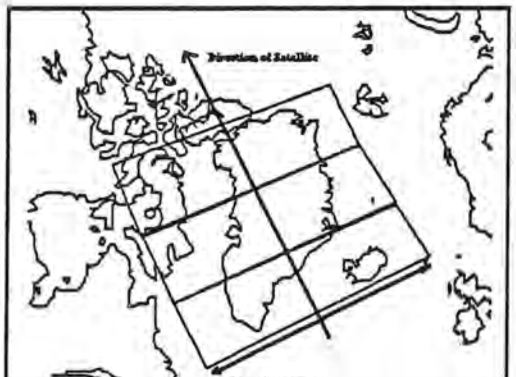


Figure 1. Three scenes of satellite data over Greenland

Although the current version of AVHTST is sequential code, numerous opportunities for parallelism exist in the *actual problem* of objectively deriving cloud properties on a global basis from high resolution satellite imagery data. Each channel of data for each scene could be processed concurrently. The scenes themselves could also be processed concurrently. The final output of the program would require communicating, collecting and correlating the results from distinct scenes and channels, but the majority of computation could take place in parallel. With suitable parallel hardware, it may be entirely possible to process 600 scenes of data in only 7 minutes (as opposed to over 100 hours sequentially). The results which were not worth 100 hours of computer time may suddenly become extremely useful if they could be computed in 7 minutes. A system with at least 1200 processors may be able to achieve these results.

Before getting into the parallel design, it will be useful to illustrate the geometry and terminology involved with this problem (from [5]).

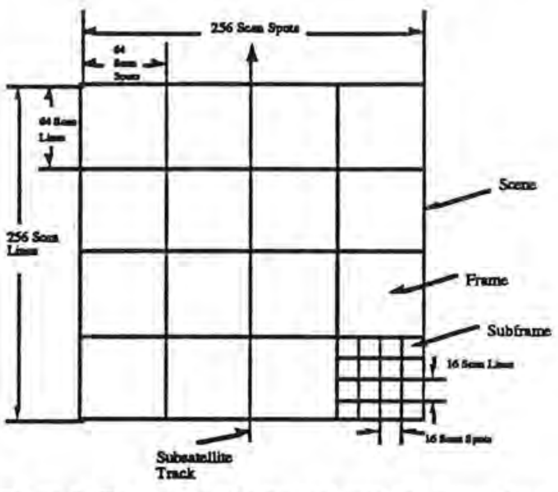


Figure 2. Illustration of a *typical* satellite scene, frame and subframe. This scene covers a geographic area of approximately $(1000\ \text{km})^2$, a frame $(250\ \text{km})^2$, and a subframe $(60\ \text{km})^2$. The problem parallelizes differently based on the geometry of the input. Scene, frame and subframe sizes are all variable.

Although the notion of achieving parallelism in this problem is clearly possible, creating actual parallel code - especially code which will be portable across a variety of parallel machines - is still an extremely difficult problem. The intuitive idea of a parallel solution needs to be clearly defined into a design for a parallel program. The primary goal of this research is to create a parallel design of the first pass routine of the Spatial Coherence Method with the help of PPSE. We are more interested in finding a parallel solution to the problem than in simply parallelizing AVHTST. That is, parallelism should be sought from the problem, not the existing sequential program. From the design, it should be possible to create an actual parallel version of the first pass routine which can be coded, compiled and run on an actual parallel system.

2. Overview of PPSE and the Application

The following sections give brief overviews of the Parallel Programming Support Environment research goals and tools, the general goals of the Spatial Coherence analysis routines (of which AVHTST is a part) and the specific tasks carried out by the sequential version of AVHTST.

2.1 Parallel Programming Support: PPSE Overview

Parallel Programming Support Environment (PPSE) research at Oregon State University addresses a series of issues related to designing and writing software for parallel computers. A number of practical tools have been developed which allow a programmer to visually design an architecture independent program, specify a high-level description of architectures on which the parallel program might run, determine a schedule or map for assigning program segments to processors, and automatically generate source code for a specific parallel computer from code fragments, the graphical description of the machine and the graphical description of the software. The major areas of research address the following general problems:

- Developing a Graphical Notation for the Design and Description of Parallel Programs.
- Developing a Graphical Notation for the Description of Parallel Machines.
- Mapping the Parallel Software to the Parallel Machine.
- Automatic Generation of Machine Dependent Parallel Source Code.
- Developing Visual Methods of Inputting the Hardware and Software Design Details.

Extended Large Grain Data Flow (ELGDF)[8,16] is a graphical language for designing parallel programs. Ideally, parallel software should be designed independent of any specific hardware on which the developed code might eventually run. ELGDF allows the development of a high level, machine independent description of a parallel program. ELGDF also allows the design of parallel software without being bound to any particular programming language. To enter descriptions of parallel programs, an ELGDF design editor which runs on a Macintosh has been developed. The design editor provides a visual method of inputting software design details in ELGDF notation. The following features have been implemented into the design editor:

- ability to produce a hierarchical design for parallel software in ELGDF notation,
- ability to add detailed textual specification to graphic notation through dialog windows,
- easy manipulation of design by resizing, encapsulating, and expanding the graphical description,
- ability to assign source code fragments to specific graphical objects,
- graphics to text (and text to graphics) transformations for interface with other PPSE tools.

Once the program design has been entered, other PPSE tools permit the analysis of the design and transformation of the design into forms such as dependency graphs, flow graphs and source code. Before these steps can be taken, however, specific implementation details must be entered.

To enter descriptions of parallel machines, a target machine editor has been developed. The Target Machine Editor provides a graphical analog to the classical Processor-Memory-Switch hardware description notation developed by Siewiorek, Newell, and Bell [20]. The present implementation of the target machine editor runs on top of the Extend^{TM} simulation package on a Macintosh. The following features are implemented:

- ability to graphically describe small irregular architectures or easily describe large regular architectures,
- graphically create shared memory, tightly coupled distributed memory or loosely coupled distributed memory architecture descriptions,
- describe system specific information by entering the information in dialog boxes which are logically attached to the graphical icons,

- graphics to text transformation for interface with other PPSE tools,
- ability to save and edit graphical descriptions of systems.

In general, a number of system level blocks (processor, memory, bus, and switch) are kept in a library. The user selects blocks from the library and enters specific information by double clicking on the block and keying in the block specific information (like processor speed or memory size) in the fields of the dialog window. A global information block, called the Topology File Generator must be present in all system descriptions. This block contains information which is global to the system. In the case of large regular architectures, the Topology File Generator block may be the only block necessary. All necessary information can be entered in its dialog.

Once the software and hardware descriptions have been gathered, the software designer should determine the optimal assignment of software processes to processors. A scheduling package, called Mac-scheduler performs an automated mapping of the software onto the hardware. Mac-scheduler maps program modules represented as nodes in a precedence task graph with communication (a proposed transformation of the ELGDF design file) onto arbitrary machine topologies and gives an allocation and ordering of tasks onto processors. It produces as output a Gantt chart, providing easy visualization of the allocation of the program modules onto the target machine processing elements, and the execution order of tasks allocated to each processing element. The Gantt chart consists of a list of all processing elements in the target machine. For each processing element, the Gantt chart shows a list of all tasks allocated to that processing element, ordered by execution time, including task start and finish times. Several mapping heuristics are included in the package. Frequently the best heuristic for a particular problem must be determined experimentally. Mac-scheduler allows the program designer to experiment with several mapping heuristics in order to determine the the best choice for process to processor assignments.

A desirable output from the PPSE design is compilable source code. A glue code module has been developed which takes the PPSE software design, code fragments written in a specific programming language, and a hardware description as input and produces machine specific source code as output. The source code can then be compiled on parallel machines. One of the primary problems with manual generation of parallel programs is the lack of portability of the finished code due to the architecture and vendor specific parallel programming primitives. Parallel programs, like sequential programs, frequently need to be transported across architectures. The glue code module allows the parallel program designer to design parallel programs without specifying architecture specific synchronization and communication primitives (such as locks on a shared memory system or message passing primitives on a distributed system). The glue code module automatically adds these primitives to the code fragments according to the specified design in the ELGDF editor.

2.2 Overview of the General Application: Automated Analysis of Cloud Properties

Satellite data has a wide range of good uses. Microwave radiometer measurements have been used to study yearly and seasonal changes in polar sea ice cover [18]. Visible and infrared radiometer measurements have been used to study the oceans, vegetation, geology, and the atmosphere. Climate models based on satellite data have been appearing frequently in the news as researchers attempt to determine the sensitivity of the climate to changes in external conditions, such as solar radiation or the atmospheric carbon dioxide amount [12]. Images produced from satellite data are shown in figures 3 and 4 (at the end of the report). Figure 3 is a scene of data over the Atlantic Ocean. In figure 4, a scene over Africa and Spain, the data has been sampled so that every third pixel is shown.

The Spatial Coherence Method is the basis of an automated procedure to determine cloud cover and cloud-free radiances of specific regions using high-resolution infrared scanner data from satellite passes over approximately $(1000 \text{ km})^2$ regions. The determination of cloud cover is useful to Atmospheric Scientists for, among other things, deriving the Earth's radiation budget. The determination of cloud-free radiances over the ocean is useful to Oceanographers because it leads to the derivation of sea-surface temperatures while allowing a very small amount of error due to the atmosphere.

The following excerpt from Coakley and Baldwin [5] explains some of the more basic reasons for creating an objective analysis scheme for deriving cloud properties:

For the better part of a century we have suspected that through their influence on the Earth's energy budget, clouds play a major role in climate dynamics. Yet what role they play remains a topic of lively debate. The debate will undoubtedly continue for years to come as there is yet no climatological data set which boasts objectively derived, quantitative estimates of clouds and their properties. Without such estimates, progress towards understanding the role clouds play in the climate system is stymied.... There is hope, however, that objectively derived estimates of clouds and their properties might come from satellite data. Recent work would suggest that this hope will shortly become reality.

Based on the Spatial Coherence Method, their analysis scheme is able to derive an estimate for fractional cloud cover and cloud radiative properties for a fixed scene of data. AVHTST is the first pass of a series of routines which implement this method. Currently, one of the major drawbacks to AVHTST is the excessive runtime necessary for analyzing large amounts of data. The Spatial Coherence Method is now being used in a number of cloud studies and is being proposed for use in many more in the next few years. It is currently in the process of being streamlined to achieve higher data throughput rates.

According to Coakley and Baldwin [5], through the use of properly designed parallel processors and associated software, an operational analysis scheme based on the Spatial Coherence Method seems entirely feasible. Thus, the Spatial Coherence Method is now a useful method for researchers wishing to automate the analysis of cloud properties for small scale studies, but to be a truly operational method - a general purpose climatology tool - substantial reductions in processing times are necessary. Such reductions could be achieved through good parallel design of the most time consuming components. The next section describes AVHTST in more detail.

2.3 AVHTST

AVHTST first takes a scene of data as input, divides it into frames (approximately covering a (250 km)^2 region and consisting of a 64 * 64 array of pixels), and calculates local means and standard deviations of 1024 2*2 pixel arrays. Figure 5 shows a plot of the local means vs local standard deviations for a frame (note: figs 5,6 and 7 are at the end of the report). For frames with clear and cloudy sky pixels, the plotted points will ideally resemble an arch. The points at the foot of the arch at higher valued means (right foot) represent cloud free pixels while the points at the left foot of the arch represent cloud covered pixels.

The first real task is to automate the process of identifying the feet of the arches. Points with small standard deviations at the feet of the arches are associated with cloud free and completely cloud covered regions. The algorithm to select the feet of the arches simulates a number of decision processes in order to determine which points belong to the feet of the arch and which belong elsewhere. Figure 6 graphically shows two of the steps involved. After a threshold test is performed to remove points with high standard deviations, a frequency distribution of the remaining points is generated. The next major step is to calculate a probability distribution as a function of the mean radiance. Finally, as shown in figure 7, only the points at the feet of the arch remain.

The second part of AVHTST examines the pixels within each (60 km)^2 subframe to determine which pixels belong to a certain foot. The following steps are performed:

- preserve statistical properties for pixels in feet,
- preserve statistical description of radiance field,
- note if no feet exist in the subframe,
- if no feet exist, interpolate the cloud properties from the nearest neighbor subframes,
- calculate and preserve several statistical properties,
- separately identify points within feet for each channel.

To obtain cloud free radiances:

- compare the channel specific means and standard deviations for certain spots and determine there is agreement in identifying a particular foot as representing a cloud free region, the radiance distributions comply with one another if the difference between their means is less than the root-mean-square of their standard deviations.
- as a second check, perform a threshold test with the visible channel data.

Finally, all the data which is to be preserved for further processing is saved to file. AVHTST reduces the data volume by a factor of 15 while preserving most of the important properties for further processing.

3. Parallel Design of AVHTST

The following sections describe the use of PPSE in parallelizing AVHTST. First, the ELGDF design editor is used to enter a parallel program design. Next, several potential target machines are described using the target machine editor. The scheduling tool is then used to experiment with program schedules and examine possible speedup due to parallelization. Glue code is briefly discussed, followed by a section on theoretical results.

3.1 ELGDF Design

A simplified top level description of a parallel version of AVHTST might look as in figure 8.

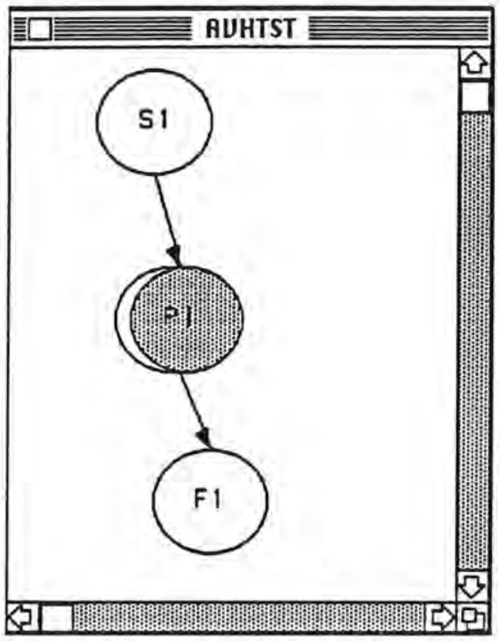


Figure 8. Top level ELGDF description of the parallel solution.

Node S1 would be responsible for all data input from file. Distinct problem instances would be passed to replicated node P1. Each replicated node of P1 can execute independently. Node F1 would collect the results and write answers to file. For some problem instances this will be all the parallelism necessary. However, the task is to design a parallel program which is able to handle all problem instances. Different parallel solutions are required to properly handle different geometries in the input. The input may consist of many scenes of data or it may consist of one extremely large scene. To hardwire a parallel solution to this problem without considering the input is a mistake. Each replicated node of P1 should be expanded into a smaller grained parallel solution. Figure 9 shows this expansion.

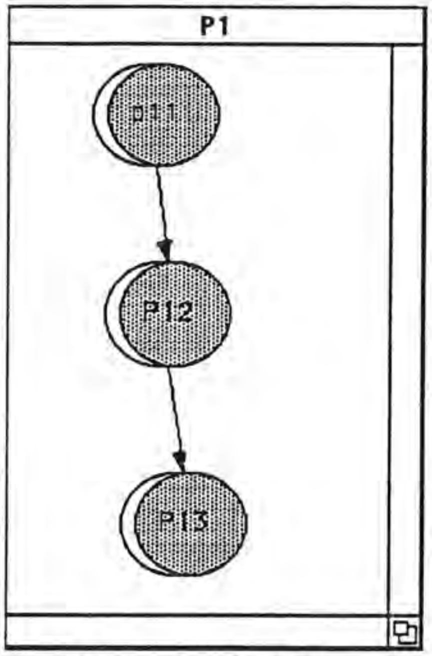


Figure 9. Expansion of node P1.

In node P11, all the means and standard deviations for each frame and each channel could be calculated simultaneously. If the data consists of 16 frames and 5 channels, the potential parallelism will be 80 concurrent processes. P12 will use these calculations to attempt to determine the subframe properties. The number of possible concurrent processes will equal the product of the number of frames and the number of subframes. However, if this level of granularity is smaller than necessary, the number of concurrent processes could be changed to equal the number of frames. Node P13 will need to determine for each subframe whether a default estimate of the subframe property is necessary. If so, then the subframe properties will be interpolated from the nearest neighbor subframe properties. An expansion of node P13 is shown in figure 10.

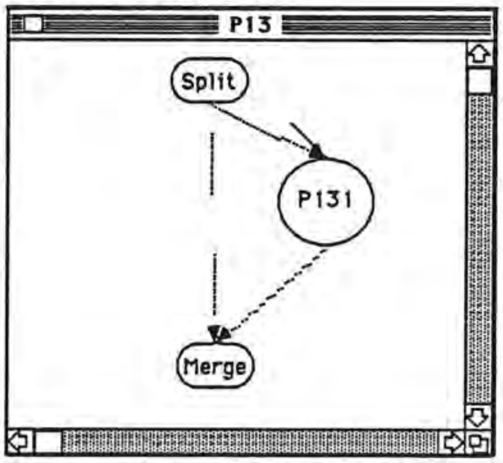
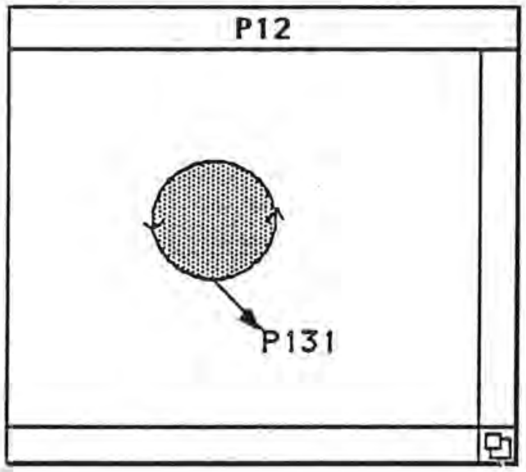
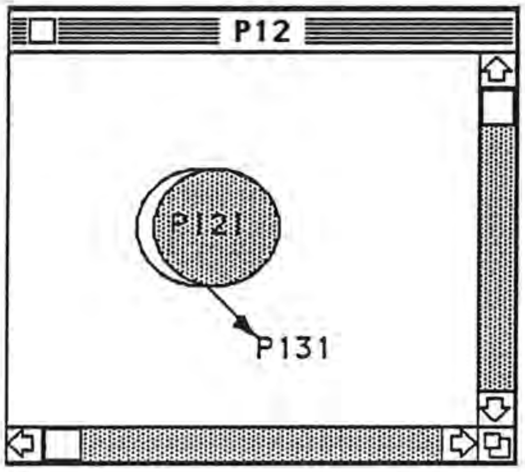


Figure 10. Expansion of node P13

As mentioned above, node P12 could be designed to concurrently perform calculations by the number of frames or by the number of subframes yielding a different level of granularity in each case. Figure 11 shows the expansions of P12 for each case. If the expansion uses a for loop, then we are sequentially making subframe calculations, but concurrently making frame calculations. If we use a replicator loop instead, then we are concurrently making calculations for all subframes. The data calculated in P12 is necessary for P131. In P131, subframes which couldn't be determined must be filled in with default estimates interpolated from nearest neighbor subframes. First a check is made to see if good data exists. If not, data must be fed in from previous calculations. This is shown in the expansion of P13 (figure 10) at node P131 by the bridge arc being fed in from P121.



(a)



(b)

Figure 11. In a, the expansion of Node P12 is done using a for loop for sequential execution. In b, another replicator is used in order to perform smaller grained concurrent analysis.

3.2 Target Machine

Two sample target machine descriptions were entered. The first for the OSU Sequent Balance 21000. The graphical input is shown in figure 12.

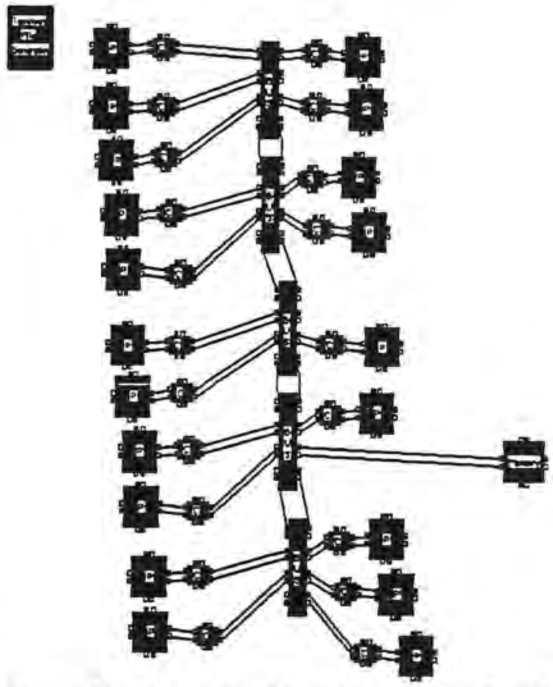
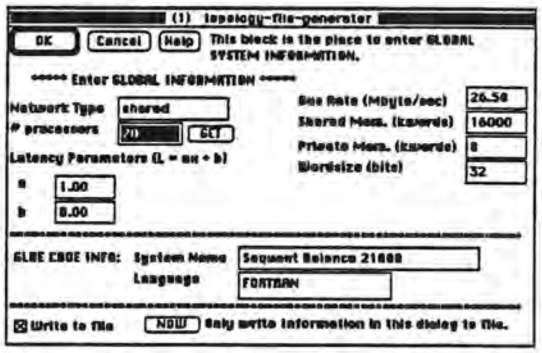


Figure 12. Target machine description of a 20 processor Sequent Balance.

The Topology File Generator block for the Sequent Balance target machine was given input as shown in figure 13.



(1) topology-file-generator

OK Cancel Help This block is the place to enter GLOBAL SYSTEM INFORMATION.

***** Enter GLOBAL INFORMATION *****

Network Type	shared		Bus Rate (Mbyte/sec)	26.50
# processors	20	SET	Shared Mem. (kwords)	16000
Latency Parameters (L = ax + b)			Private Mem. (kwords)	8
			Wordsize (bits)	32
a	1.00			
b	8.00			

GLUE CODE INFO: System Name: Sequent Balance 21000
Language: FORTRAN

☒ Write to file NOW Only write information in this dialog to file.

Figure 13. Dialog box of Topology File Generator block.

The generated Topology File is partially shown in figure 14. The topology file contains three main sections:

- general information

- block identification
- adjacency list

The topology file contains a textual description of the target machine. Potentially, other PPSE modules will be able to use this information to advantageously create a running parallel program which is optimally fit to the target machine. For effective use of parallel systems, it is essential to obtain a good match between algorithm requirements and architecture capabilities [13].

```
<$$START TARGET$$>
*procs 20
*caches 20
*buses 5
*switch 0
*links 0
*ethn 0
*mems 1
network   shared
latency_a 1
latency_b 0
busrate   26.5
sharedmem   16000
privmem   8
sysname   Sequent Balance 21000
language   FORTRAN

Block identification
processor 18
pspeed 18 1
processor 1
pspeed 1 1
   •
   •
Adjacency List (self,inputs to self from...)
18 19
1 2
16 17
14 15
12 13
   •
Adjacency List (self, outputs from self to...)
18 19
1 2
16 17
14 15
   •
21 44
23 46
45 24
<$$END TARGET$$>
```

Figure 14. Topology File

A description of the second target machine description is delayed until the next section. The idea there is to first examine the dependency graph of a particular instance of the problem and then design a fictitious target machine that might match the algorithm nicely.

3.2 Scheduling

From the ELGDF design, we need to form a dependency (or task) graph. The dependency graph is the required input for the scheduling algorithms in the scheduler tool. The program design is transformed into a dependency graph. The schedule is an assignment of processes to processors. In order to form a dependency graph for this problem, actual parameter values will need to be known. As an example, 2 channels of data for a 2 by 2 scene are specified as input parameters. The dependency graph with relative runtime estimates is shown in figure 15. In each bubble, the number on top is a node identifier and the number below is relative runtime for the specific code fragment which the bubble represents. The arcs represent communication time. In this example, these values have all been set to unity, signifying that communication time is constant.

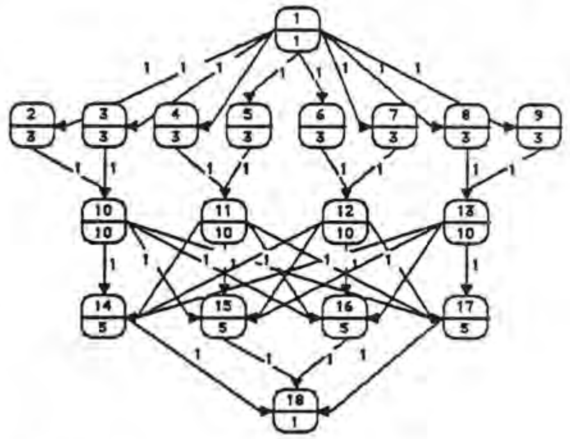


Figure 15. Dependency graph.

It is important to note that this dependency graph is for only one simple instance of the problem - when we have one scene of data with 2 frames in each direction. The typical scene of data with 4 scenes in each direction would have a different dependency graph. Its graph would contain at least 66 nodes (as opposed to only 18 for this example) and the edges between the third and fourth level would correspond to a nearest neighbor dependency (in the actual geometry of the problem).

With the PPSE scheduler software, a gantt chart and speedup graph can be estimated for this problem instance. The speedup curve is shown in figure 16 and the gantt chart in figure 17.

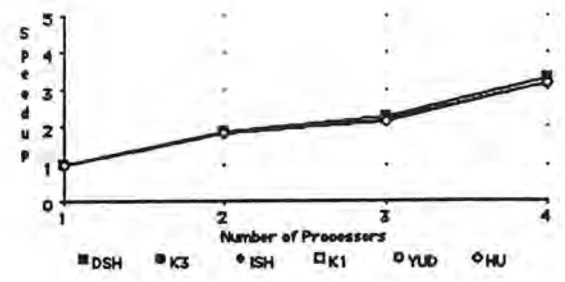


Figure 16. Speedup curve

The speedup curve shows that with two processors, we get a speedup of two. For 4 processors, the speedup is slightly more than 3. The speedup graph also shows that the choice of scheduling algorithm makes very little difference. Although the current scheduler does not take full advantage of the architecture information supplied in the topology file, the scheduler tool provides useful information to the parallel program designer. We are able to roughly determine the useful number of processors for a given application from the speedup curve, and we are able to compare different parallel solutions to a problem by comparing the gantt charts produced by each program description.

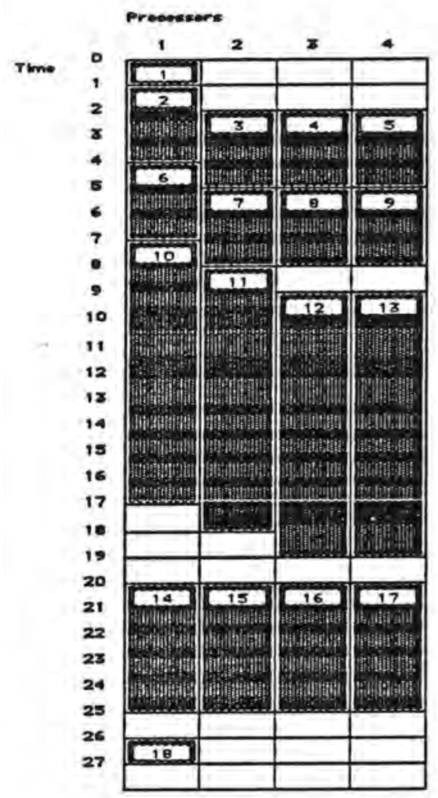


Figure 17. Gantt chart.

Across the top of the gantt chart is a column for each processor. Down the column is a schedule of processes which will be assigned to the particular processor. Blank space in a column signifies that the processor is idle (although communication from another processor may be taking place).

To a naive parallel programmer, an ideal computer for this problem (2 by 2 scene) might exactly match the dependency graph. A fictitious system meeting this description could be described with the target machine editor. Such a system is shown in figure 18.

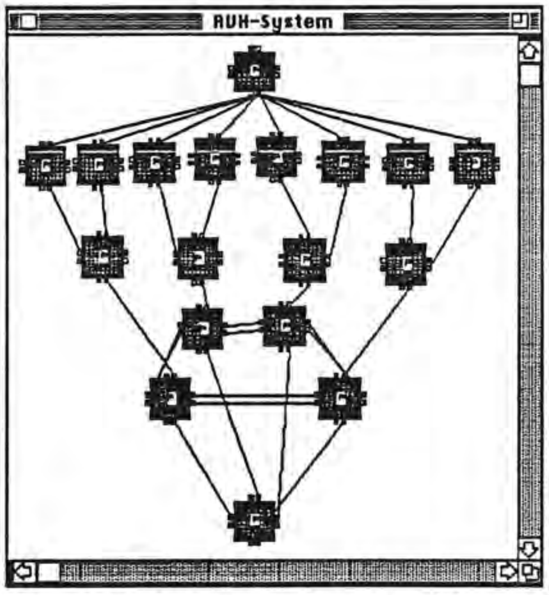


Figure 18. A hardware topology that might perform well for 2 by 2 scenes.

When this system's Topology file is input to the scheduler, we find that the system will not perform much better than a 4 processor fully connected system. The best schedule produced using this topology reveals that the program will complete in 23 time units. With 4 processors, the program completes in 26 time units. The useful processor point for this problem is 8. We can achieve the same or better results with 8 processors than we can achieve with more than 8 processors. A look at the speedup curve for this problem reveals that the curve reaches maximum at about 4 processors (see Figure 18a). Devoting more processors to the problem will not provide any benefit. The scheduler is extremely useful for determining these types of issues.

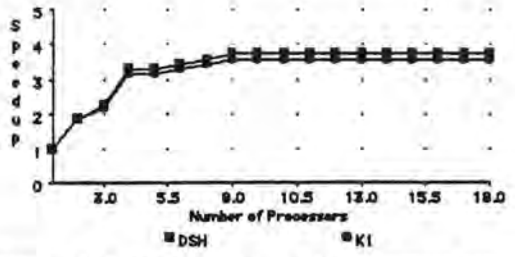


Figure 18a. Speedup Curve for alternate topology reveals that using more than 4 processors yields little benefit, and more than 8 processors yields no benefit for this instance of the problem.

For different input, another architecture possibility would be to use a distributed system with n processors, where n is equal to at least twice the number of scenes being analyzed. Each pair of processors would independently analyze a scene of data and write results to a common file system. The choice of *best* architecture depends entirely on the input. Analyzing 600 scenes would be faster on a distributed system with lots of processors. Analyzing 1 large scene would be faster on a more tightly coupled system whose topology closely matches the needs of the dependency graph.

3.4 Glue Code

Currently, PPSE does not have a module which can transform FORTRAN code fragments and an ELGDF design into a working parallel program. The glue code step was accomplished manually and is discussed in section 4.2. Recommendations for FORTRAN glue code are given in section 5.7.

3.5 Theory

At the top level of design detail, independent processors can perform calculations on independent scenes. Sequential AVHTST runs in time O(n), where n is the number of scenes being analyzed. This can be shown when actually running the program (see Table 1). Analysis of one scene takes 10 minutes on one Sequent processor. Two scenes take 20 minutes, and 50 scenes take 500 minutes. Parallelizing the program so that all scenes can be analyzed simultaneously creates a program with constant runtime (O(1)) if the number of scenes being analyzed is less than the number of processors. We will use the notation T(p,n,A) to express the execution time of algorithm A with input size n on p processors [11]. If we use twice as many processors as scenes, we should be able to achieve:

$$T(2n,n,AVHTST) < T(1,1,AVHTST)$$

That is, multiple scenes can be processed in less time in parallel than 1 scene sequentially. In the next section, an implementation of the program which achieves this goal is discussed.

4. Parallel Implementation of AVHTST

A parallel version of AVHTST was constructed which generally adheres to the PPSE design. The target machine was a Sequent Balance 21000. Parallelism was achieved using compiler directives which are preprocessed to generate a parallel version of the code acceptable to the compiler [17]. Data type and data dependency analysis were performed manually.

4.1 Preliminary Steps - Porting

In order to get parallel FORTRAN code to work on a parallel machine, it may be necessary to first port the sequential version (if one exists) and learn the tricks of the machine and the compiler. FORTRAN, unlike C, is typically difficult to port. The three most common problems are as follows:

- Format of binary input/output data - different machines maintain different internal formats for storage of FORTRAN unformatted (binary) data. When processing data in a distributed manner, each machine must be able to receive data from a remote machine and correctly handle differences in binary data format. Differences involve word size (ie. 64 bits for Cray X-MP, 32 for Sequent), floating point format (ie. VAX doesn't adhere to the IEEE standard for floating point), and record length byte counts (ie. VAX maintains a 4 byte header to all records, Sun maintains 8 byte headers and trailers, Sequent doesn't use byte counts). Many machines also maintain different byte orderings. This becomes apparent after binary file transfer occurs. Code that compiles may not work because of these input format problems.
- Input/output statements - most machines allow I/O extensions to the FORTRAN 77 standard and much of the actual code in existence uses these extensions. However, they typically aren't uniform over a range of machines. OPEN, READ, WRITE and INQUIRE statements typically need to be translated in some way from machine to machine.
- Adherence to FORTRAN 77 standard - most compilers don't strictly adhere to the FORTRAN 77 standard [19]. The FORTRAN 77 standard provides that variables internal to a subroutine are not guaranteed to be saved between successive calls to that subroutine unless the variable is declared global in a SAVE statement. If this part of the standard were enforced, a large fraction of all programs written would immediately cease to function

[19]. The Sequent Balance FORTRAN compiler does enforce this part of the standard. This creates a problem for code written on less strict compilers like the VAX compiler which is to be ported to the Sequent.

To get AVHTST ported to the Sequent Balance, the binary I/O problem and the I/O statement problems needed to be addressed. AVHTST takes binary input and produces binary output. The binary output is typically processed on a Sun Workstation. Figure 19 shows the steps necessary for running AVHTST on a parallel machine in a production mode.

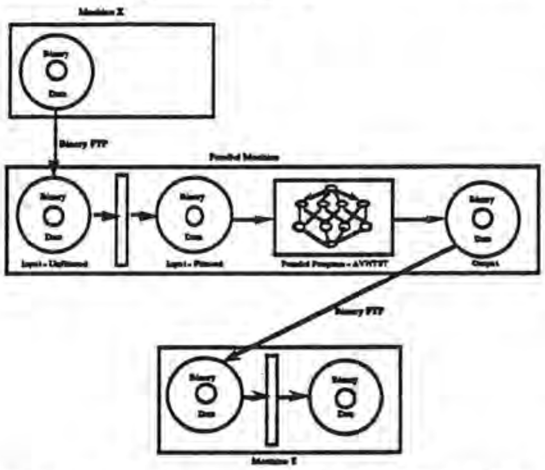


Figure 19. Production mode of AVHTST on a Parallel Machine. Input to AVHTST would typically come from another machine (Machine X). The data would be transferred to the parallel machine via binary FTP. The data then needs to be transformed into the parallel machine's desired input format. AVHTST processes the data and produces a binary output file. This file would be sent to another machine for further processing (ie imaging). Once arriving on Machine Y, the binary data must again be transformed into a form acceptable by Machine Y.

4.2 The Working Version

Once the porting problems are handled, the problem of creating compilable source code can be dealt with. PPSE will eventually attempt to automate this process for FORTRAN code, but for now it must be done manually. The simplest way to create replicated code from FORTRAN DO loops is to recode the loops as C$DOACROSS loops. An example of this is shown in figure 20.

```
C$doacross share(ip,kcor,x,y,icor,nxfrm,kchan,ichan,xx1,qywt,
C$&   xper,xlev,nplt,iplot,nywt,filenam,
C$&   ywt,nper,npicx,nslin,nflin,iyun),
C$&   local(nspic,nfpic,k,ich,i,xmin,xmax,xmean,
C$&   ndisp,iywt,numfeet,lip,x1,x2,slope,mscale,l,x3,
C$&   n1,n2,n3,np,xrnd,l1,
C$&   hplt,xplt,zplt,iscale,disp,
C$&   ndiv,mdiv,nex,nsig,ipk,npicy,histm,
C$&   lblx,llblx,klblg,lblglen,lbll,lbllen,
C$&   hist,xhist)

      do 300 ixun = 1, nxfrm
        print*,'ENTERED PARALLEL LOOP - PROCESS: ',ixun
             :
             :
300   CONTINUE
```

Figure 20. Example code

Sequent requires variable analysis for all variables within parallel portions of the code. This was accomplished manually but perhaps could have been done using the static analysis code analyzer being developed at Portland State University as part of the PPSE project. All sections specified as replicated in the PPSE design were coded with C$DOACROSS loops. Thus the PPSE design was implemented. All variables that were passed on data arcs in the design were implemented as shared variables. When necessary, access to shared variables was controlled with LOCK/UNLOCK statements.

4.3 Timing Comparisons

From section 3.5, we had hoped to achieve the following result:

$$T(2n,n,AVHTST) < T(1,1,AVHTST).$$

Using 2n processors to process n scenes of data will take less time than processing 1 scene sequentially. For real machines, this holds as long as 2n is less than or equal to the number of processors for the particular target machine. A sequent Balance with 16 processors should be able to process 8 scenes in parallel faster than 1 scene with a sequential program. Table 1 illustrates the timing results achieved with the parallel version of AVHTST.

Data Size	Sequential Time	Parallel Time	Processors Used	Speedup	Efficiency
1	10:32	7:02	2	1.49	0.745
2	21:06	7:03	4	2.99	0.748
3	31:39	7:08	6	4.44	0.739
4	42:10	7:06	8	5.93	0.742
5	52:45	7:10	10	7.36	0.736
6	63:20	7:18	12	8.67	0.723
7	73:56	7:15	14	10.18	0.728
8	84:32	7:25	16	11.40	0.712

Table 1. AVHTST timing data. Data size is number of scenes. Processors used is the number of processors used to achieve the parallel time shown. Speedup is defined as sequential time divided by parallel time for the same sized problem. Efficiency is equal to the speedup divided by the number of processors used.

The significant result is that for a problem of size n, as long as at least 2n processors are available, the complexity of the algorithm is O(1). The parallel solution has reduced the order of complexity from O(n) --> O(1).

5. Subjective Analysis of PPSE tools

Overall, the PPSE tools (even in their conceptual state) were useful for parallelizing the satellite data analysis problem. The tools provide a framework for good structured design of parallel programs. An essential feature is the splitting of the program design from the architecture description. This idea should help program designers to produce portable parallel code. The scheduler tool provides useful information to the programmer and the glue code module should prove to be a great time saver in the future. The rest of this section deals with proposed additions, enhancements and clarifications which may help to strengthen the PPSE tools.

The parallel solution to AVHTST is simple. Creating a parallel version of AVHTST is extremely difficult and time consuming to do manually, and the resulting program will likely be bound to a specific architecture. PPSE should offer support to problems which have a conceptually simple parallel solution, but unload a time consuming and difficult task of redesign and restructure on the parallel programmer. In this problem, the amount of independence in the resulting data structure is so large that the real concern of the parallel programmer is not whether to parallelize but where to parallelize. The following sections discuss possible ways to improve PPSE in order to better deal with problems like AVHTST.

5.1 Visualization of the conceptual class of the parallel program.

While ELGDF allows design specification for AVHTST, it doesn't help the parallel program designer write a parallel program. It's like being given a dictionary of a foreign language and attempting to make coherent sentences without first learning the rules of grammar. ELGDF should provide a top level structure which adheres to a parallel programming conceptual class. Parallel programs fall into several conceptual classes [4]. Three main conceptual classes are result parallelism, agenda parallelism and specialist parallelism. These three conceptual classes can be programmed with the following three parallel programming methods: message passing, distributed data structures, and live data structures. The following discussion addresses the conceptual class in which AVHTST most naturally falls - the result parallelism class.

In result parallelism, the programmer must *visualize* the resulting data structure which the program produces as a finished product. The result can be broken into components which can be computed concurrently. Some components may rely on others (and thus we have dependencies), but other components may be completely independent. The highest level concurrent processes in this model are all responsible for producing part of the final result. In AVHTST the resulting data structure is an array. The following diagram is a high level visualization of one way to design the program:

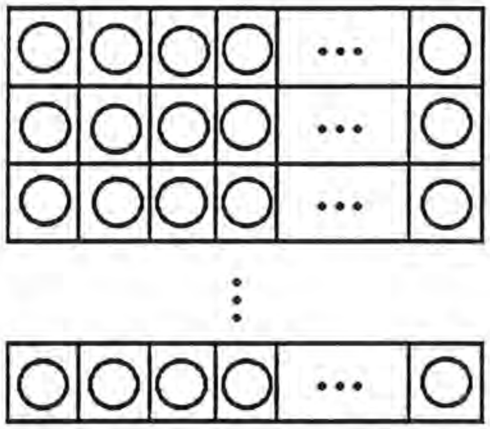


Figure 21. Result Data Structure partitioned into sections which can be computed concurrently.

The resulting data structure has been partitioned into sections whose computations can all proceed in parallel. Each circle represents a process which computes the corresponding section of the result. The dependencies between the different section will need to be shown at a lower level of detail. This type of higher level diagram could easily be animated to show the progress of the parallel program as it computes the result (idea from [4]).

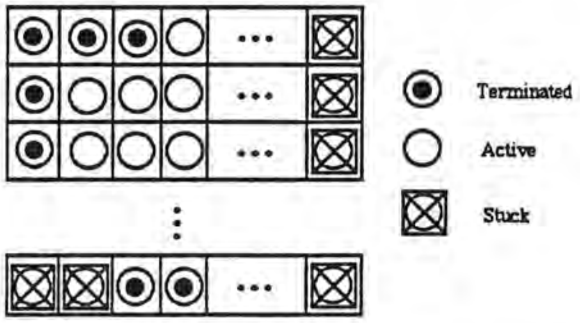


Figure 22. Possible animated scene of the parallel computation of the resulting data structure.

Each of the circles (processes) in the resulting data structure could be expanded into an ELGDF description of the necessary calculation steps.

5.2 Specify an input parameter file.

An input parameter file would help generate accurate Task Graphs, schedules and glue code while allowing the programmer to generalize the ELGDF design. The ELGDF editor should allow the programmer to enter variable parameters in the design for things like number of replicated processes and upper ranges of loops. A variable template should be created by the editor which is filled in by the user before generating task graphs, schedules or glue code. The input parameters would specify possible problem sizes.

For example, with AVHTST we would like to be able to specify a variable number of scenes of variable size, variable pixel array sizes, variable number of frames and subframes, and variable size of frames or subframes. With different input parameters, the schedulers should parallelize around different parameters. In some cases, parallelization by scene might be effective. In other cases, parallelization by channel, or by frame or some other parameter might be effective.

5.3 Data structure design should be more prevalent in the program design.

While the program flow is an important concept, the parallel data structures involved in the parallel program are equally, if not more, important. The ELGDF storage construct is woefully inadequate for the design of real parallel programs. Most programmers would like assistance in visualizing the data structures involved in parallel computation. For example, the concept of a distributed data structure is one that can be effectively implemented on both shared memory or distributed memory machines. A distributed data structure is logically shared but may exist in different physical memories. The programmer should be able to design a distributed (or any kind) of data structure and then design processes which work with the data structure.

5.4 Need a formal syntax specification of ELGDF.

The following list of examples illustrate the problem:

- In a FOR loop structure the control variable and ranges are input in a dialog box. If the next level down is a code fragment, is the loop control from the dialog automatically appended to the code fragment or should the programmer specifically include the control line and associated (syntactic characters ?). What if the control and ranges specified in the dialog box don't match the control and ranges specified in the code fragment?

- Replicated code may need different input for each node, and the input may not correspond exactly with the control variable. For example, replicated code may need to read and write according to a file name which was input to the specific code segment. Where should the input be specified in an ELGDF diagram. Another example is the representation of a queue or stack structure which provide input to a number of replicated processes. It is not clear how to represent these structures in ELGDF.

- The proper usage of storage construct is completely unclear. Do they need to be included at all in the design specification? Only for synchronization?

A clear specification for the usage of the ELGDF constructs needs to be formulated.

5.5 User/PPSE Responsibilities

The user should not have to specify bytes on arcs or run times of code fragments. Bytes on arcs should be estimated from the input parameter file and the topology file. Overloading the user with details will not make the task of parallel programming easier. Estimating the run time of code fragments is a more difficult problem, but one that should be dealt with by PPSE and not the user. With an input parameter file and a topology file, an estimate of the run time of the code fragment should be possible. The original estimates should occur when the user first attempts to generate a schedule and the values of unaltered sections of code should be saved thus reducing the run time for additional schedule generations.

5.6 Theoretical Obstacles with Common Structures

Presumably, ELGDF common structures like while loops and repeat-until loops will need to be transformed into task graph elements. They can't be expanded because the expansion of a conditional loop may be infinite. They need to be dealt with as singular computation units, but doing this will prevent any further hierarchical breakdown of the conditional loop program segment. Thus, a while loop as the top level bubble in an ELGDF description may mean that the entire program must be specified in a programming language unless a method of merging textual descriptions with iconic descriptions is worked out. Such methods are not presently available.

The use of conditional while loops will also present a difficult theoretical problem to schedulers.

First of all, it is impossible to predict the run time of a code fragment which contains a conditional while loop. In fact, it is impossible to even know if the code fragment will terminate. We can't require that conditional loops not be allowed; that would be absurd. A programming language with no conditional loops can't represent all possible programs and would be less powerful than the Basic language even. The use of common structures needs to be clearly explained and a method of transforming them to task graph and glue code elements should be thoroughly worked out.

5.7 FORTRAN Glue Code

Creating FORTRAN glue code may turn out to be slightly more difficult than for other languages (such as C) because many real FORTRAN code fragments will lack structure and strict adherence to the FORTRAN standard. There are two basic options when creating glue code:

1. translate to a portable parallel language.
2. translate to a machine specific language.

With the first option, there is FORTRAN-Linda, FORTRAN-8X, or packages such as Schedule (developed at Argonne National Labs). FORTRAN-Linda is not yet released and FORTRAN-8X is not implemented on many machines and does not support multitasking. Schedule is currently available for a number of parallel machines, but by the developer's admission, has a limited lifetime [7].

A tradeoff exists between the two options. The second option will probably yield higher performance code at the expense of code comprehension. This may be a severe drawback in the debugging phase. Also, developing separate glue code for each parallel machine may prove to be an unwieldy process.

The purpose of PPSE is to demonstrate the feasibility of the idea rather than demonstration of high performance. For this reason, we should pursue the first option of translating to a portable parallel language. Specific vendors should be responsible for implementing the portable parallel language into their system.

If we choose to implement the first option immediately for FORTRAN, then we must go with Schedule. Since Schedule programs develop naturally from data dependency graphs, this may be the best choice anyway. If the development of FORTRAN glue code is to be delayed for several months, then waiting for the release of FORTRAN-Linda may be the best move.

5.8 ELGDF Representation of Iterative Relaxation Class of Problems

According to Finkel [11], most distributed algorithms fall into one of several classes. Iterative relaxation is somewhat analogous to result parallelism. The data space is divided into adjacent regions which are then parcelled out to different processes. Each process carries out activities local to its region, communicating with neighbors when necessary. This category includes the solution to PDEs and graph problems like finding a minimum spanning tree. Figure 23 illustrates an iterative relaxation solution to AVHTST.

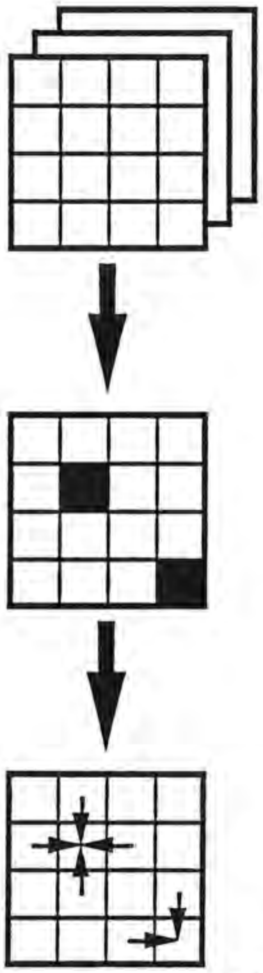


Figure 23. The data space is split by frame and by channel (top). All possible frame calculations are made - black squares indicate missing data (middle). Missing values are interpolated from nearest neighbor results (bottom).

In fact, this is exactly the intended design of the problem as specified in the data dependency graph (Figure 15). The original ELGDF design (section 3.1 and Figures 8-11) is intended to represent this same idea. However, in order to transform the ELGDF description of the algorithm to the proper task graph, more information needs to be present in the design. The representation of nearest neighbor communication does not seem possible with either the replicator or fan structures. The replicated structure would need to be bypassed in this case - the design needs to more closely resemble the task graph. But if program designers must frequently bypass the ELGDF convenient structures in order to properly represent the program design, the utility of these structures will be lost; they will become an unnecessary and inconvenient intermediary step towards the final intended design. Although a replicated loop seems to be an intuitive choice for representing this program design with ELGDF, it doesn't capture the necessary amount of information to properly represent the design. This is a common problem with ELGDF structures: convenience has been placed ahead of information content.

6. A Plan for integration of the PPSE tools

Conceivably, the ELGDF editor, task grapher, scheduler, and glue code module could all exist within the same shell program. The current Target Machine editor is based on a commercial product and wouldn't integrate as easily. The two programs could be run simultaneously using multifinder or a new target machine editor should be developed which more easily integrates into the system. A possible sequence for designing a parallel program could adhere to the following steps:

1. From within the Target Machine Editor program, enter the descriptions of several target machines. Generate topology files for these machines and exit the program.

2. Enter top level design of program based on the most natural parallel program design method. This might correspond to the resulting data structure, the agenda of activities, or specialist parallelism. From the top level design, enter the ELGDF description of the program.

3. Enter constant input parameters which correspond to the generalized ranges for ELGDF convenient structures (ie. loops, replicators, etc.). These input parameters will be needed for construction of the task graph and schedules.

4. Generate a schedule. User will need to specify the location of the Topology file and Input Parameter file. The task graph should be automatically generated and used as input to the scheduling routines. The user will be interested in viewing schedules as a means of comparing different program designs, different input parameters, and different target architectures. The user may also want to view speedup charts, and possibly a graph showing the correspondence of runtime to problem size (problem size will be a function of the input

5. Generate Glue Code. The user may want to specify variables around which to parallelize. Using AVHTST as an example, I might want to calculate all scenes in parallel, or I might want to calculate all channels concurrently, or I might want to calculate all x-direction frames concurrently. The program design should be general enough to handle all these situations and the gantt chart should have given me a hint about which parts of the program should compute concurrently. The glue code module must be told how to parallelize the program. The decision must be made by the user or by PPSE.

6. Transfer the Glue Code to the Target Machine, compile, and run.

These steps are not currently possible. While this paper is not a requirements analysis study, the tool designers should pay close attention to the needs of parallel programmers. The needs for creating a parallel FORTRAN program may not be the general needs, but they are important to an extremely large class of potential parallel programs.

7. Conclusion

In an article about a programming environment similar to PPSE [3], the authors state, "..we believe no one should be allowed to publish an article about their programming environment until it has been used by some threshold number of users." PPSE has not yet been used by an acceptable threshold, but shows extreme promise as a viable environment. The designers of PPSE realize that the system must be iteratively refined and are willing to implement their ideas into usable software tools in order to facilitate the necessary interaction between themselves and parallel programmers. The availability of these tools allows useful testing of concepts.

Clearly, PPSE is on the right track. More research needs to be done in the area of graphical program description. ELGDF has some problems. Specifically, the transformation of ELGDF graphs into dependency graphs is a difficult problem because an information void must be dealt with. Current ELGDF structures can not handle the necessary level of detail for proper transformation into accurate task graph representations of the program. On the one hand, ELGDF needs to capture more of the programmer's intent, but not at the expense of overburdening the design process with details which could properly be left until further refinement is necessary. Also, many of the structures and paradigms which are important to parallel programming are difficult or impossible to represent in ELGDF. The design description language should not inhibit program designers from designing programs, but rather facilitate the process. PPSE has this goal in mind.

The prototype parallel version of AVHTST was successful. More test cases should be worked through PPSE in a manner similar to that described in this paper. There are two benefits to this. The first is a benefit to PPSE. Whether successful or unsuccessful, the more test cases which are run through PPSE, the better the final product will be. Unsuccessful test cases will give insight into ways to improve PPSE into a more robust set of tools. Successful test cases will be forward steps toward proving a set of important concepts. The second benefit is experience with parallel programming. Designers of parallel programming environments need to have a large amount of this experience in order to understand the problems, frustrations, and complexities involved with the task of designing parallel programs.

References

1. R.G. Babb, Programming Parallel Processors, Addison-Wesley Publishing Company, Inc., New York, 1988.

2. R.G. Babb, D. DiNucci, "Design and Implementation of Parallel Programs with Large Grain Dataflow, in The Characteristics of Parallel Algorithms, L. Jamieson, D. Gannon, R. Douglass (editors), MIT Press, Cambridge, Mass.,

3. J.C Browne, M. Azam, S. Sobek, "CODE: A unified Approach to Parallel Programming", *IEEE Software*, Volume 6, Number 4, July, 1989.

4. N. Carriero, D. Gelernter, "How to Write Parallel Programs", Report #SCA-140, Scientific Computing Associates, Inc., New Haven, CT, November, 1988.

5. J. A. Coakley, D. G. Baldwin, "Towards the Objective Analysis of Clouds from Satellite Imagery Data", *Journal of Climate and Applied Meteorology*, Vol 23, No 7, July 1984.

6. J. A. Coakley, F. P. Bretherton, "Cloud Cover from High Resolution Scanner Data: Detecting and Allowing for Partially Filled Fields of View", *Journal Geophysics Research*, 87, 4917-4932,

7. J. Dongarra, D. Sorenson, "SCHEDULE: Tools for Developing and Analyzing Parallel FORTRAN Programs", in The Characteristics of Parallel Algorithms, L. Jamieson, D. Gannon, R. Douglass (editors), MIT Press, Cambridge, Mass., 1987.

8. H. El-Rewini and T. Lewis ," Software Development in Parallax: The ELGDF Language," Technical Report (88-60-17), Dept. of Computer Science, Oregon State, University, July 1988.

9. EOS: Earth Observing System , Science and Mission Requirements Working Group Report, Volume I, National Aeronautics and Space Administration, Goddard Space Flight Center, Greenbelt, Maryland, 1984.

10. EOS: From Pattern to Process: The Strategy of the Earth Observing System, EOS Science Steering Committee Report, Volume II, National Aeronautics and Space Administration, Goddard Space Flight Center, Greenbelt, Maryland, 1986.

11. R. Finkel, "Large Grain Parallelism - Three Case Studies", in The Characteristics of Parallel Algorithms, L. Jamieson, D. Gannon, R. Douglass (editors), MIT Press, Cambridge, Mass.,

12. A. Henderson-Sellers, Satellite Sensing of a Cloudy Atmosphere, Taylor & Francis Ltd., Philadelphia, PA, 1984.

13. L. Jamieson, "Characterizing Parallel Algorithms", in The Characteristics of Parallel Algorithms, L. Jamieson, D. Gannon, R. Douglass (editors), MIT Press, Cambridge, Mass.,

14. A. Karp, R. Babb, " A Comparison of 12 Parallel Fortran Dialects", *IEEE Software*, September 1988.

15. B. Kruatrachue, "Static Task Scheduling and Grain Packing in Parallel Processing Systems," Ph.D. Thesis, Oregon State University, Corvallis, Oregon, 1987.

16. T. Lewis, "Parallel Programming Support Environment Research", TR-PPSE-89-1, Oregon Advanced Computing Institute, Beaverton, Oregon, 1989.

17. A. Osterhaug, Guide to Parallel Programming on Sequent Computer Systems, Sequent Computer Systems, Beaverton, Oregon, 1987.

18. C. Parkinson, W. Campbell, et al, Arctic Sea Ice, 1973-1976: Satellite Passive-Microwave Observations, National Aeronautics and Space Administration, Washington, DC, 1987.

19. W. Press, B. Flannery, S. Teukolsky, W. Vetterling, Numerical Recipes, Cambridge University Press, New York, 1986.

20. D. Siewiorek, C. G. Bell, A. Newell, Computer Structures: Principles and Examples, Mcgraw-Hill Book Co, 1982.

Additional Figures

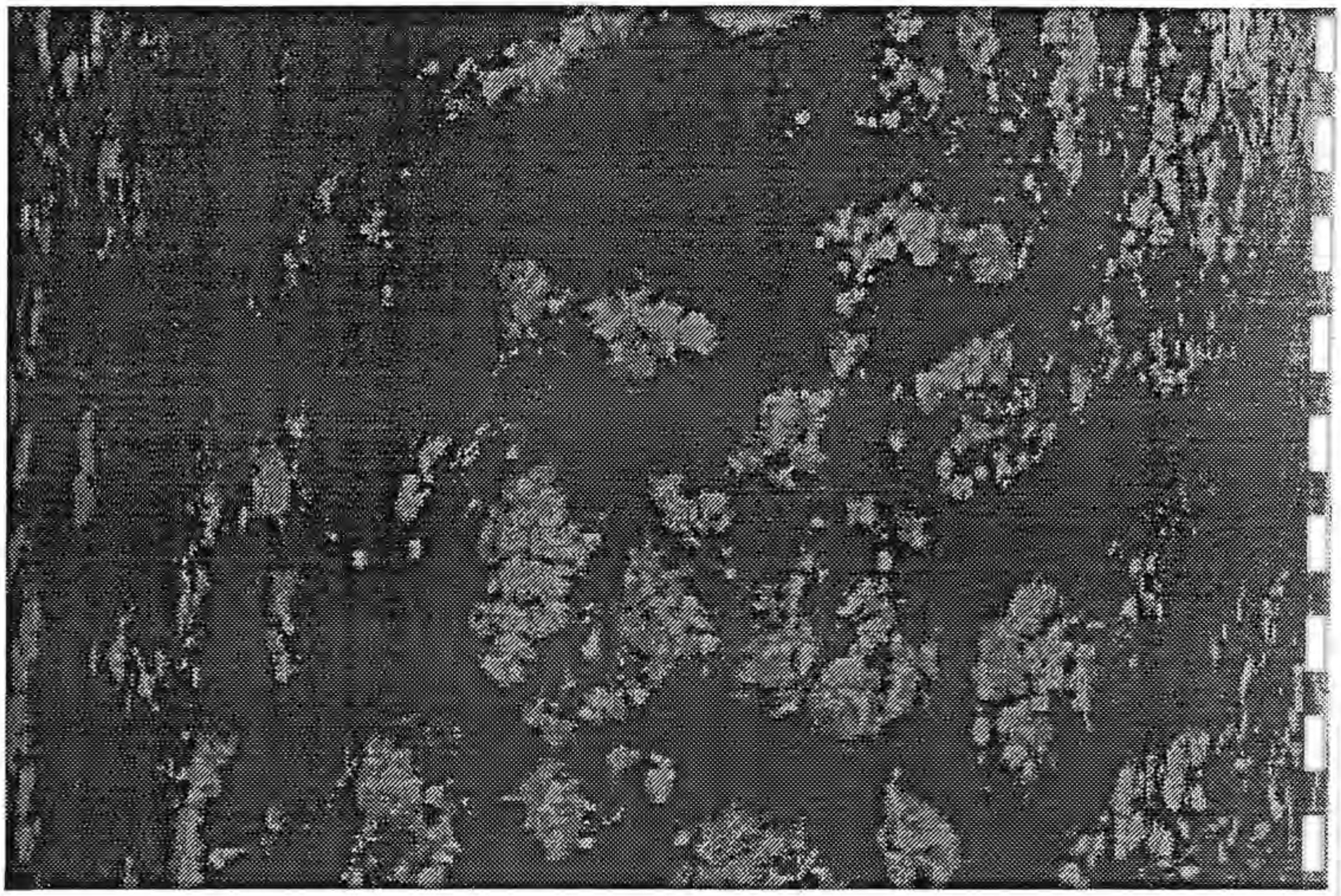


Figure 3. Infrared Image from NOAA-7 Satellite. The image is 384*256 pixels. The scene is over the Atlantic Ocean.

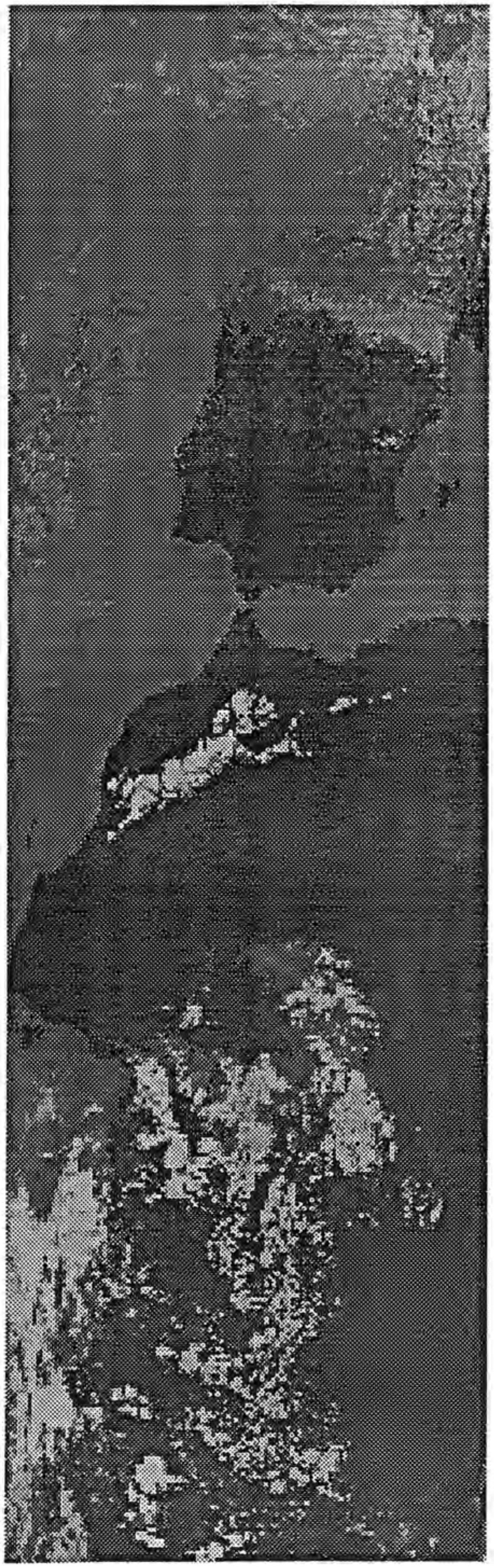


Figure 4. Infrared Image over Africa and Spain. The Image is 512*128 sampled pixels.

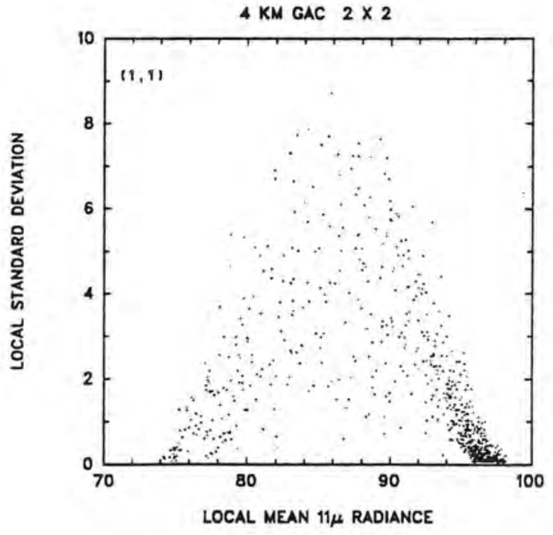
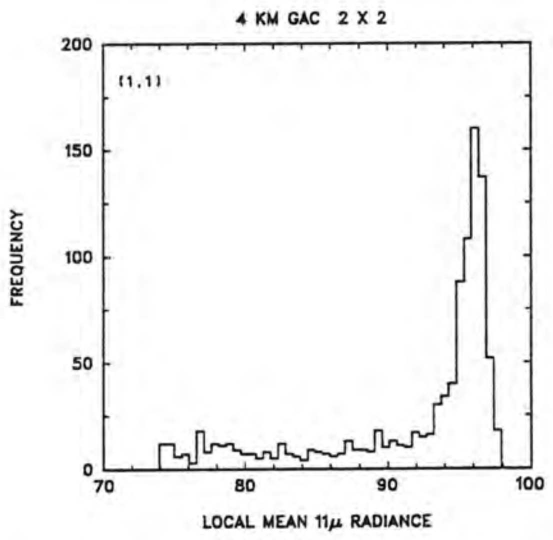


Figure 5. Local means vs standard deviations for a frame of data form an arch.



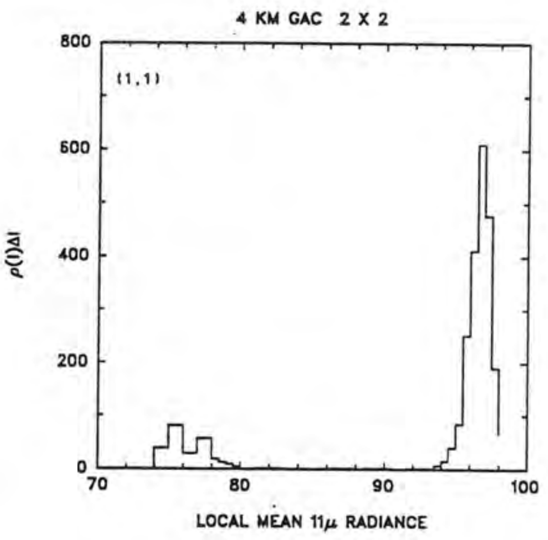


Figure 6. Frequency Distribution and Probability Distribution.

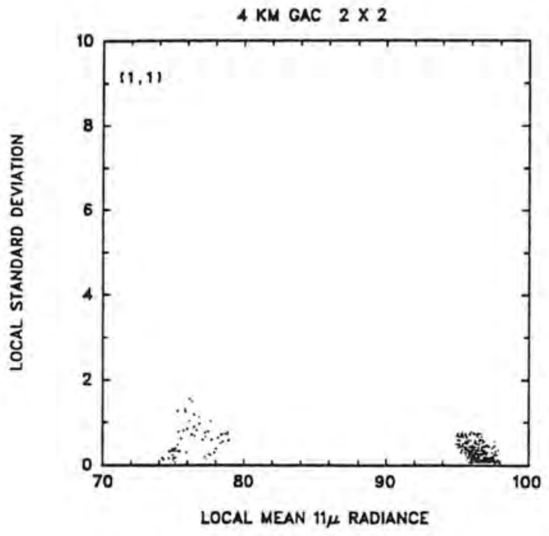


Figure 7. Feet of the arches.