AN ABSTRACT OF THE THESIS OF

BECKY JANE ROOF for the degree of Master of Science in Computer Science presented on January 13, 1988.

Title: Comparison of the VARDIG algorithm on Two Parallel Processors

Abstract approved: Redacted for privacy

Dr. Theodore G. Lewis

The specific objectives of this project are to compare the Sequent and the transputer to determine the speed of both and find out which is easiest for programming in parallel. The results are graphs showing relative and real time speedup versus number of processors and conclusions based on those graphs.

Parallel versions of the VARDIG algorithm were developed and their performance compared on two parallel processors: one for the transputer, written in occam, and one for the Sequent, written in Pascal.

Although the transputer is faster, the Sequent showed a more nearly linear speedup running this algorithm. Because the occam language was designed for parallel programming, it may lead to a better understanding of parallel algorithms.
Comparison of the VARDIG\textsuperscript{TM} Algorithm on Two Parallel Processors

by

Becky Jane Roof

A THESIS

submitted to

Oregon State University

in partial fulfillment of the requirements for the degree of

Master of Science

Completed January 13, 1988

Commencement June, 1988
APPROVED:

Redacted for privacy

Professor of Computer Science in charge of major

Redacted for privacy

Head of Department of Computer Science

Redacted for privacy

Dean of Graduate School

Date thesis is presented January 13, 1988

Typed by Becky Jane Roof for Becky Jane Roof
Acknowledgements

I would never have finished this paper without the help and encouragement of my family and friends. Every time I got down and felt like I would never finish, that I didn't have enough time, that I would never get my programs to run right, someone would make me laugh and I would realize that things weren't so bad.

I especially want to thank my CS friends Sherry Yang, Karl Schricker, Bopinderjit Singh, and Kirt Winter.

The editing job of my parents was invaluable especially on such short notice.

I really enjoyed working with Mac Cooper. He helped me immensely and always seemed glad to see me and ready to help even at 12:00 midnight or 7:00 in the morning.
TABLE OF CONTENTS

INTRODUCTION 1

IMAGE PROCESSING 2
  Statistics on Speedup 3
  The Problem 5
  What Is the VARDIG Algorithm? 7

PARALLEL PROCESSING 10

TRANSPUTER 11
  Architecture 11
  Configuration 12
  Occam 14

SEQUENT 16
  System Bus and Hardware 18

COMPARISON PROJECT 20
  Organization 20
  Specifics for the Transputer 24
  Specifics for the Sequent 26
  What Was Timed 28
  Results and Discussion 29

CONCLUSION 35

BIBLIOGRAPHY 37

APPENDICES 39
  Transputer Source Code 39
  Sequent Source Code 59
  Transputer Spreadsheet 71
  Sequent (old) Spreadsheet 72
  Sequent (new) Spreadsheet 73
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1.</td>
<td>Sample image for input</td>
</tr>
<tr>
<td>Figure 2.</td>
<td>Final output</td>
</tr>
<tr>
<td>Figure 3.</td>
<td>The VARDIG algorithm</td>
</tr>
<tr>
<td>Figure 4.</td>
<td>Configuration of the transputer</td>
</tr>
<tr>
<td>Figure 5.</td>
<td>Diagram of the Sequent</td>
</tr>
<tr>
<td>Figure 6.</td>
<td>Diagram of the VARDIG algorithm</td>
</tr>
<tr>
<td>Figure 7.</td>
<td>VARDIG illustrated for number of processors = 2 on transputer</td>
</tr>
<tr>
<td>Figure 8.</td>
<td>VARDIG illustrated for number of processors = 2 on Sequent</td>
</tr>
<tr>
<td>Figure 9.</td>
<td>Real time: Sequent versions old and new</td>
</tr>
<tr>
<td>Figure 10.</td>
<td>Relative speedup: Sequent versions old and new</td>
</tr>
<tr>
<td>Figure 11.</td>
<td>Communication and lock times</td>
</tr>
<tr>
<td>Figure 12.</td>
<td>Real time: transputer and Sequent</td>
</tr>
<tr>
<td>Figure 13.</td>
<td>Relative speedup: transputer and Sequent</td>
</tr>
</tbody>
</table>
COMPARISON OF THE VARDIG ALGORITHM ON TWO PARALLEL PROCESSORS

INTRODUCTION

There are some classes of problems that have such large data sets and require such a short processing time that they cannot be satisfactorily solved on today's conventional computers. This project is an exploratory study of this type of problem and looks at different types of hardware and software to try and present a realistic approach to solving these problems. Image processing is the specific problem chosen and parallel processing using the VARDIG algorithm is the specific approach chosen for this study. The hardware and software examined are the transputer, which is programmed using occam and the Sequent, which is programmed using Pascal.

The specific objectives of this project are to compare the Sequent and the transputer to determine the speed of both and find out which is easiest to program in parallel on. The results are graphs showing relative and real time speedup versus number of processors and conclusions based on those graphs.
IMAGE PROCESSING

Early machine vision technology in the 70's relied on algorithms developed by Stanford Research Institute (SRI) and general purpose computers. These algorithms were binary in nature and the computers ran everything sequentially. This technology was very useful for object recognition and robot guidance, but for surface defect inspection, the surface had to be smooth and the inspection speeds were too slow. A single image could take several minutes to process.

Later advances introduced "grey scale" and "neighborhood" processing. These techniques assigned a numeric value to each pixel of the image which represented its grey scale. Irrelevant information was filtered out and the image was transformed into another one which highlighted the desired information. Low pass filtering improved signal to noise ratio and high pass filtering enhanced the edges of an image.

Surface defect inspection was still difficult and slow. Inspection of complex shaped parts was now possible but custom programming to ignore normal shapes such as edges, corners and holes was required. The speed was actually decreased since grey scale and neighborhood processing required much more CPU processing. An image which could
be "inspected within a minute using SRI algorithms, increased to over fifteen minutes using grey scale/neighborhood processing" (Thomason, 1986).

To solve this problem, Advanced Computer Concepts (ACC) turned to Dr. Jack Sklansky, Professor of Electrical Engineering and Director of Image Engineering Research Programs at University of California/Irvine, for the development of some image processing algorithms. They wanted algorithms devoted solely to surface defect detection rather than ones aimed at object recognition or robot guidance. They planned to dedicate these algorithms to circuit hardware to attain the desired and needed speed (Thomason, 1986).

ACC was rewarded with the development of the VARDIG algorithm. The algorithm was dedicated to circuit hardware in order to achieve the required automation speeds and the fifteen minutes needed by the general purpose computers to calculate the result was reduced to "within one tenth of a second for a 9000 times speed increase" (Thomason, 1986).

Statistics on Speedup

GeoSpectra Corp. is working with a scientific software package called Automatic Topographic Mapper (ATOM) to automatically extract elevation data on a pixel-by-pixel basis from digitized stereo images. ATOM ran on a DEC VAX
computer which is faster than any existing special purpose electro-optical equipment, "yet it still took 24 hours to complete the extraction of 50,000,000 elevations from one stereo pair of 7000x7000 images" (Heywood, 1987). Utilizing parallel processing on the most time consuming and critical portions of ATOM, the time has been reduced "by factors of 12 to 18" (Heywood, 1987).

The Industrial Technology Institute (ITI) in Ann Arbor, MI has developed a prototype 3D machine vision inspection system that couples a full field, structured lighting technique called Moire interferometry and the processing capability of a parallel processor. This sensing system outputs Z-axis height information for every pixel in the image.

"Conventional image processing systems can be used to process the moire fringe data but resulting cycle times are in the tens of seconds: far too long for most industrial manufacturing applications" (Heywood, 1987).

ITI has introduced parallel processing to this application and created, in Moire interferometry, a practical solution for 3D inspection applications. "Current cycle time is three seconds, an optimized version could be accomplished in less than one second" (Heywood, 1987).
The Problem

When processing an image of a seed, each specific point on a seed surface must be identified in each of two distinct images taken from different positions so that the effect known as parallax can be used to locate the point in three dimensional space.

The test input in this particular case is an image such as Figure 1. It is represented internally as a disk file with a 128 byte header containing the position in pixel coordinates of the upper left and lower right corners. This is followed by \((\text{bottom row} - \text{top row} + 1) \times (\text{right column} - \text{left column} + 1)\) bytes of image data where each value represents the grey scale at that particular pixel.

The final output is a three dimensional histogram, see Figure 2. Vartest6 Image Histogram is the histogram produced from the image in Figure 1. The high peaks along the 0 line are differences of 0 and represent pixels with the same magnitude as their neighbors in the given direction. This occurs along the lines or in the background areas. All eight directions have high peaks at 0 since in general a pixel has the same value as its eight neighbors. The tails of the histogram represent large differences between pixels. Notice the long tails in the northwest, northeast, southwest and southeast directions. In these directions there is either a solid line i.e. high 0 peaks or alternating lines and background.
Figure 1. Sample image for input

VARTEST6 IMAGE HISTOGRAM

Figure 2. Final output
Going from background to line is a large difference which produces the long tails.

What Is the VARDIG Algorithm?

The VARDIG (variability of directions gradient) algorithm is a processing method for detecting differences between irregular shapes and predictable normal shapes, regardless of orientation, while exhibiting immunity to specular reflections and other changes in light intensity. It is based on grey scale and vector processing. This method of image processing can distinguish between flaws and normal shapes by comparing the grey scales or directions to internal look-up-tables. For example, straight edges are characterized by vectors in one direction and flaws are characterized by random changes in direction.

Eight differences are calculated for each pixel of the image and put into a difference array. These differences correspond to eight directions i.e. northwest, north, northeast, west, east, southwest, south and southeast in row major order,
Figure 3. The VARDIG algorithm.

see Figures 3a and 3b. The mathematical formulation for calculating the difference array for the northwest direction is given by the following formula.
difference northwest of pixel(i,j) = pixel(i-1,j-1) - pixel(i,j)

Theoretically, four of the differences may be reused by neighbors by changing the sign of the original difference but it is not until a complete row and column have been calculated that only four new calculations are needed. This means that the entire first and last rows and the entire first and last columns must be treated as special cases. (When the image is broken down into many small images i.e. large number of processors, the method would get more and more inefficient.)

The difference array contains the indices of the histogram array which need to be updated. The contents of the difference array correspond to the rows of the histogram array and the indices of the difference array correspond to the columns of the histogram array, see Figures 3b and 3c. To calculate the histogram, the following formula is used.

$$\text{histogram(diff,direction)} = \text{histogram(diff,direction)} + 1$$

The final histogram array contains a tally of all the pixels with the same magnitude in the same direction. It is from this histogram array that the actual histogram is drawn.

The differences of the pixels range from -63 through 63. An array index must be positive so after the differences are calculated 63 is added to each one to translate the range from -63 through 63 to 0 through 126 with 63 now representing 0.
PARALLEL PROCESSING

Parallel processing takes many approaches. There are shared memory systems, distributed memory systems, connectionist systems and systolic array systems. In order to decide which system is best for an application, reliable measurements must be taken.

For this application two systems were compared: a distributed memory system as illustrated by the transputer and a shared memory system as illustrated by the Sequent.
TRANSPUTER

The transputer is a high-performance, single-chip computer which is made up of a processor, memory and hardware communication links. The host CPU is connected to the PC and the rest of the CPUs through the communication links. The host distributes information to and collects information from the other CPUs. Each CPU computes the needed information and sends the information back to the host. All communication between CPUs and the PC is accomplished via hardware links and all communication between processes in a single CPU is accomplished via software channels.

The transputer is unique in that it is programmed using occam, which is a high level programming language, but at the same time it is very close to the assembly language of the transputer. Transputers act as "hardware occam processes" and execute programs more or less directly.

Architecture

The CPUs are connected through bidirectional hardware links. Occam channels are uni-directional so separate channels are needed for input and output. A link is capable of supporting two "hard" (placed) occam channels. Thus, each
link can implement both input and output. Once a link communication has been initiated by the central processor it proceeds autonomously, managed by a link processor, so the central processor can execute another process.

Transputers are very fast. "They can process instructions at a rate up to 10 million instructions per second. A link can operate at speeds of 5, 10, or 20 million bits per second in both directions at the same time" (Kerridge, 1987).

Solutions may be developed independently of the actual transputer network. No I/O to the screen may be done from within the CPUs since they are only linked to each other and the host. Debugging is difficult but since the hard channels may be imitated using soft channels, any application may be developed on the host alone and once the bugs are out, transferred to the transputer network for execution.

Configuration

Each CPU has four links. These links may be connected in any configuration as long as the following basic rules are observed: 0 links are connected to 1 links and 2 links are connected to 3 links, all the CPUs are connected in such a way that a continuous path may be followed from the host to the last CPU and the software channels match the links, see Figure 4.
In order to match the software configuration to the hardware configuration, three new statements must be used. These are: PLACED PAR, PROCESSOR number transputer.type, and PLACE channel.name AT link.address.

The PLACED PAR statement places the processes on separate CPUs. This statement may be replicated just as any PAR statement may be.

The PROCESSOR statement matches a specific processor with a number. The number may be generated within the program or it may be a constant, but it is assigned by the user to the processor. The transputer type indicates which specific transputer is being used so code can be generated by the...
compiler. T2 is used for a 16-bit transputer, T4 for a 32-bit transputer and T8 for a 32-bit transputer with an on-chip floating point processor.

The PLACE statement declares which channel name is allocated to a particular link address. This is similar to a procedure call in Pascal but instead of passing variables and executing a procedure, channel names are allocated to link addresses and the CPU specified through the PROCESSOR statement begins execution autonomously, synchronized only by the link communications.

Occam

Occam is the programming language of the transputer. It is the first language to be based upon the concept of parallel and sequential execution, and provides automatic communication and synchronization between concurrent processes.

The fundamental concept of occam is the process. An occam program consists of processes executed in parallel with all communication accomplished via channels. These channels "provide a zero-buffered, unidirectional, data path between just two processes running in parallel" (Wilson, P.).

Synchronization is managed by occam but deadlock is still possible. In order to avoid deadlock, the first process to
undertake an operation upon a channel, either input or output, must wait until the second process is ready to undertake the corresponding operation upon the same channel. While a process is waiting for communication it is suspended but as soon as communication is complete, both processes may continue.
SEQUENT

The Sequent is a true, tightly-coupled multiprocessor, not an array processor and it is designed to allow both parallel and sequential programs to run at the same time. It incorporates multiple identical processors and a single shared memory. This enhances resource sharing and communication among different processors.

The Sequent contains three elements that distinguish it from other systems. First, it incorporates a parallel architecture. Second, it uses the UNIX operating system. This provides power, flexibility, easy expandability, and is very popular. And last, it uses standard interfaces such as MULTIBUS and ethernet so it can be used without learning a completely new system.

Communication and I/O is accomplished via a single high-speed bus. Each CPU is identical and can execute both user and kernel code, see Figure 5.
Programs can be run on the Sequent whether they were written in parallel or sequentially with no modifications necessary. Load balancing is dynamic with all CPUs automatically keeping themselves busy as long as there is work to do. When a CPU is finished or just waiting for I/O, it begins executing the next available process in the system-wide run queue.

Synchronization is not done by the Sequent but user-accessible hardware locks are provided. These consist of a bit which may be set in a single atomic operation, providing a fast, hardware-based mechanism for mutual exclusion. These lock the variables in shared memory until writing or
calculations are finished and must be unlocked so the next CPU may have access. The system provides the locks but their correct use is up to the user. The user may write locks if the ones provided are not adequate.

The Sequent provides special support for parallel programming. This support includes: shared memory, system-provided hardware locks, a parallel programming library, including Pascal, C and FORTRAN, explicit control over system resources and a parallel debugger.

System Bus and Hardware

The system bus has a 64-bit buswidth and carries data among the system's CPUs, memory modules and peripheral subsystems. The system bus supports pipelined I/O, memory operations and variable sized data packets.

"In current Sequent systems, the CPUs, memory modules, and peripheral controllers use only 32 bits of the 64-bit bus and can achieve a sustained data transfer rate of 26.7 Mbytes per second. Future devices that use the full 64-bit bus will be capable of 53.3 or 64 Mbytes per second" (Sequent Computer Systems Inc., 1986).

Locks are used to ensure mutual exclusion. All shared variables are stored in global memory so each processor has access to them. Reading of shared variables is fine since
nothing is modified but writing is permitted by only one processor at a time.

There is a bit in global memory which is either set (locked) or not set (unlocked). Each processor has a corresponding bit in local memory. When access to a shared variable for modification is desired the local bit is sampled. If the local bit is not set the global bit is locked and access is granted. If the local bit is set the processor must wait until the lock is removed. Each processor samples its own local bit rather than tying up the system bus trying to access the global lock. As soon as the local bit indicates memory is no longer locked, the processor may send a request across the system bus to lock the global bit, thereby gaining access to global memory.
Organization

At first the VARDIG algorithm seems to be sequential in nature because it systematically subtracts each pixel in the image from its eight neighbors to update the histogram. How can this algorithm be computed in parallel?

It makes no difference whether pixel 1, 16, 83, or 10,000 is calculated first. This allows the image to be broken up into various chunks which may then be processed by different processors.

A variety of partitioning strategies could be used, but limiting the number of processors to a power of two means the chunks can be divided as symmetrically as possible, thus giving each processor an equal share of work and allowing for maximum speedup. The location of each chunk is given by its upper left and lower right corners. The endpoints of the chunks are sent to the appropriate processor.

The VARDIG operator requires eight nearest neighbor elements. The edge elements are ignored in calculation and only used to calculate those elements bordering them.

This raises the problem of what to do with the edges of the chunks which aren't actual edges of the original image but merely the borders between two chunks. This was solved by including extra rows or columns in the endpoints of each
chunk to act as the outer edge. The extra rows or columns are not calculated themselves but their values are used in the calculation of the original elements of the chunk.

As the problems were solved, a basic outline of how to write the VARDIG algorithm in a parallel manner was developed as follows:

1. Read in the image.

2. Based on the number of processors, create an endpoint array with the upper left and lower right corners of each nearly symmetric chunk.

   \[
   \begin{align*}
   \text{endpoint array}(\text{processor}, \text{upperleft row}) &= \text{current row} \\
   \text{endpoint array}(\text{processor}, \text{upperleft col}) &= \text{current col} \\
   \text{endpoint array}(\text{processor}, \text{lowerright row}) &= \text{current row} + \text{row element} \\
   \text{endpoint array}(\text{processor}, \text{lowerright col}) &= \text{current col} + \text{col element}
   \end{align*}
   \]

   The last chunks add last row and last col.

3. Add extra rows or columns, as necessary, to act as the outer edge for each chunk.

   If chunk = upper left corner with \(n\) processors

   \[
   \begin{align*}
   \text{endpoint array}(\text{processor}, \text{lowerrow}) &= \text{endpoint array}(\text{processor}, \text{lowerrow}) + 1 \\
   \text{endpoint array}(\text{processor}, \text{lowercol}) &= \text{endpoint array}(\text{processor}, \text{lowercol}) + 1
   \end{align*}
   \]

4. Send the image array, total row and col sizes and upper left corner of the image and the endpoint array to each processor.

5. Each processor performs the VARDIG operator over its entire chunk. The edges of the chunks are ignored by starting at (upperrow +1, uppercol+1) and going to (lowerrow -1, lowercol -1).
row1 = (upperrow of chunk -
(upperrow of image -2))
row2 = lowerrow of chunk - upperrow of
image
col1 = (uppercol of chunk -
(uppercol of image -2))
col2 = lowercol of chunk - uppercol of image

6. Each processor computes a histogram for its
chunk.
   for all pixels from row1 to row2
      for all pixels from col1 to col2
         for all directions from NW to SW
            diff(direction) = pixel(direction) -
                             current pixel
         for all directions from NW to SW
            hist(diff,direction) =
                                hist(diff,direction) + 1

7. A histogram is received from the
   previous CPU and the local copy of the histogram
   is added in before the histogram is sent on in the
   direction of the host.

8. The histogram is output. See Figure 6 for a
   data flow diagram of the algorithm.
Figure 6. Diagram of the VARDIG algorithm.
A few modifications and details were added when implementing the algorithm on the transputer and the Sequent. Since the Sequent uses a shared memory system, all the data and variables which need to be passed to the processors are simply made global so each processor may access the shared data. For the transputer, this information is passed via channels to each processor.

Specifics for the Transputer

All the partitioning is done by the host. The whole image and an array of all the endpoints for each chunk is sent through channels to the CPUs. The total row and column sizes and the upper left corner of the original image are also sent. The information is passed to the CPUs sequentially, starting at processor 1 and continuing through processor N, see Figure 4. For the specific case where processors equal 8, the information could be distributed in parallel by utilizing link 2 of the host. The information could be divided in half with one half being sent through link 1 and the other half through link 2. Processor 1 and 8 could then divide the information into 3 parts and send it out to the three processors directly linked to them. This algorithm is not directly scalable and poses a complicated distribution problem.
Each CPU reads in the image and then passes the whole image on to the next CPU. The array of endpoints is read and the particular endpoints for each CPU are taken out before the rest are sent on. The row and column sizes and the original upper left corner are read, saved, and then passed on. The last CPU retains the information since it has no where to send it, see Figure 7.

Once all the needed information is received, each CPU calculates a difference array by applying the VARDIG operator to its chunk of the image. Each CPU creates a partial histogram of the image containing only the information for its chunk. The last CPU sends its histogram back through the channel as soon as it finishes. The CPUs in the middle read in the histogram from the end, add their histogram to it, and then send it on towards the host. The host receives a complete histogram ready for output.

If the histograms are passed in a different sequence it might be possible to speed up the communications. Instead of processor N, which is the last processor to receive the information, being the first processor to send the histogram back, processor 1 could send its histogram to processor 2 and finally processor N would send the complete histogram to the host.

Since occam automatically synchronizes itself, deadlock is the only race condition to consider. This is easily avoided as long as each process alternates between sends and receives. Never send and send then receive and receive.
Figure 7 illustrates how to program the transputer using occam.

Specifics for the Sequent

First, the image is read in and the main histogram is initialized to zero. The partitioning is completed and all the endpoints for each processor are calculated and put into the endpoint array.

Since the Sequent utilizes a shared memory, nothing is passed to the processors. The needed variables are defined globally and are accessible to each processor.

Each processor applies the VARDIG operator to its chunk of the image and computes a difference array from which the
histogram array is created. Before the processor may access the main histogram array to add its information, the main histogram array must be locked to ensure single access. As soon as the information is added in, the histogram array is unlocked so it may be accessed by the rest of the processors.

Complete timing on the Sequent means that everything except I/O and initializations are timed. The Sequent is a shared memory system which has overhead associated with setting up a page table and forking the required processors which the transputer doesn't have. When timing everything this is not taken into consideration and is included in the timing results. When only the parallel parts are timed the forking is still timed but the paging is not. The parallel times and the complete times are very similar.

The Sequent provides no synchronization. That job is left up to the programmer. The Sequent does provide hardware locks. Any time mutual exclusion is not assured the user must initiate a hardware lock until the access is complete. The lock is then removed so the variable is once more accessible to all processors. Deadlock will result if the lock is left on. Figure 8 illustrates how to program the Sequent using Pascal.
What Was Timed

First, each computer was timed as it ran the entire program leaving out I/O and initializations for both machines. Second, just the parallel parts of each program were timed including distribution, computation, and collection. Third, the calculation portions of each program were timed. This last part had little affect on the project since it didn't really time the parallel capabilities of either machine but just the calculation capabilities. It was useful to ensure that the processes were behaving as expected and were consistent in their execution speeds. Finally, the channel communications of the transputer and the locks of the Sequent were timed.
Results and Discussion

After comparing the first set of times for the transputer and the Sequent it could be seen that the Sequent was not very highly optimized. The Sequent was between 8 and 14 times slower than the transputer depending on the size of the image used. A new version of the algorithm was developed for the Sequent after it was noticed that there was no distributing and collecting being done. The role of the host on the transputer is paralleled on the Sequent by the Sequent's shared memory. Collection must be done in the shared memory rather than individually by each processor. The new version increased the speed of the algorithm by a factor of almost 4 for the largest image and processors equal to 8, see Figure 9.
Relative speedup times were also vastly improved using the new algorithm. Where the old algorithm was increasing speedup by a factor of 1.5, the new algorithm reached 79% of its theoretical maximum, see Figure 10.
The transputer is a message passing system which uses communication along channels to pass information between processors. The Sequent, being a shared memory system, must lock access to its shared memory whenever a write must be done. Both of these operations were timed to see if either one had a major affect on the processing speed. Figure 11 shows the transputer is communication bound with communication time taking a little more than 1 second when processing the largest image.
The locking of the Sequent takes almost no time compared to the processing time.

The transputer code is not fully optimized but the Sequent code is close. The transputer compared to the Sequent was able to execute the VARDIG algorithm much faster, see Figure 12.
The Sequent outperforms the transputer when it comes to relative speedup. The Sequent, as mentioned above, achieves 79% of its maximum possible speedup while the transputer achieves 55% for large images, see Figure 13.
After completing the programs, it was seen that the transputer needed 737 total lines to program the VARDIG algorithm but took up only 4-11 k of memory for code depending on the number of processors used for processing. The Sequent needed 472 lines to program the VARDIG algorithm but used a little over 43 k of memory for code.
Parallel processing is a brand new field with very few answers and many questions. Each new conclusion seems to raise more questions.

This research shows the transputer is faster but the Sequent has a better relative speedup. However, the transputer was not optimized. A better configuration could have been utilized and the message passing sequence speed could have been increased. These changes may or may not affect the final speed depending on the amount of overhead needed but the transputer's computation capabilities are fast. The Sequent also may not have been treated entirely optimally. The goal was to time just the parts of the algorithm for each machine that were essentially the same. The Sequent, in order to compute in parallel must, the first time through, set up a page table and fork the correct amount of processors. The overhead times were included in the final times for the Sequent and could have affected its speedup and relative times. Finally, the algorithm itself may not have been fully optimized. All eight differences were always calculated for each pixel. It is possible to reuse four of the differences each time so that after the first row and column of pixels have been calculated, only four differences are left to calculate for the remaining pixels. This poses an indexing problem and may create more overhead time than it saves processing time.
Both algorithms were tested on a variety of image sizes so it could be seen when the number of processors became too large for the size of the image and actually made the processing slow down. The 6x6 image is the minimum size image which allows all eight processors to process at least one pixel. This image was small and mostly shows edge effects. The 50x50 image is large enough to show the full speedup curve. Four processors is the optimum and speedup decreases when more are added. The 237x255 image is the maximum size image for the current vision system and best illustrates the relative speedup values and the communication or lock times for each computer.


APPENDICES
Transputer Source Code

PROC host (CHAN OF ANY keyboard, screen, [4]CHAN OF ANY from.user.filer, to.user.filer)

-- Vars
-- Constants
VAL max.cpu IS 8: -- number of parallel processors
  -- to be used
VAL upperrow IS 0: -- upper left..
VAL uppercol IS 1: -- corner of the image
VAL lowerrrow IS 2: -- lower right..
VAL lowercol IS 3: -- corner of the image
VAL max.row IS 238: -- maximum size..
VAL max.col IS 256: -- of the image array
VAL max.histogram IS 129: -- size of the histogram
VAL number.endpoint IS 4: -- four endpoints of the image
VAL number.histogram IS 8: -- eight directions of the
  -- histogram
VAL ticks.per.second IS 15625: -- number of ticks in one
  -- second

-- Arrays
[number.endpoint]INT endpoints: -- holds original four
  -- endpoints of image
[max.cpu][number.endpoint]INT endpoint.array: -- holds
  -- endpoints for each CPU
[max.histogram][number.histogram]INT main.histogram:
  -- holds final histogram
[max.row][max.col]INT image.array: -- holds the image data

-- Integers
INT row.max, col.max: -- size of current image being
  -- processed
INT index, index2: -- loop variables
INT rowpart, colpart: -- used to calculate number of chunks
INT col.element, row.element: -- row and col size for chunks
INT last.row, last.col: -- row and col size for last chunk
INT total.row, total.col: -- row and col size of the image
INT row1, row2, col1, col2: -- endpoints of the image
INT number.part: -- number of partitions
INT current.row, current.col: -- starting row and col
-- endpoints
INT temp.processor: -- number of processors
INT time.start, time.end, runtime: -- timing variables
INT quotient, remainder: -- holds time in seconds

-- link vars
VAL link0out IS 0: -- the four channels for each cpu
VAL link1out IS 1: -- which send information
VAL link2out IS 2: -- to other cpus and the host
VAL link3out IS 3:
VAL link0in IS 4: -- the four channels for each cpu
VAL link1in IS 5: -- which receive information
VAL link2in IS 6: -- from other cpus and the host
VAL link3in IS 7:
CHAN OF ANY chan.out, chan.in: -- soft channels for
-- receiving or sending info
PLACE chan.out AT link1out: -- link #1 contains two hard
PLACE chan.in AT link1in: -- channels, out and in

TIMER clock: -- timing channel
BYTE ch: -- used to stop output display on screen

-- PROC input.data
PROC input.data (CHAN OF ANY from.ws, to.ws, []INT array,

-- PROC read.file
PROC read.file(CHAN OF ANY from.stream, to.stream,
	VAL INT fold.no, max.col.array, []INT in.array, [][]INT in.matrix, INT result)

#USE "\tdsiolib\userio.tsr"
#USE "\tdsiolib\interf.tsr"

-- Constants
VAL ft.number.error IS -11: -- error checking vars for
VAL ft.terminated IS -8: -- reading from a file
VAL tt.beep IS 13(BYTE):
VAL upperrow IS 0:
VAL uppercol IS 1: -- upper right, lower left corner of
-- image
VAL lowerrow IS 2:
VAL lowercol IS 3:

INT index1, index2:
INT max.row.matrix, max.col.matrix: -- maximum row
INT kchar, number, row, col:

SEQ

SEQ

-- channel declarations
CHAN OF INT filekeys:
CHAN OF INT keyboard IS filekeys: -- channel from
-- simulated keyboard

CHAN OF ANY echo:
CHAN OF ANY screen IS echo: -- echo channel with
-- scope local to this PAR only

PAR
-----------------------------------------------------------------------------------------
SEQ
keystream.from.file (from.stream, to.stream, keyboard, fold.no, result)
-- check input.error when real screen accessible again
-----------------------------------------------------------------------------------------
scrstream.sink (screen) -- consume everything echoed
-----------------------------------------------------------------------------------------
SEQ
-- read data to matrix
kchar:= 0
col:= 0

-- read in endpoints first
WHILE (kchar <> ft.terminated) AND (col >= 0)
SEQ
WHILE (col < max.col.array)
SEQ
read.echo.int (keyboard, screen, number, kchar)
in.array[col]:= number
col := col + 1

-- then read in the image
max.row.matrix:=(in.array[lowerrow] - in.array[upperrow]) + 1
max.col.matrix:=(in.array[lowercol] - in.array[uppercol]) + 1
SEQ row = 0 FOR max.row.matrix
SEQ col = 0 FOR max.col.matrix
SEQ
read.echo.int (keyboard, screen, number, kchar)
in.matrix[row][col]:= number
col := -1
IF
  (kchar >= 0) OR (kchar = ft.number.error)
  keystream.sink (keyboard)
  -- consume rest of the keyboard file
TRUE
SKIP
write.endstream (screen) -- terminate scrstream.sink

-- test input.error, if OK tabulate
IF
  result <> 0
  SEQ
    write.full.string (screen, "%% File reading error : ")
    write.int (screen, result, 0)
    newline (screen)
  TRUE
  SEQ
    write.full.string (screen, "%% File reading OK : ")
    newline (screen)
:

INT result1, result2:
SEQ
  read.file(from.ws, to.ws, 1, number.endpoint, array,
    matrix, result1)
:

-- PROC output.data
PROC output.data (CHAN OF ANY from.stream, to.stream,
  [[[]]INT matrix, INT quotient, remainder)
  #USE "\dsiolib\userio.tsr"
  #USE "\dsiolib\interf.tsr"
SEQ
  -- PROC writings
  PROC writings (CHAN OF ANY screen, [[[]]INT matrix,
    INT quotient, remainder)
SEQ
-- header and timing info
newline(screen)
write.full.string(screen,"Processing a ")
write.int(screen,(row2-row1),3)
write.full.string(screen," X ")
write.int(screen,(col2-col1),3)
write.full.string(screen," image.")
newline(screen)
newline(screen)
write.full.string(screen,"Using ")
write.int(screen, max.cpu, 3)
write.full.string(screen," processors and timing 
   everything")
newline(screen)
write.full.string(screen,"except the initializations and 
   I/O")
newline(screen)
write.full.string(screen,"the total number of seconds is: 
   ")
write.int(screen,quotient,4)
write.full.string(screen,".")
write.int(screen,remainder,4)
newline(screen)
newline(screen)
write.full.string(screen,"NW N NE 
   W")
write.full.string(screen,"E SW S SE")
newline(screen)

-- write result
newline(screen)
SEQ row = 0 FOR max.histogram
SEQ
   write.full.string(screen,"mag= ")
   write.int(screen,row,3)
SEQ col = 0 FOR number.histogram
   write.int(screen,matrix[row][col], 8)
newline(screen)

CHAN OF ANY fromprog, tofile:
INT fold.no, result:
PAR
SEQ  writings(fromprog, matrix, quotient, remainder)
write.endstream(fromprog)
SEQ  scrstream.fan.out (fromprog, tofile, screen)
write.endstream(tofile)
SEQ  scrstream.to.file (tofile, from.stream, to.stream,
  "output", fold.no, result)
  -- special action if result <> 0
IF  
  result = 0
  SKIP
  TRUE
  STOP -- only alternative is to call
  -- scrstream.sink(tofile)

  write.full.string(screen, "Press [ANY] key to continue")
INT any:
  read.char(keyboard , any)
:
SEQ  -- initializations
SEQ  -- initialize main histogram array to 0
SEQ index = 0 FOR max.histogram
  SEQ index2 = 0 FOR number.histogram
  SEQ
    main.histogram[index][index2]:= 0

input.data (from.user.filer[2], to.user.filer[2], endpoints,
  image.array)
clock ? time.start -- start timing
-- process partition
SEQ  -- initialize endpoints and figure out total row and col size
  row1:= endpoints[upperrow]
  col1:= endpoints[uppercol]
  row2:= endpoints[lowerrow]
  col2:= endpoints[lowercol]
  total.row:= row2-row1
  total.col:= col2-col1
-- decide how many rows and columns to partition
-- depending on the number of processor, compute the
-- number of rows and columns to divide the image array
-- into to be processed on n processors

rowpart := 1
colpart := 1
temp.processor := max.cpu
WHILE (temp.processor >= 2)
SEQ
IF
  temp.processor >= 2
  rowpart := rowpart * 2
TRUE
  SKIP
  temp.processor := temp.processor / 2
IF
  temp.processor >= 2
  colpart := colpart * 2
TRUE
  SKIP
  temp.processor := temp.processor / 2
IF
  rowpart > 1  -- if there is more than just 1 processor
SEQ
  -- divide the array up into partitions

  -- Compute the partition size for each partition
  -- Compute number of rows, columns and also last
  -- row and column size in case they don't divide
  -- evenly
  row.element := total.row / rowpart
  col.element := total.col / colpart
  last.row := (total.row - ((rowpart -1) * row.element))
  last.col := (total.col - ((colpart -1) * col.element))

  -- Find the endpoints of partitions. Having computed
  -- the size of each partition, find the endpoints of
  -- each partition with respect to the image endpoints.
  -- Store the endpoints of each partition.

current.row := row1
current.col := col1
number.part := -1
SEQ i = 0 FOR rowpart -1
SEQ
  SEQ j = 0 FOR colpart -1
  SEQ
    number.part := number.part + 1
    endpoint.array[number.part][upperrow] :=
      current.row
    endpoint.array[number.part][lowerrow] :=
      current.row + row.element
    endpoint.array[number.part][uppercol] :=
      current.col
    endpoint.array[number.part][lowercol] :=
      current.col + col.element
    current.col := current.col + col.element
    number.part := number.part + 1
    endpoint.array[number.part][upperrow] :=
      current.row
    endpoint.array[number.part][lowerrow] :=
      current.row + row.element
    endpoint.array[number.part][uppercol] :=
      current.col
    endpoint.array[number.part][lowercol] :=
      current.col + col.element
  SEQ
    current.col := col1
    current.row := current.row + row.element
    SEQ i = 0 FOR colpart -1
    SEQ
      number.part := number.part + 1
      endpoint.array[number.part][upperrow] :=
        current.row
      endpoint.array[number.part][lowerrow] :=
        current.row + last.row
      endpoint.array[number.part][uppercol] :=
        current.col
      endpoint.array[number.part][lowercol] :=
        current.col + col.element
      current.col := current.col + col.element
      number.part := number.part + 1
      endpoint.array[number.part][upperrow] :=
        current.row
      endpoint.array[number.part][lowerrow] :=
        current.row + last.row
      endpoint.array[number.part][uppercol] :=
        current.col
      endpoint.array[number.part][lowercol] :=
        current.row + last.row
endpoint.array[number.part][uppercol] := current.col
endpoint.array[number.part][lowercol] := current.col + last.col

current.col := col1
current.row := current.row + row.element

-- check for edge and corners and decide how many
-- elements to send
-- This is the main routine that checks if each
-- partition is one of the 11 cases, and decides how
-- many extra rows and columns to send to each
-- processor

SEQ i = 0 FOR max.cpu
SEQ
IF

-- left side with 2 processors

(endpoint.array[i][upperrow] = row1) AND
(endpoint.array[i][uppercol] = col1) AND
(endpoint.array[i][lowercol] = col2)
endpoint.array[i][lowerrow] :=
endpoint.array[i][lowerrow] + 1

-- upper left corner with n processors

(endpoint.array[i][upperrow] = row1) AND
(endpoint.array[i][uppercol] = col1)
SEQ
endpoint.array[i][lowerrow] :=
endpoint.array[i][lowerrow] + 1
endpoint.array[i][lowercol] :=
endpoint.array[i][lowercol] + 1

-- right side with 2 processors

(endpoint.array[i][lowerrow] = row2) AND
(endpoint.array[i][uppercol] = col1) AND
(endpoint.array[i][lowercol] = col2)
endpoint.array[i][upperrow] :=
endpoint.array[i][upperrow]
-- upper right corner with n processors

(\text{endpoint.array}[i][\text{lowerrow}] = \text{row2}) \text{ AND}  
(\text{endpoint.array}[i][\text{uppercol}] = \text{col1})  
SEQ  
\text{endpoint.array}[i][\text{upperrow}] :=  
\text{endpoint.array}[i][\text{upperrow}]  
\text{endpoint.array}[i][\text{lowercol}] :=  
\text{endpoint.array}[i][\text{lowercol}] + 1  

-- lower left corner with n processors

(\text{endpoint.array}[i][\text{upperrow}] = \text{row1}) \text{ AND}  
(\text{endpoint.array}[i][\text{lowercol}] = \text{col2})  
SEQ  
\text{endpoint.array}[i][\text{uppercol}] :=  
\text{endpoint.array}[i][\text{uppercol}]  
\text{endpoint.array}[i][\text{lowerrow}] :=  
\text{endpoint.array}[i][\text{lowerrow}] + 1  

-- lower right corner with n processors

(\text{endpoint.array}[i][\text{lowerrow}] = \text{row2}) \text{ AND}  
(\text{endpoint.array}[i][\text{lowercol}] = \text{col2})  
SEQ  
\text{endpoint.array}[i][\text{upperrow}] :=  
\text{endpoint.array}[i][\text{upperrow}]  
\text{endpoint.array}[i][\text{uppercol}] :=  
\text{endpoint.array}[i][\text{uppercol}]  

-- left edge with n processors

(\text{endpoint.array}[i][\text{upperrow}] = \text{row1})  
SEQ  
\text{endpoint.array}[i][\text{lowerrow}] :=  
\text{endpoint.array}[i][\text{lowerrow}] + 1  
\text{endpoint.array}[i][\text{lowercol}] :=  
\text{endpoint.array}[i][\text{lowercol}] + 1  
\text{endpoint.array}[i][\text{uppercol}] :=  
\text{endpoint.array}[i][\text{uppercol}]  

-- right edge with n processors
(endpoint.array[i][lowerrow] = row2)
SEQ
    endpoint.array[i][upperrow] :=
        endpoint.array[i][upperrow]
    endpoint.array[i][lowercol] :=
        endpoint.array[i][lowercol] + 1
    endpoint.array[i][uppercol] :=
        endpoint.array[i][uppercol]

-- upper edge with n processors

(endpoint.array[i][uppercol] = col1)
SEQ
    endpoint.array[i][upperrow] :=
        endpoint.array[i][upperrow]
    endpoint.array[i][lowerrow] :=
        endpoint.array[i][lowerrow] + 1
    endpoint.array[i][lowercol] :=
        endpoint.array[i][lowercol] + 1

-- lower edge with n processors

(endpoint.array[i][lowercol] = col2)
SEQ
    endpoint.array[i][upperrow] :=
        endpoint.array[i][upperrow]
    endpoint.array[i][lowerrow] :=
        endpoint.array[i][lowerrow] + 1
    endpoint.array[i][uppercol] :=
        endpoint.array[i][uppercol]
    endpoint.array[i][lowercol] :=
        endpoint.array[i][lowercol] + 1

-- middle box with n processors

TRUE
SEQ
    endpoint.array[i][upperrow] :=
        endpoint.array[i][upperrow]
    endpoint.array[i][lowerrow] :=
        endpoint.array[i][lowerrow] + 1
    endpoint.array[i][uppercol] :=
        endpoint.array[i][uppercol]
    endpoint.array[i][lowercol] :=
        endpoint.array[i][lowercol] + 1
TRUE  -- if there is only 1 processor the image endpoints
   -- are the
   -- partition endpoints since there is only one chunk
SEQ
   endpoint.array[0][upperrow]:= endpoints[upperrow]
   endpoint.array[0][uppercol]:= endpoints[uppercol]
   endpoint.array[0][lowerrow]:= endpoints[lowerrow]
   endpoint.array[0][lowercol]:= endpoints[lowercol]

-- compute data
-- distribute data to each CPU
row.max := total.row + 1  -- total row size of image
col.max := total.col + 1  -- total col size of image
-- send endpoints to CPUs
SEQ index = 0 FOR max.cpu
   SEQ index2 = 0 FOR number.endpoint
      chan.out ! endpoint.array[index][index2]

-- send row and column size information to CPUs
chan.out ! row.max
chan.out ! col.max
chan.out ! row1
chan.out ! col1

-- send image array to CPUs
SEQ index = 0 FOR row.max
   SEQ index2 = 0 FOR col.max
      chan.out ! image.array[index][index2]

-- receive histogram from CPUs
SEQ index = 0 FOR max.histogram
   SEQ index2 = 0 FOR number.histogram
   SEQ
      chan.in ? main.histogram[index][index2]

clock ? time.end  -- stop timing
-- compute total time in seconds
IF
   time.end AFTER time.start
      runtime := time.end MINUS time.start
TRUE

runtime:= time.start MINUS time.end
quotient:= runtime / ticks.per.second
remainder:= runtime REM ticks.per.second

output.data(from.user.filer[0], to.user.filer[0],
            main.histogram, quotient, remainder)

-- SC chunk.tsr

-- SC chunkN.tsr

-- Vars
VAL max.cpu IS 8:
-- define link/channel numbers
VAL link0out IS 0 :
VAL link1out IS 1 :
VAL link2out IS 2 :
VAL link3out IS 3 :
VAL link0in IS 4 :
VAL link1in IS 5 :
VAL link2in IS 6 :
VAL link3in IS 7 :

-- define link connection
VAL from.link.in IS [link0in, link3in, link3in, link3in, link0in, link3in, link3in, link3in]:
VAL from.link.out IS [link2out, link2out, link2out, link1out, link2out, link2out, link2out]:
VAL to.link.in IS [link2in, link2in, link2in, link1in, link2in, link2in, link2in]:
VAL to.link.out IS [link0out, link3out, link3out, link3out, link0out, link3out, link3out, link3out]:

[max.cpu]CHAN OF ANY from.root, to.root:

PLACED PAR
PLACED PAR cpu = 0 FOR (max.cpu - 1)
PROCESSOR cpu T4
    PLACE from.root [cpu] AT from.link.in [cpu]:
    PLACE from.root [cpu+1] AT from.link.out [cpu]:
    PLACE to.root [cpu] AT to.link.out [cpu]:
PLACE to.root [cpu+1] AT to.link.in [cpu]:
  chunk(cpu, from.root[cpu], from.root[cpu+1],
        to.root[cpu+1], to.root[cpu])
PROCESSOR (max.cpu - 1) T4
PLACE from.root [max.cpu - 1] AT from.link.in [max.cpu-1]:
PLACE to.root [max.cpu - 1] AT to.link.out [max.cpu-1]:
  chunkN(from.root [max.cpu - 1], to.root[max.cpu - 1])

PROC chunk(VAL INT cpu, CHAN OF ANY from.root1,
            from.root2, to.root1, to.root2)

-- vars
-- VARs
-- Constants
VAL max.cpu IS 8:         -- number of processors
VAL upperrow IS 0:        -- upper left ..
VAL uppercol IS 1:        -- corner of image
VAL lowerrow IS 2:        -- lower right ..
VAL lowercol IS 3:        -- corner of image
VAL northwest IS 0:
VAL north IS 1:
VAL northeast IS 2:
VAL west IS 3:            -- the directions of the histogram
VAL east IS 4:
VAL southwest IS 5:
VAL south IS 6:
VAL southeast IS 7:
VAL max.histogram IS 129: -- size of the histogram
VAL number.endpoint IS 4: -- four endpoints of the chunk
VAL number.histogram IS 8: -- the histogram holds eight
                              -- different directions
VAL max.row IS 238:       -- maximum size..
VAL max.col IS 256:       -- of the image

-- Arrays
[max.cpu][number.endpoint]INT end.array:   -- all endpoints
                                          --for each chunk
[number.endpoint]INT endpoints:   -- endpoints for this
                                    -- chunk only
[max.histogram][number.histogram]INT histogram:
                                          -- histogram for this chunk
[max.histogram][number.histogram]INT other.histogram:
                                          -- accumulated histogram
[number.histogram]INT diff: -- difference array used to
calculate histogram

[max.row][max.col]INT i.array: -- holds the image

-- Integers
INT which.cpu: -- tells which CPU each chunk is
INT value: -- difference value to mark place in histogram to
            -- add 1 to
INT index, index2, i, j: -- index variables
INT row.max, col.max: -- size of image
INT row.element, col.element: -- size of chunk to calculate
INT row1, col1: -- upper left corner of image

SEQ
-- initialize histogram to 0
SEQ index = 0 FOR max.histogram
    SEQ index2 = 0 FOR number.histogram
        histogram[index][index2]:= 0

-- receive endpoints from host
SEQ index = 0 FOR max.cpu
    SEQ index2 = 0 FOR number.endpoint
    SEQ
        --receive from host
        from.root1 ? end.array[index][index2]
        --send to next CPU
        from.root2 ! end.array[index][index2]

-- extract endpoints for this cpu
which.cpu := cpu
SEQ index = 0 FOR number.endpoint
    endpoints[index]:= end.array[which.cpu][index]

-- receive row and column sizes from host
SEQ
    --receive from host
    from.root1 ? row.max
    from.root1 ? col.max
    from.root1 ? row1
    from.root1 ? col1
    --send to next CPU
    from.root2 ! row.max
    from.root2 ! col.max
from.root2 ! row1
from.root2 ! col1

-- receive image array from host
SEQ index = 0 FOR row.max
SEQ index2 = 0 FOR col.max
SEQ
    -- receive from host
    from.root1 ? i.array[index][index2]
    -- send to next CPU
    from.root2 ! i.array[index][index2]

-- do vardig operation
---*****************************************************************************
    -- * This portion is going to perform the Vardig operator
    -- * on the array (which is part the whole image).
    -- * After calculation, it will store the results in terms
    -- * of the histogram.
---*****************************************************************************

row.element:= ((endpoints[lowerrow] - endpoints[upperrow]) - 1)
col.element:= ((endpoints[lowercol] - endpoints[uppercol]) - 1)
SEQ i = (endpoints[upperrow]-(row1-1)) FOR (row.element)
SEQ j = (endpoints[uppercol]-(col1-1)) FOR (col.element)
SEQ
    -- Vardig operator

diff[northwest] := i.array[i-1][j-1] - i.array[i][j]
diff[north] := i.array[i-1][j] - i.array[i][j]
diff[northeast] := i.array[i-1][j+1] - i.array[i][j]
diff[west] := i.array[i][j-1] - i.array[i][j]
diff[east] := i.array[i][j+1] - i.array[i][j]
diff[southwest] := i.array[i+1][j-1] - i.array[i][j]
diff[south] := i.array[i+1][j] - i.array[i][j]
diff[southeast] := i.array[i+1][j+1] - i.array[i][j]

    -- Since the result is from -64 to 63, 64 is added to it
    -- to be able to put into the histogram array.
SEQ index = 0 FOR number.histogram
diff[index] := diff[index] + 64
-- compute histogram for this chunk
SEQ index = 0 FOR number.histogram
SEQ
  value:= diff[index]
  histogram[value][index]:= histogram[value][index] + 1

-- send histogram to host
SEQ index = 0 FOR max.histogram
SEQ index2 = 0 FOR number.histogram
  -- receive accumulated histogram from other CPUs
  to.root1 ? other.histogram[index][index2]

  -- add the histogram for this chunk to the accumulated histogram
SEQ index = 0 FOR max.histogram
SEQ index2 = 0 FOR number.histogram
  other.histogram[index][index2]:= other.histogram[index][index2] +
      histogram[index][index2]
SEQ index = 0 FOR max.histogram
SEQ index2 = 0 FOR number.histogram
  -- send accumulated histogram on to host
  to.root2 ! other.histogram[index][index2]

PROC chunkN (CHAN OF ANY from.root, to.root)
  -- vars
  -- VARs
  -- Constants
VAL max.cpu IS 8: -- number of processors
VAL upperrow IS 0: -- upper left ..
VAL uppercol IS 1: -- corner of image
VAL lowerrow IS 2: -- lower right ..
VAL lowercol IS 3: -- corner of image
VAL northwest IS 0:
VAL north IS 1:
VAL northeast IS 2:
VAL west IS 3: -- the different directions of the
VAL southwest IS 5:
VAL south IS 6:
VAL southeast IS 7:
VAL max.histogram IS 129:       -- size of the histogram
VAL number.endpoint IS 4:        -- four endpoints of the chunk
VAL number.histogram IS 8:       -- the histogram has eight
       -- directions
VAL max.row IS 238:              -- maximum size ..
VAL max.col IS 256:              -- of the image

-- Arrays
[max.cpu][number.endpoint]INT end.array:       -- endpoints for
       -- all the chunks
[number.endpoint]INT endpoints:        -- endpoints for this
       -- chunk
[max.histogram][number.histogram]INT histogram:
       -- histogram for this chunk
[number.histogram]INT diff:          -- difference values used to
       -- calculate histogram
[max.row][max.col]INT i.array:       -- holds the image

-- Integers
INT which.cpu:       -- tells which CPU this is
INT value:          -- holds value of histogram to add 1 to
INT index, index2, i, j:    -- loop variables
INT row.max, col.max:      -- row and col size of the image
INT row.element, col.element:      -- row and col size of the
       -- chunk
INT row1, col1:       -- upper left corner of the image

SEQ
       -- initialize histogram to 0
SEQ index = 0 FOR max.histogram
       SEQ index2 = 0 FOR number.histogram
       histogram[index][index2]:= 0

-- receive endpoints from host
SEQ index = 0 FOR max.cpu
       SEQ index2 = 0 FOR number.endpoint
       SEQ
       from.root ? end.array[index][index2]

-- extract specific endpoints for this cpu
SEQ index = 0 FOR number.endpoint
       endpoints[index]:= end.array[max.cpu-1][index]
-- receive max row and column sizes from host
SEQ
  from.root ? row.max
  from.root ? col.max
  from.root ? row1
  from.root ? col1

-- receive image array from host
SEQ index = 0 FOR row.max
SEQ index2 = 0 FOR col.max
  from.root ? i.array[index][index2]

-- do vardig operation
  ***************************************
  -- * This portion is going to perform the Vardig operator
  -- * on the array (which is part the whole image).
  -- * After calculation, it will store the results in terms
  -- * of the histogram.
  ***************************************

  row.element:= ((endpoints[lowerrow] -
                 endpoints[upperrow]) - 1)
  col.element:= ((endpoints[lowercol] -
                 endpoints[uppercol]) - 1)
SEQ i = (endpoints[upperrow]-(row1-1)) FOR (row.element)
SEQ j = (endpoints[uppercol]-(col1-1)) FOR (col.element)
SEQ
  -- Vardig operator

  diff[northwest] := i.array[i-1][j-1] - i.array[i][j]
  diff[north] := i.array[i-1][j] - i.array[i][j]
  diff[northeast] := i.array[i-1][j+1] - i.array[i][j]
  diff[west] := i.array[i][j-1] - i.array[i][j]
  diff[east] := i.array[i][j+1] - i.array[i][j]
  diff[southwest] := i.array[i+1][j-1] - i.array[i][j]
  diff[south] := i.array[i+1][j] - i.array[i][j]
  diff[southeast] := i.array[i+1][j+1] - i.array[i][j]

  -- Since the result is from -64 to 63, 64 is added to it
  -- to be able to put into the histogram array
SEQ index = 0 FOR number.histogram
diff[index] := diff[index] + 64

-- calculate histogram for this chunk
SEQ index = 0 FOR number.histogram
SEQ
    value:= diff[index]
    histogram[value][index]:= histogram[value][index] + 1

-- send histogram to host
SEQ index = 0 FOR max.histogram
    SEQ index2 = 0 FOR number.histogram
    to.root ! histogram[index][index2]
program vardig;
const
  processor = 8; (* number of parallel processors to be used *)
  upperrow = 1; (* upper left .. *)
  uppercol = 2; (* corner of the image *)
  lowerrow = 3; (* lower right .. *)
  lowercol = 4; (* corner of the image *)
  max_row = 238; (* maximum size .. *)
  max_col = 256; (* of the image array *)
  max_histogram = 128; (* size of the histogram *)
  number_endpoint = 4; (* 4 endpoints of the image array *)
  number_histogram = 8; (* 8 directions of the histogram *)
  northwest = 1; (* the directions of the histogram *)
  north = 2; (* the directions of the histogram *)
  northeast = 3;
  west = 4;
  east = 5;
  southwest = 6;
  south = 7;
  southeast = 8;

type
  timetype = array [1..processor] of real;
  locktype = array[1..processor] of real;
  end_point_type = array [1..processor, 1..number_endpoint] of integer;
  main_histogram_type = array [1..number_histogram, 0..max_histogram] of integer;
  image_array_type = array [1..max_row, 1..max_col] of integer;
  diff_type = array [1..number_histogram] of integer;
  end_type = array [1..number_endpoint] of integer;

var
  arraytime: timetype; (* times of each processor in vardig *)
  arraylock: locktype; (* times of the lock for each
end_point: end_point_type; (* holds endpoints of each chunk *)

ends: end_type; (* holds endpoints of the image *)
main_histogram: main_histogram_type; (* histogram *)
image_array: image_array_type; (* holds the image *)
seconds, elapsed, startime, finaltime: real; (* timing variables *)

startpar, endpar, parseconds: real; (* timing variable *)
total_row, total_col: integer; (* row and col size of image *)
row1, row2, col1, col2: integer; (* row and col size of chunk *)
temp_processor: longint; (* number of processors *)

.jump

(* The next 7 functions and procedures are provided by the system. Secnds is a timing function so the time a procedure needs may be determined. M_lock and m_unlock are locks so that shared data may be accessed without race conditions. M_set_procs sets the number of parallel processors desired for processing. M_fork breaks the process into the number of processes desired and gets the system set to process code in parallel. M_get_myid returns the ID of the particular process. M_kill_procs stops the parallel processes and returns the system back to one processor. *)

function secnds (var r:real): real;
  cexternal;

procedure m_lock;
  cexternal;

procedure m_unlock;
  cexternal;

procedure m_set_procs (var i:longint);
  cexternal;

procedure m_fork(procedure a);
  cexternal;

function m_get_myid : longint;
procedure readin_image;

var
  i, j: integer; (* loop variables *)
  realnum: integer; (* image number *)
  number: integer; (* endpoints of the image *)

begin
  for i := 1 to number_endpoint do
    begin
      readln(number);
      ends[i] := number;
    end;
  for i := 1 to ends[lowerrow]-(ends[upperrow]-1) do
    for j := 1 to ends[lowercol]-(ends[uppercol]-1) do
      begin
        readln(realnum);
        image_array[i][j] := realnum;
      end;
end; (* readin_image *)

procedure initializations;

var
  i, j: integer; (* loop variables *)

begin
  for i := 1 to number_histogram do
    for j := 1 to max_histogram do
main_histogram[i][j]:= 0;
for i := 1 to processor do
arraylock[i]:= 0.0;
readin_image;
row1:= ends[upperrow];
row2:= ends[lowerrow];
col1:= ends[uppercol];
col2:= ends[lowercol];
total_row:= row2 - row1;
total_col:= col2 - col1;
end; (* initializations *)

procedure process_partition;
var
  i,j: integer; (* loop variables *)
  rowpart, colpart: integer; (* used to calculate number of chunks *)
  row_element, col_element: integer; (* row and col size for chunks *)
  last_row, last_col: integer; (* row and col size for last chunk *)
  temp_processor: integer; (* number of processors *)
  current_row, current_col: integer; (* used to calculate the .. *)
  number_part: integer; (* the size of chunks *)
begin
  (* determine number of chunks to divide image into *)
  rowpart:= 1;
colpart:= 1;
temp_processor:= processor;
while (temp_processor >= 2) do
begin
  if temp_processor >= 2 then
    rowpart:= rowpart*2;
temp_processor:= temp_processor div 2;
if temp_processor >= 2 then
  colpart:= colpart*2;
temp_processor:= temp_processor div 2;
end;
if rowpart > 1 then (* if more than one processor is
  being processed *)
begin

  (* determine the size of the rows and columns for each
  chunk *)
  row_element:= total_row div rowpart;
col_element:= total_col div colpart;

  (* if the image doesn't divide evenly into the desired
  chunks last_row and last_col will hold the irregular
  chunk and be processed by the last processor *)
  last_row:= (total_row - ((rowpart-1) * row_element));
  last_col:= (total_col - ((colpart-1) * col_element));

  (* find the endpoints of each chunk *)
current_row:= rowl1;
current_col:= col1;
number_part:= 0;
for i:= 1 to (rowpart-1) do
begin
  for j:= 1 to (colpart-1) do
begin
    number_part:= number_part + 1;
    end_point [number_part][upperrow]:= current_row;
    end_point [number_part][lowerrow]:= current_row +
    row_element;
    end_point [number_part][uppercol]:= current_col;
    end_point [number_part][lowercol]:= current_col +
    col_element;
  current_col:= current_col + col_element;
end;
number_part:= number_part + 1;
  end_point [number_part][upperrow]:= current_row;
  end_point [number_part][lowerrow]:= current_row +
  row_element;
  end_point [number_part][uppercol]:= current_col;
  end_point [number_part][lowercol]:= current_col +
  last_col;
current_col := col1;
current_row := current_row + row_element;
end;
for i := 1 to (colpart - 1) do
begin
  number_part := number_part + 1;
  end_point[number_part][upperrow] := current_row;
  end_point[number_part][lowerrow] := current_row + last_row;
  end_point[number_part][uppercol] := current_col;
  end_point[number_part][lowercol] := current_col + col_element;
end;
number_part := number_part + 1;
end_point[number_part][upperrow] := current_row;
end_point[number_part][lowerrow] := current_row + last_row;
end_point[number_part][uppercol] := current_col;
end_point[number_part][lowercol] := current_col + last_col;
current_col := col1;
current_row := current_row + row_element;

(* determine which of the 11 cases the chunk represents and then based on the case, compute the extra rows and/or columns needed for the processing of that chunk *)
for i := 1 to processor do
begin

  (* left side with 2 processors *)
  if (end_point[i][upperrow] = row1) and
    (end_point[i][uppercol] = col1) and
    (end_point[i][lowercol] = col2) then
    end_point[i][lowerrow] := end_point[i][lowerrow] + 1;

  (* upper left corner with n processors *)
  else
    if (end_point[i][upperrow] = row1) and
      (end_point[i][uppercol] = col1) then
      begin
        end_point[i][lowerrow] := end_point[i][lowerrow] + 1;
end_point[i][lowercol] := end_point[i][lowercol] + 1;
end

(* right side with 2 processors *)
else
if (end_point[i][lowerrow] = row2) and
    (end_point[i][uppercol] = col1) and
    (end_point[i][lowercol] = col2) then
    end_point[i][upperrow] := end_point[i][upperrow]

(* upper right corner with n processors *)
else
if (end_point[i][lowerrow] = row2) and
    (end_point[i][uppercol] = col1) then
    begin
        end_point[i][upperrow] := end_point[i][upperrow];
        end_point[i][lowercol] := end_point[i][lowercol] + 1;
    end

(* lower left corner with n processors *)
else
if (end_point[i][upperrow] = row1) and
    (end_point[i][lowercol] = col2) then
    begin
        end_point[i][uppercol] := end_point[i][uppercol];
        end_point[i][lowerrow] := end_point[i][lowerrow] + 1;
    end

(* lower right corner with n processors *)
else
if (end_point[i][lowerrow] = row2) and
    (end_point[i][lowercol] = col2) then
    begin
        end_point[i][upperrow] := end_point[i][upperrow];
        end_point[i][uppercol] := end_point[i][uppercol];
    end

(* left edge with n processors *)
else
if (end_point[i][upperrow] = row1) then
    begin
        end_point[i][lowerrow] := end_point[i][lowerrow] + 1;
        end_point[i][lowercol] := end_point[i][lowercol] + 1;
end_point[i][uppercol] := end_point[i][uppercol];

end

(* right edge with n processors *)
else
  if (end_point[i][lowerrow] = row2) then
    begin
      end_point[i][upperrow] := end_point[i][upperrow];
      end_point[i][lowercol] := end_point[i][lowercol] + 1;
      end_point[i][uppercol] := end_point[i][uppercol];
    end

(* upper edge with n processors *)
else
  if (end_point[i][uppercol] = col1) then
    begin
      end_point[i][upperrow] := end_point[i][upperrow];
      end_point[i][lowerrow] := end_point[i][lowerrow] + 1;
      end_point[i][lowercol] := end_point[i][lowercol] + 1;
    end

(* lower edge with n processors *)
else
  if (end_point[i][lowercol] = col2) then
    begin
      end_point[i][upperrow] := end_point[i][upperrow];
      end_point[i][lowerrow] := end_point[i][lowerrow] + 1;
      end_point[i][uppercol] := end_point[i][uppercol];
    end

(* middle box with n processors *)
else begin
  end_point[i][upperrow] := end_point[i][upperrow];
  end_point[i][lowerrow] := end_point[i][lowerrow] + 1;
  end_point[i][uppercol] := end_point[i][uppercol];
  end_point[i][lowercol] := end_point[i][lowercol] + 1;
end
end; (* for *)
end (* if more than one processor *)
else (* only one processor *)
begin
  end_point[1][upperrow] := ends[upperrow];
  end_point[1][uppercol] := ends[uppercol];
  end_point[1][lowerrow] := ends[lowerrow];
end_point[1][lowercol]:= ends[lowercol];
end
end; (* process partition *)

procedure vardig_calculations (proc:longint);
var
i,j,k: integer;       (* loop variables *)
diff: diff_type;      (* holds vardig calculations *)
temp_hist: main_histogram_type; (* local copy of histogram *)
value: integer;       (* holds value of the diff array *)
seconds, starttime, finaltime, elapsed: real; (* timing variables *)
startlock, endlock, locktime: real; (* timing variables *)

begin
for i:= 1 to number_histogram do
  for j:= 1 to max_histogram do
    temp_hist[i][j] := 0;
elapsed:= 0.0;
starttime:= secnds(elapsed);
for i:= (end_point[proc+1][upperrow])-(rowl-2) to (end_point[proc+1][lowerrow]-row1) do
  for j:= (end_point[proc+1][uppercol])-(colt-2) to (end_point[proc+1][lowercol]-coll) do
    begin
      (* vardig operation *)
      diff[northwest]:= image_array[i-1][j-1] - image_array[i][j];
      diff[north]:= image_array[i-1][j] - image_array[i][j];
      diff[northeast]:= image_array[i-1][j+1] - image_array[i][j];
      diff[west]:= image_array[i][j-1] - image_array[i][j];
      diff[east]:= image_array[i][j+1] - image_array[i][j];
diff[southwest] := image_array[i+1][j-1] - image_array[i][j];
diff[south] := image_array[i+1][j] - image_array[i][j];
diff[southeast] := image_array[i+1][j+1] - image_array[i][j];

(* the result is between -63 and 64 so 64 is added to it to make the result between 1 and 128 so it can be put into the histogram array *)
for k:= 1 to number_histogram do
diff[k] := diff[k] + 64;

(* Add the calculations from the diff array to the local copy of the histogram array *)
for k:= 1 to number_histogram do
begin
    value := diff[k];
    temp_hist[k][value] := temp_hist[k][value] + 1;
end;

startlock := secnds(elapsed);
(* Lock the main histogram and add this newly created one to it *)
m_lock;
for i:= 1 to number_histogram do
    for j:= 1 to max_histogram do
        main_histogram[i][j] := main_histogram[i][j] + temp_hist[i][j];

m_unlock;
endlock := secnds(elapsed);
locktime := endlock - startlock;
arraylock[proc+1] := locktime;

finaltime := secnds(elapsed);
seconds := finaltime - starttime;
arraytime[proc + 1] := seconds;
end; (* vardig calculations *)

(* Parallel_forks runs the parallelized procedure until all processors have processed their data *)

procedure parallel_forks;
var
nprocs: longint; (* number of processors *)  
id: integer; (* ID number of process *)

begin
    nprocs:= processor; (* number of processors desired *)
    id:= m_get_myid; (* which process this is *)
    while (id < nprocs) do (* call vardig_calculations until *)
    begin (* all parallel processors have been called *)
        vardig_calculations(id);
        id:= id+nprocs;
    end;
end; (* parallel forks *)

(******************************************************************************
    * print out the time, in seconds, of the algorithm and the
    * histogram which has been calculated.
    ******************************************************************************)
procedure print_results (time,partime:real);
var
    i, j: integer; (* loop variables *)

begin
    writeln;
    writeln('Processing a ',row2-row1:3,' X ',co12-co11:3,' image.');
    writeln;
    writeln('Using ',processor:2,' processors and');
    writeln('timing everything, including the locks.');
    write('The total number of seconds is: ');
    writeln(time:15:5);
    writeln;
    write('The time for just the parallel part is: ');
    writeln(partime:15:5);
    writeln;
    writeln('The time for each processor is: ');
    for i:= 1 to processor do
    begin
        write('for process ',i:2,': ');
        writeln(arraytime[i]:15:8);
    end;
    writeln;
writeln('The time for each lock is:');
for i:= 1 to processor do
begin
    write('for process ',i:2,:');
    writeln(arraylock[i]:15:8);
end;
writeln;
writeln;
writeln;
(* print out histogram *)
write(' NW  N  NE');
writeln(' W  E  SW  S  SE');
writeln;
for i:= 0 to max_histogram do
begin
    write('mag=',i:4,:');
    for j:= 1 to number_histogram do
        write(main_histogram[j][i]:5,:');
    writeln;
end;
end; (* print_results *)

begin (* vardig *)
temp_processor:= processor;
initializations;
elapsed:= 0.0;
starttime:= secnds(elapsed);
process_partition;
m_set_procs(temp_processor); (* set number of processors
to be used *)

startpar:= secnds(elapsed);
m_fork(parallel_forks);
endpar:= secnds(elapsed);
parseconds:= endpar - startpar;
m_kill_procs; (* stop all parallel processes *)
finaltime:= secnds(elapsed);
seconds:= finaltime - starttime;
print_results(seconds,parseconds);
end. (*vardig*)
## Transputer Spreadsheet

<table>
<thead>
<tr>
<th>TRANSN &amp; Num</th>
<th>proc img6x6</th>
<th>img50x50</th>
<th>img237x255</th>
<th>rel6x6</th>
<th>rel50x50</th>
<th>rel237/255</th>
</tr>
</thead>
<tbody>
<tr>
<td>everything</td>
<td>1</td>
<td>0.331</td>
<td>0.6756</td>
<td>10.5107</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>par_vardig</td>
<td>1</td>
<td>0.331</td>
<td>0.6756</td>
<td>10.5108</td>
<td></td>
<td></td>
</tr>
<tr>
<td>proc_time</td>
<td>1</td>
<td>0.59</td>
<td>0.569</td>
<td>9.1456</td>
<td></td>
<td></td>
</tr>
<tr>
<td>empty_tota</td>
<td>1</td>
<td>0.27</td>
<td>0.99</td>
<td>1.178</td>
<td></td>
<td></td>
</tr>
<tr>
<td>comm_time</td>
<td>1</td>
<td>0.27</td>
<td>0.989</td>
<td>1.1779</td>
<td></td>
<td></td>
</tr>
<tr>
<td>everything</td>
<td>2</td>
<td>0.701</td>
<td>0.4646</td>
<td>6.446</td>
<td>0.4721825</td>
<td>1.4541541</td>
</tr>
<tr>
<td>par_vardig</td>
<td>2</td>
<td>0.7</td>
<td>0.4645</td>
<td>6.446</td>
<td></td>
<td></td>
</tr>
<tr>
<td>proc_time</td>
<td>2</td>
<td>0.36</td>
<td>0.294</td>
<td>4.947</td>
<td></td>
<td></td>
</tr>
<tr>
<td>empty_tota</td>
<td>2</td>
<td>0.661</td>
<td>0.1741</td>
<td>1.10622</td>
<td></td>
<td></td>
</tr>
<tr>
<td>comm_time</td>
<td>2</td>
<td>0.659</td>
<td>0.1738</td>
<td>1.1062</td>
<td></td>
<td></td>
</tr>
<tr>
<td>everything</td>
<td>4</td>
<td>0.1473</td>
<td>0.3966</td>
<td>4.642</td>
<td>2.2471147</td>
<td>1.7034795</td>
</tr>
<tr>
<td>par_vardig</td>
<td>4</td>
<td>0.1469</td>
<td>0.3962</td>
<td>4.642</td>
<td></td>
<td></td>
</tr>
<tr>
<td>proc_time</td>
<td>4</td>
<td>0.21</td>
<td>0.1501</td>
<td>2.4738</td>
<td></td>
<td></td>
</tr>
<tr>
<td>empty_tota</td>
<td>4</td>
<td>0.1437</td>
<td>0.2519</td>
<td>1.11506</td>
<td></td>
<td></td>
</tr>
<tr>
<td>comm_time</td>
<td>4</td>
<td>0.1433</td>
<td>0.2515</td>
<td>1.11502</td>
<td></td>
<td></td>
</tr>
<tr>
<td>everything</td>
<td>8</td>
<td>0.3057</td>
<td>0.4897</td>
<td>2.15255</td>
<td>1.0827608</td>
<td>1.3796201</td>
</tr>
<tr>
<td>par_vardig</td>
<td>8</td>
<td>0.3048</td>
<td>0.4887</td>
<td>2.15248</td>
<td></td>
<td></td>
</tr>
<tr>
<td>proc_time</td>
<td>8</td>
<td>0.7</td>
<td>0.72</td>
<td>1.2366</td>
<td></td>
<td></td>
</tr>
<tr>
<td>empty_tota</td>
<td>8</td>
<td>0.2984</td>
<td>0.407</td>
<td>1.13097</td>
<td></td>
<td></td>
</tr>
<tr>
<td>comm_time</td>
<td>8</td>
<td>0.2974</td>
<td>0.406</td>
<td>1.13087</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Table: Sequent (old) Spreadsheet

<table>
<thead>
<tr>
<th>SEQOLD</th>
<th>Num</th>
<th>proc</th>
<th>img6x6</th>
<th>img50x50</th>
<th>img237x255</th>
<th>rel6x6</th>
<th>rel50x50</th>
<th>rel237x255</th>
</tr>
</thead>
<tbody>
<tr>
<td>everything</td>
<td>1</td>
<td>0.0820312</td>
<td>3.242187</td>
<td>78.83985</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>par_vardig</td>
<td>1</td>
<td>0.0585937</td>
<td>3.167969</td>
<td>78.77734</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>proc_time</td>
<td>1</td>
<td>0.3125</td>
<td>3.050781</td>
<td>76.37891</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>everything</td>
<td>2</td>
<td>0.171875</td>
<td>1.921875</td>
<td>42.83984</td>
<td>0.4772727</td>
<td>1.6869916</td>
<td>1.8403395</td>
<td></td>
</tr>
<tr>
<td>par_vardig</td>
<td>2</td>
<td>0.0898437</td>
<td>1.800781</td>
<td>42.24219</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>proc_time</td>
<td>2</td>
<td>0.027343</td>
<td>1.6875</td>
<td>41.42188</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>everything</td>
<td>2</td>
<td>0.01953</td>
<td>1.636719</td>
<td>41.41016</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>par_vardig</td>
<td>4</td>
<td>0.359375</td>
<td>1.558594</td>
<td>30.10937</td>
<td>0.2282608</td>
<td>2.0801998</td>
<td>2.6184490</td>
<td></td>
</tr>
<tr>
<td>proc_time</td>
<td>4</td>
<td>0.097656</td>
<td>1.167969</td>
<td>28.03125</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>everything</td>
<td>4</td>
<td>0.0195312</td>
<td>1.128906</td>
<td>28.55078</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>par_vardig</td>
<td>4</td>
<td>0.0195312</td>
<td>1.140625</td>
<td>28.30078</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>proc_time</td>
<td>4</td>
<td>0.0195312</td>
<td>1.089844</td>
<td>28.26172</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>everything</td>
<td>8</td>
<td>0.75</td>
<td>1.867187</td>
<td>28.39062</td>
<td>0.109375</td>
<td>1.7364018</td>
<td>2.7769682</td>
<td></td>
</tr>
<tr>
<td>par_vardig</td>
<td>8</td>
<td>0.4101562</td>
<td>1.53125</td>
<td>28.10937</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>proc_time</td>
<td>8</td>
<td>0</td>
<td>1.03125</td>
<td>26.75</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>0.0117187</td>
<td>1.070312</td>
<td>27.08984</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>0.0195312</td>
<td>1.070312</td>
<td>26.92969</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>0.0195312</td>
<td>1.046875</td>
<td>27.16016</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>0.0195312</td>
<td>1.070312</td>
<td>27.12891</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>0.0195312</td>
<td>1.058594</td>
<td>26.67969</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>0.0292968</td>
<td>1.117187</td>
<td>26.97656</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>0.0195312</td>
<td>1.101562</td>
<td>26.78906</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Sequent (new) spreadsheet

<table>
<thead>
<tr>
<th>SEQNEW</th>
<th>Num_proc</th>
<th>img6x6</th>
<th>img50x50</th>
<th>img237x255</th>
<th>rel6x6</th>
<th>rel50x50</th>
<th>rel237x255</th>
</tr>
</thead>
<tbody>
<tr>
<td>everything</td>
<td>1</td>
<td>0.16406</td>
<td>2.25</td>
<td>52.57031</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>par_vardig</td>
<td>1</td>
<td>0.14062</td>
<td>2.21875</td>
<td>52.53125</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>proc_time</td>
<td>1</td>
<td>0.09375</td>
<td>2.171875</td>
<td>52.47656</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lock_time</td>
<td>1</td>
<td>0.0625</td>
<td>0.0546875</td>
<td>0.0625</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>everything</td>
<td>2</td>
<td>0.35156</td>
<td>1.36719</td>
<td>26.5625</td>
<td>0.4666628</td>
<td>1.6457112</td>
<td>1.9791175</td>
</tr>
<tr>
<td>par_vardig</td>
<td>2</td>
<td>0.25</td>
<td>1.28125</td>
<td>26.5313</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>proc_time</td>
<td>2</td>
<td>0.078125</td>
<td>1.179687</td>
<td>26.28125</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lock_time</td>
<td>2</td>
<td>0.0625</td>
<td>0.09375</td>
<td>0.0546875</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>everything</td>
<td>4</td>
<td>0.65625</td>
<td>1.20313</td>
<td>13.79688</td>
<td>0.2499961</td>
<td>1.8701220</td>
<td>3.8103042</td>
</tr>
<tr>
<td>par_vardig</td>
<td>4</td>
<td>0.47656</td>
<td>1</td>
<td>13.57813</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>proc_time</td>
<td>4</td>
<td>0.0703125</td>
<td>0.7265625</td>
<td>13.17188</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lock_time</td>
<td>4</td>
<td>0.125</td>
<td>0.0703125</td>
<td>0.0703125</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>everything</td>
<td>8</td>
<td>1.3125</td>
<td>1.59375</td>
<td>7.96094</td>
<td>0.1249980</td>
<td>1.4117647</td>
<td>6.6035304</td>
</tr>
<tr>
<td>par_vardig</td>
<td>8</td>
<td>0.94531</td>
<td>1.21875</td>
<td>7.53906</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>proc_time</td>
<td>8</td>
<td>0.073125</td>
<td>0.328125</td>
<td>6.664062</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lock_time</td>
<td>8</td>
<td>0.5</td>
<td>0.3828125</td>
<td>7.078125</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>everything</td>
<td>8</td>
<td>0.4296875</td>
<td>0.6328125</td>
<td>7.007812</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>par_vardig</td>
<td>8</td>
<td>0.1328125</td>
<td>0.5703125</td>
<td>6.703125</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>proc_time</td>
<td>8</td>
<td>0.3125</td>
<td>0.5</td>
<td>6.890625</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lock_time</td>
<td>8</td>
<td>0.359375</td>
<td>0.6875</td>
<td>6.757812</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>everything</td>
<td>8</td>
<td>0.234375</td>
<td>0.4375</td>
<td>6.828125</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>par_vardig</td>
<td>8</td>
<td>0.1796875</td>
<td>0.7578125</td>
<td>6.945312</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>proc_time</td>
<td>8</td>
<td>0.0625</td>
<td>0.0703125</td>
<td>0.0703125</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lock_time</td>
<td>8</td>
<td>0.4765625</td>
<td>0.125</td>
<td>0.4296875</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>everything</td>
<td>8</td>
<td>0.40625</td>
<td>0.34375</td>
<td>0.3828125</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>par_vardig</td>
<td>8</td>
<td>0.1171875</td>
<td>0.3046875</td>
<td>0.0703125</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>proc_time</td>
<td>8</td>
<td>0.2890625</td>
<td>0.2265625</td>
<td>0.25</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lock_time</td>
<td>8</td>
<td>0.3515625</td>
<td>0.4296875</td>
<td>0.1328125</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>everything</td>
<td>8</td>
<td>0.2265625</td>
<td>0.1328125</td>
<td>0.1953125</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>par_vardig</td>
<td>8</td>
<td>0.171875</td>
<td>0.46875</td>
<td>0.3046875</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>