Synthetic Aperture Radar Processing at the Arctic Region Supercomputing Center

Tom Logan, *Arctic Region Supercomputing Center*

ABSTRACT: *A scalable parallel Synthetic Aperture Radar (SAR) software correlator has been implemented on the T3E-900 at the Arctic Region Supercomputing Center. The implementation produces a full scene (100-km2) image in less than 90 seconds, realizing a 26x speed-up over a similar serial implementation using only 28 processors. This algorithm was included in a SAR interferometric processing system capable of creating interferograms in little more than 10 minutes. This paper will discuss implementation details of this system focusing on the latency-hiding and parallel techniques that allowed an overall 15x speed-up for what initially seemed a largely serial code.*

1 Introduction

This paper will discuss the parallel implementation of several SAR processing tools at the Arctic Region Supercomputing Center (ARSC). An introduction to SAR and the hardware systems used will be covered in this section. The results of parallelizing a SAR software correlator and an inteferogram generation procedure will be the main topics of sections 2 and 3. Section 4 is a discussion of small-scale parallel implementations of several seemingly serial codes. Finally, section 5 will present the overall conclusions of this work.

1.1 Synthetic Aperture Radar (SAR)

Synthetic aperture radar is an active radar imaging system that provides high resolution images of large areas. The intensities of pixels in a SAR image are based on the spatial orientation, roughness, and dielectric constant of the surface imaged. The utility of SAR derives from its all-weather and day-night capability and its sensitivity to changes in the Earth's surface characteristics. This utility even extends to penetration of clear ice, dry snow, and dry sand. As such, SAR can be considered an invaluable tool for remote sensing, especially in remote regions of the Arctic where darkness and cloud cover limit the application of optical sensors.

SAR is an active sensor, transmitting its own energy, and then measuring the return scattered by the earth's surface back to the satellite's antenna. This is in contrast to optical sensors, which measure reflected energy transmitted from an external source, such as the sun. Active radar imaging, such as that used by SAR, operates by alternating between sending high power output chirps and receiving low-power echoes. The returning energy is measured in both magnitude and phase, which are functions of the amount of area scattering and the distance to the target. This information can then be processed from the frequency domain into a spatial image through a series of Fourier transforms.

The software in this paper operates on European Remote-Sensing Satellite (ERS) data. This satellite borne SAR system images a continuous 100-km wide swath at a 30-meter spatial resolution.

1.1.1 How SAR works

The side-looking nature of SAR provides the scan in the range, or cross-track, direction. Each output chirp is time coded. Resolution in the cross-track direction is achieved by measuring the differences in the round trip times for the beams, and by signal processing of the phase-encoded pulse. Returns in the far range will require a longer round trip time, while areas in the near range will return their signals in less time. This variation determines which backscatter values come from a particular area in the cross-track direction.

The forward motion of the SAR platform provides the scan in the azimuth, or along-track, direction. Resolution is gained by measurement of the Doppler effect on the backscatter beam. The Doppler effect is caused by the fact that two points at slightly different angles with respect to the track of a moving observer have different speeds at any instant relative to that observer. This produces a measurable shift in the frequency of waves returning to the platform, distinguishing returns from different areas in the along-track direction. Using these measurements, resolution independent of range is theoretically possible. All the more strange is the fact that the resolution for SAR imaging is directly proportional to the area of the antenna. Thus, the smaller the antenna area, the better the attainable resolution.

1.1.2 SAR Processing

SAR processing is the transformation of raw SAR signal data into a spatial image. In its most abstract form, this is the simple process of performing a frequency domain correlation of the received signal with a 2-D system transfer function. In practice, this process is performed in several

1-D steps, including range compression, range migration, and azimuth compression.

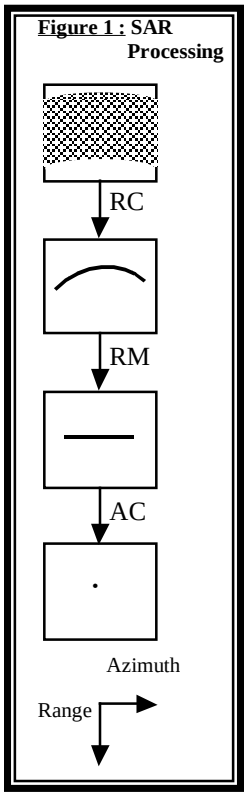


Figure 1 : SAR Processing

Range compression (RC) is matched filter pulse compression accomplished via frequency domain correlation in the range direction. The effect is to combine many range lines into a single range compressed curve.

Range migration (RM) shifts pixels in the range direction to compensate for the curvature of the earth and the rotation of the earth during imaging. The effect is to straighten out the curves into lines.

Finally, azimuth compression (AC) is matched filter pulse compression accomplished via frequency domain correlation in the azimuth direction. The effect is to combine many azimuth points into a single fully compressed point.

These three processing steps are each displayed graphically in figure 1.

For a complete description of SAR processing please refer to [Curlander] or [Franceschetti].

1.1.3 SAR Interferometry

Interferometry is a technique that capitalizes on the coherent phase inherent in SAR imaging. Given a pair of SAR images covering the same area, any phase differences between the two correspond to round-trip path length differences. Given precise knowledge of the distance between the two platforms during imaging, these path length differences can be directly related to either surface height or surface motion [Gens]. Thus, interferometry allows calculation of surface topography, surface deformation, and even surface velocity of slow-moving objects (e.g. a glacier [Fatland]).

1.2 Hardware System

ARSC operates a 272-processor 450 MHz CRAY T3E-900 system with 68 Gbyte of memory and 475 Gbyte of disk storage. All runs were performed on this system. All timings were gathered using the rtclock() function.

2 Parallel ASF SAR Processor (PASP)

SAR processing is both I/O and computationally expensive. Processing a single ERS scene requires some 300 megabytes of input, 1 gigabyte of output, and roughly 400 giga-operations (100 giga-flop) to perform. As such, it is a good candidate for high performance computing.

2.1 Serial Implementation

The starting point for this research was a range-Doppler processor originally contributed to the Alaska SAR Facility (ASF) by Howard Zebker of Stanford University [Zebker]. The code, named AISP, was re-written in ANSI C from its original FORTRAN prior to the port to ARSC.

The data are processed in patches consisting of 4096 azimuth lines by the full width of range samples (5616). Processing continues until all input patches are completed.

Each output pixel combines returns from an area equal to the length of the range reference function times the length of the azimuth reference function. Typical reference function sizes are 800 in azimuth and 700 in range. This means that for each 4096 x 5616 patch of raw data, approximately 3300 lines of 4900 samples of valid output are created.

This algorithm can be divided into six steps, 2-6 of which form the processing loop (see algorithm 1).

Algorithm 1: Range-Doppler SAR Correlator

1) ***Set-up*** *includes reading metadata information, determination of the image's Doppler centroid, and calculation of the range reference function.*

For Each Patch of Raw Data

2) ***Range Compression*** *includes reading the raw signal data, performing a forward FFT, multiplication of raw data with the range reference function, inverse FFTing the result, and transposing the data.*
3) ***Transform*** *is a forward FFT of the newly transposed data*
4) ***Range Migration*** *migrates samples in the range direction using sinc function resampling.*
5) ***Azimuth Compression*** *requires (for each line) calculation of an azimuth reference function, FFTing the function, multiplication of the function with the range migrated data, and inverse FFTing the result.*
6) ***Output*** *is the final step in which the data is transposed back into its original format and written to an output file.*

This original version of the code does all I/O in a line-by-line fashion and uses a portable FFT implementation obtained from the INFO-MAC hyper archive originally authored by John Green.

2.2 Parallel Implementation

The parallel implementation, hereafter referred to as PASP, divides the problem in both the range and azimuth directions. In the range direction, the data is split into equal regions. In the azimuth direction, patch boundaries are used to distribute the work. The result is that a single group of PEs process a patch, while multiple groups can be used to process several patches at once.

Processor groups were implemented in a strict master-slave form. The master performs all of the input, raw data distribution, processed data collection, and writes the output files. The slaves receive the raw signal data, process the data, and return the results to the master.

2.2.1 Data Distribution

Division of the data in the range direction required some analysis. Because of efficiency issues, FFTs should be run at exact powers of 2. In this case, the smallest power of 2 greater than the raw data length is used. For the range direction, this length was originally 8192. It was decided to reduce this value to 2048. However, this decision places a minimum on the number of processors per group: given an FFT of length 2048, with a range reference function of length 700, the maximum valid pixels per processor will be 1348. If more than 4044 samples are desired, we must have 4 slave processors per group and thus a minimum group size of 5.

A standard-length ASF computer-compatible signal data (ccsd) file contains approximately 26,000 lines. These allow for 8 patches per file, and thus up to 8 groups per processing run. The code was tested using groups of 5-7 processors and with 1, 2, 4, and 8 groups, for an overall range of 5 to 56 nodes.

Since the signal data is distributed in large enough chunks to compensate for the reference function overhead, no shared FFTs are required and there are no global transposes. The resulting implementation can therefore be considered embarrassingly parallel.

2.2.2 I/O Design

Each group operates independently of the others. Thus, each group master performs its own read of the single input signal data file and associated metadata file. Similarly, each group performs independent output, in this case to separate files.

During input, the first set of lines is pre-read into a buffer of configurable size. The lines are then broadcast to the group as a single message. Worker PEs are responsible for unpacking the appropriate section of raw data from each group of lines received. Once the first set of lines is broadcast, the master immediately reads the next set while the workers process the data they've received. During optimization, many different buffer sizes were tried until the value of 64 lines (718,848 byte messages) proved to minimize total input time.

During output, the group master assembles each data line into another configurable size buffer. Each portion of a line is received in order from the appropriate worker, and placed directly into the correct position in the output buffer. When the buffer gets full, it is written to the group's output file. Again, during optimization, many different buffer sizes were tried until the value of 1500 lines proved to minimize the total output time. Since the workers are sending portions of a single line, the message sizes vary from only 6K to 10K in length, depending on the group size.

2.3 Other Optimizations

Two other significant optimizations were performed on the code as well – using CRAY libsci routines and compiler options. Switching to using the CRAY science library FFT routine, GGFFT, roughly halved execution time of the various Fourier transforms required during processing. Exercising the CRAY compiler options resulted in a nearly 10% decrease in execution time. The options "–O3 –h aggress, unroll, split" gave the best results in this case.

2.4 Results

PASP shows a speed-up from 5 to 40 on from 5 to 56 PEs, decreasing the run time from 2101 seconds to just 52 (figure 2). Obvious decreases in run time when going from 1 to 2 and from 2 to 4 groups are apparent (figure 3). Less obvious is the decrease when going from 4 to 8 groups (from 28 to 40 processors).

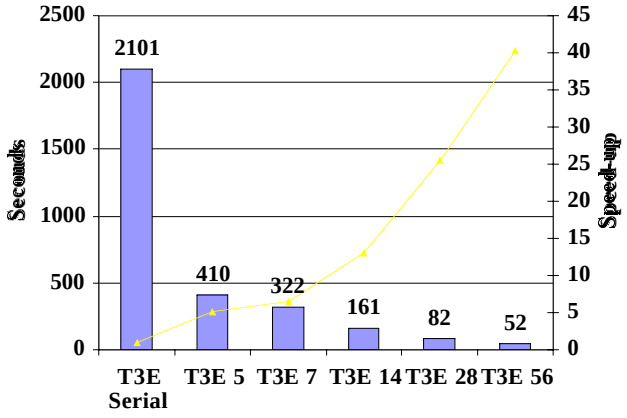


Figure 2: Run Time and Speed-up for PASP

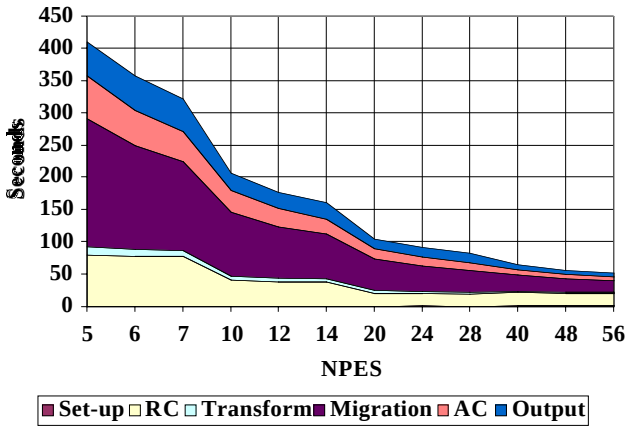


Figure 3: PASP Run Times by Task

This decrease in performance can be directly related to the scalability of the individual tasks in the algorithm. Analysis shows that the FFTs, the range migration, and the azimuth compression are completely scalable. Only the routines that contain I/O, the range compression and the output, do not scale. It was found that output to multiple files, as incorporated in this implementation, actually scaled quite well, while input from a single file failed to provide any more than a 4 times speed-up even when 8 groups were employed (see figure 4).

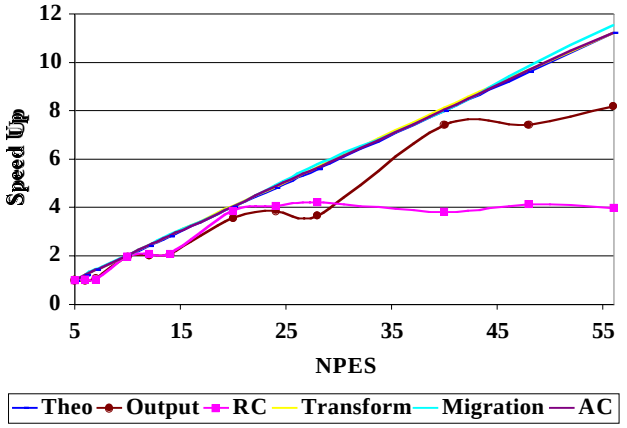


Figure 4: PASP Scalability by Task

3 Interferogram Generation

On the surface, interferogram generation seems a rather simple procedure. For each pixel (x,y) in the interferogram image:

Interferogram phase(x,y) = phase1 – phase2
Interferogram amplitude(x,y) = sqrt (pwr1*pwr2)

Where (pwr1,phase1) and (pwr2,phase2) are the power and phase at point (x,y) of image 1 and image 2, respectively.

The difficulty in interferogram generation is that point (x,y) in image 1 must cover exactly the same ground area as point (x,y) in image 2. In the case of SAR inteferometry, images must be aligned to within 1/16th or less of a pixel to achieve maximum scene coherence. This process of aligning two images is referred to as co-registration and this is the central problem solved by the register_ccsd script discussed in this section of the paper.

3.1 Serial Implementation

The register_ccsd script starts with an input pair of ERS ccsd files and creates the pair's interferogram amplitude and phase as output.

This script processes each image to the average Doppler centroid of the pair. In this way, no geometric differences due to Doppler squint are introduced by the processing.

Algorithm 2: Register_ccsd script

1. *Compute the average Doppler for the pair (estavedop).*
2. *Process the master image using the average Doppler (AISP).*
3. *Model sub-pixel offsets for the first patch in both the range and azimuth directions.*
   - *3.1. Process the first patch of each image (AISP).*
   - *3.2. Determine sub-pixel offsets*
     - *3.2.1. Obtain initial patch offset estimate using image metadata information (resolve).*
     - *3.2.2. Refine the initial estimate to pixel accuracy using FFT amplitude correlation of the two patches (resolve).*
     - *3.2.3. Using the pixel accurate offset, determine many sub-pixel offsets via complex image chip correlation (FICO). Verify offsets by performing both forward and reverse correlation*
   - *3.3. Using linear regression, fit chip correlation offsets into a function of range (fit_line).*
4. *Model sub-pixel offsets for the last patch in both the range and azimuth directions. (Using procedure from steps 3.1 – 3.5).*
5. *Using the two linear functions derived in steps 3 and 4, calculate the "patch deltas," which include the initial offsets and the amount the offsets change per patch (calc_deltas).*
6. *Process the slave image using the average Doppler and patch deltas (AISP).*
7. *Convert the complex SAR images into amplitude and phase representation and generate the interferogram (c2p, igram).*

To co-register one image to another, the second image, referred to as the slave, is resampled to closely line up with the first image, which is referred to as the master. However, each resampling performed on an image decreases its resolution. In order to avoid this resolution degradation,

PASP allows an image to be resampled using linear mapping coefficients during range migration. Since the data is already resampled during range migration, this processing schema ensures no further resolution is lost in co-registration of the pair. Instead, when the slave image is processed, it is automatically co-registered with the master. The process is summarized in algorithm 2.

3.2 Parallel Implementation

In order to determine the parallel implementation required, examination of each of the programs called by the register_ccsd script was performed. Run times for the serial code are summarized in Tables 1 and 2. Run times of the register_ccsd script show that AISP takes 57% of the time and FICO takes 30%, while the actual interferogram generation (using c2p, igram) requires roughly 11%. The other programs combined amount to only 2%. Because the programs fit_line and calc_deltas require only a few seconds each, they are not represented in these tables.

Algorithm Step	Program	Time (s)
1	Estavedop	40
2	AISP	2098
3.1	AISP	262
3.1	AISP	262
3.21-3.22	Resolve	50
3.23	FICO	702
3.23	FICO	702
4.1	AISP	262
4.1	AISP	262
4.2.1-4.2.2	Resolve	50
4.23	FICO	702
4.23	FICO	702
6	AISP	2098
7	c2p, igram	1040
	Total	9232

Table 1: Serial Run Times for Register_ccsd

Program	Total	%
AISP	5244	56.8
FICO	2808	30.4
c2p, igram	1040	11.3
Resolve	100	1.1
Estavedop	40	0.4

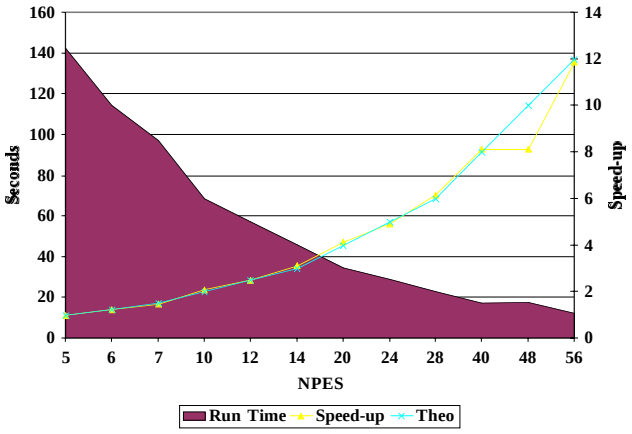
Table 2: Run Time percentage for Register_ccsd

3.2.1 Parallel FICO

Since PASP was already implemented, the next algorithm to scrutinize was FICO. FICO – or the Fast Interferometric COrrelator – operates by comparing image chips of small size (16 x 16 pixels) from the slave image with image chips of slightly larger size from the master image (32 x 32 pixels). Chips are chosen from a regular grid of configurable size (10x10 being the default). For each pair of chips, a correlation is performed that results in the offset of the center of the slave chip from the center of the master chip. Each successful correlation offset is then written to file.

Much like the SAR processor, this problem was readily distributed in an embarrassingly parallel manner. The parallel implementation of this program, PFICO, again uses a strict master-slave paradigm. In this case, a single master reads the image chips, distributes them to workers in a round-robin fashion, receives the results of the correlations, and writes the output file. The workers simply receive their designated chips, perform the correlation, and return the results. This algorithm scaled quite well, attaining nearly 100% efficiency out to 56 processors (Figure 5).

Figure 5: Parallel FICO Run Times



3.2.2 Other Optimizations

Algorithmically, only one significant change was made to register_ccsd. Since PASP creates individual patch files rather than a single large output file, the first and last patch of the master image are created in step 2 (see algorithm 2), and are available for the patch correlations. This removed the need for two calls to the processor.

Beyond PASP and FICO, none of the other register_ccsd sub-programs would lend themselves to a scalable parallel implementation. However, using implementations aimed at small-scale parallelism, further run-time reductions were realized. Please refer to Section 4 of this paper for an explanation of these programs and their results.

3.3 Results

By parallelizing 87% of the original work and employing small-scale parallelism for another 12%, an overall speed-up of from 4 to 19 was achieved when using from 5 to 56 processors on the new register_ccsd implementation. Contributions of individual programs as well as the optimizations performed are summarized in Table 3. Starting from a serial code that required more than 2 hours 30 minutes, the parallel code requires just 30 minutes on 5 processors and only 10 minutes when using 28 processors (Figure 6). It was noted that although nearly 99% of the original code was parallelized to some extent, by the time 56 PEs are employed, only 50% of the time is spent in the

scalable programs (PASP and PFICO) while the other 50% is taken up by non-scalable or serial algorithms (Figure 7).

Original	Optimization	Speed-up
Estavedop	DUO	3.6
AISP	PASP	4 to 39
Resolve	None	None
FICO	PFICO	3.6 to 56
c2p	c2igram (quad)	8.5
	Overall	**4.4 to 18.6**

Table 3: Register_ccsd Optimizations

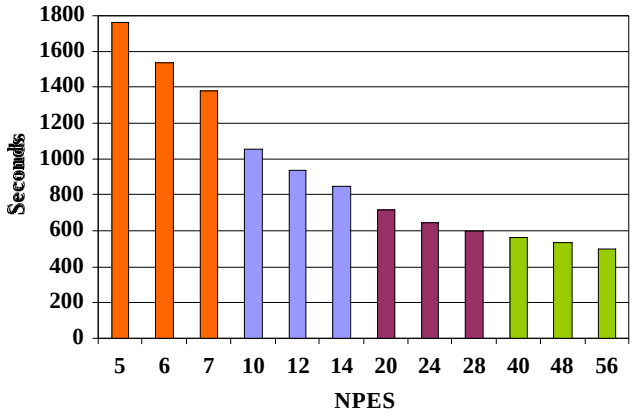


Figure 6: Parallel Register_ccsd Run Times

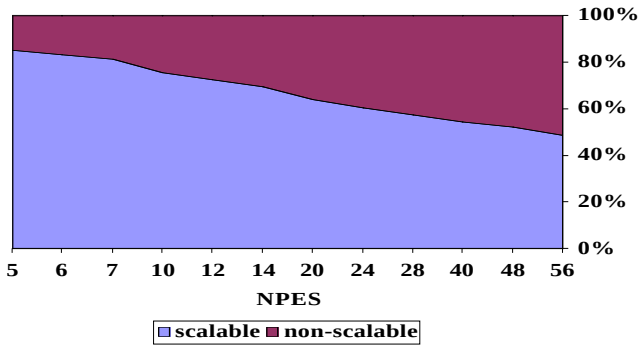


Figure 7: Time Distribution for Parallel Register_ccsd

4 DUO Programs

SAR processing includes several programs that seem to be quite serial in nature, or rather would not seem to be candidates for parallel platforms. These include algorithms with large volumes of I/O that require only small amounts of work. One example is the complex to polar coordinate conversion performed by c2p, which requires 1 Gbyte of input and 1 Gbyte of output per SAR frame, but only a square root of the sum of squares and an arctangent calculation per pixel.

In an attempt to speed-up these types of programs, small-scale parallelism, referred to here as DUO programming, was employed.

4.1 Motivation

DUO programming is an attempt to exploit whatever I/O parallelism is available on a system. The approach is to partition an algorithm into 2 phases, input and output, that can run simultaneously on separate processors. The generalized algorithm can be summarized as follows:

Original Program

```
WHILE NOT DONE
    READ INPUT DATA LINE
    PERFORM SIMPLE OPERATION
    WRITE OUTPUT DATA LINE
```

DUO Version of Program

```
WHILE NOT DONE
    IF (MY_PE == 0)
        READ INPUT DATA LINE
        SEND DATA LINE TO PE 1
    IF (MY_PE == 1)
        RECEIVE DATA LINE FROM PE 0
        PERFORM SIMPLE OPERATION
        WRITE OUTPUT DATA LINE
```

4.2 c2igram Optimizations

As an example of the effectiveness of DUO programming, a specific example will now be presented. In the original register_ccsd script, the two co-registered images are turned into an interferogram using complex to polar coordinate conversion (c2p) and interferogram generation (igram) programs. However, since PASP writes individual patch files, another piece of code that assembles the patches into a single file was required. Thus the initial 'parallel' interferogram generation section of register_ccsd was:

1) Assemble patches into full file
   Simple Unix 'cat' like command
2) Complex to polar conversion for each SAR image
   phase = atan2(imaginary,real)
   power = squared magnitude of complex value
3) Interferogram generation
   phase = phase1 – phase2
   amplitude = sqrt(power1*power2)

This code required roughly 5.6 Gbytes of input, 4.9 Gbytes of output, and a total run time of 1169 seconds.

4.2.1.1 Serial

The first step in optimizing this part of the process was to combine these three programs into a single executable. This executable, named c2igram, reads the patch complex files and writes only the interferogram. This reduced the file reads by a factor of 3, the file writes by a factor of 5, and the run time to just 500 seconds. This version of the program required about 320 seconds of file time and yet only 175 seconds of actual calculations were performed.

4.2.1.2 *DUO*

Since the file time exceeded the work time, DUO programming was employed to take advantage of I/O parallelism on the system. PE 0 is the reader, performing the reads and sending the data onto PE 1. PE 1 is the writer, catching the data, performing the calculations, and writing the output interferogram. As a latency-hiding precaution, the reader pre-caches the next block of data before it is needed. The result of the DUO program was a reduction of the run time to 256 seconds – almost half that of the serial version. Examination of the task times for the DUO version showed 196 seconds of read time, 145 seconds of work time, and 109 seconds of write time (Table 4).

	Wait	File	Work
PE 0	**38**	**196**	**0**
PE 1	**2**	**109**	**145**
Total	**40**	**305**	**145**

Table 4: DUO c2igram Task Times

4.2.1.3 *TRIO*

It is reasonable to assume that if the work time can be hidden (performed in parallel with the I/O) that the best to hope for is reduction of program execution to the longer of the read or write time – in this case roughly 200 seconds. Due to the success of the DUO version, a TRIO version was examined next. In this case, PE 0 is the reader, PE 1 is the worker, and PE 2 is the writer. The resulting code produced only moderate returns, dropping the run time to 211 seconds (Table 5).

	Wait	File	Work
PE 0	**0**	**192**	**0**
PE 1	**63**	**0**	**148**
PE 2	**94**	**114**	**3**
Total	**157**	**306**	**151**

Table 5: TRIO c2igram Task Times

4.2.1.4 *QUAD*

It was noticed that the read time had become the limiting factor in the TRIO implementation of c2igram. In an attempt to further exploit parallel I/O, a 4 processor, or QUAD, version of the program was next implemented. This time, PE 0 reads the patches of the first SAR image, PE 1 reads the patches of the second SAR image, PE 2 is the worker receiving data from both readers and sending results on to PE 3, which is the writer. This idea worked quite well, nearly halving the read time, and reducing the overall run time to just 138 seconds (Table 6). This version nearly eliminated total wait time, reducing it from 157 seconds for the TRIO version to a mere 21 seconds. At this point, the input, work, and output phases of the processing require nearly equal time, and all three are being performed simultaneously.

	Wait	File	Work
PE 0	**0**	**114**	**0**
PE 1	**0**	**124**	**0**
PE 2	**8**	**0**	**126**
PE 3	**13**	**122**	**3**
Total	**21**	**360**	**129**

Table 6: QUAD c2igram Task Times

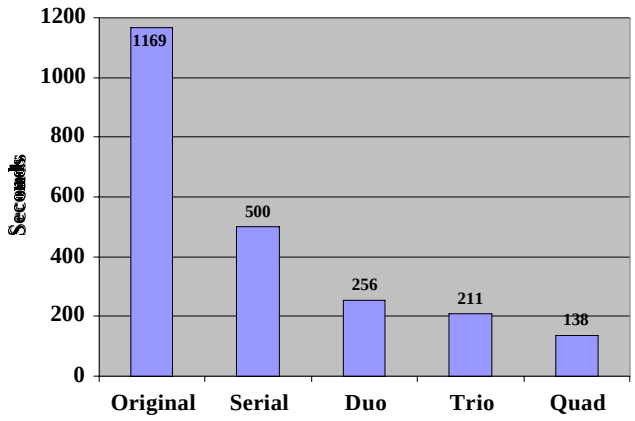
4.3 Results

DUO programs proved to be fairly useful for reducing the run-times of apparently serial codes by exploiting the I/O parallelism of the T3E. Of four different DUO programs implemented, speed-ups of from 1.6 to 3.6 were realized (table 7).

	Serial	DUO	Speed-up
c2p	263 sec	100 sec	2.6
COH	351 sec	223 sec	1.6
ML	249 sec	129 sec	1.9
Estavedop	40 sec	11 sec	3.6

Table 7: DUO Program Results

Figure 8: c2igram Optimization Stages



This idea was also extended to TRIO and QUAD versions of the program c2igram, showing a speed-up of 8.5 over the initial procedure and 3.6 over the first combined serial version (figure 8).

5 Conclusions

It has been said that supercomputing is the reduction of a CPU bound program into an I/O bound program. In general, this is done by optimizing the calculation kernel to a minimum time, and then employing latency hiding techniques to "hide" I/O and distribution in the work. If work time is greater than work distribution time, and both can be performed at once, then the overhead is hidden and the program is reduced as far as possible. This strategy was employed successfully with the program PASP, showing at least 70% efficiency for up to 56 nodes (figure 9). It was only the I/O that dropped efficiency this low – the other tasks scaled almost perfectly (even super-linearly in some cases – see figure 10). For this application, 96% or more

efficiency was obtained for up to 4 readers from a single file when using a constant group size, while 8 readers showed only 48% efficiency. In contrast, for a constant group size, nearly 90% efficiency can be maintained for up to 8 writers when writing to individual files. The conclusion here is that on the T3E, parallel input from a single file can be performed on up to 4 nodes with little side-effect, while parallel output can be performed on up to 8 nodes when writing to separate files.

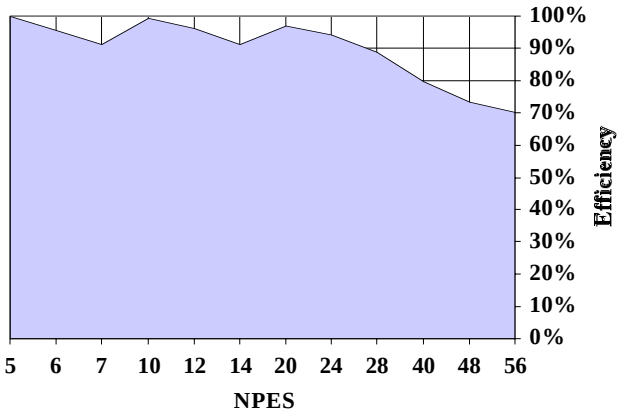


Figure 9: PASP Efficiency

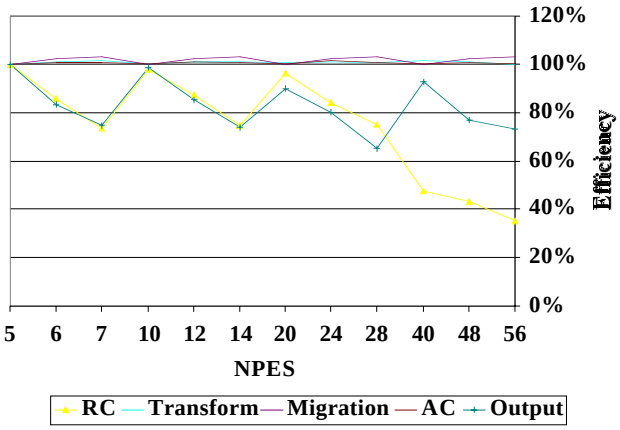


Figure 10: PASP Efficiency by Task

Not so obvious candidates for high performance computing are the I/O bound programs that inevitably accompany SAR processing packages. It is clear that whenever the kernel processing time is less than the I/O time, efficiency will be lost when trying to employ parallelism. Indeed, computation must dominate I/O for scalable parallelism. Also obvious is that a program can not execute faster than the minimum I/O time required. Thus, to reduce I/O bound programs one needs to either decrease I/O time or hide I/O time in processing time – in essence to reduce an I/O bound program to being CPU bound! To do this, small-scale parallelism can be employed using DUO, TRIO, or QUAD programs to partition work into input and output phases. In reality, I/O is not unlimitedly parallel, however a small degree of parallelism does exist and can be taken advantage of. For the examples given in this paper, speed-ups from 1.6 to 3.6 have been realized with this technique while using only 2 to 4 processors.

Acknowledgements

The author would like to acknowledge Howard Zebker and Rob Fatland for contribution of the serial algorithms used herein, Rick Guritz for stead-fast leadership during the implementation phase, Rudiger Gens for support during the writing of this paper, and Guy Robinson for his cheerful assistance throughout the entire process.

About the Author

Tom Logan is an MPP Specialist at the Arctic Region Supercomputing Center. Prior to employment at ARSC, he was employed for 6 years as a software engineer with the Alaska SAR Facility where he gained his expertise in SAR processing. Tom can be reached at

Arctic Region Supercomputing Center
University of Alaska Fairbanks
PO Box 756020
Fairbanks, AK 99775-6020 USA
E-mail: logan@arsc.edu

References

Curlander, J.C. and McDonough, R.N., (1991), Synthetic Aperture Radar: Systems and Signal Processing., John Wiley and Sons, New York.

Fatland, Dennis R. and Lingle, C.S., (1994), The surface Velocity Field on Bagley Icefield, Alaska, *Fall AGU*, December 1994.

Franceschetti, G. and Lanari, R., (1999), Synthetic Aperture Radar Processing, CRC Press, New York.

Gens, R. and Genderen, J.L. van, (1996), SAR interferometry – issues, techniques, applications. *International Journal of Remote Sensing*, vol. 17, pp. 1803-1835.

Zebker, H.A., C.L. Werner, P.A. Rosen, and Hensley, S., (1994), Accuracy of Topographic Maps Derived from ERS-1 Interferometric Radar, *IEEE Transactions on Geoscience and Remote Sensing*, Vol. 32, pp 823-836.