Parallelisation of a Satellite Signal Processing Code - Strategies and Tools

Ian Glendinning

VCPC
European Centre for Parallel Computing at Vienna
Liechtensteinstraße 22, A-1090 Vienna, Austria
ian@vcpc.univie.ac.at

Abstract. This paper presents strategies and tools that have been used in work to parallelise a satellite signal processing code. The Magellan mission to map the surface of Venus using synthetic aperture radar (SAR) is briefly described, the Magellan SAR processor code is outlined, and a parallelisation strategy is presented. The code's large size and limited documentation made the use of program analysis tools essential to implement this strategy. Three tools, FORESYS, IDA and FORGExplorer, are compared, and the use of FORGExplorer to perform code analysis is described in detail. The techniques presented are of general applicability to the parallelisation of codes in other application areas.

1 Introduction

VCPC (European Centre for Parallel Computing at Vienna) is collaborating with the ICG (Institute for Computer Graphics) in Graz, Austria, and JPL (Jet Propulsion Laboratory) in Pasadena, California, USA, to parallelize a program which performs image analysis on the data collected by the Magellan spacecraft on its mission to Venus. Due to the code's complexity, and the limited amount of documentation available for it, the use of program analysis tools has proved to be essential, and this paper reports on tools and techniques that have been employed, which are also relevant to parallelisation in other application areas.

2 The Magellan Mission

Magellan was carried into Earth orbit in May 1989 by space shuttle Atlantis. Released from the shuttle's cargo bay, it was propelled by a booster engine toward Venus, where it arrived in August 1990. Magellan used a sophisticated imaging radar to pierce the cloud cover enshrouding the planet Venus and map its surface. During its 243-day primary mission, referred to as Cycle 1, the spacecraft mapped over 80 percent of the planet with its high-resolution Synthetic Aperture Radar (SAR). By the time it completed its third 243-day period mapping the planet in September 1992, Magellan had captured detailed maps of 98 percent of the planet's surface.

Magellan orbited Venus approximately once every 3.5 hours, passing over the poles, and mapped the surface of the planet in thin north-south strips about 25 km wide and 16000 km long [2], with range and azimuth resolution of 88 and 120 metres respectively [3]. These strips were nicknamed noodles, and some 4000 of them were mapped during the whole mission, 1800 per cycle. The radar system operated in burst mode, sending out trains of pulses and listening for echoes in between the pulses, and each noodle contained data from around 5000 bursts. The radar gathered data while looking perpendicular to the direction of motion (Fig. 1), and measured the strength of reflected signals, as well as how long each signal took to make the round trip, and changes in the signal frequency. This enabled the Magellan SAR processor, running on computers back on earth, to calculate range and azimuth coordinates from the raw data, and produce high-resolution two-dimensional images of the planet's surface.

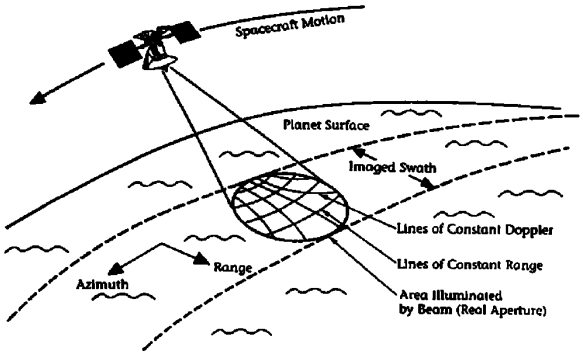

Fig. 1. Magellan SAR geometry

3 The Magellan SAR Processor

The Magellan SAR Processor is a program that implements the digital signal processing operations needed to convert the raw signal data to image form representing normalised backscatter return from the surface of Venus. The program operates on burst data and produces single-look image framelets, which are then superimposed to form multi-look image data orbit strips. Each burst is processed using an FFT range correlation for range compression, followed by a corner turn and azimuth compression using a "deramp-FFT" algorithm [1]. Side-lobe control, range-walk and phase shift compensation is also done during range compression. Azimuth processing includes side-lobe control and interpolation in azimuth spacings. Finally, geometric, radiometric and multi-look processing are performed to

produce image framelets in a (oblique) sinusoidal projection. A utility program called `mosaic` can be used to assemble the framelets into an image of a noodle for viewing.

4 Parallelisation Strategy

From the outset it was decided to perform the parallelisation using the message-passing style of programming, using the portable MPI message passing interface [6]. The 128 node QSW CS-2 machine at VCPC is being used as a development platform, together with the Argonne National Lab / Mississippi State University implementation of MPI for the CS-2.

It was quickly realised that radar bursts represent a natural unit of parallelism in the program, since each one can be analysed independently, except for the final stage, called 'look buildup', which merges results from overlapping observations in neighbouring bursts. However, the overlap is strictly local, so only a few neighbours will have to communicate relatively small amounts of data with each other, after the bulk of the processing for each burst has been done, and so an efficient parallelization of the code should be possible

Although the SAR processor is a fairly large program, the bulk of the source code is involved with a preprocessing phase, which takes relatively little time to execute compared to the burst processing for a whole noodle. For the sample dataset tested on a single CS-2 node at VCPC, the initialisation phase takes about 59 seconds, compared with about 2.5 seconds per burst, but there are over 4000 bursts in the dataset, so the burst processing time totally dominates. Thus, the focus of attention has been directed at the main burst processing loop, which is located in the routine that performs the SAR correlator processing, `process_corr`. The main work of the loop is done by calls to the routines shown in Fig. 2, whose parameters are omitted for brevity.

```
call corr_pp     ! Fill processing parameter (pp) common block
call pp_keyvar   ! Copy key pp values to correlator common block
r_nbytes = cread ! Read raw burst data
call decode_sar  ! Decompress raw intensity values
call range_comp  ! Perform range compression
call corner_turn ! Perform a non-symmetric matrix transpose
call az_comp     ! Perform the Azimuth compression
call radio_comp  ! Perform the radiometric compensation
call geo_rect    ! Perform a geometric rectification
call multi_look  ! Generate multi-look image
```

Fig. 2. Routines in the main burst processing loop in `process_corr`

The final routine `multi_look` is the only one that does not operate independently on each burst, and is the one that writes the image to disk. The basic

strategy is to re-code this loop so that each iteration can be executed as a task on a separate processor. In practice, a pool of processors, each with code to execute a single burst, would have burst data distributed to them by a master process. The resulting image data could be collected by the same master process, or by another process, but for now let's assume that there is a single master controller. In order to be able to re-code the loop like this, it is necessary to understand its data dependences. In particular it is necessary to identify:

1. Variables which are set before the loop, and read inside it, as this implies communication of initial values from the master to worker processes.
2. Variables which are set inside the loop and read after it, which implies communication of values from the workers back to the master.
3. I/O, which must be properly sequenced. File handles need special treatment, as you can't write to a handle opened on another processor.
4. Variables whose values are read within the loop, before later being updated, as their values depend on an assignment from the previous iteration, and imply communication between worker processes. This represents an anti-dependence from the first statement to the second.

Not only variables which are local to the subroutine containing the loop must be considered, but also any variables in common blocks that are used either by the routine itself, or any other routines that it calls, directly or indirectly.

Other work [5] has used a finer-grained approach, parallelising the range and azimuth compression code, but for the Magellan SAR processor that would give a limited maximum possible speedup, due to the residual sequential code. For example, it is estimated that if the `range_comp`, `corner_turn` and `az_comp` routines were parallelised with 100% efficiency, a maximum speedup of only 3.0 could be obtained.

The source code for the SAR processor consists of approximately 125 thousand lines of Fortran, divided among approximately 450 subroutines, together with approximately 6500 lines of C, and although focusing on the main burst processing loop means that only around 5500 lines (plus included common block declarations) need to be analysed in depth, this is still a substantial body of code, so manual analysis would be extremely laborious, and support from parallelisation tools is highly desirable.

5 Evaluation of Parallelisation Tools

5.1 FORESYS

FORESYS is a commercial tool marketed by SIMULOG. It is a 'Fortran engineering system', and has a range of features that help improve scientific software. Initially version 1.4.1 was used, and later version 1.5 when it became available. The features of the tool that were evaluated were code browsing, program analysis, and program restructuring.

Code Browsing FORESYS has a graphical user interface which simplifies code browsing, for example by being able to open a new window displaying the source code for a routine, simply by highlighting its name in another window and selecting a menu option. Although conceptually simple, this was found to be quite useful in practice.

Program Analysis The feature of FORESYS that seemed potentially most useful for this work was its program analysis component, called PARTITA, which was able to graphically display dependences (Fig. 3). However, in practice the display became cluttered with information about types of dependence which weren't relevant to the coarse-grained task parallelism that had been selected for the Magellan code. There was also no obvious way to identify dependences between variable references inside the main loop and those outside it.

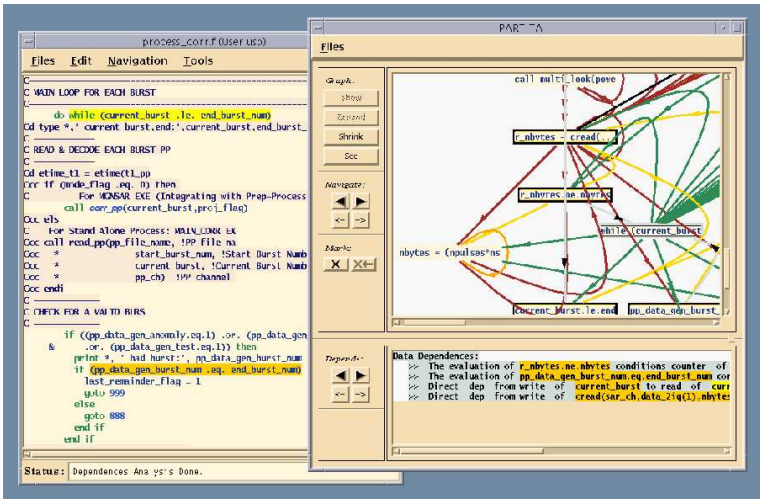

Fig. 3. Program analysis with FORESYS

Program Restructuring The tool was found to have an impressively robust front end, which was able to handle the non-standard Fortran extensions used in this large code, almost without complaint. It was also able to transform the program into equivalent code in standard Fortran 77 (it can also produce Fortran 90, but that was not relevant in this case). This rather straightforward sounding feature turned out to be its most useful one in the context of this work, as neither IDA nor FORGExplorer could have been used without it. Neither of them had a front end robust enough to handle the non-standard Fortran code of the Magellan SAR processor, which had to be cleaned up first using FORESYS.

5.2 IDA

IDA (Inter-procedural Dependency Analyser), a public domain tool available from the University of Southampton and VCPC [4]. It provides interprocedural information about Fortran programs, such as:

1. Call graphs: the calling relationships between program units
2. Traces of variables: where and how a variable is used throughout the program
3. Common block partitioning and usage: how common blocks are partitioned into variables in each program unit, and how those variables are used in that unit and its descendents
4. Procedure references and argument associations: the location and actual arguments of every call to a particular procedure

IDA was designed for speed and simplicity of operation rather than sophistication. It provides a text command interface, not a graphical one, and it only performs code analysis, not code transformation. However, it is simple to use and its analysis is fast, and its variable trace feature proved to be a useful starting point for investigating some of the dependences within the Magellan SAR code, which were later investigated more fully using FORGExplorer. IDA actually provides much of the functionality offered by FORGExplorer, but the latter tool's more sophisticated user interface made it much easier to use in practice.

5.3 FORGExplorer

FORGExplorer is a commercial tool marketed by Applied Parallel Research. It has a Motif GUI, and presents a global, interprocedural view of a program. It can perform searches and variable traces, and features an interactive, interprocedural Distributed Memory Parallelizer (DMP). Initially version 2.1 of the tool was used, and later version 2.2 when it became available. The front end of version 2.1 did not accept all of the non-standard Fortran features in the code, and so FORESYS was used to produce a cleaned up version of the code which FORGExplorer could process. The features of the tool that were evaluated were code browsing, the DMP option, and the global analysis views.

Code browsing Similar facilities to those of FORESYS are provided for code browsing, and they are similarly useful, though there are slightly more options. The Call Subchain display, which represents the call graph as a list of routine names, indented according to their call level, was found to be particularly useful.

Distributed Memory Parallelizer The idea of an automatic parallelizer was obviously attractive, and so some time was invested in 'profiling' the routines that were unknown to the tool, that is the ones for which it does not have source code, such as the C routines in the Magellan code. The types of their arguments must be specified, and whether they are read from or written to. Unfortunately,

having profiled the unknown routines, it was discovered that the program was too complex for FORGExplorer to handle, and it looped indefinitely. It may have been possible to get further with the DMP using semi-automatic parallelisation options, but it was decided to be less ambitious, and to investigate the manual inter-procedural analysis features of the basic FORGExplorer tool instead.

Global Analysis Views Manual analysis of the code using FORGExplorer's global analysis views proved to be a much more fruitful approach than the DMP option, and in particular the variable tracing and common block usage displays were found to be extremely helpful for identifying dependences. The variable tracing display (Fig. 4) lists all program lines where a variable is referenced, following the path that items passed through common or subprogram arguments take, and the common block usage display shows a grid of common blocks versus subprograms. At the intersecting cells, the usage of the common block by each routine is summarised, according to whether variables are set and/or read.

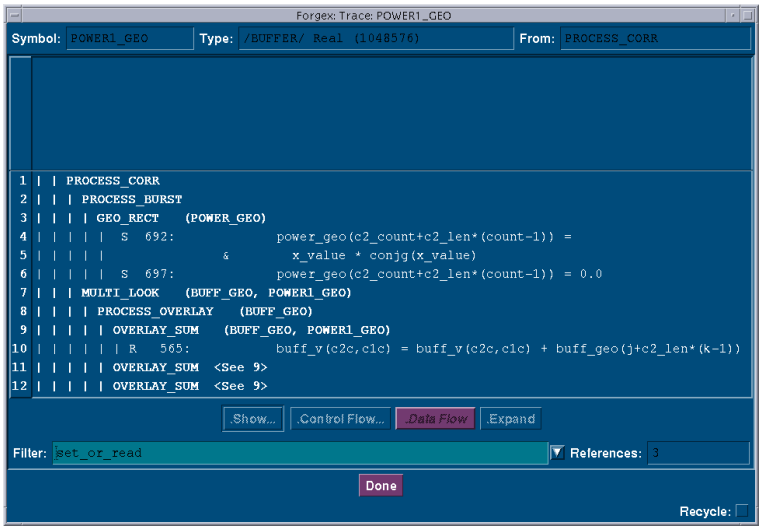

Fig. 4. The FORGExplorer Trace window

6 Program Analysis and Parallelisation

Having evaluated the tools, FORGExplorer was the clear winner in terms of its program analysis features, and so it was applied to the analysis of the Magellan SAR code, to implement the parallelisation strategy.

6.1 Isolating the Slave Process Code

The first step was to identify precisely which code from the burst loop (Fig. 2) would be run as a slave process, and to localise it in a separate subroutine, giving it a clean interface to the rest of the code, exclusively through common blocks. It was decided to keep the `cread()` statement in the master process, together with the two routines called before it, as that represented a particularly clean break in the loop. The remaining routines in the loop were moved into a new subroutine, called `process_burst`. The new routine inherited all of the common block and local data declarations from `process_corr`, and in addition, all of the local variables were declared to be in a new common block, so that values set in one routine could be read by the other.

6.2 Using FORGExplorer's 'Reference Node'

The next step was to restrict the area of interest to the `process_burst` routine, and routines called by it, by setting FORGExplorer's 'Reference Node' to it. The 'Common Blocks' display shows that the whole program uses 415 routines, 91 of which reference data in common, spread over a total of 61 common blocks. Setting the reference node to `process_burst` reduced the area of interest to 15 routines, referencing 14 common blocks. Potentially that means $15 \times 14 = 210$ common block references, but FORGExplorer showed that in fact there were only 40, and so had succeeded in reducing the size of the problem considerably.

6.3 A Simplified Parallelisation Strategy

Although the complexity of the program analysis had now been much reduced, the dependencies within the `multi_look` code were still fairly complex, and in order to obtain an initial parallel version of the code as quickly as possible, it was decided to first construct a simpler parallel version of the code than originally planned, by moving the call to `multi_look()` out of `process_burst`, into its calling routine `process_corr`. In this scheme, the maximum amount of parallelisation is limited by the sequential code remaining in `multi_look`, but the analysis is significantly simplified, as just 8 routines need to be considered in `process_burst`, and there are a total of just 14 references to 8 common blocks.

6.4 Dependence Analysis

The dependences between the master and slave processes were then analysed, so that the communications necessary between them in the parallelised code could be determined. This was done using the common block displays for both `process_burst` and `process_corr`, together with the variable tracing display.

The 'Common Blocks' display allows a display of each individual common block to be opened, which shows a table of variables in the common block versus the routines that access them, and for each variable reference it it indicated

whether it is read and/or set. Variables in the common blocks that were referenced within `process_burst` fell into three categories, according to whether they contained variables that were:

1. Read but not set within `process_burst`
2. Set within `process_burst` and read afterwards in `process_corr`
3. Set and then read within `process_burst`, but not used in `process_corr`

Those in the first category were the easiest to identify, since it was enough that all their references were reads, but for those in the other two categories, it was necessary to check the read and write usage of the individual variables in the common blocks, since different variables set in a common block may or may not be read later. This was done using the variable tracing display, which also revealed accesses to common block variables that were passed to routines as arguments, although the common block was not declared in the routine, in which case the references were not indicated in the common blocks display. The variable tracing feature was also used to check for anti-dependences within the burst loop (a variable read followed by a set), but none were found. Fig. 5 summarises the results of the dependence analysis, where the numbers in the grid cells correspond to the categories of reading and writing defined above.

	ant_weight	az_comp	cfft	geo_rect	init_fft	process_burst	radio_comp	range_comp
/buffer/						2,3		
/fft_aux/			3		3			
/key_var/	1	1		1		1	1,3	1
/overlay_c12/				2,3				
/pc_loc/						1		
/po_a/				1				
/s_weight/				1				
/test_burst_no/							1	

Fig. 5. Results of dependence analysis - Common blocks vs. routines

The communication of variables between master and slave processes can now easily be deduced. Variables in category 1 must be sent to the slave when it starts to execute, those in the category 2 must be sent back to the master when the slave has finished its processing, and those in category 3 require no action.

6.5 Parallelisation

Although the slave code was isolated from the rest at an early stage in the work, the whole program was kept sequential for as long as possible, so that it could be compiled and tested after each modification, to make sure that it still behaved as before, and so that the global analysis features of FORGExplorer could continue to be used on the whole code. Eventually, having performed the dependence analysis, the code was split into two executables, one for the slave process, containing the code for the `process_burst` routine, and one for the

master process containing the rest of the code. Further analysis showed that the large arrays in the common block /BUFFER/ could be split between the master and the slave, which reduced the size of both executables considerably, which is important as they are close to the limit of available memory on the CS-2. The insertion of the MPI communication calls is currently being performed.

7 Conclusion

The parallelisation strategy that has been adopted for the Magellan SAR processor is to re-code the burst processing loop so that part of each iteration can be executed as a task on a separate processor. Due to the complexity of the code, program analysis tools were needed to help implement this strategy. The FORESYS, IDA and FORGExplorer tools were evaluated regarding their suitability for the task, and it was found that the most effective approach was to use FORGExplorer's variable trace and common block usage facilities, in conjunction with FORESYS as a preprocessor to clean up the code. The data dependences for a slightly simplified parallelisation strategy were analysed, and work has begun to implement the corresponding parallel program. FORGExplorer is a general tool, and the techniques that have been described are also applicable to the parallelisation of codes in other application areas.

Acknowledgements

This work has been supported by the Austrian Fonds zur Förderung der Wissenschaftlichen Forschung (FWF) through FSP project S7001, "Theory and Applications of Digital Image Processing and Pattern Recognition".

The author wishes to thank Scott Hensley for discussions about the Magellan SAR code, Ivan Wolton for advice on using FORGExplorer, and Rainer Kalliany and Alois Goller for much help and advice.

References

1. Curlander, J. C., McDonough, R. N.: Synthetic Aperture Radar: Systems and Signal Processing. Wiley Interscience (1991)
2. JPL: The Magellan Venus Explorer's Guide. NASA; Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA, JPL Publication 90-24 (1990)
3. Leberl, F., Maurice, K., Thomas, J., Kober, W.: Radargrammetric Measurements from the Initial Magellan Coverage of Planet Venus. Photogrammetric Engineering & Remote Sensing, Vol. 57 No. 12 (1991) 1561-1570
4. Merlin, J. H., Reeve, J. S.: IDA - An aid to the parallelisation of Fortran codes. Technical report, Department of Electronics and Computer Science, University of Southampton (1995)
5. Miller C., Payne, D. G., Phung, T. N., Siegel, H., Williams, R.: Parallel Processing of Spaceborne Imaging Radar Data. Proceedings of Supercomputing '95, IEEE Computer Society Press, San Diego, CA (1995)
6. Message Passing Interface Forum: MPI: A message-passing interface standard. International Journal of Supercomputer Applications **8**(3-4) (1994)