A multiprocessing Framework for SAR Image Processing

Christian Andres, Torben Keil, Raik Herrmann and Rolf Scheiber
Microwaves and Radar Institute
German Aerospace Center (DLR)
Oberpfaffenhofen
Email: [christian.andres, torben.keil, raik.herrmann, rolf.scheiber]@dlr.de

***Abstract*—This paper introduces a framework developed for image processing of synthetic apterure radar (SAR) images. It encapsulates features of modern hardware architectures, including symmetric and asymmetric multiprocessing, within an easy and intuitive to use application programming interface (API). The multiprocessing part is designed for unified usage of different architectures reaching from multicore processors to cluster of workstations to grids of clusters. So an applcation using the framework can be ported from one architecture to another without any changes in the source code. The framework builds the bottom layer of the processing system developed for the German Aerospace Center's (DLR) new airborne SAR sensor, the F-SAR [1].**

I. Introduction

The German Aerospace Centers new airborne SAR sensor (F-SAR) gets more and more operative, and first data sets are already acquired. Therefore new requirements to the processing software appear, in terms of data rates and operational modes. Due to the massive amount of data, the processing time required for one data acquisition (without interferometry) has been estimated from 8 to 28 hours using one AMD Athlon64 CPU at 2.2GHz. So new strategies for data handling and processing are required. Therefore a new processor implementation is done. The processor implementation is realized object oriented using the C++ programming language which was identified to achieve maximum speed and flexibility. The processor will provide support for different kinds of hardware including symmetrical and asymmetrical multiprocessing architectures. Algorithms are developed from many different scientists, so a mechanism to handle the multiprocessing transparently is required to decrease development time of the different algorithms. This is encapsulated within a numerical framework.

A. Interface

The framework is used from many different developers from many different countries, most of them barely familiar with C++ programming. The design goal is to provide a simple interface to the numerical operations, e.g. the easies way write the summation of two arrays x and y is:

$$z = x + y, \tag{1}$$

ignoring the fact, that C++ is originally not able to add arrays. The task of the framework is to provide these operators to the application developer, encapsulating all the multiprocessing possibilities.

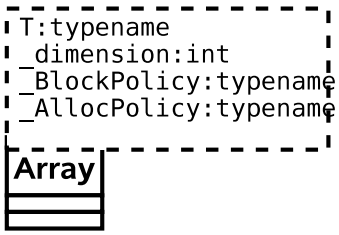

Fig. 1. Array boundary template

B. Policy based problem decomposition

First of all, the framework is designed for speed. Therefore a few rules were taken into account:

- avoid data copy as often as possible,
- avoid branches,
- keep the flexibility necessary for SAR processing.

In object oriented design, flexibility is often achieved using polymorphism. The problem with polymorphism is, that the branch it replaces, is just handled transparently by the compiler using virtual function pointers, but it is not eliminated.
A method to eliminate those branches is the policy based class configuration introduced in [2]. Here, the problem is divided into different subclasses (policies) which are passed as generic parameters to a main class template, which is derived from the different policies. The compiler generates now native classes for each possibility of the class instance. Method calls are now native calls, and flexibility is achieved by exchanging the policies, which is carried out using generic copy constructors.

II. Array abstraction

Many libraries for array abstraction are spread across the internet, but most developers prefer to create their own, because none of those libraries provides exactly the functionality needed. SAR processing requires the following features:

- iterating through array lines and columns (avoid corner turns or subscripts),
- possibility to store different data types including complex samples,
- overlapped block processing

This leads to a generic array compound template taking the datatype, the dimensions, the allocation and blocking policies as meta-parameters (see Fig. 1). The initialization is not implemented as a policy, to simplify cast operations and programming interface to the application developer. The following sections describe the significant parts of the array template, the block processing, the allocation and the iteration.

A. Iteration

SAR processors need a lot of transposed operations. Conventional processors use two approaches for acomplishing those tasks:

- turn the array: Tends to be very slow for big data blocks,
- subscripting: Requires a lot of copy operations.

Therefore a unified facility for iterating the array in multiple dimensions tends to be the more elegant way. A little drawback of such iterators is their cache coherency, when iterating higher order arrays. Also the iterator needs to support wrapping to enable operations of different array dimensions e.g. needed for range compession.

B. Allocation

The allocation policy encapsulates memory management of the array template. This is necessary to use the features of modern processors if present, and to prepare fast data exchange for small arrays using shared memory. Three policies are implemented:

- default: Simply wrapps the C++ new and delete operators,
- aligned: Provides aligned memory allocation, required to use streaming SIMD extensions (SSE) and multimedia extensions (MMX),
- shared: Allocates within the shared memory area.

C. Block Policies

To enable parallel processing, the splitting of arrays into smaller blocks is required. This can be archieved easyly through the iteration concept of the framework. For SAR Processing, different algorithms need different block processing strategies, in terms of block alignment and overlap, e.g. azimuth compression needs the full azimuth size, whereas the presumming can operate an variables patches, but needs appropriate overlap. So the block processing was designed as a policy of the array class, providing the block iterators and allowing fast and easy replacement of the strategy using generic copy constructors. These different block policies share a common interface allowing the array template to provide iterators for different block processing strategies in a unified way.

III. Operation

The execution of an operation on multiple processors is suitable for policy based decomposition, since it can be divided into the following operational tasks also shown in figure 2 assuming a simple divide and conquer strategy

- execution of the operation itself,
- splitting of the processes,

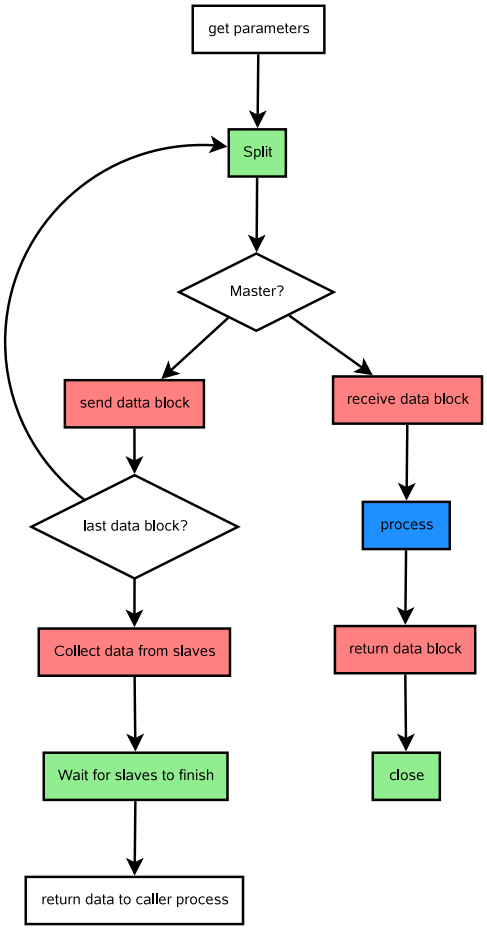

Fig. 2. Flowchart of an operation with decomposition to different components: transport (red), splitter (green) operation (blue)

- data transport to the workers.

Those policies are descibed in the following sections. The boundary template is shown in figure 3, including an input component to enable processing of arrays of different data types.

A. Splitting

Different architectures require different ways to detach the worker processes from the master process. For a multicore or multiprocessor architecture, a simple fork is sufficient, for a OpenMOSIX cluster [4], an additional process distribution command is required and PVM [5] architectures come with a specialized API for process splitting and distribution. All those environments can be fused to a common interface treated as the split policy.

B. Transport

The transport policy acts similar as the splitting policy. The task is to wrap the possibilities for data transport required in the different architectures to a common interface, which can be used within the generic operation template. Here, transport using pipes is required for multiprocessor and OpenMOSIX

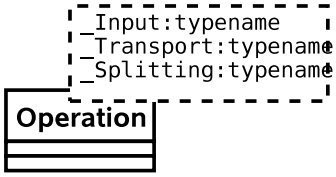

Fig. 3. Boundary operation template

TABLE I
POLICY COMBINATIONS FOR DIFFERENT ARCHITECTURES

Architecture	Split Policy	Transport Policy
Single Processor	None	Memory
Multicore / CPU	Fork	Pipe
OpenMOSIX	Fork / Distribute	Pipe
PVM	pvmspawn	pvmtransport

architecture, memory transport for single processors and PVM comes with a specialized API either.

IV. COMPLEX ENVIRONMENTS

As mentioned in the last section, different computer architectures require a dedicated combination of transport and split policies. This policies can be fused to a generic boundary template, the Environment. Table I shows the combinations for different architectures. But what happens with nested architectures, e.g. a PVM connected grid of different OpenMOSIX clusters build with multicore computers? Of course one could use the top architecture (PVM in this case) to distribute multiple workers to the different nodes. The drawback of this strategy is, that the overhead for PVM distribution is a lot higher than the overhead for Fork splitting. The solution to gain the best performance for each environment is to combine complete algorithms to higher order operations executed within a stack of environments. For example a SAR Range Doppler processor can be treated as an operation composed of different sub-operations e.g. range and azimuth compression, which contain multiple atomic operations like add or FFT. Therefore, one could spread the complete Range Doppler algorithm across the PVM architecture, parallelizing the sub-operations across the OpenMOSIX clusters and use multiple processors for the atomic operations. To achieve such a behavior multiple environments need to be connected using the GOF chain of responsiveness pattern [3]. This chain can be configured to activate the appropriate environment during the process runtime. This can be used to control the parallelization granularity of the algorithm at runtime, with the drawback of a minimal overhead for one cast operation. Of couse preconfigured operations can be used, eliminating this cast.

V. RESULTS

The framework has been tested within the architecture specified in table II. For this test system, only the coventional forking symmetrical multiprocessing, and the single processor environment is applicable. Each benchmark used 2-dimensional single precision complex arrays.

TABLE II
TEST SYSTEM CONFIGURATION

Processor	Intel Core2 Duo T7600
CPU's	2
RAM	2GB

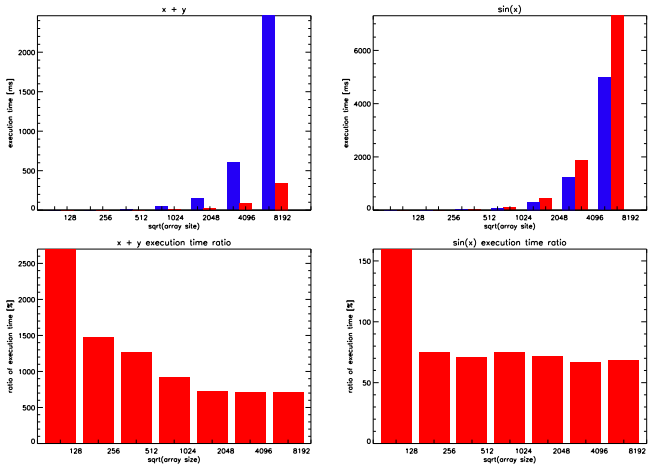

Fig. 4. Benchmarks of different operations using single (red) and smp (blue) environment [top] and ratio of the execution time $\frac{t_{smp}}{t_{single}}$. Note that the scales are different.

TABLE III
BENCHMARKS FOR ADD OPERATION

size	Single CPU	SMP with 2CPU's	Ratio
128	0.072ms	1.943ms	2698.61%
256	0.284ms	4.202ms	1479.58%
512	1.232ms	15.692ms	1273.70%
1024	5.834ms	53.965ms	925.009%
2048	20.935ms	152.861ms	730.170%
4096	83.36ms	599.335ms	718.972%
8192	344.383ms	2461.88ms	714.867%

TABLE IV
BENCHMARK FOR SINUS OPERATION

size	Single CPU	SMP with 2CPU's	Ratio
128	2.997ms	4.787ms	159.726%
256	12.009ms	8.979ms	74.7689%
512	48.276ms	34.125ms	70.6873%
1024	106.493ms	80.445ms	75.5402%
2048	434.859ms	312.461ms	71.8534%
4096	1853.01ms	1247.87ms	67.3429%
8192	7302.59ms	4992.1ms	68.3607%

A. Atomic Operations

The first investigation should demonstrate the possibility of parallelizing single (atomic) operations. Fig. 4 shows, that a simple add operation is faster using just one processor. Here, the parallelizing overhead outnumbers the execution time of the operation significantly. The second graph shows the benchmark of a sinus operation, which executes significantly faster when using two processors. The sinus operation can be treated as the 'break even point', where parallelization gets practical, at least for multiprocessor architectures.

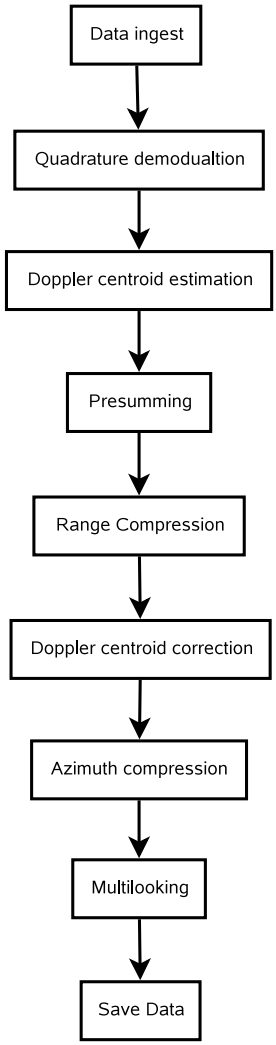

Fig. 5. Flow chart of a range doppler processor for airborne SAR including doppler centroid estimation and presumming

B. Range Doppler Processing

To test the stacked environments, a simple Range Doppler processor adapted for airborne SAR processing was used (See Fig. 5). Also real sampled input data was used so the algorithm was extended with a Hilbert transform based fast quadrature demodulation. Also a, in airborne processing typical, presumming step is performed. The processing was carried out with two different environments, a combination of single and multiprocessing environment and vice versa. This represents two different granularities, for atomic operations and for combined algorithms. Also just one CPU was used for reference purpose.

Fig. 6 shows the results of the benchmarking. It is visible, that the fine grained parallelizing does not perform as good as expected in the preceeding section. The improvement from the complex operations is nearly taken from the simple operations drawback. An operation adaptive chain, providing the optimal environment for each operation should improve the fine grained approach. The result of the coarse grained approach shows the expected speed improvement.

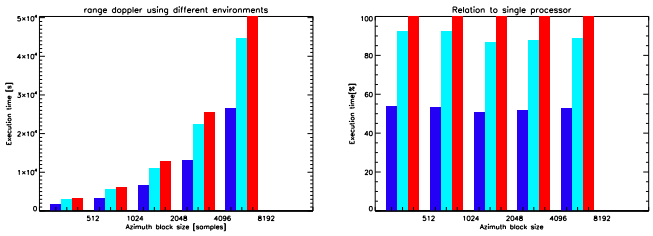

Fig. 6. Benchmark of the range doppler algorithm using 1 Processor (red), fine granular SMP (cyan) and coarse granualar SMP (blue)

TABLE V
POLICY COMBINATIONS FOR DIFFERENT ARCHITECTURES

size	Single CPU	fine grained	Coarse grained
512	3128.48ms	2883.81ms	1689.57ms
1024	6097.26ms	5628.93ms	3259.66ms
2048	12779.5ms	11088.8ms	6473.26ms
4096	25418.3ms	22250.7ms	13176.3ms
8192	50300.1ms	44554.9ms	26404.2ms

VI. CONCLUSION AND OUTLOOK

This paper showed a generic approach for providing a unified framework for symmitric and asymmetric multiprocessing adapted to purpouses of SAR processing. Although the framework is especially developed for SAR image processing, it can be used for each task requiring numerical operations on large datasets. It was demonstrated, that multiprocessing is applicable for rather “short” operations. With the stacked environment chains a step into abstract parallel programming using compiled languages has been done.
The most important improvement is the operation-adaptive environment chain, which will improve the fine-grained execution of operations. Also, the implementation of many numerical and image processing operations is required to get a fully featured image processing library.

REFERENCES

[1] *F-SAR - The new airborne SAR system* http://www.dlr.de/hr/en/desktopdefault.aspx/tabid-2326/3776_read-5691 date: 02-05-2007
[2] Andrei Alexandrescu, *Modern C++ Design*, Generic Programming and Design Patterns Applied Addison-Wesley, 2001 ISBN 0-201-70431-5
[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns*, Elements of Reusable Object-Oriented Software Addison-Wesley, 1994 ISBN 0-201-63361-2
[4] *openMosix, an Open Source Linux Cluster Project* http://openmosix.sourceforge.net/ date: 25-04-2007
[5] *PVM: Parallel Virtual Machine* http://www.csm.ornl.gov/pvm/ date: 25-04-2007