# A Parallel Programming Tool for SAR Processors*

M. Lettere, D. Guerri, and R. Fontanelli

Synapsis Srl, Livorno, Italy

**Abstract.** In the context of Italian Space Agency COSMO SkyMed project a quantitative and qualitative study of a set of image processing algorithms for *SAR Processors* has been carried out. The algorithms showed some interesting patterns in terms of structure and parallelism exploitation. During the activity of prototyping and analysis, an abstraction (**SPE Chain Model**) of the algorithmic behaviour has been defined in order to simplify performance modeling, design and implementation of parallel image processing algorithms. According to the defined abstraction, a parallel programming tool (**SPE**- Sar Parallel Executor) has been developed. SPE enables the implementation of efficient, structured and object oriented parallel image processing algorithms conforming to the **SPE Chain Model** and reuse of pre-existing sequential code. A set of image processing algorithms belonging to different classes of applications have been tested to validate both the *SPE Chain Model* and the *SPE* programming tool. The results show that no significant difficulties arise in the porting of already existing code to *SPE* and that writing new parallel algorithms is intuitive and productive and provides, at the same time, concrete high performance solutions required in real-time industry environments.

## 1 Introduction

Earth observation is based on the application of computationally challenging image processing algorithms on image data. Images are acquired at fine geometric resolutions and raw data is quite huge (26500*5600 double precision complex pixel values for a raw image [2]). This well known fact, along with requirements related to real-time industry production, led to a first study which focused on quantitative aspects (*flops*, memory usage) of a large set of algorithms [2]. The target of this study was to show how parallelism could be employed to reduce the intrinsic weight of some data and computation intensive operations.

A second study focused on *qualitative* aspects such as logical, functional and data dependencies among computational steps of an algorithm. The target was to define a parallel programming model, called **SPE Chain Model**, to help

* Big thanks goes to Gustavo Ovando, J.M.Moreno and M.J.Stefanini from Argentinian Space Agency (CONAE). This work has been partially supported by the CINECA Institute.

M. Danelutto, D. Laforenza, M. Vanneschi (Eds.): Euro-Par 2004, LNCS 3149, pp. 980–987, 2004.

developers, not necessarily (*hpc*) experts, to design parallel image processing algorithms.

According to the *SPE Chain Model*, the considered algorithms have been split into sequences of **Macro Phases (MP)** which represent aggregations of logically and functionally related computation steps. For each MP, *quantitative* aspects and *qualitative* aspects have been analyzed in order to establish *analytical performance models* used for predicting performance.

The main requirements for the *SPE Chain Model* were similar to those of similar tools [8] [7] [6] [9]: strong *object oriented (OO)* design [1], modularity, reusability and adherence to the analytical performance models. However, the *SPE Chain Model* was born from a generalization activity based on the study of a set of SAR algorithms. Thus it misses the richness and the complexity of other general purpose models (Active Objects, Distributed Shared Memory, Dynamic Load Balancing, wide area distribution and mobile agents).

A programming tool called **SPE** has been developed to implement algorithms designed according to the *SPE Chain Model*. *SPE* is based on two class libraries: **SPEAPI** used for writing parallel image processing algorithms and **SPEENG** a set of runtime support classes.

A set of different case studies using *SPE* shows that the model is very stable and general. In a scenario where domain experts cooperate with hpc developers, the design of new parallel image processing algorithms, turned out to be very intuitive and productive. There are also no significant difficulties porting already existing code to *SPE*. Tested algorithms show that the performance of their *SPE* implementations closely matches the performance of pre-existing, low level implementations, demonstrating that the use of high level programming constructs doesn't introduce a significant overhead. Moreover, algorithms implemented from scratch with *SPE*, are very efficient despite the short time it takes to develop them.

This paper presents the *SPE Chain Model* (section 2), the implementation of *SPE* (section 3) and the results obtained with two case study algorithms (section 4).

## 2 The SPE Chain Model

This section shows how an algorithm is designed and executed using *SPE Chain Model* and *SPE*.

### 2.1 Algorithm Design

Qualitative analysis of the studied algorithms, has shown that image processing algorithms can be split into sequences of *Macro Phases (MP)*. *MP*s are aggregations of computational steps that are functionally related because they exploit the same data, share a common stencil or can be executed concurrently.

For the *SPE Chain Model*, an algorithm is a sequence of *MP*s connected by entities called **Bindings**. *Bindings* are used to exchange data among *MP*s

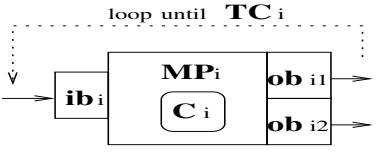


**Fig. 1.** Example of Macrophase Execution Loop.

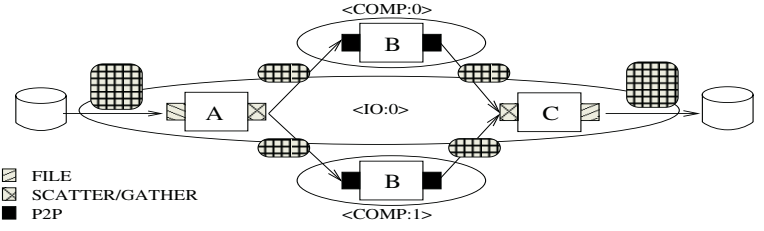


**Fig. 2.** Example of parallel program with 2 Node classes.

according to a specific transfer policy called the *type* of the *Binding*. A special kind of *Binding*, called **FILE Binding**, connects a *MP* to the file system.

Figure 1 shows the behaviour, or *Execution Loop* of a *Macro Phase* $MP_i$. $MP_i$ receives data from a set of preceding *MP*s through its *Input Binding* $ib_i$. It executes its code $C_i$ and finally it sends the output data through its *Output Bindings* (in order $ob_{i1}$ and $ob_{i2}$) to the following *MP*s. If $MP_i$ is *iterative*, its *Execution Loop* is repeated until a programmable *Termination Condition* $TC_i$ is verified.

### 2.2 Algorithm Execution

For the *SPE Chain Model*, a parallel program consists of a number of **Nodes** organized in **Node Classes**. A *Node* matches the concept of *process*. A *Node Class* is a group of possibly related *Nodes*. *Bindings* match the concept of inter-process communications [5].

A *Node* is programmed by statically assigning to it a sequence of *Macro Phases* and the parallelism degree (number of *Nodes*) of a *Node Class* is set statically in the algorithm parameters.

*Nodes* are identified by a pair <class-name:id> composed of their class and their class internal **Node ID**.

In the program of figure 2, data is read and split in two sub-images by *Macro Phase A*. Each *Macro Phase B* receives a sub-image through its *P2P Binding* and executes the algorithm specific sequential code on the sub-image. Finally the output sub-images are sent through a *P2P Output Binding* to a *Macro Phase C* who is responsible for recollecting the output image data and write it to the filesystem.

Figure 2 shows one possible allocation of the program where the *IO Class* has only one *Node* <IO:0> who is responsible for distributing and recollecting

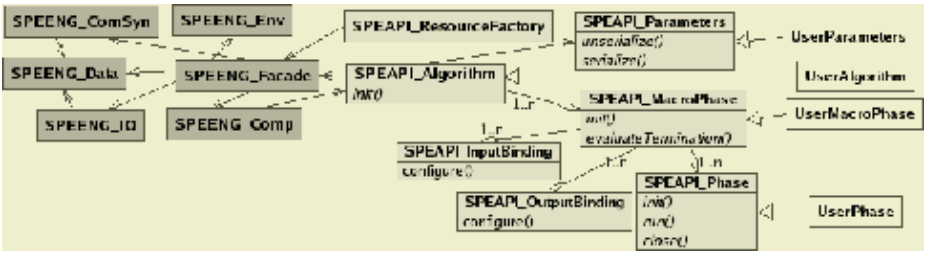


**Fig. 3.** Class diagram of SPE.

the data ($A$ and $C$) and *COMP Class Nodes* execute in parallel (2 process) the algorithm specific code.

Available types of *Bindings* are *FILE, P2P (point-to-point), MP (multi-point), BKCH (block-change), SCATTER, GATHER, ON_DEMAND and ROUND_ROBIN*.

Using pairs of *Input/Output Bindings*, it is possible to implement a great variety of communication schemes with well-known semantics and many of the common parallel patterns like *Farms or Pipelines* [5].

There is a special type of *Binding* called **BLOCKCHANGE (BKCH)**. Image data is often seen as a two-dimensional matrix. During the computation of an image processing algorithm it is often necessary to perform some operations on whole rows or on whole columns of the matrix [3]. In *SPE* the data is split among a set of *Nodes* according to a specific storing order (*row-major or column-major*). A couple of *BKCH Bindings* changes the storing order of the distributed image data. Assuming the *BKCH* operation as a standard operation in *SPE*, makes it possible for the programmer to benefit from a very optimized implementation of the routine. The way *SPE* is designed makes it possible for the *SPE* development team to rapidly integrate other highly specialized operations or communication patterns. This can be achieved by subclassing an abstract *Binding* class and overriding the methods that implement the underlying communication strategy.

## 3 A Sample Implementation of SPE

This section describes a sample implementation of *SPE* based on a class library for writing parallel algorithms conforming to the *SPE Chain Model* (*SPEAPI*) and a collection of classes that implement the runtime support for executing *SPEAPI* algorithms (*SPEENG*).

**The class library *SPEAPI*** was designed to ease the development of object oriented, modular, reusable, strong structured parallel image processing algorithms. The target is to enable a programmer with no particular *hpc* expertise, to design a parallel program by simply implementing the abstract methods *init(), run(), close()* and *evaluateTermination()* of the *SPEAPI* classes as shown in figure 3.

*SPEAPI_Algorithm* models a parallel algorithm and *contains* the sequence of *SPEAPI_MacroPhase* instances. *SPEAPI_MacroPhase* matches a *Macro Phase* and *contains* the lists of *SPEAPI_Binding* and a sequence of *SPEAPI_Phase* instances. The *SPEAPI_Phase* class is used for wrapping sequential code. *SPEAPI_Binding* classes encapsulate the concept of *Bindings*.

*SPEAPI_ResourceFactory* is a *singleton factory* class used for requesting memory buffers allocation.

***SPEENG*** implements the runtime support for algorithms written using *SPEAPI* with the idea to distribute responsibilities across a set of *systems* accessible through a singleton *facade* class called *SPEENG_Facade*.

*SPEENG_ComSyn* and *SPEENG_IO* are responsible for executing specific communication and file system access strategies related to *SPEAPI_Binding* instances.

For implementing these systems, *double-buffering* and *asynchronous communications* have been used to optimize performance and overlap computation and communication [5]. Moreover, *SPEENG* currently adopts *MPI* as communication software and standard *POSIX IO* for filesystem access. These two classes are *wrapper* classes that isolate *SPE* from all the implementation choices. Thus, adopting different choices implies just rewriting part of the code of *SPEENG_ComSyn* or *SPEENG_IO*.

*SPEENG_Data* is a *factory* class responsible for handling memory allocation and deallocation requests. *SPEENG_Env* and *SPEENG_Comp* are responsible respectively for storing environment information and managing the *Execution loop* of an *SPEAPI_Algorithm*.

As shown in figure 3, *SPEENG_Comp* accesses *SPEAPI_Algorithm* for managing its *Execution loop*. *SPEAPI_Algorithm* accesses *SPEENG_Facade* for requesting runtime support services (communication, IO, status or error notification, execution time). *SPEAPI_ResourceFactory* accesses *SPEENG_Facade* for requesting memory handling facilities.

## 4 SPE: Two Case Studies

This section shows two algorithms developed with the described sample implementation of *SPE*. The results, measured in terms of **generality of the model**, **usability** of *SPEAPI* and **performance**, show that there are no significant difficulties porting already existing code to *SPE* and the development of new efficient and scalable parallel algorithms, in a scenario where domain experts cooperate with hpc developers, is intuitive and productive.

The performance tests have been executed on the **CINECA** *Linux Beowulf Cluster* which has a peak performance of about 3 TeraFlops and is composed of 256 dual-processors (SMP) connected through a 2 Gbit/s network (*http://www.cineca.it/HPSystems/Resources/*).

**CSA***(Chirp Scaling Algorithm)* is an image focusing algorithm for *SAR* processing. The *SPE* development team had already made an experience in porting

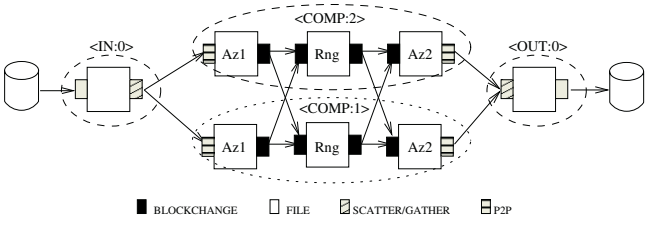


**Fig. 4.** SPEAPI structure of CSA.

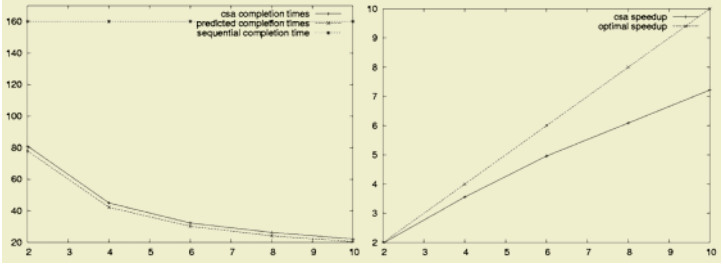


**Fig. 5.** Completion time, in seconds, (left) and speed up (right) for CSA.

a sequential prototype to a very optimized parallel implementation using low level tools (*C and MPI*). The porting to *SPE* validated the assessment related to the ease of integrating already existing code into a parallel program designed with *SPEAPI*. Moreover the performance of the *SPEAPI* version closely approximated the performance of the optimized low level implementation and of other similar solutions [4]. Figure 4 shows the *SPEAPI* structure of the algorithm. *CSA* first computes on the columns (*azimuth*) of the matrix representing the image. The computation is based on *Fourier Transforms (FT)* that require whole columns to be accessible locally on a *Node*. Data is stored inside *Macro Phase Az1* in column-major order. The second step of *CSA* computes *FT* on whole rows of the image (*range*). Thus the *Bindings* between *Az1 and Rng* are of type *BLOCKCHANGE* and a communication implies a change in the storing order. *Az2* is created to implement the last inverse FT in azimuth direction and another *BLOCKCHANGE* between *Rng1 and Az2* is necessary. The implementation requires two global exchanges of image data among all the *COMP Class Nodes*. Figure 5 shows the completion time with a varying number of *COMP Class Nodes* on image blocks sized 540 Mbyte. The adherence to the values provided by the analytical model can easily be seen and the *speedup*, compared to other results [3], is very encouraging.

**P-FLOOD** is an iterative algorithm that works on raster images and *DEM* (digital elevation model [2]) data to study the flow of water during a rainy timespan. *P-FLOOD* has been designed and developed from scratch using *SPEAPI* and its development demonstrated how *SPEAPI* can be used in a scenario where domain experts work together with programmers at the implementation

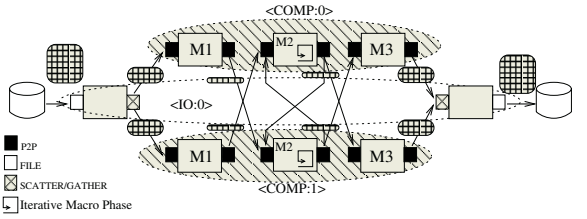


**Fig. 6.** SPEAPI structure of P-FLOOD.

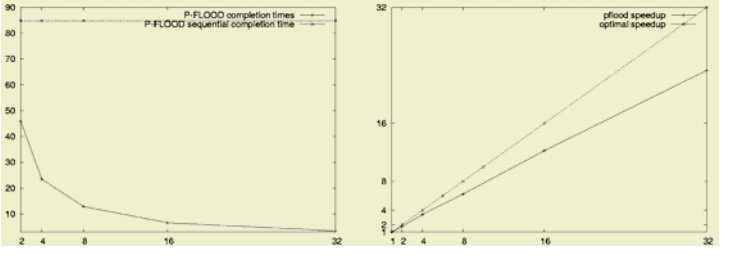


**Fig. 7.** Completion time, in seconds, (left) and speed up (right) for P-FLOOD.

of a parallel image processing algorithm. In *P-FLOOD* the data is read from a file by *Node* $< IO : 0 >$. The data is split in sub-images which are sets of rows and sent to the $< COMP : * >$ *Nodes*. The computational core is the iterative *Macro Phase M2* of figure 6. At each iteration the quantity of water at every pixel is computed. For this computation, information related to the nine neighbors of the pixel, in a 3x3 square stencil, have to be known. This implies that at each iteration of *M2*, $< COMP : i >$ has to exchange one row with both $< COMP : i - 1 >$ and $< COMP : i + 1 >$ to update the borders of its sub-image. This behaviour is modeled by the *P2P Input and Output Bindings* of *M2*. *M1* and *M3* are necessary to perform respectively the first and the last exchange of border rows.

The performance tests with a varying number of COMP Nodes (figure 7) have been executed on a *DEM* of 1000*1000 pixels (4Mbyte) with 50 iterations.

## 5 Conclusions and Future Work

This paper shows a new object oriented and strong structured abstraction called *SPE Chain Model* that simplifies design, implementation and performance modeling of parallel image processing algorithms. Moreover it presents *SPE*, a parallel programming tool that implements the *SPE Chain Model* and shows two sample algorithms that were implemented using *SPE*.

The results of this work are very positive in terms of *SPEAPI* usability, code reuse and performance. *SPE* is quite general since many new algorithms can be developed from scratch and already existing algorithms can easily be ported

to it. Performance tests showed that no significant overhead is introduced by the high level programming constructs because the performance of pre-existing implementations can be matched very closely. Moreover *SPE* enables the development of new parallel image processing algorithms that are fast and scalable.

The parallel algorithm implementations written to test *SPE* show that most of the code written with *SPEAPI* can be automated. This fact was used to produce a further abstraction layer based on an XML representation of the *SPEAPI* structure of an algorithm. The idea is to create a RAD tool which enables a programmer to easily design the parallel structure of an algorithm by simply interacting with graphical widgets.

## Acknowledgments

The design foundations of the platform utilized for carried out the work described in the paper has been funded by **Telespazio S.p.A.** within the ASI COSMO-SkyMed project.

## References

1. Connie U. Smith, LLoyd G. Williams: Performance Solutions, A Practical Guide To Creating Responsive, Scalable Software. Addison-Wesley, Object Technology (2001)
2. J.C. Curlander, R. N. McDnough: SYNTHETIC-APERTURE RADAR-SYSTEM AND SIGNAL PROCESSING. WILEY INTERSCIENCE, 1991
3. J. J. Mallorqui, M. Barà, A. Broquetas, M. Wis, A. Martinez, L. Nogueira, V. Moreno: PARALLEL ALGORITHMS FOR HIGH SPEED SAR PROCESSING.
4. Yiming Pi, Hui Long, Shunji Huang: A SAR PARELLEL PROCESSING ALGORITHM AND ITS IMPLEMENTATION. Department of Electronic Engineering, University of Electronic Science and Technology of China.
5. K. Hwang ADVANCED COMPUTER ARCHITECTURE: Parallelism, Scalability, Programmability. McGraw-Hill, Series in Computer Science (1993)
6. M. Vanneschi: The programming model of ASSIST, an environment for parallel and distributed portable applications. Parallel Computing, Vol. 28, Issue 12 (December 2002)
7. L.V. Kale, Sanjeev Krishnan: CHARM++ : A Portable Concurrent Object Oriented System Based On C++. Object Oriented Programming Systems, Languages and Applications, Sept-Oct 1993. ACM Sigplan Notes, Vol. 28, No. 10, pp. 91-108.
8. Chialin Chang, Alan Sussman, Joel Saltz: CHAOS++: RUNTIME SUPPORT FOR DISTRIBUTED DYNAMIC DATA STRUCTURES IN C++. CRPC Vol. 3 Issue 3 - Summer 1995
9. D. Caromel, F. Belloncle and Y. Roudier: The C++// system. Parallel Programming Using C++, G.Wilson and P. Lu editors, MIT Press, 1996, ISBN 0-262-73118-5