Parallelization of Dense Fluid Motion Estimation Application using OpenMP

Nitin Jain, Etienne Mémin, Christian Pérez

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Parallelization of Dense Fluid Motion Estimation Application using OpenMP

Nitin Jain — Etienne Memin — Christian Pérez

N° 4556

September 2002

THÈME 1

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--4556--FR+ENG

Parallelization of Dense Fluid Motion Estimation Application using OpenMP

Nitin Jain* , Etienne Memin , Christian Pérez

Thème 1 — Réseaux et systèmes
Projets Paris

Rapport de recherche n° 4556 — September 2002 — 19 pages

Abstract: In this paper, we discuss the parallelization of a computer vision application for dense fluid motion estimation using OpenMP. The application uses a recent algorithm based on energy-based motion estimator for predicting and analyzing the motion in image sequences showing fluid phenomenon. Standard techniques from computer vision are not well adapted for such images because of the great deal of spatial and temporal distortions in luminance patterns. The multiresolution multigrid framework of the application renders it amenable to parallelization after appropriate changes to the algorithm. We discuss why OpenMP is a suitable alternative to the conventional message passing model and compare it with other models based on various aspects. The results obtained on a SMP machine from different parallelization strategies are demonstrated and compared.

Key-words: OpenMP, dense fluid motion, parallelism, SMP

[*] This work has been supported by a grant from EGIDE, on behalf of French Ministry of Foreign Affairs, as part of collaboration between Institut de Rechereche en Informatique et en Automatique (INRIA), and Indian Institute of Technology (IIT)

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex (France)
Téléphone : +33 2 99 84 71 00 —Télécopie : +33 2 99 84 71 71

Parallélisation en OpenMP d'une application d'estimation de mouvement de fluide dense

Résumé : Ce papier décrit une version parallèle écrite en OpenMP d'une application de vision pour l'estimation du mouvement d'un fluide dense. L'application utilise un algorithme récent fondé sur un estimateur énergétique de mouvement pour prédire et calculer le mouvement d'un phénomène dense dans une séquence d'images. Les techniques standards d'imagerie numérique ne sont pas bien adaptées pour de telles images à cause des fortes distorsions spatiales et temporelles dans les schémas de lumière. L'algorithme de multi-resolution multi-grille utilisée par l'application a dû être légèrement adapté pour être parallélisé efficacement. Le papier discute l'utilisation d'OpenMP par rapport à l'alternative classique fondée sur l'échange de messages. Les résultats, obtenus sur une machine SMP avec différentes stratégies de parallélisation, sont présentés.

Mots-clés : OpenMP, mouvement de fluide dense, parallélisme, SMP

1 Introduction

In the field of computer vision, many standard techniques exist for motion estimation or optical flow determination. However, for certain specific domains like the fluid motion image sequences obtained from the environmental and medical sciences, the requirements of accuracy are sometimes very high. The general techniques often do not allow to estimate the motion fields with the required accuracy. Hence, a need is felt for specialized algorithms that are domain specific. That is to say mehtods that use data models which are specific to that domain, and hence give better results. It was such a need that motivated the VISTA project team at IRISA to develop an energy based dense motion estimator for the images showing fluid motion.

The domain specific algorithms tend to be more sophisticated than the general optical flow estimation algorithms. The Dense Motion Estimation algorithm requires a very high computation time when given large images as input. Since the application is required to calculate motion fields for images obtained from satellites, which are often very large, the use of the estimator as an operational tool which can work routinely, is not a practical proposition. The main goal then is to have a faster execution, even with large images.

We propose to reduce the computation time of the estimator by providing a parallel execution for it. We aim to parallelize the application incrementally without interfering with its multiresolution multigrid structure. Even though the parallelization is not possible without changes to the algorithm, we try to make as little change to the original algorithm as possible.

The parallel programming model that we use to parallelize the dense motion estimation application is the OpenMP language, which is an parallel programming model that is becoming popular. From the point of view of our needs, we justify our choice of OpenMP. We present a description of the features of OpenMP, and discuss how it compares with the other existing parallel programming standards like MPI.

The paper is organized as follows. In section 2, we start by describing the dense fluid motion estimator and present the pseudocode for the general understanding of the application. We present an analysis as to which parts of the application are computation intensive and the degree of parallelism that are associated with the various possible parallelization strategies. We also describe the data access patterns. We then present the expected performance for the different parallelization strategies. In section 3, we present the OpenMP programming model and compare it with the other existing standards. A brief description of OpenMP constructs and functionalities is presented. We go on to describe the parallelization strategies and the results obtained with them in section 4. We compare the actual performance of the algorithm with that of the expected performance and explain the difference. Analysis and conclusions based on these results are discussed. The paper draws some conclusion in section 5.

2 Dense Fluid Motion Estimation Application

The motion estimator on which we focus on in this study [1] aims at recovering the appparent motion field between two consecutive images of a sequence showing the evolution of a fluid flow. It is a dedicated adaptation of an Horn and schunck optical flow estimator [4]. Such an estimator is defined as the minimizer of a functionnal $H = H_1 + H_2$ composed of two terms. The first one encapsulates the data model term. This term gives an adequacy expression between the unkown (here the motion field $\boldsymbol{w}$) and the available observations (the luminance function $f(\boldsymbol{x}, t)$). The second term enforces a smoothness prior on the solution. This is a regularization function which make solvable the initial ill-posed problem (i.e recover the motion field directly from the luminance function variations).

In our case, instead of sticking to the usual brightness constancy assumption which assumes the constancy of the luminance of a point along its trajectory ($\frac{df}{dt} = 0$), the data model dedicated to fluid motion relies on an integration of the continuity equation of fluid mechanics. The corresponding functional reads:

$$H_1(\boldsymbol{w}) = \int_\Omega \rho\Big\{f\left(\boldsymbol{x} + \boldsymbol{w}(\boldsymbol{x}), t+1\right)\exp\left(\mathrm{div}\boldsymbol{w}(\boldsymbol{x})\right) - f(\boldsymbol{x}, t)\Big\}d\boldsymbol{x}. \qquad (1)$$

This data model accounts for the luminance variation which accompanies the luminance variation in areas where the fluid exhibits a divergent motion. It has to be noted that for a null divergence this term comes to the usual brightness constancy term. The penalty function ρ is a robust penalty function allowing us to deal with significant deviation of the data model. It allows to implicitely reject points which are not in accordance with such a model.

The smoothness prior used in conjonction with this data term relies on a second order div-curl regularization:

$$\int_\Omega \Big(||\boldsymbol{\nabla}\mathrm{div}\boldsymbol{w}(\boldsymbol{x})||^2 + ||\boldsymbol{\nabla}\mathrm{curl}\boldsymbol{w}(\boldsymbol{x})||^2\Big)d\boldsymbol{x}. \qquad (2)$$

Contrary to a classical first order regularization, such a smoothness term allows to preserve compact areas exhibiting high concentration of curl (also called vorticity) and/or divergence. Let us remark that an under-estimation of the divergence is all the more problematic in our case, since the data model includes an explicit use of this quantity.

This kind of regularization is nevertheless very difficult to implement directly. As a matter of fact, the associated condition of optimality (the Euler-Lagrange equations) consists in two fourth-order coupled PDE's, which are tricky to solve numerically. This functionnal has been therefore simplified by introducing auxiliary functions, and defining the alternative functional:

$$H_2(\boldsymbol{w}, \xi, \zeta) = \alpha\int_\Omega |\mathrm{div}\boldsymbol{w} - \xi|^2 + \lambda\rho(||\boldsymbol{\nabla}\xi||) + \alpha\int_\Omega |\mathrm{curl}\boldsymbol{w} - \zeta|^2 + \lambda\rho(||\boldsymbol{\nabla}\zeta||). \qquad (3)$$

The new auxiliary scalar functions ξ and ζ can be respectively seen as estimates of the divergence and the curl of the unknown motion field, and λ is a positive parameter. The first part of each integral encourages the displacement to comply with the current divergence and vorticity estimates ξ and ζ, through a quadratic goodness-of-fit enforcement. The second part equips the divergence and the vorticity estimates with a robust first-order regularization enforcing piece-wise smooth configurations. Getting rid of the auxiliary scalar fields ξ and ζ in (3) (by setting $\xi = \text{div}\boldsymbol{w}$ and $\zeta = \text{curl}\boldsymbol{w}$) would amount to the original *second-order* div-curl regularization (2), if ρ is the quadratic penalty function. From a computational point of view, regularizing functional (3) only implies the numerical resolution of first-order PDE's. It can be shown for the L_2 norm that this simplified div-curl regularization consists in a smoothed version of the original second order div-curl regularization [2].

To allow the computation of long range displacements the estimator has been encapsulated within an incremental multiresolution structure. Let us note that it is the integrated nature of the data constraint used here (the model is defined for displacements as well as for velocities) which authorizes the implementation of such a scheme.

Multiresolution scheme consists to implement an incremental estimation scheme on a pyramidal hierarchical representation of the image data [3]. At a given resolution level, a incremental displacement field is computed considering that the main components of the displacements is known. This main component is indeed supposed to be estimated at a previous resolution level. It is considered as being null at the coarsest level. Such a multiresolution estimator corresponds indeed to a kind a Gauss-Newton strategy for non-linear least squares minimization [5].

2.1 Minimization issue

The minimization of the functionnal is considered through a direct discretization of H_1 and H_2. The different functions involved in the functionnal have been dicretized on the image lattice. A particular attention has been payed for the discretization of divergence and curl operator for which an uncentered discretization scheme has been used.

The overhall system is constituted by two main sets of variables that have to be estimated. The first one is the motion field $\boldsymbol{w}$, and the second set comprizes the two scalar fields ξ and ζ. The estimation is conducted alternatively by minimizing $H_1 + H_2$ with respect to $\boldsymbol{w}$, ξ and ζ respectively. For the motion field, considering the div and curl estimates ξ and ζ as being fixed, the robust minimization with respect to $\boldsymbol{w}$ is solved with an iteratively reweighted least squares technique. This optimization is embedded in an efficient multi-parametric adaptive multigrid framework [5]. This technique consists to solve the corresponding problem for a subset of solution subspaces of increasing size. Each of these subspaces are defined as the space of piece-wise parametric solutions on a square-blocks grid. The hierarchy is built considering a hierarchy of nested square-blocks grids. For each of these grid, one has to solve a large sparse linear system.

Then in turn, the motion field $\boldsymbol{w}$ being fixed, the minimization of H with respect to ξ and ζ is in fact equivalent to the minimization of H_2 and is again conducted using an iteratively reweighted least squares technique. More details of the minimization issues can be found in [1].

2.2 Pseudocode

The application computes the incremental velocity fields and keeps adding them to the motion fields, until the convergence criteria are met. The velocity fields are stored as separate arrays for each component. To allow for a multiresolution of pyramids, and a multigrid structure at each level of the pyramid, the image array is divided into blocks. The elements of a block occur together on the image. As the computation moves to the lower grid level, each block is subdivided into 4 blocks, each having a size one-fourth of the original block. At the lowest resolution and lowest grid level, each block consists of one image pixel only.

Following is the listing of the pseudocode of the principal section of the application.

```
do from coarse resolution to fine resolution (multiresilution setup)
   do from coarse scale to fine scale        (multigrid scheme)
      /*pre-estimation operations */                   (SEQUENTIAL I)
      Construct the observations of ft, fx, and fy
      /*Estimation*/
      do until convergence of the overall system
         do until convergence
            Calculate the incremental velocity fields  (PARALLEL I)
         end do
         Estimate the new values of $\xi$ and $\zeta$            (PARALLEL II)
      end do
      /* post estimation operations */                 (SEQUENTIAL II)
      Add incremental velocity fields
      Warp the image t+1 accordingly
   end do
   project fields to next resolution level
end do
```

The outermost loop in the pseudocode that goes from coarse resolution to fine resolution corresponds to the multiresolution pyramid structure of the application. Hence one iteration over this loop computes the incremental velocity fields at a pyramid level k. This field is projected to the next level, and the same procedure is repeated for the next level k+1 in the next iteration of the loop. Within each resolution level is a multigrid structure, which is represented by the inner do loop from coarse scale to fine scale. At the end of computation at each grid level, the incremental velocity field calculated at that level is added to the velocity fields, and then this is repeated for the next grid level till the finest grid level is reached. Within every grid level, there are two nested convergence loops, which estimate the incremental velocity fields over all the blocks of the image till the convergence criteria are met.

The calculation of the incremental velocity fields can be done in parallel over the different blocks. This parallel computation, with separate blocks assigned to separate processors, is henceforth referred to as the Inter-Block parallelism. Another approach toward parallelism would be to do computations over different elements of the block in parallel. So, different processors would then be performing computation over the different parts of the same block. This sort of parallel computation is referred to as the Intra-Block parallelism. The estimation of the new divergence and the new curl is done after each convergence of the inner convergence loop. This computation occurs over the entire image array, and can also be done in parallel.

2.3 Analysis

2.3.1 Timing Percentages of the Sequential Application

The times spent in the various parts of the application are indicated in Table 1 and Table 2. The labels SEQUENTIAL I, SEQUENTIAL II, PARALLEL I, PARALLEL II are marked on the pseudocode of the previous section.

As can be seen from Tables 1 and 2, the application spends 85.3 per cent of computation time in the parallelizable part for the small image, and 94.1 per cent for the medium sized image. As the image size increases, the application spends a greater percentage of its computation time in the parallelizable part. This indicates the possibility of achieving a good speed-up for the large images on which the application will be run.

	Time in seconds	Percentage
SEQUENTIAL I	9.392	13.86
PARALLEL I	26.637	39.31
PARALLEL II	31.162	46.00
SEQUENTIAL II	0.565	0.83
Total	67.756	100

Table 1: Timing of the Sequential Application for the small image (128 × 128)

	Time in seconds	Percentage
SEQUENTIAL I	31.624	4.66
PARALLEL I	453.807	66.93
PARALLEL II	184.100	27.15
SEQUENTIAL II	8.567	1.26
Total	678.098	100

Table 2: Timing of the Sequential Application for the medium sized image (680 × 480)

2.3.2 Degree of Parallelism

Table 3 indicates the size and the number of blocks at various grid levels of the application. The Number of blocks indicates the degree for parallelism[1] between blocks (Inter Block Parallelism) while the Number of elements indicates degree for parallelism within blocks (Intra Block Parallelism)

2.3.3 Data Access

For computation within the block, each point uses the information of the actual and incremental velocity fields at the points in its neighborhood. Hence, at the block boundaries, there are interactions with the field values at the boundaries of the neighboring blocks. The list of blocks is initially constructed by a simple partitioning of the image. The consecutive blocks in the list are the blocks that are consecutive on the image row-wise. At successive grid levels, each block is broken into 4 blocks, and these 4 blocks appear consecutively on the list of blocks. For the Inter-Block parallelism strategy, since each element accesses velocity field information for only the points in its neighborhood, all the inter-processor communication occurs for the block boundary points information.

2.3.4 Parallelization Strategy

The Intra-Block parallelism aims to parallelize the computation within each block of the image. In a single iteration of the convergence loop, a function traverses the linked list of the blocks once, and on each block the incremental velocity field is calculated by solving a system of linear equations of the form $Ax = b$. The calculation of the coefficients of the matrix A and the vector b is done by using the velocity field information at each point inside the block, and this can be done in parallel because the results obtained at different points inside the block involve reductions to the coefficients of the matrix and the vector. As can be seen from table 3, the largest block that is encountered for a medium sized image is 32×32 only (for 2 resolution levels). The blocks at the other grid levels are much smaller. The parallel computation over such small blocks is not expected to surpass the overheads associated with parallelization. Hence, no significant speed-up is expected. Thus, it does not appear interesting to parallelize using the Intra-Block strategy.

Another strategy for parallelization is the Inter-Block strategy which involves computation over different blocks in parallel. So a single processor would be assigned the computation of a single block, and when it completes that block, it would be assigned another block for computation. Since the number of blocks would in general be much larger than the number of available processors, this strategy is expected to achieve good results. Table 4 summarizes the expected improvements in time based on the parts that have been parallelized.

The parallel version of the algorithm differs from the sequential one in the respect that while the sequential version follows a Gauss Seidel algorithm for convergence, the parallel version uses a Jacobi convergence scheme. In the Gauss Seidel scheme, the velocity array

[1] Let define the degree of parallelism as the amount of work that can be done in parallel.

Resolution level	Grid level	Configuration of Blocks	Number of blocks	Block Size	Elements per block
1	1	170 × 120	20400	2 × 2	4
1	0	340 × 240	81600	1 × 1	1
0	1	340 × 240	81600	2 × 2	4
0	0	680 × 480	326400	1 × 1	1
1	2	85 × 60	5100	4 × 4	16
1	1	170 × 120	20400	2 × 2	4
1	0	340 × 240	81600	1 × 1	1
0	2	170 × 120	20400	4 × 4	4
0	1	340 × 240	81600	2 × 2	1
0	0	680 × 480	326400	1 × 1	4
1	3	43 × 30	1290	8 × 8	64
1	2	85 × 60	5100	4 × 4	16
1	1	170 × 120	20400	2 × 2	4
1	0	340 × 240	81600	1 × 1	1
0	3	85 × 60	5100	8 × 8	64
0	2	170 × 120	20400	4 × 4	16
0	1	340 × 240	81600	2 × 2	4
0	0	680 × 480	326400	1 × 1	4
1	4	22 × 15	330	16 × 16	256
1	3	43 × 30	1290	8 × 8	64
1	2	85 × 60	5100	4 × 4	16
1	1	170 × 120	20400	2 × 2	4
1	0	340 × 240	81600	1 × 1	1
0	4	43 × 30	1290	16 × 16	256
0	3	85 × 60	5100	8 × 8	64
0	2	170 × 120	20400	4 × 4	16
0	1	340 × 240	81600	2 × 2	4
0	0	680 × 480	326400	1 × 1	1
1	5	11 × 8	88	32 × 32	1024
1	4	22 × 15	330	16 × 16	256
1	3	43 × 30	1290	8 × 8	64
1	2	85 × 60	5100	4 × 4	16
1	1	170 × 120	20400	2 × 2	4
1	0	340 × 240	81600	1 × 1	1
0	5	22 × 15	330	32 × 32	1024
0	4	43 × 30	1290	16 × 16	256
0	3	85 × 60	5100	8 × 8	64
0	2	170 × 120	20400	4 × 4	16
0	1	340 × 240	81600	2 × 2	4
0	0	680 × 480	326400	1 × 1	1

Table 3: Degrees of Parallelism

values were used in the same iteration that they were calculated in. This was replaced by the Jacobi scheme, where the values used in one iteration were all the values that had been calculated in the previous iteration. Two separate velocity field arrays were hence maintained, one for reading the values, and the other for writing.

Two important facts about the Gauss-Seidel method should be noted. First, the computations in Gauss Seidel method have to be serial. Since each component of the new iterate depends upon all previously computed components, the updates cannot be done simultaneously as in the Jacobi method. Second, the new iterate depends upon the order in which the equations are examined. The Gauss-Seidel method is sometimes called the method of successive displacements to indicate the dependence of the iterates on the ordering. If this ordering is changed, the components of the new iterate (and not just their order) will also change. The change of the algorithm is therefore expected to result into a change in the number of iterations toward convergence.

2.3.5 Speed-up

The speed-up and the efficiency are classically defined:

$$\text{speed-up} = \text{sequential execution time}/\text{parallel execution time}$$

$$\text{efficiency} = \text{speed-up}/\text{number of processor}$$

In order to evaluate the the benefit coming from the parallelization of the two parallel section we have identified, we estimate the parallel time for different number of processor. The number, presented in Table 4 and in Figure 1, have been obtained from Table 1 and Table 2. Assuming a perfect parallelization, the parallel time has been divided by the number of processor. So, Table 4 and Figure 1 show the speed-up increases as the number of processors increases. However, the efficiency decreases because the sequential part becomes significant.

No. of processors	Time in seconds	Speed-up	Efficiency
1	678.10	1.00	1.00
2	359.15	1.89	0.95
4	199.67	3.40	0.85
8	119.93	5.66	0.71
16	80.060	8.47	0.53
32	60.125	11.28	0.35

Table 4: Expected times and speed-ups for the medium sized image (680 × 480)

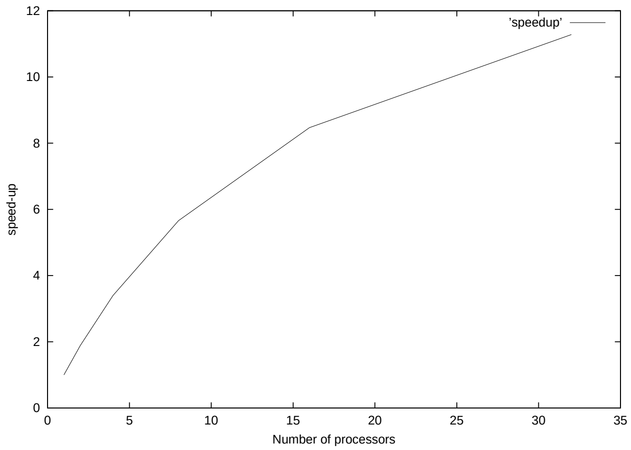

Figure 1: The speed-up graph

3 OpenMP

At its most elemental level, OpenMP is a set of compiler directives and callable runtime library routines that extend FORTRAN (and separately, C and C++) to express shared-memory parallelism. It leaves the base language unspecified, and vendors can implement OpenMP in any FORTRAN compiler. Naturally, to support pointers and allocatables, FORTRAN 90 and FORTRAN 95 require the OpenMP implementation to include additional semantics over FORTRAN 77.

3.1 OpenMP Design Objective

OpenMP was designed to be a flexible standard, easily implemented across different platforms. The standard comprises four distinct parts: control structure, the data environment, synchronization, and the runtime library.

3.1.1 Control structure

OpenMP strives for a minimalist set of control structures. Experience has indicated that only a few control structures are necessary for writing most parallel applications e.g. the parallel directive provides arguably the most widely used shared-memory programming model in scientific computing. OpenMP includes control structures only in those instances where a compiler can provide both functionality and performance over what a user could reasonably program.

3.1.2 Data environment

Associated with each process is a unique data environment providing a context for execution. The initial process at program start-up has an initial data environment that exists for the duration of the program. It contructs new data environments only for new processes created during program execution. The objects constituting a data environment might have one of three basic attributes: shared, private, or reduction.

3.1.3 Synchronization

There are two types of synchronization: implicit and explicit. Implicit synchronization points exist at the beginning and end of parallel constructs and at the end of control constructs (for example, do and single). In the case of do sections, and single, the implicit synchronization can be removed with the nowait clause. The user specifies explicit synchronization to manage order or data dependencies. Synchronization is a form of inter-process communication and, as such, can greatly affect program performance. In general, minimizing a program's synchronization requirements (explicit and implicit) achieves the best performance. For this reason, OpenMP provides a rich set of synchronization features, so developers can best tune the synchronization in an application.

3.1.4 Runtime library and environment variables

In addition to the directive set described, OpenMP provides a callable runtime library and accompanying environment variables. The runtime library includes query and lock functions. The runtime functions allow an application to specify the mode in which it should run. An application developer might wish to maximize the system's throughput performance, rather than time to completion. In such cases, the developer can tell the system to dynamically set the number of processes used to execute parallel regions. This can have a dramatic effect on the system's throughput performance with only a minimal impact on the program's time to completion.

3.1.5 Summary

Table 6 summarizes the functionalities provided by the OpenMP API. It categorizes them into one of three categories: control structure, data environment, or synchronization. The standard also includes a callable runtime library with accompanying environment variables.

3.2 Why OpenMP?

MPI [6] has effectively standardized the message-passing programming model. It is a portable, widely available, and accepted standard for writing message-passing programs. Unfortunately, message passing is generally a difficult way to program. It requires that the program's data structures be explicitly partitioned, so the entire application must be

Functionality	OpenMP
Overview	
Orphan scope	Yes, binding rules specified
Query functions	Standard
Runtime functions	Standard
Environment variables	Standard
Nested parallelism	Allowed
Throughput mode	Yes
Conditional compilation	OPENMP
Control structure	
Parallel region	Parallel
Iterative	Do
Noniterative	Section
Single process	Single,Master
Early completion	User coded
Sequential Ordering	Ordered
Data environment	
Autoscope	Default(private)
	Default(shared)
Global objects	Threadprivate
Reduction attribute	Reduction
Private initialization	Firstprivate
	Copyin
Private persistence	Lastprivate
Synchronization	
Barrier	Barrier
Synchronize	Flush
Critical section	Critical
Atomic update	Atomic
Locks	Full functionality

Table 5: OpenMP functionality.

parallelized to work with the partitioned data structures. There is no incremental path to parallelize an application.

The POSIX thread API [8], is an accepted standard for shared memory in low-end systems. However, it is not targeted at the technical, HPC space. There is little FORTRAN support for POSIX threads. Even for C applications, the POSIX threads model is awkward, because it is lower-level than necessary for most scientific applications and is targeted more at providing task parallelism, not data parallelism. Also, portability to unsupported platforms requires a stub library or equivalent workaround.

Researchers have defined many languages for parallel computing, but these have not found mainstream acceptance. High-Performance FORTRAN (HPF) [7] is the most popular multiprocessing derivative of FORTRAN, but it is mostly geared toward distributed-memory systems.

Developers need to parallelize existing code without completely rewriting it, but this is not possible with most existing parallel-language standards. Only OpenMP allows incremental and scalable parallelization of existing code. OpenMP is targeted at developers who need to quickly parallelize existing scientific code, but it remains flexible enough to support a much broader application set. OpenMP provides an incremental path for parallel conversion of any existing software. It also provides scalability and performance for a complete rewrite or entirely new development.

However, OpenMP has a major limitation that the programs written using OpenMP can not be executed over distributed memory machines, which are more popular and more economical than the SMP machines for which OpenMP has been designed. Therefore, MPI continues to be the most popular parallel programming paradigm.

Our goal is to obtained a faster execution for the application. We need to have an incremental way of parallelizing the application. At the same time, we want to make minimal changes to the algorithm and the data structures involved therein. All these factors contributed to our choice of using OpenMP for parallelization.

	MPI	POSIX threads	OpenMP
Scalable	yes	sometimes	yes
Incremental parallelization	no	yes	yes
Portable	yes	yes	yes
FORTRAN binding	yes	no	yes
High level	no	no	yes
Supports data parallelism	no	no	yes
Performance oriented	yes	no	yes

Table 6: Comparing standard parallel-programming models.

4 Parallelization of the Application

This section describes the approaches toward parallelism, and presents and analyzes the results obtained with them. After the parallelization had been done using OpenMP, we carried out the tests. All tests were done on a SMP machine using the pgcc 3.2-3 compiler from PGI. The SMP machine had four 550 MHz Pentium III processors.

4.1 Intra-Block Parallelism

Here, the computation within each block is parallelized. Table 7 presents the execution times of the parallel version of the code. As can be seen from the table, the execution times increase with the increase in the number of processors. The block of the different resolution and grid levels, presented in Table 3, are too small. So, the overheads associated with this parallelization are not surpassed by the increase in performance, and hence this strategy did not yield good results, as expected. Hence we decided not to use this strategy for the parallelization of the application.

No. of processors	Time in seconds
1	436.063
2	510.141
4	683.583

Table 7: Results obtained for the Intra-Block parallelism for the medium sized image

4.2 Inter-Block Parallelism

The Tables 8 and 9 present the times spent in various parts of the parallel code with different number of processors for the small image as well as the medium image, and compares this to the corresponding times spent in the same portions by the sequential version of the application. The times spent in the parallel region go down considerably and this indicates a good speed-up. Also, it is notable that the time spent in the part marked sequential I is decreasing. This is because this part calls a function that is also called from the parallelized parts, and that function has been parallelized. There is a difference between the expected and the actual running times of the parallel application. This difference can be accounted for by the increase in the number of iterations toward convergence at any grid level. As described earlier in section 2.3.4, the algorithm has been modified to Jacobi scheme, and this has led to the increase. The Gauss Seidel scheme converges faster, but this change was necessary because Gauss Seidel scheme does not accommodate computation in parallel. So, although the time for the execution of a single iteration goes down, the number of iterations increase and hence the program spends more time than expected. The increase in the number of iterations is presented in tables 9 and 12. For the small image, the increase in the number of iterations accounts for about 1.2 seconds, while in the case of medium sized image, this increase is about 56.6 seconds. However, despite this difference, the decrease in the execution time for the program is still significant.

4.3 Conclusion

The Intra-Block strategy of parallelization does not give good results for this application, and that is mainly because the block sizes are too small so that parallelism on them does not

#procs	SEQUENTIAL I	PARALLEL I	PARALLEL II	SEQUENTIAL II	Total
1	9.415	27.790	28.351	579.125	644.681
2	6.255	17.105	16.975	605.310	645.645
4	5.655	9.416	10.333	581.525	606.929

Table 8: Times spent in seconds in the different parts for parallel Application for the small sized image (128 × 128)

#procs	SEQUENTIAL I	PARALLEL I	PARALLEL II	SEQUENTIAL II	Total
1	30.967	518.716	151.256	9.656	710.595
2	19.816	326.956	95.083	6.210	448.065
4	14.409	181.686	66.791	8.977	271.864

Table 9: Times spent in seconds in the different parts for parallel Application for the medium sized image (680 × 480)

No. of processors	Res 1	Res 1	Res 1	Res 0	Res 0	Res 0
	Grid 2	Grid 1	Grid 0	Grid 2	Grid 1	Grid 0
Sequential	131	170	13	209	350	9
1	160	219	13	205	360	9
2	160	218	13	208	356	9
4	160	217	13	206	363	9

Table 10: Number of iterations for small image

No. of processors	Predicted time in seconds	Actual time in seconds
1	67.757	69.032
2	38.857	40.942
4	24.407	25.986

Table 11: Estimated vs Actual times for execution of the parallel code for the small image

overtake the overheads resulting from parallelization. On the other hand, the Inter-Block parallelism strategy gives good results. The difference between the expected and the actual time of execution can be accounted for by the increase in the number of iterations toward convergence at each grid level. Despite that, the speed-up is significant

No. of processors	Res 1	Res 1	Res 1	Res 0	Res 0	Res 0
	Grid 2	Grid 1	Grid 0	Grid 2	Grid 1	Grid 0
Sequential	118	166	9	216	224	15
1	147	207	9	220	265	15
2	146	206	9	222	267	15
4	148	210	9	221	270	15

Table 12: Number of iterations for medium sized image

No. of processors	Predicted time in seconds	Actual time in seconds
1	678.100	710.598
2	359.146	448.106
4	199.669	271.864

Table 13: Estimated vs Actual times for execution of the parallel code for the medium sized image (680 × 480)

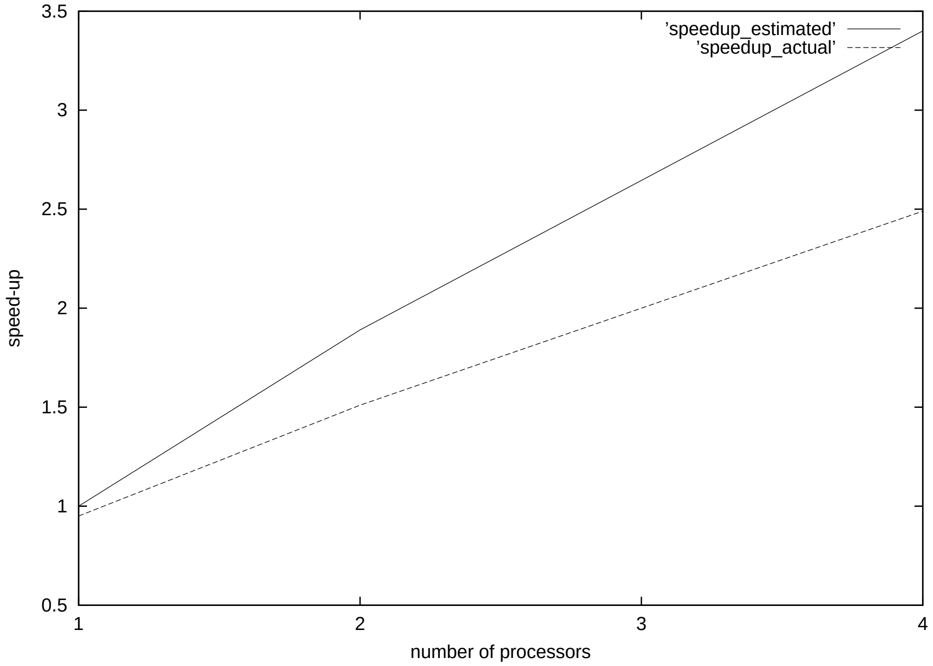

Figure 2: The speed-up graph for the medium sized image (640 × 480)

5 Conclusion

With the advent of new and sophisticated algorithms for image processing and vision, the need to parallelize them follows in order the need to effectively use them. Toward this end, we chose the dense fluid motion estimation application and aimed to parallelize it using OpenMP. OpenMP is a standard for parallelization which is more promising than the other existing standards, and is becoming popular. In this paper, we presented our experience with OpenMP parallelization with respect to a dense fluid motion estimation application on a shared memory multiprocessors.

The application that we parallelized addresses the problem of estimating and analyzing the motion in image sequences showing fluid phenomenon. It uses a dedicated energy-based motion estimator. The data model used is embedded in a multiresolution framework, and the optimization of the global energy function is solved within an efficient multigrid scheme. The application could broadly be divided into four sections, two of which took most of the execution time of the program. These two sections of the code could be parallelized, while the other two remained sequential. We have presented the times spent in various parts, the degree of parallelism, the data access patterns and the strategy of parallelization that could be expected to work best.

We used the OpenMP programming model for parallelizing our application. OpenMP provides a fast way of incrementally parallelizing the application with minimal changes to the data structures and the algorithm in general.

We have demonstrated our results achieved on the test images using Intra-Block as well as the Inter-Block strategy. The Intra-Block parallelism does not achieve good results because, due to the small size of the blocks, the speed-ups achieved do not surpass the parallelism overheads. This was as expected, and hence it is not an adequate solution for this application. On the other hand, with the Inter-Block parallelization strategy, the processors are assigned different blocks for computation, and each block's computation is done by a single processor. This achieves significant speed-ups and considerably reduces the execution time. Hence it is practical to follow this approach. With sufficiently large number of processors, it is now possible to use the application as an operational tool for routine work.

The parallel version of the application has been developed to run on shared memory multiprocessors. Work has been going on to have OpenMP execution on top of PC cluster, especially using software distributed shared memory (SDSM). In the future, it would be interesting to have an execution of the application for DSM architecture. It would considerably increase the usability of the application.

References

[1] T. Corpetti, E. Mémin, and P. Pérez. Dense estimation of fluid flows. *IEEE Trans. Pattern Anal. Machine Intell.*, 24(3):365–380, 2002.

[2] T. Corpetti, E. Mémin, and P. Pérez. Dense motion analysis in fluid imagery. In *European Conference on Computer Vision, ECCV'02*, pages 676–691, 2002.

[3] W. Enkelmann. Investigation of multigrid algorithms for the estimation of optical flow fields in image sequences. *Comp. Vision Graph. and Image Proces.*, 43:150–177, 1988.

[4] B. Horn and B. Schunck. Determining optical flow. *Artificial Intelligence*, 17:185–203, 1981.

[5] E. Mémin and P. Pérez. Hierarchical estimation and segmentation of dense motion fields. *Int. J. Computer Vision*, 46(2):129–155, 2002.

[6] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, J. Dongarra. MPI: The complete reference. MIT Press. 1995.

[7] High Performance Fortran Forum. High Performance Fortran Language Specification, Version 2.0. Rice University, Houston, Texas, october 1996

[8] IEEE Standards Department. 1003.4d8 POSIX System Application Program Interface: Threads Extensions [C language]. 1994.

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399