Integrated Computer-Aided Engineering 18 (2011) 365–378
DOI 10.3233/ICA-2011-0382
IOS Press

Integration of emerging computer technologies for an efficient image sequences analysis

Luisa D'Amore[a,*], Daniela Casaburi[b], Ardelio Galletti[c], Livia Marcellino[c] and Almerico Murli[d]
[a]*University of Naples Federico II, and SPACI, Complesso Universitario M.S. Angelo, Via Cintia, Naples, Italy*
[b]*SPACI, c/o Complesso Universitario M.S.Angelo, Via Cintia, Naples, Italy*
[c]*University of Naples Parthenope, Centro Direzionale, Is. C4, Naples, Italy*
[d]*SPACI and University of Naples Federico II, Complesso Universitario M.S. Angelo, Via Cintia, Naples, Italy*

Abstract. Real time image sequences analysis is a challenge. Using high performance computing technologies, a parallel algorithm for performing data sequence analysis is proposed. We call it *pipelined algorithm* (PA). The idea underlying the design of PA comes from the *Pipes and Filters* design approach: to partition the sequence into ordered subsets and to *overlap tasks execution via pipelining*.
Moreover, in order to improve the performance gain of the PA algorithm, tasks' execution is distributed among multicore processors. The approach chosen for introducing concurrency takes into account the hierarchical parallelism of system architecture of multicore multiprocessors. More precisely, three parallelization strategies of PA are considered: first strategy distributes the execution of each task among the same number of cores employing a fine-grained task parallelism (we call it *inter-task data parallelism*), second strategy refers to the execution of each task to one core introducing concurrency at a coarser level (we call it *intra-task functional parallelism*), and the last one combines the previous two approaches: it refers to the mapping of each task to a group of cores (intra-task functional parallelism) distributing task's execution within each group (inter-task data parallelism). We prove, both theoretically and experimentally that the third strategy is more effective than the others in terms of speed up improvement as the data length increases.
As testbed the segmentation of ultrasound sequences is considered. Experiments on real data are carried out using a multicore-based parallel computer system relying on PETSc (Portable Extensible Toolkit for Scientific computation), a high level software computing environment.

Keywords: Image sequence, pipe and filters, parallel computing, multicore processors

1. Introduction

Many problems of science and engineering often involve the application of a set of incremental transformations (i.e. ordered computations) to a finite stream of data. This set of operations can be thought of as the integration, composition and cascade of processing tasks (see, for instance [2,8,9,16–18]). Each task performs a different step of the overall computation following a precise order. The order of execution defines the dependence among tasks and synchronizes their execution in such a way that the overall computation seems to be intrinsically sequential. In such cases, a standard approach may become too expensive, especially for real-time computations.

In this paper, we propose a parallel algorithm oriented to perform such kind of operations. The idea underlying this algorithm is a quite general computing methodology for efficiently performing such kind of operations: tasks can be seen as vertices in a task graph and the dependencies from one task to another can be seen as a directed edge in a task graph. Following the *Pipe and Filter* pattern approach, concurrency

*Corresponding author: Prof. Luisa D'Amore, Università di Napoli Federico II, Complesso Universitario Monte S. Angelo, Via Cintia, 80126 Napoli, Italy. Tel.: +39 081675625; E-mail: luisa.damore@unina.it.

ISSN 1069-2509/11/$27.50

is introduced along a linear branch in the task graph by suitably *overlapping tasks execution via pipelining* while tasks along parallel branches run concurrently in parallel. This means that the first task begins to compute as soon as the first data are available. When its computation is finished the result data is passed to another task, following the prescribed order of the transformations. Then, while this computation takes place on the data, the first task is free to accept new data. This pattern is called *Pipes and Filters* since data is passed as a flow from one computation stage to another along a pipeline. The key feature is that data are passed just one way through the flow structure [5,13]. We call this algorithm: *pipelined algorithm* (PA).

Moreover, in order to reduce synchronization overheads due to loads imbalance and communications among tasks, each task of PA is further parallelized and executed on multicore processors. The approach chosen for introducing concurrency inside tasks execution takes into account the hierarchical parallelism of system architecture of multicore multiprocessors.

More precisely, three parallelization strategies of PA are considered: first strategy distributes task's execution among multiple cores employing a fine-grained task parallelism (we call this strategy *inter-task data parallelism*) of each task, second strategy introduces concurrency among tasks at a coarser level (*intra-task functional parallelism*), and the last one combines the previous two: it assigns the execution of each task to a group of cores distributing task's operations within each group of cores.

In conclusion, main contribution of this work is to combine the functional parallelism underlying pipelined computations and data parallelism underlying parallel computing with the aim of getting a parallel algorithm for data sequence analysis. This idea takes into account the hierarchical parallelism of new system architectures based on multicore multiprocessors. The algorithm was implemented using the high level software library PETSc(Portable Extensible Toolkit for Scientific computations) computing environment. PETSc was mainly chosen because of its optimal management of both data layout and communications. Finally, we provide a detailed analysis both theoretic and experimental of the performance of these three strategies on a multicore multiprocessor.

As case study, we consider the analysis of image sequences. In particular, we focus on an application in Medical Imaging: the segmentation of ultrasound image sequences.

The paper is organized as follows. In next section is briefly reviewed the Pipe and Filter pattern-oriented approach. In Section 3, the image sequence analysis problem is introduced. Then we describe the pipelined algorithm and the parallelization strategies. Section 4 reports the theoretic performance analysis of these algorithms. In Section 5, the case study aimed to verify the feasibility of the algorithms is presented. Moreover, the computational environment is presented, in particular, main features of the high level software library PETSc are discusses. Experimental results are described in Section 6, while Section 7 concludes the paper.

2. Pipes and Filters pattern oriented design

According to [5], a *pipeline* is an extensible software framework that defines links together one ore more steps (or tasks), running them in sequence to complete a specific process. Each task manipulates input data and delivers output data to other tasks after executing a specific processing function. The design of such mechanism is often implemented in systems that manipulate large volume of digital media such as images or streams of data. Usually, this design is called as *Composite Filter Pattern*.

There are several ways to combine filters. The composition of filters should be robust enough to allow addition of new filters and replacement of existing ones. It should also be flexible enough to support different ways of combining filters together. We chose to combine filters by plugging them together in a pipeline using the *Pipes and Filters pattern design*. Examples of pipes and filters occur in signal processing domains [10], functional programming [14], and distributed systems [4]. In a Pipe and Filter style, each component (filter) has a set of inputs and a set of outputs. A component reads streams of data on its inputs and produces streams of data on its outputs, delivering a complete instance of the result in a standard order. The connectors (pipes) transmit outputs of one filter to inputs of another.

In conclusion, as show in Fig. 1, main components of Pipes and Filters pattern approach are the following:

- the *filters*: data is received via the pipes with which it is connected, a filter can have any number of input pipes and any number of output pipes.
- the *pipe*: it is a directional stream of data, that is usually implemented by a data buffer to store all data, until the next filter has time to process it.
- the *data source*: it can be a static text file, or a keyboard input device, continuously creating new data.

Fig. 1. Pipes and Filters Pattern.

– the *data sink*: it can be another file, a database, or a computer screen.

This scenario becomes more complex if one (or more) of the filters needs also the output provided by two (or more) previous tasks.

Next section describes the deployment of such approach into an algorithm for image sequence analysis and its integration with multicore computing technologies.

3. The pipelined image sequence algorithm

Let us give the following:

Definition 1 [image sequence]: *Let $D \subset \Re$ be a bounded interval. Given $t \in D$, let $z(t) \equiv (x(t), y(t)) \in \Omega$, where $\Omega = \Omega_x \times \Omega_y \subset \Re^2$ is the image plane.*[1] *The image sequence on D is the piecewise smooth function*:

$$I_0 : t \in D \rightarrow z(t) \in \Omega \rightarrow I_0(z(t), t) \equiv I_0(t) \in \Re$$

Let us assume that the interval D consists of FN distinct values, that is:

$$D = \{t_1 < t_2 < \ldots < t_{FN}\}$$

For all $t \in D$ and $r \in N - \{\infty\}$, we assume that on $I_0(t)$ we have to perform r *tasks*, let us say P_1, P_2, …, P_r, to get $I_r(t)$.

More precisely, given:

$$P \equiv \{P_1, P_2, \ldots, P_r\}$$

we get:

$$P : I_0(t) \rightarrow I_r(t)$$

We assume that there is a strong *dependence* between these tasks: each task P_i, $i = 2, \ldots, r$ acts on the output provided by the *previous* one, i.e. P_{i-1}.
This means that if we set:

$$P \equiv P_r \circ P_{r-1} \circ \ldots \circ P_1$$

[1]The image plane Ω should change with the acquisition time t. In practice, it is the same at each t because it refers to the rectangular plane of the image acquisition.

then, it is:

$$P_1 : I_0(t) \rightarrow P_1(I_0(t)) \equiv I_1(t),$$
$$P_2 : P_1(I_0(t)) \rightarrow P_2(P_1(I_0(t))) \equiv I_2(t)$$
$$\ldots\ldots$$
$$P_i : P_{i-1}(I_s(t)) \rightarrow P_i(P_{i-1}(I_s(t)) \equiv I_i(t)$$

$\forall i = 2, \ldots, r, \forall s = 0, \ldots, i-2$.

In general, we have that:

$$I_s(t) = P_s(P_{s-1}(\ldots(P_1(I_0(t)))\ldots)$$

Moreover, we also assume that it may exists a task needing the output provided by two (or more) previous tasks.

A straightforward approach to perform P is based on the execution of the r tasks $P_1, \ldots P_r$ on each frame of the sequence $I_0(t)$. Schematically, this is described by the following algorithm:

```
for k = 1, ..., FN
  for j = 2, ..., r
    I_j(t_k) = P_j(I_{j-1}(t_k))
  endfor j
endfor k
```

Image Sequence Algorithm A

Of course the computing time of algorithm A, increases as the number of frames to process and the overall computation may be too expensive for $FN \rightarrow \infty$.

To address this issue we propose a new approach to perform P which is based on a pipelined computation.

Let us consider the ordered subsets I_i^k made of $r < FN$ subsequent frames of the sequence:

$$I_i^k = \{I_{i-(r-1)}(t_{k-(r-1)}), \ldots, I_{i-1}(t_{k-1}), I_i(t_k)\}$$

$k = r, \ldots, FN, \quad i = 1, \ldots, r$.

At step k of the pipelined algorithm, the r tasks act *concurrently* onto the subset I_i^k made of r consecutive frames, as described in the following way:

The main difference between algorithm A and the pipelined algorithm PA relies on the execution of the r tasks on a single frame and on the subset I_k^r, respectively. While in algorithm A the tasks are necessarily performed in a sequential way, one after the previous, because they act on the same frame, in the pipelined al-

```
for k = r, ..., FN
  to perform the pipelined
  computation on the subset
  I_k^r:
  for i = 1, ..., r,
  P_i acts onto I_{i-1}(t_{k-(i-1)})
  endfor i
endfor k.
```

Pipelined Image Sequence Algorithm PA (middle part)

gorithm PA the tasks P_i act *concurrently* on the ordered subset I_k^r, because they act on different frames. As we will describe later, this approach reduces the overall execution time by a factor depending on the size of I_k^r.

Let us now describe the first $r-1$ steps of the pipelined algorithm PA. As expected, due to the fact that for $k = 1, \ldots, r-1$ the number of available frames is less than the number of tasks to perform on these frames ($k < r$), only a part of the r tasks P_i $i = 1, \ldots, r$, may be performed. Indeed, first $r-1$ steps represent the *start-up* time of the pipelined algorithm. More precisely, it holds:

```
for k = 1, ..., r - 1
  for i = 1, ..., k
  P_i acts onto I_{i-1}(t_{k-i})
  endfor i
endfor k
```

Start up of Algorithm PA

At step $k = FN$ of the pipelined algorithm PA, P_r operates onto the frame corresponding to the acquisition time $t_{FN-(r-1)}$.

This means that it remains to still process $r-1$ frames, i.e. the *final-stage* of the pipelined algorithm. This is done within further $r-1$ steps, as described in the following:

```
for k = 1, ..., r - 1,
  P_r acts on I_{r-1}(t_{FN-(r-(k+1))})
endfor k.
```

Final stage of Algorithm PA

Hence, the pipelined computation consists of three main stages: the start-up, made of $r-1$ steps, the central computation, made of $FN - r + 1$ steps, and the final stage, made of $r-1$ steps.

In order to reduce synchronization overheads due to loads imbalance and communications, each task is further parallelized and executed on multicore processors. Taking into account hierarchical parallelism of system architecture of multicore multiprocessors we introduce three parallelization strategies:

1. in the first strategy the execution of each task is distributed among processing cores employing a fine-grained data parallelism (i.e. parallelism is introduced the lowest level of PA). We refer to this strategy as: data parallel pipelined algorithm (data PPA),
2. in the second strategy execution of each task is assigned to a single processing core introducing concurrency at a coarser level of PA. We refer to this algorithm as: functional parallel pipelined algorithm (functional PPA),
3. the last one combines the previous two approaches. The pipelined algorithm PA is first distributed among group of multicore. Then, each task of PA is parallelized within each group. We refer to this strategy as: hybrid parallel pipelined algorithm (hybrid PPA).

In Fig. 2 we show data flow of the hybrid PPA algorithm using 7 computing processors concurrently performing 3 tasks: we note that, first task is assigned to Procs. 0 and 1 (group 1), the second task is assigned to Procs. 2 and 3 (group 2) and the third task is assigned to Procs. 4, 5 and 6 (group 3).

The *hybrid PPA* approach balances the computational load of each task as for the data parallelism improving performance gain of the functional PPA.

To indicate data PPA, functional PPA and hybrid PPA algorithms the following notation is used. Let r be the number of tasks P_i then:

- **data PPA** – $\underbrace{(nproc, nproc, \ldots, nproc)}_{r}$: each tasks P_i is distributed among $nproc$ processing core.
- **functional PPA** – $\underbrace{(1, 1, \ldots, 1)}_{r}$: each task P_i is assigned to one processing core.
- **hybrid PPA** – $(nproc_1, nproc_2, \ldots, nproc_r)$: each task P_i is distributed among $nproc_i$ processing core.

In addition, observe that main difference among the third strategy and the first two is the core assignment: while in the first two strategies cores are statically assigned, in the last one they are **dynamically assigned**.

More precisely, to determine the number of core to assign to execute task P_i, $i = 1 \ldots, r$, that is $nproc_i$, *dynamic assignment* is based on a preprocessing stage on the execution time of tasks. This may be done using a theoretical performance analysis, or using a database collecting this information or, finally, at run-

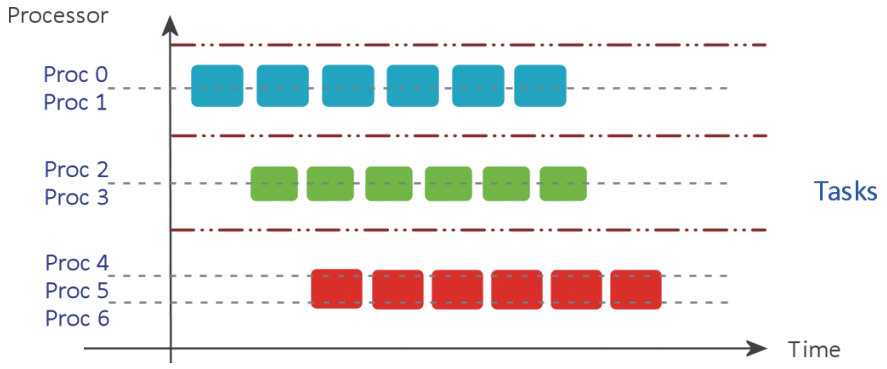

Fig. 2. Data flow of the hybrid PPA algorithm on nproc = 7 processors for an image sequence of length FN = 6 involving 3 tasks. Note that, first task is assigned to Procs. 0 and 1 (group 1), the second task is assigned to Procs. 2 and 3 (group 2) and the third task is assigned to Procs. 4, 5 and 6 (group 3).

time monitoring the total execution time (computations plus communication time) of P_i.

Let

$$T_{P_i}(nproc_j), i = 1, \ldots, r$$

be the execution time of parallel execution of P_i on $nproc_j$ core, taking into account that the number of available cores is limited, i.e.:

$$nproc_i \leqslant nproc_{tot} \quad \forall i \leqslant r$$

and that the computing machine that we use is dedicated, the number of cores ($nproc_i$) to assign to P_i is such that it provides the minimum total execution time of P_i, as described below:

for $i = 1, \ldots, r$,
$nproc_i =$
$= \min_{\{j=1,\ldots(nproc_{tot} - \sum_{1 \leqslant k \leqslant j-1} nproc_k)\}}$
$(T_{P_j}(nproc_j))$
endfor i.

Dynamic core assignment

Note that, by requiring

$$j = 1, \ldots (nproc_{tot} - \sum_{1 \leqslant k \leqslant j-1} nproc_k)$$

we guarantee that

$$\sum_{1 \leqslant i \leqslant r} nproc_i = nproc_{tot}$$

4. Performance analysis

Given the image sequence of length FN, we make use of the quantities:

- $T_{P_i}, i = 1, \ldots, r$
  execution time needed to apply P_i,
- $T_{max} = max_{i=1,\ldots,r}\{T_{P_i}\}$,
- $T(FN) = FN \cdot \sum_{i=1}^{r} T_{P_i}$
  execution time required to execute the r tasks P_i to a FN-sequence,
- $T_{P_i}(nproc_j), i = 1, \ldots, r$
  execution time of parallel execution of P_i on $nproc_j$ core,
- $S^{P_i}_{nproc}$
  speed up of task P_i on $nproc$ core,
- T_{PA}
  execution time of the middle part of PA,
- T^{fin}_{PA}:
  execution time of the final stage of the PA algorithm,
- T^{in}_{PA}:
  execution time of *start up* of the PA algorithm,
- $T^{fun}_{PPA}(FN) = T^{in}_{PA} + T^{fin}_{PA} + \sum_{k=r}^{FN} T_{max}$
  execution time of functional PPA algorithm (without I/O),
- $T^{data}_{PPA}(FN, nproc)$
  execution time of data PPA algorithm using $nproc$ cores to concurrently perform each task,
- $S^{data}_{nproc}(FN) = \frac{T(FN)}{T^{data}_{PPA}(FN, nproc)}$
  speed up of data PPA algorithm using $nproc$ cores to process each task,
- $S^{fun}(FN) = \frac{T(FN)}{T^{fun}_{PPA}(FN)}$
  speed up of functional PPA algorithm,
- $S^{hyb}_{nproc}(FN) = \frac{T(FN)}{max_{i=1,\ldots,r} T_{P_i}(nproc_i)}$
  speed up of hybrid PPA algorithm on $nproc$ processing core.

Proposition 1:
Data PPA performance is at most equals to the highest performance among the r tasks P_i.

It holds that:

$$S^{data}_{nproc}(FN) \leqslant S^{max}_{nproc}$$

$$= max\{S^{P_i}_{nproc} \ , \quad i = 1, \ldots, r\}$$

Proof:

$$S^{data}_{nproc}(FN) = \frac{T(FN)}{T^{data}_{PPA}(FN, nproc)}$$

$$= \frac{FN \cdot \sum_{i=1}^{r} T_{P_i}}{FN \cdot \sum_{i=1}^{r} T_{P_i}(nprocs)}$$

$$= \frac{FN \cdot \sum_{i=1}^{r} T_{P_i}}{FN \cdot \sum_{i=1}^{r} \frac{T_{P_i}}{S^{P_i}_{nproc_i}}}$$

$$\leqslant \frac{FN \cdot \sum_{i=1}^{r} T_{P_i}}{FN \cdot \sum_{i=1}^{r} \frac{T_{P_i}}{S^{max}_{nproc}}} = S^{max}_{nproc} \qquad \diamond$$

Then, the highest performance gain we expect from data PPA equals the maximum number of computing cores employed for the parallel execution of each task. Moreover, as it is usual for parallel algorithms implementing data parallelism, as the core number grows, overheads predominate causing speed-up degradation. This means that, in case of a fixed-size application, due to the limited amount of available parallelism, there exists an *optimal* number of processing core and each additional core contributes slightly less or do not.

Concerning second strategy, it can be proved that:

Proposition 2 (**functional PPA**):
It is:

$$\lim_{FN\to\infty} S^{fun}(FN) = \lim_{FN\to\infty} \frac{T(FN)}{T^{fun}_{PPA}(FN)}$$

$$= \frac{\sum_{i=1}^{r} T_{P_i}}{T_{max}}$$

Proof:

$$T(FN) = FN \sum_{i=1}^{r} T_{P_i}$$

and

$$T^{fun}_{PPA}(FN) = T^{in}_{PA} + T^{fin}_{PA} + \sum_{k=r}^{FN} T_{max}$$

$$= T^{in}_{PA} + T^{fin}_{PA} + (FN + r - 1) \cdot T_{max}$$

It holds:

$$\lim_{FN\to\infty} \frac{T(FN)}{T^{fun}_{PPA}(FN)}$$

$$= \frac{\sum_{i=1}^{r} T_{P_i}}{T_{max}} \qquad \diamond$$

Therefore, as the size of sequence increases, functional PPA performance depends on the computational cost of tasks P_i. In particular, it depends on a suitable load balance among tasks.

Then, it holds:

Corollary 1 (**functional PPA**):
Assume that $\forall\ i = 1, \ldots, r$, $T_{P_i} = T_{max}$, *that is execution time of tasks* P_i *is the same, then:*

$$\lim_{FN\to\infty} S^{fun}(FN) = r$$

Proof:

$$T(FN) = r \cdot FN \cdot T_{max}$$

then

$$\lim_{FN\to\infty} S^{fun}(FN)$$

$$= \lim_{FN\to\infty} \frac{T(FN)}{T^{fun}_{PPA}(FN)}$$

$$= \frac{r \cdot T_{max}}{T_{max}} = r \quad \diamond$$

This means that, as the size of the sequence increases, *the highest performance gain that we may expect of functional PPA is equal to* r*, the number of tasks,* and this occurs if all tasks require about the same execution time. On the contrary, the lowest performance occurs if the tasks are not balanced:

Corollary 2 (**functional PPA**):
Assume that there exist only one among the r *tasks, let us denote* $P_{\widetilde{i}}$*, whose execution time is*

$$T_{P\widetilde{i}} \geqslant \sum_{i \neq \widetilde{i}} T_{Pi}$$

then

$$\lim_{FN\to\infty} S^{fun}(FN) \leqslant 2$$

Proof:

$$\sum_{i=1}^{r} T_{P_i} = T_{P\widetilde{i}} + \sum_{i \neq \widetilde{i}} T_{Pi} \leqslant T_{P\widetilde{i}} + T_{P\widetilde{i}}$$

$$= 2 \cdot T_{P\widetilde{i}}$$

It follows that:

$$\lim_{FN\to\infty} \frac{T(FN)}{T^{fun}_{PPA}(FN)}$$

$$\leqslant \frac{2T_{P\widetilde{i}}}{T_{P\widetilde{i}}} \leqslant 2 \quad \diamond$$

Such results suggest us to combine data PPA and functional PPA in order to take advantages of both the first strategy and the second one. Indeed, using functional parallelism we overlap the execution of tasks P_i, while introducing data parallelism inside their computations, we balance their computing time.

Indeed, it can be proved that:

Proposition 3 (**hybrid PPA**):

Let $1 \leqslant \widehat{i} \leqslant r$ be such that $T_{P_{\widehat{i}}}(nproc_{\widehat{i}})$ is the maximum execution time of the r parallel tasks P_i, i.e.

$$T_{P_{\widehat{i}}}(nproc_{\widehat{i}}) = \max_{i=1,\ldots,r} T_{P_i}(nproc_i) = \frac{T_{P_{\widehat{i}}}}{S^{P_{\widehat{i}}}_{nproc_{\widehat{i}}}}$$

then it is:

$$S^{hyb}_{nproc} = \frac{\sum_{i=1}^{r} T_{P_i}}{T_{P_{\widehat{i}}}(nproc_{\widehat{i}})} \leqslant nproc_{\widehat{i}} \cdot S^{fun}(FN)$$

Proof:

$$S^{hyb}_{nproc}(FN) = \frac{\sum_{i=1}^{r} T_{P_i}}{T_{P_{\widehat{i}}}(nproc_{\widehat{i}})}$$

$$\leqslant \frac{\sum_{i=1}^{r} T_{P_i}}{\frac{T_{P_{\widehat{i}}}}{nproc_{\widehat{i}}}} = nproc_{\widehat{i}} \frac{\sum_{i=1}^{r} T_{P_i}}{T_{P_{\widehat{i}}}}$$

$$= nproc_{\widehat{i}} \cdot S^{fun}(FN) \quad \diamond$$

Following result is a straightforward consequence of Corollary 2 and Proposition 3.

Corollary 3 (**hybrid PPA**):

Assume that $\forall\ i = 1, \ldots, r$, *the execution time of parallel tasks* P_i *on nproc cores is the same, that is*

$$T_{P_i}(nproc_i) \equiv T_{max}, i = 1, \ldots, r$$

then, if $nproc = max_{i,\ldots r}(nproc_i)$:

$$\lim_{FN \to \infty} S^{hyb}_{nproc}(FN) = nproc \cdot r$$

Using the hybrid PPA algorithm, the highest performance gain we expect with respect to Algorithm A, as the size of the image sequence increases, occurs when tasks are balanced and this *equals to nproc times the highest performance gain we expect when employing the functional PPA algorithm, that is* r. In other words, the hybrid PPA actually may provide a significant improvement with respect to the other two strategies. Such results are confirmed by the experiments we carried out and described in Section 5.

5. A case study

As already said, the PA algorithm may be used for any applications involving a set of incremental transformations (i.e. ordered computations) onto a finite stream of data. In particular, here we show as case study the segmentation of medical structures from degraded ultrasound images [6,7]. We focus on detection and delineation of the expansion of the ventricle chamber at each frame of ultrasound image sequences. Besides the presence of the speckle noise that affects ultrasound images, the problem is to detect and delineate the expansion of the left ventricle (LV) chamber at each frame of the sequence (see Fig. 3).

This application consists of $r = 3$ tasks P_i, $i = 1,2,3$. These are: P_1 the despeckle plus contrast enhancement, P_2 is the recovery of missing edges via the optic flow computation, and P_3 which is the LV segmentation. Finally, $I_3 \equiv I_3(C(t))$, that is the image brightness of the LV contour $C(t)$.

Next sections briefly review the mathematical model of despeckle, of the segmentation task, and of the optic flow computation. Moreover we describe the dependence among these tasks.

5.1. The de-speckle task P_1

Following [3], $\forall t \in D$, consider the PDE:[2]

$$\frac{\partial I(\tau, t)}{\partial \tau} = \nabla \left(D_{if}(|\nabla I(\tau, t)|) |\nabla I(\tau, t)| \right)$$

where

$$\tau \in [0, T], P(t) \in \Omega, t \in D$$

with zero Neummann boundary conditions:

$$\frac{\partial I(\tau, t)}{\partial n} = 0,$$

where

$$\tau \in [0, T], P(t) \in \partial\Omega, t \in D$$

and $I_0(t) = I_S(t)$ as initial condition ($\tau = 0$).

The *diffusion matrix* D_{if} is defined as:

$$D_{if} = (w_1 w_2) \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix} \begin{pmatrix} w_1^T \\ w_2^T \end{pmatrix},$$

[2] Here we use the multiscale notation for the image function $I(t)$ by considering the set of functions $I(\tau, t)$, depending on the scale τ, each one representing a local approximation of $I(t)$ [1].

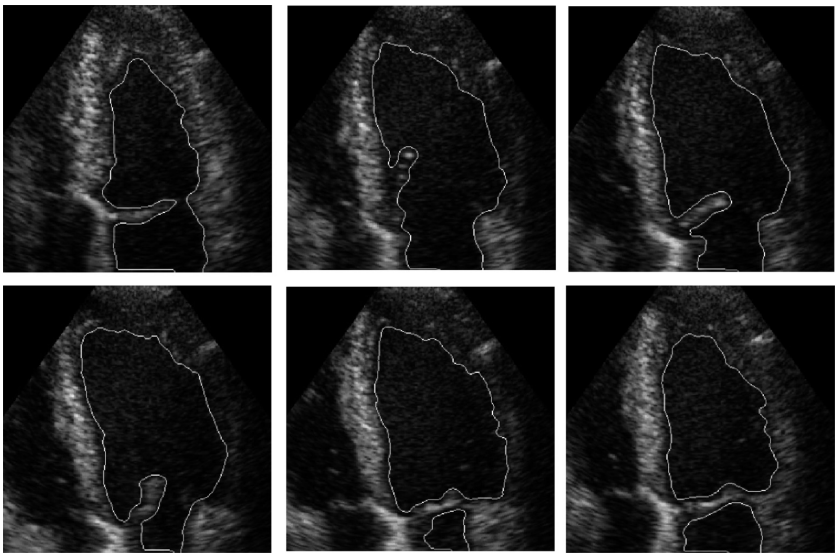

Fig. 3. From upper left to bottom right the LVA chamber segmentation on frames 3, 9, 15, 19, 22, 25.

$$\lambda_1 = \begin{cases} \alpha\left(1 - \frac{(\mu_1-\mu_2)^2}{s^2}\right), & (\mu_1-\mu_2)^2 \\ & \leqslant s^2 \\ 0, & otherwise \end{cases}$$

and $\lambda_2 = \alpha$ (s and α are constants).
The diffusion reaction matrix is obtained by the eigenvectors of the structure matrix:

$$J = G_\sigma \star \left(\nabla I \nabla I^T\right)$$

where G_σ is the Gaussian function and $\star$ is the convolution operator.

5.2. The segmentation task P_3

At each $t \in D$, the segmentation function:

$$u : t \in D \longrightarrow (P(t), t) \in \Omega \times D$$
$$\rightarrow u(t) \equiv u(P(t), t, \tau) \in \Re^2$$

is obtained by solving the PDE:

$$\frac{\partial u\,(t)}{\partial \tau} = \sqrt{\epsilon + |\nabla u\,(t)\,|^2}\,\nabla \cdot \left(g(|\nabla G_\sigma * \widetilde{I}(t)|)\frac{\nabla u\,(t)}{\sqrt{\epsilon + |\nabla u\,(t)\,|^2}}\right)$$

$\varepsilon > 0$ is the regularization parameter, $g(s) = 1/(1 + Ks^2)(K > 0)$ is the Perona-Malik edge-indicator function, G_σ is the Gaussian function, $*$ is the convolution operator. In our experiments we select $K = 0.01$ and $\varepsilon = 0.001$. The PDE is equipped with Dirichlet boundary conditions as follows:

Definition [initial condition]: *Let $t = t_0$ be the acquisition time of first frame. Let $CY_i, i = 1, \ldots, n$ be a set of cycles, located inside the interesting region, each one having center $c_i = (x_i\,(t_0)\,, y_i\,(t_0))$, then, given CY_i, $i = 1, \ldots, n$ we define:*

$$u_0\,(t_0) = \max_{i=1,n} \omega_i\,(x(t_0), y(t_0))$$

where:

$$\omega_i\,(x(t_0), y(t_0)) = \omega_i\,(x_0, y_0))$$
$$= \begin{cases} \frac{1}{|(x_0,y_0))-(x_0^i,y_0^i)|+1} & if (x_0, y_0) \in CY_i \\ \frac{1}{R+1} & if (x_0, y_0) \in \Omega - CY_i \end{cases}$$

In subsequent frames, u_0 is obtained in automatic way using functions ω_i previously introduced. More precisely, $\forall t > t_0$, and $\forall i = 1, \ldots, n$ to calculate u_0 are considered only those ω_i such that:

$$\forall (x(t), y(t)) \in CY_i, \quad I(x(t), y(t)) \leqslant H$$

where:

$$H = \alpha |max_\Omega I(t) - min_\Omega I(t)| + min_\Omega I(t), \quad \alpha \in \Re.$$

The key feature of the segmentation model that we employ for LV border detection is the definition of $\widetilde{I}$ in (2) inside the edge indicator function g. We integrate the image to segment with the new position of the contour computed at previous time. The trajectory of this curve is provided by the **motion** of the LV border.

5.3. The motion computation task P_2

Let $t_1, t_2 \in D$, where $t_2 > t_1$, and $\Delta t = t_2 - t_1$, be two consecutive frames of the sequence. If $\Gamma(t_1)$, is the LV border obtained at t_1, we consider

$$\widetilde{I}(t_2) = I(t_2) + \Gamma_{prev}(t_2)$$

where:

$$\Gamma_{prev}(t_2) = \{(x_{prev}(t_2), y_{prev}(t_2)) : (x(t_1), y(t_1)) \in \Gamma(t_1)\}$$

To define $\Gamma_{prev}(t_2)$, we first compute Γ at time t_1 then, using the motion trajectory, we predict the position of the LV border at a subsequent time as follows:

$$\begin{aligned}&(x_{prev}(t_2), y_{prev}(t_2))\\ &= (x(t_1) + v_1(t_1)\Delta t, y(t_1) + v_2(t_1)\Delta t),\\ &\forall\,(x(t_1), y(t_1)) \in \Gamma(t_1),\end{aligned}$$

where $v_1(t_1)\Delta t$, $v_2(t_1)\Delta t$ are obtained by the *motion trajectory* of $P(t)$.

Finally, we integrate this information inside the frame to segment.

[Motion trajectory]: *The motion trajectory of a point $P(t) = (x(t), y(t)) \in \Omega$ is the line (or the arc of line)* **L***, defined by the successive positions of $P(t)$, as t moves from t_1 towards t_2. The parametric equation for L is*:

$$L : \begin{cases} \Delta x = x(t_2) - x(t_1) = \Delta t \cdot v_1(t_2) \\ \Delta y = y(t_2) - y(t_1) = \Delta t \cdot v_2(t_2) \end{cases},$$

where $(v_1(t), v_2(t)) = (\frac{d}{dt}x(t), \frac{d}{dt}y(t))$ *are the components of the* **motion field***, at $P(t) \in \Omega$.*

Following [20], we find the apparent motion field (the so-called *optical flow*), by imposing that the spatial brightness gradient does not change along motion trajectory.

We get a system of two non linear parabolic (diffusion-reaction) PDE equations:

$$\begin{cases} \frac{\partial u}{\partial \tau} = \alpha \cdot div[\phi'\left(\nabla u \nabla u^T + \nabla v \nabla v^T\right)\nabla u] + \\ -2\left[I_{xx}u + I_{yx}v + I_{tx}\right] \cdot I_{xx} + \\ -2[I_{xy}u + I_{yy}v + I_{ty}] \cdot I_{xy} \\ \frac{\partial v}{\partial \tau} = \alpha \cdot div\left[\phi'\left(\nabla u \nabla u^T + \nabla v \nabla v^T\right)\nabla v\right] + \\ -2\left[I_{xx}u + I_{yx}v + I_{tx}\right] \cdot I_{yx} + \\ -2[I_{xy}u + I_{yy}v + I_{ty}] \cdot I_{xy} \end{cases} \quad (1)$$

with zero initial conditions and Dirichlet boundary conditions, $\alpha > 0$ the regularization parameter and

$$\phi'\left(s^2\right) = \varepsilon + (1-\varepsilon)/2\sqrt{1 + s^2/\lambda^2}$$

as edge-indicator function.

5.4. The computing environment and PETSc

The PDEs were discretized using the *semi implicit* scheme respect to the scale derivatives. This choice leads to inconditionally stable numerical schemes. For the spatial discretization, we use finite differences for despeckling and optic flow models and the (complementary) finite volumes for the segmentation problem. At each scale step we have to solve a linear system. We use the Additive Operator Scheme (AOS) for depeckling [21]. Regarding optic flow and segmentation models, we use GMRES iterative method equipped with the algebraic recursive multilevel preconditioner implemented in the BoomerAMG library [11].

The software has been developed using the high level software library PETSc (Portable Extensible Toolkit for Scientific Computations) [15]. PETSc provides a suite of data structures and routines as building blocks for the implementation of large-scale codes to be used in scientific applications modeled by partial differential equation. PETSc is flexible: its modules, that can be used in application codes written in Fortran, C and C++, are developed by means of object-oriented programming techniques.

The library has a hierarchical structure: it relies on standard basic computational (BLAS, LAPACK) and communication (MPI) kernels, and provides mechanism needed to write parallel application codes (see Fig. 4). PETSc transparently handles the moving of data between processes without requiring the user to call any data transfer function. This includes handling parallel data layouts, communicating ghost points, gather, scatter and broadcast operations. Such operations are optimized to minimize synchronization overheads.

6. Experimental results

Experiments were performed using 4 blade Dell PowerEdge M6000 each made of 2 processors quad core (1 blade/node = 2 processors = 8 cores) Intel Xeon E5410@ 2.33GHz (64 bit) connected by the high performance network InfiniBand. We carried out many experiments aimed at monitoring both the performance of each task and of the pipelined algorithm. We report here main results.

The starting point is a sequence of $FN = 26$ frames of size 300×300 (1 cardiac cycle). Then, sequences of length 51, 101, 201, 401, 801 are considered.

Note that for the case study that we are considering, tasks are not balanced. Figure 5 shows the execution

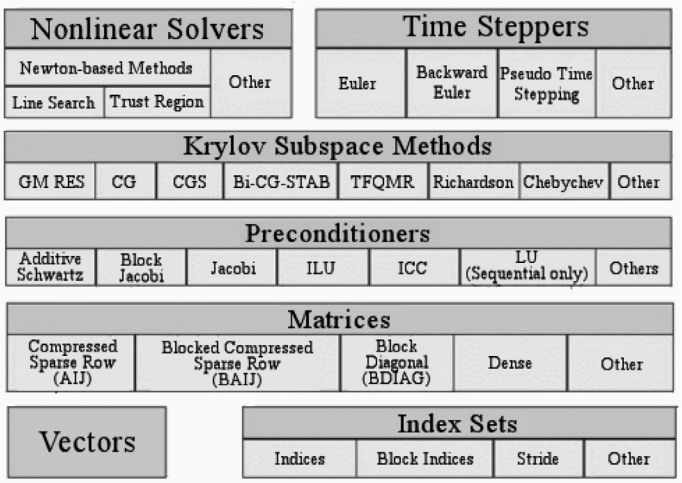

Fig. 4. Hierarchical structure of the software library PETSc.

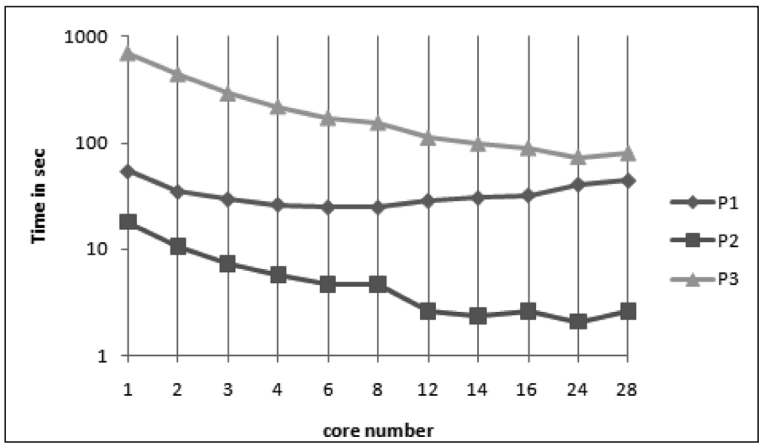

Fig. 5. Execution Time (in seconds and in semi logarithmic scale) of tasks P_i versus the core number. Triangle denotes task P_3 (segmentation), Box denotes task P_2 (optic flow), Circle denotes task P_1 (despeckle). Sequence length FN = 26.

time of tasks $P_i, i = 1, 2, 3$ versus the core number. Optic flow computation task is the cheapest (it needs about *40 secs* on 1 core and the highest performance is reached using 8 cores with *24.6 secs*) and the segmentation is the most time consuming (it requires more than *700 secs* on 1 core and *78 secs* with 24 cores). Therefore, we expect both functional PPA and Data PPA do not perform good. Hybrid PPA algorithm, instead, may perform better than the previous ones by suitably parallelizing each task in order to balance their computational loads. In the following, we report the performance of Algorithm A, functional PPA, data PPA and hybrid PPA algorithms, respectively, measured in terms of execution time and speed up.

Moreover, to verify if the concurrency helps to hide the latency and the overheads of communications we measure the execution frame/rate employed by the algorithm, which is a performance metric usually employed for any algorithms devoted to process data sequences. That is we introduce the *throughput* defined as follows:

Definition: *The throughput is:*

$$Th = \frac{frame\,number}{time} = \frac{FN}{secs}$$

From experimental results we get:

Algorithm A:
When $FN = 26$, $T(FN) = 773.92\,sec$, while the frame rate/sec is:

$$Th = \frac{FN}{T_{sec}} = 0.0336$$

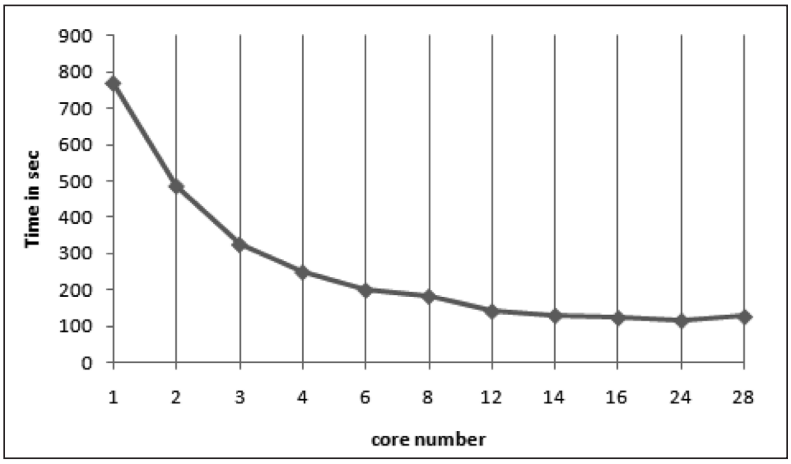

Fig. 6. Execution time of Data PPA algorithm versus the core number. Sequence length FN = 26.

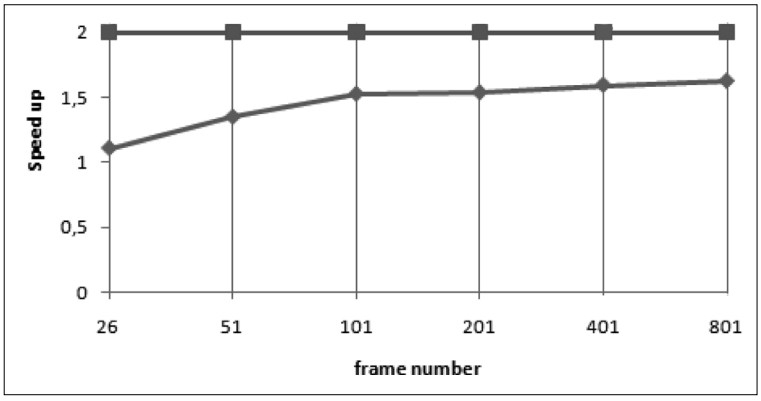

Fig. 7. Speed up of functional PPA algorithm on (1,1,1) cores as a function of FN.

Functional PPA:

In agreement with Corollary 2 we expect a low speed up. As shown in Fig. 6 we get

$$S^{fun}(FN) \leqslant 2$$

for $FN = 26$ up to $FN = 801$.

Data PPA:

Figure 6 reports the execution time of data PPA algorithm (for task P_3) versus the core number. Using $nproc = 24$ cores we get the minimum time, i.e.

$$T^{data}_{PPA}(FN) = 122.4472\, sec,$$
$$S^{data}_{24}(FN) = 6.3$$

The frame rate is

$$Th = \frac{FN}{T_{sec}} = 0.212$$

while the performance gain with respect to Algorithm A is of 84,17%.

Hybrid PPA:

As already noted, in this application tasks are quite unbalanced, both in terms of their computational load and in terms of data communications. In particular, each task exhibits a communication overhead (the ratio of communication time over computation time) which is quite different from the others.

In the hybrid PPA, by introducing concurrency both on computations and on communications, we get the right total execution time balancing by distributing the parallel execution of each task over a different number of cores.

The number of cores, are determined according to the dynamic assignment, as described in Section 3. We perform tasks monitoring at run-time and the subsequent core assignment follows this ordering: P_3, P_2 and P_1.

Dynamic assignments gives

$$nproc_3(P_3) = 24,\ nproc_2(P_2) = 6,$$

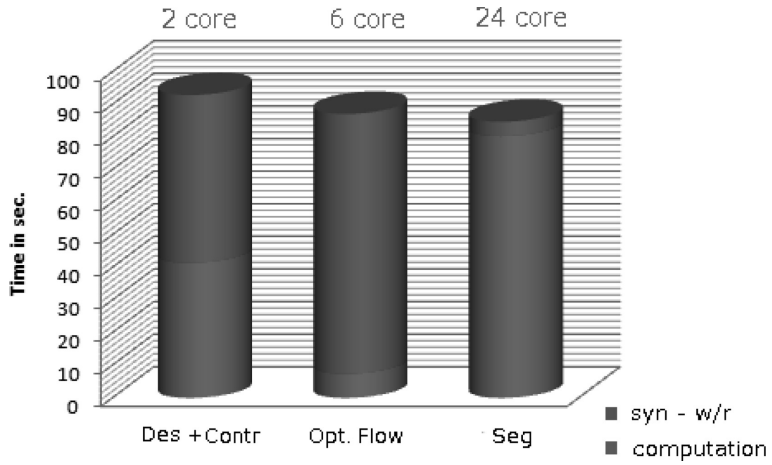

Fig. 8. Total execution time (computations plus communications) in seconds of tasks P_1 P_2 P_3, using $(2, 6, 24)$ cores respectively. Sequence length FN $= 26$.

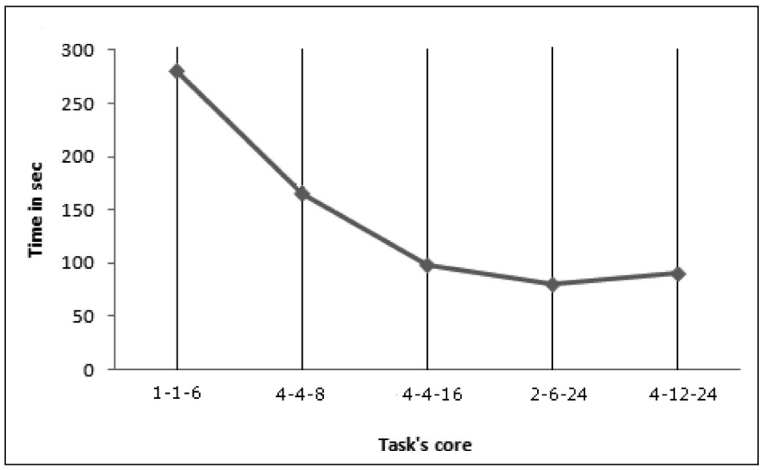

Fig. 9. Execution time (in seconds) of hybrid PPA algorithm versus the node number (1 node = 8 cores). On the x-axis, for each node the three numbers indicate the core assigned to each task P_i, $i = 1,2,3$. Sequence length FN $= 26$.

$$nproc_1(P_1) = 2$$

Feedback of the right load balancing among tasks was shown in Fig. 8: using 2 cores for task P_1, 6 cores for task P_2 and 24 cores for P_3 we get about the same total execution time for the three tasks. It is worth to note that even though P_2 is cheaper than P_1 in terms of floating point computations, for reducing the total execution time required by P_2, because P_2 needs to perform more communications than P_1, the total execution time of P_2 is greater than P_1 and the optimal number of cores is greater than that used for P_1.

Then, we get

$$T^{hyb}_{PPA}(FN) = 84.3395\ sec$$

with a frame rate of $Th = 0.314$ frame/sec. Speed up is

$$S_{hyb}(FN) = 9.1$$

and performance gain with respect to Algorithm A is of 89.29%.

Figure 9 shows the execution time of hybrid PPA algorithm as the core number increases. Observe that the sum of the cores employed for measuring the performance of the hybrid PPA algorithm is a multiple of 8 because communication network on a node (= 8 cores) of the parallel machine that we are using is optimized. On the x-axis the three numbers among parenthesis refer to the cores employed for parallelizing task P_1, P_2 and P_3, respectively. The minimum time is reached at (2,6,24). By comparing Figs 5 and 9, execution time of hybrid PPA algorithm at $FN = 26$ on (2,6,24) cores equals the execution time of task P_3 (segmen-

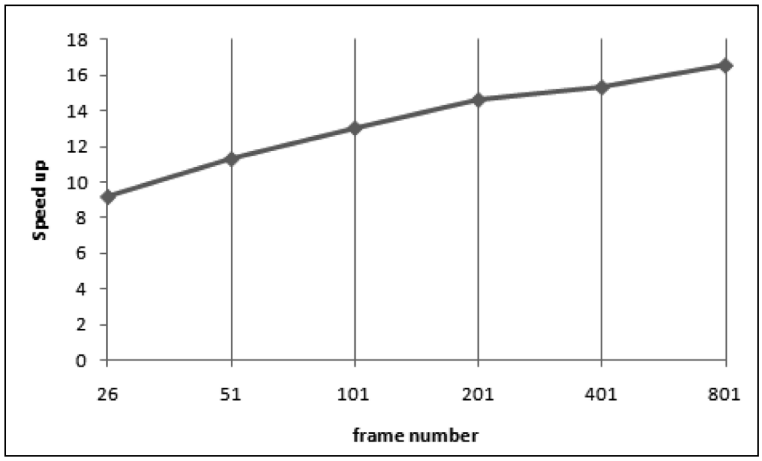

Fig. 10. Speed up of hybrid PPA algorithm on (2,6,24) cores versus frame number FN.

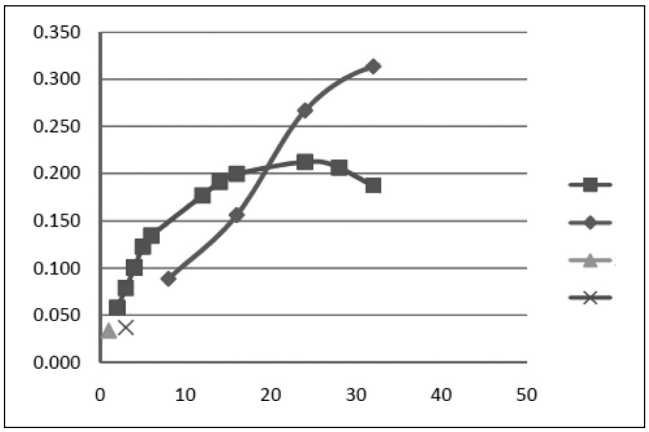

Fig. 11. Throughput versus the core number. On the x-axis we show the core number, while on the y-axis is it shown Th = frame/rate. Triangle denote throughput of Algorithm A, Box denote the throughput of data PPA, Star denotes the throughput of functional PPA and circle denotes the throughput of hybrid PPA. Sequence length FN = 26.

tation task) on 24 cores, confirming that, as expected from any pipelined computations, in the hybrid PPA algorithm tasks actually overlap.

To validate the performance gain of this algorithm as a function of FN we also compute the performance gain of this algorithm as FN grows. Observe that, in agreement with Corollary 3, we have

$$T_{P\widehat{i}}(nproc_{\widehat{i}}) = \max_{i=1,\ldots,r} T_{P_i}(nproc_i) = T_{P_3}(24)$$

and $S_{hyb} \leqslant 3 \cdot 24$. Figure 10 shows the speed up line as function of FN.

7. Conclusions

We introduce a pipelined algorithm for performing a set of ordered tasks on a finite stream of data. Main idea underlying this algorithm is to perform the tasks by *overlapping the execution of the tasks via pipelining, along the data sequence and, by concurrently performing the operations of each task.*

Concurrency is introduced at two different levels and with two different granularities: the coarse-grained parallelism to perform independent tasks and the fine-grained parallelism within the execution of a task. We analyze the performance of three parallelization strategies concluding that the hybrid strategy, which combines coarse and fine-grained parallelism, that is data and functional parallelism of the pipelined algorithm. We conclude that as data length increases speed up scales as $nproc \times r$ where $nproc$ is the maximum number of core assigned to tasks and r is the number of tasks.

As case study, the segmentation of ultrasound sequence is considered. Performance of this approach is discussed in terms of execution time, speed up and also using the number of frame per second (throughput) that measures the rate at which data are processed. Using this approach, the throughput scales of about 90% with respect to that of any sequential computation (see Fig. 11 for a sequence of $FN = 26$ frames).

References

[1] L. Alvarez, P.L. Lions and J.M. Morel, Image Selective Smoothing and Edge Detection by nonlinear Diffusion II, *SIAM J Numerical Analysis* **29** (1992), 845–866.

[2] D. Anastasia and Y. Andreopoulos, Software designs of image processing tasks with incremental refinement of computation, *IEEE Trans Image Process* **19**(8) (2010), 2099–2114.

[3] K.Z. Bbbd-Elmoniem, A.M. Youssef and Y.M Kadah, Real-Time Speckle Reduction and Coherence Enhancement in Ultrasound Imaging via Nonlinear Anisotropic Diffusion, *IEEE Trans on Biomedical Engineering* **49**(9) (2002), 997–1012.

[4] M.R. Barbacci, C.B. Weinstock and J.M. Wing, Programming at the processor-memory-switch level. In Proceedings of the 10th International Conference on Software Engineering, (Singapore), IEEE Computer Society Press, April 1988, pp. 19–28.

[5] F. Buschmann, K. Henney and D.C. Schmidt, Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing, (2007) Kindle Edition.

[6] D. Casaburi, L. D'Amore, L. Marcellino and A. Murli, Real Time ultrasound image sequence segmentation on multicores, Parallel Computing: From Multicores and GPU's to Petascale, *Advances in Parallel Computing* B. Chapman, F. Desprez, G.R. Joubert, A. Lichnewsky, F. Peters and T. Priol, eds, (Vol. 19), 2010, pp. 185–192.

[7] D. Casaburi, L. D'Amore, L. Marcellino and A. Murli, Ultrasound Image Segmentation via Motion Estimation, proceedings of ENUMATH 2009, June 29–July 3, Uppsala, Sweden, LNCS, Springer–Verlag, 2010.

[8] W.B. Chen and C. Zhang, A Hybrid Framework for Protein Sequences Clustering and Classification Using Functional Signature Motif Information, *Integrated Computer-Aided Engineering* **16**(4) (2009), 353–365.

[9] L. D'Amore, L. Marcellino and A. Murli, Image sequence inpainting: a numerical approach for blotch detection and removal via motion estimation, *JCAM* **198**(2) (2007), 396–413.

[10] N. Delisle and D. Garlan, Applying formal specification to industrial problems: A specification of an oscilloscope, IEEE Software, September 1990.

[11] V.E. Henson and U.M. Yang, BoomerAMG: A parallel algebraic multigrid solver and preconditioner, *Applied Numerical Mathemetics* **41**(1) (2002), 144–177.

[12] Y. Gousseau and J.M. Morel, Are natural images of bounded variation, *SIAM Journal of Mathematical Analysis* **33** (2001), 634–648.

[13] C. Isaacson, Software Pipelines and SOA: Releasing the Power of Multi-Core Processing, 2008, Kindle Edition.

[14] G. Kahn, The semantics of a simple language for parallel programming, Information Processing, 1974.

[15] http://www.msc.anl.gov/petsc/petsc-as.

[16] W. Mahdi, S. Werda and A.B. Hamadou, A Hybrid Approach for Automatic Lip Localization and Viseme Classification to Enhance Visual Speech Recognition, *Integrated Computer-Aided Engineering* **15**(3) (2008), 253–266.

[17] M. Rizzi, M. D'Aloia and B. Castagnolo, Computer Aided Detection of Microcalcifications in Digital Mammograms Adopting a Wavelet Decomposition, *Integrated Computer-Aided Engineering* **16**(2) (2009), 91–103.

[18] A. Sánchez, C.A.B. Mello, P.D. Suárez and A. Lopes, Automatic line and word segmentation applied to densely line-skewed historical handwritten document images, *Integrated Computer Aided Engineering* **18**(2) (2011), 125–142.

[19] A. Sarti and G. Citti, Subjective surface and Riemannian mean curvature flow graphs, *Acta Math Univ Comeniamae* **70**(1) (2001), 85–104.

[20] J. Weickert and C. Shnorr, A Theoretical Framework for Convex Regularizeds in PDE-Based Computation of Image Motion, TR 13/2000, Computer Science Series, (2000).

[21] J. Weickert and B.M. ter Haar Romeny, Efficient and Reliable Schemes for Nonlinear Diffuzion Filtering, *IEEE Trans on Image Processing* **7**(3) (1998), 398–409.