A DATA AND TASK PARALLEL IMAGE PROCESSING ENVIRONMENT
UN MEDIU DE PRELUCRARE DE IMAGINI BAZAT PE PARALELISM LA NIVEL DE DATE ŞI DE PROCESE

Cristina Nicolescu [1], Pieter Jonker [2]

Rezumat: *Articolul prezintă un mediu de prelucrare de imagini de nivel scăzut, bazat pe paralelism la nivel de date şi de procese pentru sisteme cu capacități distribuite de stocare. Operatorii de prelucrare de imagini sunt paralelizați prin descompunerea datelor utilizând structuri algoritmice. Aplicațiile de prelucrare de imagini sunt paralelizate prin descompunere în procese, pe baza grafului proceselor aplicațiilor imagistice. Astfel, o aplicație de prelucrare de imagini poate fi paralelizată prin descompunere la nivel de date şi de procese, abordare care asigură o îmbunătățire a performanțelor. Evaluăm performanța metodei folosind o aplicație de tip stereo vision multi-proces.*
Cuvinte cheie: *paralelism la nivel de date, paralelism la nivel de procese, structuri, prelucrare de imagini*
Abstract *The paper presents a data and task parallel low-level image processing environment for distributed memory systems. Image processing operators are parallelized by data decomposition using algorithmic skeletons. Image processing applications are parallelized by task decomposition, based on the image application task graph. In this way, an image processing application can be parallelized both by data and task decomposition, and thus better speed-ups can be obtained. We validate our method on the multi-baseline stereo vision application.*
Keywords: *Data parallelism; Task parallelism; Skeletons; Image processing*
2002 Elsevier Science B.V. , Parallel Computing 28(2002), 945-965

1. Introduction

Image processing is widely used in many application areas including the film industry, medical imaging, industrial manufacturing, weather forecasting etc. In some of these areas the size of the images is very large yet the processing time has to be very small and sometimes real-time processing is required. Therefore, during the last decade there has been an increasing interest in the developing and the use of parallel algorithms in image processing. Many algorithms have been developed for parallelizing different image operators on different parallel architectures. Most of these parallel image processing algorithms are either architecture dependent, or specifically developed for different applications and hard to implement for a typical image processing user without enough knowledge of parallel computing.
In this paper we present an approach of adding data and task parallelism to an image processing library using algorithmic skeletons [3–5] and the image application task graph (IATG). Skeletons are algorithmic abstractions common to a series of applications, which can

[1] Philips CFT, Industrial Vision, P.O. Box 218/SAN-6089, 5600 MD Eindhoven, Netherlands

[2] Faculty of Applied Physics, Pattern Recognition Group, Delft University of Technology, Lorentzweg 1, 2628 CJ Delft, Netherlands

be implemented in parallel. Skeletons are embedded in a sequential host language, thus being the only source of parallelism in a program. Using skeletons we create a data parallel image processing framework which is very easy to use for a typical image processing user.
It is already known that exploiting both task and data parallelism in a program to solve very large computational problems yields better speed-ups compared to either pure task parallelism or pure data parallelism [7,8]. The main reason is that both data and task parallelism are relatively limited, and therefore using only one of them limits the achievable performance. Thus, exploiting mixed task and data parallelism has emerged as a natural solution. For many applications from the field of signal and image processing, data set sizes are limited by physical constraints and cannot be easily increased. In such cases the amount of available data parallelism is limited. For example, in the multi-baseline stereo application described in Section 5, the size of an image is determined by the circuitry of the video cameras and the throughput of the camera interface. Increasing the image size means buying new cameras and building a faster interface, which may not be feasible. Since the data parallelism is limited, additional parallelism may come from tasking. By coding the image processing application using skeletons and having the IATG we obtain a both data and task parallel environment.
The paper is organized as follows. Section 2 briefly presents a description of algorithmic skeletons and a survey of related work. Section 3 presents a classification of low-level image operators and skeletons for parallel low-level image processing on a distributed memory system. Section 4 presents some related work and describes the IATG used in the task parallel framework. The multi-baseline stereo vision application together with its data parallel code using skeletons versus sequential code and the speed-up results for the data parallel approach versus the data and task parallel approach is presented in Section 5. Finally, concluding remarks are made in Section 6.

2. Skeletons and related work

Skeletons are algorithmic abstractions which encapsulate different forms of parallelism, common to a series of applications. The aim is to obtain environments or languages that allow easy parallel programming, in which the user does not have to handle with problems as communication, synchronization, deadlocks or nondeterministic program runs. Usually, they are embedded in a sequential host language and they are used to code and hide the parallelism from the application user.
The concept of algorithmic skeletons is not new and a lot of research is done to demonstrate their usefulness in parallel programming. Most skeletons are polymorphic higher-order functions, and can be defined in functional languages in a straightforward way. This is the reason why most skeletons are build upon a functional language [3,4]. Work has also been done in using skeletons in image processing. In [5] Serot et al. presents a parallel image processing environment using skeletons on top of CAML functional language.
In this paper we develop algorithmic skeletons to create a parallel image processing environment ready to use for easy implementation/development of parallel image processing applications. The difference from the previous approach [5] is that we allow the application to be implemented in a C programming environment and that we allow the possibility to use/implement different scheduling algorithms for obtaining the minimum execution time.

3. Skeletons for low-level image processing

3.1. A classification of low-level image operators

Low-level image processing operators use the values of the image pixels to modify the image in some way. They can be divided into point operators, neighborhood operators and global operators [1,2]. Below, we discuss in detail about all these three types of operator s.

3.1.1. Point operators

Image point operators are the most powerful functions in image processing. A large group of operator s falls in this category. Their main characteristics is that a pixel from the output image depends only on the corresponding pixel from the input image. Point operators are used to copy an image from one memory location to another, in arithmetic and logical operations, table lookup, image compositing. We will discuss in detail arithmetic and logic operators, classifying them from the point of view of the number of images involved, this being an important issue in developing skeletons for them.

3.1.1.1. Arithmetic and logic operations

Image ALU operations are fundamental operations needed in almost any imaging product for a variety of purposes. We refer to operations between an image and a constant as monadic operations, operations between two images as dyadic operations and operations involving three images as triadic operations.

• Monadic image operations: Monadic image operators are ALU operators between an image and a constant. These operations are shown in Table 1—s(x, y) and d(x, y) are the source and destination pixel values at location (x, y), and K is the constant.

Table 1 Monadic image operations

Function	Operation
Add constant	d(x, y) = s(x, y) + K
Subtract constant	d(x, y) = s(x, y) - K
Multiply constant	d(x, y) = s(x, y) · K
Divide by constant	d(x, y) = s(x, y) / K
Or constant	d(x, y) = K or s(x, y)
And constant	d(x, y) = K and s(x, y)
Xor constant	d(x, y) = K xor s(x, y)
Absolute value	d(x, y) = abs(s(x, y))

Monadic operations are useful in many situations. For instance, they can be used to add or subtract a bias value to make a picture brighter or darker.

• Dyadic image operators: Dyadic image operators are arithmetic and logical functions between the pixels of two source images producing a destination image. These functions are shown below in Table 2—s1(x, y) and s2(x, y) are the two source images that are used to create the destination image d(x, y). Dyadic operators have many uses in image processing. For example, the subtraction of one image from another is useful for studying the flow of blood in digital subtraction angiography or motion compensation in video coding. Addition of images is a useful step in many complex imaging algorithms like development of image restoration algorithms for modeling additive noise, and special effects, such as image morphing, in motion pictures.

Table 2 Dyadic image operations

Function	Operation
Add	d(x, y) = s1(x, y) + s2(x, y)
Subtract	d(x, y) = s1(x, y) - s2(x, y)
Multiply	d(x, y) = s1(x, y) · s2(x, y)
Divide	d(x, y) = s1(x, y) / s2(x, y)
Min	d(x, y) = min(s1(x, y), s2(x, y))
Max	d(x, y) = max(s1(x, y), s2(x, y))
Or	d(x, y) = s1(x, y) or s2(x, y)
And	d(x, y) = s1(x, y) and s2(x, y)

• Triadic image operators: Triadic operators use three input images for the computation of an output image. An example of such an operation is alpha blending. Image compositing is a useful function for both graphics and computer imaging. In graphics, compositing is used to combine several images into one. Typically, these images are rendered separately, possibly using different rendering algorithms. For example, the images may be rendered separately, possibly using different types of rendering hardware for different algorithms. In image processing, compositing is needed for any product that needs to merge multiple pictures into one final image.
All image editing programs, as well as programs that combine synthetically generated images with scanned images, need this function.
In computer imaging, the term alpha blend can be defined using two source images s1 and s2, an alpha image a and a destination image d, see formula (1).

$$d(x,y) = (1-\alpha(x,y))S_1(x,y) + \alpha(x,y)S_2(x,y) \qquad (1)$$

Another example of a triadic operator is the squared difference between a reference image and two shifted images, an operator used in the multi-baseline stereo vision application, described in Section 5.

3.1.2. Local neighborhood operators

Neighborhood operators (filters) create a destination pixel based on the criteria that depend on the source pixel and the value of pixel s in the ‘‘neighborhood’’ surrounding it. Neighborhood filters are largely used in computer imaging. They are used for enhancing and changing the appearance of images by sharpening, blurring, crispening the edges, and noise removal. They are also useful in image processing applications as object recognition, image restoration, and image data compression. We define a filter as an operation that changes pixels of the source image based on their values and those of their surrounding pixels. We may have linear and nonlinear filters.

3.1.2.1. Linear filtering versus nonlinear filtering

Generally speaking, a filter in imaging refers to any process that produces a destination image from a source image. A *linear* filter has the property that a weighted sum of the source images produces a similarly weighted sum of the destination images. In contrast to linear filters, *nonlinear* filters are somewhat more difficult to characterize. This is because the output of the

filter for a given input cannot be predicted by the impulse response. Nonlinear filters behave differently for different inputs.

3.1.2.2. Linear filtering using two-dimensional discrete convolution

In imaging, two dimensional (2D) convolution is the most common way to implement a linear filter. The operation is performed between a source image and a 2D convolution kernel to produce a destination image. The convolution kernel is typically much smaller than the source image. Starting at the top of the image (the top left corner which is also the origin of the image), the kernel is moved horizontally over the image, one pixel at a time. Then it is moved down one row and moved horizontally again. This process is continued until the kernel has traversed the entire image. For the destination pixel at row m and column n, the kernel is centered at the same location in the source image.
Mathematically, 2D discrete convolution is defined as a double summation. Given an M x N image *f(m, n)* and K x L convolution kernel *h(k, l)*, we define the origin of each to be at the top left corner. We assume that *f (m, n)* is much larger than *h(k, l).* Then, the result of convolving *f(m, n)* by *h(k, l)* is the image *g(m, n)* given by formula (2):

$$g(m,n)=\sum_{x=0}^{K-1}\sum_{y=0}^{L-1} f\left(m+\frac{K-1}{2}-x,\quad n+\frac{L-1}{2}-y\right)h(x,y) \qquad (2)$$

$$0 \le m \le M-1, \quad 0 \le n \le N-1$$

In the above formula we assume that K, L are odd numbers and we extend the image by (K-1)/2 lines in each vertical direction and by (L-1)/2 columns in each horizontal direction. The sequential time complexity of this operation is O(MNKL). As it can be observed, this is a time consuming operation, very well fitted to the data parallel approach.

3.1.3. Global operators

Global operators create a destination pixel based on the entire image information. A representative example of an operator within this class is the discrete Fourier transform (DFT). The DFT converts an input data set from the temporal/spatial domain to the frequency domain, and vice versa. It has a lot of applications in image processing, being used for image enhancement, restoration, and compression.
In image processing the input is a set of pixels forming a 2D function that is already discrete. The formula for the output pixel X_{lm} is the following:

$$X_{lm}=\frac{1}{NM}\sum_{j=0}^{N-1}\sum_{k=0}^{M-1} x_{jk}\, e^{-2\pi i\left(\frac{jl}{N}+\frac{km}{M}\right)} \qquad (3)$$

where *j* and *k* are column coordinates, $0 \le j \le N-1$ and $0 \le k \le M-1$. We also include in the class of global operators, operators like the histogram transform, which do not have an image as output, but another data structure.

3.2. Data parallelism of low-level image operators

From the operator description given in the previous section we conclude that point, neighborhood and global image processing operators can be parallelized using the data parallel paradigm with a host/node approach. A host processor is selected for splitting and distributing the data to the other nodes. The host also processes a part of the image. Each node processes its received part of the image and then the host gathers and assembles the image back together. In Figs. 1–3 we present the data parallel paradigm with the host/node approach for point, neighborhood and global operators. For global operators we send the entire image to the corresponding nodes but each node will process only a certain part of the image. In order to avoid extra inter-processor communication due to the border information exchange for neighborhood operators we extend and partition the image as showed in Fig. 2. In this way, each node processor receives all the data needed for applying the neighborhood operator.
Based on the above observations we identify a number of skeletons for parallel processing of low-level image processing operators. They are named according to the type of the low-level operator and the number of images involved in the operation. Headers of some skeletons are shown below. All of them are based on a ‘‘distribute compute and gather’’ (DCG) main skeleton, previously known as the map skeleton [4], suitable for regular applications as the low-level operators from image processing. The implementation of all the skeletons is based on the idea described in the above paragraph, see Figs. 1–3. Each skeleton can run on a set of processors. From this set of processors a host processor is selected to split and distribute the image(s) to the other nodes, each other node from the set receives a part of the image(s) and the image operator which should be applied on it, then the computation takes place and the result is sent back to the host processor. The skeletons are implemented in C using MPI-Panda library [19,20]. The implementation is transparent to the user.

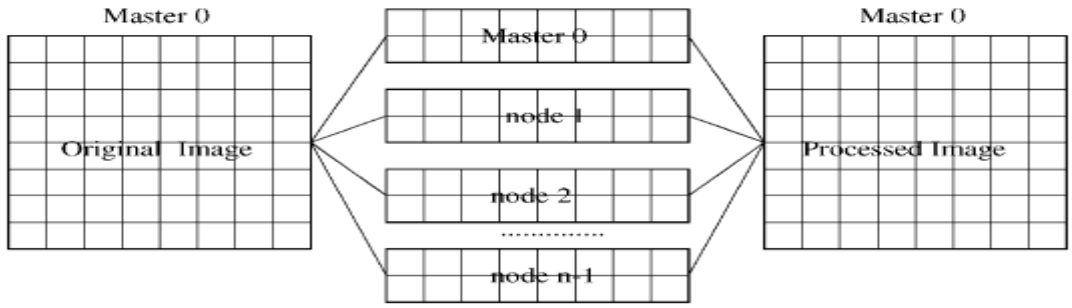

Fig. 1. DCG skeleton for point operators.

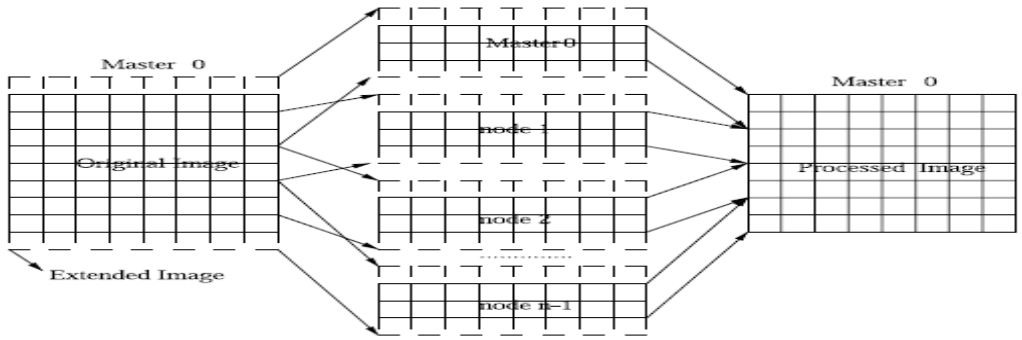

Fig. 2. DCG skeleton for neighborhood operators.

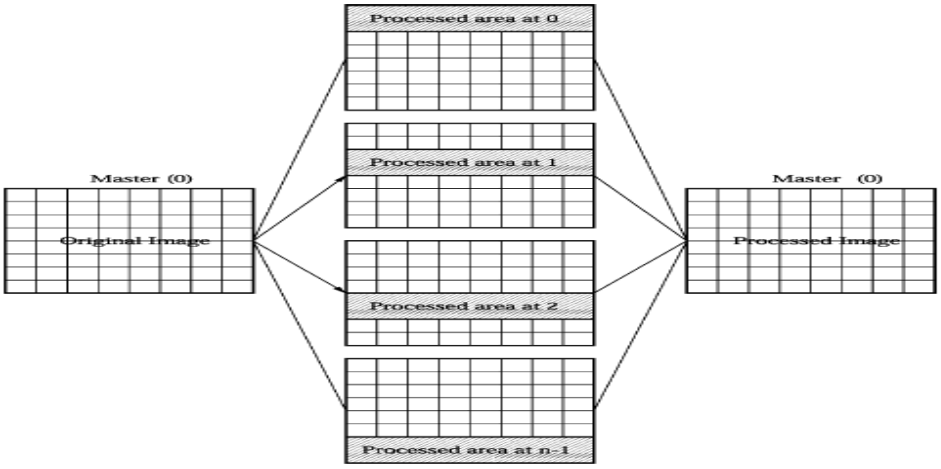

Fig. 3. DCG skeleton for global operators.

```
/* dist.h */
. . .
void ImagePointDist_1IO(unsigned int n, char *name,
void(*im_op)());
// DCG skeleton for monadic point operators - one Input/Output
void ImagePointDist_1IO_C(unsigned int n, char *name,
void(*im_op)(),float ct);
// DCG skeleton for monadic point operators which need a constant
value as pararameter
// one Input/Output
void ImagePointDist_1I_1O(unsigned int n, char *name1,
char *name2,void(*im_op)());
// DCG skeleton for monadic/dyadic point operators - one Input
and one Output
void ImagePointDist_1IO_1I(unsigned int n, char *name1,
char *name2,void(*im_op)());
// DCG skeleton for monadic/dyadic point operators - one Input/
Output and one Input.
void ImagePointDist_2I_1O(unsigned int n, char *name1,
char *name2,char *name3,void(*im_op)());
// DCG skeleton for dyadic/triadic point operators - 2 Inputs
and one Output
void ImagePointDist_2I_2O(unsigned int n, char *name1,
char *name2,char *name3,char *name4,
void(*im_op)());
// DCG skeleton - 2 Inputs and 2 Outputs
void ImagePointDist_3I_1O(unsigned int n, char *name1,
char *name2,char *name3,char *name4,
void(*im_op)());
// DCG skeleton for triadic point operators - 3 Inputs and one
```

```
Output
. . .
void ImageWindowDist_1IO(unsigned int n, char *name,
Window *win, void(*im_op)());
// DCG skeleton for neighborhood operators - one Input/Output
void ImageWindowDist_1I_1O(unsigned int n, char *name1,
char *name2,Window *win, void(*im_op)());
// DCG skeleton for neighborhood operators - one Input and
one Output
. . .
void ImageGlobalDist_1IO(unsigned int n, char *name,
void(*im_op)());
// DCG skeleton for global operators - one Input/Output
. . .
```

We develop several types of skeletons, which depend on the type of the low-level operator (point, neighborhood, global) and the number of input/output images. With each skeleton we associate a parameter which represents the task number corresponding to that skeleton. This is used by the task parallel framework. Depending on the skeleton type, one or more identifiers of the images are given as parameters. The last argument is the point operator for processing the image(s). So, each skeleton is used for a number of low-level image processing operators which perform in a similar way (for instance all dyadic point operators take two input images, combine and process them depending on the operator type and then produce an output image). Depending on the operator type and the skeleton type, there might exist additional parameters necessary for the image operator. For point operators we assigned the ImagePointDist skeletons, for neighborhood operators we assigned the ImageWindowDist skeletons, and for global operators we assigned the ImageGlobalDist skeletons. Some of the skeletons modify the input image (ImagePointDist_1IO, ImageWindowDist_1IO, ImageGlobalDist_1IO, so 1IO stands for 1 Input/Output image), other skeletons take a number of input images and create a new output image, for example the ImagePointDist_2I_1O skeleton for point operators takes 2 input images and creates a new output image. This skeleton is necessary for dyadic point operators (like addition, subtraction, etc., see Table 2) which create a new image by processing two input images. Similarly, the skeleton ImagePointDist_3I_1O for point operators takes 3 input images and creates a new output image. An example of a low-level image processing operator suitable for this type of skeleton is the squared difference between one reference image and two disparity images, operator used in the multi-baseline stereo vision application, see Table 3 and Section 5.

Table 3: Triadic image operations

Function	Operation
Alpha blend	$d(x,y) = (1-\alpha(x,y))S_1(x,y) + \alpha(x,y)S_2(x,y)$
Squarad diff	$d(x,y) = (ref(x,y) - S_1(x,y))^2 + (ref(x,y) - S_2(x,y))^2$

Similar skeletons exist also for local neighborhood and global operators. ImagePointDist_1IO_C is a skeleton for monadic point operators which need a constant value as parameter, for processing the input image, see Table 1.

Below we present an example of using the skeletons to code a very simple image processing application in a data parallel way. It is an image processing application of edge detection using Laplace and Sobel operators. First we read the input image and we create the two output images and a 3 x 3 window, and then we apply the Laplace and Sobel operators on the *num_nodes* number of processors. *num_nodes* is the number of nodes on which the application is run and is detected on the first line of the partial code showed below. *image_in* is the name of the input image given as input parameter to both skeletons and *image_l*, *image_s* are the output parameters (images) for each skeleton. We have used a ImageWindowDist_1I_1O skeleton to perform both operators. The last two parameters are the window used (which contains information about the size and the data of the window) and the image operator that is applied via the skeleton.

```
. . .
num_nodes = ImageInitPar(argc, argv);
win= CreateWindow(3,3);
im_i = ReadImage(256,256,00image_in00,file,PAR_DOUBLE);
im_l = CreateImage(256,256,00image_l00,PAR_DOUBLE);
im_s = CreateImage(256,256,00image_s00,PAR_DOUBLE);
ImageWindowDist_1I_1O(num_nodes,00image_in00,00image_l00,win,
laplace_3_3);
ImageWindowDist_1I_1O(num_nodes,00image_in00,00image_s00,win,
sobel_3_3);
. . .
```

4. The task parallel framework

Recently, it has been shown that exploiting both task and data parallelism in a program to solve very large computational problems yields better speed-ups compared to either pure data parallelism or either pure task parallelism [7,8]. The main reason is that both task and data parallelism are relatively limited, and therefore using only one of them bounds the achievable performance. Thus, exploiting mixed task and data parallelism has emerged as a natural solution. We show that applying both data and task parallelism can improve the speed-up at the application level.

There have been considerable effort in adding task parallel support to data parallel languages, as in Fx [10], Fortran M[11] or Paradigm HPF [7], or adding data parallel support to task parallel languages such as in Orca [12]. In order to fully exploit the potential advantage of the mixed task and data parallelism, efficient support for task and data parallelism is a critical issue. This can be done not only at the compiler level, but also at the application level and applications from the image processing field are very suitable for this technique.

Mixed task and data parallel techniques use a directed acyclic graph, in the literature also called a macro-dataflow graph (MDG) [7], in which data parallel tasks (in our case the image processing operators) are the nodes and the precedence relationships are the edges. For the purpose of our work we change the name of this graph to the IATG.

4.1. The image application task graph model

A task parallel program can be modeled by a macro-dataflow communication graph [7], which is a directed acyclic graph *G(V, E, w, c)*, where:

• V is the finite set of nodes which represents tasks (image processing operators);
• E is the set of directed edges which represent precedence constraints between tasks:

$e = (u, v) \in E$ if u $\prec$ v

• w is the weight function $w : V \to N^*$ which gives the weight (processing time) of each node (task). Task weights are positive integers;
• c is the communication function $c : E \to N^*$ which gives the weight (communication time) of each edge. Communication weights are positive integers.

An IATG is, in fact, an MDG in which each node stands for an image processing operator and each edge stands for a precedence constraint between two adjacent operators. In this case, a node represents a larger entity that in the MDG where a node can be any simple instruction from the program. Some important properties of the IATG are:

• It is a weighted directed acyclic graph.
• Nodes represent image processing operators and edges represent precedence constraints between them.
• The are two distinguished nodes: START precedes all other nodes and STOP succeeds all other nodes.

We define a well balanced IATG as an application task graph which has the same type of tasks (image operators) on each level. An example is the IATG of the multi-baseline stereo vision application, described in Section 5, Fig. 7, which on the first level has the squared difference operator applied to three images for each task and on the second level the error operator is executed by all the tasks. Moreover, the graph edges form a regular pattern.
The weights of node s and edges in the IATG are based on the concepts of processing and communication costs. Processing costs account for the computation and communication costs of data parallel tasks—image processing operators corresponding to nodes, and depend on the number of processors allocated to the node. Communication costs account for the costs of data communication between nodes.

4.2. Processing cost model

A node in the IATG represents a processing task (an image processing operator applied via a DCG skeleton, as described in Section 3.2) that runs non-preemptively on any number of processors. Each task *i* is assumed to have a *computation cost*, denoted $T_{exec}(t, p(t))$, which is a function of the number of processors. The computation cost function of the task can be obtained either by *estimation* or by *profiling*.
For cost estimation we use Amdahl's law. According to it, the execution time of the task t is:

$$T_{exec}(t, p(t)) = \left(\alpha(t) + \frac{1 - \alpha(t)}{p(t)} \right) \tau(t) \qquad (4)$$

where *t* is the task number, *p(t)* is the number of processors on which task *t* is executed, $\tau(t)$ is the task's execution time on a single processor and $\alpha(t)$ is the fraction of the task that executes serially.
If we use profiling, the task's execution costs are either fitted to a function similar to the one described above (in the case that data is not available for all processors), or the profiled values can be used directly through a table. The values are simple to determine, we measure the execution times of the basic image processing operators implemented in the image processing library and we tabulate their values.

4.3. Communication cost model

Data communication (redistribution) is essential for implementing an execution scheme which uses both data and task parallelism. Individual tasks are executed in a data parallel fashion on subsets of processors and the data dependences between tasks may necessitate not only changing the set of processors but also the distribution scheme. Fig. 4 illustrates a classical approach of redistribution between a pair of tasks. Task A is executed using seven processors and reads from data D. Task B is executed using four processors and reads from the same data D.

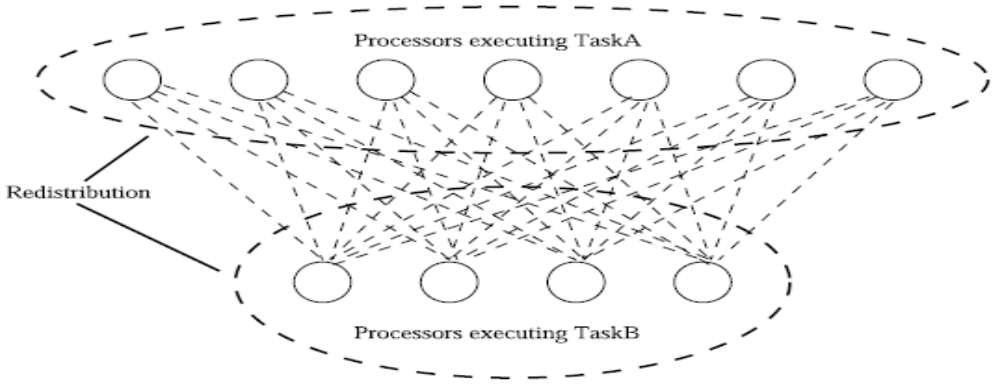

Fig. 4. Data redistribution between two tasks.

This necessitates the redistribution of the data D from the seven processors executing Task A to the four processors executing Task B. In addition to changing the set of processors we could also change the distribution scheme of the data D. For instance, if D is a 2D data then Task A might use a block distribution for D, whereas Task B might use a row-stripe distribution.

We reduce the complexity of the problem first by allowing only one type of distribution scheme (row-stripe) and second by sending images only between two processors (the selected host processors from the two sets of processors), as shown in Fig. 5.

An edge in the IATG corresponds to a precedence relationship and has associated a communication cost, denoted through $T_{comm}(i, j)$ which depends on the network characteristics (latency, bandwidth) and the amount of data to be transferred. It should be emphasized that there are two types of communication times.

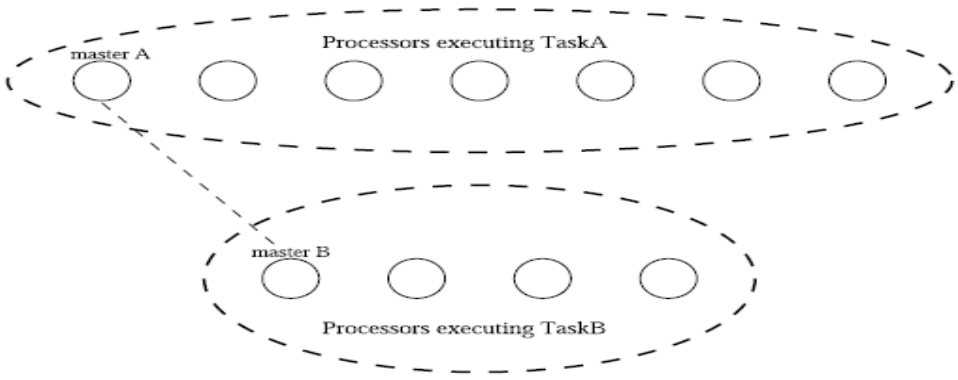

Fig. 5. Image communication between two host processors.

First, we have internal communication time which represents the time for internal transfer of data between the processors allocated to a task. This quantity is part of the term a of the execution time associated to a node oft he graph. Secondly, we have external communication time which is the time of transferring data, i.e. images, between two processors. These two processors represent the host processors for the two associated image processing tasks (corresponding to the two adjacent graph nodes). This quantity is actually the communication cost of an edge of the graph.
In this case we can also use either cost estimation or profiling to determine the communication time. In state-of-the-art of distributed memory systems the time to send a message containing L units of data from a processor to another processor can be modeled as:

$$T_{comm}(i, j) = t_s + Lt_b \qquad (5)$$

where ts, tb are the startup and per byte cost for point-to-point communication and L is the length of the message, in bytes.
We run our experiments on a distributed memory system which consists of a cluster of Pentium Pro/200 MHz PCs with 64Mb RAM running Linux, and connected through Myrinet in a three-dimensional (3D)-mesh topology, with dimension order routing [16]. Fig. 6 shows the performance of point-to-point communication operations and the predicted communication time. The reported time is the minimum time obtained over 20 executions of the same code. It is reasonable to select the minimum value because of the possible interference caused by other users' traffic in the network. From these measurements we perform a linear fitting and we extract the communication parameters ts and tb. In Fig. 6 we see that the predicted communication time, based on the above formula, approximates very good the measured communication time.

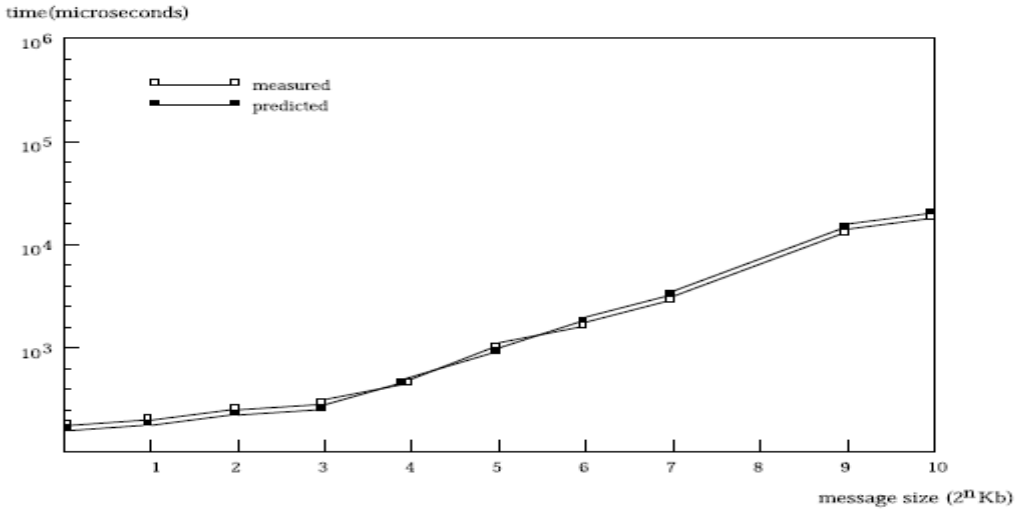

Fig. 6. Performance of point-to-point communication on DAS.

4.4. Image application task graph cost properties

A task with no input edges is called an entry task and a task with no output edges is called an exit task. The length of a path from the graph is the sum of the computation and communication costs of all nodes and edges belonging to the path. We define the *critical path* (CP) [7] as the longest path in the graph. If we have a graph with *n* nodes, where *n* is the last

node of the graph and t_i represents the finish time of node i, $T_{exec}(i, p(i))$ is the execution time of task *i* on a set of *p(i)* nodes then the CP is given by the formulas (6) and (7), where PREDi is the set of immediate predecessor nodes of node *i*.

$$CP = t_n \tag{6}$$

$$t_i = \max\left(t_p + T_{comm}(p.i)\right) + T_{exec}(i.p(i)) \quad p \in PRED_i \tag{7}$$

We define the *average area* (A) [7] of an IATG with n nodes (tasks) for a *P* processor system as in formula (8), where *p(i)* is the number of processors allocated to task T_i.

$$A = \frac{1}{P}\sum_{i=1}^{n} T_{exec}(i, p(i)) \cdot p(i) \tag{8}$$

The CP represents the longest path in the IATG and the average area provides a measure of the processor-time area required by the IATG. Based on these two formulas, processors are allocated to tasks according to the results obtained by solving the following minimization problem:

$$\phi = \min\left(\max(A, CP)\right) \tag{9}$$

subject to $1 \le p(i) \le P, \quad \forall i = \overline{1, n}$

After solving the allocation problem, a scheduler is needed to schedule the tasks to obtain a minimum execution time. The classical approach is the well-known list scheduling paradigm [13] introduced by Graham, which schedules one processor tasks (tasks running only on one processor). Scheduling is known to be NP-complete for one processor tasks. Since then several other list scheduling algorithms were proposed, and the scheduling problem was also extended to multiple processor tasks (tasks that run non-preemptively on any number of processors) [7]. Therefore, multiple processor task scheduling is also NP-complete and heuristics are used.

The intuition behind minimizing ϕ in Eq. (9) is that / represents a theoretical lower bound on the time required to execute the image processing application corresponding to the IATG. The execution time of the application can neither be smaller than the CP of the graph nor be less than the average area of the graph.

As the TSAS's convex programming algorithm [7] for determining the number of processors for each task was not available, we have used in the experimental part of Section 5 the nonlinear solver based on SNOPT [17] available on the internet [18] for solving the previous min–max problem. For solving the scheduling problem, the proposed scheduling algorithm presented in [7] is used. Another possibility is to use scheduling algorithms developed for data and task parallel graphs [8,9].

5. Experiments

To evaluate the benefits of the propose data parallel framework based on skeletons and also of the task parallel framework based on the IATG, we first compare the code of the multi-baseline stereo vision algorithm with and without using skeletons (with and without data

parallelism). Then we compare the speed-ups obtained by applying only data parallelism to the application, with the speed-ups obtained
with both data and task parallelism.
The multi-baseline stereo vision application uses an algorithm developed by Okutomi and Kanade [6] and described by Webb et al. [14,15], that gives greater accuracy in depth through the use of more than two cameras. Input consists of three *n x n* images acquired from three horizontally aligned, equally spaced cameras. One image is the *reference image*, the other two are named *match images*. For each of 16 disparities, *d = 0, ...,15*, the first match image is shifted by *d* pixels, the second image is shifted by *2d* pixels. A *difference image* is formed by computing the sum of squared differences between the corresponding pixels of the reference image and the shifted match images. Next, an *error image* is formed by replacing each pixel in the difference image with the sum of the pixels in a surrounding 13 x 13 window. A *disparity image* is then formed by finding, for each pixel, the disparity that minimizes error. Finally, the depth of each pixel is displayed as a simple function of its disparity. Fig. 7 presents the IATG of this application.
It can be observed that the computation of the difference images requires point operators, while the computation of the error images requires neighborhood operators. The computation of the disparity image requires also a point operator.

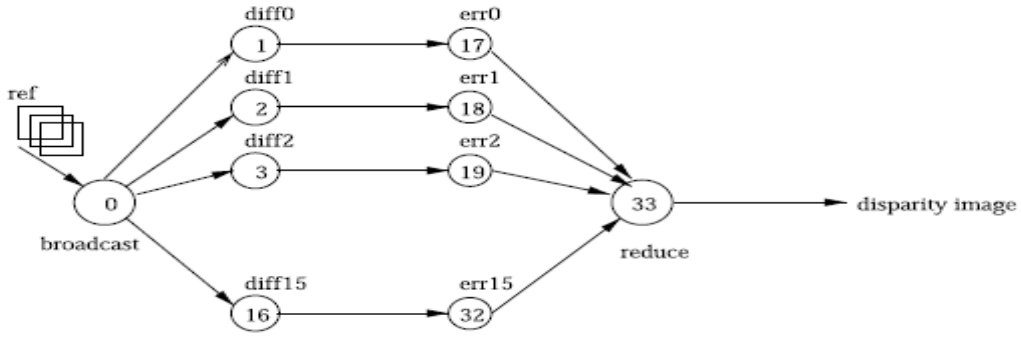

Fig. 7. Multi-baseline stereo vision IATG.

Input:ref, m1, m2 (the reference and the two match images)
for d =0,15
Task T1,d:m1 shifted by d pixels
Task T2,d:m2 shifted by 2*d pixels
Task T3,d:diff =(ref-m1)*(ref-m1)+(ref-m2)*(ref-m2)
Task T4,d:err(d) =sum diff[i,j]
Task T5:Disparity image= d which minimizes the err image

Pseudocode of the multi-baseline stereo vision application

Below we present the sequential code of the application versus the data parallel code of the application. Coding the application by just combining a number of skeletons does not require much effort from the image processing user, yet it parallelizes the application. The data and task parallel code is slightly more difficult and we do not present it here.

```
. . .
n =ImageInitPar(argc,argv);
ref =CreateImage(256,256,00ref00,PAR_DOUBLE);
```

```
InitImage_ij(ref);
m1 = CreateImage(256,256,00m100,PAR_DOUBLE);
InitImage_ij(m1);
m2 =CreateImage(256,256,00m200,PAR_DOUBLE);
InitImage_ij(m2);
im =CreateImage(256,256,00im00,PAR_DOUBLE);
InitImageZero(im);
for(k =0;k<16;k++) {
DisparityImage(m1,d);
DisparityImage(m2,d);
Image2DifSqr(im,ref,m1,m2);
ImageErr(im,win);
ImageMin(im,min);
}
. . .
```

Sequential code

```
. . .
n =ImageInitPar(argc,argv);
if(My_Processor()¼ ¼MASTER) {
ref =CreateImage(256,256,00ref00,PAR_DOUBLE);
InitImage_ij(ref);
m1 =CreateImage(256,256,00m100,PAR_DOUBLE);
InitImage_ij(m1);
m2 = CreateImage(256,256,00m200,PAR_DOUBLE);
InitImage_ij(m2);

im = CreateImage(256,256,00im00,PAR_DOUBLE);
InitImageZero(im);
}
win = CreateWindow(3,3);
for(d = 0;d<16;d++) {
if(My_Processor() = =MASTER) {
DisparityImage(m1,d);
DisparityImage(m2,2*d);
}
ImagePointDist_3I_1O(d,00im00,00ref00,00m100,
00m200,n,list_nodes,Image2DifSqr);
ImageWindowDist_1IO(d,00im00,n,list_nodes,win,ImageErr);
ImagePointDist_1IO(d,00im00,n,list_nodes,ImageMin);
}
. . .
```

DT-PIPE code based on skeletons

Besides creating the images on the host processor, the code is nearly the same, only the function headers differ. The skeleton have as parameters the name of the images, the window and the image operator, while in the sequential version operator headers have as parameters the images and the window. The skeletons are implemented in C using MPI [19].

The results of the data parallel approach are compared with the results obtained using data and task parallelism on a distributed memory system which consists of a cluster of Pentium

Pro/200MHz PCs with 64Mb RAM running Linux [16], and connected through Myrinet in a 3D-mesh topology, with dimension order routing. In the task parallel framework we use a special mechanism to register the images on the processors where they are first created. Moreover, each skeleton has associated the task number to which it corresponds. We use 1, 2, 4, 8, 16, 32 and 64 processing nodes in the pool. Three artificial reference images of sizes 256 x 256, 512 x 512 and 1024 x 1024 are used. The code is written using C and MPI message passing library. The multi-baseline stereo vision algorithm is an example of a regular well balanced application in which task parallelism can be applied without the need of an allocator of scheduler . Just for comparison reasons, we have used the algorithm described in [7] and we have obtained identical results (we divide the number of nodes to the number of tasks and we obtain the number of the nodes on which each task should run). In Fig. 8 we show the speed-ups obtained for the data parallel approach for different image sizes. Fig. 9 shows the speed-up of the same application using the data and task parallel approach, also for different image sizes. We can observe that the speed-ups become quickly saturated for the data parallel approach while the speed-ups for the data and task parallel approach perform very good. In fact, we have pure task parallelism up to 16 processors and data and task parallelism from 16 on. So, the pure task parallel speed-ups will become flattened from 16 processors on because at this type of application is better to first apply task parallelism

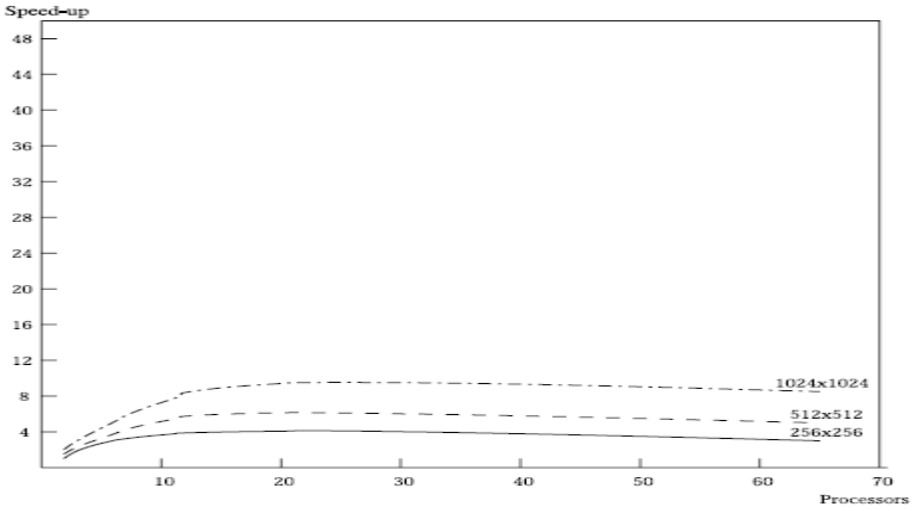

Fig. 8. Speed-up for the data parallel approach.

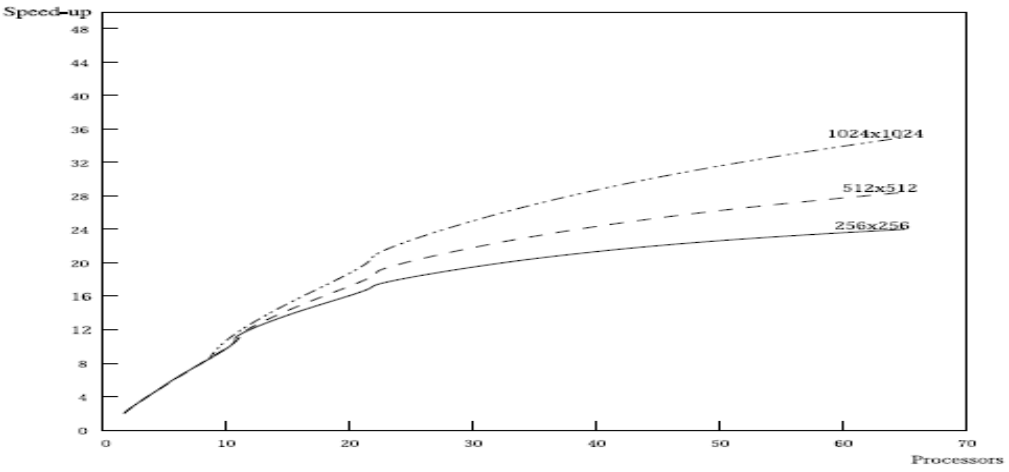

Fig. 9. Speed-up for the data and task parallel approach.

and then to add data parallelism. Using both data and task parallelism is more efficient than using only data parallelism.

6. Conclusions

We have presented an environment for data and task parallel image processing. The data parallel framework, based on algorithmic skeletons, is easy to use for any image processing user. The task parallel environment is based on the IATG and computing the IATG communication and processing costs. If the IATG is a regular well balanced graph task parallelism can be applied without the need of these computations. We showed an example of using skeletons and the task parallel framework for the multi-baseline stereo vision application. The multi-baseline stereo vision is an example of an image processing application which contain parallel tasks, each of the tasks being a very simple image point or neighborhood operator. Using both data and task parallelism is more efficient than using only data parallelism. Our code for the data and task parallel environment, written using C and the MPI-Panda library [19,20] can be easily ported to other parallel machines.

References

[1] I. Pitas, Parallel Algorithms for Digital Image Processing, Computer Vision and Neural Networks, John Wiley & Sons, New York, 1993.

[2] B. Wilkinson, M. Allen, Parallel Programming, Prentice Hall, Englewood Cliffs, NJ, 1999.

[3] M. Cole, Algorithmic skeletons: structured management of parallel computations, Pitman/MIT Press, 1989.

[4] J. Darlington, Y.K. Guo, H.W. To, Y. Jing, Skeletons for structured parallel composition, in: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 1995.

[5] J. Serot, D. Ginhac, J.P. Derutin, SKiPPER : a skeleton-based programming environment for image processing applications, in: Fifth International Conference on Parallel Computing Technologies, 1999.

[6] M. Okutomi, T. Kanade, A multiple-baseline stereo, IEEE Transactions on Pattern Analysis and Machine Intelligence 15 (4) (1993) 353–363.

[7] S. Ramaswamy, S. Sapatnekar, P. Banerjee, A framework for exploiting task and data parallelism on distributed memory multi-computers, IEEE Transactions on Parallel and Distributed Systems 8 (11) (1997).

[8] J. Subhlok, B. Yang, Optimal use of mixed task and data parallelism for pipelined computations, Journal ofParallel and Distributed Computing 60 (2000) 297–319.

[9] A. Radulescu, C. Nicolescu, A. van Gemund, P.P. Jonker, CPR: Mixed Task and Data Parallel Scheduling for Distributed Systems, Best Paper Award, in: CDROM proceedings of the 15th International Parallel and Distributed Symposium (IPDPS'2001), San Francisco, April 23–28, 2001.

[10] J. Subhlok, B. Yang, A new model for integrated nested task and data parallel programming, in: Proceedings of the Symposium on Parallel Algorithms and Architectures, 1992.

[11] T.I. Foster, K.M. Chandy, Fortran M: a language for modular parallel programming, Journal of Parallel and Distributed Computing 26 (1995) 24–35.

[12] S.B. Hassen, H.E. Bal, C.J. Jacobs, A task and data parallel programming language based on shared objects, ACM Transactions on Programming Languages and Systems 20 (6) (1998) 1131–1170.
[13] R.L. Graham, Bounds on multiprocessing timing anomalies, SIAM Journal on Applied Mathematics 17 (2) (1969) 416–429.
[14] J. Webb et al, The CMU Task Parallel Program Suite, Technical Report Carnegie Mellon University, CMU-CS-94-131, 1994.
[15] J. Webb, Implementation and performance of fast parallel multi-baseline stereo vision, in: Proceedings of Computer Architectures for Machine Perception, 1993, pp. 232–240.
[16] The Distributed ASCI supercomputer (DAS) site, Available from: <http://www.cs.vu.nl/das>.
[17] P.E. Gill, W. Murray, M.A. Sanders, User's guide for snopt 5.3: a fortran package for large-scale nonlinear programming, Technical Report SOL-98-1, Stanford University, 1997.
[18] Lucent Technologies AMPL site, Available from: <http://www.ampl.com/cm/>.
[19] M. Snir, S. Otto, S. Huss, D. Walker, J. Dongarra, MPI—The Complete Reference, vol. 1, The MPI Core, The MIT Press, Cambridge, MA, 1998.
[20] T. Ruhl, H. Bal, R. Bhoedjang, K. Langendoen, G. Benson, Experience with a portability layer for implementing parallel programming systems, in: Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications, Sunnyvale CA, 1996, pp. 1477–1488.