Complexity Analysis Of Range Image Segmentation on MasPar MP-1*

Nagarajan Ramesh
Vision & Neural Networks Lab.
Dept. of Computer Science
Wayne State University
Detroit, MI 48202

Vipin Chaudhary
Dept. of Electrical and Computer Engineering
Wayne State University
Detroit, MI 48202

Abstract — Many low level vision tasks that are computationally intensive are easily parallelizable. The lack of parallel processing systems, or their prohibitive costs, have prevented the move of vision processing algorithms from single processor systems to multiprocessor systems. With the recent spurt of parallel processing hardware, there is a need to investigate the feasibility of using such machines for some vision algorithms. Speedup is an important factor in determining the feasibility of migration from single processor systems to parallel processors. In this work, we investigate a particular segmentation algorithm and present theoretical speedup results. Our formula can work out numerical speedups by simply plugging in the parameter values.

Introduction

Computer vision tasks require an enormous amount of computation demanding high performance computers for practical, real-time applications. Parallelism appears to be the only economical way to achieve this level of performance. Most of the work in computer vision focuses on images with 2-D data, but pragmatic vision problems require 3-D data which is easily available now.

Three dimensional data may be represented by a 3-D matrix of intensity values, $f(i,j,k)$, where each intensity value represents a property associated with the location (i,j,k). A primary goal (and initial step) of computer vision is to abstract "relevant" information from an image. This may involve a process called segmentation that groups a set of homogeneous pixels into regions. Homogeneity can be defined by different criteria depending upon the image modality. Segmentation thus reduces the information content in the image to the most relevant and by defining some features of the segmented regions, computer vision scientists hope to extract just enough information to characterize those regions. Such an abstraction will be helpful in other higher level tasks like object recognition [1] and visualization.

Two major approaches to segmentation are the region growing ones and the region splitting ones. In region growing, each pixel is considered in relation to its neighbors and pixels that are "closer" in some distance metric are merged. On the other hand, in region splitting, the whole image is initially considered to be one single region and this region is recursively split into smaller regions. Both these approaches are in general amenable to parallel implementation.

In this paper we derive the formulas for the theoretical time complexity and speedups obtainable on the implementation of a segmentation algorithm on the MasPar SIMD machine. In section we briefly describe the segmentation algorithm. In the following section we discuss the architecture of the SIMD machine, MP-1. In section , we derive the time complexity. section 0.0.2 is the concluding section of the paper.

Description of the algorithm

The segmentation algorithm of Sabata et. al. [2][3] was chosen as the candidate for implementation on a SIMD machine. The first stage of the two stage process involves oversegmenting the image based on zeroth and first order surface properties. The second stage involves merging of small regions into a final segmented output with each segmented region being specified by a bivariate polynomial.

A number of intensity images are generated using the 3-D range data, each assuming a light source at different points. The first segmentation module uses a pyramid structure of $p+1$ levels: Level 0 has 2^{p+1} x 2^{p+1} nodes with each image pixel being mapped to one node. The next level contain 2^p x 2^p nodes, and the i^{th} level containing 2^i x 2^i nodes.

Each node at level i can communicate with a set of nodes at level $(i-1)$ designated its sons, and with another set of nodes at level $(i+1)$ designated its fathers. First, all nodes at level i are initialized by averaging the pixels in a 2 x 2 (called span) neighborhood at level $(i-1)$. Between each node at level i and $(i+1)$ (or $(i-1)$) three parameters are defined: w_{ij} the membership function, u_{ij} the connection weight and d_{ij} the difference in property value between two nodes given as a product norm. Starting at the bottom most level, an iterative updating procedure is use to stablize w_{ij} and u_{ij}. Then the nodes at the next level undergo the same process, and this proceeds until the top most level in the pyramid has reached a fixed point. Each node at the top most level is then assigned a label which is propagated downwards to level 0, based on the highest weighting function. Thus each pixel at level 0 is labelled. This whole process is called *pyramidal node linking*.

Each generated image undergoes pyramidal node linking. It may happen (and often does) that a region R_1 in image I_1 and region R_2 in image I_2 do not occupy the same pixel locations. In such a case, another new region is formed in the resulting image which contain the region corresponding to $R_1 \cap R_2$, in addition to regions that correspond to R_1 in I_1 and R_2 in I_2. This operation on all the image regions will yield a large number of segmented regions, resulting in oversegmentation of the image.

The second stage is the merging process. Oversegmentation is followed by region merging based on a neighborhood criteria of least mean square (LMS) error of bivariate polynomial fits to adjacent regions. Adjacent regions with an error less than a threshold are merged. This stage is driven by the requirements of the higher level vision task. A segmented image is the final output.

Parallel implementation

We attempt to parallelize the pyramidal node linking module and analyze its time complexity. To do so, we need to map the nodes of the pyramid to the PEs of MP-1 in an efficient manner to minimize idle time of each PE and achieve load balancing. Communication bottlenecks and routing of messages have to be taken care of.

The configuration of the particular MasPar, MP-1 [4] we consider has 16K PEs (processing elements) arranged as a 128 x 128 array, each with local memory of 64K. Each PE can communicate with the other by two methods. One is a "router" that allows any PE to communicate with any other PE, and the other is the faster "XNet" connections between 8 neighboring PEs. The front end is a DEC station 3500. The front end can communicate to the array control unit (ACU) that has its own processor and memory. The ACU primarily controls the operations in the PEs, though it can do limited computation. The front end machine (FE) is used to load programs and data into the ACU and DPU while it can still act as a standalone workstation.

Time Complexity Analysis

Lets assume that we have an array of PEs of size $n \times n$, where $n = 2^a$ and an image of size $k \times k$, where $k = 2^b; a, b \geq 2; b \geq a; a, b \in \mathbf{I}$.

Let the number of levels in the pyramid be $p+1 \quad (l_0, l_1, \ldots l_p)$, (see figure **2**) and the number of iteration at each level be $i_1, i_2, \ldots i_p$ We assume that the time taken for each iteration is a constant, t. This is justifiable on a parallel machine each PE operates on one pixel which makes the time taken independent of the data size. We shall use some terminology similar to [5] to describe different levels of computation. One *stage* is complete when all the iterations between any two levels l_i and l_{i+1} is completed.

We consider two allocation schemes of tasks to processors, and

*This work was supported in part by NSF Grant MIP-9309489 and Wayne State University Faculty Research Award

CH 3381-1/93/$01.00 ©1993 IEEE

compare their theoretical speedups against a single processor system.

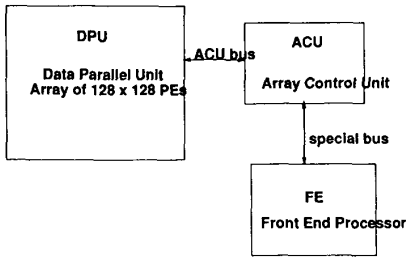

Figure 1: The ACU controls the DPU. The front end is used to load data and programs onto the ACU, DPU or just to execute them right there.

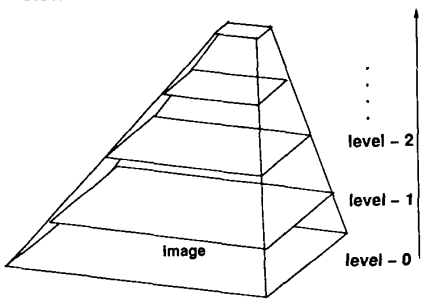

Figure 2: The image is processed in a pyramidal fashion. The processing iterates between levels l_0 and l_1, until a certain criteria is satisfied, after which processing between levels l_1 and l_2 commence. Processing proceeds in this fashion until the topmost level of the pyramid is reached.

Mapping Scheme 1

In this mapping scheme, we divide the image into a number of sub-images, each equal to the size of the PE array. The total time, T_{s1}, taken up by one subimage is given by

$$T_{s1} = t\sum_{j=1}^{p} i_j$$

Therefore, accounting for all the sub-images, the total time taken, T_{t1}, to complete the pyramidal algorithm is given by

$$T_{t1} = \frac{k^2}{n^2} t\sum_{j=1}^{p} i_j + t_{r1}(k) + t_{comm1}(k) + t_{s1}(k) \qquad (1)$$

where $t_{r1}(k)$ accounts for reconfiguration time of PEs after each subimage computation has completed, $t_{comm1}(k)$ is the communication delay in the data transfer, and $t_{s1}(k)$ is the setup time needed for file I/O which is a function of k.

This scheme is obviously not very efficient, because after the first stage has completed, $\frac{3}{4}^{th}$ of the number of PEs are freed up and never get used again, till the next subimage is processed. Again, after the completion of each stage, more and more PEs are freed up. The number of processors that are freed, N_{if}, after the i^{th} stage is

$$N_{if} = \frac{3}{4^i} n^2$$

The total idle time, T_i, of all processors, during the processing of one sub-image is

$$T_i = (0)\sum_{j=1}^{p} i_j + (\frac{3}{4}n^2)\sum_{j=2}^{p} i_j + (\frac{3}{4^2}n^2)\sum_{j=3}^{p} i_j + \ldots + (\frac{3}{4^p}n^2)(0)$$

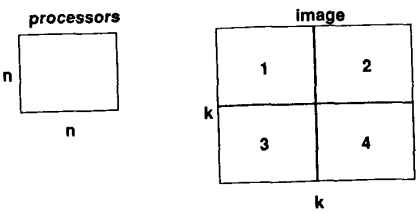

Figure 3: The image is divided into sub-images, the size of the PE array and the pixels in each subimage is assigned to the corresponding PEs. Each sub-image numbered 1,2,3 and 4 for example may be assigned to the PEs in turns.

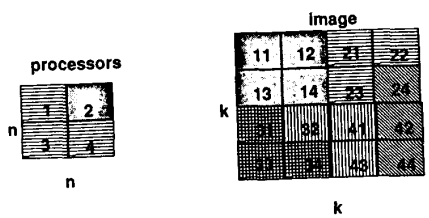

Figure 4: The array of PEs is subdivided into smaller regions. As the computation proceeds, more and more PEs are freed, which get reconfigured immediately. For example, after one iteration, only $\frac{1}{4}^{th}$ of the PEs are active, which means that three more sub-images (for example (21), (22), and (23)) can be allocated to these PEs. In the subsequent assignment, sub-images (24) (42) and (44) may be assigned to the freed PEs as indicated by the different pattern

$$\Rightarrow T_i = tn^2\Big[\sum_{j=2}^{p} i_j \sum_{i=1}^{j-1} \frac{3}{4^i}\Big]$$

The average idle time/PE, T_{ai} is then given by

$$\Rightarrow T_{ai} = t\Big[\sum_{j=2}^{p} i_j \sum_{i=1}^{j-1} \frac{3}{4^i}\Big] \qquad (2)$$

Mapping Scheme 2

Let us consider yet another scheme of mapping of tasks to processors. In this case, we try to maximize the processor utilization time, by assigning the inactive PEs to other pixels. The scheme is illustrated in figure4.

In this scheme, after each stage has completed, $\frac{3}{4}^{th}$ of the PEs are freed up. These can be reconfigured immediately. In the subsequent steps, the same number of PEs become free. Thus, we can consistently assign new tasks to the same number of PEs after the computation has completed at stage. Let us denote the total time taken for such stages to be T_{la}. In the last stages however, when there is no more tasks left, an increasing number of PEs will become idle as each level of computation completes. Let the total time in this case be T_{lb}.

In this scheme, the PE array is split into four equal sized regions, and the image is split into sub-images matching the size of these

regions. The total time to complete the computation, T_{t2}, is given by

$$T_{t2} = T_{la} + T_{lb} \tag{3}$$

Since the image size is $k \times k$ and the PE array size is $n \times n$, and the PE is split into four equal regions, we have the image split into $\frac{4k^2}{n^2}$ sub-images. In the first level of computation, four sub-images are assigned to the PEs and in the subsequent levels, three sub-images are assigned. This means, that the $\frac{4k^2}{n^2}$ sub-images would have been completely allocated to PEs in $n_s = \frac{\frac{4k^2}{n^2}-4}{3}$ steps.

Then, the time, T_{la} is given by

$$T_{la} = T_p(1 + \lceil \frac{\frac{4k^2}{n^2}-4}{3} \rceil)$$

or

$$T_{la} = T_p(1 + n_s)$$

where T_p is the equally spaced time interval after which PE reallocation takes place.

T_{lb} is the time taken for the last sub-image to iterate all the way through completion, and it given by

$$T_{lb} = t\sum_{j=1}^{p} i_j$$

$$\Rightarrow T_{t2} = T_p(1 + n_s) + t\sum_{j=1}^{p} i_j \tag{4}$$

The equation above is not exactly right. Since the number of iterations at each level are different, we have to wait till all the computations finish, before reconfiguring the freed PEs. This requires us to modify the expression for T_{la}. If the time intervals between PE reconfigurations were equal, then T_p will be a constant.

Let us define a function $\phi(i) \triangleq max(t_1, t_2, \ldots t_i)$ that computes the maximum value of the time taken for the iterations at different levels, $[l_1, l_i]$.

Also, let $t_j \triangleq 0 \; \forall \; j > p$.

We can now formulate T_{la} as

$$T_{la} = \sum_{j=1}^{n_s} \phi(j)$$

$$\Rightarrow T_{t2} = \sum_{j=1}^{n_s} \phi(j) + t\sum_{j=1}^{p} i_j \tag{5}$$

As in the previous scheme, we include a constant time $t_{r2}(k)$ for reallocation. There is also some setup time $t_{s2}(k)$ involved in file I/O. The image files have to be moved from the front end to the ACU. The time taken for the movement of data from the ACU to the DPU is accounted for by $t_{comm2}(n_s)$.

Equation(5) is modified thus,

$$T_{t2} = \sum_{j=1}^{n_s} \phi(j) + t\sum_{j=1}^{p} i_j + t_{comm2}(n_s) + t_{s2}(k) + t_{r2}(k) \tag{6}$$

Boundary conditions

In the above mapping scheme, we conveniently ignored edge conditions. When we allocated subimages whose size were equal to the PE array, we assumed that information to compute the pixels values for the next level was fully contained in the subimage. This is not the case. The pixels on the border would actually require the information from the adjacent subimages. In effect, we can only compute the next higher level for an image that is slightly smaller than the PE array size.

Say, from any level l_j to level l_{j+1}, there is a neighborhood of $n_h \times n_b (= n_t)$ at level l_j that takes part in the computation of one pixel in level l_{j+1}. Consider the allocation of one subimage of size $n \times n$ to the PE array. Since the information for some of the border pixels are missing, they cannot be computed. As a result, instead of being able to compute the values for $\frac{n^2}{n_b \times n_h}$, we can only compute the pixels values for $\left(\frac{n}{n_b} - \lceil \frac{n_b}{2} \rceil\right)\left(\frac{n}{n_h} - \lceil \frac{n_h}{2} \rceil\right)$. Incidentally, in this problem we use $n_b = n_h = 2$.

Following figure(), say we allocate the array of PEs with the subimages as shown. There will be a certain amount of overlap in the subimages. In other words, some pixels may have to be allocated to the PEs twice. If allocation proceeds in this manner,

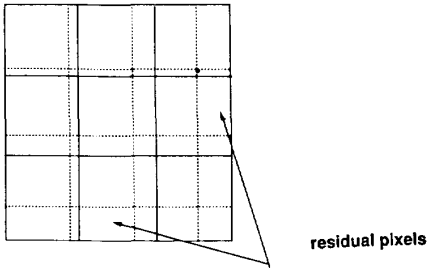

Figure 5: There is a certain overlap in the allocation scheme to take care of the edge pixel computations. The unbroken lines indicate subimages whose size are equal to the PE array. However, since the PE array is missing some neighboring information to compute the border pixels, only the areas indicated by the broken lines are computed. Each subimage allocated to the PE has some overlap as indicated by the broken lines. The *residual* pixels occur at the right end and the bottom of the image.

a rectangularly shaped region, which we shall call *residual pixels*, will be left out at the right end of the image and at the bottom of the image. These can later be allocated to the PE array. If the number of residual pixels are few, one allocation to the PE array would suffice. However, if there are too many residual pixels, a number of allocations have to be done.

Here, we attempt to compute the number of times these residual pixels have to be allocated. As before, say we have an image of size $k \times k$ and a PE array of size $n \times n$. The "thickness" of the pixels on the right hand side of the image would be $\lceil \frac{n_h}{2} \rceil \frac{k}{n}$. The height of the region is k. Similarly, the "height" of the residual pixels at the bottom of the image is $\lceil \frac{n_h}{2} \rceil \frac{k}{n}$ and the corresponding length of the region is k. The total number of residual pixels, P_r, is therefore given by

$$P_r = k\left(\lceil \frac{n_b}{2} \rceil \frac{k}{n} + \lceil \frac{n_h}{2} \rceil \frac{k}{n}\right) - \left(\lceil \frac{n_b}{2} \rceil \frac{k}{n} \times \lceil \frac{n_h}{2} \rceil \frac{k}{n}\right) \tag{7}$$

because the bottom left corner of size $\lceil \frac{n_b}{2} \rceil \frac{k}{n} \times \lceil \frac{n_h}{2} \rceil \frac{k}{n}$ has been counted twice. Considering the arrangement of the residual pixels as shown in figure 5, we would require more than R_t pixels to be allocated to the PE array, since neighborhood information is required. This bring us to the question of how many actual pixels need to be allocated to the PE array.

The row residues and the column residues do have some common pixels. However, for analytical simplicity, we assume that we allocate these common pixels, independently. Thus, the same pixel might be allocated twice in the PE array at the same time.

Therefore, the number of pixels P_t that remain to be allocated is

$$P_t = k\left[\lceil \frac{n_b}{2} \rceil \frac{k}{n} + \lceil \frac{n_b}{2} \rceil\right] + k\left[\lceil \frac{n_h}{2} \rceil \frac{k}{n} + \lceil \frac{n_h}{2} \rceil\right] \tag{8}$$

If P_t is smaller than the PE array size, one allocation would suffice. If not, the PEs have to work on the residual pixels a

number of times. The number of times, n_r is given by

$$n_r = \left\lceil \frac{k\left(\lceil \frac{n_h}{2} \rceil \frac{k}{n} + \lceil \frac{n_h}{2} \rceil \frac{k}{n}\right) + k\left(\lceil \frac{n_h}{2} \rceil \frac{k}{n} + \lceil \frac{n_h}{2} \rceil \frac{k}{n}\right)}{n^2} \right\rceil \quad (9)$$

As in our case if $n_b = n_h = 2, n = 128$ and $k = 256$, then $n_r = 1$.

Thus replacing the $\frac{k^2}{n^2}$ term by $\left(\frac{k^2}{n^2} + n_r\right)$ in equations(1) and (6), we get the actual time complexities in each of the cases to be

$$T_{t1} = \left(\frac{k^2}{n^2} + n_r\right) t \sum_{j=1}^{p} i_j + t_{r1}(k) + t_{comm1}(k) + t_{s1}(k) \quad (10)$$

and

$$T_{t2} = \sum_{j=1}^{n_t} \phi(j) + t \sum_{j=1}^{p} i_j + t_{comm2}(n_t) + t_{s2}(k) + t_{r2}(k) \quad (11)$$

where $n_s = \frac{4k^2-4}{3}$ changes to the term $n_t = \frac{4(\frac{k^2}{n^2}-n_r)-4}{3}$.

Single processor complexity

On a single processor machine, the time taken for total computation is simpler to compute. Since after each level of computation, the number of pixels reduce by four, we get the following expression for the total time taken for complete computation.

$$T_l = k^2 t \sum_{j=1}^{p} i_j \frac{1}{2^{2(j-1)}}$$

As in the previous case, we can include a factor, $t_{s3}(k)$ for the time taken for file I/O. This would change the equation to

$$T_l = k^2 t \sum_{j=1}^{p} i_j \frac{1}{2^{2(j-1)}} + t_{s3}(k) \quad (12)$$

One can use equations(1), (6) and (12) to determine theoretical speedups.

Theoretical Speedup

Using the above near exact case analysis, we can compute the speedup, S, for a particular case. As a basis of comparison, we shall refer to the speedup calculation cited by Siegel et. al. [6]. They compute the speedup, S, on a array processor for the smoothing operation to be

$$S = \frac{(k-2)^2}{k^2/n^2 + 4k/n + 4} \quad (13)$$

where k^2 and n^2 are the sizes of the image and the PE array respectively. Substituting values of $k = 256$ and $n = 128$ we get $S = 4032.25$.

Speedup for mapping scheme 1

The speedup for the mapping schemes 1 is given by

$$S = \frac{k^2 t \sum_{j=1}^{p} i_j \frac{1}{2^{2(j-1)}} + t_{s3}(k)}{\left(\frac{k^2}{n^2} + n_r\right) t \sum_{j=1}^{p} i_j + t_{comm1}(k) + t_{r1}(k) + t_{s1}(k)} \quad (14)$$

For the computations of the speedup, let us assume that the number of iterations in each stage is the same and is denoted by N. Therefore $\sum_{j=1}^{p} i_j = Np$, $p+1$ being the number of levels in the pyramid.

Also,

$$\lim_{p \to \infty} \sum_{j=1}^{p} \frac{1}{2^{2(j-1)}} = \frac{4}{3}$$

If we ignore all the communication, setup and reconfiguration times, equation(14) will reduce to

$$S = \frac{4n^2}{3p}$$

If $n = 128$ and $p = 6$ as in our case, we get a speedup of 3640. This speedup is the theoretical maximum but it is important to note that this number is given for illustrative purposes only. In actual computations, we need to take the communication and other factors into account.

For this reason, we make no attempts to give numerical values for speedups. Instead, we simple provide the equations, and if the communication and other delays are known, they can be directly plugged into the equation.

Speedup for mapping scheme 2

The speedups the mapping scheme 2 derived from equations (6) and (12) is given by

$$S = \frac{k^2 t \sum_{j=1}^{p} i_j \frac{1}{2^{2(j-1)}} + t_{s3}(k)}{\sum_{j=1}^{n_s} \phi(j) + t \sum_{j=1}^{p} i_j + t_{comm2}(n_s) + t_{s2}(k) + t_{r2}(k)} \quad (15)$$

Conclusions and Future work

In this work, we have derived a formula for the theoretical speedup obtained for the pyramidal segmentation algorithm of Sabata et. al. [2], when switching from single processor to a SIMD parallel processor. Our exact case analysis will give an accurate estimation of the speedups if the values of the different parameters are known. One could also approximately estimate these parameters. However, we have made no attempts to do so.

Currently, we are working on implementing the segmentation algorithm on the MasPar MP-1 system. This will help us to compare the performances of the system in the real world to our theoretical results.

References

[1] P. Besl and R. Jain, "Three-dimensional object recognition," *Computing Surveys*, vol. 17, pp. 75–145, 1985.

[2] B. Sabata, F. Arman, and J. K. Aggarwal, "Segmentation of 3-d range images using pyramidal data structures," *Private communication - to appear in Comput. Vision, Graphics and Image Processing*, 1992.

[3] F. Arman, B. Sabata, and J. K. Aggarwal, "Range image segmentation," in *Proc. IEEE Fifth International Conf. on Computer Vision*, (Japan), IEEE, 1991.

[4] T. Blank, "Maspar mp-1 architecture," in *Proc. Thirty Fifth IEEE Comp. Soc. Conf.*, (San Francisco, CA), IEEE, Feb-Mar 1990.

[5] V. Chaudhary and J. K. Aggarwal, "A generalized scheme for mapping parallel algorithms," *IEEE Trans. on Parallel and Distributed Systems*, vol. 4, no. 3, pp. 328–346, Mar. 1993.

[6] H. J. Siegel, J. B. Armstrong, and D. W. Watson, "Mapping computer-vision-related tasks onto reconfigurable parallel processing systems," *IEEE Computer*, vol. 25, pp. 54–63, Feb. 1992.