Load Redistribution Algorithms for Parallel Implementations of Intermediate Level Vision Tasks

Antonis A. Argyros and Stelios C. Orphanoudakis
Department of Computer Science
University of Crete, Greece
and
Institute of Computer Science
Foundation of Research and Technology – Hellas (FORTH)
Heraklion, Crete, Greece

Abstract

Parallelism can be exploited to handle the enormous computational requirements of many vision applications. However, the computational power offered by multiprocessor architectures cannot be fully harnessed to achieve the desired speedup. This is primarily due to the unbalanced distribution of computational load among the processors of a parallel architecture. Furthermore, in parallel implementations of image analysis tasks, what constitutes computational load and the load balancing requirements of specific implementations are often difficult to define in a systematic way. In this paper, we consider the load balancing requirements of parallel implementations of intermediate level vision tasks on distributed memory parallel architectures. The computational characteristics of such tasks are briefly discussed and an appropriate definition of computational load is adopted. The primary implication of this definition for load balancing is that load entities to be redistributed are allowed to have nonuniform computational cost. An existing algorithm, which assumes uniform cost loads, and two modifications of this algorithm, which handle the nonuniform cost loads encountered in parallel implementations of intermediate level vision tasks, are described. These algorithms have been implemented on the iPSC/2 hypercube and their performance has been evaluated using simulated load conditions, as well as in the context of a simple object recognition system. Results on load balancing accuracy and total execution time are presented and discussed. Algorithmic performance has also been compared with the cases of optimal load distribution and no load redistribution. This work emphasizes the importance of understanding the requirements and difficulties of load redistribution in parallel image processing applications.

1 Introduction

In this paper, parallel image processing is approached from the viewpoint of data parallelism. To achieve significant speedups by exploiting data parallelism, one must properly balance the computational load of available processors. This is particularly true in the case of intermediate level vision tasks, which are characterized by a strong dependence of load distribution on image content. At the early stages of computation, image data is uniformly distributed among all available processors. However, as the computation on image data proceeds, specific image features emerge and these may be nonuniformly distributed on the image plane. Consequently, for further computations on these features, the computational load distribution may also be nonuniform. In order to maintain a high efficiency of task execution, image features and the corresponding computational load must be properly balanced through redistribution to all available processors.

Load redistribution algorithms are characterized as *static* or *dynamic*. Static redistribution algorithms assume that the computational and communication requirements of the task being considered are known a priori and balance the load before task execution begins [1]. Dynamic redistribution strategies assume that load distribution changes with time, i.e. as task execution proceeds, and attempt to respond dynamically to the changing load conditions.

Load balancing algorithms may be further characterized as *centralized* or *distributed* [2, 3]. In centralized schemes, decisions regarding the load redistribution process are taken by just one processor, while in distributed schemes decisions are taken locally by individual processors. Centralized schemes take advantage of having a global view of the system load, but incur the overhead of concentrating this information at one processor.

In the domain of image analysis, load balancing may be viewed as the dynamic application of static redistribution algorithms. The distribution of computational load may change significantly between consecutive tasks. Thus, decisions on whether load should be redistributed must be made dynamically. However, having made the decision to redistribute the load upon completion of a particular task, a static redistribution algorithm can be used to perform the redistribution. As pointed out earlier, such algorithms require information on the current distribution and representation of computational load, as well as the computational and communication requirements of the next subtask to be executed. This information can be obtained from the output generated by previous tasks. Therefore, this paper considers static load redistribution algorithms, which take into account the load balancing requirements of parallel implementations of intermediate level vision tasks.

Many static load balancing algorithms assume the existence of computationally independent load entities that carry the same computational load. Blelloch [4] has proposed an algorithm for this instance of the load balancing problem, which is based on the fast execution of prefix operations on the Connection Machine. A similar algorithm has been implemented and evaluated by Gerogiannis [5]. The possibility of using the parallel prefix operation to distribute computational load had previously been suggested in [6]. Choudhary [7] approaches load balancing in a similar way, when dealing with a motion estimation system. The above assumption simplifies the development of load migration mechanisms that result in a uniform load distribution. However, in parallel image processing it is not always the case that load entities are computationally independent. Moreover, load entities may carry varying computational loads. In this work, we consider load entities which are computationally independent of other load entities, although they may consist of computationally dependent load units. Thus, during load redistribution, these load entities are kept intact and their associated computational cost is taken into account in the redistribution process. This approach results in reduced communication overhead incurred during the parallel execution of a complete image analysis task.

For the purpose of redistributing computationally independent load entities with different computational costs, three algorithms have been implemented and their performance has been evaluated with respect to load balancing accuracy and total time of execution. The first is an existing algorithm [8], which assumes uniform cost loads, while the other two are modifications of the first, designed to handle nonuniform cost loads.

The rest of this paper is organized as follows. In section 2, we describe in more detail the problem of redistributing nonuniform cost loads and how this arises in parallel implementations of intermediate level vision tasks. In section 3, the three redistribution algorithms used in this work are presented. These algorithms are suitable for distributed memory architectures based on the hypercube topology, the basic characteristics of which are also discussed. In section 4, preliminary experimental results are presented on the performance of these algorithms when used for the redistribution of simulated loads and loads encountered at a particular stage of a simple object recognition system. Specific conclusions derived from these results are also presented. Finally, in section 5, we draw general conclusions and discuss possible extensions of this work.

2 Problem Description

In many image analysis tasks, specific computations may be executed independently on different sets of image features. In parallel implementations of these tasks, it is desirable to keep each of these computationally independent sets of image features intact by assigning it to the same processor. This results in reduced communication overhead, which would otherwise be incurred during task execution. For example, the polygonal approximation of a list of pixels involves the coordinates of all pixels in the list. Thus, all pixels in one list form a set of computationally dependent units. However, there is no computational dependence of any list on other lists. Pixel lists may be nonuniformly distributed among available processors. Therefore, the computational load will also be nonuniformly distributed. To balance the load without incurring any communication overhead during the specific task execution, one would choose to assign all pixels of the same list to one processor.

Fig. 1 shows the assignment of computationally dependent load entities to two processors. The load entities and their dependencies form a graph. Each node i of the graph represents the execution of a certain task on a set of data and is associated with a computational cost w_i. The arcs of the graph correspond to communication needed between nodes for the task to be completed. The arc between nodes i and j is also associated with a cost c_{ij}. The nodes of this graph are assigned to available processors of a parallel architecture. The two processors p_1 and p_2 are represented as dotted oval shapes. It is assumed that the total communication cost for a specific graph assignment is determined by the costs associated with arcs between nodes residing in different processors. Such arcs are shown as dark lines in Fig. 1. In gen-

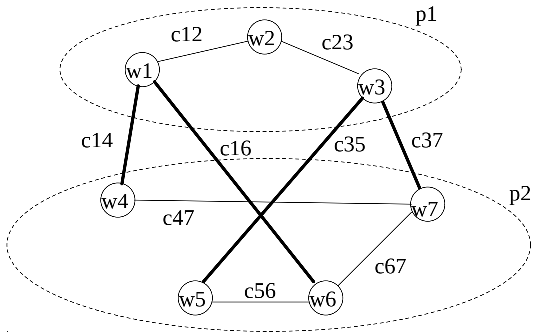

Figure 1: A possible assignment of computationally dependent load entities to two processors.

eral, a static load redistribution strategy attempts to assign nodes to processors in such a way that total execution time is minimized.

Rather than dealing with the general problem of load redistribution, this paper considers the special case of a graph consisting of many connected components, each representing a computationally independent load entity. Such a graph is shown in Fig. 2. The nodes of this graph represent computationally dependent load units. In this case, we choose to balance the load by redistributing the connected components rather than the nodes of the graph. Thus, the graph of Fig. 2 may be further simplified to the form shown in Fig. 3. In this figure, each node corresponds to a connected component of the graph of Fig. 2. Since the total computational cost of each connected component may vary, the nodes of the reduced graph may also have different computational costs.

Load balancing is much more computationally intensive when load entities to be redistributed differ in their computational costs. In this paper, two new algorithms are presented which provide solutions to this problem, by achieving quickly a good approximation to a balanced load distribution. The proposed algorithms are based on an existing algorithm for redistributing load entities of the same computational cost on hypercube machines. This algorithm is described in [8] and will be referred to as TOKENR in this paper. The algorithms we propose make use of the same general communication scheme as TOKENR, but provide different mechanisms for redistributing load entities. The first algorithm, which will be called KNAPR, is based on a heuristic solution of the 0–1 knapsack problem. The second one, which will be called SORTR, is based on the best-fit heuristic.

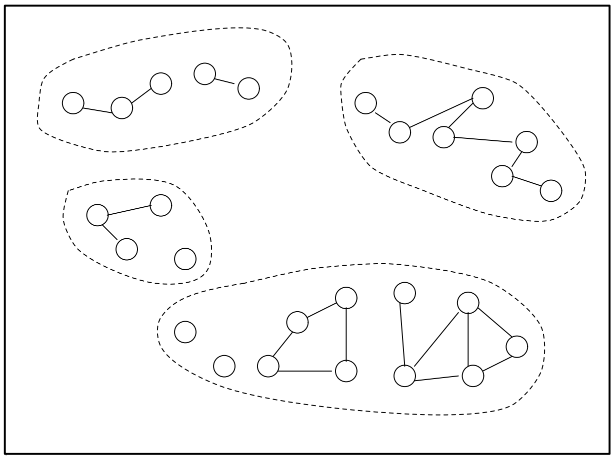

Figure 2: Graph consisting of many connected components.

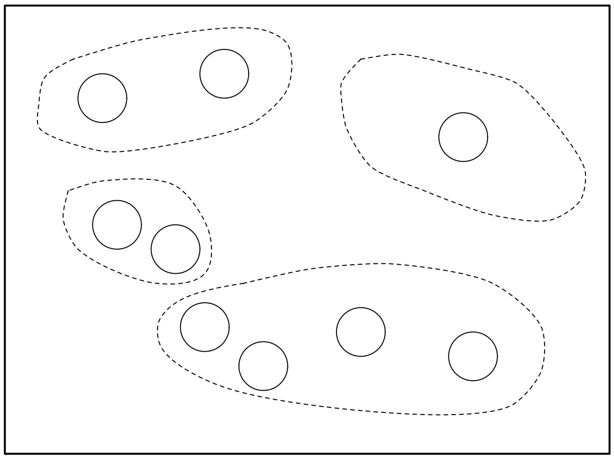

Figure 3: Reduction of connected components to nodes.

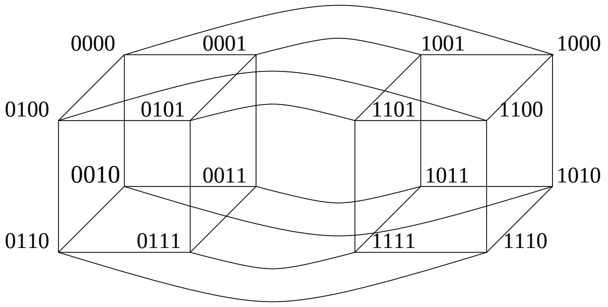

Figure 4: A 4-dimensional hypercube.

3 Redistribution Algorithms

The algorithms presented in this paper are suitable for coarse grain, distributed memory, hypercube architectures. Hypercube architectures are commonly used in parallel implementations of computer vision tasks [9, 10, 11]. A hypercube architecture is a multiprocessor with $N = 2^d$ processors, where d is the hypercube dimension. Figure 4 shows a 4-dimensional hypercube. The hypercube architecture has a recursive structure and its interconnection exhibits a number of interesting topological properties [12].

For the purpose of describing the redistribution algorithms, we use the following simplified versions of the primitives provided for interprocessor communication:

- *send*(TAG, $WhatToSend$, $Processor$)
- *recv*(TAG, $Receive$)

The *send* primitive sends the quantity $WhatToSend$ to processor $Processor$, using a message type TAG. Primitive *recv*, receives a message with type TAG, and places it in variable *Receive*. The above primitives are quite close to those provided by iPSC/2, the machine on which the proposed redistribution algorithms were implemented.

3.1 The TOKENR Algorithm

TOKENR assumes computational independence of load entities having the same computational cost. Although the algorithm has been described in the literature, we briefly discuss it here for two reasons: (1) the two proposed algorithms are modifications of TOKENR and (2) results obtained by applying this algorithm to our problem will be compared to corresponding results obtained with the other two algorithms. Specifically, it is possible to use TOKENR for load balancing nonuniform cost loads by simply ignoring differences in the computational costs of the various independent load entities.

The algorithm requires a number of stages equal to the dimension d of the hypercube. In each stage i, the d-dimensional hypercube is split into two $(d-1)$–dimensional hypercubes across the ith dimension. Because of the recursive structure of the hypercube topology, for each processor in the one subcube there exists a processor in the other, such that the binary representation of their IDs differ in just one bit. These processors are physically connected. For each of the $\frac{N}{2}$ possible pairs of processors thus defined, the processor with the most load entities is identified as overloaded relative to the other one which is characterized as underloaded. An adequate number of load entities is transferred from the overloaded processor to the underloaded one, achieving local equalization of the number of load entities. Given that all load entities are assumed to have the same computational cost, equalization on their number results in equalization of the computational cost, too.

It can be shown that, if the above procedure is repeated for each one of the hypercube dimensions, global load equalization can be reached. Upon completion of load redistribution, the difference in the number of load entities between the most and least heavily loaded processor, is at most d.

Below, the algorithm is described in an algorithmic language.

Foreach p_j, $0 \le j \le N-1$, **in parallel do**
  $MyLoad := l_j$
Foreach i, $0 \le i \le d-1$ **do**
  Foreach p_j, $0 \le j \le N-1$, **in parallel do**
    if j AND $2^i = 0$ **then**
      TarPr := p_{j+2^i}
    else
      TarPr := p_{j-2^i}
    send(**EXCHANGE**,MyLoad,TarPr)
    recv(**EXCHANGE**,HisLoad)
    if $MyLoad > HisLoad$ **then**
      $GroupsToSend = \lfloor \frac{MyLoad - HisLoad}{2} \rfloor$
      Form($F_{Transfer}$,F_j,GroupsToSend)
      send(**TRLOAD**,$F_{Transfer}$,TarPr)
    else
      recv(**TRLOAD**,$F_{Transfer}$)
      Append($F_{Transfer}$,F_j)

In the above description, d is the hypercube dimension, N the number of processors, p_j is processor with ID j, F_j the set of load entities of proces-

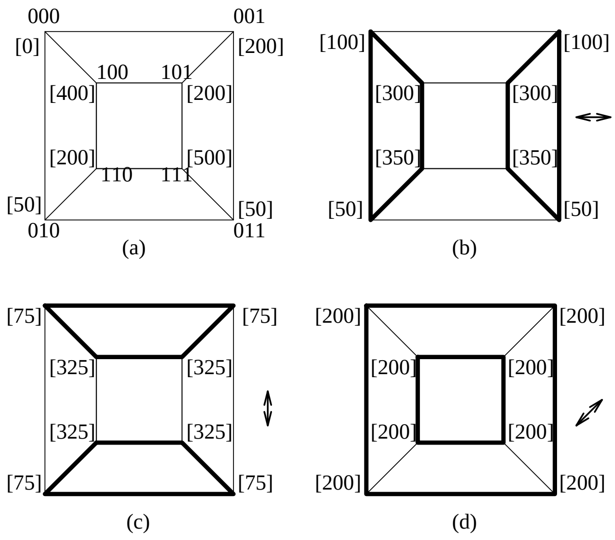

Figure 5: Execution of TOKENR on a 3-dimensional hypercube (a) shows the initial load distribution, while (b), (c) and (d) show the processor loads after each stage of the algorithm.

sor j, l_j the number of load entities of processor j, $F_{Transfer}$ the set of load entities to be transferred, and $TarPr$ the counterpart of each processor at each stage. Finally, $MyLoad$ and $HisLoad$ are also local variables in each processor, used to hold the information about the computational load of the pair of processors defined in each stage of the algorithm. Procedure Form, extracts $GroupsToSend$ load entities from F_j producing $F_{Transfer}$ and procedure Append just appends the received set of loads $F_{Transfer}$ to the local set F_j of load entities.

Fig. 5, shows schematically the execution of TOKENR on a 3-dimensional hypercube. Bold lines show the two 2–dimensional hypercubes defined at each stage. Numbers in square brackets represent the load of each processor. In Fig. 5(a), processors are also identified with the binary representation of their IDs.

TOKENR is a rather fast redistribution algorithm, since the number of required stages is logarithmic to the number of available processors.

3.2 The KNAPR Algorithm

The KNAPR algorithm improves load balancing accuracy by taking into account the differences in the computational costs of the independent load entities. KNAPR uses the same communication scheme as TOKENR. Given this communication scheme, global load equalization is achieved through a number of local redistributions. Therefore, in order to take into account differences in the costs of load entities, it is sufficient to consider redistribution of load entities between two processors.

KNAPR is based on the correspondence of the load equalization problem to the problem of 0–1 knapsack. This problem [13] may be stated as follows:

Given integers c_j, j = 1,...,n, and K, is there any subset S of $\{1, \ldots, n\}$ such that

$$\sum_{j \epsilon S} c_j = K?$$

The solution to the 0–1 knapsack problem is just a yes/no answer to the above question. However, an algorithmic procedure can be developed which yields the set S, whenever there exists a solution to the problem. The 0–1 knapsack problem belongs to the class of NP–complete problems. In the worst case, all 2^n subsets of the set of c_js should be examined as potential solutions. In an effort to reduce the average number of combinations examined, we introduce a tolerance T on the accuracy with which we approximate the constant K. Specifically, we accept as a solution to the problem, any subset S of $\{1, \ldots, n\}$ such that $K - T \leq \sum_{j \epsilon S} c_j \leq K + T$. As T increases:

- The probability that there exists a solution increases.
- The accuracy of the approximation of K decreases.
- The average number of combinations examined to yield a solution decreases.

An algorithm for the solution of the 0–1 knapsack problem is described below in an algorithmic language.

KnapSack(E, j, K, T)
1. **If** (**abs**$(K) < T$) **Then**
   Return(**TRUE**)
2. **If** $(j > n)$ **or** $(K < 0)$ **Then**
   Return(**FALSE**)
3. **If** KnapSack$(E, j+1, K - c_j, T)$ **Then**
   $Solution := Solution \cup \{c_j\}$
   Return(**TRUE**)
4. **Return**(KnapSack$(E, j+1, K, T)$)

In the above description, $Solution$ is the solution set which is initially empty. The algorithm is a recursive one. At a certain recursion depth, the jth integer c_j from set E is examined. K is the integer quantity to be approximated as a sum of elements of the set E with tolerance T. In steps 1 and 2, the recursion termination conditions are described. In step 3, if $K - c_j$ can be approximated starting from the $j+$

1 candidate, then c_j is included in the solution set. Otherwise, the algorithm tries to approximate the quantity K starting from candidate $j+1$.

The complete KNAPR algorithm proceeds in d stages, as TOKENR. At each stage, pairs of processors are identified in a way identical to that of TOKENR. In each pair, each processor is characterized as relatively overloaded or underloaded, taking into account differences in the cost of the various load entities. KnapSack is used to balance the load, in parallel for all pairs of processors. It can easily be verified that, if K is equal to one half of the difference in load of two processors, and c_js are the computational costs of the load entities residing in the relatively overloaded one, then a solution to the 0–1 knapsack problem yields a solution to the local load balancing problem. In this case, load entities which are candidates for redistribution must satisfy the condition that $c_j \leq K$. Load entities with computational costs greater than K are excluded from consideration. In implementations of this algorithm, the tolerance T is automatically calculated for each pair of processors, based on statistical characteristics of the set of computational costs associated with the load entities of the most heavily loaded processor.

KNAPR is obtained from TOKENR by modifying the procedure Form to make use of the KnapSack algorithm.

It should be noted that KNAPR involves the transfer of load entities from the overloaded processor to the underloaded one only. This one-way communication scheme is not adequate to handle load equalization in all cases. For example, consider the case in which the computational costs of the load entities for two processors i and j are given by the sets $F_i = \{50, 70\}$ and $F_j = \{40, 60\}$, with total loads $C_i = 120$ and $C_j = 100$, respectively. The load difference is 20 units and no load distribution improvement can be achieved by transferring of load entities from processor i to j. However, an optimal load distribution given by the sets ${F_i}^{opt} = \{50, 60\}$ and ${F_j}^{opt} = \{40, 70\}$ can be obtained by transferring load entities in both directions. KNAPR avoids the cost of two-way communication, thus sacrificing load balancing accuracy. SORTR, the algorithm described in the next subsection, achieves greater accuracy at the cost of a longer time of execution by allowing bidirectional transfer of load entities.

3.3 The SORTR Algorithm

SORTR is based on another heuristic for the selection of load entities that must be transferred between two processors for their load to be equalized. Pairs of processors are identified at each of the d stages of the algorithm, as in TOKENR and KNAPR. SORTR achieves equalization of load by considering all load entities of the pair of processors as candidates for redistribution.

F_A	F_B	C_A	C_B
$\emptyset$	$\emptyset$	0	0
$\{200\}$	$\emptyset$	200	0
$\{200\}$	$\{80\}$	200	80
$\{200\}$	$\{80, 60\}$	200	140
$\{200\}$	$\{80, 60, 50\}$	200	190
$\{200\}$	$\{80, 60, 50, 40\}$	200	230
$\{200, 20\}$	$\{80, 60, 50, 40\}$	220	230
$\{200, 20, 20\}$	$\{80, 60, 50, 40, 20\}$	240	230
$\{200, 20, 20\}$	$\{80, 60, 50, 40, 20, 10\}$	240	240

Table 1: An example of load assignments.

Initially, each processor in a pair sorts the computational costs of its load entities in decreasing order. One of the processors of each pair receives the list of costs of its counterpart and merges it with its own list, preserving the decreasing order of the computational costs. The best-fit heuristic is then used to redistribute the load entities between the two processors. At the start of redistribution, it is assumed that load entities do not belong to either processor. Then, every load entity is assigned to the processor that has the minimum total load up to that point of redistribution. The procedure is repeated until all load entities have been assigned. For example, consider the set $\{200, 80, 60, 50, 40, 20, 20, 10\}$ of computational costs sorted in decreasing order and gathered in one processor. Table 1 shows the assignment of these load entities to two processors. A and B are the processor IDs, F_A and F_B are the corresponding sets of computational costs (initially assumed to be empty), and C_A, C_B are the total computational costs of the two processors (initially assumed to be equal to zero).

Although in the above example the final load distribution is optimal, the algorithm does not always achieve such an optimality.

Note that in the beginning of the algorithm, what is actually transferred is the list of costs of one processor and not the load entities themselves. Therefore, for the redistribution to be completed, the actual load entities must also be transferred. If A is the processor which does the assignment and B its counterpart, there are three cases:

1. A load entity remains where it was before redistribution (processor A or B).
2. A load entity moves from processor A to processor B.
3. A load entity moves from processor B to processor A.

Case 1 is of no special interest, since it does not involve any load migration. In case 2, the load entity should simply be transferred to processor B. Finally, in case 3, processor A should first inform processor B which load entity it must receive and then the transfer is made.

4 Experimental Results

In this section we present preliminary experimental results obtained with the three load redistribution algorithms for the cases: (1) of simulated load conditions and (2) of load conditions encountered at a particular stage of a simple object recognition system. The purpose of simulations is to study algorithmic performance for a variety of load distributions. The object recognition system serves as a concrete example of the possible use of the load redistribution algorithms.

4.1 Simulated Load Conditions

The algorithms presented in the previous chapter have been implemented on the iPSC/2. The goal of such implementations is the study of algorithmic performance, both in terms of execution time and load balancing accuracy, under different load conditions.

A number of experiments were carried out under simulation of various load conditions. Each experiment is characterized by two ranges of values. The range $[l^{low}, l^{high}]$ determines how many load entities may exist in each processor. The range $[s^{low}, s^{high}]$ determines the computational cost of each load entity. In each experiment, both the number of load entities in each processor and the cost of each one are randomly selected from the above ranges. To preserve fairness in the comparison of the algorithms, once the parameter values are selected for a certain experiment, all three algorithms are tested based on the same values.

In the experiments carried out, parameters l^{low}, l^{high}, s^{low} and s^{high} were assigned values from the set $\{0, 100, 200, 300, 400, 500\}$. For each valid combination of parameter values, all three algorithms were tested. For a combination of parameter values to be valid, the following conditions must be satisfied:

- $l^{low} \leq l^{high}$.
- $s^{low} \leq s^{high}$.
- l^{low} and l^{high} are not both equal to zero.
- s^{low} and s^{high} are not both equal to zero.

The number of processors was kept constant and equal to 4 for all experiments.

As expected, the experimental results obtained with the TOKENR, indicate that, whenever $s^{low} = s^{high}$, the load redistribution is very accurate. However, the load balancing accuracy is reduced with increasing differences in the computational costs of redistributed load entities.

	TOKENR	KNAPR	SORTR
Max. diff.	7318	1126	1000
Min. diff.	0	0	0
Aver. diff.	1533	207	115

Table 2: Differences between the computational load of the most and least heavily loaded processors, after redistribution by TOKENR, KNAPR and SORTR.

The execution time of TOKENR is spent primarily on load transfers between processors. Consequently, there is a strong dependence of execution time on the initial distribution of load entities among the available processors. This is confirmed by the fact that, in the experiment with the longest execution time, l^{low} was equal to 0 and l^{high} was equal to 500, which results in maximum variation in the number of load entities per processor.

The load balancing accuracy of KNAPR algorithm depends strongly on the variation of the computational costs of redistributed load entities. Whenever $s^{low} = s^{high}$, the algorithm offers no advantage in comparison with TOKENR. As the differences in the computational costs of load entities increase, KNAPR becomes significantly more accurate than TOKENR. An important characteristic of KNAPR is that tolerance T may be selected in such a way as to give priority to accuracy or speed of execution.

The load balancing accuracy of SORTR depends on the actual values of the computational costs of the experiments. Specifically, in the great majority of experiments, the difference between the most and least heavily loaded processors after redistribution was very close to the value of s^{low}.

The execution time of SORTR depends on the total number of load entities. Most of the algorithm functions (sorting, merging, load assignment) depend on the total number of load entities in a pair of processors. SORTR redistributes load entities starting from zero. Consequently, the execution time is not influenced by the degree of load imbalance as in the case for the other two algorithms, but by the total system load. SORTR had the worst execution time for the parameter values $s^{low} = 400$, $s^{high} = 500$ and $l^{low} = l^{high} = 500$, which is in agreement with the above expectation, as these parameter values represent the experiment with the maximum system load.

Table 2 shows the maximum, minimum and average difference between the computational load of the most and least heavily loaded processors for all experiments.

	TOKENR	KNAPR	SORTR
Max. e	0.117	0.024	0.010
Min. e	0.0	0.0	0.0
Aver. e	0.019	0.002	0.001

Table 3: Error e for the three redistribution algorithms.

	TOKENR	KNAPR	SORTR
Max. time	1379	1994	5640
Min. time	3	4	107
Aver. time	207	232	1756

Table 4: Execution times for the three redistribution algorithms in msecs.

The second row of the table shows that for each algorithm, there was an experiment for which results were optimal. The third row of the table shows that, on the average, KNAPR and SORTR yield a more balanced load distribution than TOKENR.

Let C_{max} be the load of the most loaded processor after redistribution and C_{avg} the average processor load. We define the mean error of an experiment to be equal to:

$$e = \frac{C_{max} - C_{avg}}{C_{avg}}$$

Table 3 shows the maximum, minimum and average value of e for the experiments carried out. The corresponding maximum, minimum and average execution times (in milliseconds), are shown in Table 4.

A comparison of the three algorithms, based on these experimental results, leads to the following conclusions. On the average, TOKENR has the worst accuracy and the lowest execution time, while SORTR has the best accuracy and the highest execution time. On the other hand, KNAPR has an execution time which is comparable to that of TOKENR and an accuracy which is comparable to that of SORTR. Thus, KNAPR may be considered to give better overall performance in terms of execution time and load balancing accuracy.

An important difference between TOKENR and KNAPR on the one hand and SORTR on the other, is that the first two algorithms make use of the initial load distribution, while SORTR does not take it into account. This may result in long execution times for SORTR in return for only a small improvement in load distribution. The experiment with $[s^{low}, s^{high}] = [400, 500]$ and $[l^{low}, l^{high}] = [500, 500]$ is very representative of this case. For these parameter values, a possible initial load of the 4 processors is $\{224634, 224781, 224847, 225105\}$. In this case, TOKENR does not alter the load distribution since each processor has the same number of load entities. However, it does incur a computational overhead of 4 milliseconds. KNAPR, yielded the improved load distribution $\{224634, 224781, 224431, 225521\}$ in 14 msecs. Finally, SORTR yielded the optimal distribution $\{224841, 224842, 224842, 224842\}$ in 5640 msecs.

4.2 Load Conditions Encountered in an Object Recognition System

In addition to the the experimental simulation results, TOKENR and KNAPR were also tested on a simple object recognition system, which recognizes screws and nuts in images and identifies their position. The input to the system is a 512x512, 8–bit image. Images are taken under good lighting conditions so that a simple thresholding may distinguish objects and image background. No overlapping of objects is allowed. Object recognition involves the following tasks:

- **Thresholding** used to obtain a binary image.
- **Connected component labeling** used to identify the different objects.
- **Object dilation** used to fill small regions contained within the objects that are due to noise.
- **Center of mass computation** used to define the position of an object.
- **Skeletonization** resulting in object skeletons that are one pixel wide [14].
- **Polygonal approximation of object skeletons** with line segments.
- **Cycle detection** performed on the polygonal approximations of the previous step. These cycles, that are present in nuts alone, are the basis for distinguishing between the two kinds of objects.

This is neither the only nor the best recognition method. However, this work emphasizes the load balancing requirements of an object recognition system and not the recognition method itself.

Load redistribution algorithms may be applied at different stages of the process of object recognition. For example, after the labeling of connected components, each object consists of a set of computationally dependent pixels and should not be split between different processors. The computational cost of each object may be defined as the number of pixels it contains. Thus, the goal of redistribution should be to distribute the total area of all objects as uniformly as possible among the available processors. Load balancing may also be required before the task of polygonal approximation. At this stage, lists of pixels belonging to a certain object constitute a load entity. Load redistribution should equalize the total length of pixel lists assigned to different processors, while keeping lists belonging to the same object on the same processor. Finally, load redistribution may be applied before the task of cycle detection. In this case, line segments belonging to the same object should also not be separated. Given this constraint, load redistribution is needed in order to balance the number of line segments assigned to different processors. The results presented below were obtained with load redistribution applied only after the labeling of connected components.

Figures 6 and 7 show two sample images. Figure 6 shows a seemingly balanced image. In Fig. 7, objects are not uniformly distributed over the image plane. Figures 8 and 9 show the load of the most loaded processor as a function of the hypercube dimension for the cases of no load redistribution, TOKENR redistribution, KNAPR redistribution and optimal load distribution, for images 1 and 2 respectively. The optimal load distribution corresponds to a hypothetical situation, in which load is perfectly balanced and yields linear speedups. It is not implied that this hypothetical distribution could be achieved with the load entities encountered in the input images. However, points on the optimal load distribution curve represent the average system load for each cube dimension. Therefore, for each hypercube dimension and for a specific load redistribution algorithm, the error e defined earlier can also be computed from the values plotted in these figures. Figures 10 and 11 show the total execution time as a function of the hypercube dimension for the same cases and images. Finally, Figs. 12 and 13 show the efficiency of the parallel implementation of the object recognition system as a function of the hypercube dimension for images 1 and 2 respectively.

Based on these results, we may derive the following conclusions:

- The load curves of Figs. 8 and 9 display the same behaviour as the time curves of Figs. 10 and 11. This means that the particular definition of computational load is appropriate for the task under consideration.
- In all cases, load redistribution yields better results. Furthermore, KNAPR gives a better performance than TOKENR. This improvement in performance is less striking than that observed in Table 3 due to the fact that the size of all objects contained in the input images is approximately the same.
- Whenever the number of image objects is comparable to the number of available processors, load redistribution is not quite as effective. This is expected, since load balancing is achieved through redistribution of entire objects of varying size.

5 Conclusions

In this paper, three load redistribution algorithms have been described and their performance has been evaluated. These algorithms take into account specific requirements of parallel implementations of intermediate level vision tasks on coarse grain, distributed memory architectures. TOKENR redistributes the load based only on the number of computationally independent load entities. KNAPR and SORTR take also into account differences in the computational cost of these load entities. It is shown that KNAPR gives the best performance when both load balancing accuracy and execution time are taken into account.

In addition to the above considerations, the choice of a load redistribution algorithm for a specific application would also be influenced by the execution time associated with operations on a load unit. If this execution time is high, then SORTR would probably be a good candidate since it is more accurate. If, on the other hand, this execution time is relatively low, then TOKENR would be a better choice since it achieves less accurate load balancing, but in a much shorter time. When the execution time associated with operations on a load unit is not the deciding factor, KNAPR may be a better choice based on its overall performance with respect to load balancing accuracy and execution time.

A more general issue which has not been dealt with is that of estimating in advance the real impact of load balancing on the overall performance of parallel implementations of integrated vision tasks. This issue can be addressed in the context of a general methodology for efficient parallel implementations of such tasks. The development of such a methodology requires careful consideration of the computational and communication requirements of different algorithms, specific architectural features, the dependence of load distribution on image content, and the characteristics of load redistribution algorithms. This is the direction of our current and future work.

References

[1] Eager DL, Lazowska ED, and Zahorian J. Adaptive Load Sharing in Homogeneous Distributed Systems. *IEEE Transactions on Software Engineering*, 12(5):662–675, May 1986.

[2] Nicol D and Saltz J. Dynamic Remapping of Parallel Computations with Varying Resource Demands. ICASE tech. report 86-45, July 1986.

[3] Baumgartner KM and Wah BW. Load Balancing Protocols on a Local Computer System with a Multiaccess Network. In *International Confer-*

ence on Parallel Processing, St. Clarles, Illinois, pages 851–858, August 1987.

[4] Blelloch GE. Scans as Primitive Parallel Operations. *IEEE Transactions on Computers*, 38(11):1526–1538, November 1989.

[5] Gerogiannis DC. *Efficient Implementation of Intermediate Level Image Analysis Tasks om Parallel Machines*. PhD thesis, Yale University, 1992.

[6] Kruskal CP, Rudolph L, and Snir M. The Power of Parallel Prefix. In *International Conference on Parallel Processing*, pages 180–184, August 1985.

[7] Choudhary AN, Leung MK, Huang TS, and Patel JH. Parallel Implementation and Evaluation of Motion Estimation System Algorithms on a Distributed Memory Multiprocessor Using Knowledge Based Mappings. In *10th Intern. Conf. on Pattern Recognition, Atlantic City, USA*, pages 337–342, June 1990.

[8] Plaxton CG. Load Balancing on the Hypercube and Shuffle-Exchange. Technical report, Dept. Comp. Science, Stanford University, 1990.

[9] Ranka S and Sahni S. Image Template Matching on MIMD Hypercube Multicomputers.

[10] Fang Z, Li X, and Ni LM. Parallel Algorithms for Image Template Matching on Hypercube SIMD Computers. In *IEEE Computer Society Workshop on Computer Architecture for Pattern Analysis and Image Database Management*, pages 33–40, November 1985.

[11] Hummel R and Rojer A. Connected Component Labeling in Image Processing with MIMD Architectures. Technical Report 173, July 1985.

[12] Saad Y and Schultz MH. Topological Properties of Hypercubes. *IEEE Transactions on Computers*, 37(7):867–872, July 1988.

[13] Papadimitriou CH and Steiglitz K. *Combinatorial Optimization: Algorithms and Complexity*. Prentice Hall Co., 1982.

[14] Pavlidis T. *Algorithms for Graphics and Image Processing*. Prentice Hall Co., 1982.

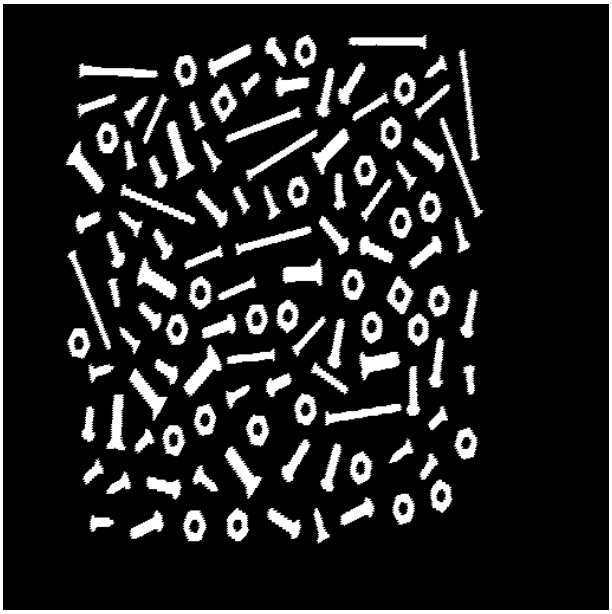

Figure 6: Input image 1 with a relatively uniform distribution of load entities.

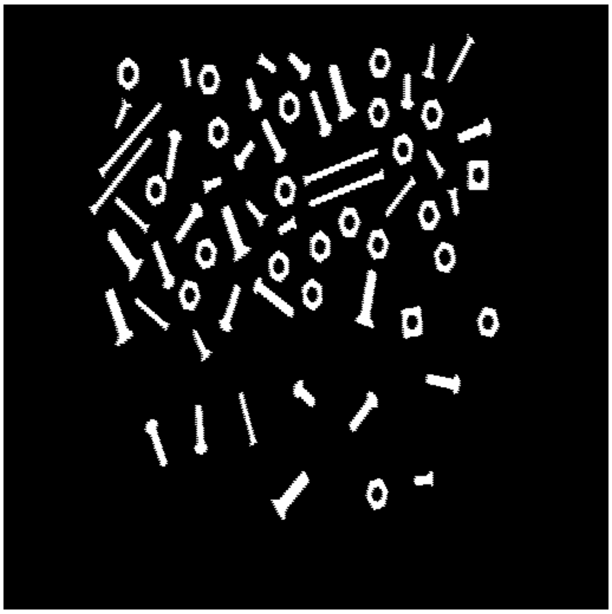

Figure 7: Input image 2 with a nonuniform distribution of load entities.

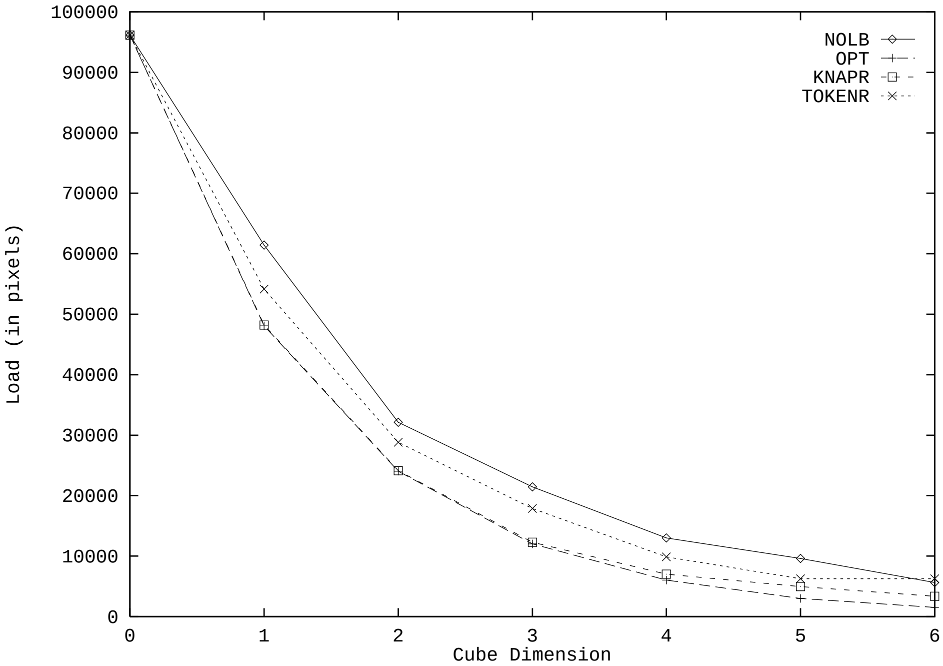

Figure 8: Load of the most heavily loaded processor as a function of the hypercube dimension for image 1.

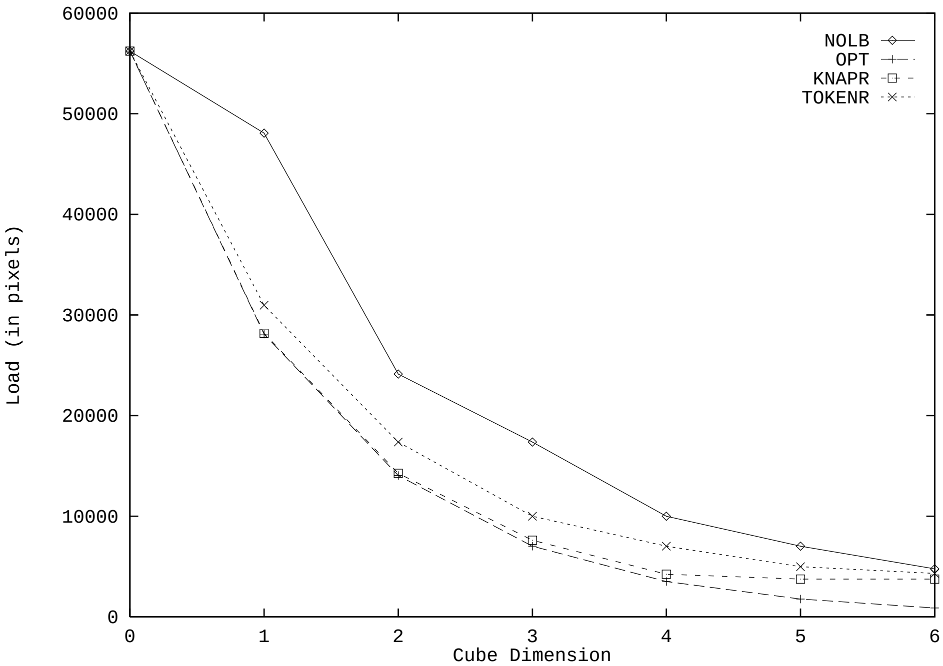

Figure 9: Load of the most heavily loaded processor as a function of the hypercube dimension for image 2.

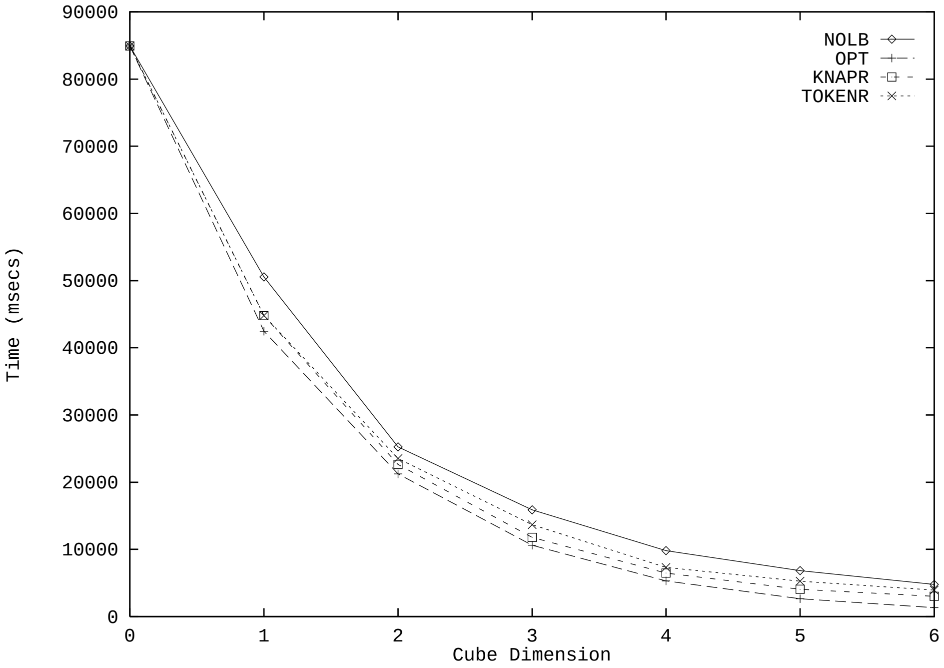

Figure 10: Total execution time for the recognition of objects contained in image 1.

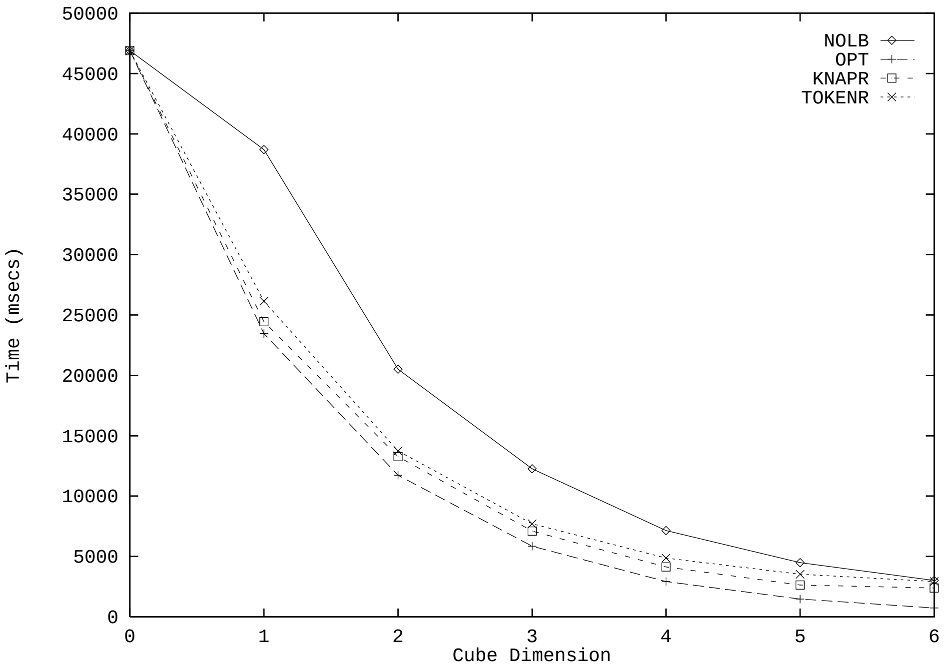

Figure 11: Total execution time for the recognition of objects contained in image 2.

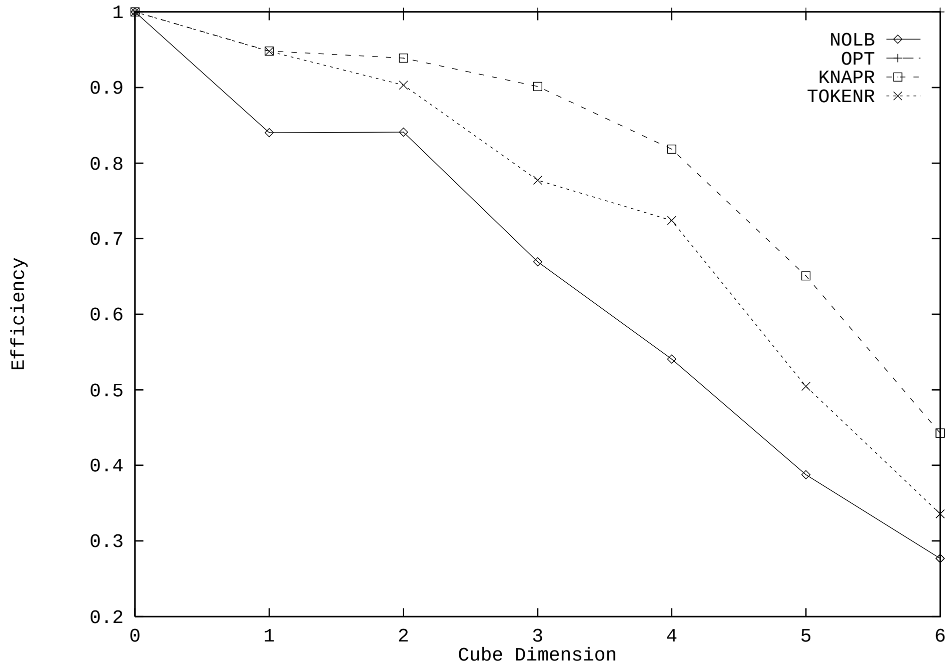

Figure 12: Efficiency of the parallel execution of the object recognition task in the case of image 1.

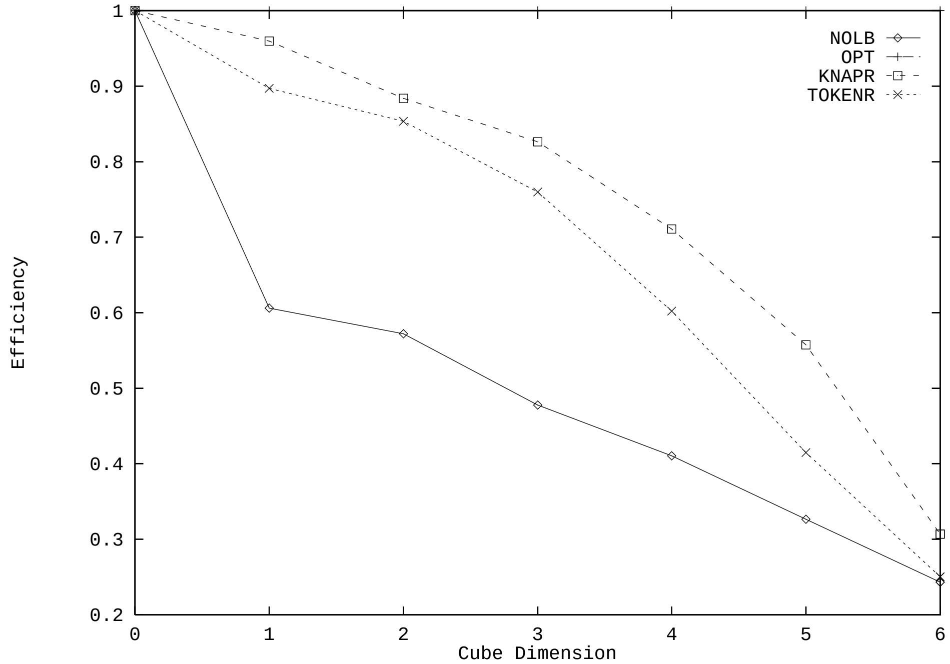

Figure 13: Efficiency of the parallel execution of the object recognition task in the case of image 2.